SYSTEMS I — LAB 8
Designing an ISA and a CPU

1 Designing your own ISA

The first stage of this project requires you to design your own *instruction set architecture (ISA)*: a specification for format of machine code words and their meaning. You must define the size of each machine code instruction, how its bits are divided, what the values for each of those bits imply, and how the flow from one instruction to the next should progress. There is nothing magical about any one design or another; each design decision involves a tradeoff. Specifically, an ISA that allows you a simpler CPU datapath and control may be more complex to program, or *vice versa*. Your primary goal is to design something that works. To the extent that you can choose your tradeoffs, you should try to do so.

Your ISA should be designed within the following constraints:

1. **Word length:** The machine word should be one byte (8 bits). That means that each register should hold one byte, and that each memory address should be one byte. Consequently, your main memory should $2^8 = 256$ bytes long.

2. **Instruction length:** A machine code instruction may consist as many words as you like. I strongly recommend that it be a fixed number—that is, don't try to vary the length of the instruction depending on the opcode, since that approach requires a more complex CPU design. You are most likely to choose to use 1- or 2-word instructions, but larger choices are possible.

3. **Registers:** Your ISA must specify how many addressable registers are available for an instruction to use. It is possible to use zero registers, having each instruction specify not a register number but a main memory address for sources and destinations for particular operations. That is, you could have an instruction that specifies addition of two values, where the source values are taken from two main memory locations, and the destination value is immediately written to a destination main memory location.

   Alternatively, you may assume a register file with as many registers as you like. I recommend choosing a power of 2 (thus avoiding the possibility of instructions with invalid register numbers), and I also recommend not exceeding 8 registers (just to keep the task of constructing the CPU tractable).

4. **Capabilities:** Your ISA must be capable of the following operations:

   - *Logic:* Your ISA must allow for bitwise AND and OR operations on two input values, and NOT on a single input value. It may also provide other logic operations, such as XOR, NAND, NOR, etc.
   - *Arithmetic:* There must be instructions to perform *addition* and *subtraction* on two input values. You may also provide instructions for *multiplication* or *division*. as well as any other operation that appeals to you (e.g., perhaps an *arctangent* would be helpful . . . but then again, perhaps not).

- *Unconditional flow control:* There must be at least one type of *jump* instruction. You may choose to use labels (i.e., immediate values) that are expressed either as literal main memory locations or as offsets from the PC (depending on the size of your instructions). The jump target could also be register-based, like the MIPS `jr` instruction. You can provide multiple *jump* variants, but you need only one.
- *Conditional flow control:* You must provide the ability to compare two values for equality or inequality, and the to *branch* (jump) only if that condition is true. How you structure these instructions is up to you. Note that you can rely on the programmer to perform some arithmetic first. For example, you may choose to provide only comparisons to zero (*equal to zero*, *less than zero* or *negative*, *greater than zero* or *positive*); with that ability to perform comparisons, any comparison between two values is possible when combined with simple arithmetic operations.
- *Setting constants:* A program must be able to load constants into either registers or main memory locations (or both). Your ISA must provide instruction(s) that load immediate (within-the-instruction) constants into a register or into main memory. For some ISA's, in order to assign all of the bits in a register, a programmer may need to use one instruction to assign one half of the bits and a second instruction to assign the other half. For example, in MIPS, to assign all 32 bits of a register, the programmer must use the *load upper immediate* (`lui`) instruction to assign the upper 16 bits, and then use the `ori` instruction to assign the lower 16 bits (without altering the upper 16 just assigned).
- *Main memory:* If your ISA has registers, then there must be instructions to *load* words from and *store* words into main memory. (If you create a registerless ISA, then you do not need such instructions, since all source and destination values will be drawn from main memory locations.)

2 Designing a CPU to implement your ISA

Once you have specified your ISA, you must build a CPU to carry out instructions of that form. In particular, you must build a full datapath and control. Your design must include a register file (if your ISA assumes any addressable registers), a main memory, a PC, and an ALU. It should be possible to load the main memory with instructions of the format specified by your ISA, and then to have your CPU carry out the program specified by those instructions.

Notice that the simulator contains pre-made ALU, main memory, and register modules that should make your task simpler. However, there are many details for which you must design and connect to your datapath and control. Expect that creating the CPU in the simulator will take some time.

2.1 Design decisions

The description above leaves a great deal of design space within which to work. Moreover, many of the choices that you make in designing your ISA will influence and constrain the datapath and control that you build to implement that ISA. You may find that once you consider these

constraints—that is, once you see how your datapath and control would need to be constructed—you may choose instead to modify your ISA in order to simplify your hardware implementation.

Instruction length: Multiple-word instructions can make the programming task easier, and make your ISA and CPU relatively similar to the MIPS design. However, that choice implies that your CPU will require multiple clock cycles to carry out each instruction. For each word of the n-word machine code instruction, at least $n - 1$ cycles will be required to load the first $n - 1$ words of the current instruction into temporary registers. At the earliest, on the n^{th} cycle the processor will have all of the bits of the instruction available to decode and execute. Thus, your control unit's inputs are no longer just the opcode, but also the cycle number for the processing of this instruction.

Unaddressable registers: Perhaps a single-word instruction would be better? Yes, single-word instructions can be loaded from main memory and then immediately decoded and executed, avoiding the need (at least in terms of fetching the instruction) for multiple clock cycles per instruction. However, you may have noticed that a single word may have insufficient space to specify all of the registers, addresses, or immediate values that you desire. For example, if you want to add the values from two registers and store the resulting sum in a third register, there simply are not enough bits in one word for an opcode and three reigster numbers.

What to do? Create instructions that **perform simpler tasks** and that store those results in an intermediate register called an *accumulator*. So, for example, consider that to add the values in two source registers and then store the sum in a third register, you split the task across multiple instructions. One possibility is that you specify an `add` on the two source registers, but you don't specify a destination register. The result of the addition is placed into the accumulator so that the next instruction can `copy` the contents of the accumulator into the destination register. The programming task becomes more difficult, but the circuit design may be simpler.

Main memory access: Each main memory address is a one-word value that specifies a particular byte in the main memory. However, depending on how you have structured your ISA and imagined your CPU datapath and control, you may have to think carefully about how main memory is used.

For example, consider the MIPS *load-word* (`lw`) instruction. To fetch, decode, and execute this instruction, a CPU would need first to load the instruction itself from main memory. Then, however, once the target main memory address is calculated, the CPU must load the requested value into some register in the register file. Thus, this instruction makes **two** uses of the main memory, and those two uses cannot be performed at the same time.

One possible solution to this problem is to separate the *fetching* of an instruction from its *decoding* and *execution* phases. That is, carry out each instruction using at least two cycles: one to fetch the instruction from main memory into a register, and a second to carry out the instruction held in that register. This approach may fit naturally with handling multi-word instructions (above).

Another approach is to divide main memory into two components. The first component can be used to store only instructions, and the machine code of any program would have to be placed here. The second component can contain only data, and be accessed only by *load* and *store* instructions. This approach is less realistic than a real CPU using a typically unified main memory, but it is a reasonable approach to creating a working CPU.

This list of design issues is hardly comprehensive, but it include a few of those likely to affect

your ISA and CPU designs. Thinking ahead about them may help you to save time, and to see the breadth of possibilities.

3 Testing your CPU

In order to determine whether your datapath and control work properly, you'll need to test it. Initially, you should hard-wire certain input patterns to your datapath to be sure that smaller components are responding properly. Once that's done, you need to write small programs to test the CPU's function overall. You should begin with extremely simple programs (e.g., add two numbers). Then move onto something slightly more complex (e.g., a simple loop).

3.1 Assembling

You should write your programs in assembly. However, you then need to assemble these programs into machine code instructions. You can either perform this translation by hand, or, if your programming skills allow it, write a simple program to perform the translation for you. If you're feeling adventurous, you could even implement a few pseudo-instructions to make the higher-level, assembly programming task easier. Doing so is not at all required, though.

3.2 Loading and executing

You must be able to load a sequence of machine code instructions into your main memory for execution by the CPU. Assuming that you load your instructions into main memory address zero, you need only then to set your PC to 0, and then let the CPU fetch, decode, and execute each instruction in turn.

4 Poorly documented aspects of `tkgate`

The `tkgate` program, while quite useful, is not as clearly and thorough documented as one might like. Below are descriptions of two of the thornier problems of using this circuit simulator. In particular, for both of these, we walk through the construction of examples that should sufficiently illustrate how to perform these tasks.

4.1 Module inputs and outputs

When creating a module, the `tkgate` tutorial shows you how to create input and output *ports*. However, it does not show you how properly to connect those ports to actual gates. Follow this sequence to see how to create a simple module where the ports are properly connected and function:

1. Run `tkgate`.
2. From the File menu, select New. Click OK to create the file `new.v`.
3. From the Module menu, select New.... Name the new module something not so clever, such as `notit`, and click OK.

4. On the left-hand side of the window, there is a list with the heading Modules. Double-click on notit. At the bottom of the window, you should see: `File: new.v  Module: notit`.

5. From the Module menu, select Edit Interfaces.... That will bring up a box, labeled notit, that represents your module.

6. On the left-hand side of that box, right-click to obtain a drop-down menu, and select Add Input.... When the Port Parameters window appears, set the Signal Name and the Port Name both to be a. Click OK. You should see the box again, now with the label a and an arrow pointing into the box.

7. Similarly, on the right-hand side of the box, right-click and select Add Output.... Set the Signal Name and the Port Name both to be y, and then click OK. You should see the box with the added label y above a little arrow pointing out of the box.

8. Once again go to the Modules list and double-click notit. You should now see, on the lower left-hand side of the window, the list Ports with $< a$ and $> y$.

9. In the middle of the window, left-click somewhere. Then, from the Make menu, select the Gate submenu, and then select Inverter. A NOT gate should appear.

10. Somewhere to the left of the NOT gate, left-click again. Then, from the Make menu, select the Module submenu followed by Module Input. In the Net Parameters window that appears, set the Net Name to a. Click OK.

11. Use the *Move/Connect* tool to solder the wire coming from a to the input of the NOT gate.

12. Somewhere to the right of the NOT gate, left-click one more time. Then, from the MAKE menu, select MODULE and then MODULE OUTPUT. In the Net Parameters window, set Net Name to y and click OK.

13. Solder the wire from the output of the NOT gate to the output wire y.

14. Under the Modules menu, double click on main+, bringing you back to the main design space.

15. Left-click somewhere in the design space. Then select Make → Modules → Module Instance. When the Gate Parameters window appears, set the Function to be `notit`. Click OK. You should see a notit box appear with the a and y ports.

16. To the left of the module, left-click somewhere. Then select Make → I/O → Switch. Solder the switch to the a input of the module.

17. To the right of the module, left-click somewhere. Select Make → I/O → LED. Solder the y output of the module to this LED.

That's it. You've defined a module (notit) and connected external inputs and outputs to an instance of it. To test it, select Simulate → Begin and then Simulate → Run. You can then click the switch to toggle its value, and the LED will always show the opposite value, demonstrating that the NOT gate inside the notit module works.

4.2 Pre-loading RAM and ROM contents

The `tkgate` simulator allows you to add RAM and ROM modules to your circuits. By default, these memory components take an 8-bit address and store 8-bits per entry, which is perfect for this assignment. The trick is to pre-load these memory modules with values from some pre-written file. To see how you can make such a memory module, pre-load its contents, and then use it, follow these steps:

1. In `emacs`, open a new file and name it, say, `test.mem`.
2. Type, as text and in the following format, a few values to be loaded into the memory:

   ```
   0/ 00 02 04 06 08 0a 0c 0e
   8/ ff ee dd cc bb aa 99 88
   ```

   In this format, the number preceeding the slash is the address (in hexidecimal) at which the next group of values should be loaded. In the example above, the values will be loaded starting at address 0. The values that follow the slash are pairs of hexidecimal digits, where each pair represents a 1-byte value. Here, I chose meaningless but distinct values that should make it easy to determine that the memory module loaded the values correctly.
3. Open `tkgate` and create a new file.
4. Left-click in the design space. Select Make $\rightarrow$ Memory $\rightarrow$ ROM.
5. Double-click on the ROM module and a Gate Parameters window will appear. Click the Details tab, and then the Browse... button. Navigate your way to your `test.mem` file and select it. Click OK.
6. To the left of the ROM, create a DIP Switch. Solder its output to the A input of the ROM.
7. To the right of the ROM, create a 7-Seg. LED (Hex). Solder the D output of the ROM to the input of that LED.
8. Below the ROM, create a `Ground` and solder it to the $\overline{OE}$ input.

To test this structure, start the simulator and run the simulation. You can double-click on the DIP switch to enter a new value (in hexidecimal) that will be presented as an address to the ROM. Given the example file above, only addresses 0 through f will produce an output. You should see, for each address, the corresponding two-digit hexidecimal value from the value corresponding to that location.

5 How to submit your work

Use the `cs16-submit` command to turn in your design. Note that only one member from each pair needs to perform this submission, and that you should label your work, clearly with text, the names of **both** members of the pair. Submit your circuit design and memory files. For example:

```
cs16-submit lab-8 cpu.v test-program-1.mem test-program-2.mem
```

This assignment is due at **5:00 pm** on **Wednesday, December 10.**