

COMS10015 lab. worksheet #10: Register Machines (RMs)

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-10_s.tar.gz

supports such cases.

§1. C-class, or core questions

▷ **S1[C].** A theme throughout this question is the increased level of *design* required, versus implementation: you will need to think more about how to produce a solution, rather than just *reproduce* one from, e.g., the lecture slot(s). Keep in mind that there is an associated, and intentional design space of valid solutions. What is presented here is therefore one approach, which emphasises clarity and ease of understanding, among many: do not automatically assume that some other, different approach is invalid.

a. Figure 1 describes the decoder implementation; an associated LogisimEvo implementation is reproduced within the archive provided.

- The *alu* signal controls a multiplexer towards the top of the data-path. The multiplexer output is connected to the input (or next value) of R' , meaning the multiplexer selects the operation applied to an input x , e.g., $x = R_i$, to produce an output r : by inspection, we can see that one of the following cases

$$alu = \begin{cases} 0 & \Rightarrow r = 0 & \text{(hard-wired to constant)} \\ 1 & \Rightarrow r = x + 1 & \text{(computed via adder component)} \\ 2 & \Rightarrow r = x - 1 & \text{(computed via subtractor component)} \\ 3 & \Rightarrow r = \text{undefined} \end{cases}$$

will apply depending on *alu*. Within the decoder, *alu* is the output of a multiplexer whose control signal is $inst_{8,7,6}$, i.e., the opcode field in the encoded instruction: basically, the idea is that we connect each i -th multiplexer input to whatever we want *alu* to be when $inst_{8,7,6} = i$. So, for example:

- if $inst_{8,7,6} = 000_{(2)} = 0_{(10)}$, this is an increment instruction: we want $alu = 1$ such that $r = x + 1$ is produced as output, so connect the 0-th multiplexer input to 1,
- if $inst_{8,7,6} = 001_{(2)} = 1_{(10)}$, this is a decrement instruction: we want $alu = 2$ such that $r = x - 1$ is produced as output, so connect the 1-st multiplexer input to 2,
- in all other cases, we do not use r : we want $alu = 0$ such that $r = 0$ is produced as a default (or placeholder) output, so connect the i -th multiplexer input to 0 for each $i \in \{2, 3, 4, 5, 6, 7\}$.
- The *we* signal controls a demultiplexer towards the bottom of the data-path. Each i -th demultiplexer output is connected to the enable input of R_i , meaning the demultiplexer allows (if $we = 1$) or disallows (if $we = 0$) update of (or write-back of a value into) that register. Within the decoder, *we* is the output of a multiplexer whose control signal is $inst_{8,7,6}$, i.e., the opcode field in the encoded instruction: basically, the idea is that we connect each i -th multiplexer input to whatever we want *we* to be when $inst_{8,7,6} = i$. So, for example:
 - if $inst_{8,7,6} = 000_{(2)} = 0_{(10)}$, this is an increment instruction: we want $we = 1$ such that a write-back occurs, so connect the 0-th multiplexer input to 1,
 - if $inst_{8,7,6} = 001_{(2)} = 1_{(10)}$, this is a decrement instruction: we want $we = 1$ such that a write-back occurs, so connect the 1-st multiplexer input to 1,
 - in all other cases, we want no write-back to occur: we want $we = 0$, so connect the i -th multiplexer input to 0 for each $i \in \{2, 3, 4, 5, 6, 7\}$.
- The *addr* signal controls a multiplexer towards the middle of the data-path. Each i -th multiplexer input is connected to the output (or current value) of R_i , meaning the multiplexer selects a register to operate on. Within the decoder, *addr* is extracted directly from the corresponding field in the encoded instruction. That is, we have $addr = inst_{5,4}$.
- The *target* signal acts as an input to a multiplexer towards the middle of the control-path. The multiplexer output is connected to the input (or next value) of PC' , meaning the multiplexer makes control-flow decisions: by inspection, we can see that one of the following cases

$$jmp \wedge cmp = \begin{cases} 1 & \Rightarrow PC' = target & \text{(hard-wired to signal)} \\ 0 & \Rightarrow PC' = PC + 1 & \text{(computed via adder component)} \end{cases}$$

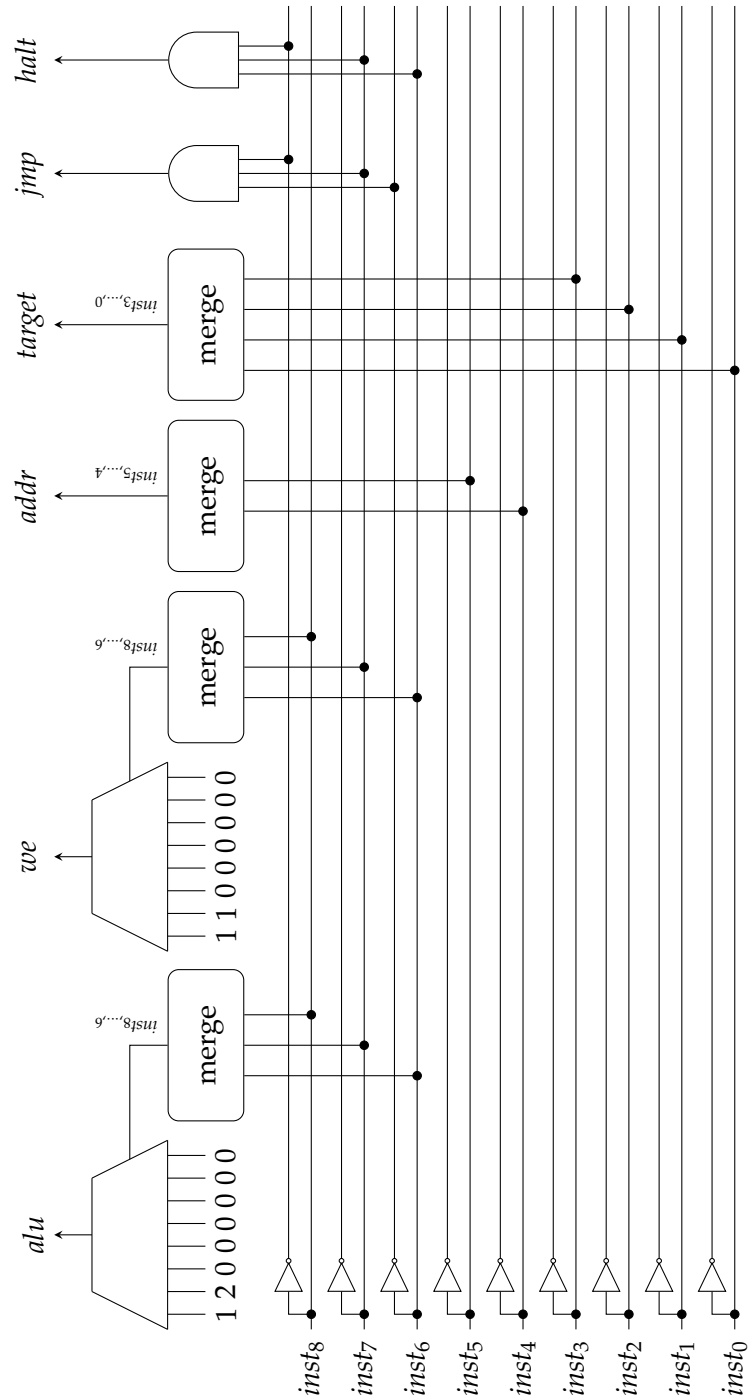


Figure 1: The decoder implementation for an example 4-register counter machine.

will apply depending on *jmp* and *cmp*. Put another way, the next instruction is fetched from either $PC + 1$ (if a branch is not required, i.e., either $jmp = 0$ or $cmp = 0$) or *target* (if a branch is required, i.e., both $jmp = 1$ and $cmp = 1$). Within the decoder, *target* is extracted directly from the corresponding field in the encoded instruction. That is, we have $target = inst_{3,2,1,0}$. In contrast, we can see by inspection that $jmp = \neg inst_8 \wedge inst_7 \wedge \neg inst_6$, or, if you prefer,

$$jmp = \begin{cases} 1 & \text{if } inst_{8,7,6} = 010_{(2)}, \text{ i.e., this is a branch instruction} \\ 0 & \text{otherwise, i.e., this is not a branch instruction} \end{cases}$$

Note that *cmp* is produced as output from the comparator component within the data-path, not by the decoder: basically,

$$cmp = \begin{cases} 1 & \text{if } R_{addr} = 0, \text{ i.e., the branch condition is satisfied} \\ 0 & \text{if } R_{addr} \neq 0, \text{ i.e., the branch condition is not satisfied} \end{cases}$$

- The *halt* signal is combined with, or, more specifically, gates the clock signals Φ_1 and Φ_2 towards the right-hand side of the data-path. The idea is that once a halt instruction is executed, the clock signals are disabled; if $halt = 1$, we gate Φ_1 and Φ_2 meaning, for example, that subsequent register updates are then disallowed. Within the decoder, we can see by inspection that $halt = \neg inst_8 \wedge inst_7 \wedge inst_6$, or, if you prefer,

$$halt = \begin{cases} 1 & \text{if } inst_{8,7,6} = 011_{(2)}, \text{ i.e., this is a halt instruction} \\ 0 & \text{otherwise, i.e., this is not a halt instruction} \end{cases}$$

- b. The sequencer is a simple FSM, which iterates through stages of the fetch-decode-execute cycle: the FSM involves three abstract states, represented as

$$\begin{aligned} S_{fetch} &\mapsto \langle 0, 0 \rangle \equiv 0_{(10)} \\ S_{decode} &\mapsto \langle 1, 0 \rangle \equiv 1_{(10)} \\ S_{execute} &\mapsto \langle 0, 1 \rangle \equiv 2_{(10)} \end{aligned}$$

i.e., a concrete 2-bit current (resp. next) state S (resp. S').

- Observe that $\langle 1, 1 \rangle \equiv 3_{(10)}$ is an unused state representation; neither the control nor data path will do anything in this state, which implies it could be viewed as an idle state. To remove it, and therefore improve efficiency with respect to instruction execution, we can replace the adder: currently it means the transition function is $S' = \delta(S) = S + 1 \pmod{2^2} = S + 1 \pmod{4}$, whereas we want $S' = \delta(S) = S + 1 \pmod{3}$. That is, if $S = 0$ then $S' = 1$, if $S = 1$ then $S' = 2$, and if $S = 2$ then $S' = 0$. To implement this behaviour we can write the truth table

S_1	S_0	S'_1	S'_0
0	0	0	1
0	1	1	0
1	0	0	0

then, by inspection, conclude that

$$\begin{aligned} S'_0 &= \neg S_1 \wedge \neg S_0 \\ S'_1 &= S_0 \end{aligned}$$

- Observe that

$$\begin{aligned} fetch &= \neg S_0 \wedge \neg S_1 \\ decode &= S_0 \wedge \neg S_1 \\ execute &= \neg S_0 \wedge S_1 \end{aligned}$$

i.e., we can replace the comparitors with implementations of these Boolean expressions.

§2. R-class, or revision questions

▷ S2[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

- ▷ S3[A]. a. The idea is that we want to set $halt = 1$ either 1) when a halt instruction is decoded, or 2) when an invalid instruction is decoded. Put another way, we

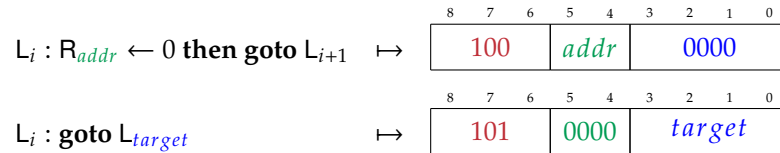
$$halt = \begin{cases} 0 & \text{if } inst_{8,7,6} = 000_{(2)} = 0_{(10)} \\ 0 & \text{if } inst_{8,7,6} = 001_{(2)} = 1_{(10)} \\ 0 & \text{if } inst_{8,7,6} = 010_{(2)} = 2_{(10)} \\ 1 & \text{if } inst_{8,7,6} = 011_{(2)} = 3_{(10)} \\ 1 & \text{if } inst_{8,7,6} = 100_{(2)} = 4_{(10)} \\ 1 & \text{if } inst_{8,7,6} = 101_{(2)} = 5_{(10)} \\ 1 & \text{if } inst_{8,7,6} = 110_{(2)} = 6_{(10)} \\ 1 & \text{if } inst_{8,7,6} = 111_{(2)} = 7_{(10)} \end{cases}$$

There are various ways to implement this, but arguably the simplest would be to use the expression

$$halt = (\neg inst_8 \wedge inst_7 \wedge inst_6) \vee (inst_8)$$

to capture the two cases: the left-hand sub-expression identifies a halt instruction and the right-hand sub-expression identifies an invalid instruction. However, this would need to be updated to reflect the additional instructions that stem from other parts of the question. Doing so is arguably easier by adopting a multiplexer-based approach, in a similar way to, e.g., the *alu* signal: doing so means using $inst_{8,7,6}$ as a control signal with the multiplexer selecting between 8 inputs, one for each possible opcode, which reflect the value of $halt$ required for an associated instruction.

- b. At a high level, “adding support for an instruction” involves (at least) two steps: 1) defining the semantics and encoding for the instruction, then 2) implementing changes to the control- and data-path that allow execution of instruction instances within a program. As such, we can approach the latter parts of this question using the same steps for each instruction type.
- The first step is arguably less difficult, in part because the instruction semantics are already defined; the remaining task is therefore to define an encoding for each instruction. Although we have some degree of freedom, the encoding *must* satisfy some required requirements and *will ideally* satisfy some further optional requirements. For example, it must allow instructions to be decoded unambiguously; ideally, it will also allow efficient decoding, use existing instruction formats, etc. So, for example, we could settle on



for the clear (top) and unconditional branch (bottom) instructions respectively.

- The second step is arguably more difficult, in part because the approach required is strongly dependent on 1) the instruction semantics, and 2) the control- and data-path used to support them; there is no generic approach to follow, for example. We focus on the approach itself in what follows, presenting an overview of the implementations only.

For the clear instruction, the implementation is straightforward. The multiplexer towards the top of the data-path controlled by op already allows selection of the value then written-back into a given register; this value is already 0 if $op = 0$. Noting that for this instruction we have $inst_{8,7,6} = 100_{(2)} = 4_{(10)}$, we make two changes within the decoder. First, for the multiplexer that outputs op , we connect the 4-th input to 0 so the value written-back into R_{addr} is selected to be 0. Second, for the multiplexer that outputs wr , we connect the 4-th input to 1 so write-back into R_{addr} is enabled.

For the unconditional branch instruction, the goal is to alter how control-flow decisions are made, i.e., to accommodate both (existing) conditional and (new) unconditional cases. Conceptually, the implementation is straightforward although more involved than for the clear instruction: control-flow decisions are made using a multiplexer towards the middle of the control-path, so implementation of unconditional branch instruction basically means altering the associated control signal. Within the control-path, we change the multiplexer control signal $jmp \wedge cmp$ to $jmp \wedge (cmp \vee uncond)$. This means *uncond* can “override” *cmp*: provided $jmp = 1$, i.e., this is a branch instruction, then whenever $uncond = 1$ said branch will be taken irrespective of *cmp*. Noting that for this instruction we have $inst_{8,7,6} = 101_{(2)} = 5_{(10)}$,

```

0B3(16) = 010110011(2)  $\mapsto$  L0 : if R3 = 0 then goto L3 else goto L1
070(16) = 001110000(2)  $\mapsto$  L1 : R3  $\leftarrow$  R3 - 1 then goto L2
080(16) = 010000000(2)  $\mapsto$  L2 : if R0 = 0 then goto L0 else goto L3
097(16) = 010010111(2)  $\mapsto$  L3 : if R1 = 0 then goto L7 else goto L4
030(16) = 000110000(2)  $\mapsto$  L4 : R3  $\leftarrow$  R3 + 1 then goto L5
050(16) = 001010000(2)  $\mapsto$  L5 : R1  $\leftarrow$  R1 - 1 then goto L6
083(16) = 010000011(2)  $\mapsto$  L6 : if R0 = 0 then goto L3 else goto L7
0AB(16) = 010101011(2)  $\mapsto$  L7 : if R2 = 0 then goto L11 else goto L8
030(16) = 000110000(2)  $\mapsto$  L8 : R3  $\leftarrow$  R3 + 1 then goto L9
060(16) = 001100000(2)  $\mapsto$  L9 : R2  $\leftarrow$  R2 - 1 then goto L10
087(16) = 010000111(2)  $\mapsto$  L10 : if R0 = 0 then goto L7 else goto L11
0C0(16) = 011000000(2)  $\mapsto$  L11 : halt

```

Figure 2: A program for an example 4-register counter machine, which sets R₃ equal to R₁ + R₂.

we make two changes within the decoder. First, we update the logic that produces *jmp* so that *jmp* = 1 for both the conditional and unconditional branches. We could do so by implementing the expression

$$jmp = (\neg inst_8 \wedge inst_7 \wedge \neg inst_6) \vee (inst_8 \wedge \neg inst_7 \wedge inst_6),$$

or, more radically, via a similar per-instruction multiplexer style approach as used for *op* and *wr*. Second, we introduce the new output *uncond*: the easiest way to do so would be to simply set

$$uncond = inst_8 \wedge \neg inst_7 \wedge inst_6,$$

meaning *uncond* = 1 for the unconditional branch instruction only.

- ▷ **S4[A].** Figure 2 describes the program implementation, which can be thought of as three parts: L₀ to L₂ clear (or zero) R₃, L₃ to L₆ add R₁ to R₃, L₇ to L₁₀ add R₂ to R₃. Also note that it depends on having R₀ = 0, allowing the construction of unconditional branches in L₂, L₆, and L₁₀. That is, it does not utilise the additional clear and unconditional branch instructions considered above.

We can demonstrate that the program computes the correct result by producing an example trace of execution. Starting with the initial configuration

$$C_0 = (l = 0, v_0 = 0, v_1 = 3, v_2 = 2, v_3 = 1),$$

Figure 3 describes such a trace. Note that initially R₁ = 3 and R₂ = 2, and the final configuration halts execution with R₃ = 3 + 2 = 5 as expected.

```

C0  = (0, 0, 3, 2, 1)
L0  ~> if R3 = 0 then goto L3 else goto L1
C1  = (1, 0, 3, 2, 1)
L1  ~> R3 ← R3 - 1 then goto L2
C2  = (2, 0, 3, 2, 0)
L2  ~> if R0 = 0 then goto L0 else goto L3
C3  = (0, 0, 3, 2, 0)
L0  ~> if R3 = 0 then goto L3 else goto L1
C4  = (3, 0, 3, 2, 0)
L3  ~> if R1 = 0 then goto L7 else goto L4
C5  = (4, 0, 3, 2, 0)
L4  ~> R3 ← R3 + 1 then goto L5
C6  = (5, 0, 3, 2, 1)
L5  ~> R1 ← R1 - 1 then goto L6
C7  = (6, 0, 2, 2, 1)
L6  ~> if R0 = 0 then goto L3 else goto L7
C8  = (3, 0, 2, 2, 1)
L3  ~> if R1 = 0 then goto L7 else goto L4
C9  = (4, 0, 2, 2, 1)
L4  ~> R3 ← R3 + 1 then goto L5
C10 = (5, 0, 2, 2, 2)
L5  ~> R1 ← R1 - 1 then goto L6
C11 = (6, 0, 1, 2, 2)
L6  ~> if R0 = 0 then goto L3 else goto L7
C12 = (3, 0, 1, 2, 2)
L3  ~> if R1 = 0 then goto L7 else goto L4
C13 = (4, 0, 1, 2, 2)
L4  ~> R3 ← R3 + 1 then goto L5
C14 = (5, 0, 1, 2, 3)
L5  ~> R1 ← R1 - 1 then goto L6
C15 = (6, 0, 0, 2, 3)
L6  ~> if R0 = 0 then goto L3 else goto L7
C16 = (3, 0, 0, 2, 3)
L3  ~> if R1 = 0 then goto L7 else goto L4
C17 = (7, 0, 0, 2, 3)
L7  ~> if R2 = 0 then goto L11 else goto L8
C18 = (8, 0, 0, 2, 3)
L8  ~> R3 ← R3 + 1 then goto L9
C19 = (9, 0, 0, 2, 4)
L9  ~> R2 ← R2 - 1 then goto L10
C20 = (10, 0, 0, 1, 4)
L10 ~> if R0 = 0 then goto L7 else goto L11
C21 = (7, 0, 0, 1, 4)
L7  ~> if R2 = 0 then goto L11 else goto L8
C22 = (8, 0, 0, 1, 4)
L8  ~> R3 ← R3 + 1 then goto L9
C23 = (9, 0, 0, 1, 5)
L9  ~> R2 ← R2 - 1 then goto L10
C24 = (10, 0, 0, 0, 5)
L10 ~> if R0 = 0 then goto L7 else goto L11
C25 = (7, 0, 0, 0, 5)
L7  ~> if R2 = 0 then goto L11 else goto L8
C26 = (11, 0, 0, 0, 5)
L11 ~> halt

```

Figure 3: An example trace of the program described in Figure ??.