

# ← Project 2: Tests

To run these tests, right click and download the link to save it as a rom file. Then, in your main circuit, right click your program ROM component and choose "load image". Then open the rom you want to run. Reset, save, and run it. The shorter programs are easier to test by single-stepping the clock.

**Also, these files are just text files** - you can open them in a text editor to see the source code on the right, as well as the PC for each instruction, so you can figure out where things go wrong.

Each test assumes the previous tests work properly. If you can't get one test working, and then try further tests, they will probably malfunction too, so don't waste your time on those. Try to get the earlier tests done first. Correct implementation of a few instructions is worth more points than partial implementation of several instructions.

---

## 0. halt

The simplest program ever: this should halt immediately.

The PC should not increment, and the HALT LED should turn on.

**If it doesn't work:** your control unit should be sending a halt signal to the PC control, and the PC control should prevent the PC from changing when the halt signal is 1.

---

## 1. PLEASE GET THESE WORKING: li and put

This is a short 4-instruction program, so it's easiest to see if it's working right by ticking the clock manually (ctrl+T or cmd+T).

This should do three things:

- load the value `0x53` into `r1`

- display `0053` on the numeric display
- clear the display back to 0 by displaying `r0`

**Importantly, the value should stay on the display for a full clock cycle.** If it instantly disappears, or never even shows up, you have the display unit logic wrong.

**If it never displays anything,** check:

- that the register actually holds 0x53
  - double click on the *register file component on the main circuit* and have a look
  - if it doesn't, `li` is broken.
- if it does, something's wrong with the way the display works.

---

## 2. ALSO IMPORTANT: Checking the register file

This checks to see if all 8 registers can be written to and read from.

It will "write" to `r0`, which should do nothing.

This should appear to do nothing for 8 instructions, and then display the numbers `0011, 0022, 0033...` up to `0077`, then halt.

**If you see `00FF`,** you implemented `r0` wrong. It's a constant 0, not a register.

**If you only see `0077` at the very end,** your write enable circuitry is messed up. Only *one* register should be written to at a time. Don't use a demux for the data, use it for the write enable.

If there are missing numbers, or they come out in a weird order, step through and see what's going into the registers after each `li` instruction. Be sure to **double click on the component in the main circuit** and not the name on the left side.

---

## 3. The immediate-loaders: `li` and `lui`

`li` should load its 8-bit immediate into the *lower* 8 bits of the register, and `lui` should load its 8-bit immediate into the *upper* 8 bits.

Furthermore, `li` should do **zero** extension.

This should display:

- 0082
- 5300

If the first number is FF82, you did sign- instead of zero-extension. Some instructions *do* use sign-extension, but not `li`.

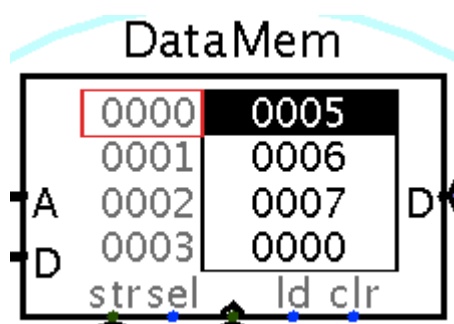
## 4. RAM: `ld` and `st`

Let's see if the RAM works. The display should stay 0 for a while, then display 0005, 0006, 0007, then halt.

- The first 3 instructions load immediates into `r1, r2, r3`;
- the next 3 store into memory locations 0, 1, 2;
- the next 3 load into `r4, r5, r6`;
- and the last 3 display `r4, r5, r6`.

That should give you an idea of where things are going wrong, if it doesn't work.

At the end of the program, use the hand tool to click the address column on the RAM and type 0000. It should look like this:



If, instead, the values 1 and 2 are stored at addresses 6 and 7, you probably swapped the address and data.

If your RAM looks correct, but it displayed all 0000, look in your register file after the program halts. `r4` through `r6` should hold 5 through 7. If they don't, your `ld` is wrong.

## 5. More tests of `ld` and `st`

This one tests the address calculation of these instructions. Basically it does:

- `r1 = 5`
- `r2 = 6`
- `MEM[r1 + 0] = r1` address 0x05 should now hold 5

- `MEM[r1 + 31] = r2` address 0x24 should now hold 6
- `r4 = MEM[r1 + 0]` r4 should now hold 5
- `r5 = MEM[r1 + 31]` r5 should now hold 6
- `display r4` should show 0005
- `display r5` should show 0006

If the addresses are totally off and it's accessing some address like 0xFFE0, you are sign-extending the immediate instead of zero-extending it.

## 6. Basic ALU: `add`, `sub`, `and`, `or`, `not`, `shl`, `shr`

This program is a bit longer, and will display 6 different numbers. They will appear in this order (the things in the parentheses are what produced that value):

- 0096 ( `add: 30 + 120` )
- 005A ( `sub: 120 - 30` )
- 0018 ( `and: 30 & 120` )
- 007E ( `or: 30 | 120` )
- FF87 ( `not: ~120` )
  - note: it should NOT be the value coming out of the `rt` read port, not `rs`
- 01E0 ( `shl: 120 << 2` )
- 001E ( `shr: 120 >> 2` )

If **all** the answers are wrong, maybe there's something up with the way you come up with the ALU Operation control signal.

If **only one** is wrong, maybe that part of your ALU needs a look.

## 7. Immediate ALU: `adi`, `sbi`, `ani`, `ori`, `sli`, `sri`

Again, this will display several numbers:

- 0050 ( `adi: 50 + 30` the output is hex, so decimal 80 is 0x50 )
- 0014 ( `adi: 50 + (-30)` )
- 0050 ( `sbi: 50 - (-30)` )
- 0014 ( `sbi: 50 - 30` )
- 0012 ( `ani: 50 & 30` )

- `003E` (`ori: 50 | 30`)
- `bEEF` (`lui` and `ori` together)
- `00C8` (`sli: 50 << 2`)
- `000C` (`sri: 50 >> 2`)

If they all fail, maybe your data going into the ALU's "B" input is wrong. It should be choosing the immediate value, not the register value.

**Pay attention to whether the immediate should be sign- or zero-extended!** This is especially important for `adi`, `sbi`, `ani`, and `ori`.

## 8. `j` (infinite loop)

This should loop infinitely, displaying an increasing count on the display (`0001`, `0002`, `0003` etc. forever). (Try doing Simulation > Enable Ticks and turn up the tick frequency... wheeee!!!)

The PC should go `0, 1, 2, 3, 1, 2, 3, 1, 2, 3...`

If it doesn't, something is wrong. ;)

## 9-12. Conditional branching 1: `blt`, `bge`, `beq`, `bne` (click the bullet point links)

These are all **for loops** which should output a sequence of numbers and then halt. Here are the tests and the numbers they should output:

- `blt`: `10, 11, 12, ... 19, 1A`, halt
- `bge`: `1A, 19, 18, ... 11, 10`, halt
- `beq`: `0, 1, 2, 3 ... 8, 9`, halt
- `bne`: `0, 1, 2, 3 ... 8, 9`, halt

If they're behaving strangely and the PC is going all over the place, make sure you are calculating the branch target correctly. It's *current* PC value plus the *sign-extended* immediate.

If the loops end one number too early or too late, or they never end, well, figure it out 😊

## 13-14. Conditional branching 2: `bez`, `bnz`

Again, these are for loops which should output a sequence of numbers and then halt.

- `bez`: 9, 8, 7, 6 ... 1, 0, halt
- `bnz`: 9, 8, 7, 6 ... 1, 0, halt

## 15. `call` and `ret`

This program should output 1, 2, 3 and then halt. It does the equivalent of:

```
f1();
put(1);
f1();
put(2);
f2();
put(3);
halt();
```

```
void f2() { f1() }
void f1() { }
```

The PC should go in this sequence (this is in **decimal**):

0, 1, 2, 3, 12, 4, 5, 12, 6, 7, 10, 12, 11, 8, 9 (halt)

If it goes like 0, 1, 2, 3, 12, 0, 1, 2, 3, 12, ... in an infinite loop, that seems like maybe the return address isn't getting pushed or popped correctly.

If it goes like 0, 1, 2, 3, 12, 3, 12, 3, 12, ... in an infinite loop, you forgot to add 1 to the return address.

If it does something else... figure it out ;)

## 16. Computed function calls with `jr`

This program should output 1111, 2222, 3333, and then halt. It does something like this:

```
for(i = 0 to 2) {
    switch(i)
    {
        case 0: func1(); break;
```

```

        case 1: func2(); break;
        case 2: func3(); break;
    }
}
halt()

void func1() { put(0x1111) }
void func2() { put(0x2222) }
void func3() { put(0x3333) }

```

except the function calls are done by shifting `i` left by 2 and adding an offset, treating the functions as if they were an array. Then it uses `jr` to jump to that calculated address.

## 17. Recursive function

At last: a real program! If everything else worked until now, this *should* work. It will push a bunch of things on the call stack, then pop them, and then it should output `0037`.

This program will take a while to run, so set the tick frequency higher and let it run. When run at maximum speed (4KHz), this should finish within a few seconds. If it's going for several seconds at max speed, maybe it's going into an infinite loop...

It does something like:

```

put(sum(10))
halt()

int sum(int x) {
    if(x == 1)
        return 1;
    else
        return x + sum(x - 1);
}

```

`sum` is a recursive function that calculates the sum of integers from 1 to n. So, `sum(10)` should give 55, or `0x37` in hex.

It uses `r7` as a "data stack pointer." The data on the stack should be stored at memory addresses `0x2F`, `0x2E`, `0x2D...`. At the end of the program, you

can right-click the RAM component, click "Edit Contents", and it should look like this:

```
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0020 0000 0000 0000 0000 0000 0000 0000 0002 0003 0004 0005 0006 0007 0008 0009 000a
```

## 18. Finding primes

This is a prime-number-finder. It runs forever, and will output the sequence of prime numbers: **2, 3, 5, 7, B, D** ... well, it's in hex, so they'll look a little funny, but hey.

The bigger the numbers get, the longer it will take to find them. Even at 4KHz, it will take several seconds for each. It's a pretty stupid software division algorithm!!

## 19. Integer square root

This computes the "integer square root" (basically the floor of the square root of **x**) for all integers starting at 1. The output will be **0, 1, 2, 3, 4...**, but the numbers will take longer and longer to change. Think of it like the graph of the square root function, but rounded to integers!

This one makes use of many different instructions, so if it works, you can be pretty confident that you've got a 100% 🏆

Now run all the tests again, just to be sure you didn't break earlier instructions when fixing later ones!

Below are the tests for the extra credit instructions.

## EC1. Keyboard (in)

Set your **Simulate > Tick Frequency** to 4 KHz. Then run this program, and with the hand tool, poke the keyboard. You'll see a blue ellipse surround it.

Now you can type hex digits (0-9, A-F) and they should show up on the hex digit display!



I mean, they should. If they don't, it's broken. :B

---

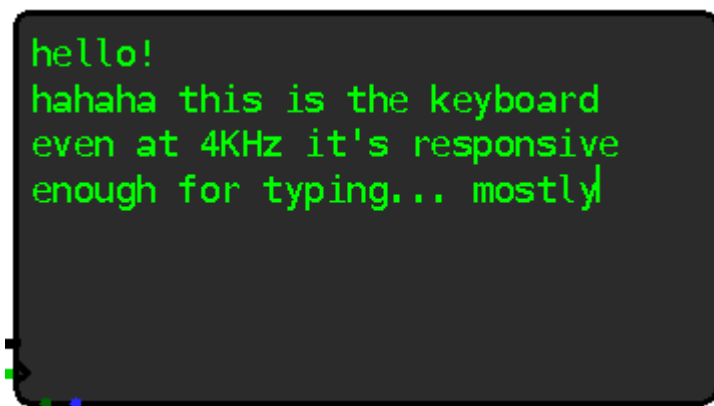
## EC2. TTY (out).

Set your **Simulate** > **Tick Frequency** to 4 KHz. Then run this program. It should show **hello!** on the TTY like so:



```
hello!  
|
```

If you implemented the **in** instruction, you can now click on the keyboard and type, and whatever you type will be echoed to the screen!



```
hello!  
hahaha this is the keyboard  
even at 4KHz it's responsive  
enough for typing... mostly
```

© 2016-2018 Jarrett Billingsley