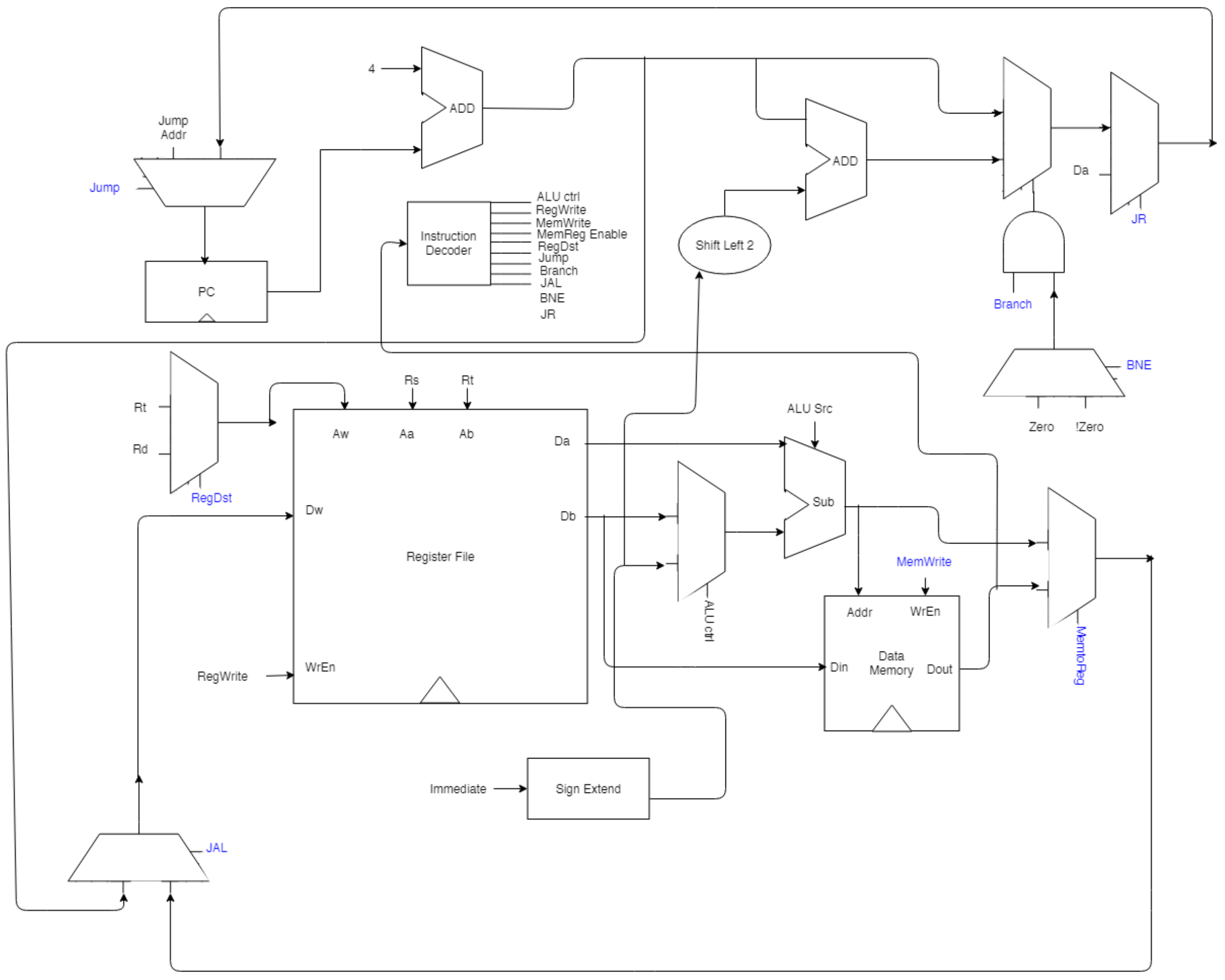
Comp-Arch: Lab 3
10/10/2018
Vicky McDermott
Minju Kang
Allison Basore

Comp-Arch Lab 3: Single Cycle CPU

Or this lab, we built a single cycle CPU, combining blocks we have made previously and integrating the idea of a single cycle design. While in modern practice a single cycle design is not often used, it is a helpful building block as a first pass CPU. Our CPU works in five stages that all perform in one clock cycle. The first stage is instruction fetching where we are reading the instructions from the stack. Next, we have the decoding state where we are reading the operation code from the instructions, which gives the inputs for the third stage: computation. Here we are simply performing the calculation with the ALU. Next, we have the memory access stage where we load and store the values, and finally, we store the output of our instruction to some register before incrementing the PC by four and starting again.

System Diagram



Comp-Arch: Lab 3
10/10/2018
Vicky McDermott
Minju Kang
Allison Basore

Description of Block Diagram

When the PC clocks a positive edge, it gets an instruction from the data memory. This is fed into the instruction wrapper which holds the decoder where it is parsed and control signals are made that will support the rest of the system. Once the control signals are set, within one cycle of the clock, the instruction is performed.

From the circuit diagram, we can clearly see and map out all the control values that need to be set each clock cycle. Below in Table 1 is a table showing controls for our instruction decoding. Since some parts of the CPU are not used during certain instructions, a few spots in the table have x's or ~. These signify that the controls do not matter here.

Our PC value is calculated using a variety of factors and control signals. Since instructions are 4 bytes long in MIPS, we know we always want to increment by four, unless we are jumping to an absolute jump address. To decide whether we take the value of PC+4 or the value of PC+4+IMM, we need to determine if we want to branch or not. There are 2 types of branching we need to support, branch equal and branch not equal. To do this, we can subtract the value of B from the value of A using our ALU to determine if A and B are equal. If they are, the zero output of our ALU operation will be 1 (as one minus the other will result in a 0 value). If they are not, the zero output will be 0. Knowing this, we can use a mux to decide whether to branch if they are equal and then another mux to decide if we want to branch at all.

Following the branch designation, we also need to feed that value into a mux with the instruction's jump value. This way we are able to determine whether we want to jump to a address or go to the other stored PC address. When an instruction is not of type J, the jump address will be an incorrect combination of an immediate and register destinations and not used. Since the instructions are word aligned, we also needed to shift and extend the values of both the jump and branch address and immediate respectively.

In the case where we want to jump register and set PC to the value in rs (which is really coming straight from the pc instruction), we have a simple mux control where we either select the value in rs that is in the register or we continue with the PC+4 value (technically this value could also be the branch address). The JR, branch, and jump functions control our next PC address. The rest of the CPU is working to perform the instruction with the ALU, register file, and memory.

Our CPU needed to be able to load and store in the data memory. The address to store is given in the instruction with the immediate and register source. For store word, all we do is write to the data memory at the address whatever value is in target register, again coming straight from the instruction. Similarly, for load word, we are putting the value already in memory into the register for easy access.

Lastly, the CPU supports calculations of immediate and of values already stored in the register file. Via a control signal, we specify if we are calculating with an immediate or a pre-saved value in the register.

Comp-Arch: Lab 3
10/10/2018
Vicky McDermott
Minju Kang
Allison Basore

OP	Type Instr	ALU control. (0=sign ext 1= b)	Reg Write (1= enable)	Mem Write (1 enabled)	Mem2Reg (1=data out 0 = alu output)	RegDst (1=rd, 0 =rt)	Jump (1=enabled)	Branch (1 = enabled)	Jump & Link (1=enable)	BNE (1=branch when not eausll,0=branch when equal)	JR (1= enable)	ALU SRC
LW	I	0	1	0	1	0	0	0	0	0	0	add
SW	I	0	0	1	0	0	0	0	0	0	0	add
J	J	0	0	0	0	0	1	0	0	0	0	~
JR	r	0	0	0	0	0	0	0	0	0	1	xxx
JAL	J	1	1	0	1	1	1	0	1	0	0	xxx
BEQ	I	1	0	0	0	0	0	1	0	0	0	sub
BNE	I	0	0	0	0	0	0	1	0	1	0	sub
XORI	I	0	1	0	0	0	0	0	0	0	0	xor
ADDI	I	0	1	0	0	0	0	0	0	0	0	add
ADD	R	1	1	0	0	1	0	0	0	0	0	add
SUB	R	1	1	0	0	1	0	0	0	0	0	sub
SLT	R	1	1	0	0	1	0	0	0	0	0	slt

Table 1: Control Signals Table

Shortcomings and Plans for Improvement

Currently are CPU is able to recognize, perform, and store calculations with the ALU. It also has the ability to branch via instructions. We are currently experiencing an issue with jumping. We believe we have a word alignment problem but have yet to understand what would be the cause. Given a jump instruction, the CPU will execute up until the jump instruction and then it jumps to the wrong instruction. Since our next project will require the workings of the CPU, we will have this problem understand and fixed soon. We anticipate that we may discover other issues once we get past the jump issue, and we are prepared to spend time debugging these as well. We plan to seek help from our instructor for this problem.

Comp-Arch: Lab 3
10/10/2018
Vicky McDermott
Minju Kang
Allison Basore

Description of Test Plan and Results

For the components that we reused from previous labs, we included the modules (muxes, register file, data memory, ALU, and basic components such as dff and address latch) without creating new test benches since they're already tested. For the components that we created (instruction wrapper and branch) for this lab, we included new test benches. *Instruction_wrapper.t.v* checks if the output is corresponding to the control signal that it was given. We tested if JAL, ADDI, LW, and JR is returning what's expected when it was given the corresponding control signal. *Branch.t.v* checks if the control signal which controls the "branch mux" is returning 1 when there's any conditional statement. In other words, the control signal should return 1 when the output of the ALU unit is either zero or not zero since using ALU indicates that we're having a conditional statement. The control signal should return zero if ALU is not returning anything (raising the flag for not branching at all).

In addition to creating a test bench for each individual component that made up the block diagram for our CPU, we also created three assembly programs for testing our CPU. One of our assembly programs tests the arithmetic capabilities of our CPU by doing "multiplication" by 3 by simply adding a number to itself 3 times. We also have an assembly program to test our CPU's jump and branch capabilities by branching to a specific block when two values are not equal and jumping back to that block until the values become equal. We also added several simple tests that test the CPU being about to lw, sw, jump, and branch. Finally, we can use jump register to go back to where we left off before loading the value and then perform a basic arithmetic operation on the loaded value. This test is useful because it tests a number of our CPU capabilities at once.

Performance Analysis

We analyzed the performance of submodules that are added newly for this lab such as instruction decoder and branching module. Then, we calculated the worst case of delay (when all components inside of our CPU will be used) by adding the propagation delay of all submodules. This will not be an accurate propagation delay since modules being used will be different for each operation. For example, lw will not use modules related to writing.

For the Instruction Decoder:

The instruction decoder is simply built with muxes since the entire block is decoding the instruction signal into control signals. The input is 6 bit-long control signal and the output will be logic values of each operation. Therefore, we will have log2(12), which is rounded to 4, muxes connected in parallel for worst cases. Therefore, the total Gate Input Equivalent(GIE) of the instruction decoder will be 4*7 = 28.

Comp-Arch: Lab 3
10/10/2018
Vicky McDermott
Minju Kang
Allison Basore

For the Branching Module:
Following is the GIE calculation of branching module

Components Used	Cost	Number of used	Total
2 to 1 Mux	7	3	21
2 Input AND (NAND + Inverter)	3	1	3
			24

Worst case of the entire system:

Components Used	Cost	Number of used	Total
2 to 1 Mux	7	5	35
Branching	24	1	24
ALU Units (32bits)	768	3	2304
Register File	13312	1	13312
PC	13	1	13
Data Memory	832	1	832
			16520

Work Reflection
Overall, this module was very challenging for us. Until the very last day, we had to continuously review our system diagram. We thought we had it correctly very first, but as we continue moving forward, we found ourselves that we were only seeing part of the CPU. We also recognize the importance of being proactive about getting the instructor's help, which will be very helpful for us to move forward. However, we don't feel we did well on following our lab work plan. We expected our work to be done a day before its due and to have a finalized system diagram by the end of the first week, which didn't happen. We think this is because we underestimated the scope of this lab. For the next lab, we will seek the instructor's help earlier, therefore, we can quickly reflect the feedback we got and finalize our system.

Comp-Arch: Lab 3
10/10/2018
Vicky McDermott
Minju Kang
Allison Basore