

COMS10015 lab. worksheet #4: sequential logic

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-04_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. Note that one can produce such a solution by taking content from the lecture slot(s), then translating (or “porting”) it into a LogisimEvo implementation: for reference, Figure 1 captures that content.
- ▷ **S2[C].** An associated solution, i.e., a (documented) implementation, for this question can be found in the archive provided. Unlike the previous question, however, we need to formulate a design before then implementing and simulating it: doing so involves consideration of two problems, or steps. First, how can we detect whether $x + y < 2^n$ or $x + y \geq 2^n$? Second, how can we use our detection mechanism to force $r = x + y$ or $r = 2^n - 1$ respectively?
 - The first problem could be solved using a general-purpose n -bit comparator; if we can perform a less-than comparison, i.e., $x + y < 2^n$, this is enough. The second problem could be solved using a general-purpose 2-way, n -bit multiplexer: in short, we use the output of the comparator as a control signal which means the multiplexer will select between $x + y$ and $2^n - 1$.
 - The first problem is easier than it sounds. A special-purpose solution is possible, because the carry-out co produced by the existing ripple-carry adder captures this information: if $co = 0$ then we know $x + y < 2^n$, whereas if $co = 1$ then we know $x + y \geq 2^n$. The second problem is *also* easier than it sounds. A special-purpose solution is possible, because the n -bit representation of $2^n - 1$ is such that $r_i = 1$ for $0 \leq i < n$. Denoting the ripple-carry adder output as r' , we can simply OR co with each r'_i therefore: this means if $co = 0$ then $r_i = co \vee r'_i = 0 \vee r'_i = r'_i$, whereas if $co = 1$ then $r_i = co \vee r'_i = 1 \vee r'_i = 1$.

There are other (somewhat) valid approaches beyond these, however. For example, instead of focusing on the middle block per se, one could instead attempt to disable the clock once $r = 2^n - 1$. Based on a similar solution for the first problem, doing so can be viewed as an alternative solution to the second problem, i.e., how to clamp the value: once the clock is disabled, the counter value and hence output will not be updated.

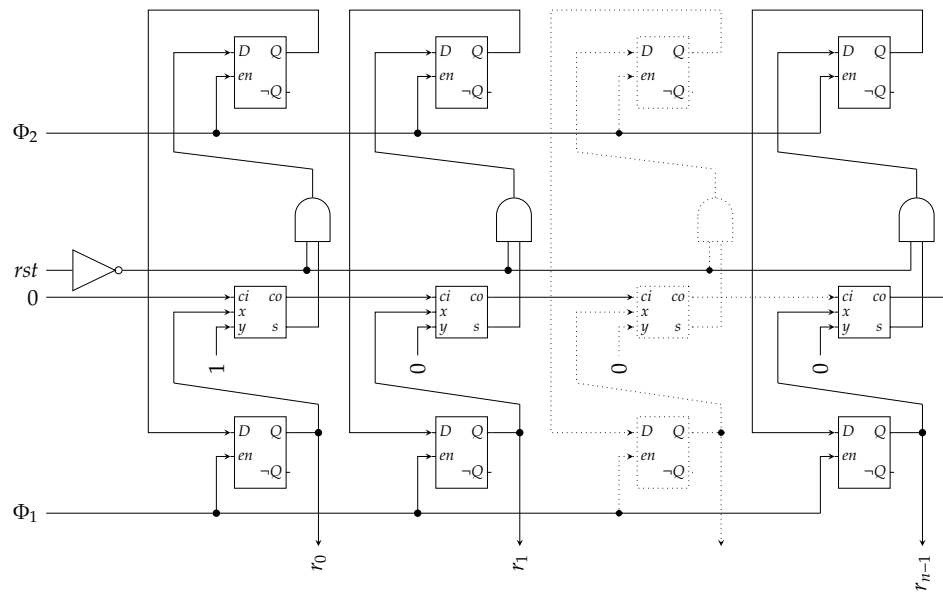
- ▷ **S3[C].** An associated solution, i.e., a (documented) implementation, for this question can be found in the archive provided. Making more overt use of built-in components (e.g., registers, multiplexers, and so on), it takes a simple but somewhat standard approach: it computes both $t_0 = x + y$ and $t_1 = x + y$, using a multiplexer controlled by *mode* to select the required output. One can view the altered middle block as a simple form of Arithmetic and Logic Unit (ALU)¹, more complex versions of which form the basis for computation (i.e., execution of instructions) within a micro-processor. Although *this* ALU only supports addition and subtraction, it should be obvious that the same structure can be extended to support a wider range of operations: as the acronym suggests, these might include both logical (e.g., bit-wise NOT, AND, and OR, shift and rotate, etc.) and arithmetic (e.g., multiplication, division, etc.) cases.

§2. R-class, or revision questions

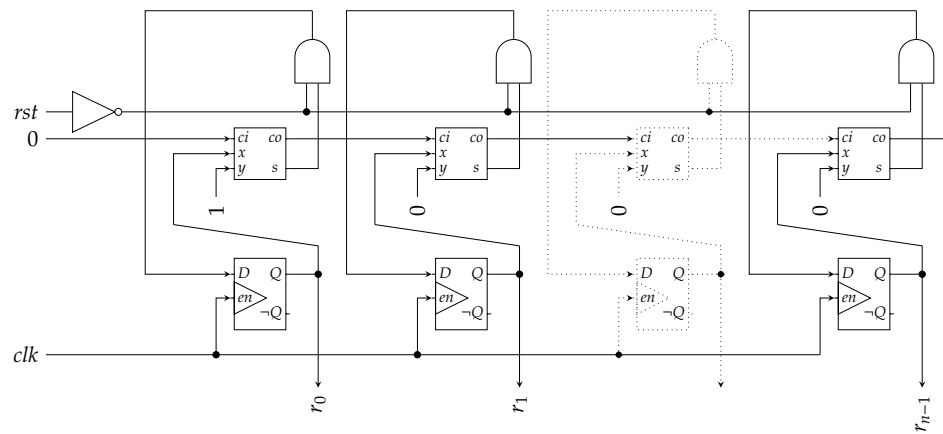
- ▷ **S4[R].** There is a set of solutions available at

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

¹https://en.wikipedia.org/wiki/Arithmetic_logic_unit



(a) A latch based design, using a 2-phase clock.



(b) A flip-flop based design, using a 1-phase clock.

Figure 1: A design for a cyclic n -bit counter.

§3. A-class, or additional questions

▷ **S5[A].** An associated solution, i.e., a (documented) implementation, for this question can be found in the archive provided. Before producing that implementation, however, we follow the same process as we did for the existing design:

- **Step #1: implementing an SR-type latch.** We already have the NOR-based version: using S' and R' to denote the internal latch inputs, given $Q = R' \bar{\vee} \neg Q$ and $\neg Q = S' \bar{\vee} Q$ we find that

- if $S' = 1, R' = 0$ then only $Q = 1, \neg Q = 0$ is valid,
- if $S' = 0, R' = 1$ then only $Q = 0, \neg Q = 1$ is valid,
- if $S' = 0, R' = 0$ then both $Q = 1, \neg Q = 0$ and $Q = 0, \neg Q = 1$ are valid (i.e., the latch is in storage mode),
- if $S' = 1, R' = 1$ then only $Q = 0, \neg Q = 0$ is valid (although not ideal since $\neg Q$ should be the inverse of Q).

With the NAND-based version, however, we need to produce a reasoned alternative. Although it seems a leap of faith, the natural question is whether we can simply replace the cross-coupled NOR gates with NAND gates. The short answer is yes, but we need to justify why and what caveats apply. Given $Q = R' \bar{\wedge} \neg Q$ and $\neg Q = S' \bar{\wedge} Q$ we find that

- if $S' = 1, R' = 0$ then only $Q = 1, \neg Q = 0$ is valid,
- if $S' = 0, R' = 1$ then only $Q = 0, \neg Q = 1$ is valid,
- if $S' = 1, R' = 1$ then both $Q = 1, \neg Q = 0$ and $Q = 0, \neg Q = 1$ are valid (i.e., the latch is in storage mode),
- if $S' = 0, R' = 0$ then only $Q = 1, \neg Q = 1$ is valid (although not ideal since $\neg Q$ should be the inverse of Q).

So the NAND-based version provides basically the same behaviour, but it differs in terms of the input required for storage mode: $S' = 0, R' = 0$ implies storage mode in the NOR-based version, but now $S' = 1, R' = 1$ implies storage mode in the NAND-based version.

- **Step #2: adding an enable signal.** In the existing design, we added an enable signal by setting

$$\begin{aligned} S' &= S \wedge en \\ R' &= R \wedge en \end{aligned}$$

The idea was that since $t \wedge 0 = 0$ and $t \wedge 1 = t$ for any t , en gated (i.e., conditionally turned off) S and R : if $en = 0$ then $S' = R' = 0$ irrespective of S and R so the internal latch is in storage mode, but if $en = 1$ then $S' = S$ and $R' = R$ so the internal latch is controlled by S and R as normal.

- For NOR we have $t \bar{\vee} 0 = \neg t$ and $t \bar{\vee} 1 = 0$. As is, this swaps the semantics for en , i.e., $en = 1$ and $en = 0$ mean not enabled (or storage mode) and enabled (or update); we can deal with this by adding a NOR-based NOT gate to swap them back again. However, the additional NOT in $\neg t$ (versus just t) implies S and R will be swapped, or, equivalently Q and $\neg Q$ are swapped.
- For NAND we have $t \bar{\wedge} 0 = 1$ and $t \bar{\wedge} 1 = \neg t$. This matches the semantics for en , i.e., $en = 0$ and $en = 1$ mean not enabled (or storage mode) and enabled (or update); no additional NAND-based NOT gate is required therefore. However, the additional NOT in $\neg t$ (versus just t) implies S and R will be swapped, or, equivalently Q and $\neg Q$ are swapped.
- **Step #3: forcibly avoiding the case where $S = R = 0$.** In the existing design, we avoided the case where $S = R = 0$ by setting $R = \neg S = D$. So with the NOR-based version we set $R = \neg S \equiv S \bar{\vee} S = D$, whereas with the NAND-based version, $R = \neg S \equiv S \bar{\wedge} S = D$.