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 propagate 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 (ram.v). We communicate to our RAM through two ports. The first is used for instruction reading, and the second is used for data Read/Write.

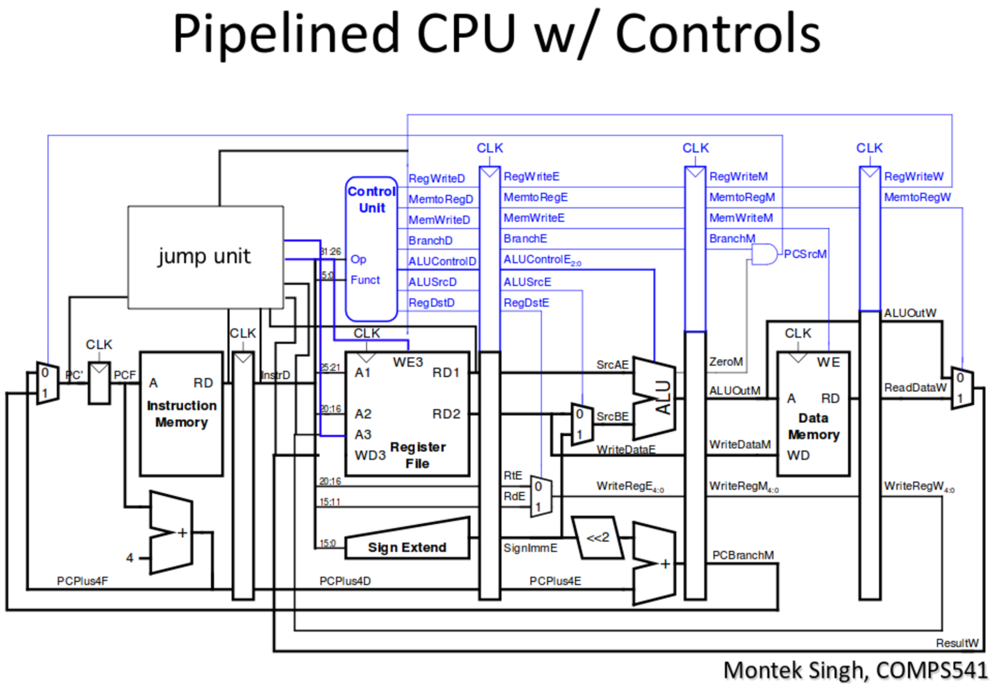

Figure 1: This is the block diagram of our CPU architecture, which is from the "Hazards" lecture of the course slide deck. We modified the design a little bit to incorporate logic such as the Jump Unit, which allows us to make jump requests faster.

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 passes that instruction to Instruction Decode. The instruction (word) is fetched from the memory address that is currently stored in the program counter (PC), and is then stored in the temporary instruction register. At the end of the fetch, the PC prepares itself with the address of the next instruction, which is read at the next cycle.

The Jump Unit also partially resides here; by determining that an instruction is meant for a jump, the CPU can directly proceed to the jumped instruction in the next clock 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. First, the instruction is divided into sections, some of which enter the register file with register addresses, others which go into the Control Unit and the sign extend, as well as our Jump Unit.

Inside the control unit, the instruction is decoded (through the opcodes and functors) to produce valid control signals that toggle our abilities to: write to register, send data to memory, branch our program counter, control our ALU, and change which registers in our regfile are written to.

The Register File contains 32 registers for fast data read/write access. The 5-bit register address is decoded in the register file data is outputed for our Execute Phase.

The Sign Extend allows for any intermediate values encoded in the instruction to be extended into 32-bits for future calculations.

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 loaded from the Register File in the ID phase or partly from the the sign-extended immediate. The ALU also requires a different set of opcodes, which are provided to us by the Control Unit.

Two other small components, the left-shift two and the adder, provides our Jump and Link instruction a way to jump to another position relative to its current instruction address. This is calculated here and passed to the next stage.

1.4 MEM Phase

In the Memory access phase, the CPU requets from our memory unit data to read or write. Since the memory unit does not live in the CPU, these are wire connections that request and resolve information from RAM.

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. This is again controlled by a signal propagated from the Control Unit.

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. Our interface to whether the CPU behaved correctly comes from toggling what the RAM is connected to. During the setup of a test, we toggle the RAM to connect to our testing interface and clock in our instructions and pre-test data. Then we toggle the RAM back to the CPU and release the program counter. The CPU starts at 0x0 and run through our instruction sequence. After a calculated amount of time (determined by how long we determine our instruction to run), we toggle the RAM back to the test and read the value deposited according to the test written.

To make this procedure work, we first had to white-box test LW and SW. This way, we can populate our registers with values and allow our CPU to write values back to our memory. This is extensively done using GTKWave / Scancion. We check the values of each gate in our "pipeline" and see if it matches what we expect. When something behaves differently, we and pinpoint easily where the issue occured. After the first wave of major tweaking / debugging, we got our LW and SW to behave appropriately, allowing us to write other tests.

For instance, when testing our ADD, we add the correct values and then take the value stored in the result and SW it back to our memory at a predetermined location. After the CPU completes, we read from that specific location in our memory. If the value is valid, then the test passes. Therefore, all of our other instructions are dependent on our LW and SW.

We also wrote an assembly test that calculates the (positive) 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.

We ran our slope.asm in MARS, producing the right answer as we tested a variety of inputs. However, the addi is also not supported in our CPU, so we took the assembly program from Jay Woo's team, which utilizes all of the commands we've implemented. After exporting their code with noop padding, we run it using Verilog with the same strategy as above. However, $readmemh could only be loaded during the beginning of the program, so it is embedded into ram.v directly. This meant that we could only run one assembly test at a time, and it had to be run at the beginning of our whole test suite.

RTL is inlined as comments in our CPU test file.

3 Performance

The performance of our current design is inefficient and bulky. The only hazard controls that are implemented are for our jump instruction, which means that all other instructions need to be padded with varying lengths of no-ops before the code can function properly.

LW, SW, and JAL take 3 clock cycles because they are done after MEM access. J takes 1 because of our Jump Unit. JR takes 2 (uniquely) because our jump do not trigger until 1 cycle in. BNE, XORI, ADD, SUB, SLT take 4 cycles because they have to write back to the register.

The most expensive components for our CPU were the control unit and the jump unit, which not be fully present in other CPU designs. The Jump Unit requires 6 muxes that operate in 2 different phases. Timing-wise, this is a source of future hazards as jumps are lined back to back.

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.

One major challenge was aligning up the timing for each component. There were many components and wires to deal with, and it was not always the easiest to use GTKWave to hunt down our specific signals. Lots of moving parts also meant having to diligently file through every signal to make sure it was behaving as expected in the beginning. Once our first command was successfully implemented, the rest came easy, but it took a lot of tweaking.

The pipeilned CPU design was a lot more work than expected. Considering the I/O of each section, as well as potentials for hazard control, made it hard to determine what are all the steps it may require to get the CPU up and running.