Homework #4 – Processor Core Design

This homework requires you to design and implement the Duke 250/16, a 16-bit MIPS-like, **word-addressed (not byte-addressed)** RISC architecture. (A word is 16-bits.) We have specified the architecture, and you will use Logisim to design a single cycle implementation of this architecture. The architecture's instructions are specified in Table 1.

Submission instructions – please read VERY carefully:

- You must do all work individually, and you must submit your work electronically via GradeScope.

- You will submit a Logisim file called cpu.circ. This file is the circuit for your processor.

- If your CPU is failing one or more tests, you are encouraged to submit a PDF file called cpu.pdf. This file is your description of your processor, and the grading TA will use this description to help assign partial credit. (This file is for your benefit!) The file should explain the following issues:

  o What parts of your processor work and which parts do not work. This helps us to find partial credit.

  o For subcircuits (e.g., register file or ALU), explain their interfaces so that we can possibly test them individually.

- Non-functioning CPUs will receive partial credit of 10 points plus incremental credit for functioning subcircuits (up to 70 points); this credit will be guided by your cpu.pdf write-up.

- All submitted circuits will be tested for suspicious similarities to other circuits, and the test will uncover cheating, even if it is "hidden." Plagiarism of Logisim code will be treated as academic misconduct.

- Logisim implementations must use only the components specified in the "Logisim restrictions" section later in this document.

- For successful automated grading, your circuit must meet the requirements specified in the "Automated testing" section.

- You may not use any pre-existing Logisim circuits (i.e., that you could possibly find by searching the internet).

Have fun!!

The instruction set

instruction	opcode	type	usage	operation
add	0000	R	add $rd, $rs, $rt	$rd = $rs + $rt
addi	0001	I	addi $rt, $rs, Imm	$rt = $rs + Imm
sub	0010	R	sub $rd, $rs, $rt	$rd = $rs - $rt
sll	0011	R	sll $rd, $rs, <shamt>	$rd = $rs << shamt *(shamt is unsigned)*
srl	0100	R	srl $rd, $rs, <shamt>	$rd = $rs >> shamt *(logical shift: no special treatment of sign bit; shamt is unsigned)*
and	0101	R	and $rd, $rs, $rt	$rd = $rs & $rt
not	0110	R	not $rd, $rs	$rd = ~$rs
lw	0111	I	lw $rt, D($rs)	$rt = Mem[$rs+D]
sw	1000	I	sw $rt, D($rs)	Mem[$rs+D] = $rt
bne	1001	I	bne $rs, $rt, B	if ($rs!=$rt) then PC=PC+1+B
ble	1010	I	ble $rs, $rt, B	if ($rs<=$rt) then PC=PC+1+B
j	1011	J	j L	PC = L *(upper 4 bits zeroed)*
jr	1100	R	jr $rs	PC = $rs
jal	1101	J	jal L	$r7=PC+1; PC = L
input	1110	I	input $rt	$rt = keyboard input
output	1111	I	output $rs	print $rs on a TTY display

Table 1: Duke 250/16 Instructions

The formats of the R, I, and J type instructions are shown below: number of bits in parenthesis, with the specific bit numbers shown in brackets (remember that the least significant or rightmost bit is bit 0).

R-Type	Opcode (4) [12..15]	Rs (3) [9..11]	Rt (3) [6..8]	Rd (3) [3..5]	Shamt (3) [0..2]
I-Type	Opcode (4) [12..15]	Rs (3) [9..11]	Rt(3) [6..8]	Immediate (6) [0..5]	
J-Type	Opcode (4) [12..15]	Address (12) [0..11]			

Immediate values are 6-bit signed 2's complement, so you must ensure that you sign extend it.

The *input* instruction is nonblocking, which means it will always complete and write something into the destination register. After a read, $rt should be the ASCII value of the character read, zero-extended to 16 bits. If no character was available from the keyboard, $rt should become zero to indicate this. You will use the keyboard input device available in Logisim.

The *output* instruction writes the 7-bit ascii character contained in the low 7 bits of $rs ($rs[6..0]) to the Logisim TTY output device. **Please use the TTY with the following specifications: 13 rows, 80 columns, and rising edge.**

Registers

There are 8 general purpose registers: $r0-$r7. The register $r7 is the link register for the `jal` instruction (similar to $ra in MIPS). The user of your CPU may write to it with other instructions, but that would mess up function call/return for them. Users of your CPU are also advised to use $r6 as the stack pointer. $r0 is the constant value 0 (i.e., an instruction can specify it as a destination but "writing" to $r0 must not change its value).

Implementation note: Your register file's read ports _must_ use Tri-state Buffers and a Decoder rather than a big Mux (as described in the class notes regarding the register file). Logically, the two approaches are equivalent, but in real implementation, the Tri-state Buffer approach is much faster. Besides, this is a great chance to play with Tri-states. Solutions using a Mux within the register file will be penalized up to 10%.

The reset input

The processor has a single input called "`reset`"; the name must match exactly. This input resets the state of the computer by doing the following:

1. Reset PC to 0 asynchronously.
2. Clear the TTY display asynchronously.
3. Clear the keyboard input buffer asynchronously.
4. Reset the registers in the register file to all-zero asynchronously.
   NOTE: the Reset input does NOT affect instruction memory or data memory.

This can be achieved by connecting `reset` to the "clear" or "reset" input pins on the underlying D Flipflops and IO devices.

Clocking your CPU

There should only be five clocked items in your design (PC register, register file, data memory, keyboard and TTY).

CLOCK REQUIREMENT:
- Clock the register file, and TTY on the _rising_ edge of the clock.
- Clock the PC register, data memory, and keyboard on the _falling_ edge.
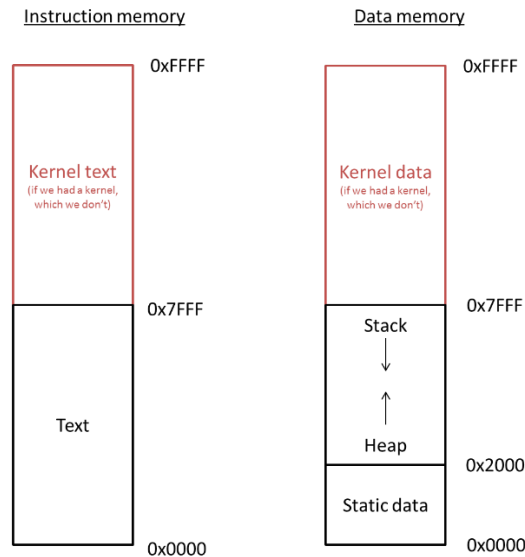
Memory layout

Figure 1: The memory model for your CPU.

The conventions for memory allocation, as performed by the assembler we provide you, are shown in Figure 1. This is what's known as a "Harvard architecture", which simply means that there is a separate memory space for instructions versus data. This maps naturally to the separate "instruction fetch" and "load word" facilities in our CPU's data path. In addition, we reserve the top half of each memory region for the kernel, even though no kernel or operating system will exist in our implementation. This means that, in instruction memory, user programs can have addresses from 0x0000 to 0x7FFF. In data memory, the first 8 Kwords (0x2000 words) are reserved for static data, with the heap starting at address 0x2000 and growing up. The stack starts at address 0x7FFF and grows down. **REMEMBER: this is WORD-addressed, not BYTE-addressed: each address leads to a full 16-bit word.**

You should use a Logisim ROM memory block for the instruction memory and a Logisim RAM block for data memory. You can edit the values in these memory blocks manually, but you can also right click (control click for Mac users) to open the popup menu that allows you to load an image file. These image files will be generated by the assembler described later in this document.

Logisim restrictions

IMPORTANT: On this assignment, you may only use the following Logisim elements:

1. Anything from the "Wiring" folder
2. Anything from the "Gates" folder
3. Anything from the "Plexers" folder
4. From the "Memory" folder: "D Flip-Flop", "RAM", and "ROM"
   Note: when deploying RAM in Logisim Evolution, you'll need to couple it with a "`memory_latch`" circuit we are providing you; see the section "Using RAM in Logisim Evolution" below.
5. From the "Input/Output" folder: "Keyboard", "TTY", and "Button".
6. The "Text" tool
7. Any sub-circuits you develop from the above

The penalty for violating these restrictions can be up to 75% of total score!

Getting started

Start by **reading this whole document**. Maybe **read it again**.

Then, as per usual, you'll clone a repo called "homework4" from git. As always, ensure this repo is set to private! This repo will provide you with the automated tester (`hwtest.py` and associated files), the example programs used by the tester (`programs/`), and the assembler+simulator (`asm-sim/`).

You can also check out the TA tips document linked next to this writeup.

Using RAM in Logisim Evolution

The model of RAM we covered in class was a little unrealistic, but Logisim Evolution's RAM is more accurately modeled: it actually takes a non-zero amount of time to process data. Hence, we need to do a bit of buffering to make it work like we learned. Don't worry – we're giving you the circuit for this:

1. Lay down your RAM in your main circuit and configure as follows:
   a. **Address** and **data width** should be **16-bit**.
   b. **Triggering** should be on **high-level** instead of on an edge.
   c. **Databus implementation** should be separate **read/write ports**.
2. Find the `memorylatch.circ` provided via git, and merge the contents your `cpu.circ`: With your CPU open, use the "File | Merge" option in Logisim Evolution to merge in the `memorylatch.circ` file. This will add a `memory_latch` subcircuit.
3. Place a `memory_latch` instance before your RAM in your main circuit. Instead of sending your data and address directly to your RAM, you should send them to the `memory_latch` instance (along with a NOT(clock) signal and a `write_enable` signal) and this latch will generate an "`address_latched`", "`data_in_latched`", write-enable latched ("`WE_latched`") and output-enable latched ("`OE_latched`") which you should then send to the RAM that is level triggered. Unfortunately, the *appearance* of the circuit can't be imported, so it shows as a big ugly box. You can use it as-is, but you could change the representation to hook into your RAM compactly, as shown below.
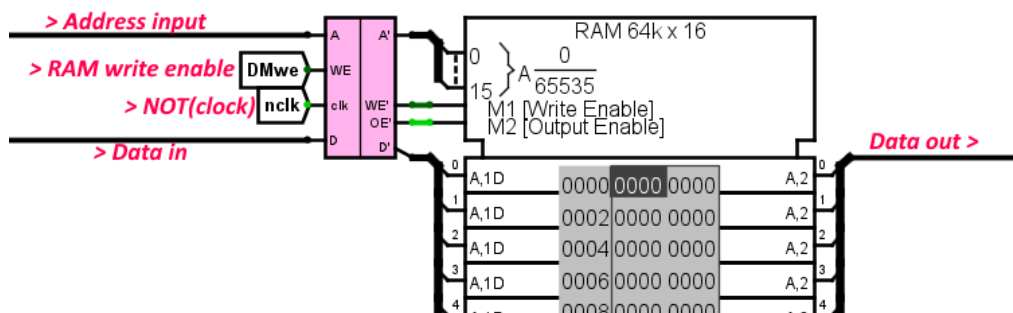
Note: A side-effect of this approach is that your sw instructions will appear to "happen" one cycle later than you expect. This is normal. Further, **any attempt to do a `lw` directly after a `sw` instruction will FAIL – this is a known limitation of this ISA, and none of the provided or grader tests programs will attempt this.** Later in the course, we'll learn that this situation is known as a *hazard*, and dealing with such things is part of modern CPU design.

"Booting" your computer

For testing, the following is the correct procedure for starting a program in your CPU:

- Load the program's instruction and data memory into ROM and RAM, respectively.
- Poke the clock so it starts **low**:
- Pulse `reset` on and off.
- If you have input to put into the keyboard, do so now.
- Fire the clock either manually ("Tick Once", Ctrl+T) or automatically ("Ticks Enabled", Ctrl+K) to run the program.

Automated testing

An automated self-test tool has been provided. **For the self-test tool to work, your circuit must meet the following requirements:**

- Circuit is called `cpu.circ` and is stored in the same directory as the test tool and associated data.

- You must name your register file component "`RegisterFile`" (case sensitive).

- You must name your reset input "`reset`" (case sensitive).

- Testing is based on the Probe component. You must place a Probe on each register in your register file and name these probes "`r0`", "`r1`", "`r2`", etc.

- Make sure that the default state of all DFFs is 0 (i.e. that you don't leave a DFF inside a register 'poked' to a 1 value when you save). Most of the tests toggle the reset line to ignore this issue, but the `io` test cannot, as that would otherwise reset the keyboard buffer.

- You may use Probes for your own purposes, but only if you leave their label *blank*. The tester filters out unlabeled probes, but any labeled probes other than "`r0`", "`r1`", etc., will throw off the results.

- Configure the TTY with the following specifications: 13 rows, 80 columns, and rising edge.

- You may not use a ROM component for any purpose other than your instruction memory, as the console automation will overwrite every ROM component in your circuit with the instruction data.

- The tool has been tested on the Duke Linux environment, but any environment with a functioning Python and Java 1.6/1.7/1.8 should work. This means the tester should work on home Mac, Linux, and Ubuntu-on-Windows environments, provided you have a Java Runtime Environment 1.6.x, 1.7.x, or 1.8.x available. See the directions on Homework 3 for more details.

The self-test tool is similar to those you've used already. You can run "`./hwtest.py`" to see a usage message. It produces "`*_actual_*.txt`" and "`*_diff_*.txt`" files so you can see your output and the differences between that and what was expected.

The Duke 250/16 assembly language source code for these tests is in "`programs/`". See the assembler and simulator section later in this document.

If you want to run an individual command line test manually, you can use the Logisim command-line version directly. See `settings.json` for the test configuration, most notably the args. For example, the "simple" test has these arguments:

```
"args": [
    "-c", "10",
    "-ic", "0,reset=1:1,reset=0",
    "-lo", "tests/simple.imem.lgsim",
    "-la", "tests/simple.dmem.lgsim"
],
```

This test can be executed manually by applying those args on the command line as follows[1] – note that is all one command:

```
java -jar logisim_ev_cli.jar -f cpu.circ -c 10 -ic 1,reset=1:2,reset=0
        -lo tests/simple.imem.lgsim -la tests/simple.dmem.lgsim
```

A note on the philosophy behind providing this tester: the goal here is to help you determine any bugs you might have missed and *supplement* your testing effort. Staring at diff files from a test you do not understand will NOT help you debug your circuit. It is expected that you'll need to develop your own specific tests using the assembler and simulator described below.

DON'T JUST RUN THE TESTER OVER AND OVER AND TRY CHANGING THINGS TO MAKE IT TURN GREEN. To be successful, you'll likely need to read assembly source code, write and assemble your own test programs, trace simulation of these with the included simulator, and manually load and step through them in your CPU in Logisim. *Guess and check won't work. You will need to <u>actively</u> debug this thing!*

See "Tips for debugging the CPU" later in this document for guidance.

---

[1] If on a local Mac, you may need to replace "`java`" with
"`/Library/Internet\ Plug-Ins/JavaAppletPlugin.plugin/Contents/Home/bin/java`". If this doesn't work, you can list other javas listed in the output of "`which -a java`", or visit office hours for help.

The Assembler and Simulator

We are providing an assembler and a simulator for you to generate test programs and to verify your program's behavior. The assembler and simulator are included in the starter kit obtained via git. These are very limited tools (e.g., no hex values for constants - only decimal integers). We have tested the assembler on the Duke Linux machines. If you have trouble compiling and running these tools locally, you will have to run it on a Duke Linux machine and copy the generated memory image files to your own machine.

The simulator is useful for debugging your design. Note that using the verbose flag of the simulator will spit out every instruction executed as well as the correct contents of every register—this is very helpful during debugging.

There are two pseudo-instructions available for use in your programs:

1. `la $rd, label    # load address`
2. `halt`

The `la` pseudo-instruction is converted into multiple actual machine instructions that have the effect of loading a 16-bit address into the specified register (specifically, a series of `addi` and `sll` instructions). Specifically, the transformation is that:

```
la $rd, ADDR
```

Will become the following, where the bracket notation indicates bits within ADDR:

```
addi $rd, $r0, ADDR[15..11]
sll  $rd, $rd, 5
addi $rd, $rd, ADDR[10..6]
sll  $rd, $rd, 5
addi $rd, $rd, ADDR[5..1]
sll  $rd, $rd, 1
addi $rd, $rd, ADDR[0]
```
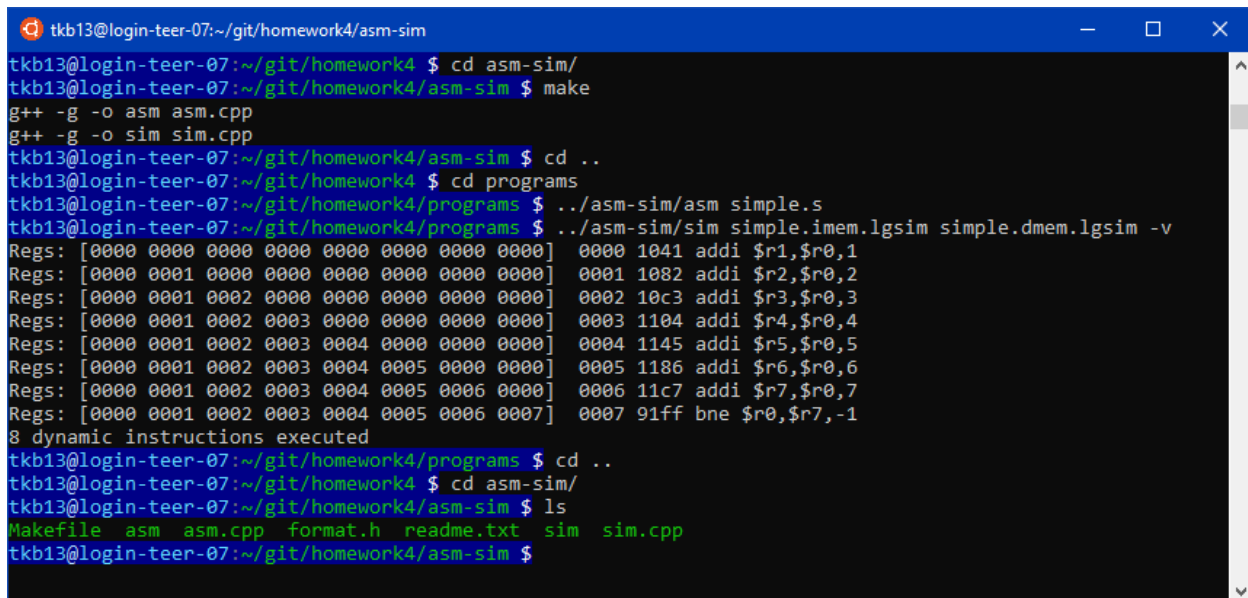
The `halt` instruction is actually a branch that simply branches back to itself, creating an infinite loop (though when run with the simulator, this special branch is detected and causes the simulator to terminate). The `halt` instruction is actually assembled into:

```
ble $r0, $r0, -1
```

For information on using these tools, see the readme.txt included with it!

Below is a screenshot of these (1) compiling these tools using `make`, (2) assembling the included `simple.s` test, and (3) simulating this assembled program.

What you don't need to worry about

There are many aspects listed above that don't actually affect your job as the CPU architect. As a result, you don't need to worry about:

- Stack management – the stack is a convention maintained by programmers writing code for your CPU; you don't have to do anything to make it exist. This means that even though we've said that $r6 is the stack pointer, you as the CPU designer don't have to do anything special to allow or enforce this.
- Heap management – same as the stack; it's maintained by the programmers so you don't have to do anything to make it exist. This means that even though the heap is supposed to start at 0x2000, you as the CPU designer don't have to do anything special to allow or enforce this.
- The kernel – there's no OS kernel for your CPU, and user programs running on your CPU will have direct access to the I/O devices (keyboard+TTY), so you don't need to worry about inventing syscalls, protected instructions, exceptions, etc.
- The "Harvard architecture" (separate instruction and data memory spaces) will happen naturally if you simply design the CPU in the way we described in class. If this were a "von Neumann architecture" (a single flat memory space for code+data), then you'd just add some multiplexers to choose between the instruction ROM and the data RAM based on the high bits of the address.

Tips for building the CPU

- You should break this project into smaller manageable chunks. You may want to design separate subcircuits (use the ADD Circuit option from the Project menu) for 1) ALU, 2) Instruction Decode, 3) Register File, 4) Next PC computation. Logisim has some documentation for subcircuits.
- Write some very simple test programs that test each instruction or incrementally include more instructions. Start with ALU ops, then memory, then branch and jumps. This will make debugging much easier.
- Think carefully about how you route wires around the circuit, keep things as neat and regular as possible else debugging gets very difficult. Tunnels are your friend.
- You will use a lot of the splitter wiring component, it can be used to both split off wires and to bring wires together to create a bundle.
- The constant wiring element is your friend, use it where you can; don't use inputs as constants!
- Register zero doesn't need to be a "real" register (no DFF-based storage is needed for it); it's hard-wired to zero.
- There will be a lot of multiplexers. No MUXes should need an enable in your design, so you can set the properties of the MUX to disable that.
- Instruction memory ROM should be set to have 16 address bits and a data width of 16 bits.
- Data memory RAM should be set to have 16 address bits and a data width of 16 bits. Be sure to attach the memory_latch described earlier.
- Remember that nearly all Logisim components have properties that allow you to change the input and/or output widths, etc. Use that to your advantage.
- There should only be a five clocked items in your design (PC register, register file, data memory, keyboard and TTY). **CLOCK REQUIREMENT**: clock the register file, and TTY on the _rising_ edge of the clock; clock the PC register, data memory, and keyboard on the _falling_ edge.
- If you have a component that should be triggered on a falling edge, but it naturally is triggered by a rising edge, you can fix this by just applying a NOT gate to the clock going into it.
- Use the ability to configure a subcircuits appearance to make it easy to understand, with a known shape and well-labeled inputs.
- You don't need to make circuits tiny – use the whole canvas to avoid getting tangled later.
- **If you don't understand any part of these directions or the usage of the tools provided, ASK. Attempting to "muddle through" is a known killer of students on this assignment.**

Tips for debugging the CPU

- For working with strings, don't forget to check an ASCII table.
- Use the "probe" feature to see what values wire bundles have at different points during execution, and you can present them in any format you wish (binary, hex, decimal, etc.).
- Note that the opcode is 4 bits, and a single hex digit is also 4 bits, so the first hex digit of the instruction *is* its opcode. This will help you keep track of what instruction you're on. The simulator's output in verbose mode also helps with it.
- When debugging, use the single Tick feature or "Poke" the clock to cause it to transition (note: you need two pokes for a full clock cycle). You can even do this from inside a sub-circuit using the "Tick Once" (Ctrl+T) option.
- When you execute programs with many instructions you can use the simulate feature to have the clock tick at a specified frequency. You'll want to do this for the demo programs provided since they execute thousands of instructions.
- You can use the "Poke" tool to double-click a subcircuit, thus *entering* it. NOTE: Because you can have multiple of a given subcircuit in use, you should use this method as opposed to just picking the subcircuit in the design tree on the left. For example, you'd want to know what's going on with *this* adder, not just *an* adder.

Here is a recommended testing procedure for what to do if you fail a provided test case:

- **OBSERVE FAULT**: If the tester is failing, observe the `actual` vs. `expected` files (summarized in the generated `diff` files). Identify what register/RAM/TTY was wrong and when. If you don't know where these are, STOP and focus on that, getting help as needed.
- **SET EXPECTATIONS**: you must know what you're *supposed* to be seeing; you cannot debug something you do not understand! You can get this from mentally tracing the *written assembly code* that every test is based on, or by running it with the provided *simulator*. If you don't know what to expect of each part of your circuit, STOP and focus on that, getting help as needed.
- **STEP**: The tester files will tell you on what cycle something is differing. Manually load the imem/dmem files, start your clock *low*, pulse the reset input, then step the CPU until the cycle before the wrong one. Step to the first half of the cycle that will screw up, and observe the state: you should see the "bad" data on a wire headed to the register, RAM, or TTY that ails you.
- **TRACE** it back – why is the data that way? Either you will <u>directly</u> find a fault (like the ALU giving the wrong answer) or you will find an <u>indirect</u> cause (e.g., it turns out we jumped to the wrong instruction).
    - In the case of a <u>direct</u> fault, trace further back until you see a place where "right" values are going in and "wrong" values are coming out. If the issue lies in a circuit, use the poke tool to enter it and trace wires similarly in there. Eventually you'll drill down to a simple case of "this gate/circuit is wrong" and you'll fix it directly.
    - In the case of an <u>indirect</u> cause, see if you can determine how you got into that situation from the current circuit state. If you can't, note the incorrect state (e.g., a wrong PC value) and re-run the trace, this time focusing on how you got into *that* incorrect state (e.g., when PC changed unexpectedly). Again, you'll either wind up with a direct cause (then you can see and fix) or indirect cause (and you can then trace to figure out *that*). Ultimately, you *will* find the direct cause.
- **If you apply this process and cannot find the root cause, come to office hours!**

Showing off

When your CPU is passing all tests, you should be able to run the included *demo* programs:

- `demo-fib-print.s`: Uses an iterative approach rather than recursive, and actually *prints* the resulting numbers to the TTY. In contrast, the `recurse.s` test performed by the tester just returns fib(4) into a register.
- `demo-prime-print.s`: Computes prime numbers, which is no simple feat for a CPU with no divide instruction!

These programs do not come pre-assembled; you'll need to use the included assembler tool to produce imem and dmem files. Running them at a high clock speed will make your CPU pulse and writhe as it cranks out number after number to the console.

This could be YOU!

You're welcome to look inside. They include code to convert integers to decimal strings and print such strings character by character, as well as a routine to perform integer division and modulo using successive subtraction. (Thanks for reading everything in this doc. For extra credit, put a picture of a fat dog in your cpu.pdf document.)

Note: Due to the size/complexity of these programs, they aren't suitable for testing a CPU that's not working yet. In the unlikely but possible situation where you pass all provided tests but can't crank out numbers to TTY, I recommend comparing execution to the result of the simulator when run in verbose mode (-v).