COMS10015 lab. worksheet #10

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_2024_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 *re*produce 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.

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., $x = \mathsf{R}_i$, 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 an 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 wr 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 $wr = 1$) or disallows (if $wr = 0$) update of (or write-back of a value into) that register. Within the decoder, wr 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 wr 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 $wr = 1$ 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 an decrement instruction: we want $wr = 1$ 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 $wr = 0$, 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 $addr = inst_{5,4}$.

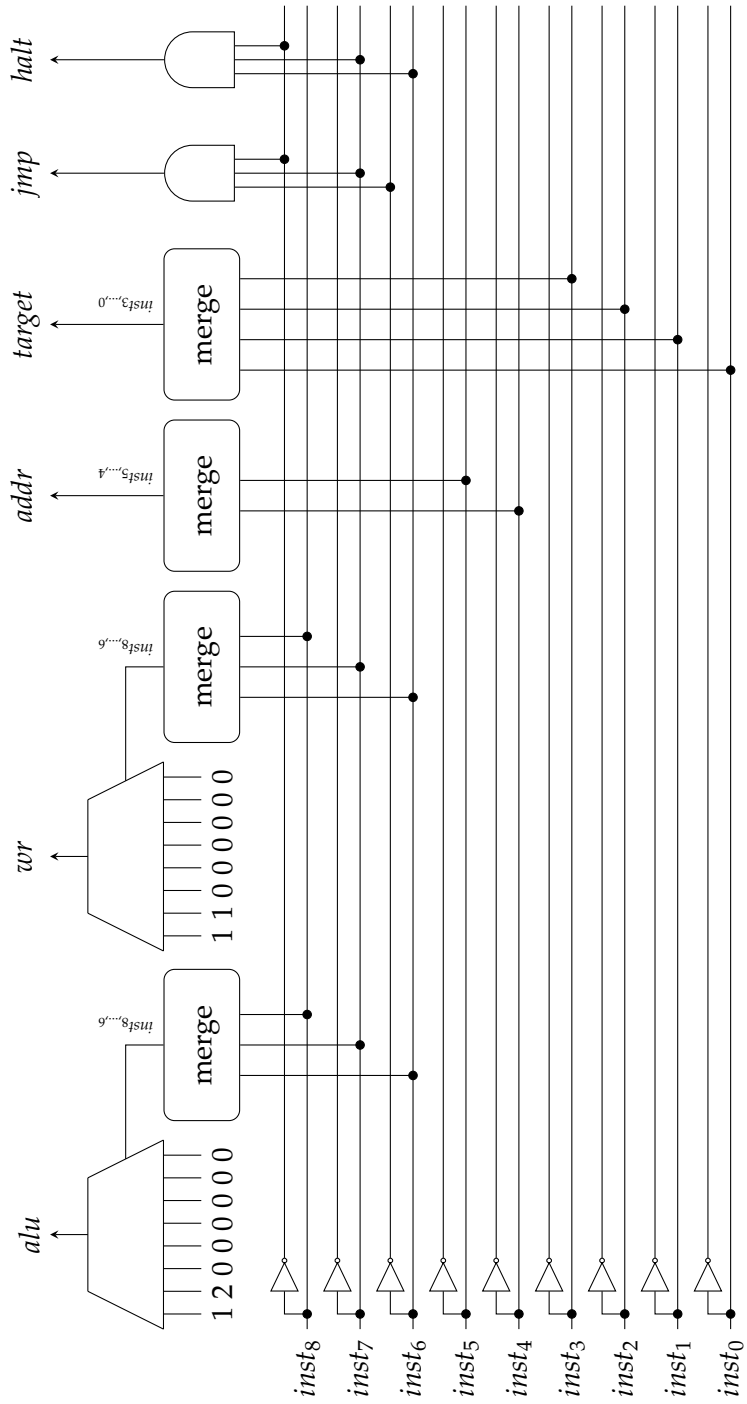


Figure 1: *The decoder implementation for an example* 4*-register counter machine.*

- 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

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

  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,

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

- 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_1 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,

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

§2. R-class, or revision questions

▷ **S2[R].** There is a set of solutions available at

https://assets.phoo.org/COMS10015_2024_TB-4/csdsp/sheet/misc-revision_s.pdf

§3. A-class, or additional questions

▷ **S3[A].** A solution for the first part can be localised to the decoder. 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.

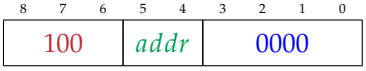
At a high level, "adding support for an instruction" involves (at least) two steps: 1) defining the semantics and encoding for the instruction, then 2) implementing changes to the control- and data-path that allow execution of instruction instances within a program. As such, we can approach the latter parts of this question using the same steps for each instruction type.

$$
\begin{array}{rcll}
0B3_{(16)} & = & 010110011_{(2)} & \mapsto \quad \mathsf{L}_0 \; : \textbf{if } \mathsf{R}_3 = 0 \textbf{ then goto } \mathsf{L}_3 \textbf{ else goto } \mathsf{L}_1 \\
070_{(16)} & = & 001110000_{(2)} & \mapsto \quad \mathsf{L}_1 \; : \mathsf{R}_3 \leftarrow \mathsf{R}_3 - 1 \textbf{ then goto } \mathsf{L}_2 \\
080_{(16)} & = & 010000000_{(2)} & \mapsto \quad \mathsf{L}_2 \; : \textbf{if } \mathsf{R}_0 = 0 \textbf{ then goto } \mathsf{L}_0 \textbf{ else goto } \mathsf{L}_3 \\
097_{(16)} & = & 010010111_{(2)} & \mapsto \quad \mathsf{L}_3 \; : \textbf{if } \mathsf{R}_1 = 0 \textbf{ then goto } \mathsf{L}_7 \textbf{ else goto } \mathsf{L}_4 \\
030_{(16)} & = & 000110000_{(2)} & \mapsto \quad \mathsf{L}_4 \; : \mathsf{R}_3 \leftarrow \mathsf{R}_3 + 1 \textbf{ then goto } \mathsf{L}_5 \\
050_{(16)} & = & 001010000_{(2)} & \mapsto \quad \mathsf{L}_5 \; : \mathsf{R}_1 \leftarrow \mathsf{R}_1 - 1 \textbf{ then goto } \mathsf{L}_6 \\
083_{(16)} & = & 010000011_{(2)} & \mapsto \quad \mathsf{L}_6 \; : \textbf{if } \mathsf{R}_0 = 0 \textbf{ then goto } \mathsf{L}_3 \textbf{ else goto } \mathsf{L}_7 \\
0AB_{(16)} & = & 010101011_{(2)} & \mapsto \quad \mathsf{L}_7 \; : \textbf{if } \mathsf{R}_2 = 0 \textbf{ then goto } \mathsf{L}_{11} \textbf{ else goto } \mathsf{L}_8 \\
030_{(16)} & = & 000110000_{(2)} & \mapsto \quad \mathsf{L}_8 \; : \mathsf{R}_3 \leftarrow \mathsf{R}_3 + 1 \textbf{ then goto } \mathsf{L}_9 \\
060_{(16)} & = & 001100000_{(2)} & \mapsto \quad \mathsf{L}_9 \; : \mathsf{R}_2 \leftarrow \mathsf{R}_2 - 1 \textbf{ then goto } \mathsf{L}_{10} \\
087_{(16)} & = & 010000111_{(2)} & \mapsto \quad \mathsf{L}_{10} : \textbf{if } \mathsf{R}_0 = 0 \textbf{ then goto } \mathsf{L}_7 \textbf{ else goto } \mathsf{L}_{11} \\
0C0_{(16)} & = & 011000000_{(2)} & \mapsto \quad \mathsf{L}_{11} : \text{halt}
\end{array}
$$

Figure 2: *A program for an example* 4*-register counter machine, which sets* R_3 *equal to* $\mathsf{R}_1 + \mathsf{R}_2$.

- 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

  $\mathsf{L}_i : \mathsf{R}_{addr} \leftarrow 0$ **then goto** $\mathsf{L}_{i+1} \quad \mapsto$

  L_i : **goto** $\mathsf{L}_{target} \quad \mapsto$

  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 $inst_{8,7,6} = 101_{(2)} = 5_{(10)}$, 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: 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

$$
\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}
$$

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

the construction of unconditional branches in L_2, L_6, and L_{10}. 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 $\mathsf{R}_1 = 3$ and $\mathsf{R}_2 = 2$, and the final configuration halts execution with $\mathsf{R}_3 = 3 + 2 = 5$ as expected.