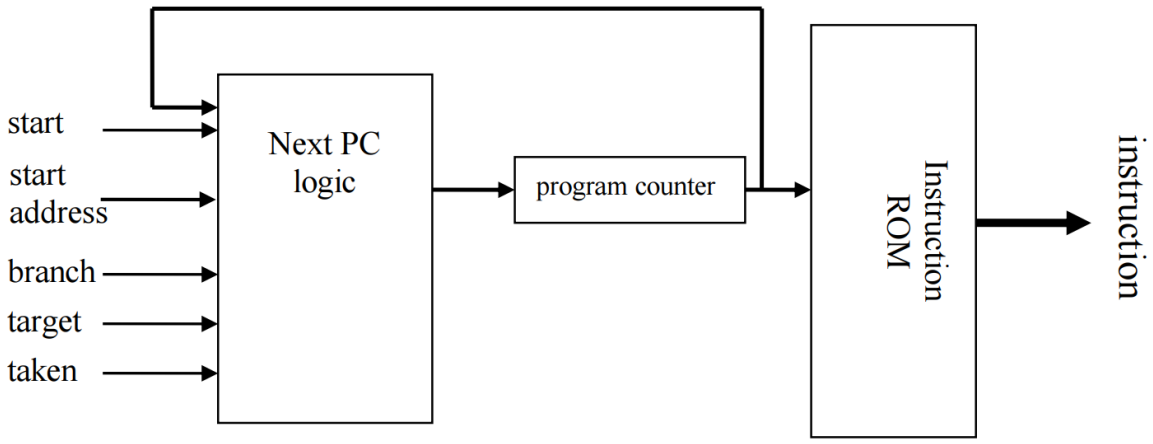
CSE 141L Lab 2. 9-bit CPU: Register file, ALU, and fetch unit

.

In this lab assignment, you will design the top level, register file, control decoder, ALU (arithmetic logic unit), data memory, muxes (signal routing switches), lookup tables, and fetch unit (program counter plus instruction ROM) for your CPU. For this and future designs, we want the *highest* level of your design to be a schematic and [System]Verilog code. You may either hand-draw the schematic or generate it using the Quartus RTL Viewer function, which I shall demonstrate in class during week 2. Anything below that can be schematic (again either drawn or generated by Quartus) and Verilog, or just Verilog. For example, to generate the in-class example, I have a project comprising several Verilog .sv files (top.sv, prog_ctr.sv, inst_mem.sv, decoder.sv, ALU.sv, reg_file.sv, data_mem.sv). The Verilog files implement the symbols included in the block diagram file. Everyone will use ModelSim for simulation and Intel (formerly Altera) Quartus II for logic synthesis in 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. The walkthrough posted on TED will take you through it – it's easy.

In addition to connecting everything together at the top level, you will demonstrate the functionality of each component separately through schematic, Verilog, and timing printouts. For demonstration purposes in my in-class example, I have connected the register file and ALU – you may make similar multimodule subassemblies, if you'd like.

The fetch unit points to the current instruction from the instruction memory and determines the next out of the program counter (PC). It should look something like the following diagram:



The *program counter* is a state element (register) which outputs the address pointer of the next instruction. *Instruction ROM* is a read only memory block that holds your 9-bit machine code. It doesn't have to hold your actual code (generated in Lab 3) yet 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 (e.g., 0) 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 – [optional – more on this in week 5 lecture] 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. (*Note: ARM and MIPS increment their respective PCs by 4, simply by convention because their machine codes are 32 bits = 4 bytes wide. We'll just increment by 1, for each 9-bit value of our machine code.*)

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 with further [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 will be demonstrated in class; 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. 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 (increment, absolute jump/branch, conditional jump/branch, etc.). 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 comprise:

1. Introduction and general comments. Overview of report.

2. Summarize your ISA from Lab 1 (general type; operations supported, with full detailed descriptions).
We have three types of instructions: R, I, and B instruction, and there is a type bit in the very beginning to indicate the type of each instruction.

I type instruction:
The type bit is 0. The rest of the 8 bits are all used to indicate an immediate. This type is to store the immediate
into the accumulator register r0.
Example: 0 1111 1111, (set 255). The first bit is type bit, and the rest 8 bits are the immediate (255), which means the
value 255 is stored into accumulator r0.

R-type instruction:
The type bit is 1. The rest of 8 bits which take 4 bits represent as opcode, and the other 4 bits assigned for operand.
Example: 1 0000 0001, (move r1). 0000 is the opcode of "move", and 0001 is a register which is r1.

Branch instruction:
Jump to the related label
Example: load r1
assign r2
load r3
beq r2 label // if r2 == r3, branch to the label

3. A list of ALU operations you will be demonstrating, including the instructions they are relevant to. Also, a brief description of the register file functionality is needed.
ALU operations:

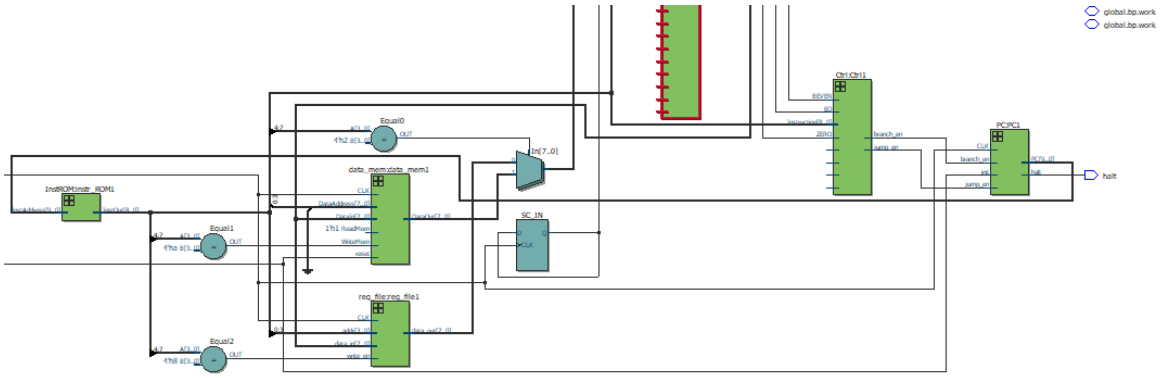
operations	func	description
set	R0 = SETNUM;	Set 8 bits immediate into the accumulator register
add	{SC_OUT, OUT} = {1'b0,INPUT} + {1'b0,R0} + SC_IN;	Add the operand register to accumulator r0,output include the carry out flag.
sub	OUT = R0 + (~INPUT) + SC_IN + 1; SC_OUT = OUT[7];	Subtract data of register from the accumulator r0, output include the carry out flag.
and	R0 & INPUT;	And gate operation register with accumulator

or	R0^INPUT;	Or gate operation register with accumulator

Calculation-independent operations are not listed in the table above.

The register file instantiates 16 8-bit registers, which are mainly used to store intermediate values during calculation.

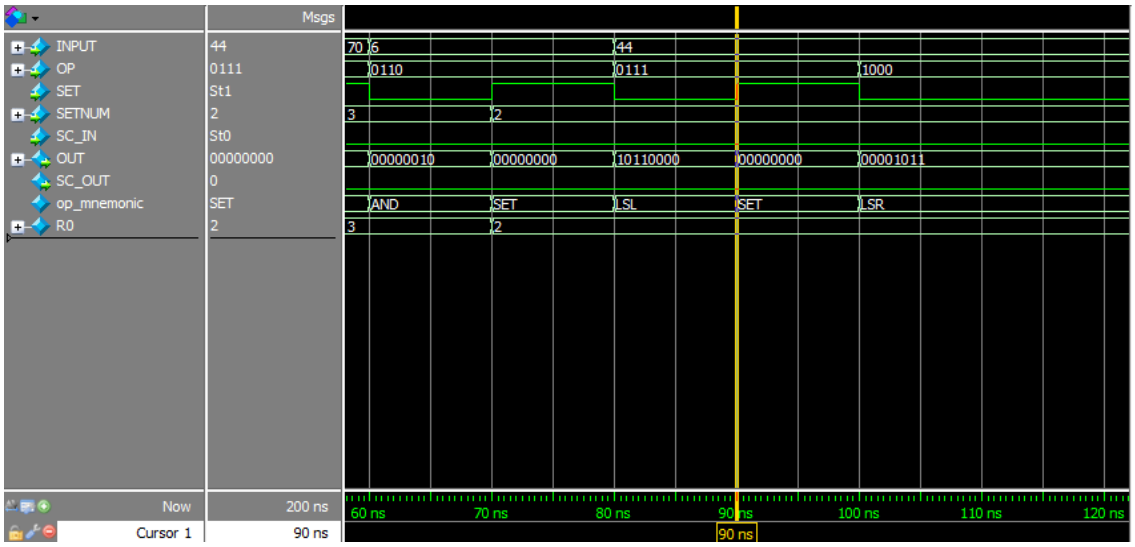
4. Full Verilog models, hierarchically organized **if** your top level module contains subassembly modules, some of which contain smaller modules.



This is the schematic diagram of the entire top level. The part that is not cut off is ALU, because it has a definition module connected to it, so the ports of this module shown in the schematic is particularly many. I don't know how to eliminate these extras,and I don't want to show it Interface

5. Well-annotated timing diagrams or transcript (diagnostic print) listings from your module level ModelSim runs, much as I have been demonstrating in class. It should be clear that your program counter / instruction memory (fetch unit) and ALU works. If your presentation leaves doubt, we'll assume it doesn't.

ALU:



fetch unit:

Answer the following question:

6. Will your ALU be used for non-arithmetic instructions (e.g., MIPS or ARM like memory address pointer calculations, PC relative branch computations, etc.)? If so, how does that complicate your design?
Yes,we do some PC relative branch computations in ALU. I think it's not the best choice, this will complicate the communication between alu and other modules. I think this situation should be avoided as much as possible, so a new module should be used to calculate branch instructions.