CSE 352 Laboratory Assignment 7

Y86 Version 3: The full Y86 instruction set

Read the Y86 slides(p16-p33) and entire lab assignment carefully before you begin working on it!

Overview

You will now add the remaining Y86 instructions to implement the full Y86 instruction set. These last instructions all have to do with reading and writing data to the memory. Up to now, we only used the memory for instruction. We will now read and write data to memory and add instructions that deal with the stack.

In particular, you will add the ability to handle the following instructions:

RMMOVL and MRMOVL: moving data between registers and an external memory address
PUSHL and POPL: pushing register data onto the external memory stack, and popping it off
CALL and RET: pushing a return address onto the stack and jumping, and then returning.

You should have enough experience now implementing instructions, but let's recap briefly how you should proceed: For each instruction, describe exactly what your data path needs to do to implement that instruction. That is, what registers should be read, how data signals need to be connected, etc. In some cases you will have to add multiplexers to handle new connections that need to be made, and you will have to add control signals to control those multiplexers. You will have to update some of your modules as well to deal with the new instructions. However, at this point you should not need to add any new modules to your CPU except multiplexers.

We will give you some pointers and tips, but mostly the design will be up to you. We strongly encourage you to follow the incremental design outlined in the instructions, whereby you make a few changes and make sure they work before proceeding.

0. Getting Started

As usual, you should start a new design that is a copy of the design you had for the y86v2. It is **highly recommended** to make an **empty** design and add all the files from your y86v2 design recompile it to get a "clean" design.

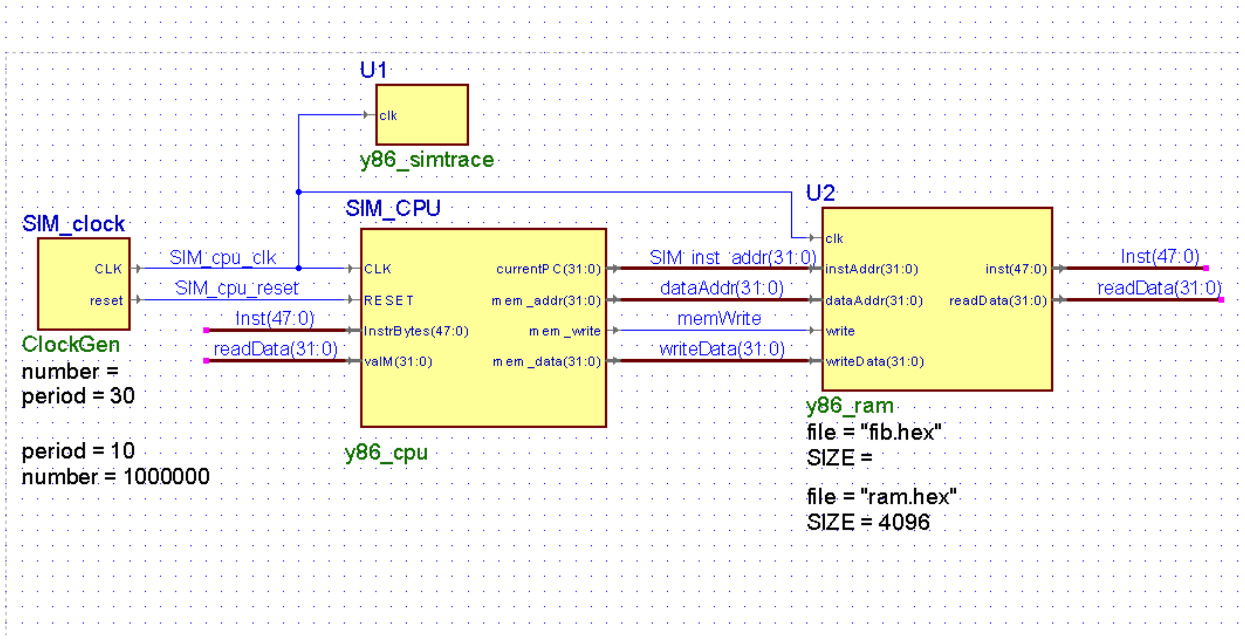
1. New memory: y86_ram

The memory we've been using so far only has one read port, for reading instructions. We will now use a new memory, `y86_ram`, that allows us to use the memory for data as well as instructions. Start by replacing your `y86_iram` module with the new **`y86_ram`** module as shown in the figure below. Connect up the wires you already have for the instruction fetch and leave the other ports unconnected. Compile and run your design using the test program for the `y86v2` to make sure it still works correctly.

2. Reading/Writing Data to Memory

We will start by implementing the `rmmovl` and `mrmovl` instructions. These will require you to connect your CPU to the data read and write port of the ram at the top level as shown in the figure. Do this by adding the following ports to the CPU block diagram:

- Data address [31:0] - Address used to read/write data to memory
- Read data [31:0] - Data read from memory
- Write data [31:0] - Data written to memory
- Write control signal. Determines whether data is actually will be written to memory



Let's start with the `rmmovl` instruction that writes data in the A register to the memory address formed by adding the B register to the constant in the instruction (you can think of B register as the base address and the constant in the instruction as the offset from the base address). First, you should use the ALU to perform this add instruction. Using ALU input multiplexers will definitely come in handy. Then all you have to do is connect the appropriate signals to the memory ports.

The `mrmovl` instruction is only a little more complicated. Note that the memory address is formed in the same way. But in this case, the value written to the register file comes from memory instead of the ALU. To make this easy, add a new "write port" to your register file for writing the memory value with a data value valM and a write register address dstM. All values written to the register file from memory use this port, so the valM will be similar to valE and dstM will be similar to dstE in terms of functionality.

Before you test your instruction, you need to update the split_instruction and controller modules to handle the `rmmovl` and `mrmovl` instructions. To do so, you will need to modify the logic for old control signals, and generate new control signals that you have added to the datapath, e.g. the `memWrite`.

Get these two memory instructions working using the unit tests in the **v3test.zip** folder before moving on to the next instructions.

3. Pushing/Popping Data on the Stack

The `pushl` and `popl` instruction also transfer data between registers and memory, but the memory address is given by the stack pointer, `%esp`, and this register must also be modified by the instruction. `pushl` subtracts 4 from `%esp` before using it as an address, while `popl` adds 4 after using it as an address. You should again figure out how to use the ALU to do this arithmetic. Note that `popl` modifies two registers, the stack pointer using the ALU value and a second register which is written with the value from memory.

You may find that mux4.v comes in handy here when expanding the inputs to your ALU to include adding or subtracting 4. You may also want to consider adding more to the Continuous Assignments block in your CPU diagram.

Test these instructions using the unit tests in the **v3tests** before you move on.

4. CALL and RET instructions

The `call` and `ret` instructions are similar to `pushl` and `popl` except that the program counter comes into play. However, you should be able to use similar logic that you used for `pushl` and `popl` to implement these instructions. To clarify more, when implementing “call” instruction the CPU should “pushl” the return address (which is the address of **next** instruction after “call” instruction) to the stack and then set the PC to the address specified in the “call” instruction. On the other hand, when implementing the “ret” instruction, CPU should “popl” the return address from the stack and set the PC to that address.

Test these instructions using the unit tests in the **v3tests** before you move on.

Take a moment to survey what you have accomplished—you have designed a fully functional Y86 processor!

5. Recursive test program

Now it's time to see if your processor can execute something a bit more complicated. Write a factorial program that uses recursion (remember to include a new line at the end of your file so it compiles correctly with the y86 simulator). At the start of the program, you should move a value into one of the registers for which you want to compute the factorial. Note that you will need to explicitly define where your stack is in your program. For example, at the end of your program, you would write:

```
.pos 0x1000
Stack:
```

This says your stack starts at address 0x1000, and it will grow to lower addresses. You'll need to initialize your stack pointer `%esp` by using the instruction `irmovl Stack, %esp` and similarly for your base pointer `%ebp`.

Recall that when you call a function, the first thing you do is push the frame pointer `%ebp` and move `%esp` to `%ebp`. Then, when you are about to return, you move `%ebp` to `%esp` and pop `%ebp`. Passing parameters to a function can be done by pushing the value on the stack before calling the function. This parameter can then be retrieved inside the function by reading a value at an offset from the frame pointer.

You want to call your factorial function from a Main function. The outline of your program should look like this.

```
#Execution begins at address 0
.pos 0
irmovl Stack, %esp
irmovl Stack, %ebp
call Main
halt

Main:
pushl %ebp
rrmovl %esp, %ebp
irmovl $11, %eax
pushl %eax
call fact  #fact(11)
rrmovl %ebp,%esp
popl %ebp
ret

fact:
<your code goes here>

.pos 0x1000
Stack:
```

Remember that to write a factorial program recursively you should pay attention to this fact:

Factorial(n) = n * Factorial(n - 1)

For this part of the lab, you should be able to compute factorial(11). Depending on how efficient your program is, this simulation could take a while.

Turn In

Please turn in the following Drop Box items.

1. Run the v3testall.ys test program on your CPU and turn in the console log of the simulation, **v3testallLog.txt**, when it completes.

2. Turn in your factorial program **factorial.ys**, along with the console log of the simulation running your program computing factorial(11), **factorialLog.txt**.

3. Archive your design as **<names(s)>_y86v3.zip** and turn it in. Please use the Archive Design command in the Design menu when preparing your design for submission. If your current design uses blocks from a previous design, don't forget to add that to your current design that you submit. When you add existing files to your design, don't forget to check the "Make local copy" box.

To make sure that the TA will be able to run your design, take the zip file you are going to submit, create a new workspace and add your design to it. Then, try running everything. If it doesn't run there, the TA will not be able to grade your assignment. So it is in your best interest to double check that it runs in a different workspace before you submit it.