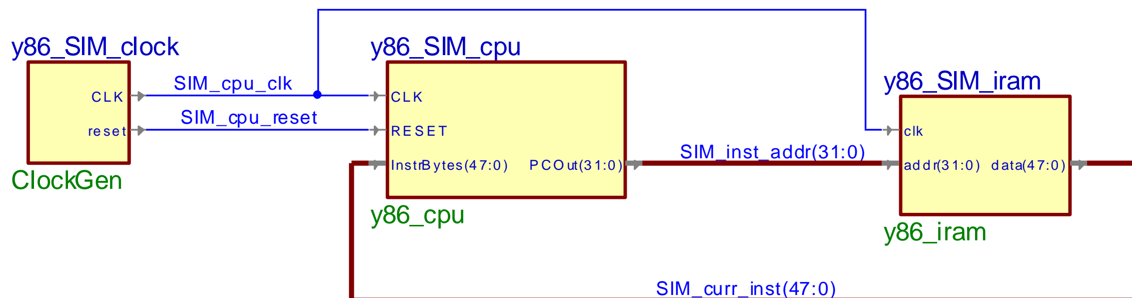
Lab 5 - Constructing the Y86 Processor Version 1

You have already designed the heart of the Y86 processor, the Arithmetic Logic Unit (ALU) and the register file. In this homework assignment you will flesh out a very basic processor by adding modules to decode a few Y86 instructions, control the flow of data between the register file and the ALU, and keep track of where the next instruction is in memory.

The figure below shows the simulation setup for your processor. Note that we are providing you with an instruction memory (we won't be accessing data in memory in this version of the processor) and the system clock, which also provides the reset signal. Your processor will generate the address of the instruction to be executed, which is returned from memory. Note that the memory returns 6 bytes since there are (i.e. there will be) instructions that are 6 bytes long. Depending on the instruction, the processor will ignore some of this data.



For this first version of the Y86 processor, you will only need to handle the following instructions: OPL (ADDL, SUBL, ANDL, XORL), RRMOVL, and IRMOVL, in addition to HALT and NOP. These instructions only use the register file. OPL instructions read values from the register file, use the ALU to perform an operation and store the result back to the register file. The RRMOVL instruction simply moves a value from one register to another, and the IRMOVL moves data stored in the instruction itself to a register.

We have provided a set of files for you to start from. Download the y86v1files.zip file and save it where you want the design to be (typically your workspace folder) on your Z: drive (your space on attu). (This zip file is also available in the courses/cse352 directory.) In Active-HDL, create a new design – you can name it whatever you like. Add the y86v1files to your new design. You will notice a set of .hex test files – you may ignore these for now.

Open the block diagram named y86_SIM_top.bde. This is the top-level of your Y86 processor design (see the schematic above), which you should not modify. Inside this schematic is the y86_cpu.bde, which contains the template of your Y86 CPU. This is where you should start to design your processor. We have made placeholder blocks for each of the modules that you should have in your design. ***You should use these files as the starting point for your design – this will***

save you a lot of time! Your job will be to design these modules, adding inputs and outputs as necessary to get all the functionality you need.

Note: You may need the constant 0 in your design. We have provided it as the signal ZERO using the Continuous Assignment block in the y86_cpu schematic.

1. `y86_splitInstruction` - Instruction Decoding

This module takes the data returned from the memory, which contains the instruction to be executed. Since Y86 instructions can be from one to six bytes long, the memory returns six bytes. The first step to determine the instruction at the beginning of these six bytes and break it into constituent parts: the instruction code and function, any register IDs, and any included constant value. This information is then used by other modules to fetch the register values, perform the appropriate ALU function, and advance the program counter by the actual length of the instruction.

1. We have provided the set of outputs that this module should provide as a guide. It is your job to write the Verilog assign statements to compute each of these outputs.
2. Although you only need to handle a few instructions for now, in coming labs you will expand this to the entire Y86 instruction set. Keep your code well organized now to help you stay sane later. In particular, define local parameters for each instruction code so that you can refer to them by name instead of a magic number. For example, if the 8-bit hexadecimal value FF represents something useful and you want to give it a good name, write this in Verilog:
   ```
   localparam GOOD_NAME = 8'hFF;
   ```
   Then you can use `GOOD_NAME` as a constant.
3. Write assign statements for icode and ifun.
4. Which instructions have a byte with register IDs? Write an assign statement for the need_regids bit.
5. Which instructions include a constant (sometimes called an "immediate") value? Write an assign statement for the need_ValC bit.
6. Assign the correct values to the rA and rB outputs. If the instruction does not address any register, then rA and rB should be the hex value F, which doesn't correspond to any register (so none of the registers will be affected).
7. Assign the correct value to the valC output. If the instruction does not include an immediate constant, set the output to 0.

2. `y86_controller` - Controller

The controller is the CPU's traffic director, setting control signals to make sure the data flows through the processor to implement the current instruction correctly. Note that the splitInstruction module only divides up the instruction into pieces; the controller module contains the logic that uses these pieces to generate the control signals. Before you design the controller, you should take a look at the register file, ALU and PC increment modules, which are discussed in the next three steps.

1. We have provided the ports for this module – you will have to figure out the logic you need to implement to generate the outputs from the inputs. These outputs are used to tell the register file what to do (which registers to read and write) and the ALU what to do.
2. We have added a halt control signal, which should go high when a HALT instruction is encountered.
3. Don't forget about the NOP instruction and make sure your control logic executes it correctly.

3. `regfile8x32` – Register File

This is the register file you designed in Homework #3. You need to add your module and wire the control inputs and the data inputs and outputs appropriately. It is useful to name the output of each individual register in the register using both number and name as follows: r0eax, r1edx, etc. as it will make debugging simulations easier.

4. `alu` – ALU

This is the ALU you designed in Homework #3, which includes the carry-lookahead adder from Homework #2. You need to wire up the control inputs and the data inputs and outputs appropriately. Note that to complete the datapath to execute all the instructions, you may need other components like multiplexers to "steer" data the right way for the instruction currently being executed. Here is a table you can use to keep track of how data should be connected for each of the instructions. Note that as you add multiplexers, you will need to use the appropriate control signals from the controller module. We have provided you a mux2 Verilog module in the y86v1files.zip that you should use.

Instruction	Operand A	Operation	Operand B
OPL		OP	
IRMOVL			
RRMOVL			

5. `y86_pc` and `y86_incrementPC` – Program Counter and PC Incrementer

The program counter (PC) is not a counter, but a simple register that contains the address of the current instruction in memory. To access the next instruction, it must be incremented by the length (in bytes) of the current instruction. This length is calculated by the incrementPC module using information from the splitInstruction module.

We have included a Verilog register for you to use for the PC. Add the appropriate inputs and logic to the y86_incrementPC module to compute the length of the current instruction. What should it do for the halt instruction? NOP instruction?

Note how the instance names have been edited to something other than U#. This makes them appear first in the hierarchical list of modules in the simulation.

6. We have provided a set of unit tests in the v1tests folder for the instructions executed by the y86v1 processor. Use these to do an initial test of your design. Use the 352 SDK to compile and simulate the assembly programs. Instructions for how to use the 352SDK are given in the SDK Startup Tutorial.
When you compile/simulate a .ys assembly program, a .hex file is generated. It is this hex file that is read by the Aldec simulator to initialize the instruction memory. After you generate the hex file, copy it to your Aldec design src folder. You can either edit the .hex file name in the top-level schematic (right click on the y86_iram module and use the Properties tab) or you can change the name of the file to "iram.hex". Either method works and which you use is a matter of taste, although keeping the file name is generally less confusing.

7. After you try the unit test programs, try out the testv1.ys program. You should get the same result in the registers after you run in the program in both the y86 simulator and the Aldec simulator.

8. Now write, compile and test a y86 program that computes the factorial function for 7, i.e. 7!

Turn In

1. Run the testv1.ys test program on your CPU and print the output and turn in a screen shot of your register file contents in the simulation when it completes.

2. Turn in a printout of your factorial program, along with a screen shot of the register file contents after it runs on your CPU.

3. Archive your design using your name and submit it electronically via the 352 dropbox. 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 (for example, homework 3 used add32 from homework 2), 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.

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.