Project 2 Details

Check out the ISA here.

You may use *any* built-in Logisim component to build your CPU! You don't have to build the ALU out of 500 gates or the registers out of flip flops. Explore the stuff in the component pane... many, many things are done for you!

Just don't use any circuits someone else made, because that's cheating, duh.

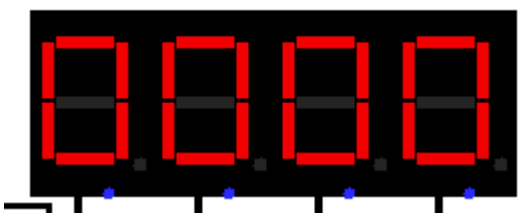
Here are the main parts of the CPU, ordered vaguely by increasing difficulty:

1. **The display unit**
2. **The program ROM and data RAM**
3. **The ALU**
4. **The register file**
5. **The return stack**
6. **The program counter control**
7. **The interconnect**
8. **The control**

---

1. The display unit

On your main circuit, place four **Input/Output > Hex Digit Display** components next to each other. If you change the colors, please make sure it's easy to read!

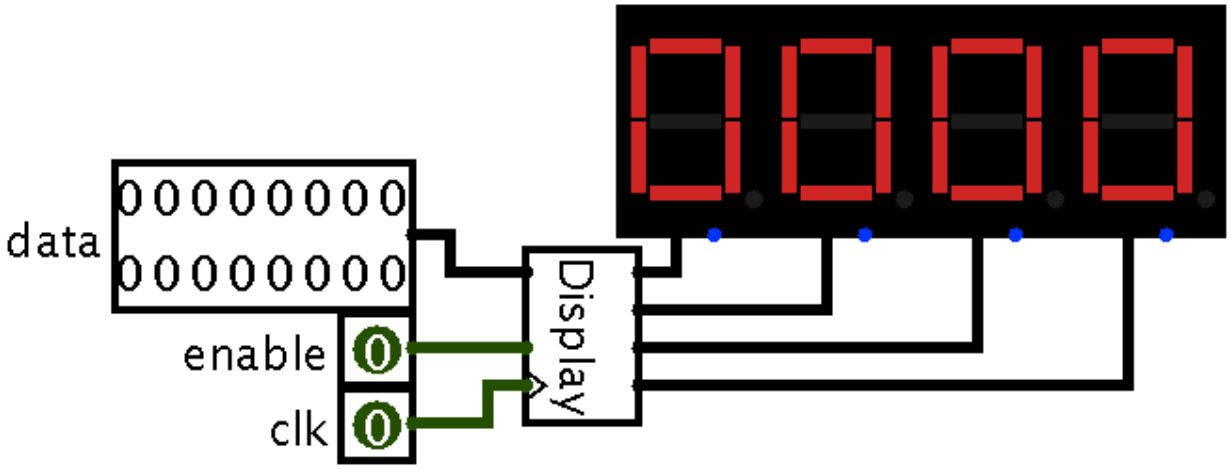


Now, to control those, **make a display unit component** like this:

- INPUTS:
  - 1-bit `Clock`

        - (that means this is a sequential circuit - it'll have a register inside!)
    - 1-bit `Write Enable`
    - 16-bit `Data`
- OUTPUTS:
    - 4 x 4-bit `Digit` values
        - do *not* put the digit displays *inside* this component!
- BEHAVIOR:
    - It **holds a 16-bit value** in a register
    - It **outputs that stored 16-bit value, split into groups of 4 bits**
    - If `Write Enable = 0`, **the register should not change.**
    - If `Write Enable = 1`, the register should **store the value of the `Data` input.**

You can test this component without any other circuitry built. Plop down the display component, wire it up to the digits, and put some inputs next to it. Poke the inputs, clock it, see if it works.



---

2. The program ROM and data RAM

On your main circuit, place a **Memory > ROM** component with these settings:

- Address Bit Width: 16
- Data Bit Width: 20

That will give you 65,536 20-bit instructions.

The ROM is super simple to use: put address (value of PC register) into the left side, get instructions out the right side.

Then place a **Memory > RAM** component with these settings:

- Address Bit Width: 16

- Data Bit Width: 16
- **Data Interface: Separate load and store ports**

> That last option is important.

This will give you 65,536 **words** of RAM. This is a **word-addressable** machine - each memory address refers to 2 bytes of data.

The address and data to *store* go in the left side, the clock in the bottom (the triangle), and the data to *load* comes out the right side.

You can ignore the `sel`, `ld`, and `clr` inputs!

The `str` input is the memory's *write enable* signal. Most instructions don't write to the memory, so you'll have to hook something up to it.

And that's it for those... for now!

---

3. The ALU

The ALU is a combinational circuit. **Remember, you're allowed to use any of the built-in Logisim components** - adders, subtractors, multipliers, shifters etc.

Make an ALU component like this:

- INPUTS:
  - 16-bit `A`
  - 16-bit `B`
  - 3-bit `Operation`
    - There are 8 possible values this can take. It's up to you which binary value means which operation, but write it down somewhere! You'll need it later.
- OUTPUTS:
  - 16-bit `Result`
- BEHAVIOR:
  - The `Result` output depends on what `Operation` is chosen:
    - Add: `A + B`
    - Subtract `A - B`
    - AND: `A & B`
    - OR: `A | B`
    - NOT: `~B` *(`A` is ignored)*
    - Shift Left: `A << B[3:0]` *(see below)*

- Shift Right: `A >> B[3:0]` *(logical, not arithmetic)*

Notes:

- the syntax `B[3:0]` means "bits 3, 2, 1, and 0 of B". It's commonly used in hardware design.
- AND, OR, and NOT gates also have a "Data Bits" property.

> Test it out thoroughly!!! It's so easy to accidentally connect things to the wrong places. Make sure it does what you expect for every operation.

---

4. The register file

The register file is pretty straightforward. It will work just like the one we looked at in class, except it has 8 registers instead of 4. Feel free to use the example circuit as well. Details:

- 1 write port (`rd`)
- 2 read ports (`rs` and `rt`)
- 8 registers, 16 bits per register
  - but register 0 is not a register - just a constant 0

There is one complication: `rd` is only the destination register *in R-type instructions*. In I-type instructions, `rt` is used as the destination register index. So, you need to choose which to use based on **whether the current instruction is I-type or not.** That's something the control unit can tell your register file.

> Test it out thoroughly!!! Do this for every component you make, ok?

---

5. The return stack

This architecture does not store function return addresses in registers *or* in the data RAM. Instead, there is a small dedicated stack RAM just for return addresses.

It implements a 256-entry stack that grows *upwards* from address 0. So if you do 3 calls in a row, the return addresses will be at stack RAM addresses 0, 1, 2, and `sp` will be 3. Then, if you do 3 returns in a row, `sp` will be back to 0.

It is a small FSM that works like so:

- INPUTS:
  - 1-bit `is call` (1 if this is a call instruction)
  - 1-bit `is ret` (1 if this is a ret instruction)
  - the clock
  - 16-bit `PC` input
- OUTPUTS:
  - 16-bit `return address`
- STATE:
  - an 8-bit stack pointer register (`sp`)
  - a RAM with an 8-bit address and 16-bit data (and "separate load and store ports" as before)
- BEHAVIOR:
  - if the `is call` input is 1...
    - store `PC + 1` into the stack RAM at the address given by `sp`
    - `sp <- sp + 1`
  - if the `is ret` input is 1...
    - load from the stack RAM at address `sp - 1` and output the RAM data on `return address`
    - `sp <- sp - 1`
  - if neither is 1, do nothing.
    - (it's not possible for both to be 1.)

---

6. The program counter control

Remember from class that the PC can move ahead by 1 instruction, or **jump** to other places, or conditionally **branch.** That's what this component is for.

The PC is a **register** which holds the address of the current instruction. Since your instruction memory uses 16-bit addresses, what size should the PC be? 😉

You could put your PC register inside or outside this component, it doesn't really matter.

There are *so many ways* to implement this. **Check out the control flow instruction list here** to see what ways the PC can change (instructions that do `PC <- ...`). The basic behaviors should be something like this:

- It needs a `Halt` input. If this input is 1, the PC is *disabled* (no longer changes).
- There are 5 places it can go:
  1. `PC + 1` (by default, or for not-taken conditional branches)
  2. `PC + sxt(imm8)` (for taken conditional branches)

3. `REG[rs]` (for `jr`)
4. `imm16` (for `j` and `call`)
5. `STK[sp - 1]` (for `ret`)

- There are also 6 kinds of conditional branches.
  - The PC control needs to know if we're doing a conditional branch, and what kind it is…
  - …and depending on how you implement it, it could do the comparisons itself, or leave that to the ALU to do.

---

7. The Interconnect

> The interconnect won't really be a component of its own. It'll be a collection of multiplexers and wires/tunnels that hook everything else together.

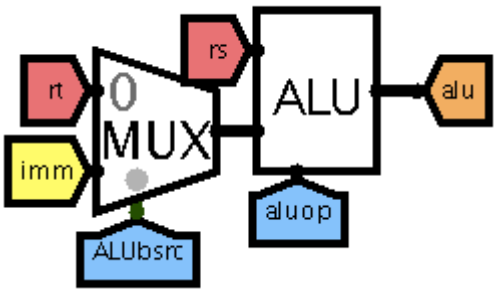
Now you can go on the main circuit, and plop down one of each of your components you made. To connect them together, you make the interconnect. Remember the "interconnect matrix" from the class slides? Think about that.

- Treat the 2 ALU inputs as separate "destinations."
- Treat the memory address and data inputs as separate "destinations" too.

Use tunnels. Tunnels will let you name your wires and move the parts of your CPU around independently.

Copy and paste your tunnels. This will avoid naming mistakes (e.g. naming one end `PC` and the other end `pc`, which are different names).

Every time you put a MUX down, make a tunnel for its select signal and name it like "___Src", e.g. "RegSrc" or "MemSrc" or "ALUSrcB". You can also use the tunnel colors to make the control signals stand out (like coloring them blue). For example, here's a tiny piece of mine:



I used red for registers, orange for the ALU, yellow for the immediates, and blue for control signals.

Testing it

What I like to do is find all the control signals (selects and write enables), make copies of all those tunnels, and gather them up in one place.

Then I can put inputs on them and "manually drive" the CPU. So I can e.g. load an immediate into a register by setting `rd`, turning on `RegWrite`, setting `RegDataSrc` to use the immediate, and then ticking the clock. Then I can set `rs` to the same register and make sure the same value is coming out.

Once you're kinda sure your interconnect is working, you can remove the fake inputs and replace them with…

---

8. The control

Your control component is a **combinational circuit** whose input is **the program ROM output** (the current instruction) and its output is ***all the control signals for the entire CPU.*** It's gonna be a complicated component.

It will *also* output the source/destination register indices and the immediate value extracted from the instruction.

There are **three main stages** to making the control:

1. **Split up the instruction.**
    - Use splitters to split the instruction into its fields.
    - Like on the slides, extract *all possible fields* from *all possible formats*.
    - You only have to extract each field once. E.g. you don't have to extract 2 copies of `rs`.
    - After splitting you should have:
        - a bit saying whether or not it's J-type
        - the R-/I-type `opcode` (5 bits)
        - the J-type `opcode` (3 bits)
        - `rs`, `rt`, and `rd` (all 3 bits, and they just go right out of the control)
        - `imm5`, `imm8`, and `imm16` (see the section below…)
2. **Decode the opcode.**
    - Since there are **two kinds of opcodes,** you will need 2 *demuxes*.
    - On the slides I used a decoder… but here you basically have 2 opcodes, and only 1 is valid.
    - What you want is for *exactly one output* of those two demuxes to be 1 at a time.
        - You never want both demuxes to output a 1.

> Try to keep the opcodes and their decoded versions *inside* the control unit. Don't send the opcodes all over the CPU and decode them elsewhere! That'll make it way harder to find and fix problems. Just like in software: the more encapsulated things are, the easier they are to work with.

3. **Come up with the control signals.**
    - Again, like on the slides: after you've decoded the opcode, it's just a matter of using OR gates and priority encoders.
    - Each multi-bit control signal will need a priority encoder, and they kind of "mirror" each other.
        - So if my `ALUOp` mux has `-` on input 1...
        - ...then my `ALUOp` priority encoder's input 1 will be all the instructions that subtract, ORed together.

Dealing with the immediates

You *could* output `imm5`, `imm8`, and `imm16` directly from your control unit and let the other parts of the CPU handle them (by extending them appropriately). I found it useful to add a layer of abstraction. I have my control choose the right immediate and the right extension based on what instruction it is, and then output a single 16-bit `immediate` value that all the CPU components use. Again, this way the control is the *only* part of the CPU that has to know the instruction.

Check out the ISA here.

Test Programs

Assembled programs for you to load into your machine to test it.

Extra Credit

Done? Bored? Try this.