COMS12200 lab. worksheet: week #8

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 archive available at

http://www.cs.bris.ac.uk/home/page/teaching/material/arch_new/sheet/lab-8_s.tar.gz

supports such cases.

S2. To capture all the techniques and components covered thus far within one motivating example, the lecture(s) focused on the **HP-35** calculator (a real product, albeit from circa 1972). While the lecture(s) dealt with the design of *this* calculator, the worksheet question asks for an implementation of *any* calculator. As such, we make a compromise by retaining a focus on HP-35 but making simplifications that include

- operating in binary, rather than **Binary Coded Decimal (BCD)**,
- using with 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 implemented using an algorithm rather than a circuit (e.g., sin and cos trigonometric functions).

On one hand, the real HP-35 has a significantly more complex[1] internal design than the one developed here: for the sake of a name we term the result a "limited" (versus real) HP-35 throughout, but it is important remember it is not an emulator of a real HP-35 device. On the other hand, however, the goal is to demonstrate we can design *and* implement a device capable of recognisable forms of computation. While arguably still some way from a modern micro-processor, similar concepts continue to apply in modern devices of this type: solid understanding of the former clearly provides a good stepping stone toward understanding of the latter.

In the same way as the associated question, this solution is unlike those in previous worksheets. Whereas they attempted to show what *should* be done (to help you learn about a given topic), the solution attempts to demonstrate what *could* be done: keep in mind it is more complex and represents more work than was expected from you

A limited HP-35: design

Figure 1 presents some photographs, both internal and external, of a real HP-35. From the perspective of a user, operation of the HP-35 is precised by Figure 1b: the rear casing essentially provides an instruction manual, or more formally a set of semantics for each key press by the user.

How the calculator is used partly dictates the internal design, and some aspects of said design are alluded to by the semantics. For example, it is clear the HP-35 maintains four internal registers (or accumulators) named X, Y, Z and T plus S, a fifth storage register (which we might colloquially term "memory", but is *not* an SRAM or similar); the value of X is shown on the LED display. These registers can be manipulated by pressing the control and arithmetic keys, each of which invokes a specific operation. Using r' to denote the next value of a 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")

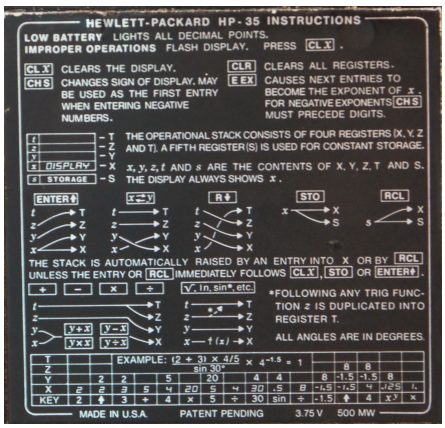
[1] A comprehensive tear-down and analysis of the real HP-35 is available via

http://www.jacques-laport.org/

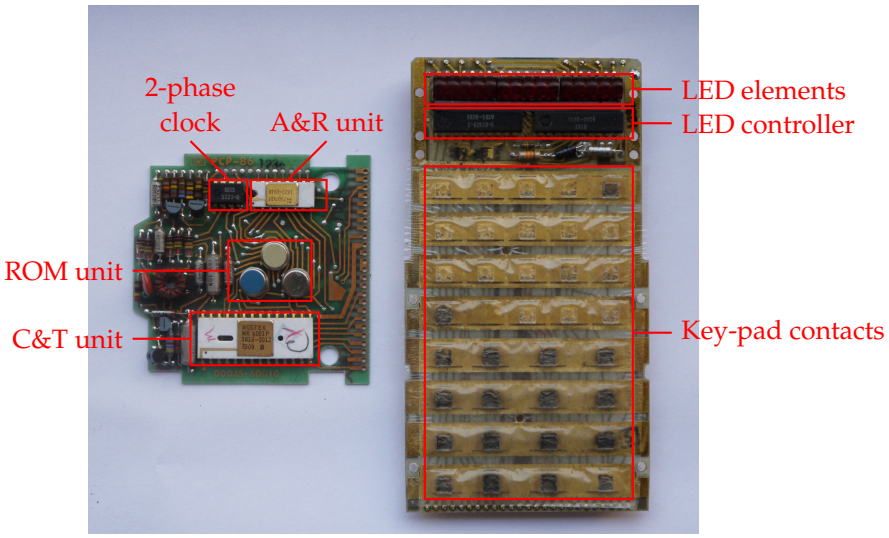
with additional miscellany at

http://www.pmonta.com/calculators/hp-35/

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



(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.*

- $S' \leftarrow X$

e 'RCL' (or "recall")

- $X' \leftarrow S$

f '↑' (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 `manual.pdf` in the support archive.

Some of the 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* parentheses. Even more useful wrt. implementation, a given expression can be evaluated very naturally using a **stack**: 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 slight variation, in that the expression above would be evaluated using the following key presses

		Key-press									
		1	9	↑	5	−	1	↑	2	+	×
Register	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 to 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, and
- a **Control and Timing (C&T)** unit

plus

- a **keypad** to provide input, and
- an **LED-based display** to provide output

which are spread over two **Printed Circuit Boards (PCB)**. Roughly the same organisation is reproduced by the limited HP-35 Logisim implementation. Each sub-component implementing C&T, A&R and ROM units (whose internal design is expanded on in the Sections below) is, at the top-most level, connected to each other and the keypad and LED display:

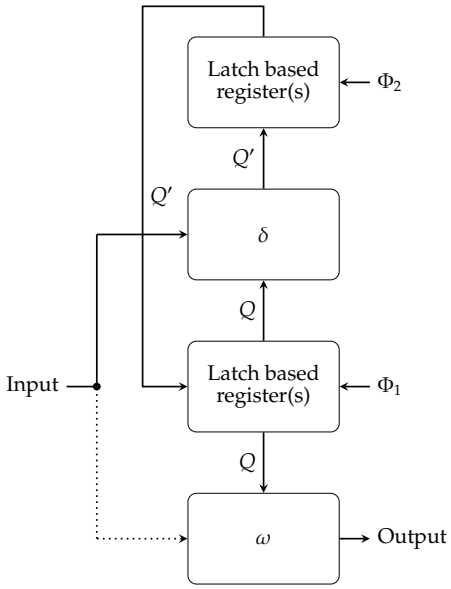


Figure 2: *A generic FSM framework (for a* 2*-phase clocking strategy) into which one can place implementations of the state, δ (the transition function) and ω (the output function).*

- A keypad (left) provides input: the state of each key forms 1 bit of a 32-bit keypad state K 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 on the left-hand side is connected to the bit of K denoted by the right-hand side: for example, the state of key '9' is attached to the 9-th bit of K. One can also view the right-hand side as a 5-bit key-code C for the key on the left-hand side.

- A set of probe components (right) allows one to inspect the values held by registers X, Y, Z, T and S. The real HP-35 shows the value of X only, via the LED display; showing them all allows easier debugging and demonstration.

The choice of key mapping is not arbitrary: several features are designed explicitly to make other tasks a lot easier. For example, notice that by testing if $C_4 = 0$ we determine whether the key pressed is numeric; if so, $C_{3...0}$ gives the associated (unsigned, 4-bit) integer value.

The Control and Timing (C&T) unit

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

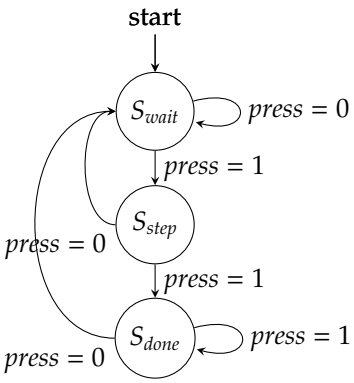
- one 32-bit keypad state K, and
- two 1-bit, 2-phase clock signals (i.e., Φ_1 and Φ_2)

as input, and produces

- one 5-bit current key-code C,
- one 8-bit current key-value V,
- two 1-bit enable signals en_1 and en_2, and
- two 1-bit control signals *append* and *raise*

	δ		ω	
Q	Q'		en_2	en_1
	$press = 0$	$press = 1$		
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 three constituent parts, read from top-to-bottom:

- The keypad state K is fed into a priority encoder which is tasked with translating a press of the i-th key, meaning $K_i = 1$ and $K_j = 0$ for $i \neq j$, into the key-code C; it also produces an output which is 1 iff. a key is currently being pressed, and can be thought of simply as

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

  The current and previous key-codes C and P are stored in two 5-bit registers; the former is enabled by *press*, meaning it is latched automatically as soon as it is pressed (and therefore remains valid even if unpressed while being processed). The least-significant 4 bits of C, i.e., $C_{3\ldots0}$, are extended into an 8-bit integer key-value V ready for use for use by the ALU if need be.

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

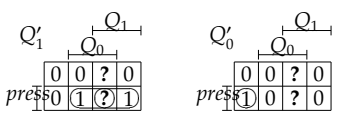
  The description in Figure 3 should read fairly intuitively for δ: we start by waiting until a key press is available, then take action to process it, before again waiting for the key to then be *un*pressed (to avoid repeated processing if the key is held down). This is achieved using 3 states with the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

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

  and we have one unused state (namely $\langle 1, 1 \rangle$). 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



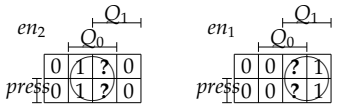
and hence write the following Boolean expressions

$$\begin{aligned} Q'_1 = (\ & press & & & \wedge & & Q_0 &) \vee \\ (\ & press & \wedge & Q_1 & & & &) \end{aligned}$$

$$Q'_0 = (\ press \wedge \neg Q_1 \wedge \neg Q_0\ )$$

for Q'_1 and Q'_0.

In contrast, ω needs some explanation. The idea is en_1 and en_2 will explicitly control latches in the A&R unit, rather than using clock signals Φ_1 and Φ_2 to do so. 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 fairly simple: from the same truth table above we form



and can hence write

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

- There is an important caveat to semantics per Figure 1b as introduced above: the "stack is automatically raised" part needs some care. 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.

  A simple key press decoder is tasked with producing these from P, the latched previous key-code.

The Arithmetic and Register (A&R) unit

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

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

as input, and produces

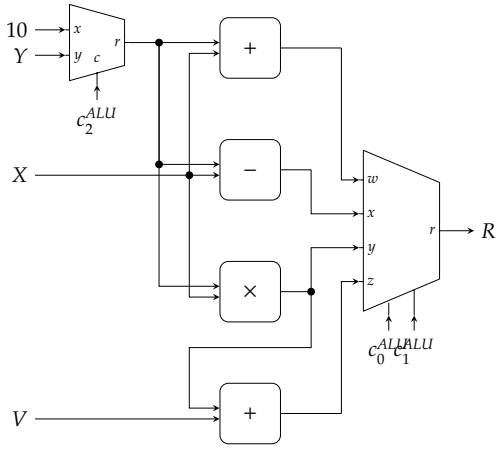
- five 8-bit values of the X, Y, Z, T and S registers

as output. There are two constituent parts, read from top-to-bottom:

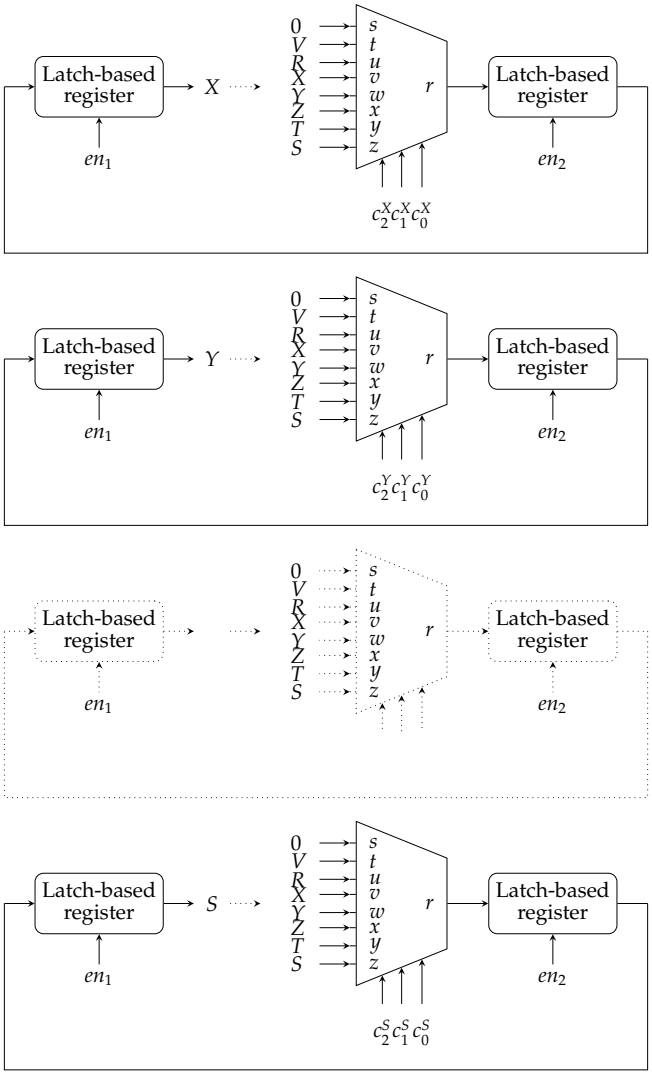
- A fairly simple **Arithmetic and Logic Unit (ALU)** is used a) to compute any result associated with pressing an arithmetic key (e.g., add Y and X when '+' is pressed), and also b) update the X register when a numeric key is pressed. Figure 4a illustrates the design, with the precise behaviour dictated both by numerical inputs (namely the value of registers Y and X, plus key-value V) *and* the 3-bit control signal c^{ALU}.

  There are four arithmetic components in the center of the ALU, each of whose inputs are controlled using c_2^{ALU} (via a multiplexer towards the left): we have that

$$x = \begin{cases} 10 & \text{if } c_2^{ALU} = 0 \\ Y & \text{if } c_2^{ALU} = 1 \end{cases}$$



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



(b) *The register file.*

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

$y = X$ and $z = V$. Each arithmetic component is combinatorial, so continuously computes an output from x, y and z. One output is selected as the ALU output R using $c^{ALU}_{1...0}$ (via a multiplexer towards the right); basically we get

$$R = \begin{cases} x + y & \text{if } c^{ALU}_{1...0} = 00_{(2)} \\ x - y & \text{if } c^{ALU}_{1...0} = 01_{(2)} \\ x \cdot y & \text{if } c^{ALU}_{1...0} = 10_{(2)} \\ x \cdot y + z & \text{if } c^{ALU}_{1...0} = 11_{(2)} \end{cases}$$

Putting everything together, we find that

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

Clearly only *some* of these combinations are useful (e.g., $R = 10 + X$ is not), but the point is that all operations required by our semantics can be computed: we just need to set c^{ALU} appropriately.

Note that this simple design can clearly be improved in various ways; the current design stresses clarity over efficiency. In Logisim, for instance, the multiplication component allows an extra input: it can perform a multiply-accumulate operation $x \cdot y + z$ without the need for an extra adder component (which is currently tasked with adding z to $x \cdot y$). Likewise, we previously encountered an adder/subtractor design; using c^{ALU}_0 as the control signal to determine whether an addition or subtraction is required, this could could be used to combine the currently separate adder and subtractor components (tasked with computing $x + y$ and $x - y$).

- The state of the calculator, wrt. evaluation of a given expression, 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 an 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 and output latch r' which house the current and next values respectively,
  - the input and output latches are enabled by en_1 and en_2 respectively (so the output latch is enabled when the C&T unit FSM is in the S_{step} state, and 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 c^r:
    - $c^r = 000_{(2)}$ selects constant 0,
    - $c^r = 001_{(2)}$ selects key-value V,
    - $c^r = 010_{(2)}$ selects ALU output R,
    - $c^r = 011_{(2)}$ selects X,
    - $c^r = 100_{(2)}$ selects Y,
    - $c^r = 101_{(2)}$ selects Z,
    - $c^r = 110_{(2)}$ selects T, and
    - $c^r = 111_{(2)}$ selects S.

Put simply, to implement the original semantics we just need to set each c^r to correctly select each next value. Based on the the latched current key-code C, 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 $\odot \in \{$'+', −, ×$\}$	?	?	$Y \odot X$	Z	T	T	S
'CLR'	?	?	0	0	0	0	0
'STO'	?	?	X	Y	Z	T	X
'RCL'	?	?	S	Y	Z	T	S
'↑'	?	?	X	X	Y	Z	S

So, for instance, if the '↑' 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 $c^X = 011_{(2)}$, $c^Y = 011_{(2)}$, $c^Z = 100_{(2)}$, $c^T = 101_{(2)}$, and $c^S = 111_{(2)}$ in order to select the next value correctly for each case.

Notice that neither C, *raise* or *append* are inputs to the A&R unit; this means *it* cannot derive c^{ALU} or c^r for $r \in \{X, Y, Z, T, S\}$. Rather, the task of generating the correct control signals for the A&R unit is delegated to the ROM unit as outlined below.

The ROM unit

The ROM unit accepts

- one 5-bit current key-code C, and
- two 1-bit control signals *append* and *raise*

as input, and produces

- five 3-bit control signals c^X, c^Y, c^Z, c^T and c^S for the X, Y, Z, T and S multiplexers, and
- one 3-bit control signal c^{ALU} for the ALU.

as output.

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

If you think about it, a given ROM is just a (fixed) look-up table: from an n'-bit address x, it yields a w-bit word $r = MEM[x] = f(x)$ 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 components in the A&R unit. Consider the top ROM which is used to produce c^{ALU}, a control signal for the ALU, for example. It has a total of 32 words, each of 3 bits. The 5-bit address used to access content is simply $x = C$, i.e., the current key-code; we use this to look-up a 3-bit $c^{ALU} = MEM[x]$. Given c^{ALU}_2 and $c^{ALU}_{1...0}$ are used to select the ALU input and operation respectively, we therefore get the following behaviour

C	$c^{ALU} = MEM[C]$		
$00000_{(2)}$	$011_{(2)}$	$\rightsquigarrow$	$C =$ '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 $10 \cdot X + 9$
⋮	⋮		
$10000_{(2)}$	$100_{(2)}$	$\rightsquigarrow$	$C =$ '+', ALU computes $Y + X$
$10001_{(2)}$	$101_{(2)}$	$\rightsquigarrow$	$C =$ '-', ALU computes $Y - X$
$10010_{(2)}$	$110_{(2)}$	$\rightsquigarrow$	$C =$ '×', ALU computes $Y \cdot X$
⋮	⋮		

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 using a combinatorial circuit to *compute* c^{ALU} from C: 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., c^X, housed in the A&R unit. In their cases, we also need to consider *raise* and *append*, meaning each ROM has 128 words, each of 3 bits; we form a 7-bit address as

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

and look-up $c^X = MEM[x]$ 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 Logisim 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., $r = MEM[x]$).

  For example, the line

  ```
  ADD 10000 100
  ```

  implies that given $x = 10000_{(2)}$, $MEM[x] = 100_{(2)}$. 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., c_2^{ALU} should be set to $1_{(2)}$ (to select Y as an input) and $c_{1...0}^{ALU}$ should be set to $00_{(2)}$ (to select an addition operation); this is exactly what happens, because

  $$c^{ALU} = MEM[x] = 100_{(2)}.$$