COMS10015 lab. worksheet #9

Although some questions have a written solution below, for others it will be more useful to experiment in a hands-on manner (e.g., using a concrete implementation). The file

https://assets.phoo.org/COMS10015_2025_TB-4/csdsp/sheet/lab-09_s.tar.gz

supports such cases.

§1. C-class, or core questions

▷ **S1[C].** An associated solution, i.e., a (documented) implementation, for this question can be found in the archive provided.

▷ **S2[C].** An associated solution, i.e., a (documented) implementation, for this question can be found in the archive provided. However, understanding that solution demands explaination of the underlying strategy involved:

a Per the description in the lecture slot(s), the combinatorial design is relatively simple: it represents a trade-off that means higher area but lower latency, and is (arguably) easier to implement and test. The implementation directly mirrors the design on paper, in the sense it consists of three layers:

- The left-hand layer is comprised of n groups of n AND gates: since the i-th such group computes $x_j \wedge y_i$ for $0 \leq j < n$, the output is either 0 if $y_i = 0$ or x if $y_i = 1$.
- The middle layer is comprised of n left-shift components: since the i-th such component shifts by a distance of i bits, the output is either 0 if $y_i = 0$, or $x \cdot 2^i$ if $y_i = 1$.
- The right-hand layer is a balanced, binary tree of adder components: since these accumulate the partial products resulting from the middle layer, the output is

$$r = \sum_{i=0}^{n-1} y_i \cdot x \cdot 2^i = y \cdot x$$

as required.

Imagine we want to compute $r = x \cdot y$ for $x = 8_{(10)}$ and $y = 14_{(10)} = 00001110_{(2)}$. Abusing notation a little with respect to AND, we find that the first two layers combine to compute

$$\begin{array}{lclcl}
p_0 & = & (x \wedge y_0) \ll 0 & = & (00001000_{(2)} \wedge 00000000_{(2)}) \ll 0 & = & 00000000_{(2)} \\
p_1 & = & (x \wedge y_1) \ll 1 & = & (00001000_{(2)} \wedge 11111111_{(2)}) \ll 1 & = & 00010000_{(2)} \\
p_2 & = & (x \wedge y_2) \ll 2 & = & (00001000_{(2)} \wedge 11111111_{(2)}) \ll 2 & = & 00100000_{(2)} \\
p_3 & = & (x \wedge y_3) \ll 3 & = & (00001000_{(2)} \wedge 11111111_{(2)}) \ll 3 & = & 01000000_{(2)} \\
p_4 & = & (x \wedge y_4) \ll 4 & = & (00001000_{(2)} \wedge 00000000_{(2)}) \ll 4 & = & 00000000_{(2)} \\
p_5 & = & (x \wedge y_5) \ll 5 & = & (00001000_{(2)} \wedge 00000000_{(2)}) \ll 5 & = & 00000000_{(2)} \\
p_6 & = & (x \wedge y_6) \ll 6 & = & (00001000_{(2)} \wedge 00000000_{(2)}) \ll 6 & = & 00000000_{(2)} \\
p_7 & = & (x \wedge y_7) \ll 7 & = & (00001000_{(2)} \wedge 00000000_{(2)}) \ll 7 & = & 00000000_{(2)}
\end{array}$$

after which the final layer computes

$$\begin{array}{lcl}
r & = & ((p_0 + p_1) + (p_2 + p_3)) + \\
  &   & ((p_4 + p_5) + (p_6 + p_7)) \\
  & = & ((00000000_{(2)} + 00010000_{(2)}) + (00000000_{(2)} + 00000000_{(2)})) + \\
  &   & ((00000000_{(2)} + 00000000_{(2)}) + (00000000_{(2)} + 00000000_{(2)})) \\
  & = & ((0_{(10)} + 16_{(10)}) + (32_{(10)} + 64_{(10)})) + \\
  &   & ((0_{(10)} + 0_{(10)}) + (0_{(10)} + 0_{(10)})) \\
  & = & 112_{(10)}
\end{array}$$

with parentheses denoting the adder instances.

b Per the description in the lecture slot(s), the sequential, iterative design is relatively complex: it represents a trade-off that means lower area but higher latency, and is (arguably) harder to implement and test.

The implementation is of Algorithm 1, which captures the left-to-right, bit-serial multiplication approach; recall that it processes y the from most- to least-significant bit. By design, the loop counter component from the

```
Input: Two unsigned, $n$-bit, base-2 integers $x$ and $y$
Output: An unsigned, $2n$-bit, base-2 integer $r = y \cdot x$
r ← 0
for $i = n - 1$ downto 0 step −1 do
    $r \leftarrow 2 \cdot r$
    if $y_i = 1$ then
        $r \leftarrow r + x$
    end
end
return $r$
```

Algorithm 1: An algorithm for left-to-right, bit-serial integer multiplication.

previous question can control the loop. Since the loop iterates downward in this case, it may *seem* we need to alter the counter to suit. Although it *is* possible to do so, the only use of i in the i-th iteration is to inspect the y_i: provided we inspect the correct y_i in each iteration, and perform n iterations overall, we can actually leave the counter as it is therefore. Instead of inspecting y_i based on i, we shift the i-th bit of y into a fixed index instead. That is, we update y via $y' \leftarrow y \ll 1$, i.e., left-shift it, after each iteration; this allows us to inspect the 7-th bit of y in *every* iteration, because in each successive i-th iteration it will hold the i-th bit. So, overall we need a data-path that includes

- combinatorial logic to compute
  - $r' \leftarrow 2 \cdot r + y_i \cdot x$, and
  - $y' \leftarrow y \ll 1$,

  and
- a pair of input and output registers to store t and y, plus an input register to store x (which is not updated, so requires no output register).

in addition to the counter. Crucially, as discussed in the lecture slot(s),

- $2 \cdot r$ can be realised using a left-shift (i.e., $2 \cdot r \equiv r \ll 1$) and
- $y_i \cdot x$ can be realised using a multiplexer (since $y_i \in \{0, 1\}$, $y_i \cdot x$ evaluates to 0 or x).

Having first implemented said data-path and then connected it to the control-path (i.e., the counter), imagine we want to compute $r = x \cdot y$ for $x = 8_{(10)}$ and $y = 14_{(10)} = 00001110_{(2)}$. We initiate the computation by first setting x and y, then $req = 1$ to signal a request. The multiplier steps through the following

i	r	y	y_7	r'	y'	
	0					
0	0	$00001110_{(2)}$	0	0	$0001110_{(2)}$	$r' \leftarrow 2 \cdot r$
1	0	$00011100_{(2)}$	0	0	$0011100_{(2)}$	$r' \leftarrow 2 \cdot r$
2	0	$00111000_{(2)}$	0	0	$0111000_{(2)}$	$r' \leftarrow 2 \cdot r$
3	0	$01110000_{(2)}$	0	0	$1110000_{(2)}$	$r' \leftarrow 2 \cdot r$
4	0	$11100000_{(2)}$	1	8	$1100000_{(2)}$	$r' \leftarrow 2 \cdot r + x$
5	8	$11000000_{(2)}$	1	24	$1000000_{(2)}$	$r' \leftarrow 2 \cdot r + x$
6	24	$10000000_{(2)}$	1	56	$0000000_{(2)}$	$r' \leftarrow 2 \cdot r + x$
7	56	$00000000_{(2)}$	0	112	$0000000_{(2)}$	$r' \leftarrow 2 \cdot r$
	112					

at which point it sets $ack = 1$ to signal computation is complete; we have the result $r = 112_{(10)}$, so can set $req = 1$ at which point the multiplier sets $ack = 0$ and is ready for any subsequent computation.

§2. R-class, or revision questions

▷ **S3[R].** There is a set of solutions available at

https://assets.phoo.org/COMS10015_2025_TB-4/csdsp/sheet/misc-revision_s.pdf

§3. A-class, or additional questions

▷ **S4[A].** The archive provided includes a LogisimEvo implementation of a “limited” (versus real) HP-35, focused more on what *could* be done (to help you learn about a given topic) than on what *should* be done: within what is a large design space, the implementation represents *a* solution rather than *the* solution. Along those lines, we compromise via simplifications that include

- operating in binary, rather than Binary Coded Decimal (BCD),
- using small 8-bit integers, rather than 14-word, 56-bit BCD-based floating-point,
- limiting the control and arithmetic keys available, and therefore the types of operation possible, and
- avoiding the issue of micro-code, and hence arithmetic operations (e.g., sin and cos) implemented using an algorithm rather than a circuit.

So, the real HP-35 clearly offers more complex functionality supported by a more complex internal design. However, the limited implementation satisfies the goal of demonstrating we can design and implement a device capable of recognisable, useful forms of computation. Although a gap clearly remains between it and a modern micro-processor, for example, conceptual similarities are also evident; understanding of the former clearly provides a good stepping stone toward understanding of the latter, therefore.

A limited HP-35: design

Figure 1 presents some (internal and external) photographs of a real HP-35. From the perspective of a user, operation of the HP-35 is precísed by Figure 1b: the rear casing essentially provides an instruction manual, or more formally a set of semantics for each key press, which dictates the internal design and implementation to some extent. For example, it is clear the HP-35 maintains four internal registers (or accumulators) named X, Y, Z and T, plus a fifth storage register S which we might colloquially term "memory"; the value of X is displayed on the LED display. Said registers are manipulated by pressing the control and arithmetic keys, each of which invokes a specific operation. Using R' to denote the next value of some register $R \in \{X, Y, Z, T, S\}$ for example, we can translate pertinent operations into

a 'V' for $V \in \{0, 1, \ldots 9\}$
   - $X' \leftarrow 10 \cdot X + V$

b '$\odot$' for $\odot \in \{+, -, \times\}$
   - $X' \leftarrow Y \odot X, Y' \leftarrow Z, Z' \leftarrow T, T' \leftarrow T$

c 'CLR' (or "clear")
   - $X' \leftarrow 0, Y' \leftarrow 0, Z' \leftarrow 0, T' \leftarrow 0$

d 'STO' (or "store")
   - $S' \leftarrow X$

e 'RCL' (or "recall")
   - $X' \leftarrow S$

f '$\uparrow$' (or "enter")
   - $X' \leftarrow X, Y' \leftarrow X, Z' \leftarrow Y, T' \leftarrow Z$

noting that we *only* support this sub-set in the limited HP-35; you can find a complete manual named `hp35-manual.pdf` in the archive provided.

Some of those semantics might seem counter-intuitive: why compute $Y \odot X$ rather than $X \odot Y$ for instance? In many cases, they are designed to support evaluation of expressions in Reverse Polish Notation (RPN) form. Considering the use of in-fix operators per

$$(19 - 5) \times (1 + 2),$$

RPN simply uses post-fix operators to specify the same result via

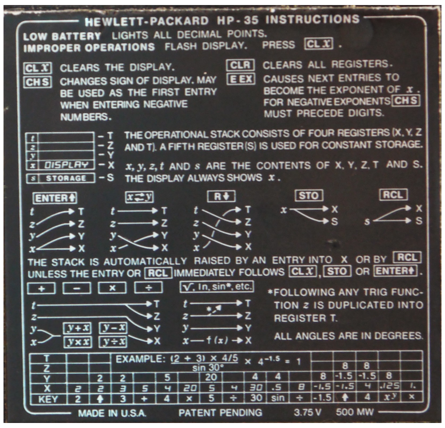
$$19 \;\; 5 \;\; - \;\; 1 \;\; 2 \;\; + \;\; \times.$$

This form is advantageous for a variety of reasons, including that fact that expressions can be specified unambiguously *without* adding parentheses. Even more useful, at least with respect to implementation, an expression can be evaluated using a **stack**: by reading an RPN expression left-to-right, the idea is that

- each time we read an operand we push it onto the stack, and
- each time we read an operator we pop operands from the stack, perform the associated operation then push a result onto the stack

Once this process is complete, the evaluated result is the single remaining entry on the stack. The HP-35 uses a

(a) *The calculator outer casing, keypad and LED display.*



HEWLETT-PACKARD HP-35 INSTRUCTIONS

LOW BATTERY LIGHTS ALL DECIMAL POINTS.
IMPROPER OPERATIONS FLASH DISPLAY. PRESS CL x.

CL x CLEARS THE DISPLAY.
CHS CHANGES SIGN OF DISPLAY. MAY BE USED AS THE FIRST ENTRY WHEN ENTERING NEGATIVE NUMBERS.

CLR CLEARS ALL REGISTERS.
E EX CAUSES NEXT ENTRIES TO BECOME THE EXPONENT OF x. FOR NEGATIVE EXPONENTS CHS MUST PRECEDE DIGITS.

t		– T
z		– Z
y		– Y
x	DISPLAY	– X
s	STORAGE	– S

THE OPERATIONAL STACK CONSISTS OF FOUR REGISTERS (X, Y, Z AND T). A FIFTH REGISTER (S) IS USED FOR CONSTANT STORAGE.

x, y, z, t AND s ARE THE CONTENTS OF X, Y, Z, T AND S. THE DISPLAY ALWAYS SHOWS x.

ENTER↑ x⇄y R↓ STO RCL

THE STACK IS AUTOMATICALLY RAISED BY AN ENTRY INTO X OR BY RCL UNLESS THE ENTRY OR RCL IMMEDIATELY FOLLOWS CL x, STO OR ENTER↑.

+ − × ÷ √, ln, sin*, etc.

y+x y−x y×x y÷x

$x — f(x) \rightarrow X$

*FOLLOWING ANY TRIG FUNCTION z IS DUPLICATED INTO REGISTER T.

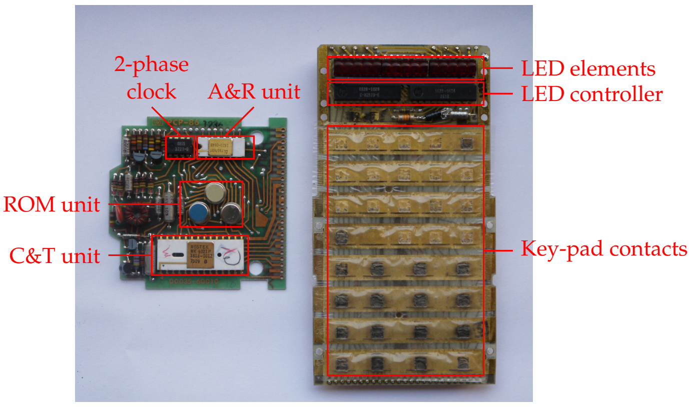
ALL ANGLES ARE IN DEGREES.

EXAMPLE: $\frac{(2+3) \times 4/5}{\sin 30^\circ} \times 4^{-1.5} = 1$

T																
Z													8	8		
Y		2	2		5		20		4	4		8	-1.5	-1.5	8	
X	2	2	3	5	4	20	5	4	30	.5	8	-1.5	-1.5	4	.125	1.
KEY	2	↑	3	+	4	×	5	÷	30	sin	÷	-1.5	↑	4	x^y	×

MADE IN U.S.A. PATENT PENDING 3.75 V 500 MW

(b) *An overview of semantics (or instructions) presented on a label attached to the rear of the casing.*



(c) *An annotated photograph of the two main internal HP-35 circuit boards.*

Figure 1: *Photographs of a real HP-35 calculator.*

(slight) variation, which implies the expression above would be evaluated using the following key presses

Register	Key-press										
		1	9	↑	5	−	1	↑	2	+	×
X	0	1	19	19	5	14	1	1	2	3	42
Y	0	0	0	19	19	0	14	1	1	14	0
Z	0	0	0	0	0	0	0	14	14	0	0
T	0	0	0	0	0	0	0	0	0	0	0

to yield the result $(19 - 5) \times (1 + 2) = 42$. The difference versus pure RPN is basically use of the '↑' key: this signals the end of a multi-digit operand, allowing a new operand to be entered into X. Either way, notice how X, Y, Z and T are used as an evaluation stack, growing downward as operands are pushed and upward as they are popped (and used by a given operation).

A limited HP-35: implementation

Figure 1c illustrates the major sub-components, namely

- a 2-phase clock generator,
- a Read Only Memory (ROM) unit,
- an Arithmetic and Register (A&R) unit,
- a Control and Timing (C&T) unit,
- a keypad to provide input, and
- an LED-based display to provide output,

which are spread over two Printed Circuit Boards (PCBs); roughly the same top-level organisation is reproduced by the LogisimEvo implementation.

- A keypad provides input: the state of each key forms 1 bit within a 32-bit keypad state pad via the following mapping

'0'	$\mapsto$	$00000_{(2)}$			
'1'	$\mapsto$	$00001_{(2)}$	'+'	$\mapsto$	$10000_{(2)}$
'2'	$\mapsto$	$00010_{(2)}$	'−'	$\mapsto$	$10001_{(2)}$
'3'	$\mapsto$	$00011_{(2)}$	'×'	$\mapsto$	$10010_{(2)}$
'4'	$\mapsto$	$00100_{(2)}$			
'5'	$\mapsto$	$00101_{(2)}$	'CLR'	$\mapsto$	$10100_{(2)}$
'6'	$\mapsto$	$00110_{(2)}$	'STO'	$\mapsto$	$10101_{(2)}$
'7'	$\mapsto$	$00111_{(2)}$	'RCL'	$\mapsto$	$10110_{(2)}$
'8'	$\mapsto$	$01000_{(2)}$	'↑'	$\mapsto$	$10111_{(2)}$
'9'	$\mapsto$	$01001_{(2)}$			

  The idea is that the key listed on the left-hand side is connected to the bit in pad which is listed on the right-hand side: the '9' key is connected to pad_9, 9-th bit of pad, for example. The right-hand side acts as a 5-bit key-code code for each key on the left-hand side. Some by-design features in the mapping make this key-code useful. Notice, for example that code_4 determine whether the key pressed is numeric; if it is, $\mathit{code}_{3...0}$ then gives the associated (unsigned, 4-bit) integer key-value.
- A set of output pins reflects the values held by registers X, Y, Z, T, and S. Although the real HP-35 displays the value of X only (using the LED display), including them all 1) makes the internal state clearer, meaning 2) it is easier to understand (and debug) internal behaviour.

The Control and Timing (C&T) unit

The C&T unit acts as the control-path: it accepts

- pad, a 32-bit keypad state,
- Φ_1 and Φ_2, two 1-bit, 2-phase clock signals, and
- rst, a 1-bit reset signal,

as input, and produces

- code, a 5-bit key-code,
- value, a 8-bit key-value,
- append and raise, two 1-bit control signals, and
- en_1 and en_2, two 1-bit enable signals,

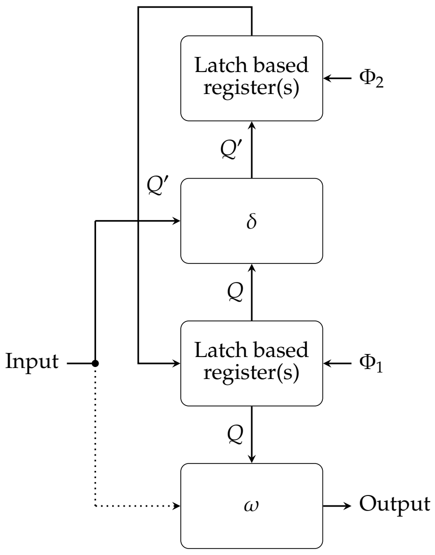
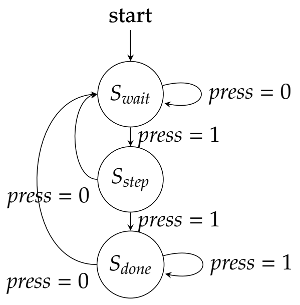


Figure 2: *A generic FSM framework (for a 2-phase clocking strategy) into which one can place implementations of the state, δ (the transition function) and ω (the output function).*

	δ		ω	
Q	Q'			
	$press = 0$	$press = 1$	en_2	en_1
S_{wait}	S_{wait}	S_{step}	0	0
S_{step}	S_{wait}	S_{done}	1	0
S_{done}	S_{wait}	S_{done}	0	1

(a) *A tabular description.*



(b) *A diagrammatic description.*

Figure 3: *An overview of components in the C&T unit.*

as output. There are two main (sub-)components within the unit:

- The keypad state $keypad$ is used by a priority encoder which is tasked with translating a press of the i-th key, meaning $keypad_i = 1$ and $keypad_j = 0$ for $i \neq j$, into the key-code $code$ and key value $value$; in doing so it produces

  $$press = \bigvee_{i=0}^{i<32} keypad_i$$

  which is 1 iff. any key is currently being pressed.

  Note that both the current and the previous key-codes are stored, using two 5-bit registers; the former is enabled by $press$, meaning 1) it is latched automatically as soon as a key is pressed, and 2) will remain valid even if that key is unpressed while being processed. The purpose of storing both is to deal with the "stack is automatically raised" aspect of the semantics listed in Figure 1b. Given the limited set of operations available (and hence control and arithmetic keys), we simplify the real semantics as follows:

  – The $raise$ signal determines whether we need to raise the stack (which is like a push operation); $raise = 1$ iff. the previous key pressed was a control or arithmetic key other than 'STO' or '↑'.

  – The $append$ signal determines whether we are forming a multi-digit value in X, appending a digit specified by the current key-press; $append = 1$ iff. the previous key pressed was a numeric key.

  Both signals are produced by a decoder using the previous key-code as input.

- An FSM is tasked with managing step-by-step processing of key presses. As always, Figure 2 shows the design at a high-level; our goal is basically to instantiate the constituent blocks to suit the problem at hand.

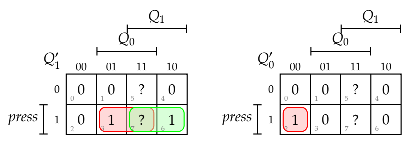
  The description in Figure 3 should read fairly intuitively for δ: we start by waiting until a key press is available, then take action to process it, before again waiting for the key to then be *un*pressed (to avoid repeated processing if the key is held down). This is achieved using 3 states with the abstract labels S_{wait}, S_{step}, and S_{done}. Since $2^2 = 4 > 3$, we can represent both the current and next states as 2-bit integers, i.e., $Q = \langle Q_0, Q_1 \rangle$ and $Q' = \langle Q'_0, Q'_1 \rangle$ respectively, where

$$\begin{array}{lcl} S_{wait} & \mapsto & \langle 0,0 \rangle \\ S_{step} & \mapsto & \langle 0,1 \rangle \\ S_{done} & \mapsto & \langle 1,0 \rangle \end{array}$$

Expanding the description in Figure 3 into a concrete truth table

			δ		ω	
$press$	Q_1	Q_0	Q'_1	Q'_0	en_2	en_1
0	0	0	0	0	0	0
0	0	1	0	0	1	0
0	1	0	0	0	0	1
0	1	1	?	?	?	?
1	0	0	0	1	0	0
1	0	1	1	0	1	0
1	1	0	1	0	0	1
1	1	1	?	?	?	?

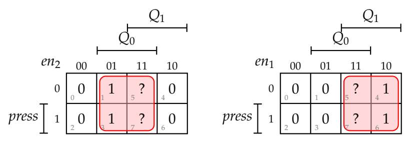
means we can form a set of Karnaugh maps



and hence write the following Boolean expressions

$$\begin{array}{rcl} Q'_1 & = & (\ press \wedge Q_0\ ) \vee \\ & & (\ press \wedge Q_1\ ) \\ \\ Q'_0 & = & (\ press \wedge \neg Q_1 \wedge \neg Q_0\ ) \end{array}$$

for Q'_1 and Q'_0.
In contrast, ω needs some explanation. The idea is that rather than using clock signals Φ_1 and Φ_2, en_1 and en_2 will explicitly control latches in the A&R unit. Put simply, when the FSM is in the S_{step} state, $en_2 = 1$ meaning the output latches in the A&R unit are enabled; this means they store the next value of each register (e.g., X). In the S_{done} state, however, $en_1 = 1$ meaning the input latches are enabled; this means the value stored in each output latch is fed back around, and stored in the associated input latch, ready to cope with the next key press. Generating en_2 and en_1 is simple: by using the same truth table above, we form



and can hence write

$$\begin{array}{rcl} en_2 & = & Q_0 \\ en_1 & = & Q_1 \end{array}$$

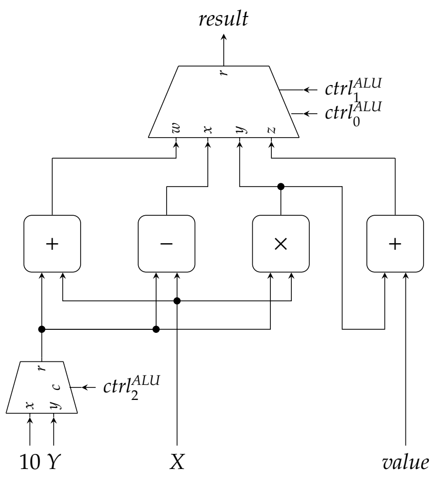
The Arithmetic and Register (A&R) unit

The A&R unit acts as the data-path: it accepts

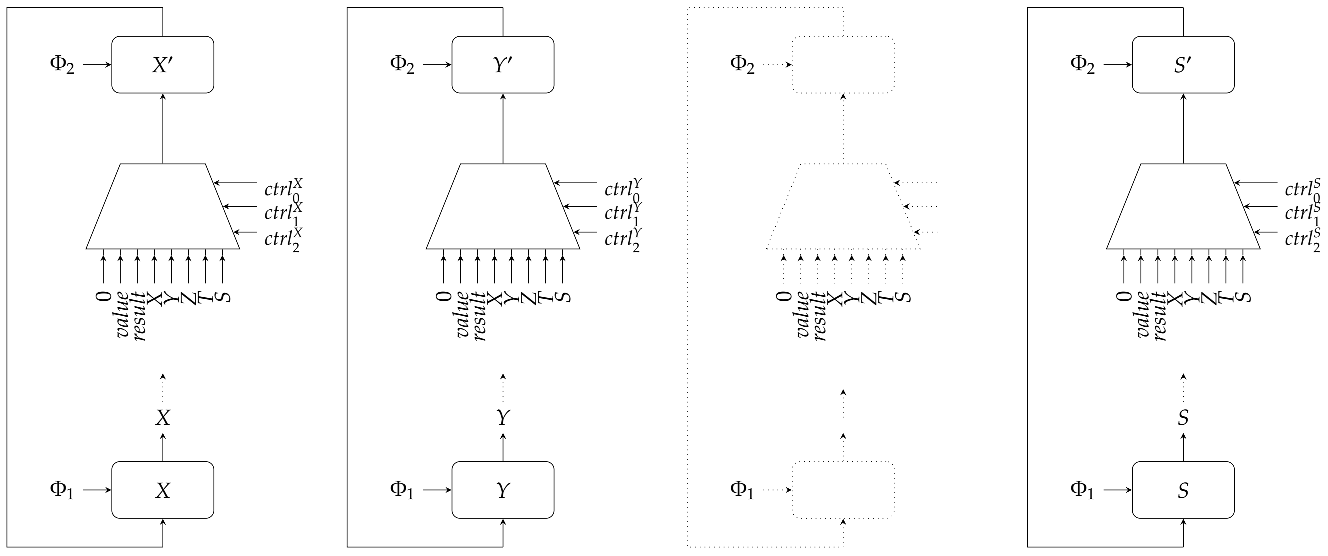
- $ctrl^X$, $ctrl^Y$, $ctrl^Z$, $ctrl^T$, and $ctrl^S$, five 3-bit control signals for the X, Y, Z, T and S multiplexers,
- $ctrl^{ALU}$, a 3-bit control signal for the ALU,
- en_1 and en_2, two 1-bit enable signals,
- $value$, the 8-bit current key-value,

as input, and produces

- X, Y, Z, T, and S, five 8-bit register values,



(a) *The Arithmetic and Logic Unit (ALU).*



(b) *The register file.*

Figure 4: *An overview of components in the A&R unit.*

as output. There are two main (sub-)components within the unit:

- A fairly simple Arithmetic and Logic Unit (ALU) is used to 1) compute any result associated with pressing an arithmetic key (e.g., add Y and X when '+' is pressed), and also 2) update the X register when a numeric key is pressed. Figure 4a illustrates the design, with the precise behaviour dictated both by the numerical inputs (namely the value of registers Y and X, plus key-value value) *and* the 3-bit control signal ctrl^{ALU}.
  There are four arithmetic components in the center of the ALU, each of whose inputs are controlled by ctrl^{ALU}_2 (via a multiplexer towards the left): specifically, we have that

  $$x = \begin{cases} 10 & \text{if } \mathit{ctrl}^{ALU}_2 = 0 \\ Y & \text{if } \mathit{ctrl}^{ALU}_2 = 1 \end{cases}$$

  $y = X$ and $z = \mathit{value}$. Each arithmetic component is combinatorial, so continuously computes an output from x, y and z. One output is selected as the ALU output result using $\mathit{ctrl}^{ALU}_{1\ldots0}$ (via a multiplexer towards the right); basically we get

  $$\mathit{result} = \begin{cases} x + y & \text{if } \mathit{ctrl}^{ALU}_{1\ldots0} = 00_{(2)} \\ x - y & \text{if } \mathit{ctrl}^{ALU}_{1\ldots0} = 01_{(2)} \\ x \cdot y & \text{if } \mathit{ctrl}^{ALU}_{1\ldots0} = 10_{(2)} \\ x \cdot y + z & \text{if } \mathit{ctrl}^{ALU}_{1\ldots0} = 11_{(2)} \end{cases}$$

  Putting everything together, we find that

  - $\mathit{ctrl}^{ALU} = 000_{(2)}$ means $\mathit{result} = 10 + X$,
  - $\mathit{ctrl}^{ALU} = 001_{(2)}$ means $\mathit{result} = 10 - X$,
  - $\mathit{ctrl}^{ALU} = 010_{(2)}$ means $\mathit{result} = 10 \cdot X$,
  - $\mathit{ctrl}^{ALU} = 011_{(2)}$ means $\mathit{result} = 10 \cdot X + \mathit{value}$,
  - $\mathit{ctrl}^{ALU} = 100_{(2)}$ means $\mathit{result} = Y + X$,
  - $\mathit{ctrl}^{ALU} = 101_{(2)}$ means $\mathit{result} = Y - X$,
  - $\mathit{ctrl}^{ALU} = 110_{(2)}$ means $\mathit{result} = Y \cdot X$, and
  - $\mathit{ctrl}^{ALU} = 111_{(2)}$ means $\mathit{result} = Y \cdot X + \mathit{value}$.

  Clearly only *some* of these combinations are useful (e.g., $\mathit{result} = 10 + X$ is not), but the point is that all operations required by our semantics can be computed: we just need to set ctrl^{ALU} appropriately.
- The state of the calculator, with respect to expression evaluation, is maintained by a set set of registers: recall that these are used as an evaluation stack to realise semantics outlined in the Section above. Although there is a simplified illustration in Figure 4b, the implementation itself may look complex. Keep in mind that it just replicates the same structure for each each register $R \in \{X, Y, Z, T, S\}$. Specifically,

  - there is an input latch R and an output latch R' which store the current and next values respectively,
  - the input latch is enabled by en_1, whereas the output latch is enabled by en_2; this means the output latch is enabled when the C&T unit FSM is in the S_{step} state, whereas the input latch is enabled in the S_{done} state, and
  - a multiplexer is used to select the next value based on the 3-bit control signal ctrl^R:

    * $\mathit{ctrl}^R = 000_{(2)}$ selects the constant 0,
    * $\mathit{ctrl}^R = 001_{(2)}$ selects the key-value value,
    * $\mathit{ctrl}^R = 010_{(2)}$ selects the ALU output result,
    * $\mathit{ctrl}^R = 011_{(2)}$ selects the value of X,
    * $\mathit{ctrl}^R = 100_{(2)}$ selects the value of Y,
    * $\mathit{ctrl}^R = 101_{(2)}$ selects the value of Z,
    * $\mathit{ctrl}^R = 110_{(2)}$ selects the value of T, and
    * $\mathit{ctrl}^R = 111_{(2)}$ selects the value of S.

  So to implement the original semantics, we just need to set ctrl^R to correctly select the next value for each register. Based on the the latched current key-code code, plus the raise and append control signals, we can be more exact about what this means:

C	$raise$	$append$	X'	Y'	Z'	T'	S'
'V' for $V \in \{$'0', '1', ... '9'$\}$	0	?		X	Y	Z	S
'V' for $V \in \{$'0', '1', ... '9'$\}$	1	?		Y	Z	T	S
'V' for $V \in \{$'0', '1', ... '9'$\}$	?	0	V				
'V' for $V \in \{$'0', '1', ... '9'$\}$	?	1	$10 \cdot X + V$				
'$\odot$' for $\odot \in \{$'+', $-$, $\times\}$	?	?	$Y \odot X$	Z	T	T	S
'CLR'	?	?	0	0	0	0	0
'STO'	?	?	X	Y	Z	T	X
'RCL'	?	?	S	Y	Z	T	S
'$\uparrow$'	?	?	X	X	Y	Z	S

So, for instance, if the '$\uparrow$' key is pressed we want $X' \leftarrow X$, $Y' \leftarrow X$, $Z' \leftarrow Y$, $T' \leftarrow Z$, and $S' \leftarrow S$; this means we must set $ctrl^X = 011_{(2)}$, $ctrl^Y = 011_{(2)}$, $ctrl^Z = 100_{(2)}$, $ctrl^T = 101_{(2)}$, and $ctrl^S = 111_{(2)}$ in order to select the next value correctly for each case.

The ROM unit

The ROM unit is tasked with generating control signals for the A&R unit: it accepts

- $code$, a 5-bit key-code,
- $append$ and $raise$, two 1-bit control signals,

as input, and produces

- $ctrl^X$, $ctrl^Y$, $ctrl^Z$, $ctrl^T$, and $ctrl^S$, five 3-bit control signals for the X, Y, Z, T and S multiplexers,
- $ctrl^{ALU}$, a 3-bit control signal for the ALU,

as output.

ROM as a look-up table for control signals: the basic concept

If you think about it, a ROM is just a (fixed) look-up table: by using an n'-bit address x, each load yields a w-bit word $r = \mathsf{MEM} = f(x)$ where the function f is determined by the ROM content. Put simply, one can encode *any* Boolean function

$$\mathbb{B}^{n'} \rightarrow \mathbb{B}^{w}$$

using the ROM. This idea is used to avoid having to design and implement complicated Boolean functions to control the A&R unit.

Consider the top-most ROM, for example, as used to produce the ALU control signal $ctrl^{ALU}$; it has a total of 32 words, each of 3 bits. The $n' = 5$ bit address used to access content is simply $x = code$, i.e., the current key-code, which yields a $w = 3$ bit word $ctrl^{ALU} = \mathsf{MEM}$. Given $ctrl^{ALU}_{2}$ and $ctrl^{ALU}_{1...0}$ are used to select the ALU input and operation respectively, we get the following behaviour

C	$c^{ALU} = \mathsf{MEM}$			
$00000_{(2)}$	$011_{(2)}$	$\rightsquigarrow$	$C =$ '0', $V = 0$,	ALU computes $10 \cdot X + 0$
$00001_{(2)}$	$011_{(2)}$	$\rightsquigarrow$	$C =$ '1', $V = 1$,	ALU computes $10 \cdot X + 1$
$00010_{(2)}$	$011_{(2)}$	$\rightsquigarrow$	$C =$ '2', $V = 2$,	ALU computes $10 \cdot X + 2$
$00011_{(2)}$	$011_{(2)}$	$\rightsquigarrow$	$C =$ '3', $V = 3$,	ALU computes $10 \cdot X + 3$
$00100_{(2)}$	$011_{(2)}$	$\rightsquigarrow$	$C =$ '4', $V = 4$,	ALU computes $10 \cdot X + 4$
$00101_{(2)}$	$011_{(2)}$	$\rightsquigarrow$	$C =$ '5', $V = 5$,	ALU computes $10 \cdot X + 5$
$00110_{(2)}$	$011_{(2)}$	$\rightsquigarrow$	$C =$ '6', $V = 6$,	ALU computes $10 \cdot X + 6$
$00111_{(2)}$	$011_{(2)}$	$\rightsquigarrow$	$C =$ '7', $V = 7$,	ALU computes $10 \cdot X + 7$
$01000_{(2)}$	$011_{(2)}$	$\rightsquigarrow$	$C =$ '8', $V = 8$,	ALU computes $10 \cdot X + 8$
$01001_{(2)}$	$011_{(2)}$	$\rightsquigarrow$	$C =$ '9', $V = 9$,	ALU computes $10 \cdot X + 9$
$\vdots$	$\vdots$			
$10000_{(2)}$	$100_{(2)}$	$\rightsquigarrow$	$C =$ '+',	ALU computes $Y + X$
$10001_{(2)}$	$101_{(2)}$	$\rightsquigarrow$	$C =$ '-',	ALU computes $Y - X$
$10010_{(2)}$	$110_{(2)}$	$\rightsquigarrow$	$C =$ '$\times$',	ALU computes $Y \cdot X$
$\vdots$	$\vdots$			

which is, of course, what we want. The point is that as long as the ROM is populated with the correct content, we can avoid use of combinatorial logic to *compute* $ctrl^{ALU}$ from $code$: we simply look it up in the ROM, which essentially provides a physical truth table.

The same strategy is used for the control signals associated with each of the registers, e.g., $ctrl^{X}$, housed in the A&R unit. In each case we also need to consider $raise$ and $append$, meaning each ROM has 128 words, each of 3 bits; we form an $n' = 7$ bit address as

$$x = raise \parallel append \parallel C$$

and look-up $ctrl^{X} = \mathsf{MEM}$ in the same way.

Easing the pain of ROM content specification

`convert.py`, found within the support archive, can be used to convert a human-readable ROM description (including don't-care states in either the input or output) into a form suitable for use by a LogisimEvo ROM component. Doing so by hand is tedious and error prone, so using the program makes the process a lot easier. As an example, and following the above, consider `hp35-rom_alu.txt` which is converted into `hp35-rom_alu.bin` and used to populate the ROM associated with ALU control signals. Each line starting with '#' is a comment, but beyond this:

- The first (non-comment) line states the number of input and output bits: here we have a 5-bit input and 3-bit output.
- Each subsequent line has three fields:
  - a label (which is otherwise unused),
  - an address (or input, i.e., x), and
  - a value (or output, i.e., $r = \mathsf{MEM}$).

  For example, the line

  ```
  ADD 10000 100
  ```

  implies that given $x = 10000_{(2)}$, $\mathsf{MEM} = 100_{(2)}$. Looking again at the Section above, this should make sense. The fact that

  $$x = C = 10000_{(2)}$$

  implies '+' is being pressed. To process this, we want the ALU to compute $Y + X$, i.e., $ctrl^{ALU}_{2}$ should be set to $1_{(2)}$ (to select Y as an input) and $ctrl^{ALU}_{1...0}$ should be set to $00_{(2)}$ (to select an addition operation); this is exactly what happens, because

  $$ctrl^{ALU} = \mathsf{MEM} = 100_{(2)}.$$