COMS10015 hand-out: exam-style revision questions

Part I: Mathematical preliminaries

▷ **S1.** The question essentially just demands application of

$$\hat{x} \mapsto x = \sum_{i=0}^{i<n} \hat{x}_i \cdot b^i,$$

which defines a mapping between the representation (on the LHS) and value (on the RHS) of x. In this case, setting $b = 3$ allows computation of a (decimal) value for *each* representation (i.e., literal); we need to select the one whose value turns out to be $123_{(10)}$. As such, it should be clear that

$$\begin{array}{rcl}
\hat{x} & = & 11120 & = & \langle 0,2,1,1,1 \rangle_{(3)} \\
 & & & \mapsto & \sum_{i=0}^{i<n} \hat{x}_i \cdot 3^i \\
 & & & \mapsto & 0 \cdot 3^0 + 2 \cdot 3^1 + 1 \cdot 3^2 + 1 \cdot 3^3 + 1 \cdot 3^4 \\
 & & & \mapsto & 0 \cdot 1 + 2 \cdot 3 + 1 \cdot 9 + 1 \cdot 27 + 1 \cdot 81 \\
 & & & \mapsto & 123_{(10)}
\end{array}$$

st. 11120 is the correct answer.

▷ **S2.** It should be clear that

$$\begin{array}{rcl}
\hat{x} & = & \langle \hat{x}_0, \hat{x}_1, \ldots, \hat{x}_{n-1} \rangle \\
 & = & \langle 1,1,0,0,1,1,0,0 \rangle \\
 & \mapsto & -\hat{x}_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} \hat{x}_i \cdot 2^i \\
 & \mapsto & 2^0 + 2^1 + 2^4 + 2^5 \\
 & \mapsto & 1 + 2 + 16 + 32 \\
 & \mapsto & 51_{(10)}
\end{array}$$

$$\begin{array}{rcl}
\hat{y} & = & \langle \hat{y}_0, \hat{y}_1, \ldots, \hat{y}_{n-1} \rangle \\
 & = & \langle 1,1,0,0,1,1,0,0 \rangle \\
 & \mapsto & (-1)^{\hat{y}_{n-1}} \cdot \sum_{i=0}^{n-2} \hat{y}_i \cdot 2^i \\
 & \mapsto & 1 \cdot (2^0 + 2^1 + 2^4 + 2^5) \\
 & \mapsto & 1 + 2 + 16 + 32 \\
 & \mapsto & 51_{(10)}
\end{array}$$

i.e., both x and y are represented by the binary literal 00110011, which yields the decimal value $51_{(10)}$ in both cases. However, if we set the MSB of $\hat{x}$ and $\hat{y}$ to 1, then we get

$$\begin{array}{rcl}
\hat{x}' & = & \langle \hat{x}'_0, \hat{x}'_1, \ldots, \hat{x}'_{n-1} \rangle \\
 & = & \langle 1,1,0,0,1,1,0,1 \rangle \\
 & \mapsto & -\hat{x}'_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} \hat{x}'_i \cdot 2^i \\
 & \mapsto & 2^0 + 2^1 + 2^4 + 2^5 - 2^7 \\
 & \mapsto & 1 + 2 + 16 + 32 - 128 \\
 & \mapsto & -77_{(10)}
\end{array}$$

$$\begin{array}{rcl}
\hat{y}' & = & \langle \hat{y}'_0, \hat{y}'_1, \ldots, \hat{y}'_{n-1} \rangle \\
 & = & \langle 1,1,0,0,1,1,0,1 \rangle \\
 & \mapsto & (-1)^{\hat{y}'_{n-1}} \cdot \sum_{i=0}^{n-2} \hat{y}'_i \cdot 2^i \\
 & \mapsto & -1 \cdot (2^0 + 2^1 + 2^4 + 2^5) \\
 & \mapsto & -1 - 2 - 16 - 32 \\
 & \mapsto & -51_{(10)}
\end{array}$$

so -77 and -51 is the correct answer.

▷ **S3.** In 16 bits, the largest possible positive value we can represent using two's-complement is

$$2^{n-1} - 1 = 2^{16-1} - 1 = 32767$$

and the largest possible negative value is

$$-2^{n-1} = -2^{16-1} = -32768$$

Using this, we can deduce the following:

a The largest possible positive product is given by

$$x \cdot y = -32768 \cdot -32768 = 1073741824$$

which is somewhat counter-intuitive given both operands are negative, but stems from the fact that in two's-complement more negative values can be represented than positive.

b The largest possible negative product is given by

$$x \cdot y = -32768 \cdot 32767 = -1073709056$$

or

$$x \cdot y = 32767 \cdot -32768 = -1073709056.$$

▷ **S4.** First, note that

$$\begin{array}{lclcl} \texttt{x} & = & 256_{(10)} & = & 0000000100000000_{(2)} \\ \texttt{y} & = & 4852_{(10)} & = & 0001001011110100_{(2)} \end{array}$$

Casting from `short` to `char` basically truncates the value from 16 to 8 bits (i.e., leaves the 8 LSBs only), meaning

$$\begin{array}{lclcr} \texttt{(char)(x)} & = & 00000000_{(2)} & = & 0_{(10)} \\ \texttt{(char)(y)} & = & 11110100_{(2)} & = & -12_{(10)} \end{array}$$

st. 0 and −12 is the correct answer.

▷ **S5.** Given

$$\begin{array}{lcl} \texttt{x} & = & 9_{(10)} \\ & = & 00001001_{(2)} \\ \texttt{0x97} & = & 97_{(16)} \\ & = & 10010111_{(2)} \end{array}$$

the expression `( ~x << 4 )` 0x97| evaluates to give

$$\begin{array}{lcl} \texttt{(~x)} & = & 11110110_{(2)} \\ \texttt{(~x << 4)} & = & 01100000_{(2)} \\ \texttt{(~x << 4)}\ \text{0x97}| & = & 11110111_{(2)} \\ & = & -9_{(10)} \end{array}$$

st. `r` = −9 is the correct answer.

▷ **S6.** For a signed, n-bit integers represented using two's-complement we know that

$$-2^{n-1} \leq \texttt{x} \leq 2^{n-1} - 1$$

so for the 8-bit examples in this case

$$-128 \leq \texttt{x} \leq 127.$$

Of the 256 possible values of `x`, 128 are negative (i.e., < 0) and 128 are positive (i.e., ≥ 0). Clearly *all* positive values are fixed points: for these, the `if` statement causes the assignment `r = x` to be executed. So the question is whether or not *any* of the negative values are fixed points?

At first glance, this would seem impossible so none is an understandable response. However, recall that negation in two's-complement is achieved via

$$\texttt{r} = -\texttt{x} = \neg\texttt{x} + 1.$$

Now consider what happens if we negate `x` = −128:

$$\begin{array}{lcll} -\texttt{x} & = & \neg\ \texttt{x} & + 1 \\ & = & \neg\ 10000000_{(2)} & + 1 \\ & = & \ 01111111_{(2)} & + 1 \\ & = & \ 10000000_{(2)} & \\ & = & \ \texttt{x} & \end{array}$$

This means `x` = −128 is a fixed point, i.e., that `abs` fails to work correctly for this value (since we cannot represent 128 in 8 bits using two's-complement). As such, 129 is the correct answer: the 128 positive values, plus the 1 negative value from above.

▷ **S7.** First, note that

$$\begin{array}{lcl} \texttt{x} & = & -10_{(10)} \\ & \mapsto & -2^7 + 2^6 + 2^5 + 2^4 + 2^2 + 2^1 \\ & \mapsto & 11110110 \end{array}$$

Next, since `x` is signed (implying an arithmetic vs. logical right-shift), we can evaluate the expression as follows

$$\begin{array}{lcl} \texttt{\~{}((x >> 2) \^{} 0xF4)} & \mapsto & \neg((11110110 \gg 2) \oplus 11110100) \\ & \mapsto & \neg(11111101 \oplus 11110100) \\ & \mapsto & \neg(00001001) \\ & \mapsto & 11110110 \\ & \mapsto & -2^7 + 2^6 + 2^5 + 2^4 + 2^2 + 2^1 \\ & \mapsto & -10_{(10)} \end{array}$$

▷ **S8.** We know that

$$0 \leq \texttt{x}, \texttt{y} < 2^8 = 256$$

due to their type and representation, so there are $256 \cdot 256 = 65536$ possible assignments to them variables, i.e., pairs

$$(\texttt{x}, \texttt{y})$$

to consider. Only the case where their sum `x + y` is zero will yield a Hamming weight of zero: the repersentation of a *non*-zero sum will have at least one bit in it equal to one, and hence a Hamming weight of greater than zero.

The obvious initial answer would be that the assignment `x` = 0 and `y` = 0 is the only case where `x + y` = 0 and hence HW(`x + y`) = 0. However, *others* exist due to the effect of overflow: `x` = 255 and `y` = 1 *should* yield `x + y` = 256, for example, but, due to overflow (i.e., the fact we we cannot represent 256 as an unsigned, 8-bit integer), *actually* yields `x + y` = 0 and hence HW(`x + y`) = 0. Applying this principle more generally, the correct answer is that 256 pairs will yield `x + y` = 0 and hence HW(`x + y`) = 0: put simply, every possible `x` has exactly one `y` that will yield the sum `x + y` = 0.

▷ **S9.** The central idea in a positional number system is

$$\hat{x} = \langle \hat{x}_0, \hat{x}_1, \ldots, \hat{x}_{n-1} \rangle \mapsto x = \sum_{i=0}^{n-1} \hat{x}_i \cdot b^i,$$

i.e., each i-th digit in the (left-hand side) literal is weighted by b^i and accumulated to yield the (right-hand side) represented value. In this case the literal has $n = 2$ digits, so, given $x_0 = 0$ and $x_1 = 1$, we can see that

$$\hat{x} = \langle \hat{x}_0, \hat{x}_1 \rangle \mapsto x = \hat{x}_0 \cdot b_0 + \hat{x}_1 \cdot b^1 = 0 \cdot 1 + 1 \cdot b = 0 + b = b.$$

This fact holds for any b, and of course if we *knew* b one of the other options *might* be correct (e.g., if $b = 10$ then $x = 10$ would also be correct).

▷ **S10.** First, note that in two's-complement representation

$$\begin{array}{rcl} 0_{(10)} & \mapsto & 00000000 \\ -1_{(10)} & \mapsto & 11111111 \end{array}$$

yield n-bit "all 0" and "all 1" sequences for any n. Now consider an arbitrary x, and each option stemming from it:

$$\begin{array}{rclcr} x & = & 01101010 & \mapsto & 106_{(10)} \\ \neg x & = & 10010101 & \mapsto & -107_{(10)} \\ \\ x \wedge \neg x & = & 00000000 & \mapsto & 0_{(10)} \\ x \vee \neg x & = & 11111111 & \mapsto & -1_{(10)} \\ x \oplus \neg x & = & 11111111 & \mapsto & -1_{(10)} \\ x + \neg x & = & 11111111 & \mapsto & -1_{(10)} \\ x - \neg x & = & 11010101 & \mapsto & -43_{(10)} \end{array}$$

Note that if the i-th bit of x is 0 then that of $\neg x$ will be 1; if the i-th bit of x is 1 then that of $\neg x$ will be 0. Based on this, the AND case will always yield the "all 0" sequence, i.e., 0, because the per-bit computation will be $0 \wedge 1$ or $0 \wedge 1$ (both yielding 0). Likewise, the OR case will always yield the "all 1" sequence, i.e., −1, because the per-bit computation will be $0 \vee 1$ or $0 \vee 1$ (both yielding 1); the same is true for both the XOR case *and* the addition case, with the latter stemming from the absence of carries. In fact, all these cases will apply for any n and x based on the same reasoning. So the subtraction case is incorrect: even the result is *slightly* confusing in that this example overflows (the actual result $213_{(10)}$ cannot be represented in 8 bits, at least when using two's-complement).

▷ **S11.** a **true**. We know that 1) $\mathrm{HW}(x)$ yields the number of times $x_i = 1$, and 2) $\mathrm{HD}(x, y)$ yields the number of times $x_i \neq y_i$. Therefore, given $r = x \oplus y$ we know that $\mathrm{HW}(r)$ yields the number of times $r_i = 1$: based on the definition of XOR, this is the number of times $x_i = 0$ and $y_i = 1$ or $x_i = 1$ and $y_i = 0$, i.e., the number of times $x_i \neq y_i$. Hence, $\mathrm{HW}(r) = \mathrm{HW}(x \oplus y) = \mathrm{HD}(x, y)$.

b **true**. When using two's-complement representation, we have that $-x \equiv \neg x + 1$. For example, given an 8-bit x equal to 1 we find that

$$\begin{array}{lcl} x & = & 000000001_{(16)} \\ \neg x & = & 111111110_{(16)} \\ \neg x\ +1 & = & 111111111_{(16)} \end{array}$$

i.e., the representation of -1 will have all bits set to 1. This holds for any n, because two's-complement representation means

$$\begin{array}{rcl} \hat{x} & = & \langle \hat{x}_0, \hat{x}_1, \ldots, \hat{x}_{n-1} \rangle \\ & \mapsto & x \\ & = & \hat{x}_{n-1} \cdot -2^{n-1} + \sum_{i=0}^{n-2} \hat{x}_i \cdot 2^i \end{array}$$

i.e., the MSB $\hat{x}_{n-1}$ has a large negative weight, whereas all other $\hat{x}_i$ for $0 \leq i < n-1$ have a small(er) positive weight. So, if $\hat{x}_i = 1$ for $0 \leq i < n$ we find that

$$\begin{array}{rcl} \hat{x} & \mapsto & 1 \cdot -2^{n-1} + \sum_{i=0}^{n-2} 1 \cdot 2^i \\ & \mapsto & -2^{n-1} + (2^{n-1} - 1) \\ & \mapsto & -1 \end{array}$$

▷ **S12.** a i $|A| = 3$.
ii $A \cup B = \{1, 2, 3, 4, 5\}$.
iii $A \cap B = \{3\}$.
iv $A - B = \{1, 2\}$.
v $\overline{A} = \{4, 5, 6, 7, 8\}$.
vi $\{x \mid 2 \cdot x \in \mathcal{U}\} = \{1, 2, 3, 4\}$.

b i +0 in sign-magnitude is 00000000, in two's-complement is 00000000.
ii −0 in sign-magnitude is 10000000, in two's-complement is 00000000.
iii +72 in sign-magnitude is 01001000, in two's-complement is 01001000.
iv −34 in sign-magnitude is 10100010, in two's-complement is 11011110.
v −8 in sign-magnitude is 10001000, in two's-complement is 11111000.
vi This is a trick question: one cannot represent 240 in 8-bit sign-magnitude or two's-complement; the incorrect guess of 11111000 in two's-complement for example is actually −8.

▷ **S13.** The population count or Hamming weight of x, denoted by $\mathcal{H}(x)$ say, is the number of bits in the binary expansion of x that equal one. Using an unsigned 32-bit integer x for example, an implementation might be written as follows:

```
int H( uint32_t x ) {
  int t = 0;

  for( int i = 0; i < 32; i++ ) {
    if( ( x >> i ) & 1 ) {
      t = t + 1;
    }
  }

  return t;
}
```

▷ **S14.** Writing

$$\begin{array}{lcl} t_0 & = & (x \wedge y) \oplus z \\ t_1 & = & (\neg x \vee y) \oplus z \\ t_2 & = & (x \vee \neg y) \oplus z \\ t_3 & = & \neg(x \vee y) \oplus z \\ t_4 & = & \neg\neg(x \vee y) \oplus z \end{array}$$

for brevity, we can write the following truth table:

x	y	z	t_0	t_1	t_2	t_3	t_4
0	0	0	0	1	1	1	0
0	0	1	1	0	0	0	1
0	1	0	0	1	0	0	1
0	1	1	1	0	1	1	0
1	0	0	0	0	1	0	1
1	0	1	1	1	0	1	0
1	1	0	1	1	1	0	1
1	1	1	0	0	0	1	0

Looking at the row where $x = 0$, $y = 0$ and $z = 1$, it is clear that $t_0 = 1$ and $t_4 = 1$, so $(x \wedge y) \oplus z$ and $\neg\neg(x \vee y) \oplus z$ are the correct answers.

▷ **S15.** You may be able to just spot which one is incorrect, but looking at each case exhaustively (via a truth table for the LHS and RHS of the supposed equivalence), we see that

x	y	z	$(x \wedge y) \wedge z$	$x \wedge (y \wedge z)$	$x \vee 1$	x	$x \vee \neg x$	1	$\neg(x \vee y)$	$\neg x \wedge \neg y$	$\neg\neg x$	x
0	0	0	0	0	1	0	1	1	1	1	0	0
0	0	1	0	0	1	0	1	1	1	1	0	0
0	1	0	0	0	1	0	1	1	0	0	0	0
0	1	1	0	0	1	0	1	1	0	0	0	0
1	0	0	0	0	1	1	1	1	0	0	1	1
1	0	1	0	0	1	1	1	1	0	0	1	1
1	1	0	0	0	1	1	1	1	0	0	1	1
1	1	1	1	1	1	1	1	1	0	0	1	1

and identify $x \vee 1 \equiv x$ as the incorrect case: it probably should be $x \vee 1 \equiv 1$, or maybe $x \wedge 1 \equiv x$.

▷ **S16.** By writing

$$\begin{array}{lcl} t_0 & = & x \vee (z \vee y) \\ t_1 & = & \neg(\neg y \wedge \neg z) \end{array}$$

we can shorten the expression to

$$t_2 = t_0 \wedge t_1.$$

Then, you can see either by enumeration, i.e.,

x	y	z	t_0	t_1	t_2	$y \vee z$
0	0	0	0	0	0	0
0	0	1	1	1	1	1
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	1	0	0	0
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

or via the derivation

$$\begin{array}{llcll} & (x \vee (z \vee y)) & \wedge & \neg(\neg y \wedge \neg z) & \\ = & (x \vee (z \vee y)) & \wedge & (y \vee z) & \text{(de Morgan)} \\ = & (x \vee (y \vee z)) & \wedge & (y \vee z) & \text{(commutativity)} \\ = & (x \vee (y \vee z)) & \wedge & ((y \vee z) \vee 0) & \text{(identity)} \\ = & ((y \vee z) \vee x) & \wedge & ((y \vee z) \vee 0) & \text{(commutativity)} \\ = & (y \vee z) \vee (x \wedge 0) & & & \text{(distribution)} \\ = & (y \vee z) \vee (0) & & & \text{(null)} \\ = & (y \vee z) & & & \text{(identity)} \end{array}$$

that the correct answer is $y \vee z$.

▷ **S17.** By writing

$$\begin{array}{rcl} t_0 & = & x \vee y \\ t_1 & = & x \wedge z \end{array}$$

we can shorten the expression to

$$t_2 = t_0 \vee t_1.$$

Then, you can see either by enumeration, i.e.,

x	y	z	t_0	t_1	t_2	$x \vee y$
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	1	0	1	1
0	1	1	1	0	1	1
1	0	0	1	0	1	1
1	0	1	1	1	1	1
1	1	0	1	0	1	1
1	1	1	1	1	1	1

or via the derivation

$$\begin{array}{cll} & (x \vee y) \vee (x \wedge z) & \\ = & (y \vee x) \vee (x \wedge z) & \text{(commutativity)} \\ = & y \vee (x \vee (x \wedge z)) & \text{(association)} \\ = & y \vee x & \text{(absorption)} \\ = & x \vee y & \text{(commutativity)} \end{array}$$

that the correct answer is $x \vee y$.

▷ **S18.** In the same way as NAND, we know NOR is functionally complete: this can be shown via

$$\begin{array}{lclcl} \neg x & & & \equiv & x \overline{\vee} x \\ x \wedge y & \equiv & \neg x \ \overline{\vee} \ \neg y & \equiv & (x \overline{\vee} x) \ \overline{\vee} \ (y \overline{\vee} y) \\ x \vee y & & & \equiv & (x \overline{\vee} y) \ \overline{\vee} \ (x \overline{\vee} y) \end{array}$$

Then, since $x \overline{\vee} y \equiv \neg(x \vee y)$, clearly $\{\vee, \neg\}$ is functionally complete: this set can be rewritten directly as $\{ \overline{\vee} \}$. We can harness these facts to show that in fact *all* other options are functionally complete.

- Given the truth table

x	y	$\oplus$
0	0	0
0	1	1
1	0	1
1	1	0

  we have $\neg x \equiv x \oplus 1$, or, put another way, we can construct $\neg$ using $\oplus$. Overall then,

  $$\{\oplus, \vee\} \rightsquigarrow \{\neg, \vee\},$$

  i.e., we have constructed the RHS from the LHS; we know the RHS is functionally complete, so the LHS is as well.
- This option is somewhat more difficult: using the same strategy as above, we now need to construct both $\neg$ *and* $\vee$.

  – Given the truth table

x	y	$\equiv$	$\not\equiv$
0	0	1	0
0	1	0	1
1	0	0	1
1	1	1	0

  it should be clear that since $x \not\equiv y \equiv x \oplus y$, we have $\neg x \equiv x \not\equiv 1$. Alternatively, given the truth table

x	y	$\Rightarrow$
0	0	1
0	1	1
1	0	0
1	1	1

  we have $x \Rightarrow 0 \equiv \neg x$.

  - Given the axiom $x \Rightarrow y \equiv \neg x \vee y$, we could write $\neg x \Rightarrow y \equiv \neg(\neg x) \vee y \equiv x \vee y$. That is, based on constructions for $\neg$ and $\Rightarrow$ (the LHS) we can construct $\vee$ (the RHS).

  Overall then,

$$\{\Rightarrow, \not\equiv\} \leadsto \{\neg, \vee\},$$

  i.e., we have constructed the RHS from the LHS; we know the RHS is functionally complete, so the LHS is as well.

- Based on the above, it should be clear that

$$\{\Rightarrow\} \leadsto \{\neg, \vee\}$$

  because we can construct both $\neg$ and $\vee$ using it alone; in the above $\not\equiv$ was potentially redundant in fact, which is also true of $\not\Rightarrow$ here. Alternatively, given the truth table

x	y	$\Rightarrow$	$\not\Rightarrow$
0	0	1	0
0	1	1	0
1	0	0	1
1	1	1	0

  we could write $1 \not\Rightarrow y \equiv \neg y$, and use $\not\Rightarrow$ as a replacement for $\not\equiv$ and so $\neg$ as required in the above. Either way, it is clearly the case that

$$\{\Rightarrow, \not\Rightarrow\} \leadsto \{\neg, \vee\}$$

  i.e., we have constructed the RHS from the LHS; we know the RHS is functionally complete, so the LHS is as well.

▷ **S19.** First, note that each input can obviously be assigned one of two values, namely 0 or 1, so there are 2^n possible assignments to n inputs. For example, if we have 1 input, say x, there are $2^1 = 2$ possible assignments because x can either be 0 or 1. In the same way, for 2 inputs, say x and y, there are $2^2 = 4$ possible assignments: we can have

$$\begin{array}{ll} x = 0 & y = 0 \\ x = 0 & y = 1 \\ x = 1 & y = 0 \\ x = 1 & y = 1 \end{array}$$

This is why a truth table for n inputs will have 2^n rows: each row details one assignment to the inputs, and the associated output.

So how many *functions* are there? *A* function with n-inputs means a truth table with 2^n rows; each row includes an output that can either be 0 or 1 (depending on exactly *which* function the truth table describes). So to count how many functions there are, we can just count how many possible assignments there are to the 2^n outputs. The correct answer is 2^{2^n}.

▷ **S20.** The derivation is as follows

$$\begin{array}{rlcll}
\text{LHS} & = (y \wedge \neg x) & \vee & (x \wedge \neg y) & \\
 & = (y \wedge \neg x) \vee 0 & \vee & (x \wedge \neg y) \vee 0 & \text{(identity)} \\
 & = (y \wedge \neg x) \vee (y \wedge \neg y) & \vee & (x \wedge \neg y) \vee (x \wedge \neg x) & \text{(inverse)} \\
 & = (y \wedge (\neg x \vee \neg y)) & \vee & (x \wedge (\neg y \vee \neg x)) & \text{(distribution)} \\
 & = ((\neg x \vee \neg y) \wedge y) & \vee & ((\neg x \vee \neg y) \wedge x) & \text{(commutativity)} \\
 & = (\neg x \vee \neg y) \wedge (x \vee y) & & & \text{(distribution)} \\
 & = (x \vee y) \wedge (\neg x \vee \neg y) & & & \text{(commutativity)} \\
 & = (x \vee y) \wedge \neg(x \wedge y) & & & \text{(de Morgan)} \\
 & = \text{RHS} & & &
\end{array}$$

suggesting that the correct option is the de Morgan axiom.

▷ **S21.** These are all (or close to) Boolean axioms, which potentially can be identified by just looking at them: for example, the first one is the association axiom. Taking a more systematic approach, the following, exhaustive truth-table

x	y	z	$(x \wedge y) \wedge z$	$x \wedge (y \wedge z)$	$x \Rightarrow y$	$\neg x \vee y$	$x \wedge (x \vee y)$	$\neg(x \vee y)$	$\neg x \wedge \neg y$	y	$x \vee 0$	x
0	0	0	0	0	1	1	0	1	1	0	0	0
0	0	1	0	0	1	1	0	1	1	0	0	0
0	1	0	0	0	1	1	0	0	0	1	0	0
0	1	1	0	0	1	1	0	0	0	1	0	0
1	0	0	0	0	0	0	1	0	0	0	1	1
1	0	1	0	0	0	0	1	0	0	0	1	1
1	1	0	0	0	1	1	1	0	0	1	1	1
1	1	1	1	1	1	1	1	0	0	1	1	1

demonstrates that

$$x \wedge (x \vee y) \not\equiv y.$$

This is an incorrect version of the absorption axiom, which could be corrected to read

$$x \wedge (x \vee y) \equiv x.$$

▷ **S22.** Adding parentheses for clarity throughout, using either derivation

$$\begin{array}{cll}
 & (x \wedge x) \vee (x \wedge \neg x) & \\
= & x \wedge (x \vee \neg x) & (distribution)
\end{array}$$

$$\begin{array}{cll}
 & (x \wedge x) \vee (x \wedge \neg x) & \\
= & x \wedge (x \vee \neg x) & (distribution) \\
= & (x \vee \neg x) \wedge x & (commutativity)
\end{array}$$

$$\begin{array}{cll}
 & (x \wedge x) \vee (x \wedge \neg x) & \\
= & x \vee (x \wedge \neg x) & (idempotency) \\
= & x \vee 0 & (inverse) \\
= & x & (identity)
\end{array}$$

$$\begin{array}{cll}
 & (x \wedge x) \vee (x \wedge \neg x) & \\
= & \neg(\neg(x \wedge x) \wedge \neg(x \wedge \neg x)) & (deMorgan) \\
= & \neg((\neg x \vee \neg x) \wedge (\neg x \vee x)) & (deMorgan)
\end{array}$$

or, failing that, enumeration

x	$x \wedge x \vee x \wedge \neg x$	$x \wedge (x \vee \neg x)$	$(x \vee \neg x) \wedge x$	x	$\neg x$	$\neg((\neg x \vee \neg x) \wedge (\neg x \vee x))$
0	0	0	0	0	1	0
1	1	1	1	1	0	1

means we can conclude that

$$x \wedge x \vee x \wedge \neg x \not\equiv \neg x.$$

▷ **S23.** At first glance, this question looks like a lot of work. However, we can immediately rule out several options because the associated Karnaugh maps are clearly invalid:

- option A is invalid because the dimensions do not match the truth table: it ignores z s is for a 3-input rather than 4-input function,
- option B is invalid because the content does not match the truth table: the truth table has 6 entries equal to 1 whereas the Karnaugh map has 5,
- option D is invalid because the 3-element red group is invalid: groups must be rectangular, but this is L-shaped.

So only options C and E remain. Even just *looking* at them, we can guess that option C will yield a more efficient expression because it uses fewer, larger groups (option E uses unit-sized groups only). In more detail

- Option C yields

$$r \;=\; f(w,x,y,z) \;=\; \begin{array}{ccccccccccc}
(& & & \neg x & & & \wedge & \neg z &) & \vee \\
(& w & & & \wedge & \neg y & \wedge & \neg z &) & \vee \\
(& w & \wedge & \neg x & \wedge & \neg y & & &)
\end{array}$$

and thus 5 AND, 2 OR, and 6 NOT operators.

- Option E yields
  $$\begin{array}{lclcccccccccccc} r & = & f(w,x,y,z) & = & (& w & \wedge & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\ &&&& (& \neg w & \wedge & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\ &&&& (& \neg w & \wedge & \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ &&&& (& w & \wedge & \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ &&&& (& w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ &&&& (& w & \wedge & \neg x & \wedge & \neg y & \wedge & z &) & \end{array}$$
  and thus 18 AND, 5 OR, and 16 NOT operators.

Even considering the (significant) potential for applying common sub-expression, e.g., computing and sharing the result of $\neg x$ once versus using using one operator for each instance, option C will clearly involve fewer operators.

▷ **S24.** Adding parentheses for clarity throughout, using either derivation

$$\begin{array}{cll} & (x \wedge y) \vee (x \wedge y \wedge z) & \\ = & (x \wedge y \wedge 1) \vee (x \wedge y \wedge z) & (identity) \\ = & (x \wedge y) \wedge (1 \vee z) & (distribution) \\ = & (x \wedge y) \wedge (z \vee 1) & (commutativity) \\ = & x \wedge y \wedge 1 & (null) \\ = & x \wedge y & (null) \end{array}$$

or, failing that, enumeration

x	y	z	$(x \wedge y) \vee (x \wedge y \wedge z)$	$x \wedge y$	$x \wedge z$	$y \wedge z$	$x \wedge y \wedge z$	1
0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	1
0	1	0	0	0	0	0	0	1
0	1	1	0	0	0	1	0	1
1	0	0	0	0	0	0	0	1
1	0	1	0	0	1	0	0	1
1	1	0	1	1	0	0	0	1
1	1	1	1	1	1	1	1	1

means we can conclude that

$$x \wedge y \vee x \wedge y \wedge z \equiv x \wedge y.$$

▷ **S25.** One can show that there are 2^{2^n} possible Boolean functions with n inputs. In this case $n = 1$ so we know there are $2^{2^1} = 2^2 = 4$ such functions, which we can enumerate as follows:

x	f_0	f_1	f_2	f_3
0	0	0	1	1
1	0	1	0	1

In essence, f_0 is the constant 0 function (noting $f_0(f_0(x)) = 0 = f_0(x)$), f_1 is the identity function (noting $f_1(f_1(x)) = x = f_1(x)$), f_2 is the complement function (noting $f_2(f_2(x)) = \neg\neg x = x \neq \neg x = f_2(x)$), and f_3 is the constant 1 function (noting $f_3(f_3(x)) = 1 = f_3(x)$), As such, only 1 of the 4 possible functions, namely f_2, is not idempotent.

▷ **S26.** One can show that there are 2^{2^n} possible Boolean functions with n inputs. In this case $n = 2$ so we know there are $2^{2^2} = 2^4 = 16$ such functions, which we can enumerate as follows:

x	y	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

The cases where $x = y = 0$ and $x = y = 1$ are naturally symmetric, so the question is basically whether or not the remaining cases are *also* symmetric, i.e., whether or not $f_i(0,1) = f_i(1,0)$ holds for a given i. By inspection of the truth table above, one can see that the relationship holds for each

$$i \in S = \{0, 1, 6, 7, 8, 9, 14, 15\},$$

noting that $|S| = 8$ possible functions are therefore symmetric. As an aside, symmetric Boolean functions are known as Boolean counting functions in some contexts; one can prove that there are 2^{n+1} such functions with n inputs, which in our case means $2^{n+1} = 2^{2+1} = 2^3 = 8$ as expected.

▷ **S27.** Adding parentheses for clarity throughout, using either derivation

$$\begin{array}{clr}
 & (x \wedge (\neg x \vee y)) \vee y & \\
= & (x \wedge \neg x) \vee (x \wedge y) \vee y & (distribution) \\
= & (x \wedge \neg x) \vee y \vee (x \wedge y) & (commutativity) \\
= & (x \wedge \neg x) \vee y \vee (y \wedge x) & (commutativity) \\
= & (x \wedge \neg x) \vee y & (absorption) \\
= & (x \wedge \neg x) \vee (y \wedge 1) & (identity) \\
= & (x \wedge \neg x) \vee (y \wedge (x \vee 1)) & (null) \\
= & (x \wedge \neg x) \vee (y \wedge (1 \vee x)) & (commutativity)
\end{array}$$

$$\begin{array}{clr}
 & (x \wedge (\neg x \vee y)) \vee y & \\
= & (x \wedge \neg x) \vee (x \wedge y) \vee y & (distribution) \\
= & 0 \vee (x \wedge y) \vee y & (inverse)
\end{array}$$

$$\begin{array}{clr}
 & (x \wedge (\neg x \vee y)) \vee y & \\
= & (x \wedge \neg x) \vee (x \wedge y) \vee y & (distribution) \\
= & 0 \vee (x \wedge y) \vee y & (inverse) \\
= & (x \wedge y) \vee 0 \vee y & (commutativity) \\
= & (x \wedge y) \vee y \vee 0 & (commutativity) \\
= & (x \wedge y) \vee y & (identity) \\
= & y \vee (x \wedge y) & (commutativity) \\
= & y \vee (y \wedge x) & (commutativity) \\
= & y & (absorption)
\end{array}$$

$$\begin{array}{clr}
 & (x \wedge (\neg x \vee y)) \vee y & \\
= & \neg(\neg(x \wedge (\neg x \vee y)) \wedge \neg y) & (deMorgan) \\
= & \neg((\neg x \vee \neg(\neg x \vee y)) \wedge \neg y) & (deMorgan) \\
= & \neg((\neg x \vee (x \wedge \neg y)) \wedge \neg y) & (deMorgan)
\end{array}$$

or, failing that, enumeration

x	y	$x \wedge (\neg x \vee y) \vee y$	$x \wedge \neg x \vee y \wedge (1 \vee x)$	$0 \vee x \wedge y \vee y$	$x \wedge y$	y	$\neg((\neg x \vee (x \wedge \neg y)) \wedge \neg y)$
0	0	0	0	0	0	0	0
0	1	1	1	1	0	1	1
1	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1

means we can conclude that

$$x \wedge (\neg x \vee y) \vee y \quad \not\equiv \quad x \wedge y.$$

▷ **S28.** Adding parentheses for clarity throughout, using either derivation

$$\begin{array}{clr}
 & (x \vee y) \wedge (x \vee \neg y) & \\
= & x \vee (y \wedge \neg y) & (distribution) \\
= & x \vee 0 & (inverse) \\
= & x & (identity)
\end{array}$$

or, failing that, enumeration

x	y	$(x \vee y) \wedge (x \vee \neg y)$	$\neg x$	x	$\neg y$	y	0
0	0	0	1	0	1	0	0
0	1	0	1	0	0	1	0
1	0	1	0	1	1	0	0
1	1	1	0	1	0	1	0

means we can conclude that

$$(x \vee y) \wedge (x \vee \neg y) \quad \equiv \quad x.$$

▷ **S29.** a **false**. Notice that f is a 2-input, 1-output Boolean function. Since it has $n = 2$ inputs, there are $2^{2^n} = 2^{2^2} = 16$ possible functions, some of which are as follows:

x	y	f_0	f_{10}	f_{12}	f_{15}
0	0	0	0	0	1
0	1	0	1	0	1
1	0	0	0	1	1
1	1	0	1	1	1

These could be described as a "constant 0" function $f_0(x, y) = 0$, an "identity" function $f_{10}(x, y) = y$, an "identity" function $f_{12}(x, y) = x$, and a "constant 1" function $f_{15}(x, y) = 1$ respectively; they can all be implemented using 0 Boolean operators.

b **true**. First, note that by disjunction and conjunction we essentially mean OR and AND, i.e., $\vee$ and $\wedge$. We can then answer the question by giving example analogies, with (at least) 3 representing obvious candidates:

i the axioms that govern them are similar: $x \vee \textbf{false} = x$ and $x \wedge \textbf{true} = x$ are analogous to $x + 0 = x$ and $x \times 1 = x$, for example.

ii the specification of associated operators is similar:

x	y	$r = x \wedge y$
false	**false**	**false**
false	**true**	**false**
true	**false**	**false**
true	**true**	**true**

is analogous to

x	y	$r = x \times y$
0	0	0
0	1	0
1	0	0
1	1	1

for example.

iii there is an explicit cross-over of some terminology: within Boolean algebra we have Sum of Products (SoP), i.e., AND'ing together terms formed by OR'ing together variables and Product of Sums (PoS), i.e., OR'ing together terms formed by AND'ing together variables, for example, both of which associated addition with AND and multiplication with OR.

▷ **S30.** a The truth table for this function is as follows

a	b	c	$(a \wedge b \wedge \neg c)$	$(a \wedge \neg b \wedge c)$	$(\neg a \wedge \neg b \wedge c)$	$f(a, b, c)$
0	0	0	0	0	0	0
0	0	1	0	0	1	1
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	0	0	0	0
1	0	1	0	1	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	0

Since there are $n = 3$ input variables, there are clearly $2^n = 2^3 = 8$ input combinations; three of these produce 1 as an the output from the function.

b The truth table for this function is as follows

a	b	c	d	$f(a,b,c,d)$
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

so since there is only one case where $f(a,b,c,d) = 1$, the only assignment given which matches the criteria is $a = 0$, $b = 1$, $c = 0$ and $d = 1$.

This hints at a general principle: when we have an expression like this, a term such as $\neg x$ can be read as "x should be 0" and x as "x should be 1". So the expression as a whole is read as "a should be 0 and b should be 1 and c should be 0 and d should be 1". Since we have basically fixed all four inputs, only *one* entry of the truth table matches. On the other hand, if we instead had

$$f(a,b,c,d) = \neg a \wedge b \wedge \neg c$$

for example, we would be saying "a should be 0 and b should be 1 and c should be 0, and d can be anything" which gives *two* possible assignments (i.e., $a = 0$, $b = 1$, $c = 0$ and either $d = 0$ or $d = 1$).

c Informally, SoP form means there are say n terms in the expression: each term is the conjunction of some variables (or their complement), and the expression is the disjunction of the terms. As conjunction and disjunction basically means the AND and OR operators, and AND and OR act sort of like multiplication and addition, the SoP name should make some sense: the expression is sort of like the sum of terms which are themselves each a product of variables. The second option is correct as a result; the first and last violate the form described above somehow (e.g., the first case is in the opposite, PoS form).

d One can easily make a comparison using a truth table such as

a	b	$a \vee 1$	$a \oplus 1$	$\neg a$	$a \wedge 1$	$\neg(a \wedge b)$	$\neg a \vee \neg b$
0	0	1	1	1	0	1	1
0	1	1	1	1	0	1	1
1	0	1	0	0	1	1	1
1	1	1	0	0	1	0	0

from which it should be clear that all the equations are correct except for the first one. That is, $a \vee 1 \neq a$ but rather $a \vee 1 = 1$.

e i Inspecting the following truth table

a	$\neg a$	$\neg\neg a$
0	1	0
0	1	0
1	0	1
1	0	1

shows this equivalence is correct (this is the involution axiom).

ii Inspecting the following truth table

a	b	$\neg a$	$\neg b$	$a \wedge b$	$\neg(a \wedge b)$	$\neg a \vee \neg b$
0	0	1	1	0	1	1
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	0

shows this equivalence is correct (this is the de Morgan axiom).

iii Inspecting the following truth table

a	b	$\neg a$	$\neg b$	$\neg a \wedge b$	$a \wedge \neg b$
0	0	1	1	0	0
0	1	1	0	1	0
1	0	0	1	0	1
1	1	0	0	0	0

shows this equivalence is incorrect.

iv Inspecting the following truth table

a	$\neg a$	$a \oplus a$
0	1	0
1	0	0

shows this equivalence is incorrect.

▷ **S31.** a The dual of any expression is constructed by using the principle of duality, which informally means swapping each AND with OR (and vice versa) and each 0 with 1 (and vice versa); this means, for example, we can take the OR form of each axiom and produce the AND form (and vice versa).

So in this case, we start with an OR form: this means the dual will the corresponding AND form. Making the swaps required means we end up with

$$x \wedge 0 \equiv 0$$

so the second option is correct.

b This question is basically asking for the complement of f, since the options each have $\neg f$ on the left-hand side: this means using the principle of complements, a generalisation of the de Morgan axiom, by swapping each variable with the complement (and vice versa), each AND with OR (and vice versa), and each 0 with 1 (and vice versa). If we apply these rules (taking care with the parenthesis) to

$$f = \neg a \wedge \neg b \vee \neg c \vee \neg d \vee \neg e,$$

we end up with

$$\neg f = (a \vee b) \wedge c \wedge d \wedge e$$

which matches the last option.

c The de Morgan axiom, which can be generalised using by the principle of complements, says that

$$\neg(x \wedge y) \equiv \neg x \vee \neg y$$

or conversely that

$$\neg(x \vee y) \equiv \neg x \wedge \neg y$$

You can think of either form as "pushing" the NOT operator on the left-hand side into the parentheses: this acts to complement each variable, and swap the AND to an OR (or vice versa). We know that

$$\begin{array}{lcl} x \overline{\wedge} y & \equiv & \neg(x \wedge y) \\ x \overline{\vee} y & \equiv & \neg(x \vee y) \end{array}$$

So pattern matching against the options, it is clear the first one is correct, for example, because

$$x \overline{\vee} y \equiv \neg(x \vee y) \equiv \neg x \wedge \neg y$$

where the right-hand side matches the description of an AND whose two inputs are complemented. Likewise, the second one is correct because

$$x \overline{\wedge} y \equiv \neg(x \wedge y) \equiv \neg x \vee \neg y.$$

▷ **S32.** a The third option, i.e., $\neg a \wedge \neg b$ is the correct one; the three simplification steps, via two axioms, are as follows:

$$\begin{array}{llllllll} & \neg & (a \vee b) & \wedge & \neg & (c \vee d \vee e) & \vee & \neg & (a \vee b) & \\ = & & (\neg a \wedge \neg b) & \wedge & \neg & (c \vee d \vee e) & \vee & & (\neg a \wedge \neg b) & \text{(de Morgan)} \\ = & & (\neg a \wedge \neg b) & \wedge & & (\neg c \wedge \neg d \wedge \neg e) & \vee & & (\neg a \wedge \neg b) & \text{(de Morgan)} \\ = & & \neg a \wedge \neg b & & & & & & & \text{(absorption)} \end{array}$$

b We can clearly see that

$$\begin{array}{cll} & (a \vee b \vee c) \wedge \neg(d \vee e) \vee (a \vee b \vee c) \wedge (d \vee e) & \\ = & (a \vee b \vee c) \wedge (\neg(d \vee e) \vee (d \vee e)) & \text{(distribution)} \\ = & (a \vee b \vee c) \wedge ((d \vee e) \vee \neg(d \vee e)) & \text{(commutativity)} \\ = & (a \vee b \vee c) \wedge 1 & \text{(inverse)} \\ = & a \vee b \vee c & \text{(identity)} \end{array}$$

meaning the first option is the correct one.

c We can clearly see that

$$\begin{array}{cll} & a \wedge c \vee c \wedge (\neg a \vee a \wedge b) & \\ = & (a \wedge c) \vee (c \wedge (\neg a \vee (a \wedge b))) & \text{(precedence)} \\ = & (c \wedge a) \vee (c \wedge (\neg a \vee (a \wedge b))) & \text{(commutativity)} \\ = & c \wedge (a \vee \neg a \vee (a \wedge b)) & \text{(distribution)} \\ = & c \wedge (1 \vee (a \wedge b)) & \text{(inverse)} \\ = & c \wedge ((a \wedge b) \vee 1) & \text{(commutativity)} \\ = & c \wedge 1 & \text{(null)} \\ = & c & \text{(identity)} \end{array}$$

meaning the last option is the correct one: *none* of the above is correct, since the correct simplification is actually just c.

d The fourth option, i.e., $a \wedge b$ is correct. This basically stems from repeated application of the absorption axiom, the AND form of which states

$$x \vee (x \wedge y) \equiv x.$$

Applying it from left-to-right, we find that

$$\begin{array}{cll} & a \wedge b \vee a \wedge b \wedge c \vee a \wedge b \wedge c \wedge d \vee a \wedge b \wedge c \wedge d \wedge e \vee a \wedge b \wedge c \wedge d \wedge e \wedge f & \\ = & (a \wedge b) \vee (a \wedge b) \wedge (c) \vee (a \wedge b) \wedge (c \wedge d) \vee (a \wedge b) \wedge (c \wedge d \wedge e) \vee (a \wedge b) \wedge (c \wedge d \wedge e \wedge f) & \text{(precedence)} \\ = & (a \wedge b) \vee (a \wedge b) \wedge (c \wedge d) \vee (a \wedge b) \wedge (c \wedge d \wedge e) \vee (a \wedge b) \wedge (c \wedge d \wedge e \wedge f) & \text{(absorption)} \\ = & (a \wedge b) \vee (a \wedge b) \wedge (c \wedge d \wedge e) \vee (a \wedge b) \wedge (c \wedge d \wedge e \wedge f) & \text{(absorption)} \\ = & (a \wedge b) \vee (a \wedge b) \wedge (c \wedge d \wedge e \wedge f) & \text{(absorption)} \\ = & (a \wedge b) & \text{(absorption)} \end{array}$$

e We can simplify this function as follows

$$\begin{array}{rcll} f(a,b,c) & = & (a \wedge b) \vee a \wedge (a \vee c) \vee b \wedge (a \vee c) & \\ & = & (a \wedge b) \vee a \vee b \wedge (a \vee c) & \text{(absorbtion)} \\ & = & a \vee (a \wedge b) \vee b \wedge (a \vee c) & \text{(commutitivity)} \\ & = & a \vee b \wedge (a \vee c) & \text{(absorbtion)} \\ & = & a \vee (b \wedge a) \vee (b \wedge c) & \text{(distribution)} \\ & = & a \vee (a \wedge b) \vee (b \wedge c) & \text{(commutitivity)} \\ & = & a \vee (b \wedge c) & \text{(commutitivity)} \end{array}$$

at which point there is nothing else that can be done: we end up with 2 operators (and AND and an OR), so the second option is correct.

f Working from the right-hand side toward the left, we have that

$$\begin{array}{cll} & \neg x \vee \neg y & \\ = & (\neg x \wedge 1) \vee \neg y & \text{(identity)} \\ = & (\neg x \wedge 1) \vee (\neg y \wedge 1) & \text{(identity)} \\ = & (\neg x \wedge (y \vee \neg y)) \vee (\neg y \wedge 1) & \text{(inverse)} \\ = & (\neg x \wedge (y \vee \neg y)) \vee (\neg y \wedge (x \vee \neg x)) & \text{(inverse)} \\ = & (\neg x \wedge y) \vee (\neg x \wedge \neg y) \vee (\neg y \wedge (x \vee \neg x)) & \text{(distribution)} \\ = & (\neg x \wedge y) \vee (\neg x \wedge \neg y) \vee (\neg y \wedge x) \vee (\neg y \wedge \neg x) & \text{(distribution)} \\ = & (\neg x \wedge y) \vee (\neg y \wedge x) \vee (\neg x \wedge \neg y) \vee (\neg x \wedge \neg y) & \text{(commutativity)} \\ = & (\neg x \wedge y) \vee (\neg y \wedge x) \vee (\neg x \wedge \neg y) & \text{(idempotency)} \end{array}$$

g By writing

$$\begin{array}{rcccc} t_0 & = & x & \wedge & y \\ t_1 & = & y & \wedge & z \\ t_2 & = & y & \vee & z \\ t_3 & = & x & \vee & z \\ t_4 & = & t_1 & \wedge & t_2 \end{array}$$

we can shorten the LHS and RHS to

$$\begin{array}{ccccc} f & = & t_0 & \vee & t_4 \\ g & = & y & \wedge & t_3 \end{array}$$

and then perform a brute-force enumeration

x	y	z	t_0	t_1	t_2	t_3	t_4	f	g
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	0	0	0
0	1	0	0	0	1	0	0	0	0
0	1	1	0	1	1	1	1	1	1
1	0	0	0	0	0	1	0	0	0
1	0	1	0	0	1	1	0	0	0
1	1	0	1	0	1	1	0	1	1
1	1	1	1	1	1	1	1	1	1

to demonstrate that $f = g$, i.e., the equivalence holds. Note that this approach is not as robust if the intermediate steps are not shown; simply including f and g in the truth table does not give much more confidence that simply writing the equivalence!

To prove the equivalence using an axiomatic approach, the following steps can be applied:

$$\begin{array}{cll} & (x \wedge y) \vee (y \wedge z \wedge (y \vee z)) & \\ = & (x \wedge y) \vee (y \wedge z \wedge y) \vee (y \wedge z \wedge z) & \text{(distribution)} \\ = & (x \wedge y) \vee (y \wedge y \wedge z) \vee (y \wedge z \wedge z) & \text{(commutativity)} \\ = & (x \wedge y) \vee (y \wedge z) \vee (y \wedge z) & \text{(idempotency)} \\ = & (x \wedge y) \vee (y \wedge z) & \text{(idempotency)} \\ = & (y \wedge x) \vee (y \wedge z) & \text{(commutativity)} \\ = & y \wedge (x \vee z) & \text{(distribution)} \end{array}$$

h Using four simplification steps, via three axioms and the AND operator, as follows

$$\begin{array}{cll} & \neg(a \vee b) \wedge \neg(\neg a \vee \neg b) & \\ = & (\neg a \wedge \neg b) \wedge (a \wedge b) & \text{(de Morgans)} \\ = & (\neg a \wedge a) \wedge (\neg b \wedge b) & \text{(association)} \\ = & (\neg a \wedge a) \wedge (\neg b \wedge b) & \text{(identity)} \\ = & \textbf{false} \wedge \textbf{false} & \text{(operator)} \\ = & \textbf{false} & \end{array}$$

we get a form that contains zero operators (which by definition *must* be the fewest).

Part II: Basics of digital logic: general

▷ **S33.** We can deal with the first two statements in one go: an N-MOSFET (or N-type MOSFET) has N-type semiconductor terminals and a P-type body. If the type of semiconductors were swapped, we have a P-MOSFET (or P-type MOSFET) not an N-MOSFET.

A CMOS cell *is* a pairing of two transistors. However, it depends on use of complementary types, namely one N-type and one P-type, rather than the same type as suggested. As such, this statement is incorrect (although subtly so).

A given N-MOSFET is deemed active (resp. inactive) when current is allowed (resp. disallowed) to flow. The flow is, in some sense, controlled by the voltage level applied: it acts to widen or narrow the associated depletion region. At some threshold, “enough” voltage is applied for there to be “enough” current flowing for source and drain to be deemed connected and hence the MOSFET active. So this statement is true, although arguably some detail is glossed over (e.g., the fact a leakage current will *always* exist, so a transistor is *always* a little bit active).

▷ **S34.** This is a NAND gate, so clearly there *are* two inputs and one output: by inspecting the circuit we can see them labelled x, y and r, plus identify the two power rails labelled V_{dd} and V_{ss}. The output r is “pulled-up” to the V_{dd} voltage level iff. a connection is made via one or other of the top two transistors. Their parallel arrangement gives a hint at their type, even if the system is not recognisable. If we look at the truth table for NAND, i.e.,

x	y	r
0	0	1
0	1	1
1	0	1
1	1	0

then if either $x = 0$ or $y = 0$ then $r = 1$, where $V_{ss} \equiv 0$ and $V_{dd} \equiv 1$: these must be P-MOSFETS, therefore, because we want a connection to be formed between r and V_{dd} if $x = V_{ss}$ or $y = V_{ss}$. There is, of course, a companion pull-down network allowing r to be "pulled-down" to the V_{ss} voltage level. However, this is constructed from a sequential arrangement of N-MOSFETS: we did not study BJT transistors, which represent a different technology from MOSFETs.

Finally, note that it is highly unlikely you will see a flux capacitor in a circuit other than during Back To The Future!

▷ **S35.** Provided you know what the behaviour of the pull-up network (top, consisting of P-MOSFETs) and pull-down network (bottom, consisting of N-MOSFETs) is, it is *reasonably* easy (if long winded) to answer the question by looking at a case-by-case analysis. A more efficient way is to spot the sequential and parallel organisation of MOSFETs:

- If $x = V_{ss}$, y and z are irrelevant because a connection between r and V_{dd} is formed (via the bottom-left P-MOSFET), while a connection between r and V_{dd} is impossible (due to the top N-MOSFET).
- If $x = V_{dd}$, y and z are relevant:
  - If $y = V_{ss}$, $z = V_{ss}$ then a path between r and V_{ss} is impossible; in this case, however, a path between r and V_{dd} is formed (via the top P-MOSFETs).
  - If $y = V_{ss}$, $z = V_{dd}$ then a path between r and V_{ss} is formed (via the top and bottom-right N-MOSFETs).
  - If $y = V_{dd}$, $z = V_{ss}$ then a path between r and V_{ss} is formed (via the top and bottom-left N-MOSFETs).
  - If $y = V_{dd}$, $z = V_{dd}$ then a path between r and V_{dd} is impossible; in this case, however, a path between r and V_{ss} is formed (via the bottom N-MOSFETs).

So given $V_{ss} \equiv 0$ and $V_{dd} \equiv 1$, we can write

x	y	z	$r = f(x,y,z)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

which, translated into an expression, is $r = \neg(x \wedge (y \vee z)$ or $\neg x \vee (\neg y \wedge \neg z)$ if you prefer: basically $r = 1$ if $x = 0$ or both $y = 0$ and $z = 0$, otherwise $r = 0$.

▷ **S36.** This question is tricky, in the sense there are *lots* of ways an XOR gate can be constructed using logic gate instances. One can show, for example, that $x \oplus y$ is equivalent to

a $(\neg x \wedge y) \vee (x \wedge \neg y)$,
b $(x \vee y) \wedge \neg(x \wedge y)$,
c $t_0 \overline{\wedge} t_1$ where $t_0 = (x \overline{\wedge} x) \overline{\wedge} y$ and $t_1 = (y \overline{\wedge} y) \overline{\wedge} x$, or
d $t_1 \overline{\wedge} t_2$ where $t_0 = x \overline{\wedge} y$, $t_1 = x \overline{\wedge} t_0$, and $t_2 = y \overline{\wedge} t_0$,
e $(x \overline{\vee} y) \overline{\vee} \neg(x \overline{\wedge} y)$

and potentially more besides: note that the question rules out the otherwise viable option of directly using transistors, for example.

To compare the options, the starting point is how each logic gate we *could* use will be realised using transistors. In reality this may increase the range of possible answers even further, but to provide *an* answer imagine that we only consider the four options above, and assume that

NOT	$\rightsquigarrow$	2 transistors
NAND	$\rightsquigarrow$	4 transistors
NOR	$\rightsquigarrow$	4 transistors
AND	$\rightsquigarrow$	6 transistors
OR	$\rightsquigarrow$	6 transistors

i.e., since we *know* we can construct NOT, NAND and NOR using the stated number of MOSFET-based transistors, the best way to form AND and OR is simply to append a NOT to a NAND or NOR. This means we can just count the logic gate instances, and translate:

$$\begin{array}{lll} 2 \cdot \text{NOT} + 2 \cdot \text{AND} + 1 \cdot \text{OR} & \rightsquigarrow & 22 \;\; \text{transistors} \\ 1 \cdot \text{NOT} + 2 \cdot \text{AND} + 1 \cdot \text{OR} & \rightsquigarrow & 20 \;\; \text{transistors} \\ 5 \cdot \text{NAND} & \rightsquigarrow & 20 \;\; \text{transistors} \\ 4 \cdot \text{NAND} & \rightsquigarrow & 16 \;\; \text{transistors} \\ 1 \cdot \text{NOT} + 1 \cdot \text{NAND} + 2 \cdot \text{NOR} & \rightsquigarrow & 14 \;\; \text{transistors} \end{array}$$

Based on this 14 is the correct answer, although keep in mind it *might* be possible to do better based on using a different set of options (for XOR) and assumptions.

▷ **S37.** This is quite a tricky question, in the sense there are several plausible answers: selecting between them really needs some justification, which of course is impossible with a multiple choice question! It is important to keep in mind that the question focuses on design of some behaviour based on transistors, *not* precision wrt. manufacture of the design. Put another way, the question is intentionally pitched at a high-level, with the various caveats attempting to limit the possible answers:

a It might *seem* possible to use 0 transistors, in that one can just connect x directly to r. This may realise the pass through behaviour required, but does not impose a delay (and more generally does not satisfy use-cases for current or voltage buffers) so is not really valid.

b One could implement a buffer using 1 N-MOSFET transistor, say, in series with a pull-down resistor: the idea is that when $x = V_{dd}$ the transistor connects r to V_{dd}, otherwise the resistor pulls r down to V_{ss}.

However, the material provided does not cover use of resistors: the question caters for this by asking for an implementation using only transistors, and not listing 1 as an answer! This is justified further by the fact by placing an emphasis on CMOS, using only an N-MOSFET might seem confusing therefore (given the argument that N- and P-MOSFETs occur in pairs as part of a pull-down and pull-up network). So although this is arguably the best answer, it is not the expected one!

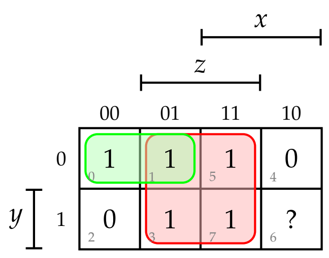
c One could start by considering a NOT gate implementation, which will clearly invert x. It does so via a P-MOSFET that connects V_{dd} to r, and an N-MOSFET that connects V_{ss} to r. As such, when $x = V_{ss}$ (resp. $x = V_{dd}$) the P-MOSFET will be connected and N-MOSFET disconnected (resp. vice versa) meaning that $r = V_{dd} \simeq \neg x$ (resp. $r = V_{ss} \simeq \neg x$). A somewhat simple observation is that if we swap the P- and N-MOSFETs, we end up with a buffer. That is, if the P-MOSFET connects V_{ss} to r and the N-MOSFET connects V_{dd} to r, then the behaviour swaps st. if $x = V_{dd}$ (resp. $x = V_{ss}$) then $r = V_{dd} \simeq x$ (resp. $r = V_{ss} \simeq x$).

So 2 transistors is a reasonable answer *if* we assume it is possible to organise them this way. In reality, this is debatable: it is the *opposite* of normal pull-down and pull-up networks connected to V_{dd} and V_{ss}, so may not be reasonable under the constraints of a given manufacturing process. However, the question is careful to ask for an unconstrained organisation for transistors, so the assumption is allowed.

d Finally, one could simply use two NOT gate implementations in series with each other: this inverts x twice, computing $r = \neg\neg x = x$ using 4 transistors. This approach needs no assumptions, but on the other hand is obviously not very efficient wrt. the number of transistors used.

In summary then, 2 transistors is a correct (or the expected) answer in this case (although you could quite reasonably argue for other answers).

▷ **S38.** From the truth table, one can form a Karnaugh map



which includes two groups: unlike some other examples, the don't care in this case is *uncovered* (i.e., we treat it as 0) since we cannot make fewer or larger groups by covering it (i.e., treating it as 1). Even so, translating each group into a term yields the SoP expression

$$r = f(x, y, z) = (\neg x \wedge \neg y) \vee z$$

which is the correct answer: the fact this is the only one in SoP form at least *hints* at the correct answer even without going through the above.

▷ **S39.** To answer this question, the idea is that we

- use a 3-level cascade to produce an 8-input, 1-bit multiplexer, then
- replicate this 8 times to produce an 8-input, 8-bit multiplexer

using a generic 2-input, 1-bit multiplexer whose behaviour can be described as

$$r = \begin{cases} x & \text{if } c = 0 \\ y & \text{otherwise} \end{cases}$$

i.e., it selects the input x (i.e., connects the output r to x) if the control signal $c = 0$, and selects the input y (i.e., connects the output r to y) if the control signal $c = 1$.

Imagine the 8, 1-bit inputs are named s, t, u, v, w, x, y and z and there is a 3-bit control signal c (to select between $2^3 = 8$ inputs). The cascade of 2-input, 1-bit multiplexers is constructed as follows

$$\begin{array}{lclclclcl}
t_0 & = & \begin{cases} s & \text{if } c_0 = 0 \\ t & \text{otherwise} \end{cases} & & & & & & \\
t_1 & = & \begin{cases} u & \text{if } c_0 = 0 \\ v & \text{otherwise} \end{cases} & t_4 & = & \begin{cases} t_0 & \text{if } c_1 = 0 \\ t_1 & \text{otherwise} \end{cases} & & & \\
 & & & & & & r & = & \begin{cases} t_4 & \text{if } c_2 = 0 \\ t_5 & \text{otherwise} \end{cases} \\
t_2 & = & \begin{cases} w & \text{if } c_0 = 0 \\ x & \text{otherwise} \end{cases} & t_5 & = & \begin{cases} t_2 & \text{if } c_1 = 0 \\ t_3 & \text{otherwise} \end{cases} & & & \\
t_3 & = & \begin{cases} y & \text{if } c_0 = 0 \\ z & \text{otherwise} \end{cases} & & & & & &
\end{array}$$

noting there are 3 layers (i.e., they form a tree of depth 3). Using a table such as

c	c_2	c_1	c_0	t_0	t_1	t_2	t_3	t_4	t_5	r
0	0	0	0	s	u	w	y	s	w	s
1	0	0	1	t	v	x	z	t	x	t
2	0	1	0	s	u	w	y	u	y	u
3	0	1	1	t	v	x	z	v	z	v
4	1	0	0	s	u	w	y	s	w	w
5	1	0	1	t	v	x	z	t	x	x
6	1	1	0	s	u	w	y	u	y	y
7	1	1	1	t	v	x	z	v	z	z

makes it clear the c-th input is selected as the output r. Given such a component, we then just replicate it 8 times: the same control signal is used for each i-th replication, which then produces the i-th bit of r by selecting between the i-th bits of the 8-bit inputs s through to z.

We use 7 instances of the 2-input, 1-bit multiplexer for each of the cascades; there are 8 replicated cascades, so the correct answer is that we need $7 \cdot 8 = 56$ instances.

▷ **S40.** First, notice that the critical path (or longest sequential path) runs through the 4 full-adder instances (as a result of the carry chain): the 3-rd (most-significant) instance cannot produce either $co = c_4$ or r_3 until the carry propagates from the 0-th (least-significant) instance, and so is dependent on $ci = c_0$, x_0, and y_0).

Next, we need some detail about each full-adder instance: the i-th such instance will compute the sum and carry-out

$$\begin{array}{lcl}
r_i & = & x_i \oplus y_i \oplus c_i \\
c_{i+1} & = & (x_i \wedge y_i) \vee (x_i \wedge c_i) \vee (y_i \wedge c_i) \\
 & = & (x_i \wedge y_i) \vee ((x_i \oplus y_i) \wedge c_i)
\end{array}$$

from summands x_i and y_i, plus the carry-in c_i (where $c_0 = ci$ and $co = c_4$). There are two different options for

c_{i+1}, so, for the quoted gate delays, we find the critical paths from input to output can be described as:

	Input(s)	Output(s)	Critical path
	x_i, y_i	r_i	$T_{XOR} + T_{XOR} = 120$ns
	c_i	r_i	$T_{XOR} = 60$ns
Option 1	x_i, y_i	c_{i+1}	$T_{AND} + T_{IOR} + T_{IOR} = 60$ns
	c_i	c_{i+1}	$T_{AND} + T_{IOR} + T_{IOR} = 60$ns
Option 2	x_i, y_i	c_{i+1}	$T_{XOR} + T_{AND} + T_{IOR} = 100$ns
	c_i	c_{i+1}	$T_{AND} + T_{IOR} = 40$ns

The correct answer will clearly differ depending on the option we select for c_{i+1}, but imagine we select the latter: irrespective of whether it is better or worse, it matches the lecture slide(s). As such, we can deduce the following:

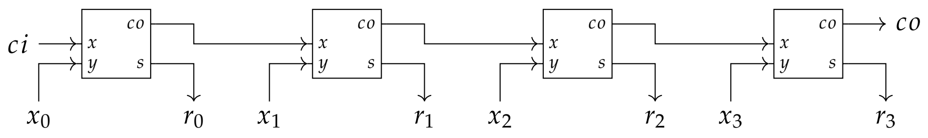
- For the 0-th full-adder instance, it takes 100ns to generate c_1 from c_0, x_0, and y_0.
- For the 1-st full-adder instance, it takes 40ns to generate c_2 from c_1, The reason it is *not* 100ns, as you might expect, is because the gate computing $x_1 \oplus y_1$ can produce an output *before* c_1 is available; this means it does not contribute to the critical path.
- For the 2-nd full-adder instance, it takes 40ns to generate c_3 from c_2, x_2, and y_2. The reason it is *not* 100ns, as you might expect, is because the gate computing $x_2 \oplus y_2$ can produce an output *before* c_2 is available; this means it does not contribute to the critical path.
- For the 3-rd instance, it takes 40ns to generate c_4 from c_3, The reason it is *not* 100ns, as you might expect, is because the gate computing $x_3 \oplus y_3$ can produce an output *before* c_3 is available; this means it does not contribute to the critical path. Likewise, it takes 60ns to generate r_3 from c_3, x_3, and y_3; for the same reason as above, this is *not* 120ns as you might expect.

So the critical path is 220ns wrt. c_4 and 240ns wrt. r_3, and therefore 240ns overall. It turns out applying similar reasoning to the former option yields a slightly longer critical path of 240ns wrt. c_4 because

- For the 0-th full-adder instance, it takes 60ns to generate c_1 from c_0, x_0, and y_0.
- For the 1-st full-adder instance, it takes 60ns to generate c_2 from c_1, x_1, and y_1.
- For the 2-nd full-adder instance, it takes 60ns to generate c_3 from c_2, x_2, and y_2.
- For the 3-rd instance, it takes 60ns to generate c_4 from c_3, x_3, and y_3, and 60ns to generate r_3 from c_3, x_3, and y_3. The reason it is *not* 120ns, as you might expect, is because the gate computing $x_3 \oplus y_3$ can produce an output *before* c_3 is available; this means it does not contribute to the critical path.

but this has no impact on the answer, which is still 240ns overall.

▷ **S41.** The optimisation they are suggested is captured by the following:



Put another way, their idea implies the half-adders used no longer have a carry-in. This is not a problem in terms of computing *an* addition: we can still use the two available inputs (vs. three for a full-adder) to provide one operand (i.e., x_i, the i-th bit of x) plus a carry-in from the previous half-adder instance. However, clearly this is not the *same* addition as previously, since the input we *would* use to provide the other operand is no longer available. That is, we can select x as normal, but can no longer provide a y input (the half-adders use that for to propagate the carry).

Other than x, the only other input we can control is ci which acts as the overall carry-in to the addition. Since $ci \in \{0, 1\}$, this means $y = 0$ and $y = 1$ are the correct answers.

▷ **S42.** Recall that a 2-input XOR operator can be described via the following truth table:

x	y	r
0	0	0
0	1	1
1	0	1
1	1	0

Crucially, this demonstrates that

$$x \oplus x \equiv 0$$

and

$$x \oplus 0 \equiv x,$$

and hence

$$x \oplus x \oplus y \equiv y$$

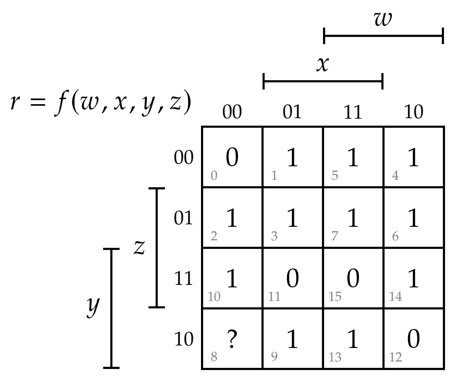
for any x and y.

Using these axioms, we can write expressions to describe the output of each XOR gate, and hence each overall output r_i. Inspecting the circuit from left-to-right and top-to-bottom, we can show

$$\begin{array}{ccccccccc}
 & & t_0 & = & x_0 \oplus x_2 & & & & \\
 & & t_1 & = & x_1 \oplus x_3 & & & & \\
r_0 & = & t_2 & = & x_0 \oplus t_0 & = & x_0 \oplus x_0 \oplus x_2 & = & x_2 \\
r_1 & = & t_3 & = & x_1 \oplus t_1 & = & x_1 \oplus x_1 \oplus x_3 & = & x_3 \\
r_2 & = & t_4 & = & t_0 \oplus t_2 & = & x_0 \oplus x_2 \oplus x_2 & = & x_0 \\
r_3 & = & t_5 & = & t_1 \oplus t_3 & = & x_1 \oplus x_3 \oplus x_3 & = & x_1
\end{array}$$

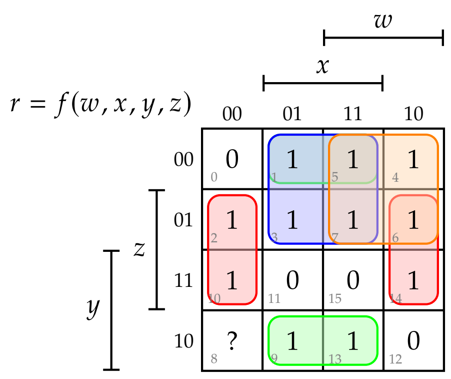
st. it becomes clear $r_0 = x_2$, $r_1 = x_3$, $r_2 = x_0$, and $r_3 = x_1$, i.e., the most- and least-significant 2-bit halves of x are swapped over to produce r.

▷ **S43.** a Although alternative organisations of the variables may alter the form, a representative Karnaugh map would be:



$r = f(w,x,y,z)$	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	0	0	1
10	?	1	1	0

The most efficient approach would attempt to form the fewest groups possible, and the largest groups possible: these will combine to minimise the number and complexity of each term in the resulting expression. As such, the 1 entries can be covered by 4 groups per



noting that a) each group covers 4 entries (with certain entries covered more than once), b) the don't care is assumed to be 0 and therefore remains uncovered, and c) 2 of the groups, in center rows and columns, wrap around the left and right, and top and bottom edges respectively.

b Performing the translation as is, we find that

$$\begin{array}{ccccl}
r & = & f(w,x,y,z) & = & (\neg x \wedge z) \vee \\
 & & & & (x \wedge \neg z) \vee \\
 & & & & (x \wedge \neg y) \vee \\
 & & & & (w \wedge \neg y)
\end{array}$$

This expression requires 4 NOT, 4 AND, and 3 OR operators and so $4 + 4 + 3 = 11$ in total. To further reduce the number of operators, we can apply various optimisation steps. First, it is possible to show that $(\neg x \wedge z) \vee (x \wedge \neg z) \equiv x \oplus z$, meaning we collapse two terms into one term (involving one XOR). Second, the term $\neg y$ is used in two terms; we can compute the result once, using one NOT, and share it between the terms. In combination, we produce the alternative

$$
\begin{array}{rcl}
r = f(w,x,y,z) & = & (\ x \oplus z\ ) \vee \\
 & & (\ x \wedge t\ ) \vee \\
 & & (\ w \wedge t\ )
\end{array}
$$

where $t = \neg y$. However, finally, it is possible to apply the distribution axiom to the latter two terms: by rewriting it as

$$
\begin{array}{rcl}
r = f(w,x,y,z) & = & (\quad (\ x \oplus z\ )\ ) \vee \\
 & & (\ t \wedge (\ x \vee w\ )\ )
\end{array}
$$

we produce an expression that requires 1 NOT, 1 AND, 2 OR and 1 XOR operators and so $1 + 2 + 2 + 1 = 5$ in total.

▷ **S44.** For SR-type latch and flip-flop components, we expect at *least* S, R and en inputs and a Q output; in contrast, for the D-type (resp. T-type) latch and flip-flop components we expect D (resp. T) and en inputs and a Q output which match. Even based on the argument that we might have Q and $\neg Q$ from one or other, this at least makes the former less likely. Additionally, we might expect S and R to be used in a controlled way so as to avoid a problematic meta-stable state; there are instances where $x = y = z = 1$ and $x = y = z = 0$, so this control is not clearly being applied.

Narrowing down the choices further requires interpretation of the signal labelling and behaviour. While somewhat tricky, it should be clear that the value of z changes to match y while $x = 1$ and is unchanged when $x = 0$. Crucially, it is not the case that z changes to match y *only* at the point where x transitions from 0 to 1 (or 1 to 0): as shown in the right-hand portion of the waveform, z changes at *any* point that $x = 1$. As such, we infer the component is likely to be a level-triggered latch not an edge triggered flip-flop. Additionally, z does not toggle between 0 and 1 while $x = 1$; it matches whatever y is.

In summary therefore, i.e., having ruled out a) SR-type components, b) flip-flop components, and c) a T-type flip-flop, we conclude that the correct answer is a D-type latch: x represents the enable signal en, y represents the input D and z represents the output Q.

▷ **S45.** First, recall that the truth table for NAND is

x	y	r
0	0	1
0	1	1
1	0	1
1	1	0

If one or other (or both) of x and y is equal to 0 then the output r *must* be 1: only when x *and* y equal 1 do we get r equal to 0. This gives a useful way to ascertain the behaviour of the circuit, in the sense we can enumerate the set of consistent states it can be in:

a If $S = 0$ then the top NAND gate must output 1, and if $R = 0$ then the bottom NAND gate must output 1. That is, $S = 0$ and $R = 0$ means we have $Q = 1$ and $\neg Q = 1$.

b If $R = 0$ then the bottom NAND gate must output 1, so if $S = 1$ then the top NAND gate outputs $1 \barwedge 1 = 0$. That is, $S = 1$ and $R = 0$ means we have $Q = 0$ and $\neg Q = 1$.

c If $S = 0$ then the top NAND gate must output 1, so if $R = 1$ then the bottom NAND gate outputs $1 \barwedge 1 = 0$. That is, $S = 0$ and $R = 1$ means we have $Q = 1$ and $\neg Q = 0$.

d If $S = 1$ and $R = 1$ then one of *two* possibilities can apply: either

  i the top NAND gate outputs 0, meaning the bottom NAND gate outputs $0 \barwedge 1 = 1$ (which is consistent with the top gate computing $1 \barwedge 1 = 0$)

  ii the bottom NAND gate outputs 0, meaning the top NAND gate outputs $0 \barwedge 1 = 1$ (which is consistent with the bottom gate computing $1 \barwedge 1 = 0$).

Put another way, $S = R = 0$ is the meta-stable state (which is inconsistent in the sense wrt. $Q = \neg Q$: they should differ) and $S = R = 1$ is the storage state (which retains whatever values Q and $\neg Q$ have already); $S = 0$ and $R = 1$ sets $Q = 1$, while $S = 1$ and $R = 0$ resets $Q = 0$, *whatever* the currently value of Q is. As such, the second excitation table is correct (the first one is for a NOR-based SR-latch), noting this is sort of the *inverse* of a NOR-based SR-latch wrt. the meaning of S and R.

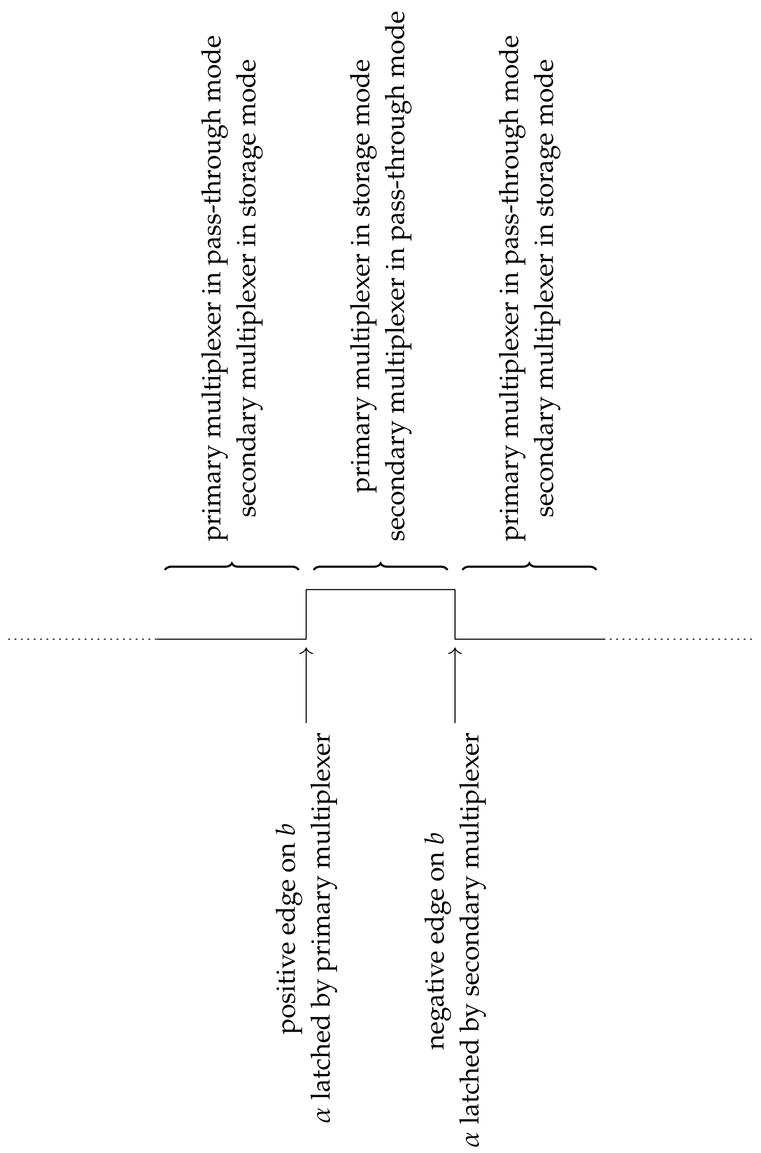

primary multiplexer in pass-through mode
secondary multiplexer in storage mode
primary multiplexer in storage mode
secondary multiplexer in pass-through mode
primary multiplexer in pass-through mode
secondary multiplexer in storage mode
positive edge on b
α latched by primary multiplexer
negative edge on b
α latched by secondary multiplexer

Figure 1: *A time line illustrating behaviour of a multiplexer-based, primary-secondary flip-flop.*

▷ **S46.** This is quite a tricky question, but the central feature to note, for both multiplexers in the circuit, is the loop between output and (an) input:

- in the left-hand case the loop connects the multiplexer output, say t_0, to the y input, whereas
- in the right-hand case the loop connects the multiplexer output, say t_1, to the x input.

In both cases, this allows a 1-bit value to be *stored* by "holding" it in the loop. Put another way, when $b = 0$ then $t_0 = a$ since the x input is selected; when $b = 1$, however, whatever value t_0 has is fed back into the multiplexer. This is conceptually similar to how SRAM cells work, for example. In that case we had a loop through two NOT gates that acted to refresh (or reinforce) the stored value, plus extra access transistors. More formally, in each case the loop results in bistability. Focusing on the left-hand multiplexer as an example, if $b = 1$ then either of the states $t_0 = 0$ or $t_0 = 1$ (meaning $y = 0$ and $y = 1$) is stable (meaning it will not transition into the other state without a stimulus), so the value is retained until updated (or lost if the power supply is removed).

In this case, the left-hand and right-hand multiplexers are organised in a primary-secondary form. The idea is basically that b acts as an enable signal (typically it will be a clock), operating both primary and secondary multiplexer in one of two modes. Per the above,

- when $b = 0$ the primary multiplexer passes a through to t_0, whereas the secondary multiplexer is in storage mode, and
- when $b = 1$ the primary multiplexer is in storage mode, whereas the secondary multiplexer passes t_0 through to t_1

To understand their combined behaviour, focus on the instant in time when a positive *edge* occurs on b and imagine that $a = \alpha$ for a value $\alpha \in \{0, 1\}$. Before the edge, because $b = 0$, the primary and secondary multiplexers will be in pass-through and storage mode respectively. This means $t_0 = a = \alpha$ and $c = t_1$. At the instance the edge occurs on b, $t_0 = \alpha$. Since the primary multiplexer flips into storage mode, this value is retained (i.e., fed back around the loop into the y input) because $b = 1$: changes to a are irrelevant. Simultaneously, the secondary multiplexer flips into pass-through mode st. $c = \alpha$. Then, at some point, there is a negative edge on b meaning both primary and secondary multiplexers flip back to the opposite mode. Given $c = t_1 = \alpha$ at this instant, the fact the secondary multiplexer is now in storage mode (again) means it retains the value of α. Likewise, since the primary multiplexer is in pass-through mode, any change to a is reflected in t_0 (but since the secondary multiplexer is in storage mode, this is irrelevant to the value, namely α, it retains). Diagrammatically, this can be viewed as in Figure 1.

A somewhat reasonable analogy is that the primary and secondary multiplexers act as latches (each being level triggered) in isolation, but as a flip-flop once combined. α can be viewed as being "passed along" a 2-step "conveyor belt". First, at a positive edge on b, the primary multiplexer will stores whatever α is passed as input by the user. Then, at the subsequent negative edge on b, that α is passed on to the secondary multiplexer which stores it; in a sense, the primary multiplexer "protects" the stored α from subsequent changes to a until another positive edge on b occurs.

So the correct answer is that the circuit represents a flip-flop, i.e., an edge triggered storage cell: a is the flip-flop input, c is the flip-flop output (i.e., the stored value), and b is the flip-flop enable signal.

▷ **S47.** Given an l-bit control signal c, the demultiplexer can select between at most 2^l outputs: we treat c as an unsigned, l-bit integer which will clearly range in value between 0 and $2^l - 1$. In general, we want an l st. $2^l \geq m$ so each output can be specified; typically m is a power-of-two, since this matches the maximum number of outputs that can be specified. However, in this case we have $m = 5$.

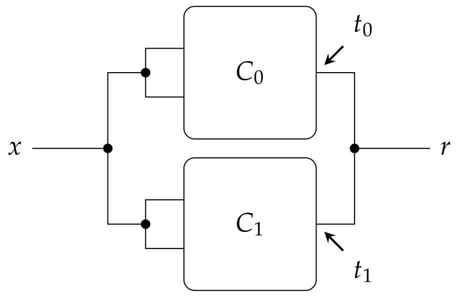
Since $2^2 = 4 < 5$ and $2^3 = 8 > 5$ we know a 2-bit control signal is not enough (it cannot select r_4 since $0 \leq c < 4$), but a 3-bit control signal is (although it *could* cope with upto $m = 8$, and since $0 \leq c < 8$ select r_5, r_6 and r_7 if they existed). In summary then, $l = 3$ is the correct answer.

▷ **S48.** a Note that if a given r_i is not connected to either V_{dd} or V_{ss}, it is deemed to have the high impedance value **Z**. This suggests the correct truth table is

x	y	r_0	r_1
0	0	1	**Z**
0	1	**Z**	**Z**
1	0	**Z**	**Z**
1	1	**Z**	0

The reason is because C_0 is st. r_0 connects to V_{dd} via two (pull-up) P-type MOSFETs; since these MOSFETs only connect source to drain if the gate is V_{ss}, we can say that $r_0 = 1$ if $x = y = 0$ and $r_0 = \mathbf{Z}$ (i.e., disconnected) otherwise. Conversely, C_1 is st. r_1 connects to V_{ss} via two (pull-down) N-type MOSFETs; since these MOSFETs only connect source to drain if the gate is V_{dd}, we can say that $r_1 = 0$ if $x = y = 1$ and $r_1 = \mathbf{Z}$ (i.e., disconnected) otherwise.

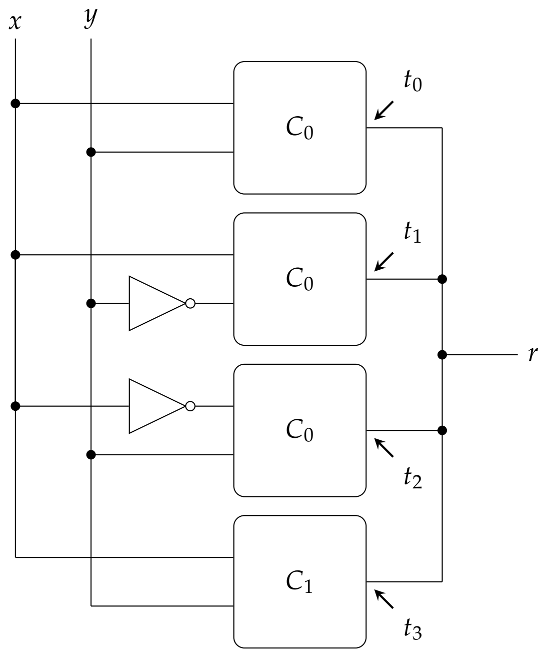
b Note that the option using 1 instance of C_0 and 1 instance of C_1 *sort of* makes sense: one *can* implement a NAND gate using 2 P-type and 2 N-type MOSFETS, matching those that exist within instances of C_0 and C_1. However, the question explicitly says we need to use instances of C_0 and C_1: we cannot, for example, "merge" their internal implementation to make this option viable. So, as a first step, we implement a NOT gate as follows:



This is useful because we can reuse it when implementing a NAND gate, but also because it explains the design approach involved: the idea is basically that the output is driven by *one* instance of C_0 or C_1 at a time, with all the others producing the high impedance value (which is "overridden" by the driving value). The behaviour can be described as follows:

x	t_0	t_1	r
0	1	**Z**	1
1	**Z**	0	0

Using the same design approach, we can now implement an NAND gate as follows:

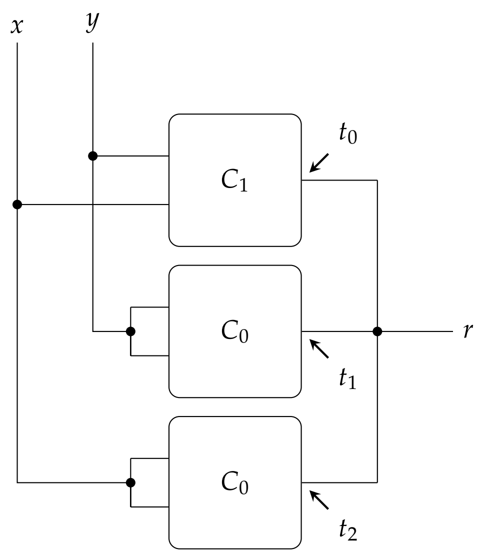


Applying the same argument wrt. behaviour, we find that

x	y	t_0	t_1	t_2	t_3	r
0	0	1	**Z**	**Z**	**Z**	1
0	1	**Z**	1	**Z**	**Z**	1
1	0	**Z**	**Z**	1	**Z**	1
1	1	**Z**	**Z**	**Z**	0	0

matches the truth table for NAND: remembering to count the components within each NOT gate, we therefore use 5 instances of C_0 and 3 instances of C_1.

As an aside, note that one can implement a NOR gate by swapping the components types in the NAND implementation: we therefore implement the required behaviour using 3 instances of C_0 and 5 instances of C_1. Also note that it is tempting to consider something like



as a solution. There is no corresponding option in the table, however: the reason for this is that it violates the stated design strategy. This can be seen by considering the (hypothetical) truth table

x	y	t_0	t_1	t_2	r
0	0	**Z**	1	1	1
0	1	**Z**	**Z**	1	1
1	0	**Z**	1	**Z**	1
1	1	0	**Z**	**Z**	0

which shows the case where $x = 0$ and $y = 0$ means both $t_1 = 1$ and $t_2 = 1$: in such a case r is driven by two non-**Z** values.

▷ **S49.** The first three options are all related to the fact that the number of transistors is increasing; the rate of increase is important in that the associated limits are reached quicker, but is not so relevant beyond that.

- The fact there are more transistors in a fixed unit of area implies the transistors are smaller, and so, as a result, that their feature size (i.e., the size of components in their design, such as channel length or layer thicknesses) is also smaller. Limits clearly exist wrt. how small feature sizes can shrink. Even if manufacturing process keep pace, at some point the feature size would be measured in some small number of atoms: at this scale it is plausible the transistor cannot operate correctly, and beyond it one would need to consider a (radically) different approach.
- Dennard scaling states, roughly, that as transistors become smaller, their power density remains constant. At face value then, power consumption *should* not represent a limit. However, Dennard scaling has now started to break down: with such small feature sizes, otherwise insignificant factors (e.g., static vs. dynamic power consumption) become significant, and thus lead to increased power consumption if the number of transistors is increased. An indefinite increase in the amount of power supplied is not plausible, meaning it acts to limit the number of transistors one can house per unit of area.
- With a small enough feature size, the channel allowing electrical current to flow through the transistor will not always be able to "contain" it, i.e., there is leakage current. This manifests itself as heat, which must be dissipated away from the transistors to ensure their correct operation. So, with a fixed capacity to dissipate heat, this will act to limit the number of transistors one can house per unit of area.

In summary then, *all* the first three options plausibly constrain or limit Moore's Law.

▷ **S50.** A logical approach to this question would likely use two steps: 1) we need to assess which option(s) will toggle s, then 2) find the shortest path from the inputs to s, which is sort of the *opposite* of the critical path, and use this to decide the correct option.

Note that the truth table for this component (including the various annotated intermediate variables) is as follows:

ci	x	y	t_0	t_1	t_2	co	s
0	0	0	0	0	0	0	0
0	0	1	0	1	1	0	1
0	1	0	0	1	1	0	1
0	1	1	0	1	0	1	0
1	0	0	0	1	1	0	1
1	0	1	0	1	0	1	0
1	1	0	0	1	0	1	0
1	1	1	1	1	0	1	1

So, as a first step, we can see that

$$\begin{array}{lclllclclll}
(0,0,0) & \Rightarrow & t_0 = 0 & t_2 = 0 & s = 0 & \rightarrow & (0,0,1) & \Rightarrow & t'_0 = 0 & t'_2 = 1 & s' = 1 \\
(0,0,1) & \Rightarrow & t_0 = 0 & t_2 = 1 & s = 1 & \rightarrow & (0,1,1) & \Rightarrow & t'_0 = 0 & t'_2 = 0 & s' = 0 \\
(0,1,1) & \Rightarrow & t_0 = 0 & t_2 = 0 & s = 0 & \rightarrow & (0,0,1) & \Rightarrow & t'_0 = 0 & t'_2 = 1 & s' = 1 \\
(1,1,1) & \Rightarrow & t_0 = 1 & t_2 = 0 & s = 1 & \rightarrow & (1,1,0) & \Rightarrow & t'_0 = 0 & t'_2 = 0 & s' = 0 \\
(1,0,1) & \Rightarrow & t_0 = 0 & t_2 = 0 & s = 0 & \rightarrow & (0,1,1) & \Rightarrow & t'_0 = 0 & t'_2 = 0 & s' = 0
\end{array}$$

i.e., all options bar the last one will toggle s (either from 0 to 1, or from 1 to 0). Next, imagine T_{NOT}, T_{AND}, and T_{OR} denote the gate delay of a NOT gate, and 2-input AND and OR gate respectively. We can consider five paths from the inputs to s, which each pass through one of the gates organised in a column on the left-hand side of the diagram.

$$\begin{array}{cll}
\text{top} & \text{3-input AND} & \rightsquigarrow & 2 \cdot T_{AND} + 1 \cdot T_{OR} \\
 & \text{3-input OR} & \rightsquigarrow & 1 \cdot T_{AND} + 3 \cdot T_{OR} \\
\updownarrow & \text{2-input AND} & \rightsquigarrow & 2 \cdot T_{AND} + 3 \cdot T_{OR} + 1 \cdot T_{NOT} \\
 & \text{2-input AND} & \rightsquigarrow & 2 \cdot T_{AND} + 3 \cdot T_{OR} + 1 \cdot T_{NOT} \\
\text{bottom} & \text{2-input AND} & \rightsquigarrow & 2 \cdot T_{AND} + 3 \cdot T_{OR} + 1 \cdot T_{NOT}
\end{array}$$

Given it is likely $T_{AND} \simeq T_{OR}$, it seems clear the top path will be the shortest. Put another way, given $s = t_0 \vee t_2$, we can toggle s by controlling t_0 or t_2; having identified the top path as the shortest, controlling t_0 will allow control over s within the shortest period of time. Of the options, the second to last one, i.e., $(1,1,1) \longrightarrow (1,1,0)$, is the only one that toggles t_0, and would therefore be deemed correct.

▷ **S51.** Since this is a cyclic counter, we know selecting $n = 4$ means the output r will step through values

$$0, 1, \ldots, 2^4 - 1 = 15, 0, 1, \ldots.$$

Writing this information in a tabular form as follows

r	r_3	r_2	r_1	r_0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1
0	0	0	0	0
1	0	0	0	1
⋮	⋮	⋮	⋮	⋮

highlights the behaviour of each i-th bit r_i as r itself changes.

The question asks which r_i transition between 0 and 1 at the lowest frequency, i.e., slowest. Obviously it cannot be r_4: for $n = 4$ there is no r_4! By looking a the table above, the solution is also obvious: the MSB r_3 transition between 0 and 1 slowest, doing so every 8 updates to r (vs. r_0, for example, which does do every 1 update).

▷ **S52.** CMOS combines *complementary* transistor types: one N-MOSFET, and one P-MOSFET. Since these transistor behave in a complementary manner, it is always that case that one will be active and one inactive. This produces an attractive feature wrt. power, in that static consumption (i.e., when there is no chance in state) is very low. The dynamic power consumption (i.e., when there is a change of state) is of course higher, but occurs only when the inputs cause switching activity.

So, at a high-level at least, we *could* argue the highest consumption is likely when the initial value differs the most from stored value. That is, we argue that the highest switching activity occurs when the largest number of bits stored by the register change. We can measure this using the Hamming distance between initial and stored value: given

$$\begin{array}{lclcl}
t_0 & = & DEAD_{(16)} & = & 1101111010101101_{(2)} \\
t_1 & = & BEEF_{(16)} & = & 1011111011101111_{(2)} \\
t_2 & = & F00D_{(16)} & = & 1111000000001101_{(2)} \\
t_3 & = & 1234_{(16)} & = & 0001001000110100_{(2)} \\
t_4 & = & FFFF_{(16)} & = & 1111111111111111_{(2)} \\
t_5 & = & 0000_{(16)} & = & 0000000000000000_{(2)}
\end{array}$$

and recalling that

$$\mathrm{HD}(x, y) = \sum_{i=0}^{i<16} x_i \oplus y_i,$$

we can compute

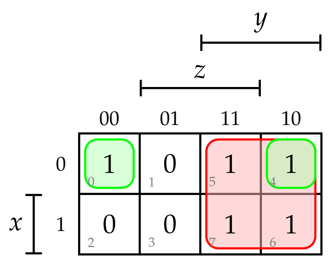
$$\begin{array}{lcr}
\mathrm{HD}(t_0, t_1) & = & 4 \\
\mathrm{HD}(t_0, t_2) & = & 6 \\
\mathrm{HD}(t_0, t_3) & = & 8 \\
\mathrm{HD}(t_0, t_4) & = & 5 \\
\mathrm{HD}(t_0, t_5) & = & 11
\end{array}$$

This means that based on our argument above, the correct answer would be $0000_{(16)}$. A more precise answer requires a much more detailed analysis of the component: we would need to assess the implementation in terms of individual transistors, and compute the so-called toggle count, i.e., switching activity, at that level (rather than via the assumption about toggling the bits stored).

▷ **S53.** Using a component set with some number of AND, OR, and NOT gates is clearly a more familiar approach. However, *all* the component sets can be used to implement f. Writing out the truth table

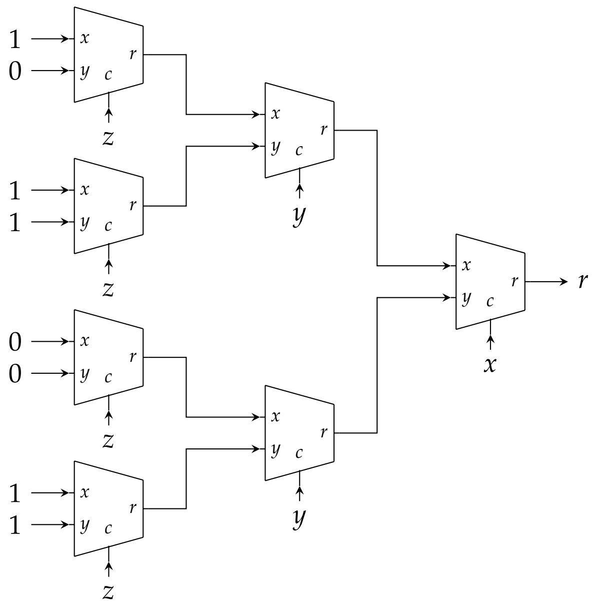
x	y	z	r
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

and a Karnaugh map

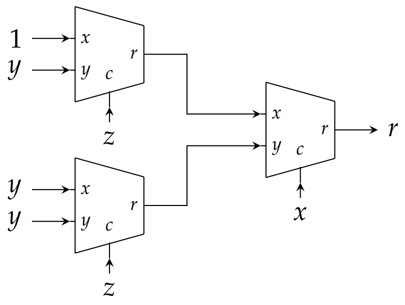


for f helps explain how and why:

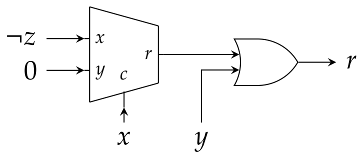
a Using 2-input, 1-bit multiplexers for clarity, component set 1 can be used to implement f as follows:



b Using 2-input, 1-bit multiplexers for clarity, component set 2 can be used to implement f as follows:



c Component set 3 can be used to implement f as follows:



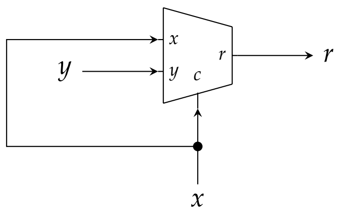
▷ **S54.** First, recall that the following truth table

c	x	y	r
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

specifies the behaviour of a 2-input, 1-bit multiplexer: in short, we find that

$$r = \begin{cases} x & \text{if } c = 0 \\ y & \text{if } c = 1 \end{cases} .$$

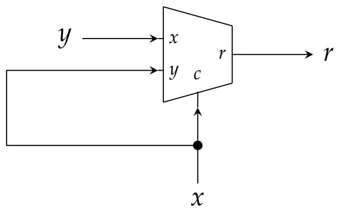
As such, we can implement an AND gate as follows:



We can show *why* the implementation is valid (i.e., produces a result matching AND) by inspection:

x	y	r
0	0	$x = 0$
0	1	$x = 0$
1	0	$y = 0$
1	1	$y = 1$

Notice that $x = 0$ implies the multiplexer selects the top input and hence $r = x$, whereas $x = 1$ implies the multiplexer selects the top input and hence $r = y$; overall, r clearly matches AND in the sense $r = 1$ if $x = 1$ and $y = 1$. Using the same approach, we can implement OR as follows



and justify validity again by inspection:

x	y	r
0	0	$y = 0$
0	1	$y = 1$
1	0	$x = 1$
1	1	$x = 1$

Overall then, the expression

$$(x \wedge y) \vee z$$

can be implemented using just two multiplexers: one to implement the AND operator, and one to implement the OR operator.

▷ **S55.** This question can be approached in several ways. First, one could employ basic pattern matching: read from left-to-write, the three dominant structures can be matched against known NAND, NOR, and NOT gate implementations. As such, the design implements the expression

$$r = \neg((x \overline{\wedge} y) \overline{\vee} z)$$

which we manipulate as follows

$$\begin{array}{lll} & \neg((x \overline{\wedge} y) \overline{\vee} z) & \\ = & \neg((\neg(x \wedge y)) \overline{\vee} z) & (NAND) \\ = & \neg(\neg((\neg(x \wedge y)) \vee z)) & (NOR) \\ = & \neg(x \wedge y \wedge \neg z) & (deMorgan) \end{array}$$

such that

$$r = \neg(x \wedge y \wedge \neg z).$$

Second, although it involves more work, one can enumerate the transistor and signal states for each input combination. For example, using + (resp. −) to denote where a given transistor is connected or activated (resp.

disconnected or deactivated), we can write

x	y	z	m_0	m_1	m_2	m_3	t_0	m_4	m_5	m_6	m_7	t_1	m_8	m_9	r
0	0	0	+	+	−	−	1	−	+	+	−	0	+	−	1
0	0	1	+	+	−	−	1	−	−	+	+	0	+	−	1
0	1	0	−	+	+	−	1	−	+	+	−	0	+	−	1
0	1	1	−	+	+	−	1	−	−	+	+	0	+	−	1
1	0	0	+	−	−	+	1	−	+	+	−	0	+	−	1
1	0	1	+	−	−	+	1	−	−	+	+	0	+	−	1
1	1	0	−	−	+	+	0	+	+	−	−	1	−	+	0
1	1	1	−	−	+	+	0	+	−	−	+	0	+	−	1

and then derive the expression

$$r = \neg(x \wedge y \wedge \neg z)$$

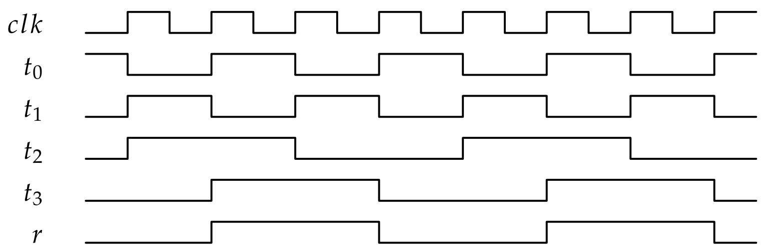
directly.

▷ **S56.** To start with, keep in mind that this design uses flip-flops: these are edge-triggered (versus latches, which are level-triggered). By focusing on and inspecting the left-hand flip-flop, we infer that the state will be updated to reflect $D = 1 \oplus Q$ on each positive edge of clk. Given the truth table

x	y	r
0	0	0
0	1	1
1	0	1
1	1	0

we find that $D = 1 \oplus Q \equiv \neg Q$, suggesting, therefore, that this is a toggle flip-flop constructed by using a D-type flip-flop: on each positive edge of clk, the state will toggle either from 0 to 1 or from 1 to 0. Note that the right-hand flip-flop has a similar construction, but that the lower input of the XOR comes from the left-hand flip-flop.

Imagine that both flip-flops are reset, so their initial state is 0. We can draw a waveform which describes each signal:



Put simply, this suggests that each toggle flip-flop acts to halve the frequency: t_1 toggles at half the frequency of clk and t_3 toggles at quarter the frequency of clk. Given that clk has a frequency of 400MHz, we therefore expect $r = t_3$ to toggle with a frequency of $\frac{400}{4} = 100$MHz.

▷ **S57.** We know that the multiplexer will select the top input if $c = 0$, or the bottom input if $c = 1$. This means the design can be expressed as

$$r = (\neg q \wedge \neg p) \vee (q \wedge p),$$

i.e., if $q = 0$ then $r = \neg p$, whereas if $q = 1$ then $r = p$. As such, we can produce a truth table

p	q	r
0	0	1
0	1	0
1	0	0
1	1	1

from which it is clear(er) that $r = \neg(p \oplus q)$.

▷ **S58.** If the current input is $S = R = 0$, then the latch must be in the invalid state $Q = 1$, $\neg Q = 1$. The question is focused on instantaneously setting $S = R = 1$, which means the latch is in storage mode: the question is, what state does it end up in? The answer is that it does not remain in the invalid state, due to the imbalanced gate delays. Let T_t and T_b denote the top and bottom gate delay respectively:

a If $T_t = x > x - \delta = T_b$, the output of the bottom gate, i.e., Q, will change state first: it changes from 1 to 0, at which point the only valid (eventual) output from the top gate is 1 (i.e., it will stay the same). So, the bottom gate "winning" by changing first is like we reset the latch via the bottom, R input.

b If $T_t = x < x + \delta = T_b$, the output of the top gate, i.e., $\neg Q$, will change state first: it changes from 1 to 0, at which point the only valid (eventual) output from the bottom gate is 1 (i.e., it will stay the same). So, the top gate "winning" by changing first is like we set the latch via the top, S input.

This means the latch outputs will be either $Q = 0$, $\neg Q = 1$ or $Q = 1$, $\neg Q = 0$.

▷ **S59.** Since the question is ultimately about Boolean expressions, we use 0 and 1 in place of V_{ss} and V_{dd} for convenience. Recall that an N-type MOSFET is is connected or activated (resp. disconnected or deactivated) if the gate terminal is 1 (resp. 0), whereas an P-type MOSFET is is connected or activated (resp. disconnected or deactivated) if the gate terminal is 0 (resp. 1).

By inspection, t_0 is clearly connected to 1 when either 1) $x = 0$ and $y = 0$ (through m_0 and m_1), or 2) $z = 0$ (through and m_2). Note that the connectives "and" and "or" used here reflect the sequential and parallel way the P-type MOSFETS are organised. But, either way, and assuming a matching pull-down network, we can write

$$t_0 = (\neg x \wedge \neg y) \vee \neg z.$$

The output r is then produced via m_3 and m_4, which form a NOT gate: this means

$$r = \neg((\neg x \wedge \neg y) \vee \neg z)$$

which, using the de Morgan axiom, we can rewrite as

$$r = (x \vee y) \wedge z.$$

▷ **S60.** This is an N-type MOSFET, used here as an enable gate: if $en = 0$ there is no connection between x and r, but if $en = 1$ there is a connection between x and r. The former case is the more interesting, in the sense that r is disconnected from any driving signal. This situation is modelled using 3-state logic, wherein an additional high impedance value **Z** is considered. Using **Z**, we can therefore model the transistor using this truth-table:

x	en	r
0	0	**Z**
1	0	**Z**
0	1	0
1	1	1

Put simply, if $en = 0$ then $r = \mathbf{Z}$ because there is no driving signal, but if $en = 1$ then $r = x \in \{0, 1\}$. Based on the potential values of x and en we conclude that $r \in \{0, 1, \mathbf{Z}\}$, i.e., it can potentially take 3 different values.

▷ **S61.** The simplest approach to producing a solution for this question is brute-force enumeration: by just inspecting the truth-table, i.e.,

x_1	x_0	y_1	y_0	r		
0	0	0	0	0	$\Rightarrow$	$0 \not> 0$
0	0	0	1	0	$\Rightarrow$	$0 \not> 1$
0	0	1	0	0	$\Rightarrow$	$0 \not> 2$
0	0	1	1	0	$\Rightarrow$	$0 \not> 3$
0	1	0	0	1	$\Rightarrow$	$1 > 0$
0	1	0	1	0	$\Rightarrow$	$1 \not> 1$
0	1	1	0	0	$\Rightarrow$	$1 \not> 2$
0	1	1	1	0	$\Rightarrow$	$1 \not> 3$
1	0	0	0	1	$\Rightarrow$	$2 > 0$
1	0	0	1	1	$\Rightarrow$	$2 > 1$
1	0	1	0	0	$\Rightarrow$	$2 \not> 2$
1	0	1	1	0	$\Rightarrow$	$2 \not> 3$
1	1	0	0	1	$\Rightarrow$	$3 > 0$
1	1	0	1	1	$\Rightarrow$	$3 > 1$
1	1	1	0	1	$\Rightarrow$	$3 > 2$
1	1	1	1	0	$\Rightarrow$	$3 \not> 3$

we can see that $r = 1$ in 6 cases.

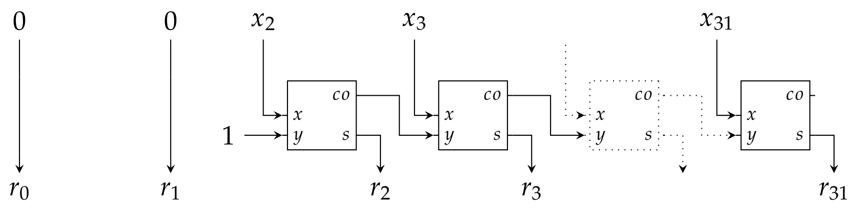
▷ **S62.** Several facts inform the optimisation one would expect.

- The adder inputs are PC and 4: whereas PC can be any 32-bit value, so the input is general-purpose, 4 is a fixed, special-purpose input.
- Since PC is word-aligned, we know that

$$\mathsf{PC} \equiv \mathsf{PC} + 4 \equiv 0 \pmod 4.$$

  As a result, the 2 LSBs of PC + 4 always equal 0: there is no need to compute them.
- For a general-purpose adder, the carry-in input and carry-out output are useful in various situations. Here, however, neither is useful and so they remain unused; as a result, we can optimise the adder by eliminating them, e.g., simplifying the associated full- or half-adder cell.

Rather than a general-purpose ripple-carry adder, which uses 32 full-adder cells, we can use the facts above to compute the same result by using 30 half-adder cells. That is, the adder design is of the form



where the half-adder that generates r_{31} is optimised even further by considering that the carry-out is unused.

Under the same assumptions, the resulting area is

$$\begin{array}{lllll} & 29 \cdot (1 \cdot \mathrm{XOR} + 1 \cdot \mathrm{AND}) & + & 1 \cdot (1 \cdot \mathrm{XOR}) \\ = & 29 \cdot (1 \cdot 4 + 1 \cdot 2) & + & 1 \cdot (1 \cdot 4) \\ = & 29 \cdot 6 & + & 1 \cdot 4 \\ = & 178 \end{array}$$

which is a factor of $448/178 = 2.52$ improvement.

▷ **S63.** Saying that the cell contains “duplicate” makes no sense, and it is not true that we do not *know* what the cell content, or output, is. Rather, we do not *care* what the cell content is: the input is impossible, so the output is irrelevant to f.

Of the two remaining options, the correct one mirrors the associated approach. That is, the don’t care cell can be treated as either 0 or 1; we select the option which will most effectively simplify the resulting term. Since the input is impossible, that selection has no impact on the functionality of f for the possible inputs.

▷ **S64.** Just by inspection, there is no way we can use either XOR, AND, or OR gate types to achieve the required functionality. So if NAND and NOR are the only viable options, the question is asking which of them has been used. The definition of NAND or NOR is as follows:

x	y	$x \overline{\wedge} y$	$x \overline{\vee} y$
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

This allows us to reason about the relationship between inputs (i.e., S and R) and outputs (i.e., Q and $\neg Q$). For example, if the current state is $Q = 0$, $\neg Q = 1$ and we have $S = 1$ and $R = 1$ then the top component is computing $0 = S \odot \neg Q = 1 \odot 1$, and the bottom component is computing $1 = R \odot Q = 1 \odot 0$; in this case the top component is consistent with either NAND or NOR, but the bottom component is consistent with only NAND. So, although we could test other cases to gain more confidence, this case alone acts as a strong enough hint that $\odot \equiv \overline{\wedge}$, i.e., NAND gates have been used. But inspecting some other cases based on this option confirms it is correct. For example, if $R = 0$ then we must have $\neg Q = 0 \overline{\wedge} Q = 1$ irrespective of Q. Then, if $S = 1$, we have $Q = S \overline{\wedge} \neg Q = 1 \overline{\wedge} 1 = 0$; this matches row 3 of the truth table. Likewise, If $S = 0$ then we must have $Q = 0 \overline{\wedge} \neg Q = 1$ irrespective of $\neg Q$. Then, if $R = 1$, we have $\neg Q = S \overline{\wedge} Q = 1 \overline{\wedge} 1 = 0$; this matches row 4 of the truth table. Note, therefore, that the semantics of set and reset for this NAND-based latch are sort of “swapped” versus the NOR-based alternative.

▷ **S65.** Note that the additional inputs are often termed preset and clear, matching the labels used here.

a The output of a synchronous circuit depends on the input in a discrete manner, i.e., it can change only at a specific point in time; this is achieved using a clock signal, which acts to control when components in the circuit, e.g., by gating them. In contrast, the output of an asynchronous circuit depends on the input in a continuous manner, i.e., it can change at any time; there is no control of when changes in the output take effect. The same terminology can be applied to the inputs themselves, or even signals more generally. S and R are most accurately classified as synchronous. The reason is that they can influence the output only when $en = 1$: we can see this by noting $en \overline{\wedge} x = 1$ if $en = 0$, whereas $en \overline{\wedge} x = \neg x$ if $en = 1$. As such, en can be said to gate S and R (and hence control their influence). P and C are most accurately classified as asynchronous. The reason is that they can influence the output at *any* point in time, whether $en = 1$ or $en = 0$ and so irrespective of en: in a sense this is obvious, because they do not interact with en (they feed directly into the two NAND gates towards the right-hand side).

b When used to describe a control signal x, the terms active low and active high relate to *when* that signal exerts control, i.e., whether that is when $x = 0$ or $x = 1$; in the former case, this is often highlighted by writing $\neg x$ or $\overline{x}$ rather than x as the input label. S and R are most accurately classified as active high: we can show that if $en = 1$ then $S = 1$ will set $Q = 1$, whereas if $en = 0$ then $R = 1$ will reset $Q = 0$. P and C are most accurately classified as active low: we can show that $P = 0$ will override S and set $Q = 1$, whereas $C = 0$ will override R and reset $Q = 0$.

▷ **S66.** a A multiplexer can be viewed as

$$r = f(c, x_0, x_1, x_2, x_3) = \begin{cases} x_0 & \text{if } c = 0 \\ x_1 & \text{if } c = 1 \\ x_2 & \text{if } c = 2 \\ x_3 & \text{if } c = 3 \end{cases}$$

i.e., the control signal c selects an x_i to produce r. Matching this with the block presented by the question, clearly x_i *and* c are the input wires whereas r is the output wires; as such, $n = (4 \cdot 4) + \log_2 4 = 16 + 2 = 18$ and $m = (1 \cdot 4) = 4$.

b For a half-adder $n = 2$, $m = 2$: there are 2 inputs, namely x and y, and 2 outputs, namely s (the sum) and co (the carry-out). For a full-adder $n = 3$, $m = 2$: there are 3 inputs, namely x, y, and ci (the carry-in), and 2 outputs, namely s (the sum) and co (the carry-out).

▷ **S67.** a **true**. Ignoring the fact that the gate terminal is an input to (i.e., exerts control over) the N- and P-type MOSFETs, a "bubble" on the latter is intended to show it is an "inverting version" of the former. Put more directly, the behaviour of an N-type MOSFET can be described as

$$\begin{array}{lcll} g = V_{dd} & = 5V & \Rightarrow & \text{transistor is activated, } s \text{ and } d \text{ connected} \\ g = V_{ss} \simeq GND & = 0V & \Rightarrow & \text{transistor is deactivated, } s \text{ and } d \text{ disconnected} \end{array}$$

whereas the behaviour of an P-type MOSFET can be described as

$$\begin{array}{lcll} g = V_{ss} \simeq GND & = 0V & \Rightarrow & \text{transistor is activated, } s \text{ and } d \text{ connected} \\ g = V_{dd} & = 5V & \Rightarrow & \text{transistor is deactivated, } s \text{ and } d \text{ disconnected} \end{array}$$

i.e., the semantics of g are the inverse (or opposite).

b **false**. Given that c, x, and y are all connected to transistor gate terminals, they represent inputs; r is not, so is therefore an output. So, as a first step or a sanity check, this does indeed match the expression. Therefore, using $V_{dd} \equiv 1$ and $V_{dd} \equiv 0$, we can describe the behaviour exhibited by considering two cases:

i If $c = 0$ the either the top-left path from V_{dd} or bottom-left path from V_{ss} is possible, depending on x: so if $c = 0$ and $x = 0$ then $r = 1$ (via the top-left path, through bottom P-MOSFET because $c = 0$ and top P-MOSFET because $x = 0$) whereas if $c = 0$ and $x = 1$ then $r = 0$ (via the bottom-left path, through top N-MOSFET because $\neg c = 1$ and bottom N-MOSFET because $x = 1$).

ii If $c = 1$ the either the top-right path from V_{dd} or bottom-right path from V_{ss} is possible, depending on y: so if $c = 1$ and $y = 0$ then $r = 1$ (via the top-left path, through bottom P-MOSFET because $\neg c = 0$ and top P-MOSFET because $y = 0$) whereas if $c = 1$ and $y = 1$ then $r = 0$ (via the bottom-right path, through top N-MOSFET because $c = 1$ and bottom N-MOSFET because $y = 1$).

In summary, this means

c	x	y	r
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

and therefore $r = \neg((x \wedge \neg c) \vee (y \wedge c))$ meaning it implements a "multiplexer with inverted output".

c **true**. A combinatorial logic component *continuously* computes the output(s) from the input(s), where the former is a function of the latter alone: there is no internal state which contributes to the output(s), for example. For a component $f : \{0,1\}^n \rightarrow \{0,1\}^m$, we can define $f_i : \{0,1\}^n \rightarrow \{0,1\}$ for $0 \leq i < n$. Each f_i takes all n original inputs, but outputs 1 (i.e., the i-th) of the m original outputs. By concatenating those outputs together, we have that

$$f(x) \equiv f_0(x) \parallel f_1(x) \parallel \cdots \parallel f_{m-1}(x)$$

due to their behaviour as combinatorial logic; as such, we have decomposed f into m separate f_i.

d **false**. To apply a Karnaugh map to this function, we would first decompose it into

$$f_i : \{0,1\}^n \rightarrow \{0,1\}$$

for $0 \leq i < m$, i.e., m separate functions which each produce a 1-bit output. Then, a separate Karnaugh map can be applied to each f_i to yield an associated Boolean expression. Doing so will yield an optimised implementation up to a point, but there may be potential for *further* manual manipulation and optimisation, e.g., via 1) common sub-expression elimination within the expression for one f_i and 2) common sub-expression elimination within the expressions for multiple f_i, i.e., within the expression for f as a whole. This potential means one cannot say simply applying the Karnaugh map will *always* yield the optimal implementation.

e **true**. Recall that the truth table for a full-adder (i.e., 1-bit adder) reads as follows

ci	x	y	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

This captures the fact that it computes two 1-bit outputs (i.e., the carry-out co and sum s) from three 1-bit inputs (i.e., the carry-in ci and addends x and y). Imagine that $+(ci,x,y)_{co}$ and $+(ci,x,y)_s$ respectfully denote the co and s outputs of a full-adder, as applied to the inputs ci, x and y: one can see that

$$\begin{array}{rcl} x \wedge y & \equiv & +(0,x,y)_{co} \\ x \vee y & \equiv & +(1,x,y)_{co} \\ \neg\, y & \equiv & +(0,1,y)_s \end{array}$$

which implies said the full-adder is functionally complete in the sense required.

f **true**. There are (at least) 2 plausible implementations to consider, both of which satisfy the condition related to number of logic gates:

i Implement a 4-input, 1-bit multiplexer directly: the following expression captures the result

$$r = (w \wedge \neg c_1 \wedge \neg c_0) \vee (x \wedge \neg c_1 \wedge c_0) \vee (y \wedge c_1 \wedge \neg c_0) \vee (z \wedge c_1 \wedge c_0)$$

which involves 15 logic gates, or 13 if the duplicate terms $\neg c_1$ and $\neg c_0$ are factored out and so shared.

ii Implement a 2-input, 1-bit multiplexer then use a cascade of 3 dependant instances of it. the following expression captures the result

$$\begin{array}{rcl} t_0 & = & (w \wedge \neg c_0) \vee (x \wedge c_0) \\ t_1 & = & (y \wedge \neg c_0) \vee (z \wedge c_0) \\ r & = & (t_0 \wedge \neg c_1) \vee (t_1 \wedge c_1) \end{array}$$

which totals 12 logic gates.

▷ **S68.** Using a Karnaugh map, for example, one can produce the result

$$r = f(x,y,z) = y \vee (z \wedge \neg x) \vee (x \wedge \neg z)$$

which, by inspection, gives

			f			
x	y	z	y	$z \wedge \neg x$	$x \wedge \neg z$	r
0	0	0	0	0	0	0
0	0	1	0	1	0	1
0	1	0	1	0	0	1
0	1	1	1	1	0	1
1	0	0	0	0	1	1
1	0	1	0	0	0	0
1	1	0	1	0	1	1
1	1	1	1	0	0	1

However, notice that the sub-expression

$$(z \wedge \neg x) \vee (x \wedge \neg z)$$

can be simplified to

$$z \oplus x$$

so per the question, the simplest implementation of f is

$$r = f(x,y,z) = y \vee (z \oplus x).$$

▷ **S69.** First, note that the following identities can be applied:

$$\begin{array}{lcl} \neg x & \equiv & x \overline{\wedge} x \\ x \vee y & \equiv & (x \overline{\wedge} x) \overline{\wedge} (y \overline{\wedge} y) \\ x \wedge y & \equiv & (x \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} y) \end{array}$$

As such, we can write

$$\neg(x \vee y) \equiv ((x \overline{\wedge} x) \overline{\wedge} (y \overline{\wedge} y)) \overline{\wedge} ((x \overline{\wedge} x) \overline{\wedge} (y \overline{\wedge} y)).$$

▷ **S70.** The excitation table of a standard SR latch is

		Current		*Next*	
S	R	Q	$\neg Q$	Q'	$\neg Q'$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	**?**	**?**	0	1
1	0	**?**	**?**	1	0
1	1	**?**	**?**	**?**	**?**

meaning the reset-dominate alteration gives

		Current		*Next*	
S	R	Q	$\neg Q$	Q'	$\neg Q'$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	**?**	**?**	0	1
1	0	**?**	**?**	1	0
1	1	**?**	**?**	0	1

Given the inputs S and R, the circuit can be constructed by using a standard SR latch with two cross-coupled NOR gates whose inputs are S' and R'. We simply set $R' = R$ and $S' = \neg R \wedge S$ so S' is only 1 when $R = 0$ and $S = 1$: if $R = 1$ (including the case when both $R = 1$ and $S = 1$), the latch it reset since $S = 0$. The circuit is as follows:

```
-------+
       |  +----+
+----+ +->|    |         +----+
|    |    |AND|-- S'  -->|    |
|NOR|---->|    |         |NOR|-- ~Q --+
|    |    +----+      +->|    |        |
+----+                |  +----+        |
                   +--|----------------+
                   |  |
                   |  +----------------+
                   |     +----+        |
                   +---->|    |        |
                         |NOR|--  Q --+
------------------ R'  -->|    |
                         +----+
```

▷ **S71.** A wide range of answers are clearly possible. Obvious examples include physical size, and power consumption or heat dissipation. Other variants include worst-case versus average-case versions of each metric, for example in the case of efficiency.

▷ **S72.** a MOSFET transistors work by sandwiching together N-type and P-type semiconductor layers. The different types of layer are doped with different substances to create more holes or more electrons. For example, in an N-type MOSFET the layers are constructed as follows

```
               gate
             +--------+
             | metal |
==== source  =========  drain  ==== silicon oxide layer
+--+--------+---------+--------+--+
|  | N-type |         | N-type |  |
|  +--------+         +--------+  |
|             P-type              |
+---------------------------------+
```

with additional layers of silicon oxide and metal. There are three terminals on the transistor. Roughly speaking, applying a voltage to the gate creates a channel between the source and drain through which charge can flow. Thus the device acts like a switch: when the gate voltage is high, there is a flow of charge but when it is low there is little flow of charge. A P-type MOSFET swaps the roles of N-type and P-type semiconductor and hence implements the opposite switching behaviour.

b One can construct an NAND to compute $z = x \overline{\wedge} y$ gate from such transistors as follows:

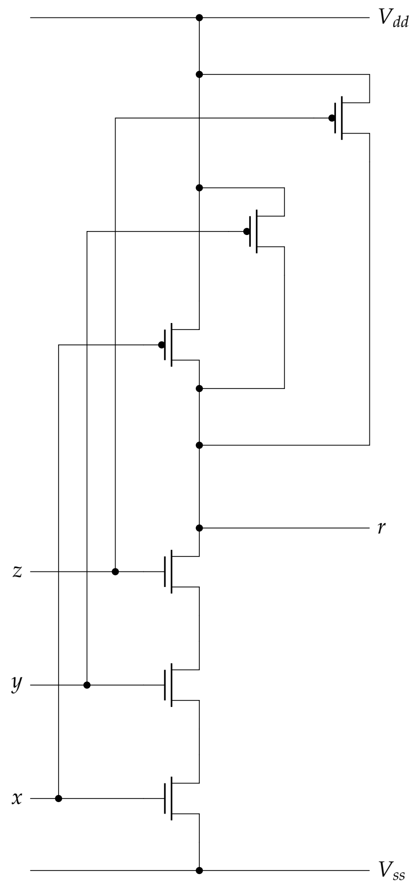
```
                 V_dd
                  |
          +-------+-------+
          |               |
          v               v
     +---------+     +---------+
x -->| P-type | y -->| P-type |
     +---------+     +---------+
          |               |
          +---------------+---> z
          |
     +---------+
x -->| N-type |
     +---------+
          |
     +---------+
y -->| N-type |
     +---------+
          ^
          |
          +-------+
                  |
                 VSS
```

If x and y are connected to V_{ss} then both top P-type transistors will be connected, and both bottom N-type transistors will be disconnected; r will be connected to V_{dd}. If x and y are connected to V_{dd} and V_{ss} respectively then the right-most P-type transistor will be connected, and both lower-most N-type transistor will be disconnected; r will be connected to V_{dd}. If x and y are connected to V_{ss} and V_{dd} respectively then the left-most P-type transistor will be connected, and both upper-most N-type transistor will be disconnected; r will be connected to V_{dd}. If x and y are connected to V_{dd} then both top P-type transistors will be disconnected, and both bottom N-type transistors will be connected; r will be connected to V_{ss}. In short, the behaviour we get is described by

x	y	r
V_{ss}	V_{ss}	V_{dd}
V_{ss}	V_{dd}	V_{dd}
V_{dd}	V_{ss}	V_{dd}
V_{dd}	V_{dd}	V_{ss}

which, if we substitute 0 and 1 for V_{ss} and V_{dd}, is matches that of the NAND operation.

▷ **S73.** This question is a lot easier than it sounds; basically we just add two extra transistors (one P-MOSFET and one N-MOSFET) to implement a similar high-level approach. That is, we want r connected to V_{ss} only when each of x, y *and* z are connected to V_{dd}; this means the bottom, N-MOSFETs are in series. If *any* of x, y or z are connected to V_{ss}, we want r connected to V_{dd}; this means the top, P-MOSFETs are in parallel. Diagrammatically, the result is as follows:



▷ **S74.** This is quite an open-ended question, but basically it asks for high-level explanations only. As such, some example answers include the following:

a CMOS transistors are constructed from atomic-level understanding and manipulation; the immutable size of atoms therefore acts as a fundamental limit on the size of any CMOS-based transistor.

b Feature scaling improves the operational efficiency of transistors, simply because smaller features reduce delay. Beyond this however, one must utilise the extra transistors to achieve some useful task if computational efficiency is to scale as well: improvements to an architecture or design are often required, for instance, to exploit parallelism and so on.

c Even assuming the transistors available can be harnessed to improve computational efficiency, this has implications: more transistors within a fixed size area will increase power consumption and also heat dissipation for example, both of which act as limits even if managed (e.g., via aggressive forms of cooling).

d On one hand, smaller transistors mean less cost per-transistor: with a fixed number of transistors, their area and manufacturing cost will decrease. With a fixed sized area and hence more transistors in it however, this probably means increase defect rate during manufacture. The resulting cost implication could act as an economic limit to transistor size.

▷ **S75.** a The most basic interpretation (i.e., not really doing any grouping using Karnaugh maps but just picking out each cell with a 1 in it) generates the following SoP equations

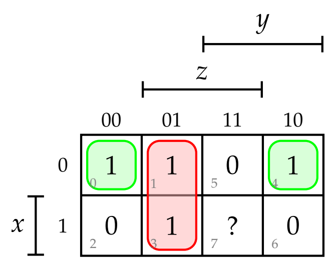
$$
\begin{array}{rcl}
e & = & (\neg a \wedge \neg b \wedge c \wedge \neg d) \vee (a \wedge \neg b \wedge \neg c \wedge \neg d) \vee (\neg a \wedge b \wedge \neg c \wedge d) \\
f & = & (\neg a \wedge \neg b \wedge \neg c \wedge d) \vee (\neg a \wedge b \wedge \neg c \wedge \neg d) \vee (a \wedge \neg b \wedge c \wedge \neg d)
\end{array}
$$

b From the basic SoP equations, we can use the don't care states to eliminate some of the terms to get

$$
\begin{array}{rcl}
e & = & (\neg a \wedge \neg b \wedge c) \vee (a \wedge \neg c \wedge \neg d) \vee (b \wedge d) \\
f & = & (\neg a \wedge \neg b \wedge d) \vee (b \wedge \neg c \wedge \neg d) \vee (a \wedge c)
\end{array}
$$

then, we can share both the terms $\neg a \wedge \neg b$ and $\neg c \wedge \neg d$ since they occur in e and f.

▷ **S76.** Simply transcribing the truth table into a suitable Karnaugh map gives



from which we can derive the SoP expression

$$r = (\neg y \wedge z) \vee (\neg x \wedge \neg z).$$

▷ **S77.** Define $\barwedge$ as the NAND operation with the truth table:

x	y	$x \barwedge y$
0	0	1
0	1	1
1	0	1
1	1	0

Using NAND, we can implement NOT, AND and OR as follows:

$$
\begin{array}{rcl}
\neg x & = & x \barwedge x \\
x \wedge y & = & (x \barwedge y) \barwedge (x \barwedge y) \\
x \vee y & = & (x \barwedge x) \barwedge (y \barwedge y)
\end{array}
$$

To prove this works, we can construct truth tables for the expressions and compare the results with what we would expect; for NOT we have:

x	$x \barwedge x$	$\neg x$
0	1	1
1	1	0

while for AND we have:

x	y	$x \barwedge y$	$(x \barwedge y) \barwedge (x \barwedge y)$	$x \wedge y$
0	0	1	0	0
0	1	1	0	0
1	0	1	0	0
1	1	0	1	1

and finally for OR we have:

x	y	$x \barwedge x$	$y \barwedge y$	$(x \barwedge y) \barwedge (x \barwedge y)$	$x \vee y$
0	0	1	1	0	0
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	1	1

such that it should be clear all three are correct.

▷ **S78.** Conventionally a 4-input, 1-bit multiplexer might be described using a truth table such as the following:

c_1	c_0	w	x	y	z	r
0	0	?	?	?	0	0
0	0	?	?	?	1	1
0	1	?	?	0	?	0
0	1	?	?	1	?	1
1	0	?	0	?	?	0
1	0	?	1	?	?	1
1	1	0	?	?	?	0
1	1	1	?	?	?	1

This assumes that there are four inputs, namely w, x, y and z, with two further control signals c_1 and c_0 deciding which of them provides the output r. However, another valid way to write the same thing would be

c_1	c_0	r
0	0	w
0	1	x
1	0	y
1	1	z

This reformulation describes a 2-input, 1-output Boolean function whose behaviour is selected by fixing w, x, y and z, i.e., connecting each of them directly to either 0 or 1. For instance, if $w = x = y = 0$ and $z = 1$ then the truth table becomes

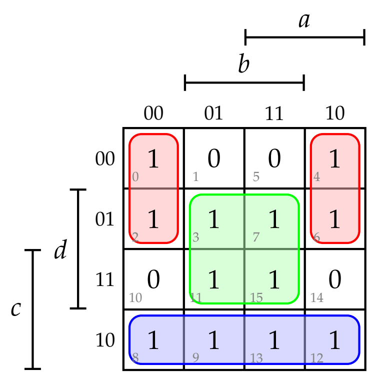
c_1	c_0	r
0	0	$w = 0$
0	1	$x = 0$
1	0	$y = 0$
1	1	$z = 1$

which is of course the same as AND. So depending on how w, x, y and z are fixed (on a per-instance basis) we can form *any* 2-input, 1-output Boolean function; this includes NAND and NOR, which we know are universal, meaning the multiplexer is also universal.

▷ **S79.** a The expression for this circuit can be written as

$$e = (\neg c \wedge \neg b) \vee (b \wedge d) \vee (\neg a \wedge c \wedge \neg d) \vee (a \wedge c \wedge \neg d)$$

which yields the Karnaugh map



and from which we can derive a simplified SoP form for e, namely

$$e = (b \wedge d) \vee (\neg b \wedge \neg c) \vee (c \wedge \neg d)$$

b The advantages of this expression over the original are that is is simpler, i.e., contains less terms and hence needs less gates for implementation, and shows that the input a is essentially redundant. We have probably also reduced the critical path through the circuit since it is more shallow. The disadvantages are that we still potentially have some glitching due to the differing delays through paths in the circuit, although these existed before as well, and the large propagation delay.

c The longest sequential path through the circuit goes through a NOT gate, two AND gates and two OR gates; the critical path is thus 90ns long. This time bounds how fast we can used it in a clocked system since the clock period must be at least 90ns. So the shortest clock period would be 90ns, meaning the clock ticks about 11111111 times a second (or at about 11MHz).

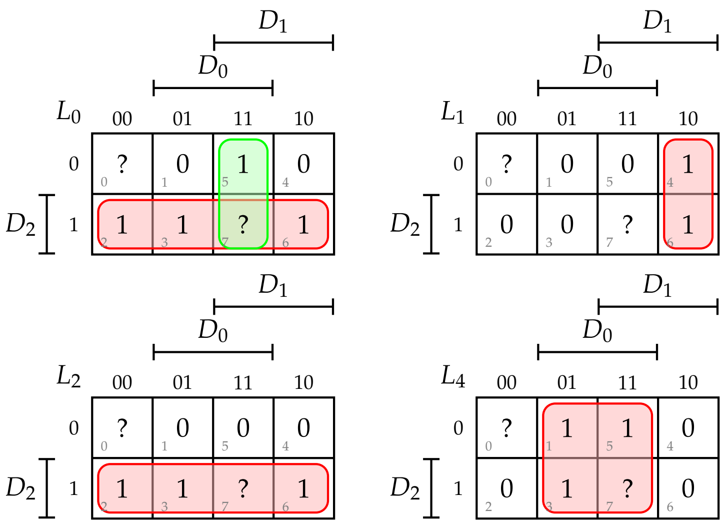
▷ **S80.** a Examining the behaviour required, we can construct the following truth table:

D_2	D_1	D_0	L_8	L_7	L_6	L_5	L_4	L_3	L_2	L_1	L_0
0	0	0	?	?	?	?	?	?	?	?	?
0	0	1	0	0	0	0	1	0	0	0	0
0	1	0	0	1	0	0	0	0	0	1	0
0	1	1	1	0	0	0	1	0	0	0	1
1	0	0	1	0	1	0	0	0	1	0	1
1	0	1	1	0	1	0	1	0	1	0	1
1	1	0	1	1	1	0	0	0	1	1	1
1	1	1	?	?	?	?	?	?	?	?	?

Note that

$$\begin{array}{lcl} L_3 &=& 0 \\ L_5 &=& 0 \\ L_6 &=& L_2 \\ L_7 &=& L_1 \\ L_8 &=& L_0 \end{array}$$

so actually we only need expressions for $L_{0...2}$ and L_4, and that don't care states are used to capture the idea that $D = 0$ and $D = 7$ never occur. The resulting four Karnaugh maps



can be translated into the expressions:

$$\begin{array}{lcl} L_0 &=& D_2 \vee (D_1 \wedge D_0) \\ L_1 &=& (D_1 \wedge \neg D_0) \\ L_2 &=& D_2 \\ L_4 &=& D_0 \end{array}$$

b All the LEDs can be driven in parallel, i.e., the critical path relates to the single expression whose critical path is the most. $L_{2...6}$ have no logic involved, so we can discount them immediately. Of the two remaining LEDs, we find

$$\begin{array}{lcl} L_0 &\rightsquigarrow& 20\text{ns} + 20\text{ns} \\ L_1 &\rightsquigarrow& 10\text{ns} + 20\text{ns} \end{array}$$

hence L_0 represents the critical path of 40ns. Thus if one throw takes 40ns, we can perform

$$\frac{1\text{s}}{40\text{ns}} = \frac{1 \cdot 10^9\text{ns}}{40\text{ns}} = 25000000$$

throws per-second. Which is quite a lot, and certainly too many to actually see with the human eye!

c i A rough block diagram would resemble

```
              +-----+           +-----+           +-----+
co <----------|co ci|<- - - - - |co ci|<----------|co ci|<---------- ci = 0
           +--|s   y|<----+  +--|s   y|<----+  +--|s   y|<----+
           |  |    x|<-+  |  |  |    x|<-+  |  |  |    x|<-+  |
           |  +-----+  |  |  |  +-----+  |  |  |  +-----+  |  |
           |           |  |  |           |  |  |           |  |
           |     x_{n-1}  |  |        x_1   |  |        x_0   |
           |              |  |              |  |              |
           |        y_{n-1}  |           y_1   |           y_0
           v                 v                 v
           r_{n-1}           r_1               r_0
```

ii If we sum 8 values $1 \leq x_i \leq 6$, where x_i is the i-th throw (or i-th value of D supplied), then the maximum total is $8 \cdot 6 = 48$. We can represent this in 6 bits, hence $n = 6$.

iii Using the left-shift method, we compute $D' = 2 \cdot D$ by simply relabelling the bits in D. That is, $D'_0 = 0$ and $D'_{i+1} = D_i$ for $0 \leq i < 3$. For example, given $D = 6_{(10)} = 110_{(2)}$ we have

$$\begin{array}{lclcl} D'_0 & & & = & 0 \\ D'_1 & = & D_0 & = & 0 \\ D'_2 & = & D_1 & = & 1 \\ D'_3 & = & D_2 & = & 1 \end{array}$$

and hence $D' = 1100_{(2)} = 12_{(10)}$. Since there is no need for any logic gates to implement this method, the critical path is essentially nil: the only propagation delay relates to (small) wire delays. In comparison to the larger critical path of a suitable n-bit adder, this clearly means the left-shift approach is preferable.

▷ **S81.** a A basic design would use two building blocks:

- lth_8bit compares two 8-bit inputs a and b and produces a 1-bit result r, where $r = 1$ if $a < b$ and $r = 0$ if $a \geq b$:

```
    a   b
    |   |
    v   v
+-----------+
|  lth_8bit |
+-----------+
      |
      v
      r
```

- mux2_8bit selects between two 8-bit inputs; if the inputs are a and b, the output $r = a$ if the control signal $s = 0$, or $r = b$ if $s = 1$:

```
    a   b
    |   |
    v   v
+-----------+
| mux2_8bit |<-- s
+-----------+
      |
      v
      r
```

Based on these building blocks, one can describe the component C as follows:

```
              x   y
              |   |
              v   v
          +-----------+
          |  lth_8bit |
          +-----------+
    y   x       |       x   y
    |   |       |       |   |
    v   v       |       v   v
+-----------+   |   +-----------+
| mux2_8bit |<--+-->| mux2_8bit |
+-----------+   r   +-----------+
      |                   |
      v                   v
   min(x,y)            max(x,y)
```

From a functional perspective, C compares x and y using an instance of the `lth_8bit` building block, and then uses the result r as a control signal for two instances of `mux2_8bit`. The left-hand instance selects y as the output if $r = 0$ and x if $r = 1$; that is, if $x < y$ then the output is $x = \min(x, y)$ otherwise the output is $y = \min(x, y)$. The right-hand instance swaps the inputs so it selects x as the output if $r = 0$ and y if $r = 1$; that is, if $x < y$ then the output is $y = \max(x, y)$ otherwise the output is $x = \max(x, y)$.

b The short answer (which gets about half the marks) is that the longest path through the mesh will go through $2n - 1$ of the C components: this is the path from the top-left corner down along one edge to the bottom-left and then along another edge to the bottom-right. So in a sense, if we write the propagation delay associated with each instance of C as T_C then the overall critical path is

$$(2n - 1) \cdot T_C.$$

In a bit more detail, the critical path through C is through one instance of `lth_8bit` and one instance of `mux2_8bit`. So we could write the overall critical path is

$$(2n - 1) \cdot (T_{\texttt{lth\_8bit}} + T_{\texttt{mux2\_8bit}}).$$

To be more detailed than this, we need to think about individual logic gates. Imagine we assume $T_{AND} = 50\text{ns}$, $T_{AND} = 20\text{ns}$, $T_{OR} = 20\text{ns}$ and $T_{NOT} = 10\text{ns}$.

- `mux2_8bit` is simply eight `mux2_1bit` instances placed in parallel with each other; that is, the i-th such instance produces the i-th bit of the output based on the i-th bit of the inputs (but all using the same control signal). Assuming that the propagation delay of AND and OR gates dominates that of a NOT gate, the critical path through `mux2_1bit` will be $T_{AND} + T_{OR}$.
- `lth_8bit` is a combination of eight sub-components:

```
  x_i   y_i      x_i   y_i
   |     |        |     |
   v     v        v     v
+-----------+ +-----------+
| lth_1bit  | | equ_1bit  |
+-----------+ +-----------+
   |               |
   |               |  t_{i-1}
   |               |     |
   |               v     v
   |          +-----------+
   |     +----|    AND    |
   |     |    +-----------+
   v     v
+-----------+
|    OR     |
+-----------+
      |
      v
     t_i
```

  Each of these sub-components is placed in series so that t_{i-1} is an input from the previous sub-component and t_i is an output provided to the next.
  Based on simple circuits derived from their truth tables, the critical paths for `lth_1bit` and `equ_1bit` are $T_{AND} + T_{NOT}$ and $T_{XOR} + T_{NOT}$ respectively. Thus the critical path of the whole sub-component is $T_{XOR} + T_{NOT} + T_{AND} + T_{OR}$ (since the critical path of `equ_1bit` is longer). Overall, the critical path of `lth_8bit` is

$$8 \cdot (T_{XOR} + T_{NOT} + T_{AND} + T_{OR}),$$

  or more exactly

$$7 \cdot (T_{XOR} + T_{NOT} + T_{AND} + T_{OR}) + T_{AND} + T_{NOT}$$

  because the 0-th sub-component is "special": there is no input from the previous sub-component.

Using this we can write the overall critical path for the mesh as

$$(2n - 1) \cdot (7 \cdot T_{XOR} + 8 \cdot T_{NOT} + 9 \cdot T_{AND} + 8 \cdot T_{OR})$$

or roughly $(2n - 1) \cdot 770\text{ns}$ if we plug in the assumed delays.

c One problem is that the mesh does not always give the right result! If you were to build a 4×4 mesh and feed $5, 6, 7, 8$ into the top and $4, 3, 2, 1$ into the left-hand side, the bottom reads $5, 6, 7, 8$ and the right-hand side reads $4, 3, 2, 1$: numbers cannot move from bottom to top or from right to left, so there are some inputs a mesh cannot sort.

Beyond this trick question, the main idea is that the mesh is special-purpose, the processor is general-purpose: this implies a number of trade-offs in either direction that could be viewed as advantages or disadvantages in certain cases. For example, depending on n, one might argue that the processor will be require more logic to realise it (since it will include features extraneous to the task of sorting). Since it operates a fetch-decode-execute cycle to complete each instruction, there is an overhead (i.e., the fetch and decode at least) which means it potentially performs the task of sorting less quickly. On the other hand, once constructed the mesh is specialised to one task: it cannot be used to sort strings for example, and the size of input (i.e., n) is fixed. The processor makes the opposite trade-off; it should be clear that while it might be slower and potentially larger, it is vastly more flexible.

▷ **S82.** a Imagine a component which is enabled (i.e., "turned on") using the input en:

- The idea of the component being level triggered is that the value of en is important, not a change in en: the component is enabled when en has a particular value, rather than at an edge when the value changes.
- The fact en is active high means that the component is enabled when $en = 1$ (rather than $en = 0$ which would make it active low). Though active high might seem the more logical choice, this is just part of the component specification: as long as everything is consistent, i.e., uses the right semantics to "turn on" the component, there is sometimes no major benefit of one approach over the other.

b Assume that M is a 4-state switch represented by a 2-bit value $M = \langle M_0, M_1 \rangle$: $\langle 0, 0 \rangle$ means off, $\langle 1, 0 \rangle$ means slow, $\langle 0, 1 \rangle$ means fast and $\langle 1, 1 \rangle$ means very fast. Also assume there is a clock signal called clk available, for example supplied by an oscillator of some form.

One approach would basically be to take clk and divide it to create two new clock signals c_0 and c_1 which have a longer period: each of the clock signals could then satisfy the criteria of toggling the fire button on and off at various speeds. A clock divider is fairly simple: the idea is to have a counter c clocked by clk and to sample the $(i-1)$-th bit of the counter: this behaves like clk divided by 2^i. For example the 0-th bit acts like clk but with twice the period.

A circuit to do this is fairly simple: we need some D-type flip-flops to hold the counter state, and some full-adders to increment the counter:

```
   +-----+           +-----+
   |co ci|<----------|co ci|<-- 0
+--|s   y|<-- 0   +--|s   y|<-- 1
|  |    x|<-+     |  |    x|<-+
|  +-----+  |     |  +-----+  |
|           |     |           |
|     c_1 --+     |     c_0 --+
|           |     |           |
|  +-----+  |     |  +-----+  |
+->|D   Q|--+     +->|D   Q|--+
   |    <|<-+        |    <|<-+
   |     |  |        |     |  |
   +-----+  |        +-----+  |
            |                 |
            +-----------------+-- clk
```

Given such a component which runs freely as long as it is driven by clk, we want to feed the original fire button F_0 through to form the new fire button input F'_0 when $M = 0$, and c_1, c_0 or clk through when $M = 1$, $M = 2$ or $M = 3$ (meaning a slow, fast or very fast toggling behaviour). We can describe this as the following truth table:

M_1	M_0	F'_0
0	0	F_0
0	1	c_1
1	0	c_0
1	1	clk

This is essentially a multiplexer controlled by M, and permits us to write

$$\begin{array}{rclcccccl} F'_0 & = & (& \neg M_0 & \wedge & \neg M_1 & \wedge & F_0 &) \\ & & (& M_0 & \wedge & \neg M_1 & \wedge & c_1 &) \\ & & (& \neg M_0 & \wedge & M_1 & \wedge & c_0 &) \\ & & (& M_0 & \wedge & M_1 & \wedge & clk &) \end{array}$$

c i A synchronous protocol demands that the console and controller share a clock signal which acts to synchronise their activity, e.g., ensures each one sends and receives data at the right time. The problem with this is ensuring that the clock is not skewed for either component: since they are physically separate, this might be hard and hence this is not such a good option.

An asynchronous protocol relies on extra connections between the components, e.g., "request" and "acknowledge", that allow them to engage in a form of transaction: the extra connections essentially signal when data has been sent or received on the associated bus. This is more suitable given the scenario: the extra connections could potentially be shared with those that already exist (e.g., F_0, F_1, F_2 and D) thereby reducing the overhead, plus performance is not a big issue here (the protocol will presumably only be executed once when the components are turned on or plugged in).

Both approaches have an issue in that

- once the protocol is run someone could just plug in another, fake controller, or
- or simply intercept c and $T(c)$ pairs until it recovers the whole look-up table and then "imitate" it using a fake controller

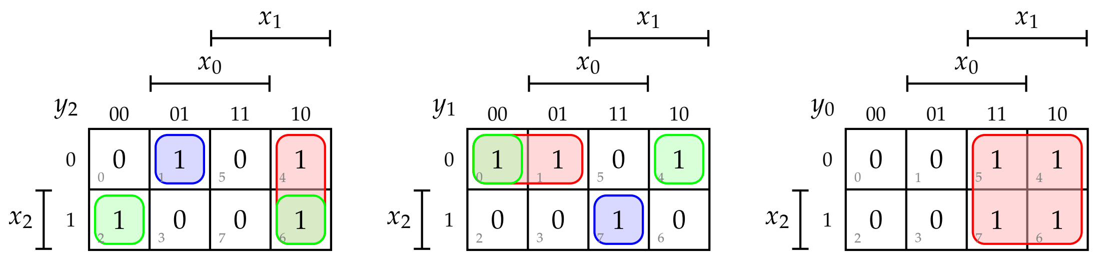
so neither is particularly robust from a security point of view!

ii The temptation here is to say that the use of a 3-bit memory (or register) is the right way to go. Although this allows some degree of flexibility which is not required since the function is fixed, the main disadvantage is retention of the content when the controller or console is turned off: some form of non-volatile memory is therefore needed.

However, we can easily construct some dedicated logic to do the same thing. If we say that $y = T(x)$, the we can describe the behaviour of T using the following truth table:

x_2	x_1	x_0	y_2	y_1	y_0
0	0	0	0	1	0
0	0	1	1	1	0
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	1	0	0
1	0	1	0	0	0
1	1	0	1	0	1
1	1	1	0	1	1

This can be transformed into the following Karnaugh maps for y_0, y_1 and y_2



which in turn can be transformed into the following equations

$$\begin{array}{rcl}
y_2 & = & (\quad\quad\quad x_1 \wedge \neg x_0) \vee \\
 & & (x_2 \quad\quad\quad \wedge \neg x_0) \vee \\
 & & (\neg x_2 \wedge \neg x_1 \wedge x_0) \\
y_1 & = & (\neg x_2 \wedge \neg x_1 \quad\quad\quad) \vee \\
 & & (\neg x_2 \quad\quad\quad \wedge \neg x_0) \vee \\
 & & (x_2 \wedge x_1 \wedge x_0) \\
y_0 & = & (\quad\quad\quad x_1 \quad\quad\quad)
\end{array}$$

which are enough to implement the look-up table: we pass x as input, and it produces the right y (for this fixed T) as output.

▷ **S83.** This is a classic "puzzle" question in digital logic. There are a few ways to describe the strategy, but the one used here is based on counting the number of inputs which are 1. In short, we start by computing

$$\begin{array}{rcl}
t_1 & = & \neg(x \wedge y \vee y \wedge z \vee x \wedge z) \\
t_2 & = & \neg((x \wedge y \wedge z) \vee t_1 \wedge (x \vee y \vee z))
\end{array}$$

which use our quota of NOT gates. The idea is that $t_1 = 1$ iff. one or zero of x, y and z are 1, and in the same way $t_2 = 1$ iff. two or zero of x, y and z are 1. This can be hard to see, so consider a truth table

x	y	z	$x \wedge y$	$y \wedge z$	$x \wedge z$	$x \wedge y \wedge z$	$x \vee y \vee z$
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1
0	1	0	0	0	0	0	1
0	1	1	0	1	0	0	1
1	0	0	0	0	0	0	1
1	0	1	0	0	1	0	1
1	1	0	1	0	0	0	1
1	1	1	1	1	1	1	1

meaning

x	y	z	$x \wedge y \vee y \wedge z \vee x \wedge z$	t_1	$(x \wedge y \wedge z) \vee t_1 \wedge (x \vee y \vee z)$	t_2
0	0	0	0	1	0	1
0	0	1	0	1	1	0
0	1	0	0	1	1	0
0	1	1	1	0	0	1
1	0	0	0	1	1	0
1	0	1	1	0	0	1
1	1	0	1	0	0	1
1	1	1	1	0	1	0

and hence t_1 and t_2 are as required. Now, we can generate the three results as

$$\begin{array}{rcl}
\neg x & = & (t_1 \wedge t_2) \vee (t_1 \wedge (y \vee z)) \vee (t_2 \wedge y \wedge z) \\
 & = & (t_1 \wedge t_2) \vee (t_1 \wedge y) \vee (t_1 \wedge z) \vee (t_2 \wedge y \wedge z) \\
\neg y & = & (t_1 \wedge t_2) \vee (t_1 \wedge (x \vee z)) \vee (t_2 \wedge x \wedge z) \\
 & = & (t_1 \wedge t_2) \vee (t_1 \wedge x) \vee (t_1 \wedge z) \vee (t_2 \wedge x \wedge z) \\
\neg z & = & (t_1 \wedge t_2) \vee (t_1 \wedge (x \vee y)) \vee (t_2 \wedge x \wedge y) \\
 & = & (t_1 \wedge t_2) \vee (t_1 \wedge x) \vee (t_1 \wedge y) \vee (t_2 \wedge x \wedge y)
\end{array}$$

▷ **S84.** Imagine that for some n-bit input x, we let $y_i = C_i(x)$ denote the evaluation of C_i to get an output y_i. As such, the equivalence of C_1 and C_2 can be stated as a test whether $y_1 = y_2$ for all values of x; another way to say the same thing is to test whether an x exists such that $y_1 \neq y_2$ which will distinguish the circuits, i.e., imply they are not equivalent.

Using the second formulation, we can write the test as $y_1 \oplus y_2$ since the XOR will produce 1 when y_1 differs from y_2 and 0 otherwise. As such, we have n Boolean variables (the bits of x) and want an assignment that implies the expression $C_1(x) \oplus C_2(x)$ will evaluate to 1. This is the same as described in the description of SAT, so if we can solve the SAT instance we prove the circuits are (not) equivalent.

▷ **S85.** a The latency of the circuit is the time taken to perform the computation, i.e., to compute some r given x. For this circuit, the latency is simply the sum of the critical paths.

b The throughput is the number of operations performed per unit time period. This is essentially the number of operations we can start (resp. that finish) within that time period.

By pipelining the circuit, using say 3 stages, one might expect the latency to increase slightly (by virtue of having to add pipeline registers between each stage) but the throughput to increase (by virtue of decreasing the overall critical path to the longest stage, and hence increasing the maximum clock frequency). The trade-off is strongly influenced by the number of and balance between stages, meaning careful analysis of the circuit before applying the optimisation is important.

▷ **S86.** a The latency of a circuit is the time elapsed between when a given operation starts and when it finishes. The throughput of a circuit is the number of operations that can be started in each time period; that is, how long it takes between when two subsequent operations can be started.

b The latency of the circuit is the sum of all the latencies of the parts,i.e.,

$$40\text{ns} + 10\text{ns} + 30\text{ns} + 10\text{ns} + 50\text{ns} + 10\text{ns} + 10\text{ns} = 160\text{ns}.$$

The throughput relates to the length of the longest pipeline stage; the circuit is not pipelined, so more specifically we can say it is $\frac{1}{160 \cdot 10^{-9}}$.

c The new latency is still the sum of all the parts, but now includes the extra pipeline register:

$$40\text{ns} + 10\text{ns} + 30\text{ns} + 10\text{ns} + 10\text{ns} + 50\text{ns} + 10\text{ns} + 10\text{ns} = 170\text{ns}.$$

However, the throughput is now more because the longest pipeline stage only has a latency of 100ns (including the extra register). Specifically, the throughput increases to $\frac{1}{100 \cdot 10^{-9}}$ which essentially means we can start new operations more often than before.

d To maximise the throughput we need to minimise the latency of the longest pipeline stage (i.e., the one whose individual latency is the largest) since this will act as a limit. The latency of part E is largest (at 50ns) and hence represents said limit: the longest pipeline stage cannot have a latency of less than 60ns (i.e., the latency of part E plus the latency of a pipeline register).

We can achieve this by creating a 4-stage pipeline: adding two more pipeline registers, between parts B and C and parts E and F, ensures the stages have latencies of

$$\begin{array}{lcl}
A + B + REG & \rightsquigarrow & 40\text{ns} + 10\text{ns} + 10\text{ns} & = & 60\text{ns} \\
C + D + REG & \rightsquigarrow & 30\text{ns} + 10\text{ns} + 10\text{ns} & = & 50\text{ns} \\
E + REG & \rightsquigarrow & 50\text{ns} + 10\text{ns} & = & 60\text{ns} \\
F + REG & \rightsquigarrow & 10\text{ns} + 10\text{ns} & = & 20\text{ns}
\end{array}$$

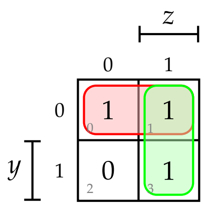
Overall, the latency is increased to 190ns but the throughput is $\frac{1}{60 \cdot 10^{-9}}$.

Part III: Basics of digital logic: minimisation via Karnaugh maps

▷ **S87.** a Reference implementation:

$$\begin{array}{lclcccl}
r & = & (& \neg y & \wedge & \neg z &) \vee \\
 & & (& \neg y & \wedge & z &) \vee \\
 & & (& y & \wedge & z &)
\end{array}$$

b Annotated Karnaugh map



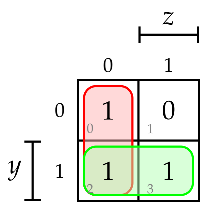
and associated, optimised implementation:

$$\begin{array}{lclcl}
r & = & (& \neg y &) \vee \\
 & & (& z &)
\end{array}$$

▷ **S88.** a Reference implementation:

$$\begin{array}{lclcccl}
r & = & (& \neg y & \wedge & \neg z &) \vee \\
 & & (& y & \wedge & \neg z &) \vee \\
 & & (& y & \wedge & z &)
\end{array}$$

b Annotated Karnaugh map



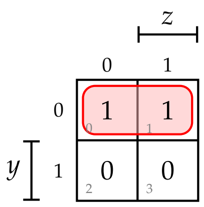
and associated, optimised implementation:

$$\begin{array}{lclcl}
r & = & (& \neg z &) \vee \\
 & & (& y &)
\end{array}$$

▷ **S89.** a Reference implementation:

$$r = (\neg y \wedge \neg z) \vee (\neg y \wedge z)$$

b Annotated Karnaugh map



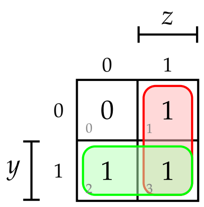
and associated, optimised implementation:

$$r = (\neg y)$$

▷ **S90.** a Reference implementation:

$$r = (\neg y \wedge z) \vee (y \wedge \neg z) \vee (y \wedge z)$$

b Annotated Karnaugh map



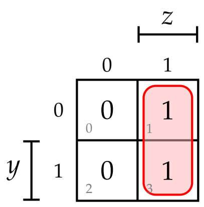
and associated, optimised implementation:

$$r = (z) \vee (y)$$

▷ **S91.** a Reference implementation:

$$r = (\neg y \wedge z) \vee (y \wedge z)$$

b Annotated Karnaugh map



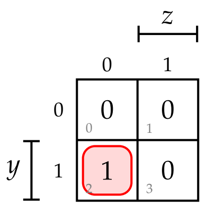
and associated, optimised implementation:

$$r = (z)$$

▷ **S92.** a Reference implementation:

$$r = (y \wedge \neg z)$$

b Annotated Karnaugh map



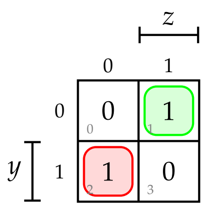
and associated, optimised implementation:

$$r = (y \wedge \neg z)$$

▷ **S93.** a Reference implementation:

$$\begin{array}{lllllll} r & = & (& \neg y & \wedge & z &) \vee \\ & & (& y & \wedge & \neg z &) \end{array}$$

b Annotated Karnaugh map



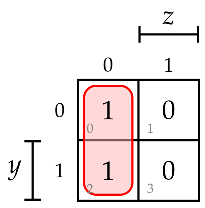
and associated, optimised implementation:

$$\begin{array}{lllllll} r & = & (& y & \wedge & \neg z &) \vee \\ & & (& \neg y & \wedge & z &) \end{array}$$

▷ **S94.** a Reference implementation:

$$\begin{array}{lllllll} r & = & (& \neg y & \wedge & \neg z &) \vee \\ & & (& y & \wedge & \neg z &) \end{array}$$

b Annotated Karnaugh map



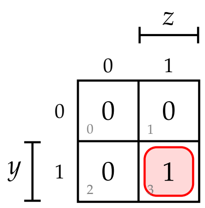
and associated, optimised implementation:

$$r = (\neg z)$$

▷ **S95.** a Reference implementation:

$$r = (y \wedge z)$$

b Annotated Karnaugh map



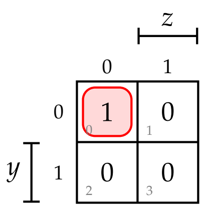
and associated, optimised implementation:

$$r = (y \wedge z)$$

▷ **S96.** a Reference implementation:

$$r = (\neg y \wedge \neg z)$$

b Annotated Karnaugh map



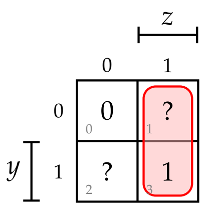
and associated, optimised implementation:

$$r = (\neg y \wedge \neg z)$$

▷ **S97.** a Reference implementation:

$$r = (y \wedge z)$$

b Annotated Karnaugh map



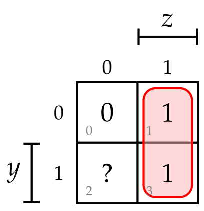
and associated, optimised implementation:

$$r = (z)$$

▷ **S98.** a Reference implementation:

$$\begin{array}{rcl} r & = & (\neg y \wedge z) \vee \\ & & (y \wedge z) \end{array}$$

b Annotated Karnaugh map



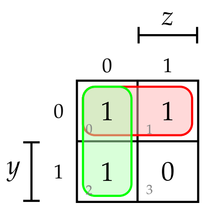
and associated, optimised implementation:

$$r = (z)$$

▷ **S99.** a Reference implementation:

$$\begin{array}{rcl} r & = & (\neg y \wedge \neg z) \vee \\ & & (\neg y \wedge z) \vee \\ & & (y \wedge \neg z) \end{array}$$

b Annotated Karnaugh map



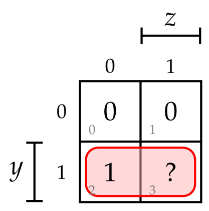
and associated, optimised implementation:

$$\begin{array}{rcl} r & = & (\neg y) \vee \\ & & (\neg z) \end{array}$$

▷ **S100.** a Reference implementation:

$$r = (y \wedge \neg z)$$

b Annotated Karnaugh map



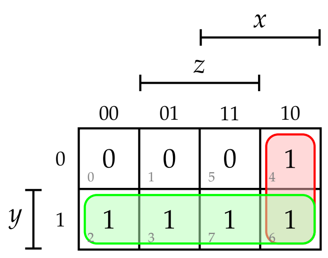
and associated, optimised implementation:

$$r = (y)$$

▷ **S101.** a Reference implementation:

$$\begin{array}{rcl} r & = & (\neg x \wedge y \wedge \neg z) \vee \\ & & (\neg x \wedge y \wedge z) \vee \\ & & (x \wedge \neg y \wedge \neg z) \vee \\ & & (x \wedge y \wedge \neg z) \vee \\ & & (x \wedge y \wedge z) \end{array}$$

b Annotated Karnaugh map



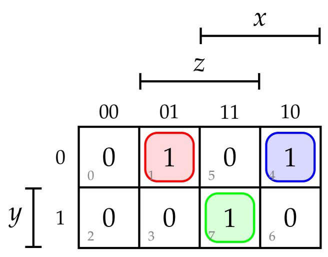
and associated, optimised implementation:

$$\begin{array}{rcl} r & = & (x \wedge \neg z) \vee \\ & & (y) \end{array}$$

▷ **S102.** a Reference implementation:

$$\begin{array}{rcl} r & = & (\neg x \wedge \neg y \wedge z) \vee \\ & & (x \wedge \neg y \wedge \neg z) \vee \\ & & (x \wedge y \wedge z) \end{array}$$

b Annotated Karnaugh map



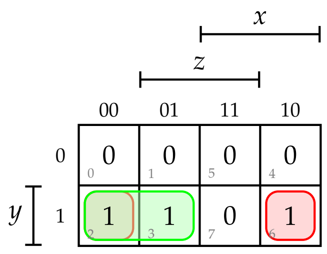
and associated, optimised implementation:

$$\begin{array}{rcl} r & = & (\neg x \wedge \neg y \wedge z) \vee \\ & & (x \wedge y \wedge z) \vee \\ & & (x \wedge \neg y \wedge \neg z) \end{array}$$

▷ **S103.** a Reference implementation:

$$\begin{array}{lcl} r & = & (\neg x \wedge y \wedge \neg z) \vee \\ & & (\neg x \wedge y \wedge z) \vee \\ & & (x \wedge y \wedge \neg z) \end{array}$$

b Annotated Karnaugh map



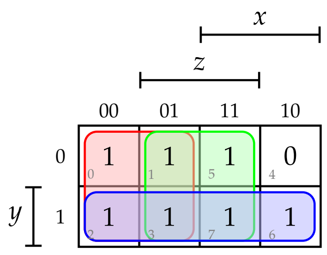
and associated, optimised implementation:

$$\begin{array}{lcl} r & = & (y \wedge \neg z) \vee \\ & & (\neg x \wedge y) \end{array}$$

▷ **S104.** a Reference implementation:

$$\begin{array}{lcl} r & = & (\neg x \wedge \neg y \wedge \neg z) \vee \\ & & (\neg x \wedge \neg y \wedge z) \vee \\ & & (\neg x \wedge y \wedge \neg z) \vee \\ & & (\neg x \wedge y \wedge z) \vee \\ & & (x \wedge \neg y \wedge z) \vee \\ & & (x \wedge y \wedge \neg z) \vee \\ & & (x \wedge y \wedge z) \end{array}$$

b Annotated Karnaugh map



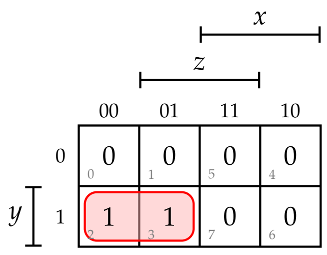
and associated, optimised implementation:

$$\begin{array}{lcl} r & = & (\neg x) \vee \\ & & (z) \vee \\ & & (y) \end{array}$$

▷ **S105.** a Reference implementation:

$$\begin{array}{lcl} r & = & (\neg x \wedge y \wedge \neg z) \vee \\ & & (\neg x \wedge y \wedge z) \end{array}$$

b Annotated Karnaugh map



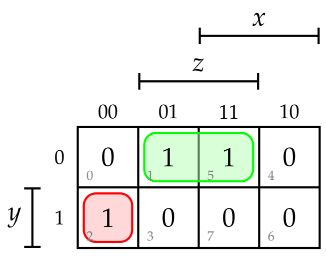
and associated, optimised implementation:

$$r = (\neg x \wedge y)$$

▷ **S106.** a Reference implementation:

$$\begin{array}{lcl} r & = & (\neg x \wedge \neg y \wedge z) \vee \\ & & (\neg x \wedge y \wedge \neg z) \vee \\ & & (x \wedge \neg y \wedge z) \end{array}$$

b Annotated Karnaugh map



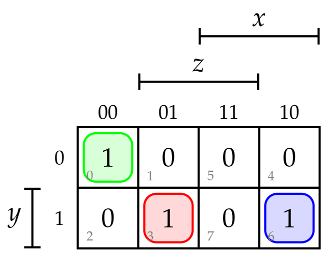
and associated, optimised implementation:

$$\begin{array}{lcl} r & = & (\neg x \wedge y \wedge \neg z) \vee \\ & & (\neg y \wedge z) \end{array}$$

▷ **S107.** a Reference implementation:

$$\begin{array}{lcl} r & = & (\neg x \wedge \neg y \wedge \neg z) \vee \\ & & (\neg x \wedge y \wedge z) \vee \\ & & (x \wedge y \wedge \neg z) \end{array}$$

b Annotated Karnaugh map



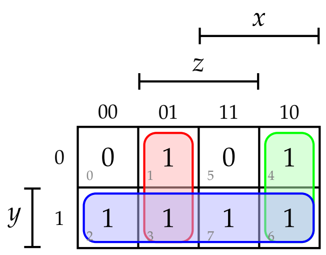
and associated, optimised implementation:

$$\begin{array}{lcl} r & = & (\neg x \wedge y \wedge z) \vee \\ & & (\neg x \wedge \neg y \wedge \neg z) \vee \\ & & (x \wedge y \wedge \neg z) \end{array}$$

▷ **S108.** a Reference implementation:

$$\begin{array}{lcl} r & = & (\neg x \wedge \neg y \wedge z) \vee \\ & & (\neg x \wedge y \wedge \neg z) \vee \\ & & (\neg x \wedge y \wedge z) \vee \\ & & (x \wedge \neg y \wedge \neg z) \vee \\ & & (x \wedge y \wedge \neg z) \vee \\ & & (x \wedge y \wedge z) \end{array}$$

b Annotated Karnaugh map



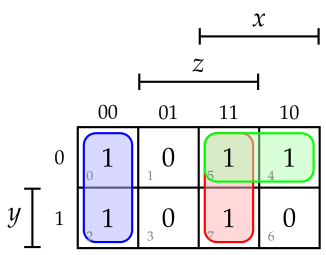
and associated, optimised implementation:

$$\begin{array}{rcllllllll} r & = & (& \neg x & & & \wedge & z &) & \vee \\ & & (& x & & & \wedge & \neg z &) & \vee \\ & & (& & & y & & &) & \end{array}$$

▷ **S109.** a Reference implementation:

$$\begin{array}{rcllllllll} r & = & (& \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& \neg x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& x & \wedge & \neg y & \wedge & z &) & \vee \\ & & (& x & \wedge & y & \wedge & z &) & \end{array}$$

b Annotated Karnaugh map



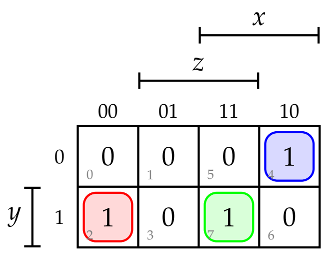
and associated, optimised implementation:

$$\begin{array}{rcllllllll} r & = & (& x & & & \wedge & z &) & \vee \\ & & (& x & \wedge & \neg y & & &) & \vee \\ & & (& \neg x & & & \wedge & \neg z &) & \end{array}$$

▷ **S110.** a Reference implementation:

$$\begin{array}{rcllllllll} r & = & (& \neg x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& x & \wedge & y & \wedge & z &) & \end{array}$$

b Annotated Karnaugh map



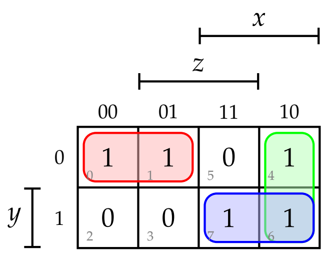
and associated, optimised implementation:

$$\begin{array}{rcllllllll} r & = & (& \neg x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& x & \wedge & y & \wedge & z &) & \vee \\ & & (& x & \wedge & \neg y & \wedge & \neg z &) & \end{array}$$

▷ **S111.** a Reference implementation:

$$\begin{array}{lcl} r & = & (\neg x \wedge \neg y \wedge \neg z) \vee \\ & & (\neg x \wedge \neg y \wedge z) \vee \\ & & (x \wedge \neg y \wedge \neg z) \vee \\ & & (x \wedge y \wedge \neg z) \vee \\ & & (x \wedge y \wedge z) \end{array}$$

b Annotated Karnaugh map



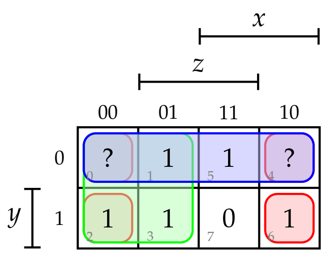
and associated, optimised implementation:

$$\begin{array}{lcl} r & = & (\neg x \wedge \neg y) \vee \\ & & (x \wedge \neg z) \vee \\ & & (x \wedge y) \end{array}$$

▷ **S112.** a Reference implementation:

$$\begin{array}{lcl} r & = & (\neg x \wedge \neg y \wedge z) \vee \\ & & (\neg x \wedge y \wedge \neg z) \vee \\ & & (\neg x \wedge y \wedge z) \vee \\ & & (x \wedge \neg y \wedge z) \vee \\ & & (x \wedge y \wedge \neg z) \end{array}$$

b Annotated Karnaugh map



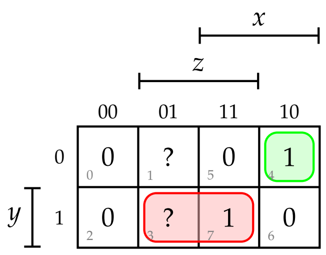
and associated, optimised implementation:

$$\begin{array}{lcl} r & = & (\neg z) \vee \\ & & (\neg x) \vee \\ & & (\neg y) \end{array}$$

▷ **S113.** a Reference implementation:

$$\begin{array}{lcl} r & = & (x \wedge \neg y \wedge \neg z) \vee \\ & & (x \wedge y \wedge z) \end{array}$$

b Annotated Karnaugh map



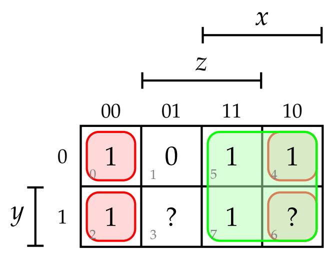
and associated, optimised implementation:

$$\begin{array}{lllllllll} r & = & (& & y & \wedge & z &) & \vee \\ & & (& x \wedge & \neg y & \wedge & \neg z &) & \end{array}$$

▷ **S114.** a Reference implementation:

$$\begin{array}{lllllllllll} r & = & (& \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& \neg x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& x & \wedge & \neg y & \wedge & z &) & \vee \\ & & (& x & \wedge & y & \wedge & z &) & \end{array}$$

b Annotated Karnaugh map



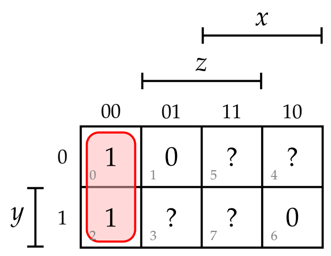
and associated, optimised implementation:

$$\begin{array}{lllll} r & = & (& \neg z &) & \vee \\ & & (& x &) & \end{array}$$

▷ **S115.** a Reference implementation:

$$\begin{array}{lllllllllll} r & = & (& \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& \neg x & \wedge & y & \wedge & \neg z &) & \end{array}$$

b Annotated Karnaugh map



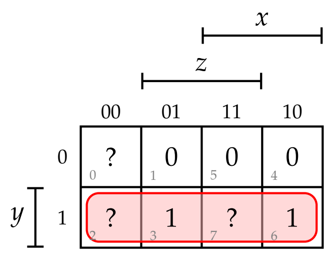
and associated, optimised implementation:

$$r = (\neg x \wedge \neg z)$$

▷ **S116.** a Reference implementation:

$$\begin{array}{lllllllllll} r & = & (& \neg x & \wedge & y & \wedge & z &) & \vee \\ & & (& x & \wedge & y & \wedge & \neg z &) & \end{array}$$

b Annotated Karnaugh map



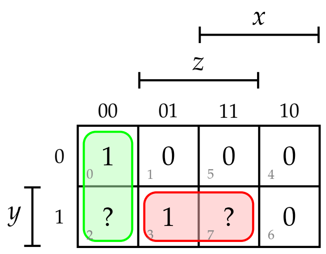
and associated, optimised implementation:

$$r = (\; y \;)$$

▷ **S117.** a Reference implementation:

$$\begin{array}{rcl} r & = & (\neg x \wedge \neg y \wedge \neg z) \vee \\ & & (\neg x \wedge y \wedge z) \end{array}$$

b Annotated Karnaugh map



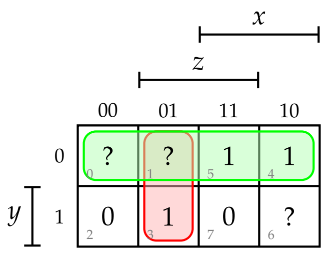
and associated, optimised implementation:

$$\begin{array}{rcl} r & = & (y \wedge z) \vee \\ & & (\neg x \wedge \neg z) \end{array}$$

▷ **S118.** a Reference implementation:

$$\begin{array}{rcl} r & = & (\neg x \wedge y \wedge z) \vee \\ & & (x \wedge \neg y \wedge \neg z) \vee \\ & & (x \wedge \neg y \wedge z) \end{array}$$

b Annotated Karnaugh map



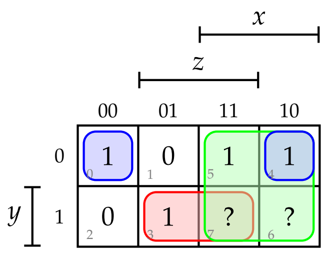
and associated, optimised implementation:

$$\begin{array}{rcl} r & = & (\neg x \wedge z) \vee \\ & & (\neg y) \end{array}$$

▷ **S119.** a Reference implementation:

$$\begin{array}{rcl} r & = & (\neg x \wedge \neg y \wedge \neg z) \vee \\ & & (\neg x \wedge y \wedge z) \vee \\ & & (x \wedge \neg y \wedge \neg z) \vee \\ & & (x \wedge \neg y \wedge z) \end{array}$$

b Annotated Karnaugh map



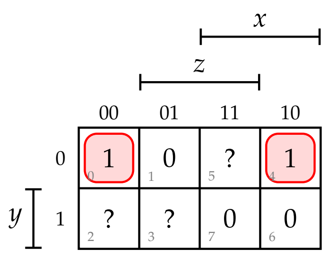
and associated, optimised implementation:

$$\begin{array}{rcl} r & = & (\quad y \wedge z) \vee \\ & & (x) \vee \\ & & (\neg y \wedge \neg z) \end{array}$$

▷ **S120.** a Reference implementation:

$$\begin{array}{rcl} r & = & (\neg x \wedge \neg y \wedge \neg z) \vee \\ & & (x \wedge \neg y \wedge \neg z) \end{array}$$

b Annotated Karnaugh map



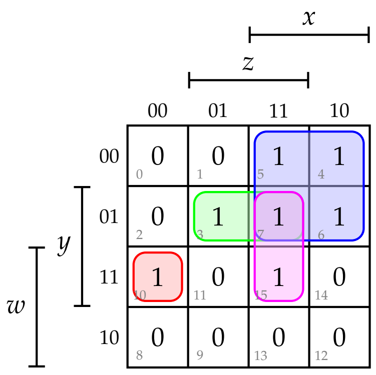
and associated, optimised implementation:

$$r = (\neg y \wedge \neg z)$$

▷ **S121.** a Reference implementation:

$$\begin{array}{rcl} r & = & (\neg w \wedge \neg x \wedge y \wedge z) \vee \\ & & (\neg w \wedge x \wedge \neg y \wedge \neg z) \vee \\ & & (\neg w \wedge x \wedge \neg y \wedge z) \vee \\ & & (\neg w \wedge x \wedge y \wedge \neg z) \vee \\ & & (\neg w \wedge x \wedge y \wedge z) \vee \\ & & (w \wedge \neg x \wedge y \wedge \neg z) \vee \\ & & (w \wedge x \wedge y \wedge z) \end{array}$$

b Annotated Karnaugh map



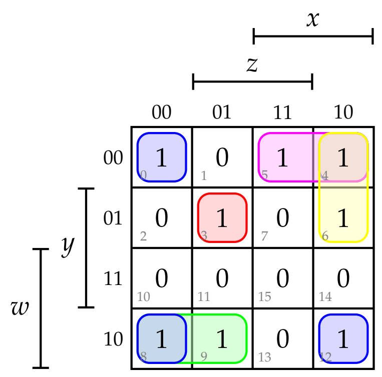
and associated, optimised implementation:

$$\begin{array}{rcl} r & = & (w \wedge \neg x \wedge y \wedge \neg z) \vee \\ & & (\neg w \wedge y \wedge z) \vee \\ & & (\neg w \wedge x) \vee \\ & & (x \wedge y \wedge z) \end{array}$$

▷ **S122.** a Reference implementation:

$$\begin{array}{rcl} r & = & (\neg w \wedge \neg x \wedge \neg y \wedge \neg z) \vee \\ & & (\neg w \wedge \neg x \wedge y \wedge z) \vee \\ & & (\neg w \wedge x \wedge \neg y \wedge \neg z) \vee \\ & & (\neg w \wedge x \wedge \neg y \wedge z) \vee \\ & & (\neg w \wedge x \wedge y \wedge \neg z) \vee \\ & & (w \wedge \neg x \wedge \neg y \wedge \neg z) \vee \\ & & (w \wedge \neg x \wedge \neg y \wedge z) \vee \\ & & (w \wedge x \wedge \neg y \wedge \neg z) \end{array}$$

b Annotated Karnaugh map



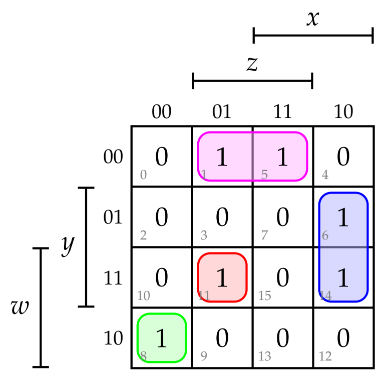
and associated, optimised implementation:

$$\begin{array}{ccccccccccccc} r & = & (& \neg w & \wedge & \neg x & \wedge & y & \wedge & z &) & \vee \\ & & (& w & \wedge & \neg x & \wedge & \neg y & & &) & \vee \\ & & (& & & & & \neg y & \wedge & \neg z &) & \vee \\ & & (& \neg w & \wedge & x & \wedge & \neg y & & &) & \vee \\ & & (& \neg w & \wedge & x & & & \wedge & \neg z &) & \end{array}$$

▷ **S123.** a Reference implementation:

$$\begin{array}{ccccccccccccc} r & = & (& \neg w & \wedge & \neg x & \wedge & \neg y & \wedge & z &) & \vee \\ & & (& \neg w & \wedge & x & \wedge & \neg y & \wedge & z &) & \vee \\ & & (& \neg w & \wedge & x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& w & \wedge & \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& w & \wedge & \neg x & \wedge & y & \wedge & z &) & \vee \\ & & (& w & \wedge & x & \wedge & y & \wedge & \neg z &) & \end{array}$$

b Annotated Karnaugh map



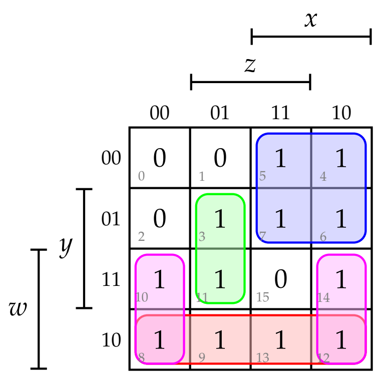
and associated, optimised implementation:

$$\begin{array}{ccccccccccccc} r & = & (& w & \wedge & \neg x & \wedge & y & \wedge & z &) & \vee \\ & & (& w & \wedge & \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& & & x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& \neg w & & & \wedge & \neg y & \wedge & z &) & \end{array}$$

▷ **S124.** a Reference implementation:

$$
\begin{array}{rcllllllllllll}
r & = & (& \neg w & \wedge & \neg x & \wedge & y & \wedge & z &) & \vee \\
  &   & (& \neg w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \vee \\
  &   & (& \neg w & \wedge & x & \wedge & \neg y & \wedge & z &) & \vee \\
  &   & (& \neg w & \wedge & x & \wedge & y & \wedge & \neg z &) & \vee \\
  &   & (& \neg w & \wedge & x & \wedge & y & \wedge & z &) & \vee \\
  &   & (& w & \wedge & \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\
  &   & (& w & \wedge & \neg x & \wedge & \neg y & \wedge & z &) & \vee \\
  &   & (& w & \wedge & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\
  &   & (& w & \wedge & \neg x & \wedge & y & \wedge & z &) & \vee \\
  &   & (& w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \vee \\
  &   & (& w & \wedge & x & \wedge & \neg y & \wedge & z &) & \vee \\
  &   & (& w & \wedge & x & \wedge & y & \wedge & \neg z &) &
\end{array}
$$

b Annotated Karnaugh map



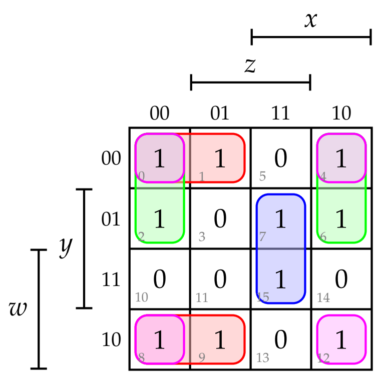
and associated, optimised implementation:

$$
\begin{array}{rcllllllllllll}
r & = & (& w & & & \wedge & \neg y & & &) & \vee \\
  &   & (& & & \neg x & \wedge & y & \wedge & z &) & \vee \\
  &   & (& \neg w & \wedge & x & & & & &) & \vee \\
  &   & (& w & & & & & \wedge & \neg z &) &
\end{array}
$$

▷ **S125.** a Reference implementation:

$$
\begin{array}{rcllllllllllll}
r & = & (& \neg w & \wedge & \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\
  &   & (& \neg w & \wedge & \neg x & \wedge & \neg y & \wedge & z &) & \vee \\
  &   & (& \neg w & \wedge & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\
  &   & (& \neg w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \vee \\
  &   & (& \neg w & \wedge & x & \wedge & y & \wedge & \neg z &) & \vee \\
  &   & (& \neg w & \wedge & x & \wedge & y & \wedge & z &) & \vee \\
  &   & (& w & \wedge & \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\
  &   & (& w & \wedge & \neg x & \wedge & \neg y & \wedge & z &) & \vee \\
  &   & (& w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \vee \\
  &   & (& w & \wedge & x & \wedge & y & \wedge & z &) &
\end{array}
$$

b Annotated Karnaugh map



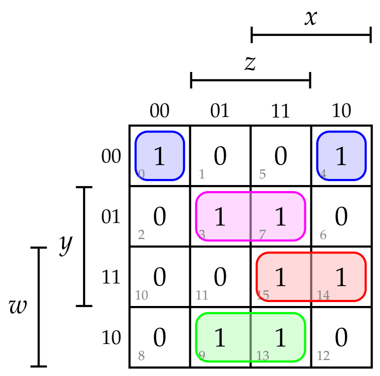
and associated, optimised implementation:

$$\begin{array}{rcllllllllll} r & = & (& & & \neg x & \wedge & \neg y & & &) & \vee \\ & & (& \neg w & & & & & \wedge & \neg z &) & \vee \\ & & (& & & x & \wedge & y & \wedge & z &) & \vee \\ & & (& & & & & \neg y & \wedge & \neg z &) & \end{array}$$

▷ **S126.** a Reference implementation:

$$\begin{array}{rcllllllllll} r & = & (& \neg w & \wedge & \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& \neg w & \wedge & \neg x & \wedge & y & \wedge & z &) & \vee \\ & & (& \neg w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& \neg w & \wedge & x & \wedge & y & \wedge & z &) & \vee \\ & & (& w & \wedge & \neg x & \wedge & \neg y & \wedge & z &) & \vee \\ & & (& w & \wedge & x & \wedge & \neg y & \wedge & z &) & \vee \\ & & (& w & \wedge & x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& w & \wedge & x & \wedge & y & \wedge & z &) & \end{array}$$

b Annotated Karnaugh map



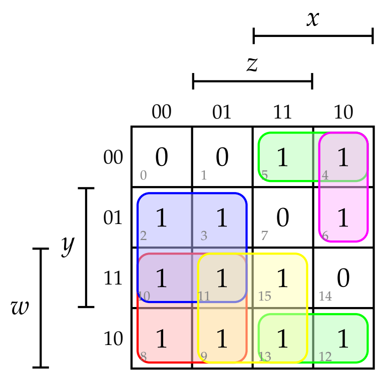
and associated, optimised implementation:

$$\begin{array}{rcllllllllll} r & = & (& w & \wedge & x & \wedge & y & & &) & \vee \\ & & (& w & & & \wedge & \neg y & \wedge & z &) & \vee \\ & & (& \neg w & & & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& \neg w & & & \wedge & y & \wedge & z &) & \end{array}$$

▷ **S127.** a Reference implementation:

$$\begin{array}{rcllllllllll} r & = & (& \neg w & \wedge & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& \neg w & \wedge & \neg x & \wedge & y & \wedge & z &) & \vee \\ & & (& \neg w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& \neg w & \wedge & x & \wedge & \neg y & \wedge & z &) & \vee \\ & & (& \neg w & \wedge & x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& w & \wedge & \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& w & \wedge & \neg x & \wedge & \neg y & \wedge & z &) & \vee \\ & & (& w & \wedge & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& w & \wedge & \neg x & \wedge & y & \wedge & z &) & \vee \\ & & (& w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& w & \wedge & x & \wedge & \neg y & \wedge & z &) & \vee \\ & & (& w & \wedge & x & \wedge & y & \wedge & z &) & \end{array}$$

b Annotated Karnaugh map



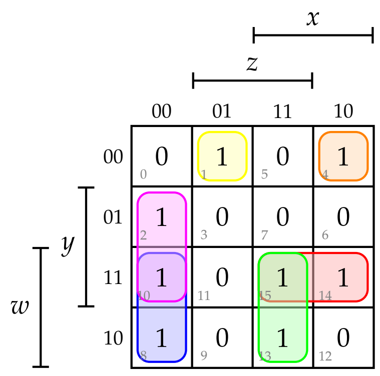
and associated, optimised implementation:

$$\begin{array}{lllllllllllll} r & = & (& w & \wedge & \neg x & & & & &) & \vee \\ & & (& & & x & \wedge & \neg y & & &) & \vee \\ & & (& & & \neg x & \wedge & y & & &) & \vee \\ & & (& \neg w & \wedge & x & & & \wedge & \neg z &) & \vee \\ & & (& w & & & & & \wedge & z &) & \end{array}$$

▷ **S128.** a Reference implementation:

$$\begin{array}{lllllllllllll} r & = & (& \neg w & \wedge & \neg x & \wedge & \neg y & \wedge & z &) & \vee \\ & & (& \neg w & \wedge & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& \neg w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& w & \wedge & \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& w & \wedge & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& w & \wedge & x & \wedge & \neg y & \wedge & z &) & \vee \\ & & (& w & \wedge & x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& w & \wedge & x & \wedge & y & \wedge & z &) & \end{array}$$

b Annotated Karnaugh map



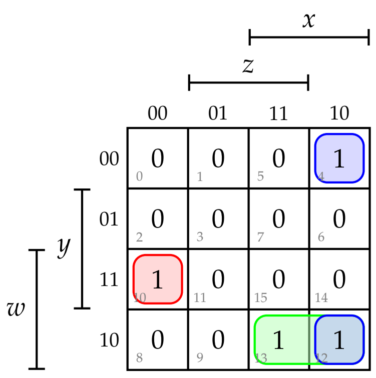
and associated, optimised implementation:

$$\begin{array}{lllllllllllll} r & = & (& w & \wedge & x & \wedge & y & & &) & \vee \\ & & (& w & \wedge & x & & & \wedge & z &) & \vee \\ & & (& w & \wedge & \neg x & & & \wedge & \neg z &) & \vee \\ & & (& & & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& \neg w & \wedge & \neg x & \wedge & \neg y & \wedge & z &) & \vee \\ & & (& \neg w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \end{array}$$

▷ **S129.** a Reference implementation:

$$\begin{array}{lllllllllllll} r & = & (& \neg w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& w & \wedge & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& w & \wedge & x & \wedge & \neg y & \wedge & z &) & \end{array}$$

b Annotated Karnaugh map



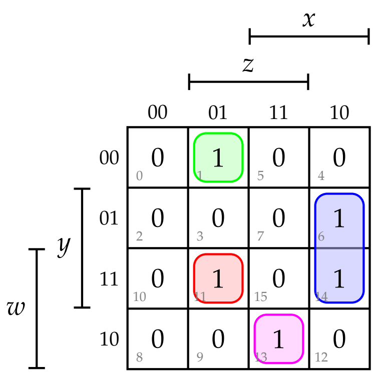
and associated, optimised implementation:

$$
\begin{array}{lclcccccccccl}
r & = & (& w & \wedge & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\
  &   & (& w & \wedge & x & \wedge & \neg y & & &) & \vee \\
  &   & (& & & x & \wedge & \neg y & \wedge & \neg z &) &
\end{array}
$$

▷ **S130.** a Reference implementation:

$$
\begin{array}{lclcccccccccl}
r & = & (& \neg w & \wedge & \neg x & \wedge & \neg y & \wedge & z &) & \vee \\
  &   & (& \neg w & \wedge & x & \wedge & y & \wedge & \neg z &) & \vee \\
  &   & (& w & \wedge & \neg x & \wedge & y & \wedge & z &) & \vee \\
  &   & (& w & \wedge & x & \wedge & \neg y & \wedge & z &) & \vee \\
  &   & (& w & \wedge & x & \wedge & y & \wedge & \neg z &) &
\end{array}
$$

b Annotated Karnaugh map



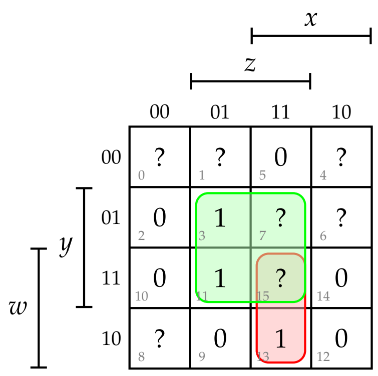
and associated, optimised implementation:

$$
\begin{array}{lclcccccccccl}
r & = & (& w & \wedge & \neg x & \wedge & y & \wedge & z &) & \vee \\
  &   & (& \neg w & \wedge & \neg x & \wedge & \neg y & \wedge & z &) & \vee \\
  &   & (& & & x & \wedge & y & \wedge & \neg z &) & \vee \\
  &   & (& w & \wedge & x & \wedge & \neg y & \wedge & z &) &
\end{array}
$$

▷ **S131.** a Reference implementation:

$$
\begin{array}{lclcccccccccl}
r & = & (& \neg w & \wedge & \neg x & \wedge & y & \wedge & z &) & \vee \\
  &   & (& w & \wedge & \neg x & \wedge & y & \wedge & z &) & \vee \\
  &   & (& w & \wedge & x & \wedge & \neg y & \wedge & z &) &
\end{array}
$$

b Annotated Karnaugh map



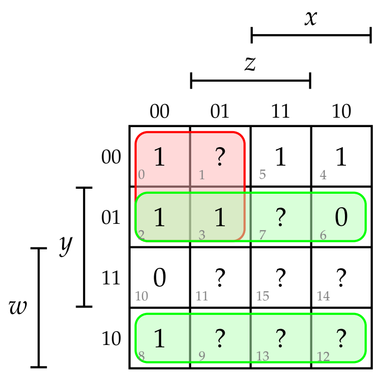
and associated, optimised implementation:

$$\begin{array}{rcl} r & = & (w \wedge x \wedge z) \vee \\ & & (y \wedge z) \end{array}$$

▷ **S132.** a Reference implementation:

$$\begin{array}{rcl} r & = & (\neg w \wedge \neg x \wedge \neg y \wedge \neg z) \vee \\ & & (\neg w \wedge \neg x \wedge y \wedge \neg z) \vee \\ & & (\neg w \wedge \neg x \wedge y \wedge z) \vee \\ & & (\neg w \wedge x \wedge \neg y \wedge \neg z) \vee \\ & & (\neg w \wedge x \wedge \neg y \wedge z) \vee \\ & & (w \wedge \neg x \wedge \neg y \wedge \neg z) \end{array}$$

b Annotated Karnaugh map



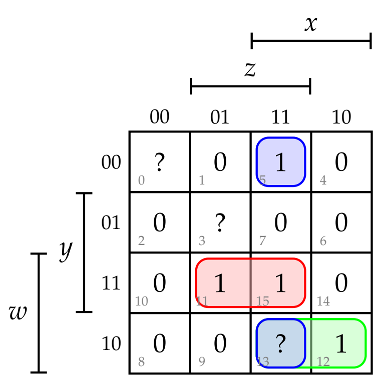
and associated, optimised implementation:

$$\begin{array}{rcl} r & = & (\neg w \wedge \neg x) \vee \\ & & (\neg y) \end{array}$$

▷ **S133.** a Reference implementation:

$$\begin{array}{rcl} r & = & (\neg w \wedge x \wedge \neg y \wedge z) \vee \\ & & (w \wedge \neg x \wedge y \wedge z) \vee \\ & & (w \wedge x \wedge \neg y \wedge \neg z) \vee \\ & & (w \wedge x \wedge y \wedge z) \end{array}$$

b Annotated Karnaugh map



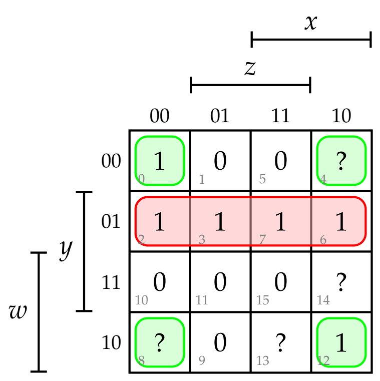
and associated, optimised implementation:

$$\begin{array}{lllllllllll} r & = & (& w & & & \wedge & y & \wedge & z &) & \vee \\ & & (& w & \wedge & x & \wedge & \neg y & & &) & \vee \\ & & (& & & x & \wedge & \neg y & \wedge & z &) & \end{array}$$

▷ **S134.** a Reference implementation:

$$\begin{array}{lllllllllll} r & = & (& \neg w & \wedge & \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& \neg w & \wedge & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& \neg w & \wedge & \neg x & \wedge & y & \wedge & z &) & \vee \\ & & (& \neg w & \wedge & x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& \neg w & \wedge & x & \wedge & y & \wedge & z &) & \vee \\ & & (& w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \end{array}$$

b Annotated Karnaugh map



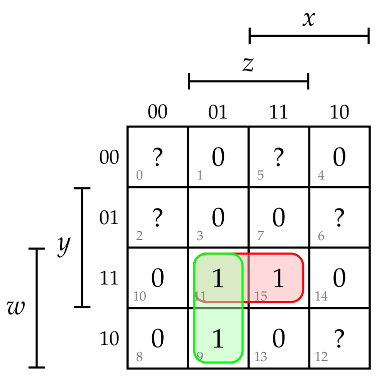
and associated, optimised implementation:

$$\begin{array}{lllllllll} r & = & (& \neg w & \wedge & y & & &) & \vee \\ & & (& & & \neg y & \wedge & \neg z &) & \end{array}$$

▷ **S135.** a Reference implementation:

$$\begin{array}{lllllllllll} r & = & (& w & \wedge & \neg x & \wedge & \neg y & \wedge & z &) & \vee \\ & & (& w & \wedge & \neg x & \wedge & y & \wedge & z &) & \vee \\ & & (& w & \wedge & x & \wedge & y & \wedge & z &) & \end{array}$$

b Annotated Karnaugh map



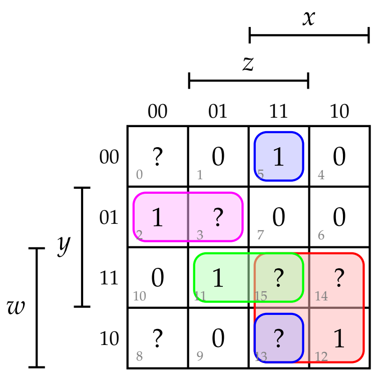
and associated, optimised implementation:

$$
\begin{array}{rcccccccccccc}
r & = & (& w & & & \wedge & y & \wedge & z &) & \vee \\
 & & (& w & \wedge & \neg x & & & \wedge & z &) &
\end{array}
$$

▷ **S136.** a Reference implementation:

$$
\begin{array}{rcccccccccccc}
r & = & (& \neg w & \wedge & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\
 & & (& \neg w & \wedge & x & \wedge & \neg y & \wedge & z &) & \vee \\
 & & (& w & \wedge & \neg x & \wedge & y & \wedge & z &) & \vee \\
 & & (& w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) &
\end{array}
$$

b Annotated Karnaugh map



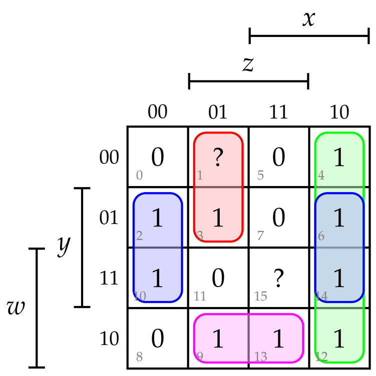
and associated, optimised implementation:

$$
\begin{array}{rcccccccccccc}
r & = & (& w & \wedge & x & & & & &) & \vee \\
 & & (& w & & & \wedge & y & \wedge & z &) & \vee \\
 & & (& & & x & \wedge & \neg y & \wedge & z &) & \vee \\
 & & (& \neg w & \wedge & \neg x & \wedge & y & & &) &
\end{array}
$$

▷ **S137.** a Reference implementation:

$$
\begin{array}{rcccccccccccc}
r & = & (& \neg w & \wedge & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\
 & & (& \neg w & \wedge & \neg x & \wedge & y & \wedge & z &) & \vee \\
 & & (& \neg w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \vee \\
 & & (& \neg w & \wedge & x & \wedge & y & \wedge & \neg z &) & \vee \\
 & & (& w & \wedge & \neg x & \wedge & \neg y & \wedge & z &) & \vee \\
 & & (& w & \wedge & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\
 & & (& w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \vee \\
 & & (& w & \wedge & x & \wedge & \neg y & \wedge & z &) & \vee \\
 & & (& w & \wedge & x & \wedge & y & \wedge & \neg z &) &
\end{array}
$$

b Annotated Karnaugh map



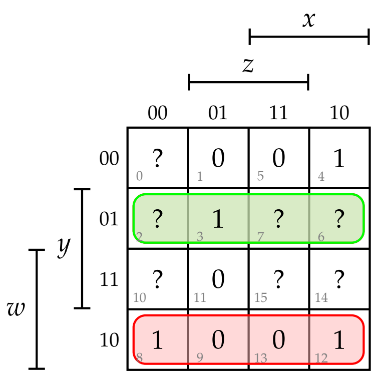
and associated, optimised implementation:

$$\begin{array}{llcccccccccccc}
r & = & (& \neg w & \wedge & \neg x & & & \wedge & z &) & \vee \\
 & & (& & & x & & & \wedge & \neg z &) & \vee \\
 & & (& & & & & y & \wedge & \neg z &) & \vee \\
 & & (& w & & & \wedge & \neg y & \wedge & z &) &
\end{array}$$

▷ **S138.** a Reference implementation:

$$\begin{array}{llcccccccccccc}
r & = & (& \neg w & \wedge & \neg x & \wedge & y & \wedge & z &) & \vee \\
 & & (& \neg w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \vee \\
 & & (& w & \wedge & \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\
 & & (& w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) &
\end{array}$$

b Annotated Karnaugh map



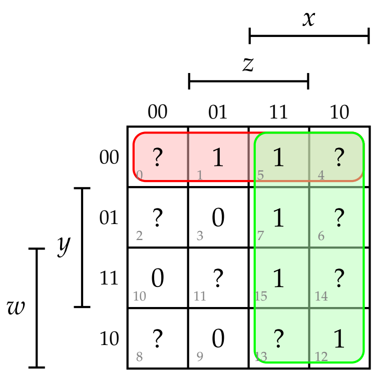
and associated, optimised implementation:

$$\begin{array}{llcccccc}
r & = & (& & & \neg z &) & \vee \\
 & & (& \neg w & \wedge & y &) &
\end{array}$$

▷ **S139.** a Reference implementation:

$$\begin{array}{llcccccccccccc}
r & = & (& \neg w & \wedge & \neg x & \wedge & \neg y & \wedge & z &) & \vee \\
 & & (& \neg w & \wedge & x & \wedge & \neg y & \wedge & z &) & \vee \\
 & & (& \neg w & \wedge & x & \wedge & y & \wedge & z &) & \vee \\
 & & (& w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \vee \\
 & & (& w & \wedge & x & \wedge & y & \wedge & z &) &
\end{array}$$

b Annotated Karnaugh map



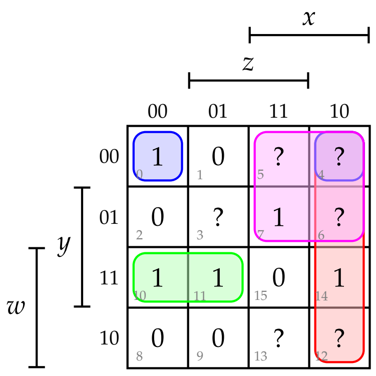
and associated, optimised implementation:

$$\begin{array}{rcl} r & = & (\neg w \wedge \neg y) \vee \\ & & (x) \end{array}$$

▷ **S140.** a Reference implementation:

$$\begin{array}{rcl} r & = & (\neg w \wedge \neg x \wedge \neg y \wedge \neg z) \vee \\ & & (\neg w \wedge x \wedge y \wedge z) \vee \\ & & (w \wedge \neg x \wedge y \wedge \neg z) \vee \\ & & (w \wedge \neg x \wedge y \wedge z) \vee \\ & & (w \wedge x \wedge y \wedge \neg z) \end{array}$$

b Annotated Karnaugh map



and associated, optimised implementation:

$$\begin{array}{rcl} r & = & (x \wedge \neg z) \vee \\ & & (w \wedge \neg x \wedge y) \vee \\ & & (\neg w \wedge \neg y \wedge \neg z) \vee \\ & & (\neg w \wedge x) \end{array}$$

Part IV: Basics of computer arithmetic

▷ **S141.** First, notice that the function does not use y at all, so the function cannot add x to y. All other options are plausible: to assess which is correct, one could take a brute-force approach and execute it via

```
int main( int argc, char* argv[] ) {
  for( int i = 0; i < 256; i++ ) {
    printf( "%3d %3d\n", i, f( i ) );
  }

  return 0;
}
```

Doing so shows that that the function increments x. But why is this? Consider some specific examples:

- For x $= 14_{(10)} = 00001110_{(2)}$, the loop performs 0 iterations then terminates: the value

$$\begin{array}{lcl} \texttt{x m |} & = & 00001110_{(2)} \vee 00000001_{(2)} \\ & = & 00001111_{(2)} \\ & = & 15_{(10)} \end{array}$$

  i.e., x + 1 is returned.
- For x $= 7_{(10)} = 00000111_{(2)}$, the loop performs 3 iterations

$$\begin{array}{lclcl} \texttt{x} & = & \texttt{x \& \textasciitilde m} & = & 00000111_{(2)} \wedge \neg 00000001_{(2)} \\ & & & = & 00000111_{(2)} \wedge 11111110_{(2)} \\ & & & = & 00000110_{(2)} \\ \texttt{m} & = & \texttt{m << 1} & = & 00000001_{(2)} \ll 1 \\ & & & = & 00000010_{(2)} \\ \\ \texttt{x} & = & \texttt{x \& \textasciitilde m} & = & 00000110_{(2)} \wedge \neg 00000010_{(2)} \\ & & & = & 00000110_{(2)} \wedge 11111101_{(2)} \\ & & & = & 00000100_{(2)} \\ \texttt{m} & = & \texttt{m << 1} & = & 00000010_{(2)} \ll 1 \\ & & & = & 00000100_{(2)} \\ \\ \texttt{x} & = & \texttt{x \& \textasciitilde m} & = & 00000100_{(2)} \wedge \neg 00000100_{(2)} \\ & & & = & 00000100_{(2)} \wedge 11111011_{(2)} \\ & & & = & 00000000_{(2)} \\ \texttt{m} & = & \texttt{m << 1} & = & 00000100_{(2)} \ll 1 \\ & & & = & 00001000_{(2)} \end{array}$$

  then terminates: the value

$$\begin{array}{lcl} \texttt{x m |} & = & 00000000_{(2)} \vee 00001000_{(2)} \\ & = & 00001000_{(2)} \\ & = & 16_{(10)} \end{array}$$

  i.e., x + 1 is returned.

More generally, the idea is that, as a result of initialising m to 1 and then left-shifting it in each iteration, the `while` loop iterates over each LSB of x which is 1; while doing so, x is assigned the value `x & ~m` which toggles the current bit. Once the loop terminates, the value x m | is returned: this sets the current bit of x, which we know is 0 due to the loop condition, to 1.

▷ **S142.**
- `x == 0` evaluates to non-zero if x = 0, i.e., every bit of x is 0. `x == -1` evaluates to non-zero if x $= -1 \equiv 2^{32} - 1$, i.e., every bit of x is 1: although x is unsigned, the signed value -1 will "wrap around" to the representation

$$\texttt{x} \mapsto -2^{31} + 2^{30} \cdots + 2^{1} + 2^{0},$$

  i.e., where every bit of x is 1. `( x == 0 )` | (x == -1) | therefore yields the required behaviour.
- `!x` evaluates to non-zero if x = 0, i.e., every bit of x is 0. `!(~x)` evaluates to non-zero if x $= -1 \equiv 2^{32} - 1$, i.e., every bit of x is 1: if every bit of x is 1, `~x` means every bit is 0 and thus `!(~x)` evaluates to non-zero in exactly that case. !x | !(x) | therefore yields the required behaviour.
- This option is arguably trickier, in the sense there is only one term; it is therefore harder to see how it captures the two cases. To see why it does, you could apply sort of the opposite reasoning to the above. If every bit of x is 0, then x = 0 and so `x+1 = 1`; `( x + 1 ) < 2` evaluates to non-zero in this case. If every bit of x is 1, then x $= -1 \equiv 2^{32} - 1$ and so `x+1 = 0`; `( x + 1 ) < 2` evaluates to non-zero in this case. Given x is unsigned, all other representations it could take imply $1 \leq$ x $\leq 2^{32} - 2$ and so $2 \leq$ `( x + 1 ) < 2` $\leq 2^{32} - 1$; in such cases we conclude that `( x + 1 ) > 2` evaluates to zero.

▷ **S143.** Although one *could* select the correct option by inspection, the easiest approach is simply work through each one. Doing that via a complete trace (e.g., of each intermediate computation, by each full-adder instance) is overly verbose, so in the below, we capture the pertinent details only (noting the sequence representing the carry-chain is read left-to-right; this matches the ripple-carry diagram, but might seem odd wrt. the right-to-left order of digits in the literals):

$$\begin{array}{lclcr} x &=& 0000_{(2)} &=& 0_{(10)} \\ y &=& 0000_{(2)} &=& 0_{(10)} \\ r &=& 0000_{(2)} &=& 0_{(10)} \\ c &=& \langle 0,0,0,0,0 \rangle && \\ c_2 &=& 0 && \end{array}$$

$$\begin{array}{lclcr} x &=& 1100_{(2)} &=& 12_{(10)} \\ y &=& 0001_{(2)} &=& 1_{(10)} \\ r &=& 1101_{(2)} &=& 13_{(10)} \\ c &=& \langle 0,0,0,0,0 \rangle && \\ c_2 &=& 0 && \end{array}$$

$$\begin{array}{lclcr} x &=& 0100_{(2)} &=& 4_{(10)} \\ y &=& 0100_{(2)} &=& 4_{(10)} \\ r &=& 1000_{(2)} &=& 8_{(10)} \\ c &=& \langle 0,0,0,1,0 \rangle && \\ c_2 &=& 0 && \end{array}$$

$$\begin{array}{lclcr} x &=& 1011_{(2)} &=& 11_{(10)} \\ y &=& 1001_{(2)} &=& 9_{(10)} \\ r &=& 0100_{(2)} &=& 4_{(10)} \\ c &=& \langle 0,1,1,0,1 \rangle && \\ c_2 &=& 1 && \end{array}$$

$$\begin{array}{lclcr} x &=& 0110_{(2)} &=& 6_{(10)} \\ y &=& 0101_{(2)} &=& 5_{(10)} \\ r &=& 1011_{(2)} &=& 11_{(10)} \\ c &=& \langle 0,0,0,1,0 \rangle && \\ c_2 &=& 0 && \end{array}$$

▷ **S144.** In general, an overflow condition occurs when the correct result of some arithmetic operation cannot be represented (meaning the result is then incorrect). Within the context outlined by the question, two instances of this can occur: either 1) x is positive and y is positive, but r is negative, or 2) x is negative and y is negative, but r is positive. In both instances, the sign of r is incorrect because the correct value of r has too large a magnitude to represent in 8 bits.

In two's-complement the MSB indicates the sign, meaning that x_7, y_7, and r_7 indicate whether x, y, and r are positive or negative respectively. Using this information we can translate each condition above into a Boolean expression, i.e.,

$$x_7 \wedge y_7 \wedge \neg r_7$$

indicates that x is positive, y is positive, and r is negative, while

$$\neg x_7 \wedge \neg y_7 \wedge r_7$$

indicates that x is negative, y is negative, and r is positive. As such, simply OR'ing these expressions together produces the flag we require, i.e.,

$$f = (x_7 \wedge y_7 \wedge \neg r_7) \vee (\neg x_7 \wedge \neg y_7 \wedge r_7).$$

▷ **S145.** a **true**. Consider some w-bit word x. A logical right-shift (resp. left-shift) of x will fill the MSBs (resp. LSBs) with 0, whereas an arithmetic right-shift will fill the MSBs with x_{w-1} (i.e., the existing MSB). The rationale for the latter is that if x is interpreted using a signed representation such as two's-complement, it will retain the sign: if x represents a positive (resp. negative) value, that value is *still* positive (resp. negative) after the shift is applied. Based on this, an arithmetic left-shift is not useful because the LSB does not influence sign in the same way as the MSB.

b **true**. A right-*shift* by 2 bits would move all bits in `x` 2 places "downward" toward the least-significant end; in doing so the 2 least-significant bits are discarded, and two 0 bits are introduced to the most-significant end (because `x` is unsigned in this case). The action of a right-*rotate* can be described as a "circular right-shift", in the sense that those discarded bits are now moved from the least- back around to the most-significant end.

To match the left/right semantics of the shift operators, imagine that we express the bits in `x` (and also the intermediate values) as follows

`x`	=	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
`x >> 2`	=	0	0	x_7	x_6	x_5	x_4	x_3	x_2
`x << 6`	=	x_1	x_0	0	0	0	0	0	0
`( x >> 2 ) ( x « 6 ) |`	=	x_1	x_0	x_7	x_6	x_5	x_4	x_3	x_2

First, right-shifting `x` by 2 bits discards x_1 and x_0 while retaining x_7 to x_2 in the 6 least-significant bits. Second, left-shifting `x` by 6 bits discards x_7 to x_2 while retaining x_1 and x_0 in the 2 most-significant bits. Noting that $t \vee 0 = t$, third, and finally, OR'ing the right- and left-shifted `x` retains the same values of each bit in `x` but now in a different order: that order matches our description of right-rotation above.

▷ **S146.** $O(n)$ implies the critical path is proportional to the number of bits (including some constant factor) required to represent each of the operands. The reason is the carry chain which runs through all n full-adders in the design: each i-th full-adder produces a carry-out used as a carry-into the $(i + 1)$-th full-adder. This means each i-th bit of the result depends on, and cannot be computed before, all j-th bits for $0 \leq j < i$.

An alternative, carry look-ahead design separates computation of carries from the full-adder cells themselves; this allows an organisation whose critical path can be described as $O(\log n)$, although the number of logic gates required is less attractive.

▷ **S147.** The relationship

$$x \text{ is a power-of-two} \equiv (x \wedge (x - 1)) = 0$$

performs the test which can be written as the C expression

```
( x & ( x - 1 ) ) == 0
```

This works because if x is an exact power-of-two then $x - 1$ sets all bits less-significant that the n-th to one; when this is AND'ed with x (which only has the n-th bit set to one) the result is zero. If x is not an exact power-of-two then there will be bits in x other than the n-th set to one; in this case $x - 1$ only sets bits less-significant than the least-significant and hence there are others left over which, when AND'ed with x, result in a non-zero result. Note that the expression fails, i.e., it is non-zero, for $x = 0$ but this is allowed since the question says $x \neq 0$.

▷ **S148.** `x` is of type `char`, so is therefore represented using two's-complement in 8 bits; values for such a representation range between $2^{n-1} - 1 = 2^{8-1} - 1 = 127$ and $-2^{n-1} = -2^{8-1} = -128$ inclusive. This means that by

a decrementing `x` we get the value before 127, which is 126, or

b incrementing `x` we get the value after 127, which is -128: the reason for this is that the representation of 127 is $01111111_{(2)}$, but the next value $10000000_{(2)}$ is the largest negative value possible. That is, there has been an overflow with the result "wrapping around".

▷ **S149.** The expression computes the comparison $0 < x$. This is because if $x < 0$ then $x_3 = 1$, and if $x = 0$ then $x_3 = x_2 = x_1 = x_0 = 0$. Therefore, $x > 0$ if both $x_3 = 0$ and one of $x_i \neq 0$ for $i \in \{2, 1, 0\}$. Strictly speaking, it tests whether $0 < x \leq 7$ but the upper bound is implied by the representation of x: it cannot take a value greater than 7 by definition.

▷ **S150.**

The initial temptation is to use six adder components to compute

$$r = 7 \cdot x = x + x + x + x + x + x + x$$

where the size of inputs and outputs increases as one progresses through the computation; a considered approach might utilise carry save adders to reduce the critical path associated with the multiple summands, but here we consider ripple-carry designs only.

A more efficient alternative would use three adders to compute

$$r = 7 \cdot x = 4 \cdot x + 2 \cdot x + 1 \cdot x = 2^2 \cdot x + 2^1 \cdot x + 2^0 \cdot x$$

noting that the multiplications by powers-of-two are "free" since they can be achieved by simply relabelling bits rather than computation. This approach can be further refined to compute

$$r = 7 \cdot x = 8 \cdot x - 1 \cdot x = 2^3 \cdot x - 2^0 \cdot x$$

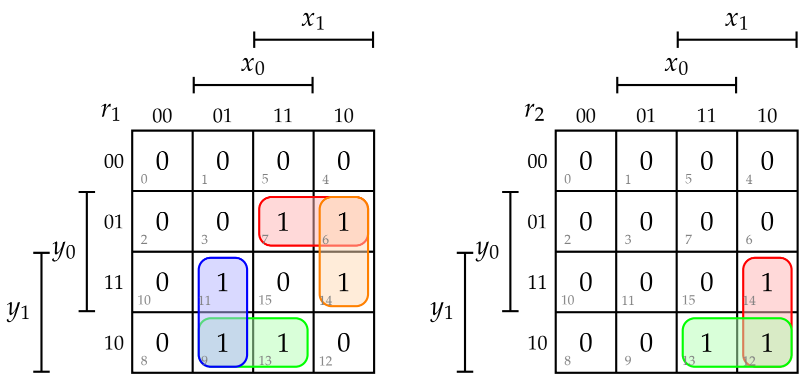
using just one adder (assuming addition and subtraction can be realised using the same component). Clearly this will produce the shortest critical path, and relates to the following diagram:

```
                +-------------------+                +---+
x -- n-bit -->| 3-bit left-shift |-- (n+3)-bit -->|   |
                +-------------------+                |sub|-- (n+4)-bit --> r
x -- n-bit ----------------------------------------->|   |
                                                     +---+
```

▷ **S151.** The truth table for this operation is as follows:

x_1	x_0	y_1	y_0	r_3	r_2	r_1	r_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

Using four Karnaugh maps to produce each r_i is overkill, since we can easily derive expressions for r_0 and r_3 by inspection. Therefore, transcribing the truth table into suitable Karnaugh maps for just r_1 and r_2 gives



from which we can derive the SoP expressions

$$
\begin{array}{rcl}
r_0 & = & (x_0 \wedge y_0) \\
r_1 & = & (x_1 \wedge \neg y_1 \wedge y_0) \vee \\
 & & (x_0 \wedge y_1 \wedge \neg y_0) \vee \\
 & & (\neg x_1 \wedge x_0 \wedge y_1) \vee \\
 & & (x_1 \wedge \neg x_0 \wedge y_0) \\
r_2 & = & (x_1 \wedge \neg x_0 \wedge y_1) \vee \\
 & & (x_1 \wedge y_1 \wedge \neg y_0) \\
r_3 & = & (x_1 \wedge x_0 \wedge y_1 \wedge y_0)
\end{array}
$$

▷ **S152.** a Clearly we can implement $\neq$ by negating the result of $=$ and likewise for $<$ and $\geq$, and $>$ and $\leq$. Furthermore, we can build $\geq$ from $>$ and $=$, and $\leq$ from $<$ and $=$. So essentially we only need two comparisons, say $=$ and $<$ to be able to compute the rest so long as we have the logic operations as well. The choice of which three is simply a matter of which ones you want to go faster: the ones built from a combination of other comparison and logic instructions will take longer to execute. One might take the approach of looking at C programs and selecting the set most used. For example $=$ and $<$ are used a lot to program typical loops; one might select them for this reason.

b You can be as fancy as you want with any optimisations or special cases, for example checking for multiplication by zero, one or a power-of-two might be a good idea. But basically, the easiest way to do this is as follows:

```
uint16_t mul( uint16_t x, uint16_t y ) {
  switch( x ) {
    case  0 : return 0;
    case  1 : return y;
    case  2 : return y << 1;
    case  4 : return y << 2;
    case  8 : return y << 3;
    case 16 : return y << 4;
  }
  switch( y ) {
    case  0 : return 0;
    case  1 : return x;
    case  2 : return x << 1;
    case  4 : return x << 2;
    case  8 : return x << 3;
    case 16 : return x << 4;
  }

  uint16_t t = 0;

  for( int i = 15; i >= 0; i-- ) {
    t = t << 1;

    if( ( y >> i ) & 1 ) {
      t = t + x;
    }
  }

  return t;
}
```

c A basic implementation might look like the following:

```
int H( uint16_t x ) {
  int t = 0;

  for( int i = 0; i < 16; i++ ) {
    if( ( x >> i ) & 1 ) {
      t = t + 1;
    }
  }

  return t;
}
```

but this has a number of drawbacks. First, the overhead of of operating the loop quite high in comparison to the content; for example the loop body needs only a few instructions, while it takes nearly as many again to test and increment i during each iteration. Second, the number of branches in the code means that pipelined processors might not execute them efficiently at all. An improvement is to use some form of divide-and-conquer approach where we split the problem into 2-bit then 4-bit chunks and so on. The result might look like:

```
int H( uint16_t x ) {
  x = ( x & 0x5555 ) + ( ( x >> 1 ) & 0x5555 );
  x = ( x & 0x3333 ) + ( ( x >> 2 ) & 0x3333 );
  x = ( x & 0x0F0F ) + ( ( x >> 4 ) & 0x0F0F );
  x = ( x & 0x00FF ) + ( ( x >> 8 ) & 0x00FF );

  return ( int )( x );
}
```

▷ **S153.** First, note that the result via a naive method would be

$$\begin{array}{rcl} r_2 & = & x_1 \cdot y_1 \\ r_1 & = & x_1 \cdot y_0 + x_0 \cdot y_1 \\ r_0 & = & x_0 \cdot y_0. \end{array}$$

However, we can write down three intermediate values using only three multiplications as

$$\begin{array}{rcl}
t_2 &=& x_1 \cdot y_1 \\
t_1 &=& (x_0 + x_1) \cdot (y_0 + y_1) \\
t_0 &=& x_0 \cdot y_0 .
\end{array}$$

The original result can then be expressed in terms of these intermediate values via

$$\begin{array}{rclcl}
r_2 &=& t_2 &=& x_1 \cdot y_1 \\
r_1 &=& t_1 - t_0 - t_2 &=& x_0 \cdot y_0 + x_0 \cdot y_1 + x_1 \cdot y_0 + x_1 \cdot y_1 - x_0 \cdot y_0 - x_1 \cdot y_1 \\
 & & &=& x_1 \cdot y_0 + x_0 \cdot y_1 \\
r_0 &=& t_0 &=& x_0 \cdot y_0 .
\end{array}$$

So roughly speaking, over all we use three $(n/2)$-bit multiplications and four $(n/2)$-bit additions.

▷ **S154.** a In binary, the addition we are looking at is

$$\begin{array}{rcrl}
10_{(10)} &=& 1010_{(2)} & \\
12_{(10)} &=& \underline{1100_{(2)}} & + \\
 & & 10110_{(2)} &
\end{array}$$

where $10110_{(2)} = 22_{(10)}$. In 4 bits this value is $0110_{(2)} = 6_{(10)}$ however, which is wrong.

b A design for the 4-bit clamped adder looks like this:

```
      +-------+            +-------+            +-------+            +-------+
+-----|co   ci|<-----------|co   ci|<-----------|co   ci|<-----------|co   ci|<-   0
|     |      x|<- x_3      |      x|<- x_2      |      x|<- x_1      |      x|<- x_0
|  +--|s     y|<- y_3   +--|s     y|<- y_2   +--|s     y|<- y_1   +--|s     y|<- y_0
|  |  +-------+         |  +-------+         |  +-------+         |  +-------+
|  |                    |                    |                    |
|  |  +-------+         |  +-------+         |  +-------+         |  +-------+
|  +->|  OR   |-> r_3   +->|  OR   |-> r_2   +->|  OR   |-> r_1   +->|  OR   |-> r_0
|     +-------+            +-------+            +-------+            +-------+
|          |                    |                    |                    |
+----------+--------------------+--------------------+--------------------+
```

Essentially the idea is that if a carry-out occurs from the most-significant adder, this turns all the output bits to 1 via the additional OR gates. That is, if the carry-out occurs then we get $1111_{(2)} = 15_{(10)}$ as the result, i.e., the largest 4-bit result possible.

▷ **S155.** Since we know nothing about N, there is no obvious short-cut to performing the modular reduction after the multiplication. Instead, the most simple way to approach the design is to recall that

$$x \cdot y = \underbrace{x + x + \cdots + x + x}_{y \text{ copies}} .$$

So to compute $x \cdot y \pmod{N}$, we just have to make sure that each of the additions is modulo N; then we can use whatever method we want. A circuit for modular addition is actually quite simple:

```
    +------+                +------+
x ->|      |---+--------------->|      |
    | add  |   |                | sub  |---> r
y ->|      |   |            +-->|      |
    +------+   |            |   +------+
               v            |
            +------+    +------+
            |      |    |      |
        N ->| lth  |--->| mux  |
            |      |    |      |
            +------+    +------+
                         ^    ^
                         |    |
                         N    0
```

In short, we add x and y together, and then compare the result t with N: if t is smaller, we select 0 as the output from the multiplexer otherwise we select N. Then, we subtract the value we selected from t. The end result is that we get $x + y - 0 = x + y \pmod N$ if $x + y < N$, and $x + y - N = x + y \pmod N$ if $x + y \geq N$.

Recall that an 8-bit, bit-serial multiplier would compute the product $x \cdot y$ as follows:

Input: An 8-bit multiplicand x, and 8-bit multiplier y
Output: The product $x \cdot y$

```
1 $t \leftarrow 0$
2 for $i = 7$ downto $0$ do
3     $t \leftarrow t + t$
4     if $y_i = 1$ then
5         $t \leftarrow t + x$
6     end
7 end
8 return $t$
```

Armed with our modular adder circuit, we can rewrite this as

Input: An 8-bit modulus N, a multiplier $0 \leq x < N$ and multiplicand $0 \leq y < N$
Output: The product $x \cdot y \pmod N$

```
1 $t \leftarrow 0$
2 for $i = 7$ downto $0$ do
3     $t \leftarrow t + t \pmod N$
4     if $y_i = 1$ then
5         $t \leftarrow t + x \pmod N$
6     end
7 end
8 return $t$
```

which then simply demand eight iterations, under control of a clock, over the circuit

```
    +---------+       +---------+        +---------+
N ->|         |   N ->|         |    N ->|         |
t ->| mod add |   x ->| mod add |------->|   mux   |---> t'
t ->|         |---+-->|         |   +--->|         |
    +---------+   |   +---------+   |    +---------+
                  |                 |         ^
                  +-----------------+         |
                                             y_i
```

Notice that we first perform the operation $t + t \pmod N$, then use a multiplexer to decide if we take $t + t \pmod N$ or $t + t + x \pmod N$ as the next value of t. So each iterated use of the circuit represents an iteration of the algorithm loop. Of course, one could construct a combinatorial multiplier using the same approach, i.e., replacing any standard adder circuits with modular alternatives.

Part V: Basics of memory technology

▷ **S156.** Various clues should (in combination) be strong enough to hint that

$$\begin{array}{lcl} \alpha & \mapsto & \text{row address buffer} \\ \beta & \mapsto & \text{column address buffer} \\ \gamma & \mapsto & \text{row address decoder} \\ \delta & \mapsto & \text{column address decoder} \end{array}$$

is the correct answer. For example, note that:

- α is provided input from A_i (the address pins), and is controlled (indirectly) by RAS: this is the row address strobe. As such, this is likely to be the row address buffer.
- β is provided input from A_i (the address pins), and is controlled (indirectly) by CAS: this is the column address strobe. As such, this is likely to be the column address buffer.
- γ is taking the content of α and controlling signals on the left-hand side (horizontal orientation) of the memory array, suggesting it computes the row address: it is likely to be the row address decoder.
- δ is taking the content of β and controlling signals on the top side (vertical orientation) of the memory array, suggesting it computes the column address: it is likely to be the column address decoder.

▷ **S157.** SRAMs have a lower access latency in part because of their design: by using only transistors means their operation is very fast. Therefore, the first statement is true. On the other hand, SRAMs are larger than DRAMs since their design includes more components (typically six or so transistors versus one transistor and a capacitor); as a result, their density (i.e., how many one can fit into unit area) is lower, and the second statement is true as well. The third statement is false, and basically nonsense: the access latency should not depend on the order. Finally, the forth statement is also false. Rather, a stored program or von Neumann architecture holds both instructions and data in the same memory: a Harvard architecture segregates them into separate memories.

▷ **S158.** There are 2^{16} addressable bytes, meaning a 16-bit address needs to be supplied. However, in contrast to an SRAM memory, a DRAM memory will normally use a 2-step (or more, potentially) approach: half the address is supplied by each of the steps (under control of row and column address strobe signals), which requires only half the number of address pins.

The memory stores bytes, i.e., 8-bit elements, so we expect there to be 8 duplicated arrays each consisting of 65536 cells. Overall, there will be $8 \cdot 65536 = 524288$ cells. So, in summary, an answer of 8-bit address pins, and 524288 cells is correct; the alternative of 16-bit address pins, and 524288 cells is not *wrong* per se, but certainly less likely in practice.

▷ **S159.** We want a 32KiB memory, i.e., $32 \cdot 1024 = 32768$ addressible words each of 8 bits (or 1 byte). The memory devices we have use a 4-bit data bus and 12-bit address bus: this implies that each one has $2^{12} = 4096$ addressible words each of 4 bits. Two such devices could be combined to support an 8-bit word size: we simply take the LSBs of each byte from one device, and the MSBs from the other. Therefore, to construct the memory required we need

$$\frac{8}{4} \cdot \frac{32768}{4096} = 2 \cdot 8 = 16$$

devices.

▷ **S160.** a A 1kB, byte-addressable SRAM would usually require $n = 10$ address wires. The n-bit A means addresses between 0 and $2^n - 1 = 2^{10} - 1 = 1023$ are accessible.

Consider a small(er) example of an SRAM where $n = 3$: the addresses

A_2	A_1	A_0	A		
0	0	0	$000_{(2)}$	$\equiv$	$0_{(10)}$
0	0	1	$001_{(2)}$	$\equiv$	$1_{(10)}$
0	1	0	$010_{(2)}$	$\equiv$	$2_{(10)}$
0	1	1	$011_{(2)}$	$\equiv$	$3_{(10)}$
1	0	0	$100_{(2)}$	$\equiv$	$4_{(10)}$
1	0	1	$101_{(2)}$	$\equiv$	$5_{(10)}$
1	1	0	$110_{(2)}$	$\equiv$	$6_{(10)}$
1	1	1	$111_{(2)}$	$\equiv$	$7_{(10)}$

are accessible. Now imagine the m-th address wire is misconnected where $m = 1$, meaning $A_1 = 0$: this yields

A_2	A_1	A_0	A		
0	0	0	$000_{(2)}$	$\equiv$	$0_{(10)}$
0	0	1	$001_{(2)}$	$\equiv$	$1_{(10)}$
0	0	0	$000_{(2)}$	$\equiv$	$0_{(10)}$
0	0	1	$001_{(2)}$	$\equiv$	$1_{(10)}$
1	0	0	$100_{(2)}$	$\equiv$	$4_{(10)}$
1	0	1	$101_{(2)}$	$\equiv$	$5_{(10)}$
1	0	0	$100_{(2)}$	$\equiv$	$4_{(10)}$
1	0	1	$101_{(2)}$	$\equiv$	$5_{(10)}$

so now only addresses $0_{(10)}$, $1_{(10)}$, $4_{(10)}$, and $5_{(10)}$ are accessible. Put another way, 1/2 of the originally accessible addresses will remain accessible. The same fact applies for any m, so for $n = 1024$ we conclude that $1024/2 = 512$ addresses are accessible.

b A 1kB, byte-addressable DRAM would usually require $n = 5$ address wires. The n-bit A means addresses between 0 and $2^{2 \cdot n} - 1 = 2^{2 \cdot 5} - 1 = 2^{10} - 1 = 1023$ are accessible, because the address is communicated via A in two steps: each communicates n bits, so produce a $(2 \cdot n)$-bit address overall.

Consider a small(er) example of a DRAM where $n = 2$: if A_i^j denotes the i-th address wire as used in the j-th step, the addresses

A_1^1	A_0^1	A_1^0	A_0^0	A
0	0	0	0	$0000_{(2)} \equiv 0_{(10)}$
0	0	0	1	$0001_{(2)} \equiv 1_{(10)}$
0	0	1	0	$0010_{(2)} \equiv 2_{(10)}$
0	0	1	1	$0011_{(2)} \equiv 3_{(10)}$
0	1	0	0	$0100_{(2)} \equiv 4_{(10)}$
0	1	0	1	$0101_{(2)} \equiv 5_{(10)}$
0	1	1	0	$0110_{(2)} \equiv 6_{(10)}$
0	1	1	1	$0111_{(2)} \equiv 7_{(10)}$
1	0	0	0	$1000_{(2)} \equiv 8_{(10)}$
1	0	0	1	$1001_{(2)} \equiv 9_{(10)}$
1	0	1	0	$1010_{(2)} \equiv 10_{(10)}$
1	0	1	1	$1011_{(2)} \equiv 11_{(10)}$
1	1	0	0	$1100_{(2)} \equiv 12_{(10)}$
1	1	0	1	$1101_{(2)} \equiv 13_{(10)}$
1	1	1	0	$1110_{(2)} \equiv 14_{(10)}$
1	1	1	1	$1111_{(2)} \equiv 15_{(10)}$

are accessible. Now imagine the m-th address wire is misconnected where $m = 1$, meaning $A_1 = 0$: this yields

A_1^1	A_0^1	A_1^0	A_0^0	A
0	0	0	0	$0000_{(2)} \equiv 0_{(10)}$
0	0	0	1	$0001_{(2)} \equiv 1_{(10)}$
0	0	0	0	$0000_{(2)} \equiv 0_{(10)}$
0	0	0	1	$0001_{(2)} \equiv 1_{(10)}$
0	1	0	0	$0100_{(2)} \equiv 4_{(10)}$
0	1	0	1	$0101_{(2)} \equiv 5_{(10)}$
0	1	0	0	$0100_{(2)} \equiv 4_{(10)}$
0	1	0	1	$0101_{(2)} \equiv 5_{(10)}$
0	0	0	0	$0000_{(2)} \equiv 0_{(10)}$
0	0	0	1	$0001_{(2)} \equiv 1_{(10)}$
0	0	0	0	$0000_{(2)} \equiv 0_{(10)}$
0	0	0	1	$0001_{(2)} \equiv 1_{(10)}$
0	1	0	0	$0100_{(2)} \equiv 4_{(10)}$
0	1	0	1	$0101_{(2)} \equiv 5_{(10)}$
0	1	0	0	$0100_{(2)} \equiv 4_{(10)}$
0	1	0	1	$0101_{(2)} \equiv 5_{(10)}$

so now only addresses $0_{(10)}$, $1_{(10)}$, $4_{(10)}$, and $5_{(10)}$ are accessible.
Put another way, $1/4$ of the originally accessible addresses will remain accessible. The same fact applies for any m, so for $n = 1024$ we conclude that $1024/4 = 256$ addresses are accessible.

▷ **S161.** By checking all possible addresses

$$0 \leq A < 2^{18} = 262144_{(10)} = 40000_{(16)},$$

against the enable signals, we find that

$$\begin{array}{llllcll} en_0 = 1 & \text{for} & A \in \{ & 00000_{(16)}, & \ldots, & 07FFF_{(16)} & \} \Rightarrow \text{MEM}_0 \text{ is enabled} \\ en_1 = 1 & \text{for} & A \in \{ & 08000_{(16)}, & \ldots, & 0BFFF_{(16)} & \} \Rightarrow \text{MEM}_1 \text{ is enabled} \\ en_2 = 1 & \text{for} & A \in \{ & 10000_{(16)}, & \ldots, & 1FFFF_{(16)} & \} \Rightarrow \text{MEM}_2 \text{ is enabled} \\ en_3 = 1 & \text{for} & A \in \{ & 3FFE0_{(16)}, & \ldots, & 3FFFF_{(16)} & \} \Rightarrow \text{MEM}_3 \text{ is enabled} \end{array}$$

Since

$$A = 48350_{(10)} = 0BCDE_{(16)},$$

this address therefore maps to memory device MEM$_1$ because

$$08000_{(16)} \leq 0BCDE_{(16)} \leq 0BFFF_{(16)}.$$

Of course, doing a similar search by hand is very time consuming; a manual solution would therefore use the *form* of each en_i as a short-cut. For example, we know

$$en_0 = \neg A_{17} \wedge \neg A_{16} \wedge \neg A_{15},$$

i.e., $en_0 = 1$ when the 3 MSBs of A are 0. This fact leads to the range

$$A \in \{00000_{(16)}, \ldots, 07FFF_{(16)}\}$$

fairly directly, because it captures all 18-bit values whose 3 MSBs are 0, i.e.,

$$\begin{array}{rcl} 000000000000000000_{(2)} & = & 00000_{(16)} \\ 000000000000000001_{(2)} & = & 00001_{(16)} \\ & \vdots & \\ 000111111111111111_{(2)} & = & 07FFF_{(16)} \end{array}$$

▷ **S162.** a **false**. The implication of this statement is that a larger word size leads to a higher access latency, because the two memory devices are identical in all other respects. This argument is not correct, however. The reason is that a w-bit memory device is basically constructed using w replicated instances of a 1-bit memory device: each one operates in parallel, performing an access with respect to 1 of the w bits in each element. Given the available information, therefore, there is no real evidence for the claimed difference in access latency.

b **true**. In essence, the reason for this is that the value stored by a DRAM is maintained in a capacitor: if the capacitor is charged the value is 1, whereas if the capacitor is dis-charged the value is 0. Even when powered-on that charge has to be periodically refreshed because it will degrade (or decay); when powered-off the same sort of degradation occurs, likely at a faster rate, until the charge and value is lost. But in neither case is the degradation instantaneous. So, for a short period after a DRAM cell has been powered-off, it will retain some decaying amount of charge and so the value stored.

c **false**. The address and data bus widths imply this SRAM supports 2^{16} addressable words, each of 8 bits (or 1 byte): it therefore has a total capacity of $2^{16} \cdot 8 = 524288$ bits, or $524288/8 = 64 \cdot 1024$ bytes, i.e., 64KiB.

▷ **S163.** The main components to include in such a diagram are

a the four internal transistors that form a loop of two NOT gates, and

b the two external transistors that allow access to the loop via the bit- and word-lines.

The diagram is as follows:

```
                            wl
                            |
                            v
    +-----------------------+-----------------------+
    |                                               |
    |        ----+--------------------+----- V_dd   |
    |            |                    |             |
    |            v                    v             |
    |        +---------+         +---------+        |
    |        | P-type  |--+   +--| P-type  |        |
    |        +---------+  |   |  +---------+        v
    |            |        |   |       |        +---------+
    v            |        +---|-------+--------| N-type  |<-- ~bl
+---------+      |        |   |       |        +---------+
| N-type  |------+--------|---+       |
+---------+      |        |   |       |
             +---------+  |   |  +---------+
             | N-type  |--+   +--| N-type  |
             +---------+         +---------+
                 ^                    ^
                 |                    |
             ----+--------------------+----- VSS
```

▷ **S164.** a Two main answers are clear. First, use of the DRAM device could imply a somewhat more involved, 2-step access algorithm: it is common to latch the row and column buffers (under control of two dedicated row and column strobes) in two steps, hence allowing half the number of pins to address the same number of cells. Second, the DRAM cells need to be refreshed periodically since their content will decay. Typically a mechanism to do this might be built into the device, but if not then the system itself will need to be responsible for doing so.

b The obvious reason an SRAM device might have a lower access latency is because the individual cells have a lower access latency: since SRAM cells are constructed from transistors, they can be accessed (read from or written to) more quickly than a capacitor-based DRAM cell (which takes longer to charge and discharge).

▷ **S165.** a As the number of cells grows larger, providing enough address pins to identify each physical cell can become impractical. One way to combat this problem is to multiplex the address pins; roughly this means using less pins in more steps, e.g., $n'/2$ pins in two steps rather than n' pins in one step. Thus, under control of the row and column strobes, two steps will see the $(n'/2)$-bit row and column addresses latched into the row and column buffer: once latched, the combined content forms a usable n'-bit address. So in short, the buffers are required to retain the row and column addresses during this process.

b Once the row and column buffers are latched with the address, the device is ready to access the identified cell. However, depending on the geometry, i.e., number of rows and columns, a translation needs to be made: this is the task of the row and column decoders. Essentially they implement the translation between logical address and physical cell, activating said cell to perform the required operation (which is either a read or a write).

Part VI: Digital logic design using Verilog

▷ **S166.** a `wire [ 7 : 0 ] a;`

b `wire [ 0 : 4 ] b;`

c `reg [ 31 : 0 ] c;`

d `reg signed [ 15 : 0 ] d;`

e `reg [ 7 : 0 ] e[ 0 : 1023 ];`

f `genvar f;`

▷ **S167.** a `c = 2'b01`

b `c = 2'b11`

c `d = 4'b010X`

d `d = 4'b10XX`

e `d = 4'b1101`

f `d = 4'b0111`

g `c = 2'bXX`

h `c = 2'b11`

i `e = 1'b0`

j `e = 1'b1`

▷ **S168.** a One potential problem is the if `p` and `q` can change at any time, and hence trigger execution of the processes at any time, the two might change at the exact *same* time. In this case, it is not clear which values `x` and `y` will be assigned to. Maybe the top assignment to `x` beats the bottom one, but the bottom assignment to `y` beats the top one. Any combination is possible; since it is not clear which will occur, it is possible that `x` and `y` do not get assigned to the values one would expect.

As an attempt at a solution, we can try to exert some control over which block of assignments is executed first. For example, we might try to place a guard around the assignments:

```
always @ ( posedge p ) begin
  if( !q ) begin
    x <= a;
    y <= b;
  end
end

always @ ( posedge q ) begin
  if( !p ) begin
    x <= b;
    y <= a;
  end
end
```

An alternative might be to combine the two blocks into one:

```
always @ ( posedge p, posedge q ) begin
  if     ( p ) begin
    x <= a;
    y <= b;
  end
  else if( q ) begin
    x <= b;
    y <= a;
  end
end
```

since now the process is at least deterministic: if p is equal to 1 then the first block executes, if q is equal to 1 then the second block executes and if both are equal to 1 we execute the first block as the default.

b The problem with this is that the `state` signal is not initialised; to start with it could be any value which might either result in the state machine operating in the wrong sequence or, since `case` is used and not `casex` or `casez`, none of the cases being matched at all. A slightly more minor issue is that we have to assume that no other process assigns to `state`. For example, if another process sets `state` to 3 the state machine process will malfunction.

The best way to rectify this problem is by introducing a reset signal called `rst` and initialising the state variable whenever it is equal to 1:

```
always @ ( posedge clk, posedge rst ) begin
  if( rst ) begin
    state = 0;
  end
  else begin
    case( state )
      0 : begin do_0; state = 1; end
      1 : begin do_1; state = 2; end
      2 : begin do_2; state = 0; end
    endcase
  end
end
```

▷ **S169.** This is a bit of a vague question; it does not mention styles of Verilog and so you can assume any valid style is allowed. With this in mind, a rough solution might look something like this:

```
module DDD( clk, rst, bit, out );

  input  wire          clk;
  input  wire          rst;

  input  wire          bit;
  output wire          out;

         reg [ 3 : 0 ] t;

  assign out = ( t >= 0 ) && ( t <= 9 );

  always @ ( posedge clk ) begin
    if( rst ) begin
      t = 0;
    end
    else begin
      t = { bit, t[ 3 : 1 ] };
    end
  end

endmodule
```

▷ **S170.** Essentially we want a design that looks, at a high-level, like this:

```
       y
       |
       v
    +-----+
x-->|  C  |-->min(x,y)
    +-----+
       |
       V
    max(x,y)
```

and thus need some sort of comparison inside; we already know how to design a less than comparison which is good enough. Thus, the Verilog module could look something like this:

```
module C( min, max, x, y );

  parameter n = 8;

  output wire                  r;

  output wire [ n - 1 : 0 ] min;
  output wire [ n - 1 : 0 ] max;

  input  wire [ n - 1 : 0 ]   x;
  input  wire [ n - 1 : 0 ]   y;

         wire [ n - 1 : 0 ] w0 = ~(  x ^ y );
         wire [ n - 1 : 0 ] w1 =  ( ~x & y );
         wire [ n - 1 : 0 ] w2;

  assign w2[ 0 ] = w1[ 0 ];

  genvar i;

  generate
    for( i = 1; i < n; i = i + 1 ) begin
      assign w2[ i ] = w1[ i ] | ( w0[ i ] & w2[ i - 1 ] );
    end
  endgenerate

  assign min = ( w2[ n - 1 ] ) ? x : y;
  assign max = ( w2[ n - 1 ] ) ? y : x;

endmodule
```

▷ **S171.** In software the task is fairly simple; we simply have an 8-bit variable called `Q` which maintains the content of the shift register, and allow two functions to initialise and update the value (i.e., clock the register) as follows:

```
uint8_t Q = 0;

void seed( uint8_t s ) {
  // seed the state
  Q = s;
}

uint8_t lfsr() {
  // compute outgoing bit
  uint8_t r = Q & 1;

  // compute incoming bit
  uint8_t t = ( Q >> 7 ) ^
              ( Q >> 5 ) ^
              ( Q >> 4 ) ^
              ( Q >> 3 ) ^
              ( Q >> 0 ) ;

  // update state
          Q = ( Q >> 1 ) |
              ( x << 7 ) ;

  return r;
}
```

Following this approach, in Verilog the design is also simple: we just need to deal with the required action each time a positive clock edge triggers an update. For example, the following does more or less the same thing as the C version:

```
module lfsr( input  wire           clk,
             input  wire           rst,

             input  wire [ 7 : 0 ]   s,
             output reg              r );

  reg [ 7 : 0 ] Q;

  always @ ( posedge clk ) begin
    if ( rst ) begin
      // seed the state
      Q = s;
    end else begin
      // compute outgoing bit
      r =   Q[ 0 ] ;

      // compute incoming bit, update state
      Q = { Q[ 7 ] ^
            Q[ 5 ] ^
            Q[ 4 ] ^
            Q[ 3 ] ^
            Q[ 0 ], Q[ 7 : 1 ] };
    end
  end

endmodule
```

That is, we again have 8-bit register called Q; each time a positive edge occurs on clk we make a choice

- If rst is **true** then we initialise the LFSR by setting Q equal to the seed value s,
- If rst is **false** then we update the LFSR by first setting the output r equal to the 0-th bit of Q, then updating Q via a concatenation expression (which performs the shift, meaning the $(n-1)$-th bit of the result is the XOR of the tap bits).

Part VII: Computational machines: Finite State Machines (FSMs)

▷ **S172.** Throughout the following, keep in mind that three main component groups can be identified in the implementation: read from bottom-to-top, these are

- an input register (bottom),
- some combinatorial logic (middle), and
- an output register (top).

We know this is an FSM, so we expect the input register to hold the current state and the combinatorial logic to compute both the transition and output functions. Specifically, the input register and x are provided as input to combinatorial logic (in the middle-left and -center) that represents the transition function. It computes the next state, then stored in the output register; the rest of this logic (in the middle-right) clearly represents the output function, since it computes r.

a i Saying a signal is digital is the same as saying it takes the values 0 and 1 only; for a clock signal, this is the same as saying it the form is a perfect square wave.

In practice this is difficult to achieve since transitions between 0 and 1 cannot be perfectly instantaneous. This implies each edge has a slope, however shallow this is. Even so, the same issue is true for all digital logic components: if the inputs to an AND gate are neither 0 or 1 (or their associated voltage levels), the output is undefined. So it is also fair to say this is a requirement for Φ_1 and Φ_2, at least as far as is practical.

ii Φ_1 and Φ_2 are said to be non-overlapping in the sense a positive level on one always occurs at the same time as a negative level on the other.

If this were *not* true, e.g., $\Phi_1 = \Phi_2$, then a "loop" would form during the overlap: the output register would be updated with whatever was computed by the combinatorial logic, which is fed by the input register *also* being updated at the same time by the output register. This would likely result in a malfunction of some sort: the input register could not settle into a stable state, for example.

iii To gate *any* signal, Φ_1 and Φ_2 included, means to (conditionally) disable them. This is typically realised by adding extra logic, e.g., an AND gate, so the clock signal can be forced to 0.

This might be useful; it allows one to disable latch updates and so "pause" the FSM (e.g., to save power when idle). However, doing so is not a requirement and not evident in this implementation.

iv In general, the concept of skew describes a situation where the clock signal arrives at two components at different times; with a 2-phase clock, we might *also* find cases where Φ_1 and Φ_2 can arrive at the same component at different times.

This is clearly undesirable, since the clock is meant to synchronise the component. If they were *not* synchronised then malfunction is the likely result: one latch might be updated at a different time, and hence with an unrelated value, than another, for example.

v The duty cycle of a clock (signal) is the percentage of each clock period in which it has a positive level. For example, saying that Φ_1 has a duty cycle of 33% is the same as saying $\Phi_1 = 1$ for a third of the time (and hence $\Phi_1 = 0$ for two thirds of the time).

Here, there is no reason the duty cycle of Φ_1 and Φ_2 must be 33%. If is was 40%, for example, the implementation would still function correctly. Assuming Φ_1 and Φ_2 have the same form, then of course it must be true their duty cycles are less than 50% otherwise they would have to overlap. Other than that, however, getting closer to 50% just means less separation between their positive levels.

b This is not a *trick* question per se, but the correct answer is that the register might hold any 2-bit value when powered-on. Although the register must settle into *some* state, it is not clear how we could predict what this will be: the stored value is basically random, or more precisely determined by physics of the underlying implementation and fabrication process.

As an aside, this FSM has an related, unattractive design feature: there is no reset input. This means there is no way to enforce a start state, i.e., whatever value is held in the register at power-on is used as the start state: the only way to alter this, and hence make the FSM function as required, is to power-cycle the implementation and hope the initial stored value is the required start state!

c Partly as a result of the multiple-choice format, this question might seem odd. Exactly the same *concepts* are involved, but the *steps* are the opposite way around: rather than derive an implementation from a given specification, it asks you to reverse engineer a specification from a given implementation.

Each of the registers constitutes 2 D-type latches, so the FSM can be in at most $2^2 = 4$ possible states. Denoting the current (resp. next) state $Q = \langle Q_0, Q_1 \rangle$ (resp. $Q' = \langle Q'_0, Q'_1 \rangle$), we can write an expression for the next state and output in terms of the current state and input: this basically just means translating the logic gate symbols into a Mathematical form. That is, we can write

$$\begin{array}{lclclcl} Q'_1 & = & (Q_1 \wedge Q_0) & \vee & (x \wedge Q_1) & \vee & (x \wedge Q_0) \\ Q'_0 & = & (Q_1 \wedge Q_0) & & & \vee & (x \wedge \neg Q_0) \\ r & = & (Q_1 \wedge Q_0) & \vee & (x \wedge Q_1) & & \end{array}$$

By enumerating the possible values of x, Q_0 and Q_1 we find

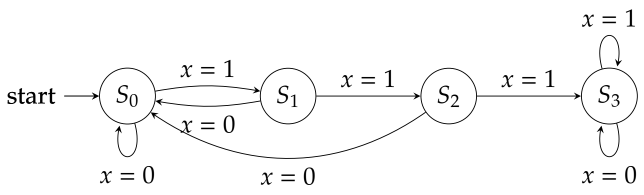
x	Q_1	Q_0	Q'_1	Q'_0	r
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	1	1	1
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	1	1
1	1	1	1	1	1

i.e., we reverse engineer the transition and output functions, expressed as a truth table. For example, if the current state is $Q = \langle 0, 0 \rangle$ and the input is $x = 1$ then we can see the next state will be $Q' = \langle 1, 0 \rangle$ and the output will be $r = 0$.

The truth table encodes the same information as a diagrammatic alternative. The only difference is the use of a concrete representation, rather than an abstract label for each state. We *need* the former, because of course the implementation stores and computes Boolean values: it cannot deal with a label such as S_0 or S_3 means unless we give that label a value. So imagine we make such an (reverse) assignment, namely

$$\begin{array}{lcl} \langle 0, 0 \rangle & \mapsto & S_0 \\ \langle 1, 0 \rangle & \mapsto & S_1 \\ \langle 0, 1 \rangle & \mapsto & S_2 \\ \langle 1, 1 \rangle & \mapsto & S_3 \end{array}$$

Now we can say, for example that if the current state is S_0 and the input is $x = 1$ then the next state will be S_1 and the output will be $r = 0$. This makes drawing the diagrammatic alternative a little easier: if we draw 4 nodes for the four states, we join them with edges based on rows of the truth table. The end result, and correct answer, is



d The central difference between Mealy- and Moore-type FSMs stems from how the output function is defined. In the former, the output is a function of the current state *and* input; in the latter, *only* the current state is relevant. For a set of states S and input and output alphabets Σ and Γ, this means for a Mealy-type FSM we have

$$\omega : S \times \Sigma \to \Gamma$$

whereas for a Moore-type FSM we have

$$\omega : S \to \Gamma.$$

For this FSM, we already know from the previous question that the output is described by

$$r = (Q_1 \wedge Q_0) \vee (x \wedge Q_1).$$

As such, it should be obvious $r = \omega(Q, x)$ is a function of both Q (the current state) and x (the input): this is therefore a Mealy-type FSM.

e The behaviour of this FSM can be described as repeated iteration over two steps under control of the clock. That is, it repeatedly does

- Step #1:
  - the combinatorial logic compute the next state $Q' = \delta(Q, x)$ and output $r = \omega(Q, x)$, and
  - the output register latches Q'.

  Note that the critical path of this step is that from the the Q output of the input register to the Q output of the output register.
- Step #2:
  - the input register latches Q' as Q.

  Note that the critical path of this step is that from the the Q output of the output register to the Q output of the input register.

f For example, the first step occurs during the period when $\Phi_1 = 1$ and the second when $\Phi_2 = 1$. Within every clock period, i.e., within the "time limit" represented by ρ, *both* steps must be completed. Therefore, we can say

$$\rho \geq (T_{\text{logic}} + T_{\text{latch}}) + (T_{\text{latch}})$$

where T_{latch} and T_{logic} are the critical paths associated with a D-type latch and the combinatorial logic respectively.

Note the critical path runs through the middle-left *or* middle-center of the combinatorial logic: although the former includes a NOT gate, the latter includes a 3- versus 2-input OR gate. Either way the delay is 50ns, so we can write

$$\begin{array}{rcl} \rho & \geq & 50 + 60 + 60\text{ns} \\ & \geq & 170\text{ns} \end{array}$$

then compute the maximum clock frequency as

$$\begin{array}{rcl} f\text{max} & = & 1/\rho \\ & = & 1/170\text{ns} \\ & \simeq & 5.9\text{MHz} \end{array}$$

g From the definition of the transition function (above), it *should* be clear the FSM will progress from left-to-right as consecutive inputs $x = 1$ are encountered. Moreover, encountering an input of $x = 0$ means restarting in state S_0, and when eventually the FSM reaches state S_3 it stays in that state (whether $x = 1$ or $x = 0$). So it basically counts the number of consecutive times $x = 1$ until that count is 3 (if it is in state S_i, then the count is i). This already provides the correct answer, but is further confirmed by inspecting the output function: $r = 1$ when the FSM is in state S_3, i.e., when the count is 3.

▷ **S173.** a Before t_0, we can see that a pulse on rst at the same time as $\Phi_2 = 1$; this acts as a reset, storing s (as a result of the multiplexers) into the top register. Then, at t_0 we find that $\Phi_1 = 1$: during this period, the design stores a the bottom register as provided by the top register (which, at that point, is fixed since $\Phi_2 = 0$). As such, at t_0 we expect the bottom register to store s and hence r to be the MSB of s, i.e., $r = s_7 = 1$.

b At t_1 the design has performed one cycle relative to t_0: the value stored in the bottom register at t_0 is updated by the middle of the design, then stored in the top register, and finally stored back in the bottom register (ready for the next cycle). The middle of the design is fairly simple. Ignoring the less-significant end since this does not impact r (yet), it basically just shifts the bits toward the more-significant end. At t_1, we therefore expect the bottom register to be st. $r = s_6 = 0$.

c This design is a Linear Feedback Shift Register (LFSR); such a design might be used to support a variety of use-cases, with a common example being the generation of (pseudo-)random bits. As the name suggests, an LFSR is essentially an n-bit shift register. After initialising (or seeding) the register state with s, successive updates are performed; each such update a) shifts-out an output bit (wlog. the MSB), which forms the LFSR output, and b) shifts-in an input bit (wlog. the LSB), which is computed using a linear function of the state. A set T captures the tap bits, which specify the function of x used to compute the input bit; given n, T is selected to maximise the period of the LFSR, noting that $x = 0$ should be disallowed to avoid trivial behaviour.

Both Fibonacci- and Galois-form LFSR designs are possible; in this case, we have an example of the former, with $n = 8$ and $T = \{3, 4, 5, 7\}$. Given a state x, the update process, yielding an output bit r and a next state x', can be formalised as

$$\begin{array}{rcl} r &=& x_7 \\ x' &=& (x \lll 1) \parallel (\bigoplus_{i \in T} x_i) \\ &=& (x_6 \parallel x_5 \parallel \cdots \parallel x_0) \parallel (x_3 \oplus x_4 \oplus x_5 \oplus x_7) \end{array}$$

As such, we can use a table to trace the state and output as it is updated:

i	x	x'	r	
		$A6_{(16)}$		seed x with s
0	$A6_{(16)}$	$4C_{(16)}$	1	generate 0-th output bit
1	$4C_{(16)}$	$99_{(16)}$	0	generate 1-st output bit
2	$99_{(16)}$	$33_{(16)}$	1	generate 2-nd output bit
3	$33_{(16)}$	$66_{(16)}$	0	generate 3-rd output bit
4	$66_{(16)}$	$CD_{(16)}$	0	generate 4-th output bit
5	$CD_{(16)}$	$9A_{(16)}$	1	generate 5-th output bit
6	$9A_{(16)}$	$35_{(16)}$	1	generate 6-th output bit
7	$35_{(16)}$	$6A_{(16)}$	0	generate 7-th output bit
8	$6A_{(16)}$	$D4_{(16)}$	0	generate 8-th output bit
⋮	⋮	⋮	⋮	

Using this table, we can infer that at time t_2 (where the 8-th output bit is generated, which is the first bit which is computed from x vs. matching s), $r = 0$.

d Within the clock period (i.e., within the "time limit" which ρ dictates), two steps must be completed; those steps are completed when $\Phi_1 = 1$ and $\Phi_2 = 1$ respectively, and can be described as 1) the top register must be updated with a value computed by the middle of the design (i.e., the combinatorial logic) from the value in the bottom register, then 2) the bottom register must be updated with the value in the top register. So if T_{latch} and T_{logic} are the critical paths associated with a D-type latch and said combinatorial logic respectively, then we can write

$$\rho \geq (T_{\text{logic}} + T_{\text{latch}}) + (T_{\text{latch}}).$$

Adding more detail, we could then reflect the critical path of components constituting the combinatorial logic: writing

$$T_{\text{logic}} = T_{xor} + T_{xor} + T_{mux}$$

then reflects the fact that the critical path includes two XOR gates and one multiplexer. Overall then, we have

$$\begin{array}{rcl} \rho &\geq& (T_{\text{xor}} + T_{\text{xor}} + T_{\text{mux}} + T_{\text{latch}}) + (T_{\text{latch}}) \\ &\geq& 2 \cdot T_{\text{latch}} + 2 \cdot T_{\text{xor}} + T_{\text{mux}} \end{array}$$

Since we have the design of each component, we can, as a next step, be more concrete about each term above: inspecting the NAND based designs, we can deduce

$$\begin{array}{rcccl} T_{\text{latch}} &=& 4 \cdot T_{\text{nand}} &=& 40\text{ns} \\ T_{\text{xor}} &=& 3 \cdot T_{\text{nand}} &=& 30\text{ns} \\ T_{\text{mux}} &=& 3 \cdot T_{\text{nand}} &=& 30\text{ns} \end{array}$$

and thus

$$\begin{array}{rcl} \rho &\geq& 2 \cdot T_{\text{latch}} + 2 \cdot T_{\text{xor}} + T_{\text{mux}} \\ &\geq& 2 \cdot 40\text{ns} + 2 \cdot 30\text{ns} + 30\text{ns} \\ &\geq& 80\text{ns} + 60\text{ns} + 30\text{ns} \\ &\geq& 170\text{ns} \end{array}$$

T_{latch} arguably represents the more tricky case, noting that the cross-coupled right-hand side means the path is through 4 NAND gates. Finally, the maximum clock frequency is inversely proportional to this critical path so we find

$$\begin{array}{rcl} f_{\text{max}} &=& 1/\rho \\ &=& 1/170\text{ns} \\ &\simeq& 5.9\text{MHz} \end{array}$$

is correct.

▷ **S174.** Basically this question is asking us to reverse engineer the FSM implementation into a design and hence functionality; to do that, we can step *backwards* through the process we have that would normally step *forwards*.

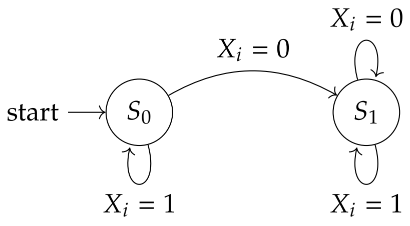
The first step is therefore be to inspect the implementation and extract pertinent features: 1) the bottom and top D-type latches capture 1-bit current and next states, i.e., Q and Q', respectivly, 2) between the two we can identify an output function $r = \omega(Q) = \neg Q$ and a transision function $Q' = \delta(Q, rst) = (\neg rst) \wedge (\neg X_i \vee Q)$. Note that we can classify this as a Moore-type FSM, since the output r is determined by the current state Q alone.

The next step is to reconstruct a concrete, tabular description of the FSM, i.e., a truth table, using ω and δ:

			δ	ω
rst	X_i	Q	Q'	r
0	0	0	1	1
0	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	1
1	1	1	0	0

Because Q and Q' are each represented by a single D-type latch, we can infer the FSM has (at most) two states. Other assignments are possible provided we are consistent, but the most natural would be to say $Q = 0 \mapsto S_0$ and $Q = 1 \mapsto S_1$. Given that $rst = 1$ forces $Q' = 0$, we can infer than S_0 is an initial state; Given that $r = \neg Q$ and so $r = 1$ iff. $Q = 0$, we can infer than S_0 is an accepting state.

The next step is to reconstruct a abstract, diagrammatic description of the FSM:



Note that we infer the start state from the behaviour of the reset signal, i.e., the Q' implied by $rst = 1$; there is no accepting state per se.

The final step demands some creativity, in the sense that we need to interpret the functionality realised: although doing so is not trivial, we can approach it by tring to explain in words what the FSM does step-by-step. For example, note that the FSM starts in state S_0 and stays there while the input is $X_i = 1$. However, as soon as it encounters an input st. $X_i = 0$ it will transistion to state S_1: it stays there whether the input is $X_i = 0$ or $X_i = 1$. So, put another way, the FSM

- stays in state S_0 if $X_i = 1$ for all i; if therefore accepts such an input, meaning $r = 1$,
- transistions to and stays in state S_1 if $X_i = 0$ for any i; if therefore rejects such an input, meaning $r = 0$.

This description matches the definition of AND: we have that

$$r = X_0 \wedge X_1 \wedge \cdots \wedge X_{n-1}.$$

▷ **S175.** Interpreting the design, we can see that

$$\begin{array}{lclclcrclcl}
Q'_0 & = & x & \wedge & (& Q_0 & \vee & Q_1 &) \\
Q'_1 & = & x & \wedge & (& \neg Q_0 & \vee & Q_1 &) \\
 \\
r & = & \neg x & \wedge & (& Q_0 & \wedge & Q_1 &)
\end{array}$$

where the expressions for Q'_0 and Q'_1 constitute the transition function δ, and the expression for r constitutes the output function ω. This means:

a 3 gates are involved in the output function implementation (2 AND gates, and 1 NOT gate), and

b 5 gates are involved in the transition function implementation (2 AND gates, 2 OR gates, and 1 NOT gate).

▷ **S176.** Note that the AND gate labelled $\odot$ clearly forms part of the transition function, because the output function (which generates r) is independent from it. We can produce a solution in two steps, 1) reproduce a complete implementation from Figure **??**, then 2) match that implementation against Figure **??** to infer what the placeholder component should be instantiated with. First, we deal with the state. From Figure **??** we can see there are 4 possible states, which are represented via a binary encoding (per the question) using 2 bits, i.e.,

$$\begin{array}{lcl} S_0 & \mapsto & \langle 0,0 \rangle \\ S_1 & \mapsto & \langle 1,0 \rangle \\ S_2 & \mapsto & \langle 0,1 \rangle \\ S_3 & \mapsto & \langle 1,1 \rangle \end{array}$$

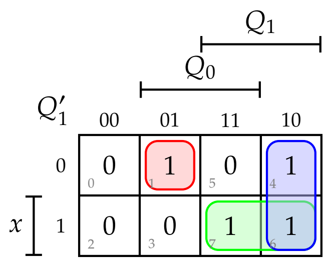
This allows

$$\begin{array}{lcl} Q & = & \langle Q_0, Q_1 \rangle \\ Q' & = & \langle Q'_0, Q'_1 \rangle \end{array}$$

to define the current and next state respectively: in Figure **??**, these are realised using 2-bit registers (each of which comprises 2 D-type latches) shown toward the bottom and top. Second, we deal with the transition function. From Figure **??** we can derive the following truth table:

x	Q_1	Q_0	Q'_1	Q'_0
0	0	0	0	0
1	0	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	0	1	0
1	1	0	1	1
0	1	1	0	0
1	1	1	1	1

Then, using a Karnaugh map for Q'_1



we produce

$$\begin{array}{lcccccccc} Q'_0 & = & (& x & & & & &) \\ Q'_1 & = & (& \neg x & \wedge & \neg Q_1 & \wedge & Q_0 &) & \vee \\ & & (& x & \wedge & Q_1 & & &) & \vee \\ & & (& & & Q_1 & \wedge & \neg Q_0 &) \end{array}$$

Then, third, we match our expressions against Figure **??**: it should be clear that left- and right-hand AND gates implement the first and third terms of the expression above, so the middle AND gate should implement the second term using the inputs x and Q_1.

▷ **S177.** a **true**. The question does not specify an FSM style, so, without loss of generality, assume a focus on Mealy-style FSMs. Fundamentally, such an FSM is driven by a clock signal: during the current clock cycle, it will 1) compute $r = \omega(Q, x)$, i.e., an output, and, simultaneously, 2) compute $Q' = \delta(Q, x)$, i.e., a next state, in both cases using current state Q and input x. The same process is repeated in each next clock cycle, once the current state updated to equal the previously computed next state.

Using the term *during* is important, in the sense that if the current clock cycle ends (i.e., next clock cycle begins) before r or Q' is computed, the FSM will malfunction. Put another way, it will malfunction if computation of r or Q' exceeds beyond the current clock cycle; this could occur when the critical path of ω or δ is too long. So, to conclude, the clock period and thus clock frequency is limited by the critical path of ω or δ (more specifically, by whichever is longer).

b **true**. Let $\delta_M(\cdot)$ and $\omega_M(\cdot)$ respectively denote the transition and output functions for some FSM labelled M. Every Moore-style FSM X can be converted into a Mealy-style FSM Y, by defining $\delta_Y(Q, x) = \delta_X(Q, x)$ and $\omega_Y(Q, x) = \omega_X(\delta_X(Q, x))$. That is, the input x is accommodated in ω_Y by 1) applying δ_X to "look ahead" at the next state, then 2) applying ω_X to produce the output for said state. However, not every Mealy-style FSM can be converted into a Moore-style FSM: some can, but even then with a "shift" in when the output is produced.

c **true**. We can see from the implementation that the transition and output functions are

$$Q' = \delta(Q, x) \mapsto \left\{ \begin{array}{lcl} Q'_0 & = & Q_0 \oplus Q_1 \\ Q'_1 & = & Q_0 \;\overline{\wedge}\; x \end{array} \right.$$

and $r = \omega(Q) = \neg Q_0 \wedge \neg Q_1$ respectively. So if the current state is initialised to $Q = \langle Q_0, Q_1 \rangle = \langle 0, 1 \rangle$ and we fix $x = 0$, then we see that

x	Q_1	Q_0	Q'_1	Q'_0	r
0	1	0	1	1	0
0	1	1	1	0	0
0	1	0	1	1	0
0	1	1	1	0	0
0	1	0	1	1	0
⋮	⋮	⋮	⋮	⋮	⋮

i.e., Q will alternate between only 2 of 4 possible states: r will never equal 1, therefore, because the FSM will never have a current state of $Q = \langle Q_0, Q_1 \rangle = \langle 0, 0 \rangle$. A more direct way to see this is to notice that since it is always true that $x = 0$, it will always be true that $Q'_1 = Q_0 \;\overline{\wedge}\; x = 1$ (because NAND only produces an output of 0 if both inputs are 1). Therefore, since it is always true that $Q_1 = 1$ and so that $\neg Q_1 = 0$, it will always be true that $r = \neg Q_0 \wedge \neg Q_1 = 0$ (because AND only produces an output of 1 if both inputs are 1).

▷ **S178.** A generic, block diagram style framework for FSMs is as follows:

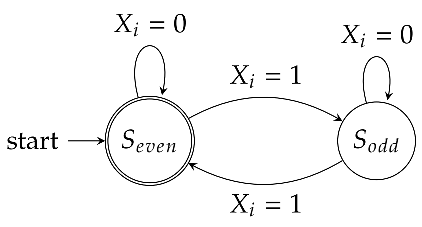
```
          +---------+
      +-->| \delta  |
      |   +---------+
      |      ^   |
      |    Q |   | Q'
      |      |   v
      |   +---------+
input --+  |  state  |<-- clock
      |   +---------+
      |      |
      |    Q |
      |      v
      |   +---------+
      +-->| \omega  |--> output
          +---------+
```

st.

- An n-bit register (middle component) holds Q, the current state of the FSM.
- Within a given clock period, the current state is provided as input to δ, the transition function: based on Q and any input, this computes the next state Q'.
- At the same time that δ is computing the next state, the output function ω computes any output from the FSM; depending on the type of FSM, this might be based on Q only, or on Q and any input.
- A positive edge of the clock signal causes the state to be updated with the output from δ. That is, the FSM advances from the current to next state; computation by δ and ω is performed in the same way during the subsequent clock period, once Q has been updated with Q'.

Note that this framework is assumed in *any* of the following questions that ask for it.

This FSM can be in one of two states: either the bits of X processed so far have an even or odd number of elements equal to 1; we give each of the states a label, so in this case S_{even} and S_{odd} for example. Next we can describe how the FSM can transitions from some current state to a next state, i.e., how the transition function δ works: based on an input X_i provided at each step, we might draw



or equivalently say

	δ	
Q	Q'	
	$X_i = 0$	$X_i = 1$
S_{even}	S_{even}	S_{odd}
S_{odd}	S_{odd}	S_{even}

where Q is the current state and Q' is the next state.

Given the FSM has two states only, we can store the current state using a 1-bit register. Based on a natural mapping of the abstract to concrete state labels (i.e., $S_{even} \mapsto 0$ and $S_{odd} \mapsto 1$), we can rewrite the transition function as a truth table:

X_i	Q	Q'
0	0	0
0	1	1
1	0	1
1	1	0

and see clearly that $Q' = Q \oplus X_i$. Inspecting Q directly provides the output: if $Q = 0$ we have (so far) even parity, and in contrast if $Q = 1$ we have odd parity. So in a sense the output function ω is simply the identity function. In short, the low-level detail filled into the high-level design is very simple (in this case at least) once the question has been digested.

One obvious addition would be some form of mechanism to reset the FSM: as stated above, we assume it starts in the state S_{even} when powered-on but clearly this may not be true (the content in Q will essentially be random initially).

▷ **S179.** a There are several approaches to solving this problem. Possibly the easiest, but perhaps not the most obvious, is to simply build a shift-register: the register stores the last three inputs, when a new input is available the register shifts the content along by one which means the oldest input drops off one end and the new input is inserted into the other end. One can then build a simple circuit to test the current state of the shift-register to see if the last three inputs match what is required..

Alternatively, one can take a more heavy-weight approach and formulate the solution as a state machine. First we need to decide on an encoding for our state; when searching though the input we can have matched zero through three correct tokens we denote this by the integer S stored in two bits using Q_1 and Q_0 as the most-significant and least-significant respectively. We also need an encoding of the actual input tokens I which are being passed to the matching circuit. Arbitrarily we might select $A = 0$, $C = 1$, $G = 2$ and $T = 3$ although other encodings are valid and might actually simplify things; we use I_1 and I_0 to denote the most and least-significant bits of the input token I. From this we can now create a table describing the mapping between current state S and input I to next state S' which can be roughly written as

I_1	I_0	Q_1	Q_0	Q'_1	Q'_0
0	0	0	0	0	1
0	1	0	0	0	0
1	0	0	0	0	0
1	1	0	0	0	0
0	0	0	1	0	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	0	1	0	0
0	0	1	0	0	1
0	1	1	0	0	0
1	0	1	0	0	0
1	1	1	0	1	1
0	0	1	1	0	1
0	1	1	1	0	0
1	0	1	1	0	0
1	1	1	1	0	0

Thus, if we are in state $(Q_1, Q_0) = (0, 0) = 0$ and see an A input, we move to state $(Q'_1, Q'_0) = (0, 1) = 1$ otherwise we stay in state $(Q'_1, Q'_0) = (0, 0) = 0$. Now we can define the transition function from current state to next state as

$$\begin{array}{rcl} Q_0 & = & (\neg Q_1 \wedge \neg Q_0 \wedge \neg I_1 \wedge \neg I_0) \vee \\ & & (\neg Q_1 \wedge Q_0 \wedge \neg I_1 \wedge \neg I_0) \vee \\ & & (Q_1 \wedge \neg Q_0 \wedge \neg I_1 \wedge \neg I_0) \vee \\ & & (Q_1 \wedge Q_0 \wedge \neg I_1 \wedge \neg I_0) \vee \\ & & (Q_1 \wedge \neg Q_0 \wedge I_1 \wedge I_0) \\ Q_1 & = & (\neg Q_1 \wedge Q_0 \wedge \neg I_1 \wedge I_0) \vee \\ & & (Q_1 \wedge \neg Q_0 \wedge I_1 \wedge I_0) \end{array}$$

with simplifications as appropriate. Finally, the output flag F will be set only according to

$$F = Q_1 \wedge Q_0$$

to signal when we have matched three characters. As such, we can realise the FSM framework described in Solution 178 by filling each component with the associated implementation above.

b Making a general-purpose matching circuit will probably use less logic than having three separate circuits; this will reduce the space required. As an extension one might consider implementing the transition and output functions as a look-up table instead of hard-wiring them; this will mean the circuit could be used to match any sequence providing the tables were correctly initialised. Introducing a more complex circuit design could have the disadvantage of increasing the critical path (the longest sequential path though the entire circuit). If the critical path is longer, the design will have to be clocked slower and hence will not perform the matching function as quickly.

▷ **S180.** a A basic diagram should show the four states and transitions between them which relate the movement from one to the other as a result of the washing cycle, and movement as a result of input from the buttons; for example a (very) basic diagram would be:

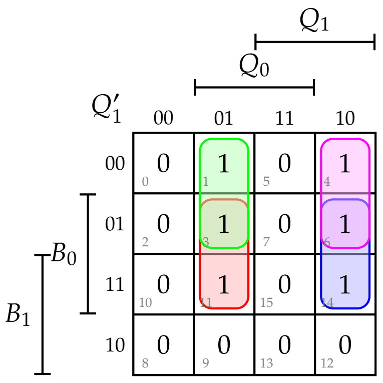
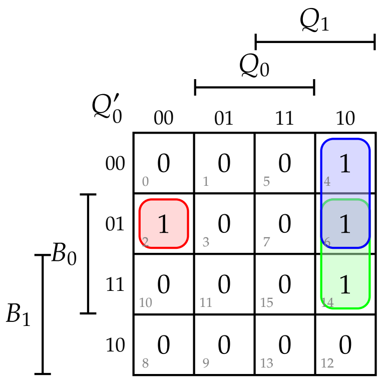
```
    +------+   +------+   +------+   +------+
+-->| idle |-->| fill |-->| wash |-->| spin |
|   +------+   +------+   +------+   +------+
|      |           |          |          |
+------+-----------+----------+----------+
```

b Since there are four states, we can encode them using one two bits; we assign the following encoding *idle* = 00, *fill* = 01, *wash* = 10 and *spin* = 11. We use Q_1 and Q_0 to represent the current state, and Q'_1 and Q'_0 to represent the next state; B_1 and B_0 are the input buttons. Using this notation, we can construct the following state transition table which encodes the state machine diagram:

B_1	B_0	Q_1	Q_0	Q'_1	Q'_0
0	0	0	0	0	0
0	1	0	0	0	1
1	0	0	0	0	0
1	1	0	0	0	0
0	0	0	1	1	0
0	1	0	1	1	0
1	0	0	1	0	0
1	1	0	1	1	0
0	0	1	0	1	1
0	1	1	0	1	1
1	0	1	0	0	0
1	1	1	0	1	1
0	0	1	1	0	0
0	1	1	1	0	0
1	0	1	1	0	0
1	1	1	1	0	0

so that if, for example, the machine is in the *wash* state (i.e., $Q_1 = 1$ and $Q_0 = 0$) and no buttons are pressed then the next state is *spin* (i.e., $Q'_1 = 1$ and $Q'_0 = 1$); however if button B_1 is pressed to cancel the cycle, the next state is *idle* (i.e., $Q'_1 = 0$ and $Q'_0 = 0$).

c From the state transition table, we can easily extract the two Karnaugh maps:

Basic expressions can be extracted from the tables as follows:

$$\begin{array}{lcl} Q_1' & = & (\neg Q_1 \wedge Q_0 \wedge B_0) \vee (\neg Q_1 \wedge Q_0 \wedge \neg B_1) \vee (Q_1 \wedge \neg Q_0 \wedge B_0) \vee (Q_1 \wedge \neg Q_0 \wedge \neg B_1) \\ Q_0' & = & (Q_1 \wedge \neg Q_0 \wedge B_0) \vee (Q_1 \wedge \neg Q_0 \wedge \neg B_1) \vee (\neg Q_0 \wedge B_0 \wedge \neg B_1) \end{array}$$

and through sharing, i.e., by computing

$$\begin{array}{lcl} t_0 & = & \neg B_1 \\ t_1 & = & \neg B_0 \\ t_2 & = & \neg Q_1 \\ t_3 & = & \neg Q_0 \\ t_4 & = & t_2 \wedge Q_0 \\ t_5 & = & t_3 \wedge Q_1 \end{array}$$

we can simplify these to

$$\begin{array}{lcl} Q_0' & = & (t_5 \wedge B_0) \vee (t_5 \wedge t_0) \vee (t_3 \wedge B_0 \wedge t_0) \\ Q_1' & = & (t_4 \wedge B_0) \vee (t_4 \wedge t_0) \vee (t_5 \wedge B_0) \vee (t_5 \wedge t_0) \end{array}$$

▷ **S181.** a The two properties are defined as follows:

i The Hamming weight of X is the number of bits in X that are equal to 1, i.e., the number of times $X_i = 1$. This can be computed as

$$\mathrm{HW}(X) = \sum_{i=0}^{n-1} X_i.$$

ii The Hamming distance between X and Y is the number of bits in X that differ from the corresponding bit in Y, i.e., the number of times $X_i \neq Y_i$:

$$\mathrm{HD}(X, Y) = \sum_{i=0}^{n-1} X_i \oplus Y_i.$$

b There are two main approaches to constructing a flip-flop of this type; since both start with an SR-latch, the difference is mainly in how the edge-triggered behaviour is realised. Use of a primary-secondary organisation is probably the more complete solution, but a simpler alternative would be to use a pulse generator. The overall design can be described roughly as follows:

```
                                 +---+       +---+
D--+----------------------------->|   |---S-->|   |
   |                             |AND|       |NOR|-->r_0 = ~Q
   v                         +-->|   | r_1-->|   |
 +---+       +------------+  |   +---+       +---+
 |   |       |            |  |
 |NOT|  en-->| pulse gen. |--+
 |   |       |            |  |
 +---+       +------------+  |   +---+       +---+
   |                         +-->|   |---R-->|   |
   |                             |AND|       |NOR|-->r_1 = Q
   +---------------------------->|   | r_0-->|   |
                                 +---+       +---+
```

There are basically four features to note:

i An SR-latch has two inputs S and R, and two outputs Q and $\neg Q$. When

- $S = 0$, $R = 0$ the component retains Q,

- $S = 1, R = 0$ the component updates to $Q = 1$,
- $S = 0, R = 1$ the component updates to $Q = 0$,
- $S = 1, R = 1$ the component is meta-stable.

The component is level-triggered in the sense that Q is updated within the period of time when $S = 1$ or $R = 1$ (rather than when they transition to said values).

ii To provide more fine-grained control over the component, the two inputs are typically gated using (i.e., AND'ed with) an enable signal en: when $en = 0$, the latch inputs are always zero and hence it retains the same state, when $en = 1$ it can be updated as normal.

iii In order to change from the current level-triggered behaviour into an edge-triggered alternative, one approach is to use a pulse generator. The idea here is to intentionally create a mismatch in propagation delay into the inputs of an AND gate: each time en changes, the result is that we see a small pulse on the output of the AND gate. Provided this is small enough, one can argue it acts like an edge rather than a level.

iv Finally, the gated S and R inputs are tied together and controlled by one input D meaning $S = D$ and $R = \neg D$. This prevents the component being used erroneously: it can *only* retain or update the state.

c The power consumed by CMOS transistors can be decomposed into two parts: the static part (which relates to leakage) and the dynamic part (which relates to power consumed when the transistor switches). In short, a value switching (i.e., changing from one value to another) consumes much more power than staying the same. In this case, clearly we have an advantage in the all but one of the n bits in the register will stay the same; hence in terms of power consumption, storing elements of the the Gray code (versus some other sequence for example) is an advantage.

d See Solution 178.

e As an aside, a potentially neat approach here is to use a Johnson counter. This is basically an n-bit register (initialised to zero) whose content is shifted by one place on each clock edge. The new incoming, 0-th bit is computed as the NOT of the outgoing, $(n-1)$-th bit and every other bit is shifted up by one place (e.g., each i-th bit for $1 \le i < n-1$ becomes the $(i+1)$-th bit). For $n = 3$, this produces the sequence

$$\begin{array}{c} \langle 0,0,0\rangle \\ \langle 1,0,0\rangle \\ \langle 1,1,0\rangle \\ \langle 1,1,1\rangle \\ \langle 0,1,1\rangle \\ \langle 0,0,1\rangle \\ \vdots \end{array}$$

which satisfies the Hamming distances property, but does not include all possible values: for example, $\langle 1,0,1\rangle$ is not included. So this does not really answer the question in the sense that we require a component that cycles through the full 2^n-element sequence, an example of which is

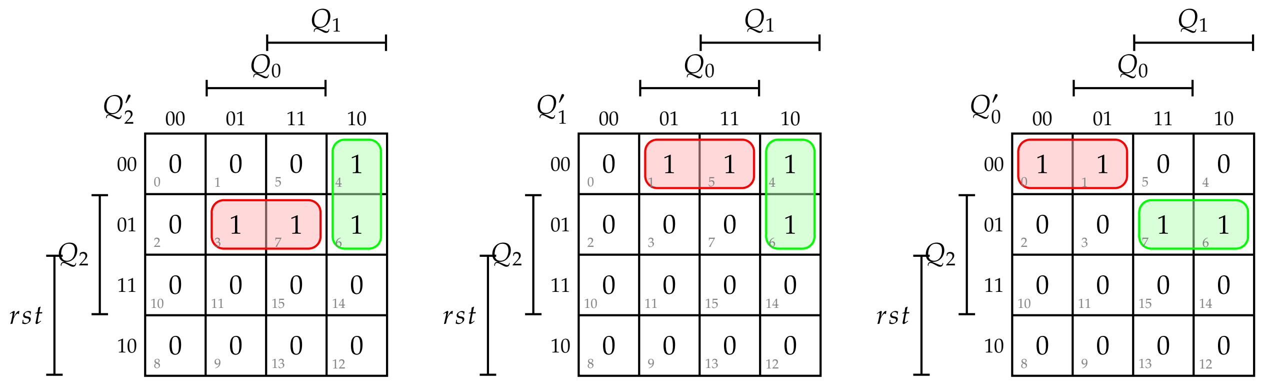
$$\begin{array}{c} \langle 0,0,0\rangle \\ \langle 1,0,0\rangle \\ \langle 1,1,0\rangle \\ \langle 0,1,0\rangle \\ \langle 0,1,1\rangle \\ \langle 1,1,1\rangle \\ \langle 1,0,1\rangle \\ \langle 0,0,1\rangle \end{array}$$

As a result, we can use an FSM-based approach based on the framework in the question above. For $n = 3$ there are $2^3 = 8$ elements in the Gray code, and so a 3-bit state $Q = \langle Q_0, Q_1, Q_2\rangle$ is enough to store the current element. The output function ω is basically free: we simply provide the current state Q as output, which is also the current element in the Gray code sequence. Based on the inputs Q and rst, the state transition function δ

can be described as follows:

rst	Q_2	Q_1	Q_0	Q'_2	Q'_1	Q'_0
0	0	0	0	0	0	1
0	0	0	1	0	1	1
0	0	1	1	0	1	0
0	0	1	0	1	1	0
0	1	1	0	1	1	1
0	1	1	1	1	0	1
0	1	0	1	1	0	0
0	1	0	0	0	0	0
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	1	0	0	0
1	0	1	0	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0
1	1	0	1	0	0	0
1	1	0	0	0	0	0

From this truth table we can (more easily that usual perhaps) extract Karnaugh maps for each bit of the next state Q'



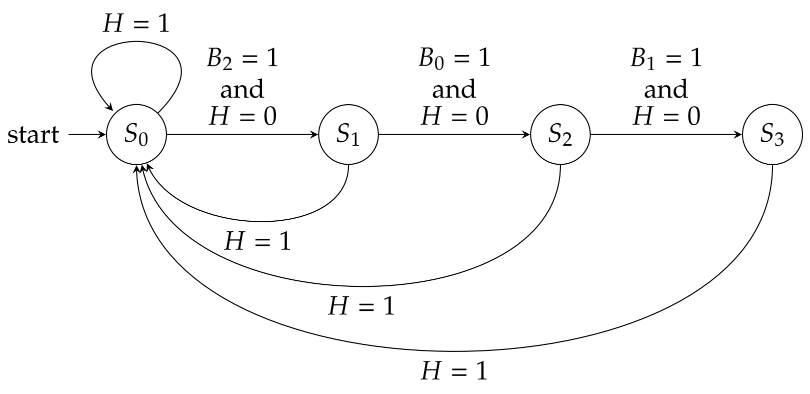
and hence Boolean expressions

$$
\begin{array}{lcccccccccccc}
Q'_2 & = & (& \neg rst & \wedge & Q_2 & & & & Q_0 &) & \vee \\
 & & (& \neg rst & & & \wedge & Q_1 & & \neg Q_0 &) & \\
Q'_1 & = & (& \neg rst & \wedge & \neg Q_2 & & & \wedge & Q_0 &) & \vee \\
 & & (& \neg rst & & & \wedge & Q_1 & \wedge & \neg Q_0 &) & \\
 & & & & & & & & & & & \\
Q'_0 & = & (& \neg rst & \wedge & \neg Q_2 & \wedge & \neg Q_1 & & &) & \vee \\
 & & (& \neg rst & \wedge & Q_2 & \wedge & Q_1 & & &) &
\end{array}
$$

Placing the associated combinatorial logic and a 3-bit, D-type flip-flop based register to store Q into the generic framework, we end up with a component that cycles through our 3-bit Gray code sequence under control of a clock signal.

▷ **S182.** a See Solution 178.

b There are a few different ways to interpret some parts of the problem definition, but one reasonable approach is as follows:



Essentially, the idea is that by pressing buttons we advance from the stating state S_0 toward the final state S_3 (as long as the handle is not turned, which means we go back to the start): when in S_3 the door is unlocked, otherwise it remains locked. In particular, if the buttons are pressed in the wrong order we get "stuck" half way along the sequence and never reach S_3. For example if B_1 is pressed while in state S_1, the FSM does not (and cannot ever) transition into S_2 since the button stays pressed: the only way to "unstick" the FSM is to turn the handle, reset the mechanism and start again.

There are four states in total; since $2^2 = 4$ we can represent the current state Q as a 2-bit integer, making the concrete assignment

$$\begin{array}{lcl} S_0 & \mapsto & \langle 0,0 \rangle \\ S_1 & \mapsto & \langle 1,0 \rangle \\ S_2 & \mapsto & \langle 0,1 \rangle \\ S_3 & \mapsto & \langle 1,1 \rangle \end{array}$$

The FSM diagram can be expressed as a truth table, particular to this P, which captures the various transitions:

H	B_2	B_1	B_0	Q_1	Q_0	Q'_1	Q'_0
0	0	?	?	0	0	0	0
0	1	?	?	0	0	0	1
1	?	?	?	0	0	0	0
0	?	?	0	0	1	0	1
0	?	?	1	0	1	1	0
1	?	?	?	0	1	0	0
0	?	0	?	1	0	1	0
0	?	1	?	1	0	1	1
1	?	?	?	1	0	0	0
0	?	?	?	1	1	1	1
1	?	?	?	1	1	0	0

Implementing this truth table via a 6-input Karnaugh map is a little more tricky than with fewer inputs; instead, we simply derive the expressions by inspection (i.e., by forming a term for each 1 entry in a given output) to yield

$$\begin{array}{lcl} Q'_0 & = & (\neg H \wedge B_2 \wedge \neg Q_1 \wedge \neg Q_0) \vee \\ & & (\neg H \wedge \neg B_0 \wedge \neg Q_1 \wedge Q_0) \vee \\ & & (\neg H \wedge B_1 \wedge Q_1 \wedge \neg Q_0) \vee \\ & & (\neg H \wedge Q_1 \wedge Q_0) \\ & & \\ Q'_1 & = & (\neg H \quad B_0 \wedge \neg Q_1 \wedge Q_0) \vee \\ & & (\neg H \wedge \neg B_1 \wedge Q_1 \wedge \neg Q_0) \vee \\ & & (\neg H \wedge B_1 \wedge Q_1 \wedge \neg Q_0) \vee \\ & & (\neg H \wedge Q_1 \wedge Q_0) \end{array}$$

with minor optimisation possible thereafter. Returning to the framework, the idea is then that we

i instantiate the middle box with a 2-bit register, using D-type flip-flops for example, to store Q,
ii instantiate the top box to implement δ using the equations above,
iii instantiate the bottom box to implement ω using the equation

$$L = \neg(Q_1 \wedge Q_0)$$

so the door is locked unless the FSM is in state S_3.

c The purpose of a clock signal is to control the FSM, advancing it through steps (i.e., transitions) with all components synchronised. However, the only updates of state occur on positive transistors of B_i or H. That is, the FSM only chances state when one of the buttons is pressed, or the handle turned: in each case, this means the associated value transitions from 0 to 1. As a result, one *could* argue the expression

$$H \vee B_0 \vee B_1 \vee B_1 \vee B_2$$

can be used to advance the FSM (i.e., latch the next state produced by the transition function), rather than "polling" the buttons and handle at each clock edge to see if their value has changed.

d Among various valid answers, the following are clear:

i The content stored in an SRAM memory is lost if the power supply is removed: such devices depend on a power supply so transistors used to maintain the stored content can operate. In the context of the proposed approach, this means if a power cut occurs, for example, then the password will be "forgotten" by the lock.

ii When the power supply comes back online the password might be essentially random due to the way SRAMs work. If this is not true however, and the SRAM is initialised into a predictable value (e.g., all zero), this could offer an attractive way to bypass the security offered!

iii Given physical access to the lock, one might simply read the password out of the SRAM. With an FSM hard-wired to a single password, the analogue is arguably harder: one would need to (invasively) reverse engineer the gate layout and connectivity, then the FSM design.

Less attractive answers include degradation of performance (e.g., as a result of SRAM access latency) or increase in cost: given constraints of the application, neither seems particular important. For example the access latency of SRAM memory is measured in small fractions of a second; although arguably true in general, from the perspective of a human user of the door lock the delay will be imperceptible.

e This is quite open-ended, but one reasonable approach would be as follows:

i This is a slightly loaded question in that it implies some alteration *is* needed; as such, marks might typically be given for identifying the underlying reason, and explaining each aspect of the proposed alteration.

The crucial point to realise is testing implementations of δ and ω, for example, depends on being able to set (and possibly inspect) the state Q which acts as input to both. An example technology to allow this would be JTAG, which requires an additional interface (inc. TDI, TDO, TCLK, TMS and TRST pins) and also injection of a scan chain to access all flip-flops. This allows the test process to scan a value into Q one bit at a time, run the system normally, then scan out Q to test it.

ii The idea would be to place each system under the control of a test stimulus that automates a series of tests: the test stimulus has access to all inputs (i.e., the JTAG interface, each button and the handle) and outputs (e.g., the JTAG interface, and the lock mechanism), and is tasked with making sure the overall behaviour matches some reference.

In this context, the number of states, inputs and outputs is small enough that a brute force approach is reasonable; this is also motivated by the fact there are no obvious boundary cases and so on. The strategy would therefore be: for each entry in the truth table

- put the device in test mode,
- scan-in the state Q and drive each B_i with the associated values,
- put the device in normal mode, and force an update of the FSM using the clock signal,
- put the device in test mode,
- check the value of L matches that expected,
- scan-out and check the value of Q matches that expected.

An alternative answer might focus on some form of BIST, but in essence this just places all the above *inside* the system rather than viewing it as something done externally.

▷ **S183.** a At least three advantages (or disadvantages, depending on which way around you view the options) are evident:

- With option one, extracting each digit of the current PIN to form a guess is trivial; with option two this is much harder, in that we need to take the integer P and decompose it into a decimal representation (through repeated division and modular reduction).
- With option one, incrementing the current PIN is harder (since the addition is in decimal); with option two this is much easier, in that we can simply use a standard integer adder.
- With option one, the total storage requirement is $4 \cdot 4 = 16$ bits; with option two this is only 14 bits, since $2^{14} = 16384 > 9999$.

Based on this, and reading ahead to the next question, the decimal representation seems more attractive: designing a decimal adder is significantly easier than a binary divider.

b Given the choice, and although both options are viable, we focus on a design for the second, decimal representation: this is simpler by some way, so the expected answer. At a high-level, the component can be described as follows:

```
      P_3             P_2             P_1             P_0
       |               |               |               |
G_3 <--+        G_2 <--+        G_1 <--+        G_0 <--+
       |               |               |               |
       v               v               v               v
 +-----------+   +-----------+   +-----------+   +-----------+
 |     x     |   |     x     |   |     x     |   |     x     |
 |           |   |           |   |           |   |           |
 |co  add  ci|<--|co  add  ci|<--|co  add  ci|<--|co  add  ci|<-- 1
 |           |   |           |   |           |   |           |
 |     r     |   |     r     |   |     r     |   |     r     |
 +-----------+   +-----------+   +-----------+   +-----------+
       |               |               |               |
       v               v               v               v
     P'_3            P'_2            P'_1            P'_0
```

$P_i = G_i$ so production of the guess is trivial; the other output is a little harder. The basic idea is to use something similar to a ripple-carry adder. Each i-th cell takes a decimal digit P_i and a carry-in from the previous, $(i-1)$-th cell; it produces a decimal digit P'_i and a carry-out into the next $(i+1)$-th cell. The difference from a binary ripple-carry adder then is that it only accepts one digit rather than two as input (since it increments P rather than computes a general-purpose addition), plus it obviously works with decimal rather than binary digits.

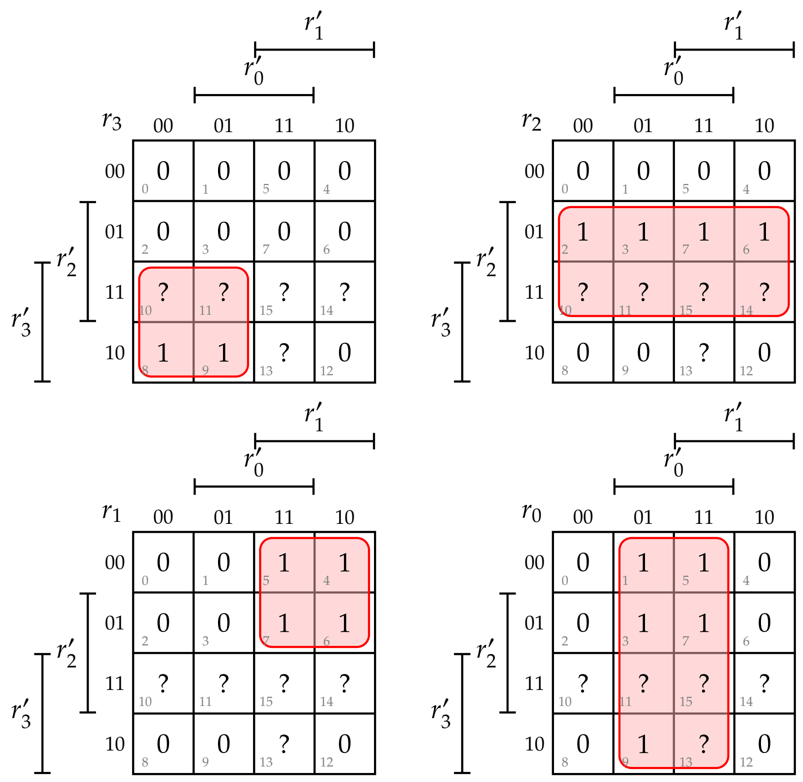
There are various ways to approach the design of each decimal adder cell, but perhaps the most straightforward uses two stages:

```
           x_3              x_2              x_1              x_0
            |                |                |                |
            v                v                v                v
      +------------+   +------------+   +------------+   +------------+
      |     x      |   |     x      |   |     x      |   |     x      |
      |            |   |            |   |            |   |            |
      | co  ha  ci |<--| co  ha  ci |<--| co  ha  ci |<--| co  ha  ci |<-- 0
      |            |   |            |   |            |   |            |
      |     r      |   |     r      |   |     r      |   |     r      |
      +------------+   +------------+   +------------+   +------------+
            |                |                |                |
          r'_3             r'_2             r'_1             r'_0
            |                |                |                |
            v                v                v                v
      +---------------------------------------------------------------+
      |                                                               |
co <--|                       modular reduction                       |
      |                                                               |
      +---------------------------------------------------------------+
            |                |                |                |
           r_3              r_2              r_1              r_0
            |                |                |                |
            v                v                v                v
```

The first stage computes an integer sum $r' = x + ci$. Although this could be realised using a standard ripple-carry adder, we can make a more problem-specific improvement: a ripple-carry adder normally uses full-adder cells that compute $x + y + ci$, but we lack the second input y. Thus we can use half-adder cells instead, which use half the number of gates; we assume such a half-adder is available as a standard component. The second stage takes $r' = x + ci$ as input, and produces the outputs r and co, implementing the modular reduction. The range of each input means $0 \le r' < 11$, or equivalently that cases where $r' > 10$ are impossible. We can describe the behaviour of the stage using the following truth table:

r'_3	r'_3	r'_3	r'_3	co	r_3	r_2	r_1	r_0
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	1	0
0	0	1	1	0	0	0	1	1
0	1	0	0	0	0	1	0	0
0	1	0	1	0	0	1	0	1
0	1	1	0	0	0	1	1	0
0	1	1	1	0	0	1	1	1
1	0	0	0	0	1	0	0	0
1	0	0	1	0	1	0	0	1
1	0	1	0	1	0	0	0	0
1	0	1	1	**?**	**?**	**?**	**?**	**?**
1	1	0	0	**?**	**?**	**?**	**?**	**?**
1	1	0	1	**?**	**?**	**?**	**?**	**?**
1	1	1	0	**?**	**?**	**?**	**?**	**?**
1	1	1	1	**?**	**?**	**?**	**?**	**?**

As such, we can produce a set of Karnaugh maps



which translate fairly easily into Boolean expressions

$$\begin{array}{rclcccc} r_3 & = & r'_3 & \wedge & & & \neg r'_1 \\ r_2 & = & & & r'_2 & & \\ r_1 & = & \neg r'_3 & \wedge & & & r'_1 \\ r_0 & = & & & & & \quad r'_0 \end{array}$$

that allow implementation.

c The FSM maintains a current state Q. Given there are five states, we can represent the current state as

$$Q = \langle Q_0, Q_1, Q_2 \rangle$$

i.e., three bits (since $2^3 = 8 > 5$), so the device could store it in a register comprised of three D-type flip-flops; doing so accepts there are three unused state representations.

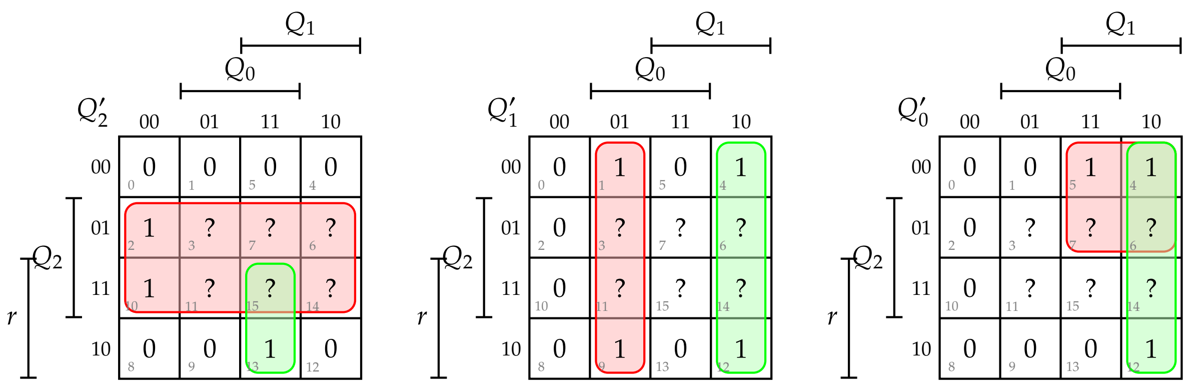
We can represent the states as follows

$$\begin{array}{rcl} S_0 & = & \langle 0,0,0 \rangle \\ S_1 & = & \langle 1,0,0 \rangle \\ S_2 & = & \langle 0,1,0 \rangle \\ S_3 & = & \langle 1,1,0 \rangle \\ S_4 & = & \langle 0,0,1 \rangle \end{array}$$

and therefore formulate a tabular transition function δ:

b	r	Q_2	Q_1	Q_0	Q'_2	Q'_1	Q'_0
0	?	0	0	0	0	0	0
1	?	0	0	0	0	0	1
?	?	0	0	1	0	1	0
?	?	0	1	0	0	1	1
?	0	0	1	1	0	0	1
?	1	0	1	1	1	0	0
?	?	1	0	0	1	0	0

Turning these into Karnaugh maps and then Boolean expressions is a little tricky due to the need for five inputs. To cope, we assume there are no transitions from S_0 and ignore b, then patch the equation for Q'_0 (the only bit of the next state influenced by moving out of S_0) appropriately. That is, we get the following



which then translate into

$$\begin{array}{rclcl} Q'_2 & = & (\quad r \wedge \quad Q_1 \wedge Q_0 \quad) & \vee \\ & & (\quad Q_2 \quad) & \\ Q'_1 & = & (\quad \neg Q_1 \wedge Q_0 \quad) & \vee \\ & & (\quad Q_1 \wedge \neg Q_0 \quad) & \\ Q'_0 & = & (\quad b \wedge \neg Q_2 \wedge \neg Q_1 \wedge \neg Q_0 \quad) & \vee \\ & & (\quad \neg r \wedge Q_1 \quad) & \vee \\ & & (\quad Q_1 \wedge \neg Q_0 \quad) & \end{array}$$

Part VIII: Computational machines: Register Machines (RMs)

▷ **S184.** a First we need to decode the machine code program: using Figure **??**, we find that

$$\begin{array}{rclcl} 0A3_{(16)} & = & 010100011_{(2)} & \mapsto & \mathsf{L}_0 : \textbf{if } \mathsf{R}_2 = 0 \textbf{ then goto } \mathsf{L}_3 \textbf{ else goto } \mathsf{L}_1 \\ 060_{(16)} & = & 001100000_{(2)} & \mapsto & \mathsf{L}_1 : \mathsf{R}_2 \leftarrow \mathsf{R}_2 - 1 \textbf{ then goto } \mathsf{L}_2 \\ 080_{(16)} & = & 010000000_{(2)} & \mapsto & \mathsf{L}_2 : \textbf{if } \mathsf{R}_0 = 0 \textbf{ then goto } \mathsf{L}_0 \textbf{ else goto } \mathsf{L}_3 \\ 097_{(16)} & = & 010010111_{(2)} & \mapsto & \mathsf{L}_3 : \textbf{if } \mathsf{R}_1 = 0 \textbf{ then goto } \mathsf{L}_7 \textbf{ else goto } \mathsf{L}_4 \\ 050_{(16)} & = & 001010000_{(2)} & \mapsto & \mathsf{L}_4 : \mathsf{R}_1 \leftarrow \mathsf{R}_1 - 1 \textbf{ then goto } \mathsf{L}_5 \\ 020_{(16)} & = & 000100000_{(2)} & \mapsto & \mathsf{L}_5 : \mathsf{R}_2 \leftarrow \mathsf{R}_2 + 1 \textbf{ then goto } \mathsf{L}_6 \\ 083_{(16)} & = & 010000011_{(2)} & \mapsto & \mathsf{L}_6 : \textbf{if } \mathsf{R}_0 = 0 \textbf{ then goto } \mathsf{L}_3 \textbf{ else goto } \mathsf{L}_7 \\ 0C0_{(16)} & = & 011000000_{(2)} & \mapsto & \mathsf{L}_7 : \mathsf{halt} \end{array}$$

Next we can produce a trace of execution for the program: starting with the initial configuration given, we find that

$$\begin{array}{rcl}
C_0 & = & (0,0,2,1,0) \\
\mathsf{L}_0 & \rightsquigarrow & \textbf{if } \mathsf{R}_2 = 0 \textbf{ then goto } \mathsf{L}_3 \textbf{ else goto } \mathsf{L}_1 \\
C_1 & = & (1,0,2,1,0) \\
\mathsf{L}_1 & \rightsquigarrow & \mathsf{R}_2 \leftarrow \mathsf{R}_2 - 1 \textbf{ then goto } \mathsf{L}_2 \\
C_2 & = & (2,0,2,0,0) \\
\mathsf{L}_2 & \rightsquigarrow & \textbf{if } \mathsf{R}_0 = 0 \textbf{ then goto } \mathsf{L}_0 \textbf{ else goto } \mathsf{L}_3 \\
C_3 & = & (0,0,2,0,0) \\
\mathsf{L}_0 & \rightsquigarrow & \textbf{if } \mathsf{R}_2 = 0 \textbf{ then goto } \mathsf{L}_3 \textbf{ else goto } \mathsf{L}_1 \\
C_4 & = & (3,0,2,0,0) \\
\mathsf{L}_3 & \rightsquigarrow & \textbf{if } \mathsf{R}_1 = 0 \textbf{ then goto } \mathsf{L}_7 \textbf{ else goto } \mathsf{L}_4 \\
C_5 & = & (4,0,2,0,0) \\
\mathsf{L}_4 & \rightsquigarrow & \mathsf{R}_1 \leftarrow \mathsf{R}_1 - 1 \textbf{ then goto } \mathsf{L}_5 \\
C_6 & = & (5,0,1,0,0) \\
\mathsf{L}_5 & \rightsquigarrow & \mathsf{R}_2 \leftarrow \mathsf{R}_2 + 1 \textbf{ then goto } \mathsf{L}_6 \\
C_7 & = & (6,0,1,1,0) \\
\mathsf{L}_6 & \rightsquigarrow & \textbf{if } \mathsf{R}_0 = 0 \textbf{ then goto } \mathsf{L}_3 \textbf{ else goto } \mathsf{L}_7 \\
C_8 & = & (3,0,1,1,0) \\
\mathsf{L}_3 & \rightsquigarrow & \textbf{if } \mathsf{R}_1 = 0 \textbf{ then goto } \mathsf{L}_7 \textbf{ else goto } \mathsf{L}_4 \\
C_9 & = & (4,0,1,1,0) \\
\mathsf{L}_4 & \rightsquigarrow & \mathsf{R}_1 \leftarrow \mathsf{R}_1 - 1 \textbf{ then goto } \mathsf{L}_5 \\
C_{10} & = & (5,0,0,1,0) \\
\mathsf{L}_5 & \rightsquigarrow & \mathsf{R}_2 \leftarrow \mathsf{R}_2 + 1 \textbf{ then goto } \mathsf{L}_6 \\
C_{11} & = & (6,0,0,2,0) \\
\mathsf{L}_6 & \rightsquigarrow & \textbf{if } \mathsf{R}_0 = 0 \textbf{ then goto } \mathsf{L}_3 \textbf{ else goto } \mathsf{L}_7 \\
C_{12} & = & (3,0,0,2,0) \\
\mathsf{L}_3 & \rightsquigarrow & \textbf{if } \mathsf{R}_1 = 0 \textbf{ then goto } \mathsf{L}_7 \textbf{ else goto } \mathsf{L}_4 \\
C_{13} & = & (7,0,0,2,0) \\
\mathsf{L}_7 & \rightsquigarrow & \text{halt}
\end{array}$$

where the final configuration halts execution.

As a result, stating that the program will "copy the value in R_1 into R_2, clearing the value in R_1" is the best match. Note that the program itself is in two parts: L_0 to L_2 clear (or zero) R_2, and L_3 to L_6 move R_1 into R_2. Also note that it depends on having $\mathsf{R}_0 = 0$, allowing the construction of unconditional branches in L_2 and L_6.

b First we need to decode the machine code program: using Figure **??**, we find that

$$\begin{array}{rclcl}
0B3_{(16)} & = & 010110011_{(2)} & \mapsto & \mathsf{L}_0 : \textbf{if } \mathsf{R}_3 = 0 \textbf{ then goto } \mathsf{L}_3 \textbf{ else goto } \mathsf{L}_1 \\
070_{(16)} & = & 001110000_{(2)} & \mapsto & \mathsf{L}_1 : \mathsf{R}_3 \leftarrow \mathsf{R}_3 - 1 \textbf{ then goto } \mathsf{L}_2 \\
080_{(16)} & = & 010000000_{(2)} & \mapsto & \mathsf{L}_2 : \textbf{if } \mathsf{R}_0 = 0 \textbf{ then goto } \mathsf{L}_0 \textbf{ else goto } \mathsf{L}_3 \\
097_{(16)} & = & 010010111_{(2)} & \mapsto & \mathsf{L}_3 : \textbf{if } \mathsf{R}_1 = 0 \textbf{ then goto } \mathsf{L}_7 \textbf{ else goto } \mathsf{L}_4 \\
030_{(16)} & = & 000110000_{(2)} & \mapsto & \mathsf{L}_4 : \mathsf{R}_3 \leftarrow \mathsf{R}_3 + 1 \textbf{ then goto } \mathsf{L}_5 \\
050_{(16)} & = & 001010000_{(2)} & \mapsto & \mathsf{L}_5 : \mathsf{R}_1 \leftarrow \mathsf{R}_1 - 1 \textbf{ then goto } \mathsf{L}_6 \\
083_{(16)} & = & 010000011_{(2)} & \mapsto & \mathsf{L}_6 : \textbf{if } \mathsf{R}_0 = 0 \textbf{ then goto } \mathsf{L}_3 \textbf{ else goto } \mathsf{L}_7 \\
0AB_{(16)} & = & 010101011_{(2)} & \mapsto & \mathsf{L}_7 : \textbf{if } \mathsf{R}_2 = 0 \textbf{ then goto } \mathsf{L}_{11} \textbf{ else goto } \mathsf{L}_8 \\
030_{(16)} & = & 000110000_{(2)} & \mapsto & \mathsf{L}_8 : \mathsf{R}_3 \leftarrow \mathsf{R}_3 + 1 \textbf{ then goto } \mathsf{L}_9 \\
060_{(16)} & = & 001100000_{(2)} & \mapsto & \mathsf{L}_9 : \mathsf{R}_2 \leftarrow \mathsf{R}_2 - 1 \textbf{ then goto } \mathsf{L}_{10} \\
087_{(16)} & = & 010000111_{(2)} & \mapsto & \mathsf{L}_{10} : \textbf{if } \mathsf{R}_0 = 0 \textbf{ then goto } \mathsf{L}_7 \textbf{ else goto } \mathsf{L}_{11} \\
0C0_{(16)} & = & 011000000_{(2)} & \mapsto & \mathsf{L}_{11} : \text{halt}
\end{array}$$

Next we can produce a trace of execution for the program: starting with the initial configuration given, we find that

$$\begin{array}{rcl}
C_0 & = & (0,0,3,2,1) \\
\mathsf{L}_0 & \rightsquigarrow & \textbf{if } \mathsf{R}_3 = 0 \textbf{ then goto } \mathsf{L}_3 \textbf{ else goto } \mathsf{L}_1 \\
C_1 & = & (1,0,3,2,1) \\
\mathsf{L}_1 & \rightsquigarrow & \mathsf{R}_3 \leftarrow \mathsf{R}_3 - 1 \textbf{ then goto } \mathsf{L}_2 \\
C_2 & = & (2,0,3,2,0) \\
\mathsf{L}_2 & \rightsquigarrow & \textbf{if } \mathsf{R}_0 = 0 \textbf{ then goto } \mathsf{L}_0 \textbf{ else goto } \mathsf{L}_3 \\
C_3 & = & (0,0,3,2,0) \\
\mathsf{L}_0 & \rightsquigarrow & \textbf{if } \mathsf{R}_3 = 0 \textbf{ then goto } \mathsf{L}_3 \textbf{ else goto } \mathsf{L}_1 \\
C_4 & = & (3,0,3,2,0) \\
\mathsf{L}_3 & \rightsquigarrow & \textbf{if } \mathsf{R}_1 = 0 \textbf{ then goto } \mathsf{L}_7 \textbf{ else goto } \mathsf{L}_4 \\
C_5 & = & (4,0,3,2,0) \\
\mathsf{L}_4 & \rightsquigarrow & \mathsf{R}_3 \leftarrow \mathsf{R}_3 + 1 \textbf{ then goto } \mathsf{L}_5 \\
C_6 & = & (5,0,3,2,1) \\
\mathsf{L}_5 & \rightsquigarrow & \mathsf{R}_1 \leftarrow \mathsf{R}_1 - 1 \textbf{ then goto } \mathsf{L}_6 \\
C_7 & = & (6,0,2,2,1) \\
\mathsf{L}_6 & \rightsquigarrow & \textbf{if } \mathsf{R}_0 = 0 \textbf{ then goto } \mathsf{L}_3 \textbf{ else goto } \mathsf{L}_7 \\
C_8 & = & (3,0,2,2,1) \\
\mathsf{L}_3 & \rightsquigarrow & \textbf{if } \mathsf{R}_1 = 0 \textbf{ then goto } \mathsf{L}_7 \textbf{ else goto } \mathsf{L}_4 \\
C_9 & = & (4,0,2,2,1) \\
\mathsf{L}_4 & \rightsquigarrow & \mathsf{R}_3 \leftarrow \mathsf{R}_3 + 1 \textbf{ then goto } \mathsf{L}_5 \\
C_{10} & = & (5,0,2,2,2) \\
\mathsf{L}_5 & \rightsquigarrow & \mathsf{R}_1 \leftarrow \mathsf{R}_1 - 1 \textbf{ then goto } \mathsf{L}_6 \\
C_{11} & = & (6,0,1,2,2) \\
\mathsf{L}_6 & \rightsquigarrow & \textbf{if } \mathsf{R}_0 = 0 \textbf{ then goto } \mathsf{L}_3 \textbf{ else goto } \mathsf{L}_7 \\
C_{12} & = & (3,0,1,2,2) \\
\mathsf{L}_3 & \rightsquigarrow & \textbf{if } \mathsf{R}_1 = 0 \textbf{ then goto } \mathsf{L}_7 \textbf{ else goto } \mathsf{L}_4 \\
C_{13} & = & (4,0,1,2,2) \\
\mathsf{L}_4 & \rightsquigarrow & \mathsf{R}_3 \leftarrow \mathsf{R}_3 + 1 \textbf{ then goto } \mathsf{L}_5 \\
C_{14} & = & (5,0,1,2,3) \\
\mathsf{L}_5 & \rightsquigarrow & \mathsf{R}_1 \leftarrow \mathsf{R}_1 - 1 \textbf{ then goto } \mathsf{L}_6 \\
C_{15} & = & (6,0,0,2,3) \\
\mathsf{L}_6 & \rightsquigarrow & \textbf{if } \mathsf{R}_0 = 0 \textbf{ then goto } \mathsf{L}_3 \textbf{ else goto } \mathsf{L}_7 \\
C_{16} & = & (3,0,0,2,3) \\
\mathsf{L}_3 & \rightsquigarrow & \textbf{if } \mathsf{R}_1 = 0 \textbf{ then goto } \mathsf{L}_7 \textbf{ else goto } \mathsf{L}_4 \\
C_{17} & = & (7,0,0,2,3) \\
\mathsf{L}_7 & \rightsquigarrow & \textbf{if } \mathsf{R}_2 = 0 \textbf{ then goto } \mathsf{L}_{11} \textbf{ else goto } \mathsf{L}_8 \\
C_{18} & = & (8,0,0,2,3) \\
\mathsf{L}_8 & \rightsquigarrow & \mathsf{R}_3 \leftarrow \mathsf{R}_3 + 1 \textbf{ then goto } \mathsf{L}_9 \\
C_{19} & = & (9,0,0,2,4) \\
\mathsf{L}_9 & \rightsquigarrow & \mathsf{R}_2 \leftarrow \mathsf{R}_2 - 1 \textbf{ then goto } \mathsf{L}_{10} \\
C_{20} & = & (10,0,0,1,4) \\
\mathsf{L}_{10} & \rightsquigarrow & \textbf{if } \mathsf{R}_0 = 0 \textbf{ then goto } \mathsf{L}_7 \textbf{ else goto } \mathsf{L}_{11} \\
C_{21} & = & (7,0,0,1,4) \\
\mathsf{L}_7 & \rightsquigarrow & \textbf{if } \mathsf{R}_2 = 0 \textbf{ then goto } \mathsf{L}_{11} \textbf{ else goto } \mathsf{L}_8 \\
C_{22} & = & (8,0,0,1,4) \\
\mathsf{L}_8 & \rightsquigarrow & \mathsf{R}_3 \leftarrow \mathsf{R}_3 + 1 \textbf{ then goto } \mathsf{L}_9 \\
C_{23} & = & (9,0,0,1,5) \\
\mathsf{L}_9 & \rightsquigarrow & \mathsf{R}_2 \leftarrow \mathsf{R}_2 - 1 \textbf{ then goto } \mathsf{L}_{10} \\
C_{24} & = & (10,0,0,0,5) \\
\mathsf{L}_{10} & \rightsquigarrow & \textbf{if } \mathsf{R}_0 = 0 \textbf{ then goto } \mathsf{L}_7 \textbf{ else goto } \mathsf{L}_{11} \\
C_{25} & = & (7,0,0,0,5) \\
\mathsf{L}_7 & \rightsquigarrow & \textbf{if } \mathsf{R}_2 = 0 \textbf{ then goto } \mathsf{L}_{11} \textbf{ else goto } \mathsf{L}_8 \\
C_{26} & = & (11,0,0,0,5) \\
\mathsf{L}_{11} & \rightsquigarrow & \text{halt}
\end{array}$$

where the final configuration halts execution.

As a result, stating that the program will "add the values in R_1 and R_2, setting R_3 to reflect the result" is the best match. Note that the program itself is in three parts: L_0 to L_2 clear (or zero) R_3, L_3 to L_6 add R_1 to R_3, L_7 to L_{10} add R_2 to R_3. Also note that it depends on having $\mathsf{R}_0 = 0$, allowing the construction of unconditional branches in L_2, L_6, and L_{10}.

▷ **S185.** First, note that

$$0A5_{(16)} \equiv 000010100101_{(2)} \mapsto 010100101_{(2)}.$$

We can see that the (red) opcode determines the instruction type, i.e.,

$$\textbf{if } \mathsf{R}_{addr} = 0 \textbf{ then goto } \mathsf{L}_{target} \textbf{ else goto } \mathsf{L}_{i+1}.$$

More specifically, the (green) register address and the (blue) branch target address mean the instruction semantics are

$$\textbf{if } \mathsf{R}_{2} = 0 \textbf{ then goto } \mathsf{L}_{5} \textbf{ else goto } \mathsf{L}_{i+1},$$

i.e., if register 2 equals 0 then goto instruction 5, else goto instruction $i + 1$.

▷ **S186.** Once fetched, the instruction $inst = 100111001_{(2)}$ is provided as input to the decoder: based on the implementation given, the decoder will therefore produce

- $op = 0_{(10)}$ because $inst_{8,\dots6} = 100_{(2)} = 4_{(10)}$,
- $wr = 1_{(10)}$ because $inst_{8,\dots6} = 100_{(2)} = 4_{(10)}$,
- $addr = 3_{(10)}$ because $inst_{5,\dots4} = 11_{(2)} = 3_{(10)}$,
- $target = 9_{(10)}$ because $inst_{3,\dots0} = 1001_{(2)} = 9_{(10)}$,
- $jmp = 0_{(10)}$ because $\neg inst_8 \wedge inst_7 \wedge \neg inst_6 = 0 \wedge 0 \wedge 1 = 0$,
- $halt = 0_{(10)}$ because $\neg inst_8 \wedge inst_7 \wedge inst_6 = 0 \wedge 0 \wedge 0 = 0$.

as output. Looking then at the data- and control-path to assess how these outputs are used to execute the instruction, we conclude that

- since $addr = 3_{(10)}$, the register R_3 is read from,
- since $op = 0_{(10)}$, this value is discarded and a 0 output by the multiplexer is written into register R',
- since $wr = 1_{(10)}$ register R' is written into register R_3,
- since $halt = 0_{(10)}$ the counter machine does not halt,
- since $jmp = 0_{(10)}$, the program counter is incremented as normal (i.e., not set to $target$, which in fact is unused).

In general then, this instruction will writes 0 into register R_{addr}; given $addr = 3$ here,

$$\mathsf{L}_i : \mathsf{R}_3 \leftarrow 0 \textbf{ then goto } \mathsf{L}_{i+1}$$

is therefore the correct semantics.

▷ **S187.** a Given the information provided *and* not provided in the question, we need to base our solution on some assumptions. For example, assume that the i-th configuration

$$C_i = (l, \mathsf{sp})$$

is such that the 0-th element captures the current label (cf. program counter) and the 1-st element captures the stack pointer; using this we can define the initial configuration as $C_0 = (0, 5)$ and memory content as $\mathsf{MEM} = \langle a, b, c, d, 0, 0, 0, 0, 0, \dots \rangle$, and

$$\begin{array}{lcl} \textsc{push}(\mathsf{sp}, x) & \mapsto & \mathsf{MEM}[\mathsf{sp}] \leftarrow x \; ; \; \mathsf{sp} \leftarrow \mathsf{sp} + 1 \\ \textsc{pop}(\mathsf{sp}) & \mapsto & \mathsf{sp} \leftarrow \mathsf{sp} - 1 \; ; \; x \leftarrow \mathsf{MEM}[\mathsf{sp}] \; ; \; \textbf{return} x \end{array}$$

Using this starting point, we can then produce a trace which reflects execution of the program:

- `ld 0` is translated to $t_0 \leftarrow \mathsf{MEM}[0] = a \; ; \; \textsc{push}(\mathsf{sp}, a)$, which, when applied to C_1, yields $C_2 = (1, 6)$ and $\mathsf{MEM} = \langle a, b, c, d, 0, a, 0, 0, 0, \dots \rangle$.
- `ld 1` is translated to $t_0 \leftarrow \mathsf{MEM}[1] = b \; ; \; \textsc{push}(\mathsf{sp}, b)$, which, when applied to C_2, yields $C_3 = (2, 7)$ and $\mathsf{MEM} = \langle a, b, c, d, 0, a, b, 0, 0, \dots \rangle$.
- `add` is translated to $t_0 \leftarrow \textsc{pop}(\mathsf{sp}) = b \; ; \; t_1 \leftarrow \textsc{pop}(\mathsf{sp}) = a \; ; \; \textsc{push}(a + b)$ which, when applied to C_3, yields $C_4 = (3, 6)$ and $\mathsf{MEM} = \langle a, b, c, d, 0, a + b, b, 0, 0, \dots \rangle$.
- `ld 2` is translated to $t_0 \leftarrow \mathsf{MEM}[2] = c \; ; \; \textsc{push}(\mathsf{sp}, c)$, which, when applied to C_4, yields $C_5 = (4, 7)$ and $\mathsf{MEM} = \langle a, b, c, d, 0, a + b, c, 0, 0, \dots \rangle$.
- `ld 3` is translated to $t_0 \leftarrow \mathsf{MEM}[3] = d \; ; \; \textsc{push}(\mathsf{sp}, d)$, which, when applied to C_5, yields $C_6 = (5, 8)$ and $\mathsf{MEM} = \langle a, b, c, d, 0, a + b, c, d, 0, \dots \rangle$.
- `sub` is translated to $t_0 \leftarrow \textsc{pop}(\mathsf{sp}) = d \; ; \; t_1 \leftarrow \textsc{pop}(\mathsf{sp}) = c \; ; \; \textsc{push}(c - d)$ which, when applied to C_6, yields $C_7 = (6, 7)$ and $\mathsf{MEM} = \langle a, b, c, d, 0, a + b, c - d, d, 0, \dots \rangle$.

- `mul` is translated to $t_0 \leftarrow \textsc{pop}(\mathsf{sp}) = c - d$; $t_1 \leftarrow \textsc{pop}(\mathsf{sp}) = a + b$; $\textsc{push}((a+b) \cdot (c-d))$ which, when applied to C_7, yields $C_8 = (7, 6)$ and $\mathsf{MEM} = \langle a, b, c, d, 0, (a+b) \cdot (c-d), c-d, d, 0, \ldots \rangle$.
- `st 4` is translated to $t_0 \leftarrow \textsc{pop}(\mathsf{sp}) = a + b \cdot c - d$; $\mathsf{MEM}[4] \leftarrow a + b \cdot c - d$ which, when applied to C_8, yields $C_9 = (8, 5)$ and $\mathsf{MEM} = \langle a, b, c, d, (a+b) \cdot (c-d), (a+b) \cdot (c-d), c-d, d, 0, \ldots \rangle$.

So, using the instruction semantics listed, the program will therefore compute $r = (a+b) \cdot (c-d)$, and store r in memory at address 4.

b Given the information provided *and* not provided in the question, we need to base our solution on some assumptions. For example, assume that the i-th configuration

$$C_i = (l, \mathsf{sp})$$

is such that the 0-th element captures the current label (cf. program counter) and the 1-st element captures the stack pointer; using this we can define the initial configuration as $C_0 = (0, 5)$ and memory content as $\mathsf{MEM} = \langle a, b, c, d, 0, 0, 0, 0, 0, \ldots \rangle$, and

$$\begin{array}{rcl} \textsc{push}(\mathsf{sp}, x) & \mapsto & \mathsf{MEM}[\mathsf{sp}] \leftarrow x \;;\; \mathsf{sp} \leftarrow \mathsf{sp} + 1 \\ \textsc{pop}(\mathsf{sp}) & \mapsto & \mathsf{sp} \leftarrow \mathsf{sp} - 1 \;;\; x \leftarrow \mathsf{MEM}[\mathsf{sp}] \;;\; \mathbf{return} x \end{array}$$

Using this starting point, we can then produce a trace which reflects execution of the program:

- `ld 0` is translated to $t_0 \leftarrow \mathsf{MEM}[0] = a$; $\textsc{push}(\mathsf{sp}, a)$, which, when applied to C_1, yields $C_2 = (1, 6)$ and $\mathsf{MEM} = \langle a, b, c, d, 0, a, 0, 0, 0, \ldots \rangle$.
- `ld 1` is translated to $t_0 \leftarrow \mathsf{MEM}[1] = b$; $\textsc{push}(\mathsf{sp}, b)$, which, when applied to C_2, yields $C_3 = (2, 7)$ and $\mathsf{MEM} = \langle a, b, c, d, 0, a, b, 0, 0, \ldots \rangle$.
- `st 0` is translated to $t_0 \leftarrow \textsc{pop}(\mathsf{sp}) = b$; $\mathsf{MEM}[0] \leftarrow b$ which, when applied to C_3, yields $C_4 = (3, 6)$ and $\mathsf{MEM} = \langle b, b, c, d, 0, a, b, 0, 0, \ldots \rangle$.
- `st 1` is translated to $t_0 \leftarrow \textsc{pop}(\mathsf{sp}) = a$; $\mathsf{MEM}[1] \leftarrow a$ which, when applied to C_4, yields $C_5 = (4, 5)$ and $\mathsf{MEM} = \langle b, a, c, d, 0, a, b, 0, 0, \ldots \rangle$.

So, using the instruction semantics listed, the program will therefore swap the values of a and b.

Part IX: Hardware test and debug

▷ **S188.** Based on the cited evidence, we find that

$$\begin{array}{rcl|ccccccccc|c} x & = & 50_{(16)} & \mapsto & & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & \\ y & = & 56_{(16)} & \mapsto & & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & + \\ \hline c & = & & & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \\ r' & = & A6_{(16)} & \mapsto & & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & \\ r & = & 86_{(16)} & \mapsto & & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \\ \hline \end{array}$$

so clearly the correct output $r' = x + y = A6_{(16)}$ does not match the observed output $r = 86_{(16)}$.

The first description cannot be correct: if it were Andy would observe that $r_0 = x_0 \oplus y_0 \oplus c_0 = 1 \oplus 0 \oplus 0 = 1$, but does not. The second description cannot be correct: if it were Andy would observe that $r_1 = 0$, but does not. The third description cannot be correct: if it were Andy would observe that because $c_3 = (x_2 \vee y_2) \vee ((x_2 \oplus y_2) \vee c_2) = (0 \vee 1) \vee ((0 \oplus 1) \vee 0) = 1$ it follows that $r_3 = x_3 \oplus y_3 \oplus c_3 = 0 \oplus 0 \oplus 1 = 1$, but does not. The fifth description cannot be correct: if it were Andy would observe that because $c_7 = 0$ it follows that $r_7 = x_7 \oplus y_7 \oplus c_7 = 0 \oplus 0 \oplus 0 = 0$, but does not.

This leaves the forth description as the only one consistent with the evidence available: since $c_5 = (x_4 \wedge y_4) \wedge ((x_4 \oplus y_4) \wedge c_4) = (1 \wedge 1) \wedge ((1 \oplus 1) \wedge 0) = 0$ Andy observes $r_5 = x_5 \oplus y_5 \oplus c_5 = 0 \oplus 0 \oplus 0 = 0$ which is a mismatch.

▷ **S189.** Since n is large, exhaustive search through all x, y and z is not possible; the approach must therefore be heuristic, with the aim of a high coverage level in order to achieve the highest confidence possible (trying many not not all possibilities). There are plenty of reasonable answers, two of which are as follows:

a Pre-compute many r based on random selections of x, y and z using a second, known-correct device (e.g., do it in software on a processor you have access to). These act as test vectors: feed each set of x, y and z to the circuit as input, and compare the output with the known-correct r.

b Use a BIST-type approach, by generating many random x and y and using the circuit to compute a result with known arithmetic properties, e.g., test whether

$$((x + y) - x) - y \stackrel{?}{=} 0.$$

▷ **S190.** Ignoring any carry-in for the adder, two examples are as follows:

a By selecting $x = 0000_{(2)}$ and $y = 0000_{(2)}$ as input, we can test whether $r \overset{?}{=} 0000_{(2)}$: if the test fails, e.g., $r = 0001_{(2)}$, this suggests the r_0 is not being driven correctly given $x_0 = 0$ and $y_0 = 0$.

b By selecting $x = 0010_{(2)}$ and $y = 0010_{(2)}$ as input, we can test whether $r \overset{?}{=} 0100_{(2)}$: if the test fails, e.g., $r = 0000_{(2)}$, this suggests the carry-out that *should* occur given $x_1 = 1$ and $y_1 = 1$ has not been propagated.

In both cases, failure of the associated test lends weight to the original suspicion. It might then be important however to repeat the tests (e.g., to characterise the faults as permanent or transient) and potential use other forms of testing (e.g., improve coverage via an exhaustive set of test vectors) to give a more complete, more confident outcome.

Part X: Processor design: general

▷ **S191.** a All of 1) the number of clock cycles elapsed, 2) the amount of energy consumed, 3) the order that accesses to main memory are performed, and 4) the number of accesses to main memory performed, depend on the micro-architecture so *could* plausibly differ CPU_0 and CPU_1. For example, the number of accesses to main memory performed might depend on the choice to use or not use cache memory in CPU_0 and CPU_1; the number of clock cycles elapsed might depend on the choice to use or not use pipelining in CPU_0 and CPU_1. In contrast, the number of instructions executed will likely remain the same: the same M is executed by CPU_0 and CPU_1, and, to remain compliant with the semantics of the ISA, one typically expects all instructions in M to be executed.

b All of 1) the amount of memory used to store data, 2) the amount of memory used to store instructions, 3) the content of memory before execution, and 4) the content of memory after execution, depend on M and/or the ISA; given the same M is executed by both CPU_0 and CPU_1, they will likely remain the same. In contrast, the average Cycles Per Instruction (CPI) depends on the micro-architecture, so *could* plausibly differ CPU_0 and CPU_1.

▷ **S192.** The correct order is: fetch, decode, execute, write-back.

▷ **S193.** Access to data memory, i.e., loading data from memory, is typically a task performed during execution or a dedicated memory access phase. Incrementing the program counter is typically associated with instruction fetch though this is sometimes performed in the decode phase. Thus, the first and last tasks (and possibly the third) are most associated with instruction decode: loading operands from the register file prepares register operands for execution, sign extension prepares immediate operands by expanding them from their encoded form into w-bit signed values.

▷ **S194.** **The instruction and data memories can be built in different ways.** For example one might remove logic associated with store operations from the instruction memory, or allow the instruction memory to have a different resolution (i.e., hold different sized values). This permits the designer to capitalise on differences use cases and satisfy different metrics for quality (e.g., area, power consumption or performance).

The instruction and data memories can be accessed in parallel. For example, in order to reduce the time taken to complete a given fetch-decode-execution cycle, the processor might attempt to fetch the next instruction at the same time as executing the current one. The fact that there are two memories (and hence two memory interfaces) allows this, whereas in a "pure" stored program architecture one has to perform the accesses in sequence.

Separation of instructions and data can prevent self-modifying programs. Example problems that stem from self-modifying programs include security threats; these result from the ability for an attacker to alter the program at run-time in a way the programmer did not intend.

Part XI: Processor design: Instruction Set Architecture (ISA)

▷ **S195.** The option that stands out as unlikely to be included in the ISA is execution latency. This is essentially a performance metric, measuring how long, e.g., number of cycles, an instruction takes to execute. As a result, it is a property of the micro-architecture not the ISA. Put another way, since the ISA is an interface that allows flexibility wrt. the micro-architectural implementation, it is unlike the former would include such a measure: doing so would constrain the later, reducing said flexibility.

▷ **S196.** Some of the statements describe other ways to classify a branch; for example, the last statement describes a conditional branch. Some of the statements are nonsense; for example, the term "relatively far" is subjective so has no clear meaning in this context.

The semantics of a relative branch instruction can be written as

$$\mathsf{PC} \leftarrow \mathsf{PC} + x,$$

st. the branch target (i.e., new program counter), is an offset from the current program counter value. The question does not specify whether the offset x is an immediate, but, either way, the branch target is clearly dependant on the current program counter value.

▷ **S197.** Clearly is must be possible for PC to point at any instruction in memory. There would be 2^{32} addressable bytes in the specified 32-bit address space, but 2^{30} addressable *instructions* as a result of the requirement for instructions to be aligned. As such, one could argue a 30-bit PC will suffice: since the address x of an instruction is required to satisfy $x \equiv 0 \pmod 4$, due to their fixed 32-bit (or 4-byte) length, the 2 LSBs of the full 32-bit PC will always be 0 and so need not be stored in a D-type latch.

▷ **S198.** The correct option is the one related to behaviour vs. function: the ISA cannot specify behavioural properties of micro-architectures that implement it, so cannot, for example, specify the energy consumed by execution of *any* instruction (division included). The reason for this boils down to the by-design separation of design (the ISA) from implementation (the micro-architecture), which allows flexibility in the latter. For example, one micro-architecture might focus on minimising instruction execution latency at the expense of increased energy consumption, whereas another might do the opposite: the former might be intended for server-class micro-processors, whereas the latter might be intended for embedded-class micro-processors. To support this, the ISA cannot assume or dictate one or the other. Put another way, and for example, it can specify *what* execution of a division instruction means, but not *how* that meaning is

▷ **S199.** For 16 general-purpose registers, 4-bit register addresses would be required since $2^4 = 16$; 3 such addresses are required by the arithmetic instructions, meaning $3 \cdot 4 = 12$ bits in total. As a result, there are $16 - 12 = 4$ bits remaining for an opcode: this suggests the ISA specification can include a maximum of $2^4 = 16$ such arithmetic instructions. In reality of, actually using this maximum makes little sense: there would be no encoding space left to support other instruction classes (e.g., memory access).

▷ **S200.** The RHS of the diagram, after execution of the instruction, shows:

- The program counter, i.e., GPR[15], has incremented by 4, i.e., 1 instruction, st. it has the value $00000100_{(16)} = 256_{(10)}$. Therefore, the instruction cannot have been an unconditional branch to address $100_{(10)} = 00000064_{(16)}$.
- The memory contents shown has not changed. Therefore, the instruction cannot have been a store of GPR[0] (irrespective of the addressing mode).
- Although GPR[2] $= 00000086_{(16)} = 134_{(10)}$ is equal to the sum of GPR[1] $= 00000082_{(16)} = 130_{(10)}$ and GPR[2] $= 00000004_{(16)} = 4_{(10)}$, the contents of GPR[0] has *also* changed. Therefore, the instruction cannot have been an addition.
- GPR[2] and GPR[1] are not equal either before or after the instruction is executed. Therefore, the instruction cannot have been a move of GPR[1] into GPR[2].

The remaining instruction `ldrh r0, [r1], #4` is a load, with some specific features: 1) it loads a (zero extended) 16-bit half-word, and 2) it uses an auto-indexed addressing mode st. the address in GPR[1] is (post-)incremented by $4_{(10)}$. These facts match the state of both general-purpose registers and memory, so is the correct answer.

▷ **S201.** If the number of general-purpose registers is halved, then the number of bits required for each register address is decreased by one: before there are 32 general-purpose registers requiring a 5-bit address, whereas afterwards there are 16 general-purpose registers requiring a 4-bit address. This change leads to there being 3 and 2 unused bits in the R- and I-type formats respectively.

Some of the statements cannot be true, and some could be but are not likely to be true. We can consider the statements one-by-one:

- There could be a greater number of R- or I-type instructions. This is unlikely, however, because it implies the opcode field will become unaligned between fields. Either way, the "must" term is too strong: there is no requirement for this to be the case, even if it were possible.

- There could be a greater number of R-type instructions, but the 3 unused bits would allow at most 8 times as many.
- MIPS32 uses a fixed-length encoding: based on this, no format can be larger *or* smaller than another, meaning it would not be viable for R-type instructions to be smaller (irrespective of how much).

As such, the final statement is correct: there are 2 unused bits in the I-type instruction format, so the *imm* field could indeed by 2 bits larger (so 17 bits afterwards, vs. 15 bits before).

▷ **S202.** Classifying any ISA in absolute terms can be difficult, and open to opinion and debate. However, this particular case is fairly clear. In short, this is a load instruction which uses a fairly complex addressing mode. Rather than including instructions with simple semantics which can be used as a "building'block" to realise more complex functionality, that functionality is combined into this single instruction. For example the same functionality *could* realised via

$$\begin{array}{l} \mathsf{GPR}[t] \leftarrow \mathsf{MEM}[\mathsf{GPR}[y]] \\ \mathsf{GPR}[t] \leftarrow \mathsf{GPR}[t] + \mathsf{GPR}[z] \\ \mathsf{GPR}[x] \leftarrow \mathsf{MEM}[\mathsf{GPR}[t]] \end{array}$$

instead, and hence a simpler indirect addressing mode. However, that same complexity allows a higher code density because fewer instructions (i.e., 1 vs. 3) are required. Finally, note that the instruction performs two memory accesses. Since the value loaded by the first is used as an address in the second, these accesses must be done in sequence; this fact would suggest the instruction has a multi-cycle execution latency. Overall this suggests that the ISA, or this instruction at least, is best classified as CISC.

▷ **S203.** It is of course hard to capture the features of an entire ISA using one instruction. However, the central idea is that the two instruction differ in one crucial respect: although both perform an addition, instruction 2. uses R_1 as source and R_0 as destination whereas instruction 1. uses R_0 as source and destination. As a result, instruction 2. clearly represents a register machine (or 3-operand machine: the operands are R_0, R_1, and 10), whereas instruction 1. could represent a register machine (or 2-operand machine: the operands are R_0 and 10) or an accumulator machine (or 1-operand machine: the operand is 10). The most precise options among those available is therefore such that instruction 2. represents a register machine whereas instruction 1. represent an accumulator machine.

▷ **S204.** The explanation about overflow is nonsense. Beyond that, some other explanations need an argument or evidence which is hard given the information available. For example, one could argue about how often each instruction is used and perhaps use of `addi` is more frequent than `subi`. But claiming there is *never* a need for `subi` seems too strong; either way, we have no evidence for it. Likewise, there *are* unused opcodes in MIPS32: by inspecting the ISA specification, for example, it is clear that some regions of the opcode space marked as reserved and so *could* have been allocated to `subi`. Even without the ISA itself, using the entire opcode space, while not impossible, is at least unlikely.

However, both explanations related to semantic equivalence are more clear-cut. We can see that

$$\mathsf{GPR}[x] \leftarrow \mathsf{GPR}[y] - imm \mapsto \left\{ \begin{array}{l} \mathsf{GPR}[t] \leftarrow \mathsf{GPR}[0] + imm \\ \mathsf{GPR}[x] \leftarrow \mathsf{GPR}[y] - \mathsf{GPR}[t] \end{array} \right.$$

and

$$\mathsf{GPR}[x] \leftarrow \mathsf{GPR}[y] - imm \mapsto \mathsf{GPR}[x] \leftarrow \mathsf{GPR}[y] + (-imm)$$

for example. Given the latter equivalence, which is correct due to the properties of two's-complement representation, we can see that `subi` and `addi` are the same instruction: the difference is that the programmer either uses the immediate as is for `addi`, or negates it for `subi`. Versus the former equivalence, this seems the more plausible explanation because it both avoids increasing the number of instructions (and probably execution latency therefore) and use of any temporary registers (i.e., $\mathsf{GPR}[t]$).

▷ **S205.** The question here is basically "which instructions can update PC", and the answer is *all* the instructions. For some of the options, the reason is obvious:

- `j r3` is an unconditional, computed branch, whose semantics are

  $$\mathsf{PC} \leftarrow \mathsf{GPR}[3].$$

  So if $\mathsf{GPR}[3] = x$, execution of it would satisfies the requirement that $\mathsf{PC} = x$ afterwards.

- `mv pc, r7` is a register-to-register move, whose semantics are

  $$\mathsf{PC} \leftarrow \mathsf{GPR}[7].$$

  So if $\mathsf{GPR}[7] = x$, execution of it would satisfies the requirement that $\mathsf{PC} = x$ afterwards.

The remaining option is therefore `add r15, r15, r2`, whose semantics are

$$\mathsf{GPR}[15] \leftarrow \mathsf{GPR}[15] + \mathsf{GPR}[2].$$

However, ARMv7-A exposes PC as GPR[15]: they are the same register. Therefore, the semantics are actually

$$\mathsf{PC} \leftarrow \mathsf{PC} + \mathsf{GPR}[2].$$

So if $\mathsf{GPR}[2] = -8$, execution of it would also satisfy the requirement that $\mathsf{PC} = x$ afterwards: this is because adding -8 "undoes" the act of incrementing PC (i.e., adding $+8$) during the fetch stage.

▷ **S206.** The question itself is quite verbose. This is intentional, in the sense that the challenge involved is broad and shallow as a result (vs. narrow and deep). Put another way, there is a lot of information to absorb; however, neither it nor the steps involved in producing a solution are very complicated.

First, there are 120 instructions that we know of, so, given $2^7 = 128 > 120 > 64 = 2^6$, we need *at least* a 7-bit opcode to identify them. Next, each operand is comprised of 1) an immediate-based base, plus 2) a register-based offset. Since the are $64 \cdot 1024 = 2^{16}$ addressable bytes in the memory, we need a 16-bit base; since there are $16 = 2^4$ registers we need a 4-bit offset. Overall then, we need *at least*

$$7 + 3 \cdot (16 + 4) = 7 + 3 \cdot 20 = 67$$

bits to encode each 3-address instruction.

▷ **S207.** a A RISC-style ISA is usually comprised of primitive instructions with simple "building-block" semantics, from which complex functionality (i.e., solutions) can be synthesised. This approach aligns with that of a load-store architecture, in which only a small set of dedicated instructions can access operands in memory: all others instructions can access operands held in registers only. A smaller number of less complex addressing modes is typical. As such, this statement can be aligned with the CISC design philosophy where an alternative, register-memory architecture (allowing operands held in a mixture of registers and memory), may be preferred: a larger number of more complex addressing modes is possible.

b In full, Wulf claims it is better for an ISA to "*provide good primitives from which solutions to code generation problems can be synthesized than to provide the solutions themselves*". This statement can be aligned with the RISC design philosophy: a RISC-style ISA is usually comprised of primitive instructions with simple "building-block" semantics, from which complex functionality (i.e., solutions) can be synthesised. In contrast, a CISC-style ISA might provide said functionality directly as an instruction.

▷ **S208.** The 16-bit immediate is left-shifted by 2 bits, i.e., padded with 2 least-significant zero bits. The reason for this is to ensure the offset is 4-byte (or 32-bit) aligned, meaning the new PC is always a multiple or 4 and thus points to an instruction (vs. say part way through an instruction). It is then sign-extended from 18 to 32 bits, so that adding it to PC yields the intended result.

Either way, recall that an n-bit signed integer represented by using two's-complement will be in the range

$$-2^{n-1} \leq imm < +2^{n-1} - 1.$$

So, in this case, where $n = 16$, we know

$$\begin{array}{rcccl} -32768 & \leq & imm & < & +32767 \\ -131072 & \leq & (\, imm \ll 2 \,) & < & +131071 \end{array}$$

where the former describes the range of *instructions*, whereas the latter describes the range of *bytes*: given that $131072 = 128 \cdot 1024$, a reasonable description of the latter range would therefore be ±128KiB.

▷ **S209.** Two stages of the fetch-decode-execute cycle are clearly relevant to the question, namely execute and fetch. First, this instruction is a zero-extended, half-word (or rather 16-bit), post-indexed load: the semantics are

$$\begin{array}{lcl} \mathsf{GPR}[0] & \leftarrow & \mathsf{ext}_0^{32}(\mathsf{MEM}[\mathsf{GPR}[1]]^2) \\ \mathsf{GPR}[1] & \leftarrow & \mathsf{GPR}[1] + \mathsf{GPR}[2] \end{array}$$

So, the ISA suggests 2 bytes will be transferred during execution. There is a caveat, in the sense that the micro-architecture may opt to, e.g., perform a word-sized transfer then extract the half-word from the result; in terms of the ISA, however, implementation detail of this type is not considered. Second, modulo extensions such as Thumb (noting that this post-indexed addressing mode is only available in the 32-bit ARM encoding), ARMv7A uses a fixed-length, 32-bit instruction encoding. As such, fetching the instruction in the first place implies transfer of 4 bytes. So, across the entire fetch-decode-execute cycle, the ISA suggests that $2 + 4 = 6$ bytes will be transferred between the micro-processor and memory.

▷ **S210.** A pointer is just an address, so the fact that

$$\texttt{sizeof(p)} = 8$$

means each addresses is 8 bytes or 64 bits. As such, the program can access addresses ranging from 0 to $2^{64} - 1$; the total number of addressable elements (bytes in this case) is therefore 2^{64}.

▷ **S211.** For clarity, imagine that MEM[101] and MEM[102] contain the variables `a` and `b`, whose initial values are $\alpha \geq 0$ and $\beta \geq 0$ respectively; MEM[100] contains the variable `t`, which represent some uninitialised, intermediate storage.

a Based on this, we can produce a trace of execution

$$\begin{array}{lclcrcl cccc} \mathsf{PC} = 0 & \rightsquigarrow & \texttt{sbn 100, 100, 0} & \mapsto & \texttt{t} \leftarrow & \texttt{t} - \texttt{t} & & & = & 0 \\ & & & & \mathsf{PC} \leftarrow & \mathsf{PC} + 1 & & & = & 1 \\ \\ \mathsf{PC} = 1 & \rightsquigarrow & \texttt{sbn 100, 101, 2} & \mapsto & \texttt{t} \leftarrow & \texttt{t} - \texttt{a} & = & 0 - \alpha & = & -\alpha \\ & & & & \mathsf{PC} \leftarrow & 2 & & & = & 2 \\ \\ \mathsf{PC} = 2 & \rightsquigarrow & \texttt{sbn 102, 100, 3} & \mapsto & \texttt{b} \leftarrow & \texttt{b} - \texttt{t} & = & \beta - -\alpha & = & \beta + \alpha \\ & & & & \mathsf{PC} \leftarrow & \mathsf{PC} + 1 & & & = & 3 \\ \\ \mathsf{PC} = 3 & \vdots \end{array}$$

As such, one can see that the program adds the content of `a` (or MEM[101]) to `b` (or MEM[102]) using `t` (or MEM[100]) for intermediate storage; the result is stored in `b` (or MEM[102]).

b Based on this, we can produce a trace of execution

$$\begin{array}{lclcrcl cccc} \mathsf{PC} = 0 & \rightsquigarrow & \texttt{sbn 100, 100, 0} & \mapsto & \texttt{t} \leftarrow & \texttt{t} - \texttt{t} & & & = & 0 \\ & & & & \mathsf{PC} \leftarrow & \mathsf{PC} + 1 & & & = & 1 \\ \\ \mathsf{PC} = 1 & \rightsquigarrow & \texttt{sbn 100, 101, 2} & \mapsto & \texttt{t} \leftarrow & \texttt{t} - \texttt{a} & = & 0 - \alpha & = & -\alpha \\ & & & & \mathsf{PC} \leftarrow & 2 & & & = & 2 \\ \\ \mathsf{PC} = 2 & \rightsquigarrow & \texttt{sbn 102, 102, 0} & \mapsto & \texttt{b} \leftarrow & \texttt{b} - \texttt{b} & = & \beta - \beta & = & 0 \\ & & & & \mathsf{PC} \leftarrow & \mathsf{PC} + 1 & & & = & 3 \\ \\ \mathsf{PC} = 3 & \rightsquigarrow & \texttt{sbn 102, 100, 4} & \mapsto & \texttt{b} \leftarrow & \texttt{b} - \texttt{a} & = & 0 - -\alpha & = & \alpha \\ & & & & \mathsf{PC} \leftarrow & \mathsf{PC} + 1 & & & = & 4 \\ \\ \mathsf{PC} = 4 & \vdots \end{array}$$

As such, one can see that the program copies the content of `a` (or MEM[101]) into `b` (or MEM[102]) using `t` (or MEM[100]) for intermediate storage.

▷ **S212.** There are basically 3 decision points here: 1) whether the instruction is a load or a store, 2) what the access granularity is (e.g., 8-, 16-, or 32-bit, meaning byte, half-word, or word), and 3) what the address mode is (i.e., how the effective address is formed).

a Re. 1), it is clear that MEM is the same before and after execution of the instruction whereas GPR has changed: this instruction is therefore more likely to be a load than a store. Re. 2), use of ARM-style assembly language implies the target register of the load is GPR[0]. However, the value loaded, i.e., 39 does not appear in MEM: this suggests an 8-bit access granularity was not used. The next option would be 16-bit, and, in this case, we find that

$$(\mathsf{MEM}[5] \ll 8) + (\mathsf{MEM}[4] \ll 0) = (2 \ll 8) + (7 \ll 0) = 32 + 7 = 39.$$

As such, a 16-bit load using the effective address 4 could result in the value 39 we find in GPR[0]. Re. 3), the question is basically how the effective address 4 was computed using GPR[0], GPR[1], and GPR[2]. Assuming no implicit access to *other* registers, the only way to do that is GPR[1] + GPR[2] = 3 + 1 = 4. But GPR[2] is *also* updated, from 1 to 3. Using both facts, we can conclude that the instruction is using an auto-indexed addressing mode: as well as the load into GPR[0], GPR[2] is incremented by 2 (matching the access granularity). As such, this is an auto-indexed, 16-bit load: we could describe the semantics more formally as

$$\mathsf{GPR}[0] \leftarrow \mathsf{MEM}[\mathsf{GPR}[1] + \mathsf{GPR}[2]]^2; \mathsf{GPR}[2] \leftarrow \mathsf{GPR}[2] + 2.$$

b Re. 1), it is clear that MEM is the same before and after execution of the instruction whereas GPR has changed: this instruction is therefore more likely to be a load than a store. Re. 2), use of ARM-style assembly language implies the target register of the load is GPR[0]. The only occurance of 7 in MEM is at address 4: the load could have used an 8-bit access granularity, therefore. No other options are possible. For example, a 16-bit load from address 4 implies MEM[5] would also be accessed; doing so would results in

$$(\mathsf{MEM}[5] \ll 8) + (\mathsf{MEM}[4] \ll 0) = (2 \ll 8) + (7 \ll 0) = 32 + 7 = 39.$$

Re. 3), the question is basically how the effective address 4 was computed using GPR[0], GPR[1], and GPR[2]. Assuming no implicit access to *other* registers, the only way to do that is GPR[1] + GPR[2] = 4 + 0 = 4. Note that *only* GPR[0] changes, which basically rules out use of an auto-indexed addressing mode (which would typically aim to update either GPR[1] and/or GPR[2] somehow). As such, this is an indexed, 8-bit load: we could describe the semantics more formally as

$$\mathsf{GPR}[0] \leftarrow \mathsf{MEM}[\mathsf{GPR}[1] + \mathsf{GPR}[2]]^1.$$

▷ **S213.** Given a predicate p and an instruction I, execution via predicated execution is such that

$$\textbf{if } p = \textbf{true then } I \ \textbf{ else } \mathsf{nop}.$$

As such, clearly the semantics of nop remain unchanged: we end up with

$$\textbf{if } p = \textbf{true then } \mathsf{nop} \textbf{ else } \mathsf{nop}$$

and so execute nop whether or not $p = \textbf{true}$.

▷ **S214.** a **false**. A load-store architecture might be termed register-register, because it divides instructions into two classes: these are storage-related, i.e., load a value from memory into the general-purpose register file or store a value into memory from the general-purpose register file, and compute-related, e.g., arithmetic on values in the general-purpose register file. So, in a register-register architecture we can say that 1) only storage-related instruction access memory, and 2) storage-related instructions are divided strictly cases which load from *or* store into memory. If an instruction were to both load from *and* store into memory, it would have semantics similar to $\mathsf{MEM}[x] \leftarrow f(\mathsf{MEM}[y])$ and so the underlying architecture would have to be memory-memory or register-memory. Put another way, this differs from a register-register architecture which would not support instructions with such semantics.

b **false**. To show why, it suffices to offer a counter-example: glossing over the impact on flags (since the question refers to the state of `r0` alone), instructions such as `xor r0, r0, r0` and `sub r0, r0, r0` suffice.

c **false**. It is important to distinguish between 1) has a general purpose, and 2) is addressable as a general-purpose register. The program counter *does* have a special purpose, i.e., to keep track of the next instruction to execute, and so is not an instance of 1). However, this purpose may or may not be realised via 2). Put another way, in some ISAs one can *explicitly* read from or write into PC by using it in a general-purpose instruction in the same way as some GPR[i]: an example would be ARMv7-A, where

$$\texttt{mov rd, r15} \mapsto \mathsf{GPR}[rd] \leftarrow \mathsf{PC}$$

reads PC, because GPR[15] acts as an alias for it. In other ISAs this is not possible. Instead one must *implicitly* read from or write into PC by using a special-purpose instruction: an example would be MIPS, where

$$\texttt{jalr rs} \mapsto \mathsf{GPR}[31] \leftarrow \mathsf{PC} + 4; \mathsf{PC} \leftarrow \mathsf{GPR}[rs].$$

reads and writes PC, even though it is not referenced as an operand.

d **true**. One pitfall here is interpreting micro-architecture as (physical) hardware, i.e., a design which is manufactured. Then, altering a specific instance of that hardware may be quite tough of course: hardware is typically immutable, modulo use of techniques such as micro-code. So it might *seem* that altering the ISA, which is just a specification of course, would be "easier".

But the question specifically refers to the micro-architectural design. By definition, an ISA is exposed to software (or the developer of it) whereas a micro-architecture design is abstracted from software (or the developer of it), i.e., the implementation detail is "hidden" behind the interface-like ISA. As such, altering an ISA potentially implies alteration of software: to be executed as intended the software must comply with the ISA, so, if the ISA is altered, the software might need to be altered to match. The implication is wider, however, because other components, e.g., compilers, operating systems, etc., might also need to be altered for the same reason. In contrast, altering a micro-architecture design implies no alteration of software: provided both it and the software still comply with the same ISA, the software will be executed correctly.

Another way of looking at this is that altering an ISA implies that one must alter a micro-architecture which implements it (so it remains compliant): the former must be at least as hard as the latter therefore.

e **true**. The reason for this is fairly simple. We know the program is stored in memory; a von Neumann architecture implies use of a unified memory, storing both data and instructions. Execution of each instruction demands the micro-processor completes a fetch-decode-execute cycle: this means it will 1) fetch the instruction, i.e, load from and increment the Program Counter (PC) via $\mathsf{IR} \leftarrow \mathsf{MEM}[\mathsf{PC}]$ and $\mathsf{PC} \leftarrow \mathsf{PC} + 1$, 2) decode the instruction, i.e., decide what IR means, then 3) execute the instruction, i.e., do what IR means. Therefore, execution of each instruction requires at least one access to memory (to load said instruction, during the fetch stage), and, as a result, the latency of each fetch-decode-execute cycle is limited by the latency of that access. Put another way, the the number of instructions it executes per unit of time is limited by how quickly those instructions can be fetched (and hence the access latency of memory).

f **false**. One could view doubling the word size as increasing the "amount of work done" by execution of instructions that operate on word-sized values: those values are larger, so, in a sense, more computation is being done.

In that sense, the execution latency of *some* programs might be reduced. An example is computation using multi-precision integers: given a word size w these are n-bit integers for $n > w$, and so too large to represent using one word, represented instead by using a sequence of $m = n/w$ words. Addition of two such integers will take $O(m^2)$ steps: larger w allows a smaller m, and so fewer steps therefore. But this is not true of *all* programs. An example might be a program which sorts an array of w-bit integers, e.g., by using the quicksort algorithm. For an m-element array, the number of steps will be $O(m \log m)$: this means m (the number of values), not w (their size) is the central factor here.

▷ **S215.** Since a variable-length scheme allows instructions that can grow and shrink to suit the number of operands, it is easy to conclude the first statement is true. It is almost certainly true that a fixed-length scheme will imply less complex instruction decoding; a simple example is the fact there will typically only ever be one access to memory per-fetch (versus multiple accesses depending on the variable length). Therefore the second statement is also true. With a fixed-length scheme, PC must always point at an address which is some multiple of the instruction length; a variable-length scheme typically does not have such a restriction. This is only one example, but acts as enough motivation to conclude the fourth statement is probably true in general. A fixed-length scheme only has a fixed amount of bits to allocate for an immediate operand, hence it cannot be true it would allow larger operands in general: the third statement is the incorrect one.

▷ **S216.** a An immediate addressing mode allows fixed constants to be encoded within the instruction itself. One might use such an addressing mode within a C statements such as

```
a=b+10;
```

where the instruction would include the register addresses for a and b, and the immediate 10.

b A direct addressing mode is where we pass an address to the memory and get back the value at exactly that address, i.e., we provide the effective address x and get back $\mathsf{MEM}[x]$. This might be used to load a variable from memory. Considering the statement

```
a = b + 10;
```

again, if a were held in memory rather than a register then it might be loaded using a direct addressing mode *if* we know the exact address, e.g., if a were global and hence existed within the static data section.

c An indirect addressing mode employs a two-stage process; we pass the effective address x and load $\mathsf{MEM}[\mathsf{MEM}[x]]$ so the access is like a pointer dereference. That is, if we want to dereference a pointer p which is stored at a particular address x in memory, we first load p from $\mathsf{MEM}[x]$ and then dereference the pointer loading from where it points to; in C this is essentially the expression *p.

d Finally, indexed addressing means that the value we get back is located at an address formed from the sum of a base and an offset. For example we might load from `A[ 10 ]` by accessing MEM[`&A` + 10], i.e., using `&A` as the base and 10 as the offset.

▷ **S217.** a The basic idea behind predicated execution is that each instruction as a simple predicate function associated with it. When the instruction is fetched and is ready to be executed, the predicate function is evaluated: if it evaluates to true, the instruction is executed, otherwise it is discarded.

Assume the symbols `t`, `x`, `i` and `n` are allocated the registers GPR[1], GPR[2], GPR[3] and GPR[4]. The ARM scheme for predicated execution represents a somewhat standard approach: execution of each arithmetic instruction updates a status register which contains flags that detail whether the result generated a carry (C), generated an overflow (O), was negative (N) or was zero (Z). Each instruction can include a predicate which allows execution or non-execution based on these flags. As such, we could translate the fragment into the assembly language style operations:

```
        ...
        GPR[1] ← 0
        GPR[3] ← 0
loop :  if GPR[3] ≥ GPR[4], PC ← exit
        GPR[5] ← GPR[2] ≪ GPR[3]
        GPR[5] ← GPR[5] ∧ 1
        if Z = false, GPR[1] ← GPR[1] + 1
        GPR[3] ← GPR[3] + 1
        PC ← loop
exit :  ...
```

Two conventional branches have been used to implement the loop: the reasoning is that they transfer control-flow over a (fairly) large number of dissimilar instructions which might themselves update the status register. Furthermore, the branches can be fairly accurately predicted using even static techniques so one would not expect too much of a problem in terms of their preventing effective speculative execution. On the other hand, the branch over the single instruction update of `t` is essentially random (for arbitrary `x`) so branch prediction is difficult and expensive in relation to the instruction which the change of control-flow is skipping over. As a result, predicated execution is used to skip the instruction based on the Z flag which would have been set via calculation of the conditional expression.

b Roughly speaking, code density refers to the number of instructions (or amount of work done) per-unit of memory. That is, if we have 100 instructions that fit into 100B of memory the code density is the ratio of the two, i.e., $\frac{100}{100} = 1$. If we fit *more* instructions into our 100B, say 150, then the code density increases to $\frac{150}{100} = 1.5$. Good code density is viewed as an advantage because it typically reduces the code footprint for programs, i.e., the amount of memory they consume, or means that instructions are smaller, and hence easier or faster to retrieve.

There are plenty of ways to improve code density. One very involved example is to redesign the ISA so that each instruction can accomplish more; this is roughly the approach of CISC processors. In this way, since each instruction performs more useful work, there need be less instructions in total. Another way is to write better programs, or use a better compiler, that is able to use less instructions to achieve the same high-level goal. Clearly there is a limit to how successful this can be, but clever programming can often drive the code footprint underneath some fixed barrier that prevents execution; perhaps the processor has 100kB of memory for example and the program needs 101kB before one optimises it.

The concept of variable length instruction potentially increases code density at the cost of increased complexity within the instruction decoding stage. By allowing each instruction to be as long or short as required, instructions that might normally consume a lot of space because they are forced to be the same length as others, can be smaller and hence save space.

▷ **S218.** First we look at the types of instruction required:

a 0 registers, 1 immediate of 7 bits to store 0 . . . 127 (e.g., LDA).
b 1 registers, 1 immediate of 4 bits to store −8 . . . 7 (e.g., BEZ).
c 2 registers, 0 immediate (e.g., MOV).
d 1 registers, 0 immediate (e.g., NOT, NEG).
e 0 registers, 0 immediate (e.g., NOP).

Given register indicies are representing as 2-bit operands, a possible encoding has 5 instruction types:

a `0xxxxxxx`

b `10rrxxxx`

c `11ccrrrr` ↦ `1100rrrr` . . . `1110rrrr`

d `1111ccrr` ↦ `111100rr` . . . `111110rr`

e `111111cc` ↦ `11111100` . . . `11111111`

where `c` represents space for opcodes, `x` represents space for an immediate and `r` represents space for register indicies. From this we can see that there are

- 3 possible instructions with 2 register operands (i.e., the encodings `1100rrrr`, `1101rrrr` and `1110rrrr`),
- 4 possible instructions with 1 register operand (i.e., the encodings `111100rr`, `111101rr` and `111110rr`, and `10rrxxxx`), and
- 5 possible instructions with 0 register operands (i.e., the encodings `11111100`, `11111101` and `11111110`, `11111111`, and `0xxxxxxx`).

From this we can start encoding the actual instructions already given:

```
LDA x      ↦   0xxxxxxx
MOV R S    ↦   1100RRSS
NOT R      ↦   111100RR
NOP        ↦   11111100
NEG R      ↦   111101RR
BEZ R x    ↦   10RRxxxx
```

As such, we can add

- 2 extra instructions with 2 register operands (since we only use 1 encoding for MOV),
- 1 extra instructions with 1 register operands (since we only use 3 encodings for NOT, NEG and BEZ), and
- 3 extra instructions with 0 register operands (since we only use 2 encodings for LDA and NOP).

The additional instructions should ideally come from the obvious classes: so far we have no load or store instructions, no add instruction which is quite fundamental, and no comparisons.

▷ **S219.** This is quite a vague question and demands that you make (stated) assumptions about anything that is unclear.

A basic approach is as follows; there are 64 squares on the board so one can encode a board position as a 6-bit integer. To encode a move, we simply use two such board positions, a source s and a target t with the meaning that one moves the piece at board position s to board position t. This is a 12-bit representation which is not too compact but is quite easy to decode.

To be more compact, we need to be more clever. Firstly, note that the robot keeps track of whose turn it is so to identify a piece, in some sense we don't need to encode its colour: this is implicit in whose move it is. Secondly note that each piece cannot move to an arbitrary board position: the movement rules constrain how many places it can end up. For example, a king can only move to the squares directly around it: 8 possible moves in total not 64. Drawing a table for the other pieces we find that

Piece	Moves	$\log_2$(Moves)
King	8	3
Queen	27	5
Bishop	13	4
Knight	8	3
Rook	14	4
Pawn	4	2

That is, for example, a queen can move in 27 possible ways and we can encode these ways as a 5-bit integer. So, gathering the different types together we find that there are First we look at the types of instruction required and find there are four:

a 1 instruction (move queen) with 5-bit operands.

b 4 instructions (move bishop or rook) with 4-bit operands.

c 3 instructions (move king and knight) with 3-bit operands.

d 8 instructions (move pawn) with 2-bit operands.

With four types, we can produce an encoding using 8-bit integers:

a `00cxxxxx` ↦ `000xxxxx` . . . `001xxxxx`

b `01ccxxxx` ↦ `0100xxxx` . . . `0111xxxx` c `10cccxxx` ↦ `10000xxx` . . . `01111xxx`

d `11ccccxx` ↦ `110000xx` . . . `011111xx`

where `c` represents space for opcodes, `x` represents space for an immediate. This is slightly harder to decode but is much more compact. One can probably form a 7-bit encoding by simply enumerating the possible pieces against their possible moves and associating an integer with each combination. However, this could be harder to decode (although this is maybe debatable it one simply has a big look-up table).

▷ **S220.** a A fixed length encoding is where each instruction is of the same length; this is an advantage in that the decode and fetch mechanisms within the processor can be simple and each instruction typically only requires one access to memory. A variable length encoding allows the instructions to be longer or shorter depending on how many operands and so on they require. This makes the processor more complex but can potentially improve the code density of programs.

Following the MIPS32 instruction set architecture, we can define three fixed length instructions formats (which roughly correspond to the MIPS32 I, R and J formats) as follows:

```
    6      5      5                   16
+------+------+------+--------------------+
|opcode| rs   | rt   |                imm | Format #1
+------+------+------+--------------------+
    6      5      5      5       5      6
+------+------+------+------+------+------+
|opcode| rs   | rt   | rd   |shamt |funct | Format #2
+------+------+------+------+------+------+
    6                                 26
+------+----------------------------------+
|opcode|                              imm | Format #3
+------+----------------------------------+
```

Format #1 is for immediate based memory access instructions and conditional branches, #2 is used for ALU and register based memory instructions, and #3 is for unconditional jumps. Having three formats is preferable to more in that fewer formats means less complex instruction decoding, as does fixed width 32-bit formats and organising the instruction fields so they are aligned; for example the opcode is always the most-significant 6 bits and the next 5 bits is always a source register operand.

Other advantages include the facility for large immediate values in both branch and ALU instructions; disadvantages include a waste of space in some cases which could be put to better use if more information about the processor requirements was given (for example maybe the space could be used to implement predicate flags for predicated execution).

b To encode the "add immediate" we would use format #3 from our design; assume that the opcode for this instruction is 3 and that immediate values are held in two's-complement form. Thus, the encoded instructions would look like

$$\begin{array}{lcl} \mathsf{GPR}[4] \leftarrow \mathsf{GPR}[8] + 10 & \mapsto & 000011\ 01000\ 00100\ 0000000000001010_{(16)} \\ \mathsf{GPR}[4] \leftarrow \mathsf{GPR}[8] - 10 & \mapsto & 000011\ 01000\ 00100\ 1111111111110110_{(16)} \end{array}$$

Notice the subtle sign case of -10 in the sense that we need to encode this as a 15-bit signed value.

c This function computes the population count or Hamming weight of x, denoted by $\mathsf{HW}(x)$ say, i.e., the number of bits in the binary expansion of x that equal one.

One simple way to help the customer achieve good performance in their application is to support this function in hardware as an ALU operation rather than have them implement it as a C function. We need to expose this operation to the programmer via the instruction set so that, for example, an instruction of the form `pop $1,$2` would take the value in general-purpose register #2, perform the population count operation and write the result into general-purpose register #1. The customer could then replace every call to the C function with a single, fast machine instruction.

Of course, an appropriate circuit is required within the ALU to perform the population count operation. A clocked circuit to do this would roughly mirror the C implementation, but this would take 32 clock cycles to complete which is still quite slow. A better approach would be to build a combinatorial circuit; the goal with a such a circuit would be to minimise the critical path so that the operation can complete in one cycle. One approach would be to construct a binary adder tree which starts (at the leaves of the tree) by adding two bits of the input together: this gives the number of bits set in that region. Roughly, the circuit computes:

$$((x_0 + x_1) + (x_2 + x_3)) \cdots ((x_{28} + x_{29}) + (x_{30} + x_{31}))$$

so that the end result is the number of bits set to 1. This circuit would have a depth of $\log_2(32) = 5$ adders where each need only add together 5-bit numbers.

d First note that the memory is byte addressed, thus when we store the 32-bit constant at address 0 we are actually storing the constituent bytes at addresses $0 \ldots 3$. Also note that the decimal value $305419896_{(10)}$ equals $12345678_{(16)}$.

A little-endian byte ordering convention places the least-significant byte at the lowest address and the most-significant byte at the highest address; thus one would expect to find the memory to look like

$$\begin{array}{lcl} \mathsf{MEM}[0] & = & 78_{(16)} \\ \mathsf{MEM}[1] & = & 56_{(16)} \\ \mathsf{MEM}[2] & = & 34_{(16)} \\ \mathsf{MEM}[3] & = & 12_{(16)} \end{array}$$

after the instruction completes. A big-endian byte ordering convention uses the opposite ordering; it places the least-significant byte at the highest address and the most-significant byte at the lowest address to give

$$\begin{array}{lcl} \mathsf{MEM}[0] & = & 12_{(16)} \\ \mathsf{MEM}[1] & = & 34_{(16)} \\ \mathsf{MEM}[2] & = & 56_{(16)} \\ \mathsf{MEM}[3] & = & 78_{(16)} \end{array}$$

e If the number of registers is increased from 32 to 64 then one more bit per-register operand will be required to address them; so each register operand will require 6 bits in an encoded instruction rather than 5. Assuming that the actual instructions are not changed, this will result in less opcode space in the instruction formats and hence the instruction set will be able to support less instructions. If the number of registers is decreased from 32 to 16 then one less bit per-register operand will be required to address them and the converse is true: there will be more space in the instruction formats for the opcode. A larger (or richer) set of instructions (that support more addressing modes for example) means a given program can probably be shorter. Potentially this means the program can execute faster, if the micro-architecture can execute all instructions as fast as before, and that the code density is better. The converse is true of a smaller (or more sparse) set of instructions.

An additional implication for having less registers is that the programmer (or compiler) will be able to keep less data items in the fast registers and will need to keep more in the slower main memory; this is termed spilling in the context of compilers. So with less registers there will probably be more memory accesses which result in worse performance (and visa versa).

▷ **S221.** a This is an open-ended question, and many characteristics (as long as they are well motivated and explained) are viable answers. A limited list of examples is as follows:

- RISC processors typically have an ISA that includes a small set of simple instructions; each instruction can be executed efficiently (often using pipelining).
- RISC processors typically employ a fixed-length instruction encoding scheme that includes a small number of formats.
- RISC processors typically have a low Cycles Per Instruction (CPI) rating, but also low code density as a result of their ISA design.
- RISC processors typically have large general-purpose register files.
- RISC processors typically utilise a small number of simple addressing modes within the ISA, and separate register-based arithmetic and logic instructions from memory access (i.e., load and store).

b i Various advantages and disadvantages can be stated; in a sense the two options represent a compromise between efficiency and invasiveness.

For example, explicit access means alteration to the ISA is limited; instructions to transfer the status register to and from the general-purpose register file are enough since they permit *any* subsequent computation involving and update to the content. This means less pollution of the instruction encoding, and hence more space for other instructions. In contrast the use of implicit access can imply lower overhead, represented for example by instructions required to load then extract a specific flag from the status register.

ii Examples of other flags included in a typical status register include:

- overflow flag (e.g., to signal when arithmetic overflow occurs, during unsigned integer addition),
- zero, positive and negative flags (to signal when some arithmetic operation produces a zero positive or negative result, which can be useful for producing comparisons),
- error flags (e.g., signalling conditions such as division by zero).

These are often complemented by additional fields including processor mode (i.e., user or kernel) and configuration (e.g., interrupt masks).

c i Instead of signalling a carry as an implicit side effect of an instruction, an integer addition for example, add a second instruction to explicitly generate the carry-out.
For example, imagine an addition instruction of the form

$$\mathsf{GPR}[x] \leftarrow \mathsf{GPR}[y] + GPR[z]$$

implicitly signals a carry by using the status register. If the status register (and hence update) is removed, an extra instruction of the form

$$\mathsf{GPR}[x] \leftarrow (\mathsf{GPR}[y] + \mathsf{GPR}[z]) \ / \ 2^{32}$$

could still generate the associated carry, and store it into a general-purpose register. Along similar lines, one could permit 4-address instructions (two source operands, and *two* targets rather than one) so as to produce the sum *and* carry.

ii This is a tough question, but the idea is as follows:

$$\begin{array}{lcccc} \mathsf{GPR}[6] & \leftarrow & \mathsf{GPR}[1] & + & \mathsf{GPR}[2] \\ \mathsf{GPR}[7] & \leftarrow & \mathsf{GPR}[6] & < & \mathsf{GPR}[1] \\ \mathsf{GPR}[4] & \leftarrow & \mathsf{GPR}[6] & + & \mathsf{GPR}[3] \\ \mathsf{GPR}[8] & \leftarrow & \mathsf{GPR}[4] & < & \mathsf{GPR}[6] \\ \mathsf{GPR}[5] & \leftarrow & \mathsf{GPR}[7] & \vee & \mathsf{GPR}[8] \end{array}$$

The basic idea is that instructions #1 and #3 perform the actual additions of $x + y$ and then ci; each addition is followed by a comparison, in instructions #2 and #4. Each comparison checks whether the associated addition caused an overflow: if the output is smaller than one of the inputs, clearly overflow has occurred since this implies the output has "wrapped around" from a large value to a smaller value. If overflow occurs in either of the additions, this implies a carry-out overall: instruction #5 stores this, OR'ing the two partial carries together.

▷ **S222.** a i General-purpose registers are accessible for general-purpose tasks: essentially they are under control of the programmer and used to perform computation and form addresses. Examples in this case are A and B. In contrast, special-purpose registers are used for specific tasks typically under control of the processor. Examples in this case are the program counter PC, which is used to specify the next instruction that will be executed and is automatically incremented during the decode phase.

ii An absolute branch means the branch target is given directly: an absolute branch to x means updating PC to x. In contrast, a relative branch means the branch target is formed by adding an offset to the current value of PC; the idea is that a relative branch to x means updating PC to $PC + x$.

b i The first case uses B, an 8-bit register, as the effective address; as a result, the largest address is the maximum value B can assume, i.e.,

$$2^8 - 1 = 255.$$

The second cases uses i as the effective address; this needs a bit more thought. Clearly $i < 2^8$ since it needs to be encoded within the load instruction. The problem is, some space is also required for an opcode (i.e., to identify this instruction type). The theoretical minimum number of bits needed for such an opcode would be one, though of course this depends on a variety of other issues; probably it would have to be two or more in reality. Following this line of reasoning however, the theoretical maximum value i can assume (which is also the largest address) is

$$2^7 - 1 = 127.$$

ii Clearly there are a wide variety of approaches: the key thing is good reasoning about both how the instructions solve the problem, and what the impact is on the processor design (e.g., potential disadvantages). Here are two examples:

i. The two registers could be combined into one effective address; for example, an instruction with the semantics

$$A \leftarrow \mathsf{MEM}[A + (B \ll 8)]$$

could be included. This essentially allows A and B to be combined into a single 16-bit address and works a little like segmented memory model. The largest effective address they can combine to specify is

$$(2^8 - 1) + (2^8 - 1) \ll 8 = (2^8 - 1) + (2^8 - 1) \cdot 2^8 = 255 + 65280 = 65535.$$

The advantage is that no extra space is required in the instruction encoding: A and B can both be implicit in the instruction type. However, given that the processor only has two registers, one could argue this approach has to resolve the problem of register pressure. One idea would be to add an extra, dedicated segment register rather than use B.

ii. The requirement for fixed-length instructions could be replaced to allow variable-length alternatives; this could allow the use of larger immediate operands. The theoretical maximum in our original example was a 7-bit immediate; if we encode the load instruction as two 8-bit instruction parts rather than one, this increases the maximum to 15 bits. The largest effective address from such an immediate is then

$$2^7 + 2^8 = 2^{15} = 32768.$$

As an aside, it is tempting to answer with some form of scaled access only, i.e., something like

$$A \leftarrow \mathsf{MEM}[B \ll i].$$

This clearly grants the ability to access larger addresses; the disadvantage however is that there are some "gaps" in the range of accessible addresses. For example, say $i = 8$: how do we access address $(255 \cdot 2^8) + 1 = 65281$?

c The processor cannot accommodate 16-bit data-types in the sense that it only has 8-bit registers: one cannot for example load `i` into A or B.

We have already described a load instruction where a 16-bit effective address to be formed from A and B. Therefore, `S[ i ]` (i.e., loading the i-th character of the string `S`) is possible if we store the least-significant 8-bits of `i` in A and the most-significant 8-bits in B. However, we also need to increment `i`; to achieve this, we need a carry-flag (say C) and also some instructions that use it. Specifically, we need an add instruction

$$C, A \leftarrow A + i$$

that computes the sum into A and the carry-out into C, and an add-with-carry instruction

$$B \leftarrow B + C$$

that adds the carry flag to B. Armed with these additions, and given we have `i` stored as described above, the increment can be performed as follows:

$$\begin{array}{lcl} C, A & \leftarrow & A + 1 \\ B & \leftarrow & B + C \end{array}$$

First we add 1 to A and, if there was a carry-out from the addition, we add this to B since in such a case $C = 1$.

▷ **S223.** a i A logical right-shift of some n-bit x by d bits discards the least-significant d bits, and fills the most-significant d bits with a zero.

ii An arithmetic right-shift of some n-bit x by d bits discards the least-significant d bits, and fills the most-significant d bits with the sign of x, i.e., x_{n-1}.

As a result, one can view them as unsigned and signed shifts respectively; as a result, the operation `A[ i ] >> j` requires a logical right-shift operation since `A[ i ]` is an unsigned integer.

b Basically,

$$CPI = \sum_{t \in T} \mathcal{F}_t \cdot \mathcal{L}_t$$

where $\mathcal{F}_t$ and $\mathcal{L}_t$ denote the frequency of execution and latency of instruction type t respectively, and here we have that $T = \{\text{arithmetic}, \text{branch}, \text{memory access}\}$. Hence,

$$\begin{array}{rcl} CPI & = & (0.5 \cdot 1) + (0.1 \cdot 4) + (0.4 \cdot 2) \\ & = & 1.7 \end{array}$$

c There are a variety of valid answers here, for example:

i immediate, for `t = t + 1`,

ii indexed, for `A[ i ]`,

iii scaled, for `A[ i ]` given each element is 32-bit, or

iv auto-indexed, combining `A[ i ]` and `i++`.

d There are a variety of valid answers here, for example:

In software:

- Simply remove the branch, and alter the next instruction to read

$$\mathsf{GPR}[2] \leftarrow \mathsf{GPR}[2] + \mathsf{GPR}[5].$$

This works because GPR[5] is either 1 when we want to add 1 to GPR[2], or 0 otherwise: so we can simply add it rather than checking it and conditionally adding 1.

In hardware:

- Offer a facility for predicated execution, then remove the branch and alter the next instruction to read something like
$$\text{if } Z = \textbf{false}, \mathsf{GPR}[2] \leftarrow \mathsf{GPR}[2] + 1.$$
That is, if the zero flag Z is **false** (meaning the result of the previous operation was not zero) perform the addition, otherwise skip the addition.

e There are a variety of valid answers here, for example:

In software:

- Optimise the program by moving the load from `A[ i ]` out of the inner-loop (since it is fixed for all `j`).
- Optimise the program by using some form of pre-computation so that one can look-up the Hamming weight rather than compute it.
- Depending on the probability of a given `A[ i ]` being zero, early abort from such iterations via a single instruction test (i.e., `A[ i ] == 0`) rather than checking each bit when we know the outcome.

In hardware:

- Offer a single instruction that extracts the `j`-th bit of some `x`, i.e., computes `( x >> j ) & 1`.
- Offer a means of SIMD-style parallelism, e.g., via vector unit, whereby multiply elements in the array can be processed simultaneously.
- Employ a cache or more advanced memory system so that the 4-cycle latency of (each) memory access is reduced.

Part XII: Processor design: basic micro-architecture

▷ **S224.** If there are more read ports, a register file will need more logic to implement it and hence the implementation cost will be higher: each read port is roughly equivalent to a multiplexer which, based on the register address acting as a control signal, decides which of n registers to feed through to a single output. If there are more read ports, one can perform more reads in parallel which therefore typically means less sequential reads. Finally, if there are n registers then one needs $\lceil \log_2 n \rceil$ bits to address them using a basic scheme; some $m > n$ will potentially need more bits and hence larger register addresses. The incorrect statement is therefore the third: more registers typically implies *less* register pressure because with more registers, it is more likely we have a free register and therefore less likely we have to spill one into memory.

▷ **S225.** a We have a total of 16 registers to address; each address therefore requires 4 bits. There are 16 instructions in total; the opcode therefore requires 4 bits. To accommodate this, we can define three formats:

```
    4      4      4      4
+------+------+------+------+
|opcode| src  | dst  | imm  |  Format #1
+------+------+------+------+
    4      4      4      4
+------+------+------+------+
|opcode| src  | dst  | src  |  Format #2
+------+------+------+------+
    4      4         8
+------+------+-------------+
|opcode| src  |        imm  |  Format #3
+------+------+-------------+
```

Format #1 is for immediate based memory access instructions, #2 is used for ALU and register based instructions, and #3 is for the branch instructions. Having three formats is preferable to more in that fewer formats means less complex instruction decoding, as does fixed width 16-bit formats and organising the instruction fields so they are aligned; for example the opcode is always the most-significant 4 bits and the next 4 bits is always a source register operand.

b A reasonable diagram might resemble:

```
+---------+             +---------+
|         |       +---->|         |
|         |       |     |   ALU   |-----+
|         |       |  +->|         |     |
|         |       |  |  +---------+     |
|         | A Bus |  | B Bus            | C Bus
|         |       |  |  +---------+     |
| CONTROL |-----> |  +--|         |     |
|  UNIT   |       |     |   GPR   |<----+
|         |       +-----|         |     |
|         |       |     +---------+     |
|         |       |                     |
|         |       |     +---------+     |
|         |       +-----|   PC    |<----+
+---------+       |     +---------+     |
                  +-----|   IR    |<----+
+---------+       |     +---------+     |
|         |<----> +-----|   MBR   |<----+
| MEMORY  |       |     +---------+     |
|         |<----- +-----|   MAR   |<----+
+---------+             +---------+
```

where

- *ALU* is the Arithmetic and Logic Unit.
- *GPR* is the General-Purpose Register file.
- *PC* is the Program Counter.
- *IR* is the Instruction Register.
- *MBR* is the Memory Buffer Register.
- *MAR* is the Memory Address Register.

A register file holds general-purpose, addressable registers which store data operated on by instructions. The program counter stores the address of the next instruction to be executed; the instruction register holds the instruction currently being executed. The memory data and address registers operate the interface to memory: the processor sends and receives data in the memory data register; it offers the address for loads and stores in the memory address register. The control unit operates the data-path signals to actually make the processor run. The arithmetic and logic unit performs computation on source values to produce results so that meaningful processing can be carried out. The components are connected with several buses, most notably between the ALU and the registers; between memory and the processor. Smaller control signals connect the control unit with all the components so it can operate them.

c The fetch stage loads the instruction from memory. The processor first copies the program counter to the memory address and then starts the access cycle to retrieve the instruction. The program counter is then incremented to point at the next instruction. The decode stage decides what the fetched instruction means in terms of what the processor must do to execute it. Typically this will include extracting and sign-extending any immediate values and fetching source values from the register file. The execute stage performs the actual computation of the instruction, in this case the processor sends the source values through the ALU which adds them together and stores the result into a latch register before writing them into the register file.

d Hard-wiring the control logic means to explicitly encode all the conditions the processor must make when controlling the data-path signals into dedicated logic. This is easier from a design perspective but has the drawback that once implemented, is much harder to change. A micro-code based design takes the opposite approach and stores the information the control unit needs in the form of a very simple program that controls the data-path signals. This program is much lower-level than that which is exposed to the programmer and must be stored in some dedicated on-chip memory. The advantage is that the micro-code can change, and hence the processor operation can be altered, without changing the actual data-path logic.

▷ **S226.** a Quite often, ARM included, the latency of multiplication is higher than that of other arithmetic instructions: it takes multiply cycles to complete one multiplication, whereas it might take only one cycle to complete an addition. As such, it can be attractive to decompose multiplication by known constants into additions. For example, if we want to evaluate the expression

```
x * 3
```

this might be achieved via

```
x  + ( x * 2 )
```

or

```
x  + ( x << 1 )
```

This is possible because

$$x + (x \ll 1) = x + (x \cdot 2^1) = x + (2 \cdot x) = 3 \cdot x.$$

Notice that in the former case, we need two additions whereas in the latter we need one addition and one shift. However, using the form of instruction we have available in the ARM, the cost of the shift is zero and hence we can perform the multiplication for the cost of a single addition.

b This is a composite addressing mode: it uses direct access to memory via a address taken from $GPR[*][y]$. In the second stage, it updates $GPR[*][y]$ to point at the next byte in memory; this is an example of auto-indexing (postfix rather than prefix in this case). As an example, consider the following contrived fragment that implements a `strcpy`-like function:

```
void strcpy( char* dst, char* src ) {
  while( *src != '\0' ) {
    *( dst++ ) = *( src++ );
  }

  *dst = '\0';
}
```

Notice that the operation `*( src++ )` maps directly onto the instruction we have: the meaning is to load a byte from wherever the pointer `src` says to, and then increment `src` to point at the next byte.

An alternative use might be as a stack pop operation: if the stack pointer is held in $GPR[*][y]$ then we essentially load an element from the top of the stack, then increment the stack pointer.

c The most obvious answer is that since ARM targets the mobile and embedded market, issues of low-power and low-area are paramount: excluding caches can reduce power consumption and area and is therefore a reasonable choice. A second reason might be to do with deterministic performance. When a data or instruction cache is included, it is harder to reason about how many cycles a program will take to execute: this depends on the cache behaviour, e.g., the number of cache-hits and cache-misses. In some situations, such a feature is viewed as disadvantageous even if it means higher performance on average; examples include real-time control systems which one might argue are within the remit of embedded processors. Finally, if the clock speed of the processor is low (which for an embedded processor seems possible) then the difference between the processor and main memory speed is less apparent; as such, it may not be as worthwhile having a cache as if there was a larger difference (as in a desktop processor).

d By making the program counter a general-purpose register, the trade-off made is that with a fixed register address size,

- one has less addressable general-purpose registers (since the program counter uses one up), but
- there is no need for special-purpose addressing modes or instructions to access the program counter.

As such, there is no need for a "branch using register" instruction (since this can simply be a move into GPR[15]), and no need for "program counter relative" offset addressing modes (since one can simply use GPR[15] as the base address). When viewed across the entire instruction set, a general trend is toward fewer special-purpose instructions (and therefore arguably greater flexibility); this has a knock-on effect of freeing space within the instruction encoding for use elsewhere.

Part XIII: Processor design: advanced micro-architecture

▷ **S227.** First, note that the question is deliberately deceptive by mixing the use of indexed *and* aliased register identifiers. In particular, because

$$\begin{array}{ccccc} \mathsf{GPR}[0] & \equiv & \texttt{r0} & \equiv & \texttt{a1} \\ \mathsf{GPR}[2] & \equiv & \texttt{r2} & \equiv & \texttt{a3} \end{array}$$

the `sub` instruction should be interpreted as using the operands GPR[0] and GPR[2] as produced by the `ldr` and `add` instructions respectively; it does not use GPR[0] as produced by the `mov` instruction, because `ldr` overwrites it and therefore yields the most recent value.

Assuming no stalls occur, e.g., due to the memory access, if the `sub` instruction has reached the execute stage, then `add` instruction has reached the memory access stage, the `ldr` instruction has reached the write-back stage, and the mov instruction has completed execution (i.e., is no longer in-flight within the pipeline). This implies that, with respect to the sub instruction,

- GPR[0] is forwarded from the pipeline register between memory access and write-back stages (i.e., M/W),
- GPR[2] is forwarded from the pipeline register between execute and memory access stages (i.e., E/M).

▷ **S228.** Although the question is under-defined in the sense that a correspondence between the fetch-decode-execute cycle and bus cycles is unclear, notice that the processor uses a pipelined micro-architecture: as such, it will fetch an instruction close to every cycle. This suggests it will access memory one per cycle, irrespective of whether the instruction executed is a ldr, str, or indeed nop.

The most plausible explanation is therefore that the processor has an L1 cache: this means instruction fetches and data accesses can often (based on the principle of locality) be satisfied by the cache without a need for interaction between the processor and memory via the bus.

▷ **S229.** a The idea is that the original data-path is split up into stages which match the sequential steps it takes. By operating these stages in parallel with each other, the pipeline can be executing instruction #1 while decoding instruction #2 and fetching instruction #3 for example. Although there are some caveats to this, for example dependencies between instructions might prevent the pipeline from advancing, generally this will improve performance significantly. For example, the instruction throughput of an n stage pipelined design will approach an n-fold improvement on the multi-cycle design given suitably written programs.

The basic design for a pipelined processor data-path can be split into five stages with pipeline registers between them to prevent a stage updating the working data for another stage:

```
      +-+         +-+         +-+         +-+
+---+ |R| +---+ |R| +---+ |R| +---+ |R| +---+
|FET| |E| |DEC| |E| |EXE| |E| |MEM| |E| |WRI|
+---+ |G| +---+ |G| +---+ |G| +---+ |G| +---+
      +-+         +-+         +-+         +-+

       ---- flow of instructions ---->
```

The fetch stage will perform an instruction fetch, it will house the PC and an interface to the instruction memory and typically an adder to increment the PC. The decode phase decodes the instruction and fetches register operands, therefore it houses the GPR. The execution stage does the actual instruction execution and will therefore house the ALU that performs arithmetic and comparisons. The memory stage simply has to perform data memory access so only needs an interface to the data memory. Finally, the write-back stage houses no actual components but has hooks back into the GPR in the decode stage so it can write values. The IR is sort of spread along the whole pipeline in the pipeline registers; each stage is operating on a different instruction so each stage has its own IR in some sense.

b A structural dependency is where two stages of the pipeline need to access a single hardware module at the same time. For example, this is commonly caused when one needs to increment the program counter and perform an ALU operation at the same time. A control dependency exists since the processor does not know where to fetch instructions from until the branch is executed and so could, if not controlled, execute instructions from the wrong branch direction. A data dependency is where a computational instruction is being processed by the pipeline but the source values have not been generated or written-back to the register file.

c The solution for structural dependencies is simple: one either replicates the hardware module so both competing stages have their own, dedicated device; or one ensures somehow that they use one single device in different phases of the same cycle. One way of dealing with control dependencies is to delegate the problem to the compiler and ensure that the delay slot following the branch is either filled with null operations whose execution does not effect either branch direction, or valid instructions scheduled from before the branch. We can also reduce the chance of stall by better predicting which direction the branch will take and hence lessening the chance of having to squash invalid instructions in the pipeline. A branch target buffer of some sort will usually accomplish this. Data dependencies are usually solved by forwarding results directly from within the pipeline back to the execution unit. Using this technique, the results do not need to be written-back before they are used during execution. Using these techniques, the processor seldom needs to stall as a result of any dependency although clearly they complicate the actual processor operation.

d A branch delay slot is the space in time between when a branch is fetched and when it is executed. During this time the processor might have to stall since it cannot know instructions that would be fetched actually need to be executed. However, if the processor does not stall but just carries on fetching and executing instructions regardless, there is an opportunity to place instructions we want to execute in the branch delay slot. The advantage of doing so is that there is less wasted time but in order to realise this improvement we must be able

…	…	…
		GPR[1] ← 0
GPR[3] ← MEM[]B + GPR[1]]	GPR[4] ← MEM[]C + GPR[1]]	
		GPR[1] ← GPR[1] + 1
GPR[5] ← MEM[]B + GPR[1]]	GPR[6] ← MEM[]C + GPR[1]]	GPR[7] ← GPR[3] + GPR[4]
MEM[]A + GPR[1]] ← GPR[7]		GPR[1] ← GPR[1] + 1
GPR[3] ← MEM[]B + GPR[1]]	GPR[4] ← MEM[]C + GPR[1]]	GPR[8] ← GPR[5] + GPR[6]
MEM[]A + GPR[1]] ← GPR[8]		GPR[1] ← GPR[1] + 1
GPR[5] ← MEM[]B + GPR[1]]	GPR[6] ← MEM[]C + GPR[1]]	GPR[7] ← GPR[3] + GPR[4]
MEM[]A + GPR[1]] ← GPR[7]		GPR[1] ← GPR[1] + 1
GPR[3] ← MEM[]B + GPR[1]]	GPR[4] ← MEM[]C + GPR[1]]	GPR[8] ← GPR[5] + GPR[6]
MEM[]A + GPR[1]] ← GPR[8]		GPR[1] ← GPR[1] + 1
…	…	…

Figure 2: *A VLIW implementation of the vector addition loop body.*

to reorder instructions in the program so that the slot is filled; if we cannot fill the slot then NOPs must be placed there instead to ensure correct execution. This all means that the processor can be simpler due to the lack of stall condition, but the programmer must do more work to extract the improvement in performance and ensure correctness.

In most RISC architectures, alteration of control-flow is done explicitly using dedicated branch instructions. Predicated execution offers an alternative to this model by associating a predicate function to each instruction and only executing the instruction if the predicate is true. Rather than set the *PC* to some new value, this approach allows one to skip over instructions based on the associated predicate. It is ideal for conditional execution of short sections of code due to the low overhead associated. In this case, both the processor and programmer must be more complicated; the processor must be able to decode and evaluate predicates and discard instructions which fail, the programmer must be able to write programs in a style that takes advantage of this feature.

Branch prediction reduces the number of stalls by guessing which way a branch will complete and continuing to fetch instructions from that path; if the prediction is wrong then the fetched instructions must be squashed but if it is right execution can continue with no loss of time. Common prediction mechanisms are simple, for example one might predict that backwards branches (that model loops) are always true: this will be correct most of the time because a loop only exists once and iterates many times. The complexity in branch prediction is entirely in the processor; the programmer need do nothing more than normal unless the ISA allows hints to be passed about the probably direction of branches.

▷ **S230.** After partially unrolling the loop by a factor of four, it is clear that the four additions that represent the new loop body are independent. With a vector processor, either dedicated or in the form of an extension to a conventional processor (e.g., MMX/SSE), these can be executed in parallel. Starting with the scalar implementation

```
        $\ldots$
        $\mathsf{GPR}[1] \leftarrow 0$
        $\mathsf{GPR}[2] \leftarrow 400$
$loop:$  if $\mathsf{GPR}[1] \geq \mathsf{GPR}[2], PC \leftarrow exit$
        $\mathsf{GPR}[3] \leftarrow \mathsf{MEM}[]\mathtt{B} + \mathsf{GPR}[1]]$
        $\mathsf{GPR}[4] \leftarrow \mathsf{MEM}[]\mathtt{C} + \mathsf{GPR}[1]]$
        $\mathsf{GPR}[5] \leftarrow \mathsf{GPR}[3] + \mathsf{GPR}[4]$
        $\mathsf{MEM}[]\mathtt{A} + \mathsf{GPR}[1]] \leftarrow \mathsf{GPR}[5]$
        $\mathsf{GPR}[1] \leftarrow \mathsf{GPR}[1] + 1$
        $PC \leftarrow loop$
$exit:$  $\ldots$
```

the basic idea would be to utilise the vector register file and associated ALUs to load and execute four additions at once. This could be described as

```
        $\ldots$
        $\mathsf{GPR}[1] \leftarrow 0$
        $\mathsf{GPR}[2] \leftarrow 400$
$loop:$  if $\mathsf{GPR}[1] \geq \mathsf{GPR}[2], PC \leftarrow exit$
        $\mathsf{VR}[1] \leftarrow \mathsf{MEM}[]\mathtt{B} + 0 + \mathsf{GPR}[1] \ldots \mathtt{B} + 3 + \mathsf{GPR}[1]]$
        $\mathsf{VR}[2] \leftarrow \mathsf{MEM}[]\mathtt{C} + 0 + \mathsf{GPR}[1] \ldots \mathtt{C} + 3 + \mathsf{GPR}[1]]$
        $\mathsf{VR}[3] \leftarrow \mathsf{VR}[1] + \mathsf{VR}[2]$
        $\mathsf{MEM}[]\mathtt{A} + 0 + \mathsf{GPR}[1] \ldots \mathtt{A} + 3 + \mathsf{GPR}[1]] \leftarrow \mathsf{VR}[3]$
        $\mathsf{GPR}[1] \leftarrow \mathsf{GPR}[1] + 4$
        $PC \leftarrow loop$
$exit:$  $\ldots$
```

whereby we get roughly a four fold improvement in performance as a result. Of course this depends on being able to load and store operands quickly enough (this is assisted by the regular nature of such accesses), and that there are enough ALUs to execute the parallel additions.

The case for VLIW is more complicated, in part because such an architecture is less well defined. Imagine we have a VLIW processor which can accept instruction bundles that include two memory accesses and one arithmetic operations. The idea here would be to software pipeline the loop so that we "skew" the iterations: this allows us to load the operands for iteration 1 at the same time as performing the addition for iteration 0 for example. Reading operations on the same row as being executed in parallel (the first two columns being memory access and the third being arithmetic), an example code sequence is show in Figure 2. Notice that by overlapping the loop iterations, we are roughly executing one iteration in two instruction bundles rather than via five conventional instructions as above (even though there are some empty slots in selected bundles).

▷ **S231.** a Consider an example function which sums elements from two arrays and stores the result into a third:

```
void add( int* A, int* B, int* C, int p ) {
  for( int i = 0; i < p; i++ ) {
    A[ i ] = B[ i ] + C[ i ];
  }
}
```

On a scalar processor, each iteration of the loop is executed separately. The use of processor designs such as superscalar might capitalise on the implicit parallelism between loop iterations but essentially any given operation only deals with one (scalar) data item at a time. The idea of a vector processor is to make this parallelism explicit and equip the processor with the ability to operate on many data items (vector) using a single operation. As such, instead of executing many scalar operations of the form

```
A[ 0]   ←  B[ 0] + C[ 0]
A[ 1]   ←  B[ 1] + C[ 1]
A[ 2]   ←  B[ 2] + C[ 2]
            ⋮
A[p-1]  ←  B[p-1] + C[p-1]
```

$$
\begin{array}{lclcccccccl}
r_0 & = & (& D & , & C & , & B & , & A &) \\
r_1 & = & (& 2D & , & 2C & , & 2B & , & 2A &) & \text{add } r_0 \text{ and } r_0 \\
r_2 & = & (& 2D+1 & , & 2C+1 & , & 2B+1 & , & 2A+1 &) & \text{add } r_1 \text{ and } 1 \\
r_3 & = & (& D(2D+1) & , & C(2C+1) & , & B(2B+1) & , & A(2A+1) &) & \text{mul } r_2 \text{ and } r_0 \\
r_4 & = & (& (D(2D+1)) \lll 5 & , & (C(2C+1)) \lll 5 & , & (B(2B+1)) \lll 5 & , & (A(2A+1)) \lll 5 &) & \text{rotate } r_3 \text{ by } 5 \\
\multicolumn{11}{c}{\text{At this point we rename the sub-words to make life easier}} \\
r_4 & = & (& D' & , & C' & , & B' & , & A' &) \\
r_5 & = & (& C' & , & D' & , & A' & , & B' &) & \text{shuffle } r_4 \\
r_6 & = & (& C' & , & B' & , & A' & , & D' &) & \text{shuffle } r_4 \\
r_7 & = & (& D \oplus C' & , & C \oplus D' & , & B \oplus A' & , & A \oplus B' &) & \text{XOR } r_5 \text{ and } r_0 \\
r_8 & = & (& (D \oplus C') \lll C' & , & (C \oplus D') \lll B' & , & (B \oplus A') \lll A' & , & (A \oplus B') \lll D' &) & \text{rotate } r_7 \text{ by } r_6 \\
r_9 & = & (& 0 & , & K[i+1] & , & 0 & , & K[i+0] &) & \text{load round key} \\
r_{10} & = & (& (D \oplus C') \lll C' & , & ((C \oplus D') \lll B') + K[i+1] & , & (B \oplus A') \lll A' & , & ((A \oplus B') \lll D') + K[i+0] &) & \text{add } r_8 \text{ and } r_9 \\
r_{11} & = & (& A & , & C & , & D & , & B &) & \text{shuffle } r_0 \\
r_{12} & = & (& (B \oplus A') \lll A' & , & ((D \oplus C') \lll C') + K[i+1] & , & (A \oplus B') \lll D' & , & ((C \oplus D') \lll B') + K[i+0] &) & \text{shuffle } r_8 \\
r_{13} & = & (& (A \oplus B') \lll D' & , & D & , & (C \oplus D') \lll B' & , & B &) & \text{unpack } r_{11} \text{ and } r_{12} \\
\multicolumn{11}{c}{\text{At this point we rename the sub-words to make life easier}} \\
r_{13} & = & (& A'' & , & D'' & , & C'' & , & B'' &)
\end{array}
$$

Figure 3: *A vector implementation of the RC6 loop body.*

the vector processor executes one vector operation of the (pseudo-C) form

$$\texttt{A[0 ... p-1]} \leftarrow \texttt{B[0 ... p-1] + C[0 ... p-1]}.$$

To achieve this, the processor needs two key components: a vector register file capable of storing vectors (i.e., many data items rather than single data items), and a number of processing elements each of which operates on one element of the input vectors to produce one element of the output vector. The processing elements can be sub-pipelined if if there are less of them than there are elements in a vector, they may need to be used iteratively to operate across the entire vector content.

Typically these components are gathered together into a single co-processor type unit which is invoked by a standard scalar processor (i.e., rather than use the scalar register and ALU) when vector instructions are encountered. This approach needs careful management of memory access so that a single interface to memory can be used effectively to load and store vector data.

b The use of a vector processor can yield higher performance than a scalar processor (for a selected class of programs) because:

- Parallelism in the program is explicit. The processor can therefore take advantage of this parallelism, with very little effort or overhead in terms of hardware complexity, by using multiple processing elements. That is, processing element i can operate on element i of the input vectors to produce element i of the output at the same time as other elements $j \neq i$ are being produced in parallel. Consider a processor that operates on vectors of length p with r processing elements; the processor achieves a theoretical peak performance improvement of p/r over the scalar processor if the program is easily vectorised.
- In scalar, multiple-issue architectures such as superscalar, the demand for new instructions is very high: they must be fetched from memory fast enough to satisfy the aim of issuing one operation per-cycle. Although cache memories and other techniques can help, this can present a problem. In contrast, the amount of work done by per-operation in a vector is much higher (e.g., a factor of p higher if we work with vectors of length p). As such, the pressure for operations to be fetched quickly is less and hence the impact of the bottleneck represented by main memory is less pronounced.
- Access to vector data held in memory is typically more regular than that to scalar data. That is, if we load a p element vector at address A, we explicitly load from addresses $A + 0 \ldots A + p - 1$. The memory hierarchy can capitalised on this feature via specialist cache design and operational policies, prefetch and striding, burst access and so on. In the case of access to scalar is not as easy to deal with: single accesses are disjoint and cannot be easily related to the overall task of access to the larger vector.

c Say we represent the input at the start of the loop body as (D, C, B, A), i.e., a packed 128-bit register of 32-bit sub-words. We assume that the array K has been padded with space so that when elements are loaded during iteration i, we get the vector $(0, K[i + 1], 0, K[i + 0])$. Finally, we assume that all computation is performed component-wise on sub-words (i.e., using strict SIMD) but that there are a number of standard sub-word organisation operations such as shuffle and pack. Equipped as such, we can implement the loop body using the operation sequence in Figure 3 so that at the end, we have the vector (A'', D'', C'', B'') which is used as input to the next iteration.

d Implementation of the loop body using 3-address style RISC operations can be achieved using the following sequence

$$\begin{array}{lcl}
t_0 & \leftarrow & B + B \\
t_1 & \leftarrow & t_0 + 1 \\
t_2 & \leftarrow & B \cdot t_1 \\
t_3 & \leftarrow & t_2 \lll 5 \\
t_4 & \leftarrow & D + D \\
t_5 & \leftarrow & t_4 + 1 \\
t_6 & \leftarrow & D \cdot t_5 \\
t_7 & \leftarrow & t_6 \lll 5 \\
t_8 & \leftarrow & A \oplus t_3 \\
t_9 & \leftarrow & t_8 \lll t_7 \\
t_{10} & \leftarrow & \mathsf{MEM}[](K + 0) + i] \\
t_{11} & \leftarrow & t_9 + t_{10} \\
t_{12} & \leftarrow & C \oplus t_7 \\
t_{13} & \leftarrow & t_{12} \lll t_3 \\
t_{14} & \leftarrow & \mathsf{MEM}[](K + 1) + i] \\
t_{15} & \leftarrow & t_{13} + t_{14} \\
A & \leftarrow & B \\
B & \leftarrow & t_{13} \\
C & \leftarrow & D \\
D & \leftarrow & t_8
\end{array}$$

This scalar sequence requires 20 operations; the vector equivalent requires 13 operations. However, the four move operations at the end of the scalar implementation are essentially just renaming intermediate variables: the actual moves are not required. Therefore by unrolling the loop somewhat it should be possible to remove these and reduce the operation count to 16. Although the sequences are not scheduled intelligently, it seems possible that with some rearrangement (plus possibly some software pipelining) it is possible to execute roughly one operation per-cycle. As such, the vectorisation has produced roughly an 19% improvement in performance (versus roughly 35% without unrolling).

The theoretical limit is a four-fold improvement in performance since one would expect roughly four operations to occur in parallel in the vector case. The two main factors which prevent the improvement being higher are wasted work in computing values we do not need (e.g., the $(D(2D + 1)) \lll 5$ sub-word of r_4), and the amount of shuffle type operations required to move sub-words into the right place for actual computation. In short, the loop looks easily vectorisable but the architecture (i.e., packing four 32-bit sub-words into 128-bit registers) does not capitalise on this well. One might change the processor to pack two 32-bit sub-words into 64-bit registers; this would prevent wasted computation and reduce either the amount of processing elements required if there is one processing element for each sub-word (e.g., two rather than four), or the time taken for each operation if there are less processing elements than sub-words (e.g., in the initial design, having two processing elements rather than four means iterating twice for each operation).

▷ **S232.** a Speculative execution is a mechanism that acts to reduce the cost of branches. The problem caused by branches is that until any branch condition and target are known, the processor cannot proceed: it cannot know where to fetch the next instructions from and hence the front-end becomes stalled. In turn, this can lead to under utilisation of the computational resources which are idle until provided with work.

The basic idea is to construct a group of components that guess which way a branch will direct control-flow, and execute instructions from that path. If the guess turns out to be correct, then execution can continue more or less as normal: the instructions allowed into the pipeline really were those the programmer intended to execute so in a sense no execution time has been wasted. If the guess turns out to be incorrect however, the instructions in the pipeline are invalid (i.e., they are from the wrong path) and need to be deleted. After cleaning up, execution is restarted along the correct path and although there is a associated cost, this is not significantly more than the case without speculation. As a result, as long as guesses made by the processor are correct more often than not, performance is improved because more often than not the front-end does not stall and can supply a more consistent stream of work to the back-end.

Note that use of a reorder buffer makes speculation much easier: we can simply include a flag in the RoB to mark speculatively executed instructions and only allow them to complete (i.e., exit the RoB) if the associated condition turns out to be true.

b The case of static (rather than fixed) branch prediction demands that the compiler and/or programmer does some analysis of the program to work out the probable control-flow. For example, the compiler might:

- Examine a particular loop statement and reason that since the loop bound is large, a branch from the end of the loop body back to the start is much more likely to be taken than not taken.

- Examine a particular conditional statement and reason that the condition is very unlikely to evaluate to false, and hence that a branch that skips the conditional body is very unlikely to be taken.

The idea is to pass the results of this analysis to the processor in order to guide any decisions. One mechanism to do this is to allow "hints" or predicates in the branch instructions that inform the processor that they are likely or unlikely to be taken: based on such a "hint", the processor might speculatively execute one control-flow path rather than another. Disadvantages of this approach is include the fact that data-dependent behaviour of the program is not considered (unless we use some form of profiling), and that there is a limited amount of space in any instruction encoding with which to accommodate the "hints".

Use of dynamic branch prediction means that the processor tries to perform analysis of the program behaviour at run-time. Although this requires extra hardware to implement the scheme, it means that any data-dependent behaviour can be detected and capitalised upon. Typically the processor would include a mechanism for predicting whether a branch is taken or not *and* where the branch target is. The branch target is possible simpler: a cache-like structure called the Branch Target Buffer (BTB) can be maintained whereby the address of a branch is mapped to a predicted target. If the branch address has no mapping in the BTB then the processor simply has to guess, but if there is a mapping (e.g., the branch was executed recently) then the processor can speculatively use the previous target. Predicting the direction of a branch can be achieved using a Branch Prediction Buffer (BPB) which maps the address of a branch to a history of that branches behaviour, i.e., a history of whether it was taken or not. Based on this history, one can make predictions about whether it will be taken or not in the future. Examples include the use of saturating counters which effectively treat the problem like a state machine where the branch history represents the current state (which acts as the prediction), each executed branch updates the state in a manner that depends on the real behaviour.

c i Assuming for example that the "call" and "return" instructions operate in a similar way to those in MIPS, a simple calling convention could be:

```
caller: push arguments
        call callee
        pop  return value
        pop  arguments
        ...

callee: push return address
        push space for locals
        ...
        pop  space for locals
        pop  return address
        push return value
        return to caller
```

In this case, the "call" places the return address in a specific register (this is a so-called "jump and link") while the "return" is implemented via a branch to an address held within any register: the address is popped from the stack into a register and then used.

ii The "call" instruction is essentially an unconditional branch: it does not require any branch prediction since it is *always* taken and the branch target is *always* known if it can be encoded as an immediate operand. The "return" instruction is more tricky: although unconditional, in this case the processor will not know the branch target (i.e., the return address) until it is retrieved from the stack.

One approach would be to employ a system whereby there is segregation between data and address related stack content, i.e., use a dedicated return address stack. This would require the "call" instruction to push the return address onto the address stack and the "return" instruction to pop it: these operations could form part of the instruction semantics. Since the address stack will probably be more constrained in terms of size and use, it could be possible to retain some of it on-chip (rather than in memory) so that access to the last say n return addresses is very fast.

Although there are potentially some disadvantages, the main advantage is that we can now "snoop" the address stack for a return address. That is, as soon as the "return" instruction is fetched we can look at the on-chip address stack and retrieve the associated branch target quickly and accurately.

Part XIV: The memory hierarchy

▷ **S233.** Note that the bus connects the processor and memory, or, at least from a logical perspective, the cache (which, since it is an L1 cache, would be internal to the processor) and memory because the processor will access memory via the cache. This means the number of transactions across the bus basically means the number of accesses the cache makes to memory.

The first choice is write-through vs. write-back. The former writes to the cache *and* memory, whereas the latter only writes to the cache: it defers writing to memory until "dirty" content is evicted. As such, write-through would generate more accesses to memory. The second choice is write-allocate vs. write-around, which relate to what happens when a write causes a cache-miss. The former means a cache-miss is similar to a cache-hit, in the sense the associated line is filled from memory, whereas the latter means the cache state is unchanged. As such, write-allocate would generate more accesses to memory. In combination then, write-through plus write-allocate is the combination that would generate more accesses to memory.

▷ **S234.** Each cache line is 64B = 2^6B, so we need 6 bits to identify a specific word (i.e., byte) in a given line. The total total capacity is 128KiB = 2^{17}B, and there are 8 = 2^3 cache lines per set; this means there are $2^{17}/(2^6 \cdot 2^3) = 2^{17}/2^{6+3} = 2^{17-(6+3)} = 2^8$ sets, so we need 8 bits to identify a specific set. The tag is essentially the rest of the address, i.e., the bits which are not used to identify the set or sub-word: such a tag is therefore $32 - (8 + 6) = 18$ bits.

▷ **S235.** The memory hierarchy is arranged like a pyramid with higher levels relying on the performance and functionality of lower levels. The idea is that higher levels are smaller and more expensive but faster, lower levels are large and inexpensive but slower. The principles of locality means that we can keep the working set of a program, i.e., the data and instructions used most often, in the small, fast areas of the hierarchy. This means the most often used data can be accessed very quickly but we can still store very large amount of data since the hierarchy is backed by large, inexpensive storage.

```
             ^
            / \
           /   \
          /     \
         /       \
        /         \
       / registers \         smallest   fastest   expensive
      /-------------\            |          |          |
     /    cache      \           |          |          |
    /-----------------\          |          |          |
   /    main memory    \         |          |          |
  /---------------------\        V          V          V
 /     disk and tape     \   largest    slowest   inexpensive
---------------------------
```

At the top of the hierarchy are registers, these are word sized areas of on-chip memory used to produce operands for instructions. Typically they total around 1kB in size. The next level is cache memory, typically made from SRAM, which can be on or off-chip and ranges in size from around 512kB upto several megabytes. Beyond any cache structures is the main RAM memory which is typically built from DRAM and is of the order of gigabytes in size. Finally, at the bottom of the hierarchy are the longer terms storage options such as disk and tape which are multi-gigabyte in size.

▷ **S236.** a To address cache lines with 8 words (i.e., 8B in each line) we need a 3-bit word address since $2^3 = 8$. There are a total of 512B which means there are $512/8 = 64$ lines; this means we need a 6-bit line number since $2^6 = 64$. This means there are $32 - 3 - 6 = 23$ bits remaining which form the tag.

b To address cache lines with 8 words (i.e., 8B in each line) we need a 3-bit word address since $2^3 = 8$. Since there are 64 lines, assuming that access in each of the $64/2 = 32$ sets is fully-associative, we need an 5-bit set number since $2^5 = 32$. This means there are $32 - 3 - 5 = 24$ bits remaining which form the tag.

c To address cache lines with 8 words (i.e., 8B in each line) we need a 3-bit word address since $2^3 = 8$. Since there are 64 lines, assuming that access in each of the $64/4 = 16$ sets is fully-associative, we need an 4-bit set number since $2^4 = 16$. This means there are $32 - 3 - 4 = 25$ bits remaining which form the tag.

d To address cache lines with 8 words (i.e., 8B in each line) we need a 3-bit word address since $2^3 = 8$. In a fully associative cache lines can be placed anywhere so there is no need to derive them from the address; this means there are $32 - 3 = 29$ bits remaining which form the tag.

▷ **S237.** • Compulsory misses are caused by the first access to some content held in memory. In a sense these seem unavoidable: before an element X is accessed by the program, how can the processor know it will be accessed and therefore prevent the cache-miss? The answer is to take a speculative approach and pre-fetch content into the cache that has a reasonable probability of use; if the speculation is correct then the associated compulsory miss is prevented, if it is incorrect then there will be a cache-miss which is no worse than our starting point. The major disadvantage of incorrect speculation is the potential for cache pollution which can increase conflict misses. Thus, some balance needs to be arrived at between the two approaches.

- Capacity misses are misses that are independent of all factors aside from the size of the cache; essentially the only solution is to increase the cache size so a larger working set can be accommodated.
- Conflict misses are caused by the eviction of resident cache content by a newly accessed content that maps to the same cache line. The implication is that if element X is evicted by element Y, there has been a conflict in the address translation: since both map to the same line the access to Y causes a cache-miss. This effect can continue in the sense that if X is accessed again later, it will cause another cache-miss. The use of set-associative caches can reduce the level of contention in the cache by allowing conflicting addresses to coexist within a fully-associative set. Alternatively one might take a software approach and skew the addresses at which X and Y are placed so they no longer map to the same cache line.

▷ **S238.** a There are 4 bytes per-line so we need 2 bits to address these since $2^2 = 4$. These are taken from the least-significant end of the address. There are 256 lines since the cache is 1kB in size; we need 8 bits to address these lines since $2^8 = 256$. These are again taken from the least-significant end, starting after the bits for the word address. The tag constitutes the rest of the bits so we can uniquely identify a given address in a line, it is therefore 22 bits in size.

b Spatial locality is the tendency that if one item is accessed, those items close to it in memory are also accessed. For example if `A[ 1 ]` is accessed there is a good chance `A[ 0 ]` or `A[ 2 ]` will be accessed rather than say `A[ 1000 ]`. Temporal locality is the tendency that if one item is accessed, it will be accessed again in the near future. For example, when the add instruction which implements the addition of `B[ i ]` and `C[ i ]` is accessed, it has a high probability of being accessed again since it is in a loop. Cache interference is where items compete for space in the cache. If two items map to the same line and one is loaded into the line, an access to the other will evict or displace the original since the line cannot hold both.

c The load access to `B[ 0 ]` will miss since this item is not in the cache. Such a miss means the item is loaded into cache line one, along with the values `B[ 1 ]`, `B[ 2 ]` and `B[ 3 ]`. The load access to `C[ 0 ]` also misses but since it maps to the same line as `B[ 0 ]`, the data loaded evicts the values of `B` loaded in the previous load. Finally, the store to `A[ 0 ]` misses and again maps to the same line as `C[ 0 ]` so again evicts the current values. The loop continues in this manner, essentially never producing any cache-hits because each access evicts the previous data.

d i This alteration changes matters entirely since now, `A[ 0 ]` maps to line 0, `B[ 0 ]` maps to line 1 and `C[ 0 ]` maps to line 2. As a result each access to `B[ i ]` will provoke one miss to load the values into the line, followed by three hits as the values `B[ i + 1 ]`, `B[ i + 2 ]` and `B[ i + 3 ]` will sit in the same line. The same is true for accesses to `C[ i ]` and `A[ i ]` so that the program will run much faster due to the decreased number of accesses to main memory.

ii The basic problem here is one of cache interference: we want to prevent or reduce the probability of two addresses mapping to the same line in the cache and evicting each other. Associative caches are one solution to this problem; they allow any address to map to any line and include some decisional mechanism that places them. For example, Least Recently Used (LRU) or random placement both solve the interference problem. The problem will associative caches is that they require significant resource to build. As a compromise, set-associative caches are a better option. A set-associative cache has number of sets each of which is like a small cache; inside each set look-up is fully associative but since the sets are small, the resulting cost is manageable. The idea is that we select the set a data item will go into using some more bits from the address: items that would map to the same line like `A`, `B` and `C` now map into the same line within different sets so don't interfere.

▷ **S239.** a Consider the example program

```
void vecvec_add(       float* A,
                 const float* B,
                 const float* C,
                       int    n ) {
  for( int i = 0; i < n; i = i + 1 ) {
    A[ i ] = B[ i ] + C[ i ];
  }
}
```

which performs the addition of `B` and `C`, two `n`-element vectors, to produce `A`.

i Temporal locality is the tendency that if one address is accessed, it will be accessed again in the near future. For example, when the instruction which performs the addition of `B[ i ]` and `C[ i ]` is fetched, there is a good chance that it will be fetched again since it is within a loop.

ii Spatial locality is the tendency that if one address is accessed, those addresses close to it are also accessed. For example given an access to `A[ 1 ]`, there is a good chance that either `A[ 0 ]` or `A[ 2 ]` will be accessed next rather than say `A[ 1000 ]`.

b First, note that

$$\begin{array}{rcl} 2^2 &=& 4 \\ 2^3 &=& 8 \\ 2^4 &=& 16 \end{array}$$

so in the first case there are 8 lines each of 8 words, and in the second case there are 4 lines each of 16 words; either way the cache holds $t = 64$ bytes in total.

i For $l' = 3$ and $w' = 3$, the least-significant 3 bits of x are the word number and the next least-significant 3 bits of x are the line number; this implies the most-significant $8 - 3 - 3 = 2$ bits of x are the tag. Since $x = 42 = 00101010_{(2)}$, the word number is $010_{(2)} = 2$, the line number is $101_{(2)} = 5$, and the tag is $00_{(2)} = 0$.

ii For $l' = 2$ and $w' = 4$, the least-significant 4 bits of x are the word number and the next least-significant 2 bits of x are the line number; this implies the most-significant $8 - 4 - 2 = 2$ bits of x are the tag. Since $x = 8 = 01010001_{(2)}$, the word number is $0001_{(2)} = 1$, the line number is $01_{(2)} = 1$, and the tag is $01_{(2)} = 1$.

c i For larger l and smaller w, example answers include

- A smaller line size (i.e., smaller w) means that each time a line is filled from or written-back to main memory, less content is transferred. This might be viewed as an advantage if the latency of access to main memory is particularly high (e.g., if there is no facility for burst access) and there are numerous accesses (e.g., there is a high miss-ratio or a write-through policy is used).
- If the level of spatial locality is low, a smaller line size (i.e., smaller w) might be advantageous: there is less effort wasted by fetching content from main memory that will not be used.

ii For larger w and smaller l, example answers include

- For this cache, one always has the same sized tag: for an n-bit address, the tag is formed by taking the $n - w' - l'$ most-significant bits. Given this fact, if there are less lines (i.e., smaller l) there there are less tags to store; hence a smaller l might imply a more space-efficient implementation.
- If the level of spatial locality is high, a larger line size (i.e., larger w) might be advantageous: there is a higher chance that the next address accessed will be within the same cache-line as the current previous address (and hence a cache-hit).

d i The idea a segregated cache architecture is that there are separate data and instruction caches: instruction fetches are performed through the instruction cache (or "I-cache"), data loads and stores are performed through the data cache (or "D-cache"). The alternative is a unified architecture where all access are performed through a single cache. In a stored-program or von Neumann architecture it is important to note that there is *still* a single main memory which backs *both* caches in a segregated architecture; there are simply two interfaces to this memory.

Use of a segregated architecture can provide a variety of advantages. For example:

- A dedicated instruction cache can often allow a more efficient implementation since it does not (necessarily) need to cater for stores: the only operations performed are instruction fetches which are essentially loads from memory.
- The fact that there are two interfaces to memory (and these are used by different stages of the fetch-decode-execute cycle) means that instruction fetches and data load/store operations can be performed at the same time. This is particularly advantageous where the micro-architecture is pipelined and there are separate fetch and memory access stages: the two interfaces mean that the two stages are not "blocked" by each other.
- Since the instruction and data cache are separate, they can operate using different geometries and policies (e.g., line size, write policy etc.) that suite their use. For example, one might reason that a stream of instruction fetches has a high(er) degree of spatial locality and hence use a long(er) line size for the instruction cache to capitalise on this; such a choice can be made independently from the line size selected for the data cache.

ii Imagine we want to write an encoded instruction i into address x in memory; subsequent execution of the instruction could be performed by simply branching to it, e.g., setting $PC = x$. In order to store i in memory, we actually need to execute a store instruction, e.g.,

$$\mathsf{MEM}[x] \leftarrow i.$$

Such a store will be performed through the data cache, and (at least) two issues occur:

i. If the data cache employs a write-through policy, the store of i into $\mathsf{MEM}[x]$ will cause $\mathsf{MEM}[x]$ to be allocated space in the data cache but also for i to be written into main memory. If however a write-back policy is used, main memory may not be updated by the time we set $PC = x$. That is, if the data cache content is not evicted and thereby written into main memory, any fetch from MEM$[x]$ will not "see" the new value.

ii. If the instruction cache holds MEM$[x]$ at the time when the store is executed, and it has not been evicted by the time we set $PC = x$, then, even if the data cache is flushed and hence main memory correctly reflects the store, a fetch will yield the stale value held in the instruction cache rather than i.

Beyond the obvious approach of unifying the two caches, the concept of cache flushing could help: the idea is that once self-modification is complete, i.e., after each i has been stored, the caches are emptied of content. This means that all values within either cache will be written into main memory, and neither cache will hold values relating the instructions written. This can be achieved in (at least) two ways:

i. With suitable hardware support, one can rely on the hardware to perform the flushing operation. For example, x86 includes an instruction called `clflush` to do just this.
ii. If there is no support from hardware, one can still flush the caches by forcing eviction of data; most simply, this can be achieved (in the case of the data cache for example) by loading or storing values from or into a "dummy" array of the same size as the cache:

```
void flush() {
  uint8_t* T = ( uint8_t* )( malloc( CACHE_SIZE ) ), t;

  for( int i = 0; i < CACHE_SIZE; i++ ) {
    t = T[ i ];
  }
}
```

After the function executes, the contents of the cache will have been filled with `T` hence forcing eviction of any data previously resident.

▷ **S240.** a The motivation for including a cache in the processor design is the relatively high latency of access to main memory in comparison with the length of a processor cycle: since the processor will potentially access memory as often as every cycle, this problem acts as a limit to how fast instructions (and hence programs) can be executed. Access to a cache memory can have a lower latency provided the working set of data can be retained in it (given the cache is typically has a much smaller capacity).

The description of the cache as general-purpose refers to the fact it holds data *and* instructions, rather than segregating the cache into special-purpose devices for each type of access.

b The theme in this part of the question is *how* the cache satisfies the motivation above, i.e., how it is able to reduce access latency.

i Consider the example program

```
void vecvec_add(       float* A,
                 const float* B,
                 const float* C,
                           int  n ) {
  for( int i = 0; i < n; i = i + 1 ) {
    A[ i ] = B[ i ] + C[ i ];
  }
}
```

which performs the addition of `B` and `C`, two n-element vectors, to produce `A`.

i. Spatial locality is the tendency that if one address is accessed, those addresses close to it are also accessed. For example given an access to `A[ 1 ]`, there is a good chance that either `A[ 0 ]` or `A[ 2 ]` will be accessed next rather than say `A[ 1000 ]`.
ii. Temporal locality is the tendency that if one address is accessed, it will be accessed again in the near future. For example, when the instruction which performs the addition of `B[ i ]` and `C[ i ]` is fetched, there is a good chance that it will be fetched again since it is within a loop.

These features mean that if the cache keeps a small (relative to the size of main memory) set of data resident, there will be a high probability that a given access can be satisfied by the cache alone (i.e., be a cache-hit) rather than require an access to main memory (i.e., be a cache-miss).

ii One class of (or reason for) cache-misses is termed "capacity misses" and occur purely because the cache is not large enough to hold the working set; if the cache were larger, the cache-miss might be turned into a cache-hit by virtue of the cache having more resident data.

As such, a potential advantage of the larger cache is less cache-misses and hence lower overall latency. This does not happen without an associated cost however: the larger cache requires more physical area and will probably consume more power, both of which represent tangible disadvantages.

iii Following the above, another class of cache-miss are those described as "conflict misses" whereby an access to one address causes the eviction of data associate with another; the addresses conflict, or interfere with each other.

The concept of associativity can help reduce conflicts by allowing an address to map into one of *many* lines rather than one as in the case of a direct-mapped cache. Instead an address first maps into a set (which can be the same size as the cache for fully-associative designs) which is then searched for the corresponding line (via comparison of tags). As a result, addresses that might have previously mapped to the same line of a direct-mapped cache and hence implied conflict, they can coexist within a set.

c Keeping in mind that the processes uses 32-bit addresses, we find:

i the word address requires 2 bits, since there are 4 words per-line,

ii the line address requires 0 bits, since a set-associative cache does not directly map an address to a line,

iii the set address requires 9 bits, since a total of $8192/4 = 2048$ lines are shared into $2048/4 = 512$ sets, and

iv the tag requires $32 - 2 - 9 = 21$ bits.

d The obvious alternative would be a write-back approach. The idea is to associate another flag (in addition to the valid flag) with each cache line: the "dirty" flag keeps track of whether the data in a cache line has been altered or not. Initially, the dirty bit for each line is set to **false**. To deal with a store into address x, the cache operates as follows:

i Translate x into a set number x_{set} and (sub)word address x_{word}.

ii Check whether x is resident within x_{set}; if not, a fetch is performed to make it resident. To implement the write-back policy, if a line is evicted during the fetch then the associated dirty flag is checked. If the flag is **true** then the data held by the line has been altered and needs to be stored in main memory to ensure consistency; otherwise no store is required (since the cache and main memory are already consistent).

iii Updates word x_{word} within the appropriate line in x_{set} with the data, and sets the dirty flag to **true** so the line is viewed as altered.

The advantage of this approach is a reduction in accesses to main memory (even though the latency can be hidden, the extra accesses made by a write-through policy can still imply unnecessary power consumption); disadvantages include the extra space required for the additional dirty flags, and extra complexity in the algorithms for access (e.g., to check and update said flags).

▷ **S241.** a The main components to include in such a diagram are

i the four internal transistors that form a loop of two NOT gates, and

ii the two external transistors that allow access to the loop via the bit- and word-lines.

The diagram is as follows:

```
                                  wl
                                  |
                                  v
           +----------------------+----------------------+
           |                                             |
           |       ----+-------------------+----- V_dd   |
           |           |                   |             |
           |           v                   v             |
           |       +---------+         +---------+       |
           |       | P-type  |--+   +--| P-type  |       |
           |       +---------+  |   |  +---------+       v
           |           |        |   |      |        +---------+
           v           |        +---|------+--------| N-type  |<-- ~bl
      +---------+      |        |   |      |        +---------+
bl -->| N-type  |------+--------|---+      |
      +---------+      |        |   |      |
                   +---------+  |   |  +---------+
                   | N-type  |--+   +--| N-type  |
                   +---------+         +---------+
                       ^                   ^
                       |                   |
                   ----+-------------------+----- VSS
```

b The algorithm consists of three main parts, namely address translation, line filling and line access. Assuming $\mathsf{CACHE}[i]$ denotes the i-th line, and $\mathsf{CACHE}[i]_{valid}$ and $\mathsf{CACHE}[i]_{data}$ denote associated valid flag and data (i.e., the sequence of words), the algorithm is as follows:

```
Input: An 32-bit address $x$
Output: The $w$-bit value held in MEM$[x]$
$x_{word} \leftarrow x \bmod w$
$x_{line} \leftarrow \lfloor x/w \rfloor \bmod l$
$x_{tag} \leftarrow \lfloor (x/w)/l \rfloor$
$x_{wordless} \leftarrow \lfloor x/w \rfloor \cdot w$
if CACHE$[x_{line}]_{valid}$ = false or CACHE$[x_{line}]_{tag} \neq x_{tag}$ then
    for $i$ from 0 upto $w - 1$ do
        CACHE$[x_{line}]_{data[i]} \leftarrow$ MEM$[x_{wordless} + i]$
    end
    CACHE$[x_{line}]_{valid} \leftarrow$ true
    CACHE$[x_{line}]_{tag} \leftarrow x_{tag}$
end
return CACHE$[x_{line}]_{data[x_{word}]}$
```

c i First, we find that

i. the word address requires 3 bits, since there are $2^3 = 8$ words per-line,
ii. the line address requires 7 bits, since there are $2^7 = 128$ lines, and
iii. the tag requires 22 bits, since of a 32-bit address 10 are used to specify the word and line address.

The total number of bits required to store the data is $l \cdot w \cdot 8 = 128 \cdot 8 \cdot 8 = 8192$. In addition, meta-data consisting of a 1-bit valid flag and 22-bit tag is required for each line; this totals $l \cdot (1 + 22) = 128 \cdot 23 = 2944$ bits. Overall, this means 11136 bits and hence the same number of SRAM cells.

ii This is an open-ended question, but two obvious example policies are as follows:

i. A basic policy would be to set the control-signal for each cell in a given line equal to the associated valid flag: if the line is valid the cells operate as normal, but are in low-power mode otherwise. Put another way, if the line contains no valid data no power is wasted storing it.
ii. A more advanced idea would be to maintain a counter for each line; this would be analogous to the Least Recently Used (LRU) counter maintained by a set-associative cache. Once the counter associated with a given line reaches a threshold, the cells are turned into low-power mode: the intuition is that if the content in a line has not been accessed for a long time, there is less chance it will be accessed in the future and hence we can save power by evicting the content. The trade-off is that by doing so, a subsequent access that would have been cache-hit may now be a cache-miss.

Part XV: Performance measurement

▷ **S242.** a MIPS stands for Millions of Instructions Per Second; it is calculated by measuring how many instructions are executed in a one second period. The advantages of this are that it is easy to calculate but is is not a good measure of performance since it ignores how much work gets done during execution: a processor executing many simple instructions per second might do exactly the same amount of work as another processor which executes fewer more complex instructions in the same time period. CPI stands for Cycles Per Instruction. The idea is to decompose the performance in to the product of how long each instruction type takes to execute and how often that instruction type is executed. Typically, a RISC processor will have a low CPI rating whereas a CISC processor will have a high CPI rating. The advantages are that we can now factor in how much work is being done by each instruction and ignore clock rate, the disadvantage is that it relies on an average program which clearly never exists in real-life.

b We know that the overall CPI can be computed using

$$CPI = \sum_{t \in T} \mathcal{F}_t \cdot \mathcal{L}_t$$

where $\mathcal{F}_t$ and $\mathcal{L}_t$ denote the frequency of execution and latency of instruction type t respectively, and here we have that $T = \{\text{arithmetic}, \text{branch}, \text{load}, \text{store}\}$. So applying this formula to the initial numbers, we find:

Arithmetic	$\mathcal{F} \cdot \mathcal{L}$	=	$0.5 \cdot 1$	=	0.5
Branch	$\mathcal{F} \cdot \mathcal{L}$	=	$0.2 \cdot 2$	=	0.4
Load	$\mathcal{F} \cdot \mathcal{L}$	=	$0.2 \cdot 5$	=	1.0
Store	$\mathcal{F} \cdot \mathcal{L}$	=	$0.1 \cdot 3$	=	0.3
Total					2.2

Using the same method on the numbers after the change, we find:

Arithmetic	$\mathcal{F} \cdot \mathcal{L}$	=	$0.55 \cdot 1$	=	0.55
Branch	$\mathcal{F} \cdot \mathcal{L}$	=	$0.25 \cdot 2$	=	0.50
Load	$\mathcal{F} \cdot \mathcal{L}$	=	$0.10 \cdot 5$	=	0.50
Store	$\mathcal{F} \cdot \mathcal{L}$	=	$0.10 \cdot 3$	=	0.30
Total					1.85

The speed-up can be calculated as the ration of the new and old performance: we have that the new processor is 1.19 times faster than the old one, i.e., about a 16% improvement.

▷ **S243.**
- Firstly, it is sometimes difficult to know what one should be measuring. Latency (i.e., execution time) is often the most important factor, but throughput for example is also a valid performance metrics. Without some idea of what the interesting feature of performance is, constructing a benchmark program to measure it is difficult.
- Next it is difficult, if not impossible, to find a suite of benchmark programs that perfectly represent "typical" workloads. Within a restricted domain (e.g., scientific or media kernels) this is more tractable, but generally one can always point at examples that are somehow atypical and hence which are not modelled by the benchmark.
- Even once the benchmark programs themselves are selected, the problem of input data is important. Processor features such as branch prediction and cache memories have data-dependent behaviour; this implies that benchmarking them with worst-case input will yield much different behaviour than best-case.
- Finally, it can often be difficult to actually perform measurements if a high degree of accuracy is required; often this requires support from the processor (e.g., performance counters) and long experimental trials to smooth over any local anomalies.

Following the description above, one might clearly "cheat" by using input data which is best-case in some way and hence does not represent typical use of the benchmark. Another approach might be to develop a special-purpose compiler that behaves in some special way for the benchmark program. A compiler is supposed to be general-purpose, but if it can make assumptions about the input (i.e., that the input is a particular benchmark program), it can potentially generate much more efficient output than one might expect.

Part XVI: Techniques for efficient implementation

▷ **S244.** a A basic approach would be to use c as an index into an array which we initialise with the right value. For example, the C function would be

```
uint16_t choose( bool c, uint16_t y, uint16_t z ) {
  uint16_t T[ 2 ] = { z, y }

  return T[ c ];
}
```

Depending on the characteristics of memory access, this might be slow however: although it is straight-line and hence does not suffer from pipeline stalls from branch instructions, it includes three memory accesses (depending on the addressing modes in the processor. An improvement might be constructed by first considering a mask value $m = c - 1$ so that if $c = 0$ we have that m=-1 and if $c = 1$ then $m = 0$.

Because of the way the two's-complement representation works, if $c = 0$ then m is a value where all the bits are set to 1 and if $c = 0$ then m is a value where all the bits are set to 0. We can use this mask value to select the right result from y and z using the logical expression

$$x = (m \wedge z) \vee (\neg m \wedge y).$$

That is, if $c = 0$ then m is a value with all bits set to 1 and we have that $m \wedge z = z$ and $\neg m \wedge y = 0$ so $(m \wedge z) \vee (\neg m \wedge y) = z$. Conversely, if $c = 1$ then m is a value with all bits set to 0 and we have that $m \wedge z = 0$ and $\neg m \wedge y = y$ so $(m \wedge z) \vee (\neg m \wedge y) = y$. So we can implement the function in C as

```
uint16_t choose( bool c, uint16_t y, uint16_t z ) {
  uint16_t m = ( uint16_t )( c ) - 1;

  return ( m & z ) | ( ~m & y );
}
```

This is both straight-line and memory access free; it can be executing using only five instructions.

b A basic approach might be to find an addition chain for 15 so that we can decompose the multiplication into a series of additions. By forcing summands to be powers-of-two (which can be efficiently generated using shifts), we can implement the operation as

```
uint16_t mul( uint16_t x ) {
  return ( x << 3 ) +
         ( x << 2 ) +
         ( x << 1 ) +
         ( x      ) ;
}
```

such that the return value is

$$\mathtt{x} \cdot 2^3 + \mathtt{x} \cdot 2^2 + \mathtt{x} \cdot 2^1 + \mathtt{x} \cdot 2^0 = \mathtt{x} \cdot 8 + \mathtt{x} \cdot 4 + \mathtt{x} \cdot 2 + \mathtt{x} \cdot 1 = \mathtt{x} \cdot 15$$

However, a more efficient addition chain can be constructed by noticing $15 = 16 - 1$; since we can generate $\mathtt{x} \cdot 16$ efficiently, we can rewrite our solution as

```
uint16_t mul( uint16_t x ) {
  return ( x << 4 ) -
         ( x      ) ;
}
```

which uses two less shifts and two less additions.

c The basic approach here is to iterate through all the bits in x, incrementing a counter if for each one that we find is equal to 1. For example, the C function would be

```
int H( uint16_t x ) {
  int t = 0;

  for( int i = 0; i < 16; i++ ) {
    if( ( x >> i ) & 1 ) {
      t = t + 1;
    }
  }

  return t;
}
```

However, this is inefficient for a number of reasons. Specifically, the cost of operating the loop is high in comparison with the cost of the loop body: the overhead is significant. Furthermore, the instruction sequence required will include several branches which might stall the processor pipeline and degrade performance. A better approach is the straight-line C function

```
int H( uint16_t x ) {
  x = ( x & 0x5555 ) + ( ( x >> 1 ) & 0x5555 );
  x = ( x & 0x3333 ) + ( ( x >> 2 ) & 0x3333 );
  x = ( x & 0x0F0F ) + ( ( x >> 4 ) & 0x0F0F );
  x = ( x & 0x00FF ) + ( ( x >> 8 ) & 0x00FF );

  return ( int )( x );
}
```

This works using a divide-and-conquer approach; each line of the function is counting bits within a subset of the original value. We mask x with the constant $5555_{(16)}$ which extracts all the low bits in 2-bit regions within x; we also shift x right by one and again mask to extract all the corresponding high bits. These values are added to count the number of bits within the 2-bit regions. We divide-and-conquer again by considering 4-bit regions and so on until we end up with a total for the full 16-bit region.

d Perhaps the easiest example to understand is the population count function. Since the input x is only 16-bits, there are only 2^{16} potential values the caller can use for x. Thus, we can easily compute HW(x) for every one of these inputs, store them in a table and then look them up rather than running the original function. For example, the C functions could be

```
int* T = NULL;

void precomp() {
  T = ( uint16_t* )( malloc( 65536 * sizeof( int ) ) );

  for( int i = 0; i < 65536; i++ ) {
    uint16_t x = i;

    x = ( x & 0x5555 ) + ( ( x >> 1 ) & 0x5555 );
    x = ( x & 0x3333 ) + ( ( x >> 2 ) & 0x3333 );
    x = ( x & 0x0F0F ) + ( ( x >> 4 ) & 0x0F0F );
    x = ( x & 0x00FF ) + ( ( x >> 8 ) & 0x00FF );

    T[ i ] = ( int )( x );
  }
}

int H( uint16_t x ) {
  return T[ x ];
}
```

In the first function, we run through all possible values of x and store the results HW(x) in the table T. Then, in the actual function for HW(x) we look the value up from the table. Clearly this only produces an improvement if we use the function very often otherwise we have wasted time computing all the possible results; additionally we have made a trade-off in terms of time and space. That is, our pre-computation requires some memory to hold T so although it takes less time per-call to HW(x), it requires more space than the initial version.

▷ **S245.** Obviously combinations of the optimisation techniques can be applied, here we simply treat each one separately.

Offline pre-computation means doing some one-off computation before the function is called so that we can accelerate execution when it is called; since the pre-computation is offline in this case, it should be possible at compile-time i.e., before the program is even execute. In this context, we can pre-compute a table, say T, where each T[i] is the result of applying the function to i. This is possible since elements of the input A are only 16-bits in size: there are only $2^{16} = 65536$ 16-bit entries in the table so although this consumes some memory, it is not a lot of memory. Then, when the function is called we simply look-up T[i] rather than doing any actual computation:

```
uint16_t* T = NULL;

void precomp() {
  T = ( uint16_t* )( malloc( 65536 * sizeof( uint16_t ) ) );

  for( int i = 0; i < 65536; i++ ) {
    uint8_t l = ( i      ) & 0xFF;
    uint8_t h = ( i >> 8 ) & 0xFF;

    l = l / 2;
    h = h / 2;

    T[ i ] = ( ( uint16_t )( l )      ) |
             ( ( uint16_t )( h ) << 8 ) ;
  }
}

void packed_div2( uint16_t* A ) {
  for( int i = 0; i < 3; i++ ) {
    A[ i ] = T[ A[ i ] ];
  }
}
```

Note that rather than running the pre-computation function at run-time (i.e., when the program is loaded), we could create the table at compile-time and simply load it as a constant array or even embed it in the program source code.

Roughly speaking, specialisation is the act of replacing general-purpose operations with special-purpose operations that are less expensive. In this function, one example that occurs is division by the constant two; using a general-purpose division is expensive (division takes longer than most operations on a typical processor). However, division by 2^k is equivalent to a right-shift by k bits which is less expensive. We can therefore re-write the function:

```
void packed_div2( uint16_t* A ) {
  for( int i = 0; i < 3; i++ ) {
    uint8_t l = ( A[ i ]      ) & 0xFF;
    uint8_t h = ( A[ i ] >> 8 ) & 0xFF;

    l = l >> 1;
    h = h >> 1;

    A[ i ] = ( ( uint16_t )( l )      ) |
             ( ( uint16_t )( h ) << 8 ) ;
  }
}
```

Note that in this case, we could also include the right-shift of h with the shift required to mask the value in the first place; this saves another operation. Also, it is possible to avoid unpacking and repacking all together by performing the shift in one go and using a mask to patch up the result:

```
void packed_div2( uint16_t* A ) {
  for( int i = 0; i < 3; i++ ) {
    A[ i ] = ( A[ i ] >> 1 ) & 0x7F7F;
  }
}
```

Optimisation using parallelism is slightly tricky to write down in C, but roughly speaking you have to imagine that the host processor is equipped with the facility for SIMD parallelism: it can execute the same instruction on many data items at once (where the data items are packed into one blob). So if one can perform two divisions at once for example, one can clearly perform the divisions of l and h at the same time and get a speed-up of roughly two for this fragment of the function. This could be thought of as similar to this function:

```
void packed_div2( uint16_t* A ) {
  for( int i = 0; i < 3; i++ ) {
    uint8_t l = ( A[ i ]      ) & 0xFF;
    uint8_t h = ( A[ i ] >> 8 ) & 0xFF;

    par {
      l = l / 2;
      h = h / 2;
    }

    A[ i ] = ( ( uint16_t )( l )      ) |
             ( ( uint16_t )( h ) << 8 ) ;
  }
}
```

where the par keyword is denoting parallelism. For more able processors (e.g., multi-core), one might actually be able to run the (independent) iterations of the loop in parallel with each other.

Finally, program restructuring roughly means re-writing the program control-flow so that some advantage is gained: optimisations of this type include loop unrolling, loop fusion, loop fission and so on. Loop unrolling is a good candidate in this case since we are paying a high price for executing loop control (such as the test, the counter increment and so on) even though we know there will only ever be three iterations. Therefore, we can eliminate this overhead and re-write the function as:

```
void packed_div2( uint16_t* A ) {
  uint8_t l = ( A[ 0 ]      ) & 0xFF;
  uint8_t h = ( A[ 0 ] >> 8 ) & 0xFF;

          l = l / 2;
          h = h / 2;

  A[ 0 ] = ( ( uint16_t )( l )      ) |
           ( ( uint16_t )( h ) << 8 ) ;

          l = ( A[ 1 ]      ) & 0xFF;
          h = ( A[ 1 ] >> 8 ) & 0xFF;

          l = l / 2;
          h = h / 2;

  A[ 1 ] = ( ( uint16_t )( l )      ) |
           ( ( uint16_t )( h ) << 8 ) ;

          l = ( A[ 2 ]      ) & 0xFF;
          h = ( A[ 2 ] >> 8 ) & 0xFF;

          l = l / 2;
          h = h / 2;
  A[ 2 ] = ( ( uint16_t )( l )      ) |
           ( ( uint16_t )( h ) << 8 ) ;
}
```

which looks longer but potentially executes faster.

▷ **S246.** a In the fragment, i is the induction variable (i.e., the loop counter) and n is the loop bound (i.e., the number of times the loop body is executed). If we can determine n statically, we can replace the loop with n copies of the loop body where in the j-th copy we replace the induction variable n with the constant j. This is termed full loop unrolling; for example with n = 4 we would unroll the loop to get

```
A[ 0 ] = B[ 0 ] + C[ 0 ];
A[ 1 ] = B[ 1 ] + C[ 1 ];
A[ 2 ] = B[ 2 ] + C[ 2 ];
A[ 3 ] = B[ 3 ] + C[ 3 ];
```

The unrolled alternative might increase performance because:

i The overhead of operating the loop (i.e., initialisation, test and update of the induction variable) is eliminated; essentially the unrolled loop requires less instructions to be executed.

ii At least one branch per-iteration is required to direct control-flow from the bottom of the loop body back to the top again. This is overhead as above, but potentially also causes a problem in pipelined processors if we cannot accurately predict the branch direction and successfully use speculative execution. By eliminating the branch, this problem is also eliminated and the straight-line unrolled loop might execute faster as a result.

However, it might decrease performance because:

i Depending on n, the number of instructions required to implement the loop will typically be smaller than the number required to implement the unrolled alternative (although the loop might still execute more instructions at run-time). As a result instructions in the unrolled loop are not as spatially local as in the loop: a significant distance might exist between the first and last instructions for example. This means an instruction cache might be less effective. Furthermore, it might be hard to keep the unrolled loop in the cache since there is more chance of interference. Both features mean there is a chance that performance could be decreased.

b Continuing from the solution above, if we do not know n or it is too large to permit full unrolling we might partially unroll the loop; this means that we reduce the number of iterations but do not totally eliminate the loop. For example, imagine n = 400: we probably would not want to fully unroll the loop in this case but could instead partly unroll by a factor of say four to get:

```
for( int i = 0; i < 400; i += 4 ) {
  A[ i+0 ] = B[ i+0 ] + C[ i+0 ];
  A[ i+1 ] = B[ i+1 ] + C[ i+1 ];
  A[ i+2 ] = B[ i+2 ] + C[ i+2 ];
  A[ i+3 ] = B[ i+3 ] + C[ i+3 ];
}
```

Now consider the case of n = 402. If we use the same method as above, we end up with two extra iterations that we cannot accommodate in the main loop since the unrolling factor (of four) does not exactly divide n. We need to add a loop epilogue after the main loop to cope with the cases of i = 400 and i = 401:

```
for( int i = 0; i < 400; i += 4 ) {
  A[ i+0 ] = B[ i+0 ] + C[ i+0 ];
  A[ i+1 ] = B[ i+1 ] + C[ i+1 ];
  A[ i+2 ] = B[ i+2 ] + C[ i+2 ];
  A[ i+3 ] = B[ i+3 ] + C[ i+3 ];
}

A[ 400 ] = B[ 400 ] + C[ 400 ];
A[ 401 ] = B[ 401 ] + C[ 401 ];
```

or alternatively we could add a loop prologue before the main loop:

```
A[ 0 ] = B[ 0 ] + C[ 0 ];
A[ 1 ] = B[ 1 ] + C[ 1 ];

for( int i = 2; i < 402; i += 4 ) {
  A[ i+0 ] = B[ i+0 ] + C[ i+0 ];
  A[ i+1 ] = B[ i+1 ] + C[ i+1 ];
  A[ i+2 ] = B[ i+2 ] + C[ i+2 ];
  A[ i+3 ] = B[ i+3 ] + C[ i+3 ];
}
```