Lab 7 - Y86 Version 3: The full Y86 instruction set

Due Monday, Nov. 21 at 1:30 – No late turnins accepted

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 datapath 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.

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. I prefer to make an empty design and add all the files I need and recompile so that I get a "clean" version. It is probably best to start with the files provided with the **y86v2 solution**.

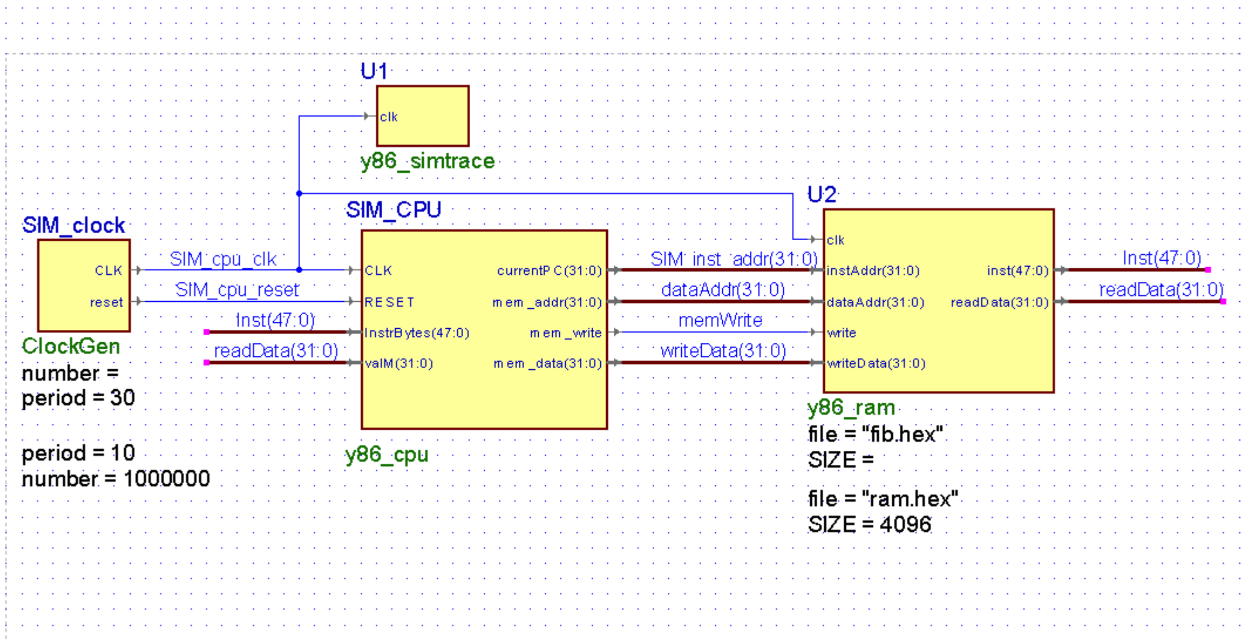
1. New memory: y86_ram

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 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. First, you should use the ALU to perform this add instruction. You will note that the y86v1 CPU that we give you makes this convenient: setting the ALU input muxes appropriately allows the ALU to perform this. 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.

Before you test your instruction, you need to update the split_instruction and controller modules to handle the `rmmovl` and `rmmovl` instructions, which includes modifying the logic for old control signals, and generating 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.

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

3. 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. For example, you should use the extra stack pointer port of the register file.

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!

4. Recursive test program

Now it's time to see if your processor can execute something a bit more complicated. Run the recursive version of the factorial program you wrote for Homework 4. 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 all the Drop Box items in a single zip file with your name.

1. Run the v3testall.ys test program on your CPU and turn in the console log of the simulation when it completes in your Drop Box zip file.

2. Turn in a printout of your factorial program (the .ys version!), along with ***just the last two pages of the console log*** of the simulation running your program, in your Drop Box zip file.

3. Archive your design using your name and turn it in your Drop Box zip file.

4. Print out and turn in all the schematics and Verilog files that you designed for this assignment. Make sure your Verilog has the appropriate number of comments. You do not have to print any schematics or Verilog files that you did not modify.