Computer Architecture Lab 3: CPU

David Zhu, Bonnie Ishiguro, Zarin Bhuiyan

November 2016

1 Processor Architecture

We originally intended to implement a pipelined CPU with controls, modeled after the design we were introduced to in class. However, we did not build the mechanisms for handling structural, control, or data hazards. Our resulting processor has the multi-cycle style Instruction Fetch, Instruction Decode, Execute, Memory Access, and Write Back phases with registers between each phase to store and propogate the output values at every clock cycle. This design could theoretically be extended to incorporate the full pipelined functionality.

Although the Instruction Memory and Data Memory components of our processor appear as separate components in the block diagram below, we implemented them as one Random Access Memory component.

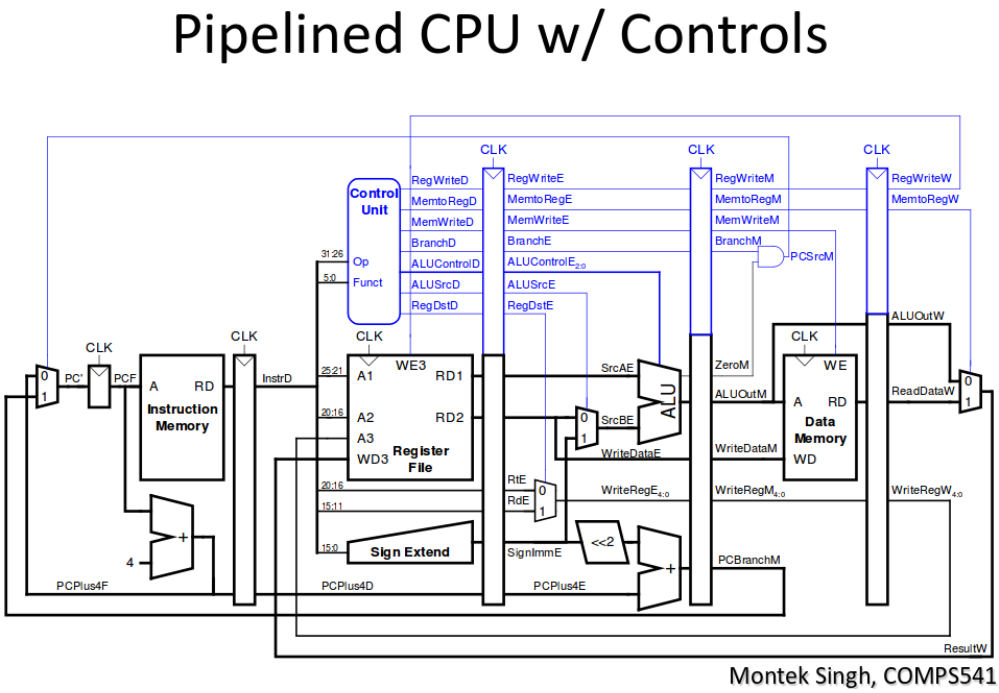

Figure 1: This is the block diagram of our CPU architecture, which is from the "Hazards" lecture of the course slide deck.

1.1 IF Phase

Instruction fetch, which is the phase between the first two clocked registers in the above diagram, is the process in which the CPU retrieves a program instruction from its memory and determines what actions should be taken to carry out that instruction. The next instruction is fetched from the memory address that is currently stored in the program counter (PC), and then stored in the temporary instruction register. At the end of the fetch, the PC points to the next instruction that will be read at the next cycle.

1.2 ID Phase

This step is where the instruction decode happens. The decoding is a process that allows the CPU to determine which instruction should be performed, so that the CPU knows how many operands it needs to fetch to perform the instruction. The operation code (opcode) is fetched from memory and is decoded for the next steps. The decoding is done by the CPU's control unit.

1.3 EX Phase

The execute phase of the pipeline executes the instruction with the decoded operands and operation. This phase contains an Arithmetic Logic Unit that handles the basic gate operations (and, nand, or, nor, xor), add, subtract, and set less than. Operands are either both loaded from the Register File in the ID phase, or one is loaded from the register file and the other is loaded as an immediate, or constant value. This phase also determines, if needed, which register to write to in the Register File. The execute phase handles branching as well, using an adder to add the PC counter to an appropriate immediate value.

1.4 MEM Phase

The memory access phase is the interface to the Data Memory. The output of the ALU is a Data Memory address, which can be written to with data from the Register File if the appropriate control signal, MemWrite_MEM, is high. The Data Memory can also output data already stored if given an appropriate address from the ALU. If the current instruction calls for a branch, the MEM phase also sets the control signal, PCSrc_MEM, high and sends the new PC address to the PC counter in the WB phase. Rather than incrementing by 4 bytes, the PC counter will instead be set to this new address.

1.5 WB Phase

The write back phase is where the results of the current instruction (data read from the Data Memory or the result of an ALU operation) are written into the Register File, using the specific destination register that was chosen in the execute phase.

2 Test Plan and Results

Our test plan consists of a top-level CPU test bench that contains a test for every MIPS instruction in our reduced instruction set: Load Word, Store Word, Jump, Jump Register, Jump and Link, Branch On Not Equal, XORI, Add, Sub, and Set Less Than. To test the Load Word and Store Word instructions, we created a mock data memory that contains the value 3 at address 7. To test the Jump, Jump Register, Jump and Link, and Branch Not Equal instructions, we verify the value of the PC Counter at the end of the instruction cycle. For the XORI, Add, Sub, and Set Less Than instructions, we check that value living at the appropriate destination register at the end of the instruction cycle.

To test the ability of our CPU to run programs, we wrote an assembly test that calculates the slope of a line between two points. The program takes four integers as parameters x1, y1, x2, and y2, which represent the x and y coordinates of the points, and calculates the floored result, as we are working with integers rather than floats. This test does not handle negative values as x and y coordinate parameters. We attempted to use a .data section to store the initial parameters, but it appears that loading parameters values from .data requires the use of the la (load address) MIPS instruction, which is not included in the reduced instruction set for our processor designs. Instead we used the addi instruction to load our parameters into the first four $a registers.

3 Performance

4 Work Plan Reflection

In this lab, we made an improved effort in delegating tasks so that we could work more in parallel. We specifically split up the instruction tests and writing the pipeline phases of our processor architecture. This strategy, however, led to us taking more time than we anticipated connecting our phases into a top-level CPU module.