

CSE 141L Project 2: CPU Internals

Due Thursday, August 24th at the beginning of class.

Introduction

In this project, you will design the ALU (arithmetic logic unit) and the internal storage of your CPU. You will implement your design in LogicWorks, where you will connect the parts together and demonstrate that they work through schematic diagrams and timing printouts.

By "internal storage", we mean the register file, stack, accumulator, or whatever similar structure your ISA relies on. If your ISA has none of these, or if your machine operates completely out of memory, you will have to rig up something to supply operands to the ALU in order to demonstrate it. For the rest of this document, we'll refer to the internal storage as "the register file", for simplicity.

Sample Layout

You should connect your register file to your ALU in a manner similar to that shown in the attached schematic.

In this schematic, the ALU has two 8-bit inputs (A and B), one 8-bit output (C), and three control bits (OC). The ALU is a *combinational logic* block: it computes the output C, as a function of the inputs A and B, with the function selected by the control bits. With three control bits, the ALU can support eight different functions, though it may implement fewer. In this example, we've shown control inputs (ALU "op codes") for eight different functions. Your ALU and ISA will likely support different functions, and use different ALU control bits to select them.

This sample register file has two 8-bit data outputs (DOA, DOB), and one 8-bit data input (DI). The register file also has three address inputs: RAA ("Read Address A"), RAB ("Read Address B"), and WA ("Write Address"). Finally, there are the "write enable" and clock signals. The register file continuously retrieves the values of the registers addressed by RAA and RAB, and drives those values on the data outputs DOA and DOB. The register file may also write to up to one register per cycle: at a positive clock edge, if the "write" signal is active, the value at the data input DI is copied into the register addressed by WA. Note that, since this register file uses two bits for addressing, it contains a maximum of four registers.

Besides the ALU and register file, there are LogicWorks hex entry pads, a few switches, and a big multiplexer. The hex entry blocks allow you to manually select register addresses, ALU opcodes, and enter data values. The multiplexer is used to manually write values into the register file in order to demonstrate your data path, without having to re-wire the circuit.

This entire circuit is just a sample. Yours will differ, especially in the ALU operations supported, the bit-patterns you use to encode them, the number of registers addressed, etc. (Note: the specific figure and timing diagram shown were prepared with an earlier version of LogicWorks.)

Detailing Your Design: Schematics

As part of this project, you will provide detailed LogicWorks schematics (printed on paper). In addition to your top-level testing circuit, which will look something like the sample, you must also show *all* internal circuits used in your design: your ALU implementation, your register file, etc. (Note: you may not use any of the VHDL features offered by LogicWorks.)

You are strongly encouraged to use the LogicWorks features which allow you to collect circuitry into a block, and then re-use that block in other circuits. In the example circuit, observe that the ALU is shown as a module, instead of directly as a multitude of gates. The implementation of the ALU is a separate diagram, showing only the internals of the ALU; within the ALU, the adder itself might be a module containing a sub-circuit. Organizing your circuitry in a hierarchy like this is akin to using subroutines in software development: it helps you manage complexity, and can make your system much easier to understand.

Demonstrating Your Circuit

In addition to providing schematics, you must demonstrate correct operation of all of your ALU operations using LogicWorks timing diagrams (printed on paper). An example of a partial timing diagram is included; yours will have more detail.

The timing diagram should demonstrate all ALU operations (this includes math to support load/store address computation and branch target computation, but not PC incrementing), each with a couple of interesting inputs. Make sure that "corner cases" (unusual inputs) are demonstrated, if relevant.

The timing diagram should also demonstrate all of the functionality required of your internal storage, such as writing ALU outputs to

registers, and reading different registers. In particular, you should demonstrate that you can use the same register as a source and destination operand of the same instruction, if allowed by your ISA.

The schematics and timing diagrams will be difficult for us to understand without a great deal of annotation: add notes telling us what's going on. Good organization and use of hierarchical design also help. (If we can't understand it, we'll assume it doesn't work.)

Project Report

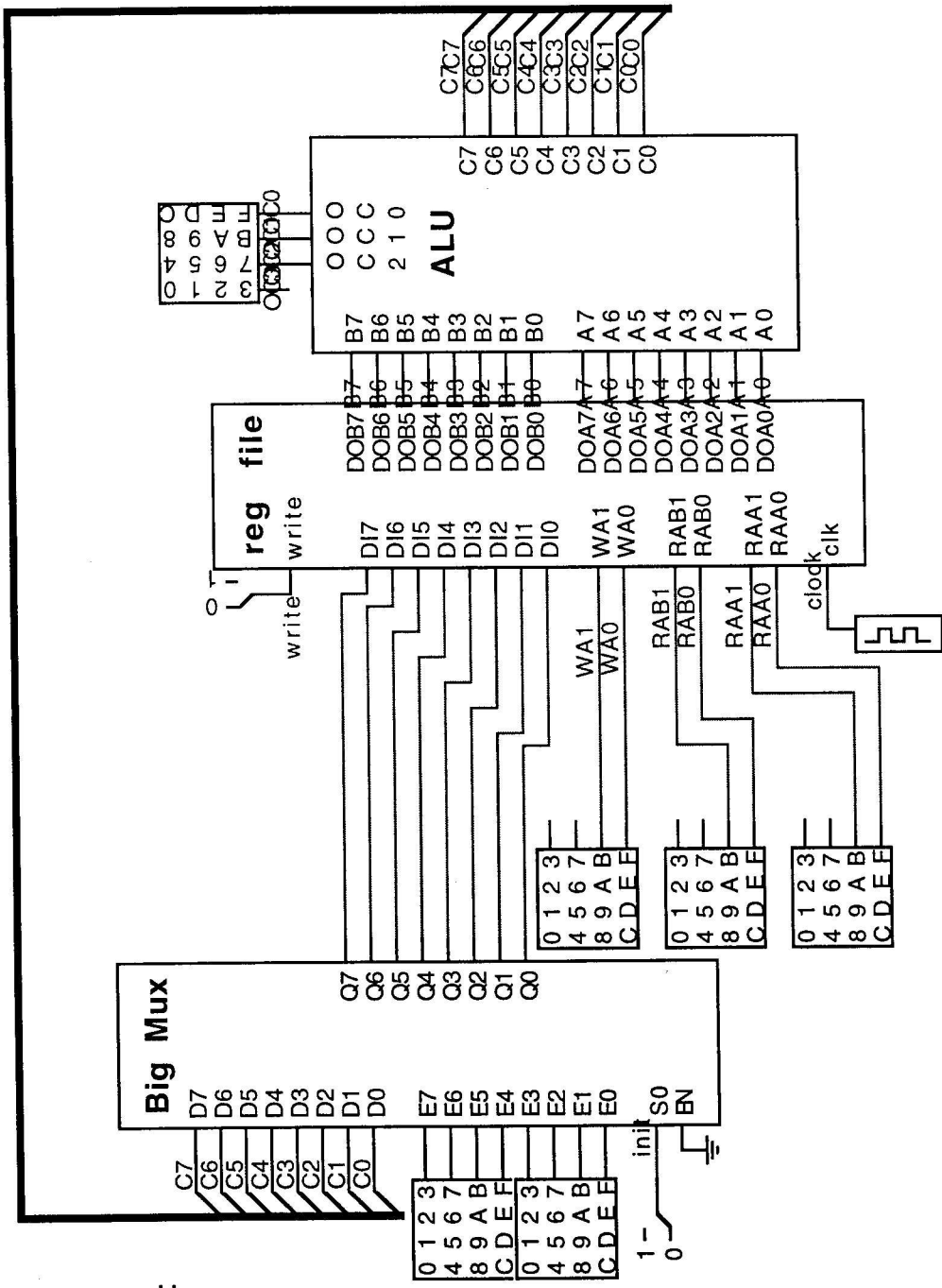
You must turn in a project report, printed on paper, in addition to your printed schematics and annotated timing diagrams. Be sure to list the names of everyone in your group on the report. Groups are expected to be the same as those used in the first project; changes must be approved in advance.

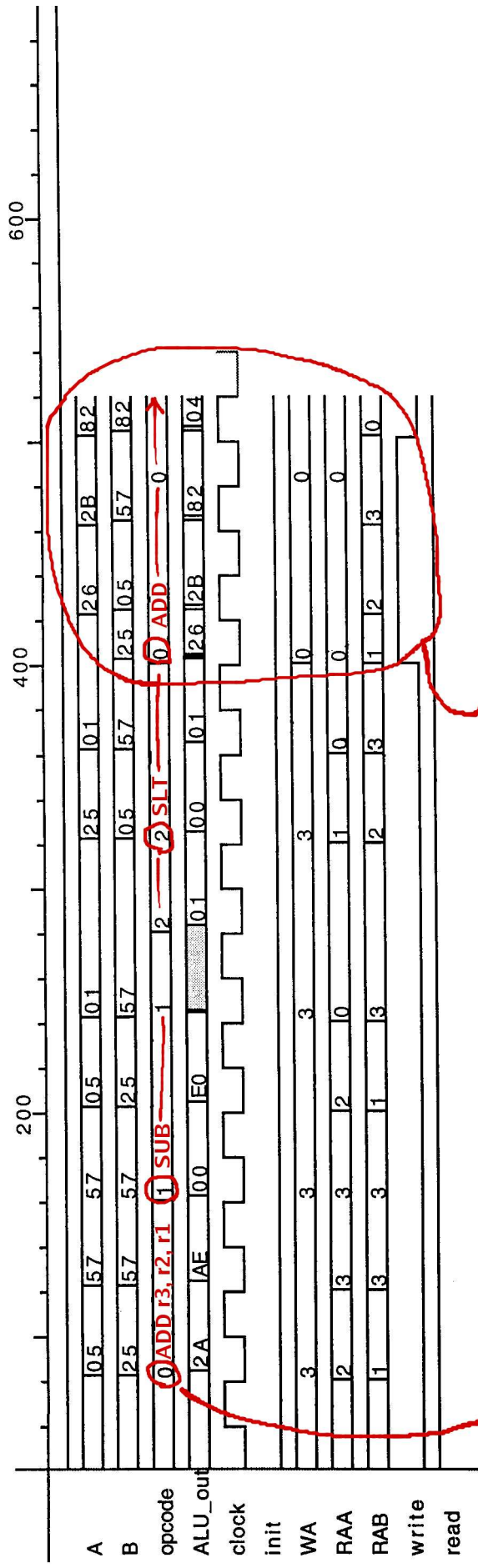
Your writeup is to be organized as follows. Be sure to answer all questions:

1. Introduction, general comments.
2. Summarize your ISA from Project 1: list the instructions and operations in your ISA, with descriptions of each.
3. List all operations that your ALU supports, indicating which ALU operations are used by which instructions.
4. Schematics: provide full schematic diagrams for your ALU and internal storage. Organize them hierarchically; label each diagram so that we can tell where they fit within your hierarchy.
5. Timing diagrams: provide just enough detail to show that every ALU operation your ISA requires is implemented correctly, but no more. If you leave an operation out, we'll assume it doesn't work and deduct points ("prove it works"). Make your timing diagrams as concise as possible ("get to the point"); if you spend 10 pages demonstrating a simple comparison operation, we'll be grumpy after having to read through it all, and we'll deduct points.
6. How many (and what) ALU operations do you support?
7. Will your ALU be used for non-arithmetic instructions (e.g., address calculation, branches)? If so, how does that complicate your design?
8. Do any instructions need to use the ALU twice (or need multiple ALUs)? If so, how does that complicate your design?
9. What is your most complex instruction, from the standpoint of the ALU?
10. Are your ALU control signals easily derived from your opcode bits? How much translation is needed? How could you have designed your ISA to eliminate (or at least reduce) that translation?
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?
12. Is there anything you could have done in your ISA that would have made the job of designing your ALU easier?
13. Is there anything you could have done in your ISA that would have made the design of your register file (or other internal storage) easier?
14. Reflecting on your ALU and storage implementation, what's the slowest path through your ALU and storage? What's your slowest instruction? How might you change your design – hardware and ISA – to improve the cycle time?

registers
 initialized to:
 R0 0x1
 R1 0x25
 R2 0x5
 R3 0x57

opcodes
 000 add
 001 sub
 010 slt
 011 seq
 100 sgt
 101 shift-left
 110 shift-right
 111 and





Here, we demonstrate the ADD ALU operation by selecting the appropriate opcode value. The input operands are A and B; the ALU output ALUOut reflects the result.

Here, we're reading and writing registers simultaneously: we're adding $r0+r1+r2+r3$, accumulating the total in r0. Observe that the new values for r0 are only copied from ALU_out at positive clock edges. In every cycle, r0 is both read from and written to, and it works correctly.

Observe that, as we change the source register numbers, new operands A & B are output from the register file, and the ALU computes the new result a short time later.

(Only three operations are demo'd here; you'll need to demonstrate all of your ALU operations thoroughly.)