Implementation of the Hack Computer

Prof. Naga Kandasamy
ECE Department, Drexel University

The problems are due August 4, 2023, by 11:59 pm. Please submit original work.

Build the Hack computer platform, culminating in the topmost computer chip, in the following order:

1. (**15 points**) Using the supplied `Memory.hdl` file as the starting point, build the memory chip using the RAM16K, screen, and keyboard chips along with some additional logic. The screen and keyboard chips are available as built-in chips. Also use the built-in version of the RAM16K chip since it provides a GUI that allows for contents of memory to be inspected during program execution.

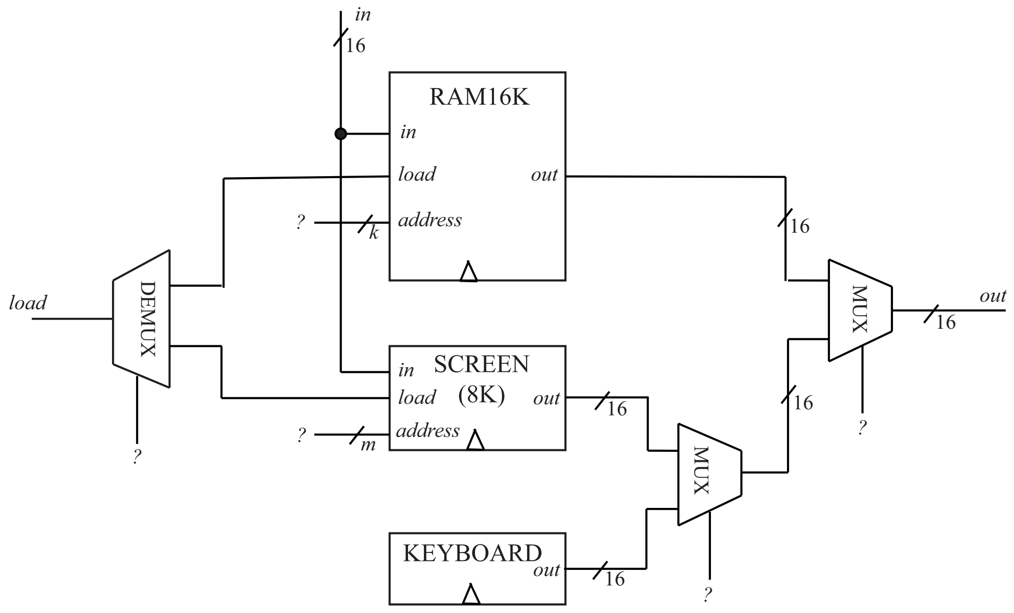

Fig. 1: Implementation of the memory chip.

Figure 1 shows one way to implement the memory chip. It is your job to devise how to decompose the incoming 15-bit address — shown as "?" in the figure — to achieve the correct functionality. Referencing an address within the range 0 to 16383 (0x0000 to 0x3FFF) results in the program accessing the 16K RAM (used to store data). An address reference within the range 16384 to 24575 (0x4000 to 0x5FFF) accesses the 8K screen memory map. Finally, referencing address 24576 (0x6000) accesses the keyboard memory map.[1] Any memory reference beyond 0x6000 is considered invalid.[2]

[1]When running a program within the hardware simulator, whenever a key is pressed on the physical keyboard, its 16-bit ASCII code appears as the output of the keyboard chip. When no key is pressed, the chip outputs 0. In addition to the usual ASCII codes, the keyboard chip also recognizes some additional keys that are listed in the textbook.

[2]Your implementation is not required to detect an illegal access since there is nothing we can do in this case — the Hack CPU does not support exception handling in hardware (yet).

Test your memory chip for correctness by executing the supplied `Memory.tst` script within the hardware simulator. Before running this script, select the 'screen' option from the 'View' menu.

2. (**25 points**) Using the supplied `CPU.hdl` file as the starting point, build the datapath for the Hack CPU using the blueprint in Fig. 2. Use built-in chips for the A and D registers, PC, and for the ALU. Built-in versions of the A and D registers, `ARegister` and `DRegister`, respectively, function exactly the same as the parallel-load registers you have built previously, but with the added benefit of a GUI that allows you to inspect register contents during program execution.

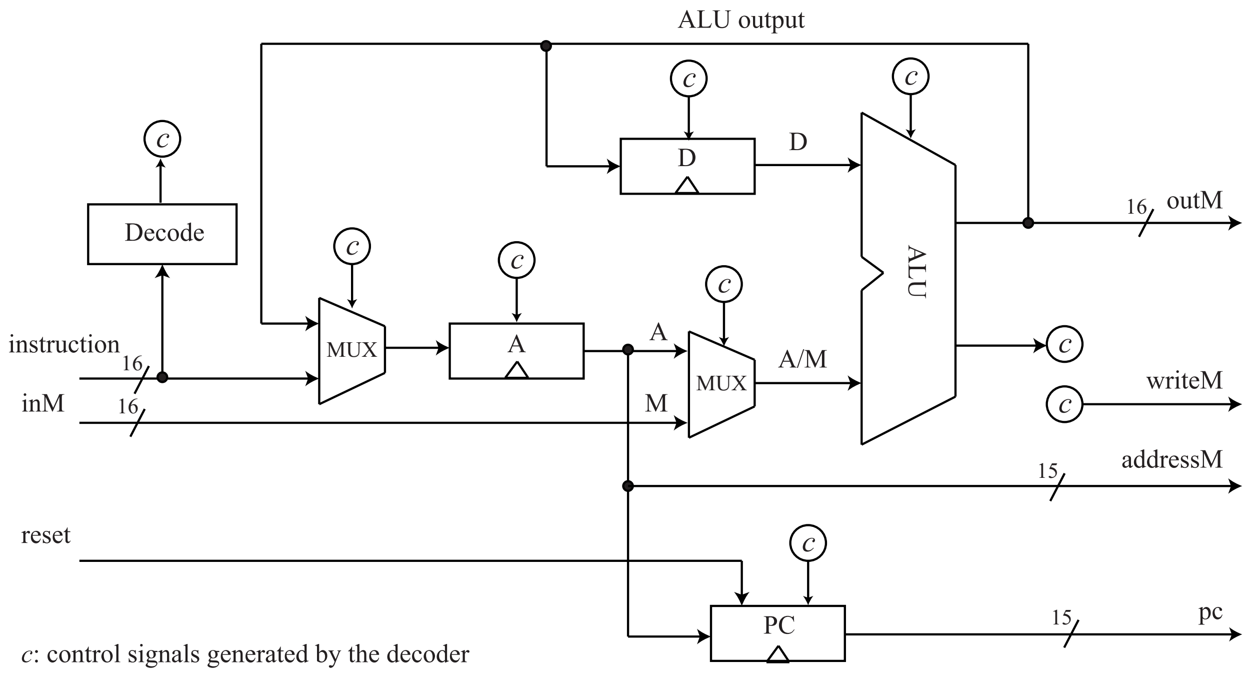

Fig. 2: Datapath for the Hack CPU showing the data and address paths.

I implemented the datapath in about 30 lines of HDL code. Wiring the various datapath components as shown in Fig. 2 is relatively straightforward. In terms of designing the decode logic, a separate "decoder" chip is not necessary. Rather determine how to decompose the various bit-fields in the incoming 16-bit instruction to generate the control signals, circled in 'c,' for the various datapath components. In some cases, you must implement simple Boolean circuits using basic built-in gates to generate these signals. To determine the various control signals, use the following two pieces of information: what type of instruction is it (A or C)?; and what do the bits corresponding to the various instruction fields (`comp`, `dest`, and `jump`) indicate?

Let's examine two examples:

- The load signal for the `ARegister` must be asserted under the following two conditions: (1) when the instruction type is A, as indicated by the `instruction[15]` bit; and (2) when the instruction type is C, again indicated by `instruction[15]`, and the destination bit `instruction[5]` indicates that the ALU output must be written to `ARegister`. The corresponding Boolean expression is:

```
loadARegister = instruction[15]' OR
                (instruction[15] AND instruction[5])
loadARegister = instruction[15]' OR instruction[5]
```

- Once the program begins execution, the program counter must be incremented every clock cycle. This behavior, however, can be overridden by the *load* signal which loads the `PC` chip with contents of `ARegister` in case of a taken branch instruction and by the *reset* signal which resets the counter value to 0. Let's take the assembly code `D;JGT` as an example. The load signal for `PC` must be asserted if the instruction is C type, the `jump` field `j1 j2 j3` indicates a JGT instruction (bit pattern 001), and if the result is > 0, which can be inferred using the *ng* and *zr* signals provided by the ALU. The corresponding Boolean expression is

  ```
  instruction[15] AND JGT AND (zr' AND ng')
  ```

  which can be simplified a little using De-Morgan's law as

  ```
  instruction[15] AND JGT AND (zr OR ng)'
  ```

  Develop functions for other branch conditions along similar lines and generate the final load signal to the `PC` as an OR-ed value of these functions.

Test the completed `CPU.hdl` design using the supplied `CPU.tst` script which supplies a battery of tests to your design and records responses in `CPU.out`. Responses are compared line-by-line to expected responses stored in `CPU.cmp`. If responses do not match, an error message is displayed along with the offending line number. Compare those specific lines in the `CPU.out` and `CPU.cmp` files for discrepancies and use that information to debug your design.

3. (**10 points**) Using the supplied `Computer.hdl` file as a starting point, compose the topmost computer chip with the memory and CPU chips developed earlier. Use the built-in ROM32K chip for instruction memory. The blueprint is shown in Fig. 3.

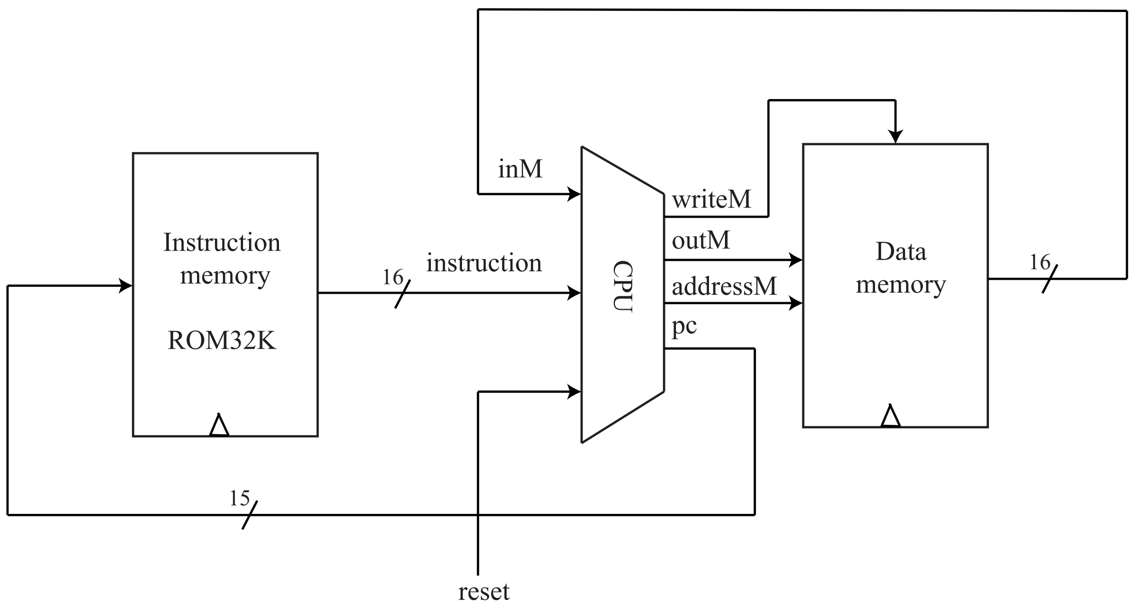

Fig. 3: Top-level design of the Hack computer.

Test the completed Hack computer in the hardware simulator using the supplied scripts which execute sample programs written in the Hack machine language. They load the computer chip into the hardware simulator, load a program from an external text file into its ROM chip, and then run the clock some number of cycles to ensure program completion. The supplied files are as follows:

- `ComputerAdd.tst` executes `Add.hack` which adds two constants, 2 and 3, and writes the result to RAM[0].
- `ComputerMax.tst` executes `Max.hack` which computes the maximum of RAM[0] and RAM[1] and writes the result to RAM[2].
- `ComputerRect.tst` executes `Rect.hack` which draws a rectangle of width 16 pixels and length RAM[0] at the top left of the screen.
- `ComputerMult.tst` executes `mult.hack` which multiplies two constants, 5 and 10, and writes the result to RAM[0].
- `ComputerSeriesSum.tst` executes `seriesSum.hack` which calculates the sum of the series $1 + 2 + \ldots + 10$ and writes the result to RAM[17].

Please read through the test scripts carefully to understand the instructions given to the simulator. Appendix B in the textbook may be a useful reference here. Before running these scripts, select the 'screen' option from the 'View' menu in the simulator if you would like to visualize ROM, RAM, and register contents during program execution.

Submission Instructions

Submit the completed `Memory.hdl`, `CPU.hdl`, and `Computer.hdl` files as a single `.zip` file via BBLearn.