CSE 352 Laboratory Assignment 5

Creating an ALU/Register File and Constructing the Y86 Processor Version 1

Read the entire lab assignment carefully before you begin working on it!

Please read the given test fixtures codes before you use it!

In this lab, we will begin to design a rudimentary version of the Y86 processor. Use those slides as a reference to the Y86 architecture specifications.

Note: For this lab, don't use the lib370 symbols in your schematic entry files, rather, use the built-in symbols.

Part I: Carry Look Ahead Adder Design

In this part of the lab, you will build the first component of our y86 processor: a 32-bit carry-lookahead adder. This adder will be the main part of the ALU of the y86 processor. You will use ActiveHDL to complete this portion of the lab. You can use the lab machines for this, of course, but you may also want to install the tools on your own computer. You can follow the instructions on how to install ActiveHDL on your machine here.

We will do this design one step at a time, from the "inside-out", aka "bottom-up". You will find test fixtures for each of the components and the final circuit in this zip file. (**Read the code before using**)

Extract the files in hw2files.zip. Create a new design in your workspace in ActiveHDL. Double click "Add New File" and click "Add Existing File". Navigate to where you extracted the hw2files.zip file. Select all of the files in the folder, and then ***check the Make Local Copy box***. This is very important, as your design may not work correctly otherwise. You will always want to make sure this box is checked when adding files that we give you to your workspace. Click open and the files should be in your new workspace.

Remember that before initializing simulation, you should make sure that under "Design"->"Settings"->"Simulation"->"Verilog", you have "Verilog Optimization" unchecked.

The test fixtures test for both functionality and for delay. If your test fails, you can determine whether it's a functionality or delay problem by changing the "#<number>" line in the test fixture. This tells the simulation how long to wait between changing the inputs to your circuit and checking the outputs of your circuit. If you increase this number, it will wait longer and if your circuit computes the right result but takes too long, the test will work. To pass the final test, your circuit must pass the 32-bit adder test fixture as is – the whole point of carry-lookahead addition is to make a fast adder.

Part II: ALU and register file design

ALU Design

The top-level test schematics and test fixtures for this section are in the aluregfiles.zip file located here. (**Read the code before using**)
You should start a new design adding these files (don't forget to check "Make Local Copy").

In this first part, you will design the basic components for the y86 processor that we will be implementing this quarter: the ALU and the register file.

Design the 32-bit y86 ALU which has two inputs, A and B and implements the 4 following operations:

Operation	ALU result	OP Code
addl	A + B	0
subl	B – A	1
andl	A AND B	2
xorl	A XOR B	3

Your design will need to take in a 2 bit OP code, along with two 32-bit inputs. The output of your ALU should be decided by the OP code given to it.

To create the alu Verilog module, open the alutop.bde schematic file, and double click on the "alu" symbol. A "Create New Implementation" window will appear. Select "Verilog Source Code" to create a new Verilog module. Active HDL will generate the skeleton code for the alu module.

Test your design using the provided ALU test fixture and top-level schematic (alutop). Remember to turn off Verilog optimizations under Design, Settings, Simulation, Verilog, Verilog Optimizations. Also don't forget to set your test fixture as top level in your workspace.

Register Design

Now design the y86 register file, which contains 8 32-bit registers. This register file has two read "ports". That is, it can read two different register values at the same time since the ALU needs two operands. It also has one write port: one register can be written with a new value. The 2 registers to be read and the register to be written are each specified using 3-bit

"addresses". In addition, a Write control signal is used to enable the write. If Write is not asserted, then the register file does not perform a write. Your register file will have a clk and reset input that is used to clock and reset all the registers. Reset will be used on startup to clear all the registers to 0. You should name the 8 different register outputs using the y86 register names, i.e. r0eax[31:0], r1ecx[31:0], …, r7edi[31:0]. This way it will be easy to follow the register values in the simulation as either the number or the y86 register name.

To create the register file Verilog module, open the open **rfiletop.bde**, double click regfile8x32. A "Create New Implementation" window will appear. Select "Verilog Source Code" to create a new Verilog module. Active HDL will generate the skeleton code for the alu module. Also don't forget to set your test fixture as top level in your workspace.

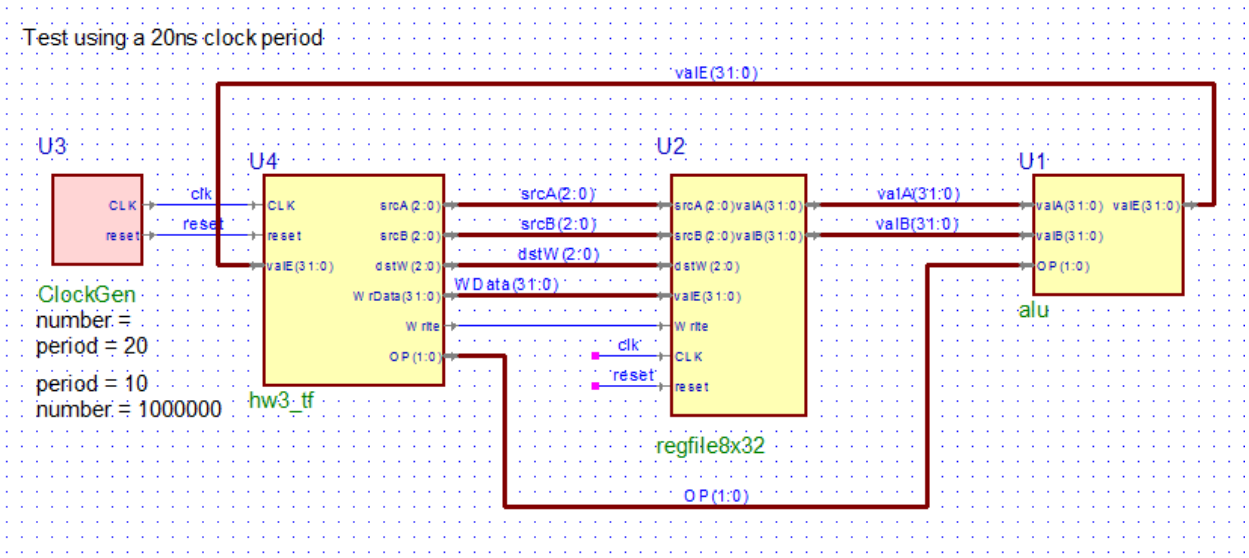
Test your design using the rfile_tf test fixture and top level schematic (rfiletop).

Simulate your "processor"

Note: The text fixture uses ClockGen block from the lib370 library. Make sure you add the library the same way you've added it in the first labs to have your design compile.

We have provided a top-level test schematic (called hw3top, ignore the naming) along with a test fixture that performs a simple test of the register file and the ALU. The test fixture specifies which registers to read and which ALU operation to perform and checks the result from the ALU. It also tests the write to registers in the register file. Note that the ALU result goes back to the test fixture and not the register file.

You can single-step your simulation by setting the simulation time step to the value of the clock period, in this case 20ns. This stops the simulation just before the clock tick, which means that you can see the outputs of all the registers, as well as the result of all the combinational logic and the input values to the registers that will be loaded on the next clock tick.



The test fixture executes “instructions” which tells the ALU what operation to perform and which registers to use for the inputs of the operation. It also tells the register file when and what to write (not the ALU result). Single-step your test fixture and write down what is executed by the test fixture.

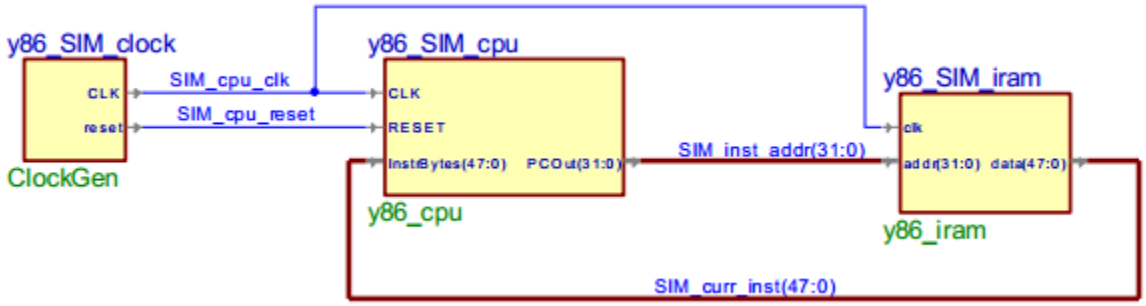
You should make a table that looks as follows. Show all the values just before the clock ticks (see above). The easiest way to do this is probably to make a waveform and include the values in the table. The registers will come from the regfile8x32 block and the others from hw3top.

Step	reset	eax	ecx	edx	ebx	esp	ebp	esi	edi	srcA	valA	srcB	valB	OP	valE	dstW	Write	WData
0																		
1																		
2																		
3																		
4																		
5																		
6																		
7																		
8																		
9																		

Part III: Constructing the Y86 Processor v.1

In part I, you designed the heart of the Y86 processor, the Arithmetic Logic Unit (ALU) and the register file. In the rest of this lab 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 the 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. 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.

Note: We recommend you to create a new workspace for part II of this lab.

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. Add the y86v1 files to your design, making sure the "Make Local Copy" box is checked. You will notice a set of .hex test files – you may ignore these for now. If Active-HDL asks if you want to overwrite ClockGen.v, select yes. Run compile all, this will generate errors (that you fix by implementing the modules), but is necessary for the block diagram(s) to properly link to the Verilog source modules contained within them.

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. Don't forget to comment your code well!

Note 1: 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. You can connect to this by name, making the name of a bus "ZERO(31:0)".

Note 2: When wiring buses in the y86_cpu schematic, make sure that you specify the bus width by naming the bus. ActiveHDL for some reason may use width of 8 as a default.

Note 3: After you implement your Verilog code inside the .v file we provided you with. Go into the y86_cpu schematic, right click on the block corresponding to your code and choose "Compare Symbol with Contents". Just click OK, and your Verilog file should be linked with the symbol.

Note 4: You will not actually implement y86_regfile or y86_alu. Simply right click these blocks, choose Replace Symbol, and choose the register file and alu that you created in the previous section.

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. NOTE: Figuring out the mapping between instrBytes and the outputs of the y86_splitInstruction module can be tricky. To help you here's how icode and ifun should be mapped: instrBytes[7:4] => icode, instrBytes[3:0] => fcode.
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 design in the previous section. 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 the previous section. 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.

Note: When running the 352Shell.bat script, don't bother specifying your code directory. Rather, copy and paste your .ys files into the 352SDK/your-code/ directory.

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!

Lab Submission Requirements

There are no check offs for this lab. Instead, you will submit your design to the CSE352 dropbox by the due date and your design will be graded based on functionality. Follow the instructions below after you have completed the lab:

1. Submit the table you filled out in Part I as table1.pdf
2. Submit a screenshot of the register file contents after testv1 runs on your CPU and name it testv1.png.
3. Submit your factorial program as factorial.ys, along with a screen shot of the register file contents after it runs on your CPU, named factorial.png.

4. Archive your design using your CSEID and submit it electronically via the 352 dropbox (example: jdoe_y86v1.zip). 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.
5. Submit your files to the catalyst dropbox: https://catalyst.uw.edu/collectit/dropbox/vlee2/29452

   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 testv1. If it doesn't run, 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.