cs281: Introduction to Computer Systems

CPUlab – Y86 Hardwired Control

Assigned: Nov. 3, Due: Nov. 10

1 Specification

The objective of this second phase of the CPUlab is to build the hardwired Control Logic that drives automatic instruction execution on our SEQ datapath. Most of what you need is already in place, but we've updated and provided you with a new version of the Datapath. The Logisim circuits provided now consist of three files.

1. Y86Memory.circ – This file has not changed and defines the Instruction Memory and a handful of subcircuits exactly as provided in the datapath assignment.
2. Y86Control.circ – This file currently only has one circuit defined in it, and is where you should be doing all of your work in this lab. The main circuit in this file is called Y86Control, just like its filename, and the circuit simply consists of the input pins and output pins so that the interface can be used in the parent circuit, Y86.circ.
3. Y86B.circ – This is the modified datapath with connections to the Y86Control circuit that you will build in this lab. In addition to placing the Y86Control circuit interface on the datapath, we have added the Stat register along with control signals for statWrite and statValue. Error signals from instruction and data memory have also been tunneled to provide input to the Y86Control circuit.

Your task is to add combinational logic for all the control outputs of the Y86 circuit. The inputs to the circuit include `icode`, `ifun`, `CC`, `Stat`, `imemError`, and `dmemError`. All of these are defined on the left side of your Y86Control circuit. Your goal is to build a subcircuit for each one or two control signals. These subcircuits (all defined in Y86Control.circ) should implement the correct functionality based on the current instruction and other inputs to Y86Control, but not all subcircuits will require *all* the inputs.

You may design this control as you like, for instance, you could have subcircuits that generate `needs_regids` and `needs_valC` as described in the textbook and use these as inputs to the subcircuit that determines `PCIncSrc`, or you could simply use the four bits of `icode` as the input to `PCIncSrc`. You are, however, required to use subcircuits, so the top level of Y86Control should be your collection of subcircuits along with appropriate wiring of inputs into the subcircuits, and wiring from the outputs of the subcircuits to the output pins of Y86Control.

The Table below gives the meanings for each of the control signals on the Y86 datapath, and have not changed since phase 1, other than the additions mentioned above. For each, you should first determine the input(s) for determining the correct control signal value. Then build a truth table for each, employing "Don't Care" values to enable minimal circuits.

Control	Width	Description
continue	1	When 1, allows execution to continue since this control is ANDed with the clock before the clock continues to the rest of the datapath. So this value should be 1 as long as Stat is 00, and should be 0 otherwise.
statWrite	1	Determines whether or not the Stat register should be written.
statVal	2	Value to be stored in the Stat register when statWrite is 1. 00 means execution is OK, 01 means an invalid instruction was encountered, 10 means an invalid address (instruction address *or* data address), and 11 means a halt instruction was encountered. Note that these differ slightly from the textbook values of Stat.
PCIncSrc	2	Determines value to add to PC to get to next instruction. 00 - 1, 01 - 2, 10 - 5, 11 - 6.
valCsrc	1	Determine value for valC. 0 means valC ←Dest (M4[PC+1]), 1 means valC ←V/D (M4[PC+2]).
valAsrc	1	Determine read register file output for valA. 0 means valA ←R[rA], 1 means valA ←R[%esp].
valBsrc	1	Determine read register file output for valB. 0 means valB ←R[rB], 1 means valB ←R[%esp].
dstEsrc	2	Determine destination register for write of valE. 00 means R[rB] ←valE, 01 means R[%esp] ←valE, 10 and 11 mean send 0xf to dstE, indicating no write.
dstMsrc	1	Determine destination register for write of valM. 0 means R[rA] ←valM, 1 means send 0xf to dstM, indicating no write.
aluAsrc	2	Determine value to send to ALU_A. 00 means ALU_A ←valA, 01 means ALU_A ←valC, 10 means ALU_A ←4, 11 means ALU_A ←-4.
aluBsrc	1	Determine value to send to ALU_B. 0 means ALU_B ←valB, 1 means ALU_B ←0.
setCC	1	Determine whether or not to update the CC register on the next clock. 0 indicates do not update, 1 indicates update.
aluOp	1	Determine operation to route to ALUfun. 0 means 0000 (add), 1 means use ifun.
dmemAddr	1	Determine address routed to data memory address line. 0 means dAddr ←valE. 1 means dAddr ←valA.
dmemData	1	Determine value routed to data memory D input. 0 means D ←valA, 1 means D ←valP
dmemWrite	1	Determine whether or not to store D at M4[dAddr] on the next clock. 0 indicates do not write memory, 1 indicates write memory.
newPC	2	Determine source of next Program Counter to be routed to input of PC register. 00 means newPC ←valP, 01 means newPC ←valC, 10 means newPC ←valM, and 11 is undefined.

2 Testing and Evaluation

This part of the CPUlab is worth 50 points, allocated as follows:

- 10 points for clean and clear design and documentation of circuit truth tables.
- 10 points for the provision of a set of simple Y86 test programs, targeted at demonstrating correct operation of your CPU. These test should, in aggregate, cover all 13 instructions, with sufficient variety to hit likely combinations of instruction variations.
- 30 points for correctness. Correctness will be determined by executing test programs on your Y86 CPU, and comparing against the same program execution when run by `yis`. Like your own, my test programs will consist of unit tests to check individual instructions, and two integrated tests that use a variety of instructions in a "real" program.

3 Hints

1. Start by hardcoding `continue` to 1, and set up "sensible" defaults (as constants) for most of the control signals. Then work a single control signal at a time (consistent with the rest of the defaults).
2. Build your set of test programs as you go, so that you don't have to do the same work twice.
3. I would begin with getting `irmovl` to work, since your test programs will need a mechanism to load registers before you can test much else.
4. Be careful with `dmemError`. This signal is asserted by the data memory unit anytime `dmemAddr` has either too high an address or anytime the data address is not word aligned. It is only truly an error, however, when the current instruction is actually using the data memory to load or store data, and so this must be taken into account in your circuit to determine `statWrite` and `statVal`.
5. Save the integrated program tests for *after* you have worked through the individual instruction tests.
6. Be careful with the `rrmovl`, because you must provide for the general case of `cmovXX` that `rrmovl` is a specific instance of.