# CSE 141L Lab 2. 9-bit CPU: Register file, ALU, and fetch unit

*Due 11:59pm Fri 5 May.*

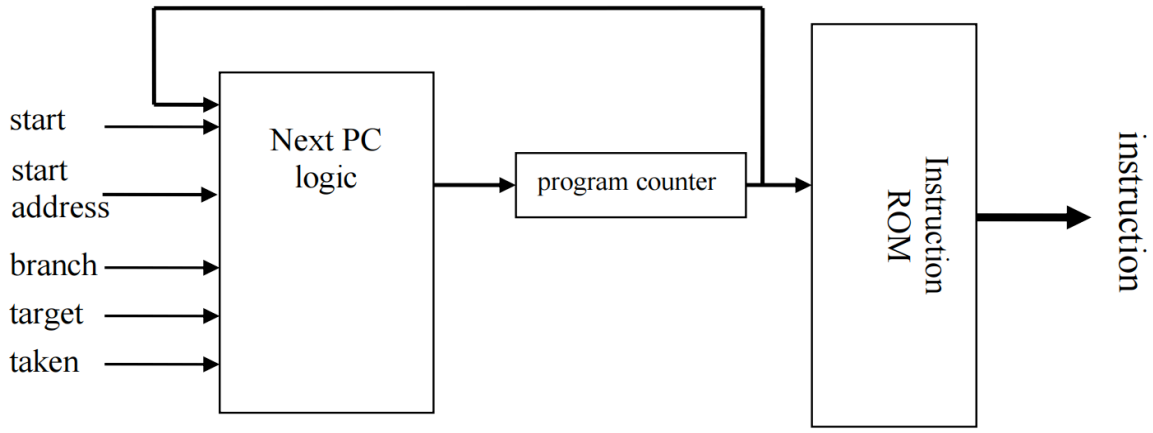
In this lab assignment, you will design the register file, ALU (arithmetic logic unit), fetch unit (program counter and instruction memory), and data memory for your CPU. Other modules of the processor (e.g. instruction decoder) will be implemented later. For this and future designs, we want the *highest* level of your design to be schematic (like the handout) *and* [System]Verilog. Anything below that can be schematic and Verilog, or just Verilog.

Everyone will use the Altera Quartus II tools, so we can compare timing between groups. Configure it for the Cyclone IVE family, device **EP4CE40F29C6**. Although we permit you to use regular "last century" Verilog, we strongly encourage the use of SystemVerilog. Many hardware designers find the combination of schematic and [System]Verilog helpful in visualizing how the design works. The Walkthrough posted on Piazza will take you through it – it's easy.

You need not connect everything together yet; you will demonstrate the functionality of each component separately through schematic, Verilog, and timing printouts. For demonstration purposes in my example, I have connected the register file and ALU – you may do the same if you'd like.

You will have an ALU and register file, possibly something like the accompanying schematic. The one shown is simple and not necessarily reflective of what you will be designing. In this picture, the ALU has two 8-bit inputs, an 8-bit output, some control bits. It also has a zero signal as an output. My register file has 8 registers (3-bit operand specifiers), can read 2 registers per cycle and write one. Registers are 8 bits wide.

The fetch unit is the logic responsible for fetching the current instruction from the instruction memory and determining the next out of the program counter (PC). It should look roughly like the following diagram:



The *program counter* is a state element (register) which outputs the address of the next instruction. *Instruction ROM* is a read only memory block that holds your code. It doesn't have to actually hold your code at this point, but it might as well – but it should hold something so we can see the effect of changing PCs. The *next PC logic* takes as input the previous PC and several other signals and calculates the next PC value. The inputs to the next PC logic are:
*start* – when asserted during a clock edge, it sets the PC to the starting address of your program. *start_address* has the starting address of your program.

*branch* – when asserted it indicates that the prior instruction was a branch.

*taken* – when the instruction is a branch, this signal when asserted indicates the branch was resolved as taken. On non-branch instructions, the next PC should be PC+1 (regardless of the value of *taken*). For branch instructions, the new PC is either PC+1 (branch not taken) or *target* (branch taken). If your branches are ALWAYS PC-relative, then you can redefine *target* to be a signed distance rather than an absolute address if you want. Make sure you tell us this is what you're doing. Otherwise, just do it as described here.

You will demonstrate each element of your design in two ways. **First**, with schematics such as the one shown, plus your Verilog code. Obviously, you must also show all relevant internal circuits. That will be further schematics and [System]Verilog code. **Second**, you must demonstrate correct operation of all ALU operations, register file functionality, and fetch unit functions with timing diagrams. An example of a (partial) timing diagram is also included; yours will be longer. The timing diagrams, for example, should demonstrate all ALU operations (this includes math to support load address computation, or any other computation required by your design), each with a couple of interesting inputs. Make sure any relevant corner/unusual cases are demonstrated. If you support instructions that do multiple computations at the same time, you need to demonstrate them happening at the same time unless you've already determined that they will happen in different pipeline stages. Note that you're demonstrating **ALU operations**, not instructions. So, for example, instructions that do no computation (e.g., branch to address in register) need not be demonstrated. There will also be a timing diagram for the fetch unit, showing it doing everything interesting. The schematics and timing diagrams will be difficult for us to understand without a *great deal* of annotation. Good organization of files and Verilog modules also helps.

The lab report will contain the following:

1. Introduction and general comments. Overview of report. (10 pts)

2. Summarize your ISA from Lab 1 (operations supported, with full detailed descriptions). (10 pts)

3. A listing of ALU operations you will be demonstrating, including the instructions they relate to. Also, a description of the required register file functionality. (20 pts)

4. Full Verilog models, hierarchically organized. (60 pts)

5. Well-annotated timing diagrams. It should be clear everything works. If your presentation leaves doubt, we shall assume it doesn't. (30 pts)

Answer the following questions:

6. Will your ALU be used for non-arithmetic instructions (e.g., address calculation, branches)? If so, how does that complicate your design? (20 pts)

7. Name one thing you could have done differently in your ISA design to make your fetch unit design job easier. (20 pts)

8. Name one thing you could have done differently in your ISA design to make your ALU design easier. (20 pts)

9. Name one thing you could have done differently in your ISA design to make your register file design easier. (20 pts)

10. What is your most complex instruction, from the standpoint of the ALU? (20 pts)

11. Now that your ALU is designed, are there any instructions that would be particularly straightforward to add given the hardware that is already there? (20 pts)