Lab 4 Report

Coleman Ellis and Paige Pfenninger

How to run our tests:
Go to our Lab4 directory
make full_tests.sh executable
Run ./full_tests.sh

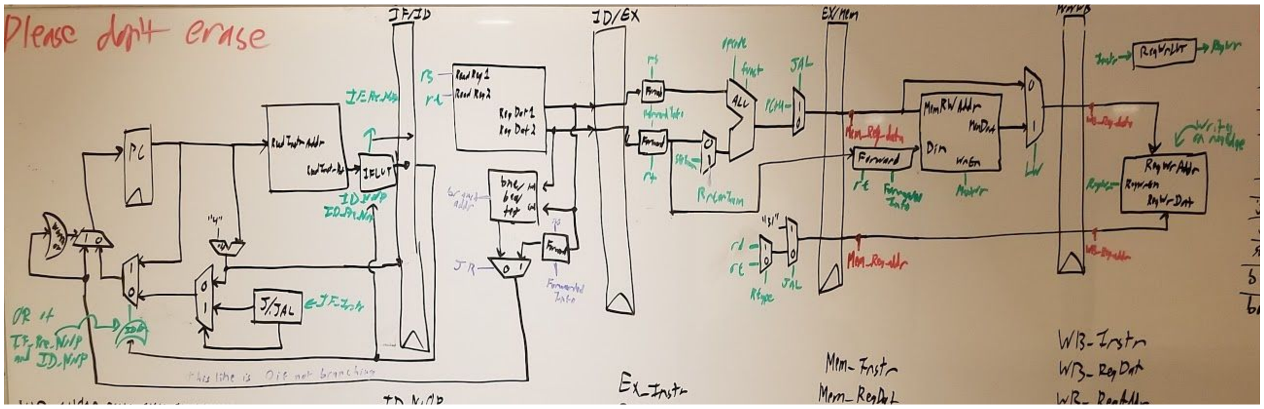
In our micro-proposal, we proposed making a pipeline CPU with only ADD, ADDI, LW, and SW. We quickly realized with the help of Ben that our micro-proposal was under-scoped too much. We then decided to make a pipeline CPU that has the same instructions as the first single cycle CPU.

The instructions our Pipeline CPU can do are:

1. LW: R[rt] = M[R[rs] + SE Immediate]
    a. The load word function inputs two registers (rs and rt) and a immediate. The ALU adds R[rs] and the sign extended immediate together and that sum becomes the address input to the memory block. The memory outputs the value at that address and then it gets written to the register file at rt.
2. SW: M[R[rs]+SignExtImm] = R[rt]
    a. A store word stores the value of R[rt] to the memory address of R[rs] plus a sign extended immediate. To complete this operation, write is enabled for the memory and the ALU is set to addition.
3. J: PC = JumpAddr
    a. Jump causes the PC address to be set to the jump address. The jump address is only 26 bits long, so it is concatenated with the most significant 6 bits of the current program counter.
4. JR: PC=R[rs]
    a. In this instruction, the program counter is just being set to whatever is stored in the register file at address rs.
5. JAL: R[31]=PC+4;PC=JumpAddr
    a. Jump and link performs two functions. First it saves the current program counter plus 4 to register 31 of the register file. It then changed the current program counter to the jump address.
6. BEQ: if(R[rs]==R[rt]) PC=PC+4+BranchAddr
    a. In our CPU, we used the subtract feature and the zero flag of the ALU to determine BEQ and BNE. If the zero flag is high, that means that the two inputs are equal, and so our CPU jumps.
7. BNE: if(R[rs]==R[rt]) PC=PC+4+BranchAddr
    a. For BNE, if the zero flag is low when R[rs] is subtracted from R[rt], then the two numbers are not equal so our CPU branches.

8. XORI: R[rt] = XOR(R[rs], SignExtImm)
    a. For an XORI operation, the ALU takes the XOR of R[rs] and the sign extended immediate and that value is then stored in rt
9. ADDI: R[rt] = R[rs] + SignExtImm
    a. For an ADDI operation, the ALU adds R[rs] and the sign extended immediate and that value is then stored in rt
10. ADD: R[rd] = R[rs] + R[rt]
    a. For an ADD operation, the ALU adds R[rs] and R[rt] and that value is then stored in rd
11. SUB: R[rd] = R[rs] - R[rt]
    a. For an SUB operation, the ALU takes the difference between R[rs] and R[rt] and that value is then stored in rd
12. SLT: R[rd] = (R[rs] < R[rt]) ? 1 : 0
    a. For an SLT operation, the ALU calculates if R[rs] is less than R[rt] and that value is then stored in rd

Here is our CPU - a larger version is at the end of this document
Please note that this CPU shows two memory locations and two register files, but we actually only have one of each in our CPU. This is just done to make the diagram easier to read.



Our CPU like a typical pipeline CPU has five phases: instruction fetch, instruction decode, execute, memory, and write back.

1. In instruction fetch, the program counter is incremented by the appropriate amount depending on the instruction and that instruction is located in the memory. The instruction is then stored in the IF/ID register on the positive clock edge.
2. In instruction decode, the register locations are taken from the instruction and the appropriate register values are found. We also do the branch and jump register calculations in instruction decode, these are typically done in execute, but we pushed them up to limit the number of hazards. The register values and instruction is then stored in the ID/EX register on the positive clock edge.

3. The execute phase is where the math happens. Our ALU can perform the following commands: SLT, ADD, SUB, and XOR. Before the instructions go into the ALU, we check to see if the register values need to be forwarded from EX/MEM or MEM/WB. We will talk more about forwarding later. The ALU result along with the instruction and register value that is being written to is then stored in EX/MEM on the positive clock edge.
4. Nothing happens in this cycle for all commands except for SW and LW. LW and SW both either load a word or save a word from memory during this instruction. Then either the ALU result or the output of memory and the instruction is stored in the MEM/WB register on the positive clock edge.
5. The write back phase actually occurs on the negative clock edge. This is done so that words can be written to in WB and read in ID in the same clock cycle. The only thing that happens in write back is a value is written to the register file if write is enabled. After this point we can access that value without forwarding.

Hazards

Because we decided to make a pipeline CPU, we knew that we would have to deal with many data and structural hazards.

To deal with the vast majority of data hazards, we created a forwarding unit. The forwarding unit compares the current register address and value to the register address and values in both the EX/MEM register and the the MEM/WB register if the register address match, the forwarding unit outputs the value that will be written to that register address (EX/MEM takes priority over MEM/WB). If the register addresses don't match, it outputs the current register value. The values are also only forwarded if the regWrite flag is enabled - so that we don't forward values that will never actually be written.

We use this forwarding in both the instruction decode and the execute phase. In the instruction decode we use the forwarding unit to check the inputs going into the branch calculator. We need to do this because we need the most up to date inputs going into the branch calculations, but sometimes those inputs have not been updated in the register file. We also use the forwarding in execute to make sure that we have the most up-to-date versions of R[rs] and R[rt] going into the ALU. We do the forwarding in both places because the branch calculations need it in instruction decode, but we want to wait until the last possible moment to check the inputs into the ALU so that we have the fewest data hazards - and that has to happen in execute.

Some data hazards cannot the accounted for with forwarding. The only place that this really happens is with LW when you try to use the word that was loaded in the instruction directly after it was loaded and JR/BNE/BEQ if you calculate the value of the register right before you jump or branch to that location. Forwarding doesn't work with the LW example because the word isn't available until the WB phase which is one cycle after execute needs it. Forwarding also doesn't work with the JR/BNE/BEQ example because the register value is being calculated at the same time that JR/BNE/BEQ needs it.

To fix the data hazards that could not be fixed with forwarding, we decided to just always insert a no-op instruction after LW and before JR/BNE/BEQ. The no-op instruction is just and ADDI instruction that adds zero to the zero register and saves it in the zero register (so it basically does nothing). When a no-op instruction occurs, the program counter does not advance. The no-op instruction basically stalls the CPU for one instruction to give the other instructions time to calculate.

The last type of hazard was what we are going to call the program counter hazards. These occur when a BNE/BEQ/JR happens because the next program counter doesn't actually get calculated until the instruction decode phase, so we don't know where the next instruction cycle should go. We decided to deal with this by just inserting a no-op instruction cycle directly after a BNE/BEQ/JR. This gives the CPU time to calculate where the program counter should go next and the program counter proceeds to that point after the one no-op cycle. We felt that this was the easiest solution to implement and still get a fully functional pipeline CPU.

Our Process

We again started this project by making a diagram on a chalkboard. Unfortunately, we forgot to take a picture of that diagram and it got erased. We then recreated it on paper and finally put it up on a large (and mostly available) whiteboard with several large please do not erase signs.

When the diagram recreation was happening, we were also creating additional individual components (along with test files) that we needed for the pipeline CPU. We needed to make a forwarding unit as well several branch/jump test units because those instructions needed to happen further up in our CPU.

We then ported everything from the whiteboard to verilog. All of our variables start with the phase name that they belong to. This helps organize our code and made debugging a lot easier. We also make very distinct sections in our code that correspond to sections of the CPU to aid with debugging.

We then started debugging our CPU. We again had a few issues with the control logic in our code, but we had significantly fewer issues with our pipeline CPU than we did with the single cycle CPU. I think going through the process of having to rewrite our CPU diagram 3 times really helped with getting the control logic right.

Our biggest bug was initially we were not checking if write was enabled when we were forwarding. This caused issues with instructions like BEQ and BNE that do not write to the register files but the ALU was still outputting a value and the register addresses matched the register address of the upcoming instructions so the garbage ALU result from BNE or BEQ were being forwarded when they shouldn't be. We fixed this issue by inputting whether or not the register file was actually being written to in our forwarding module. The forwarding module would then only update the data if the register file was actually being written to.

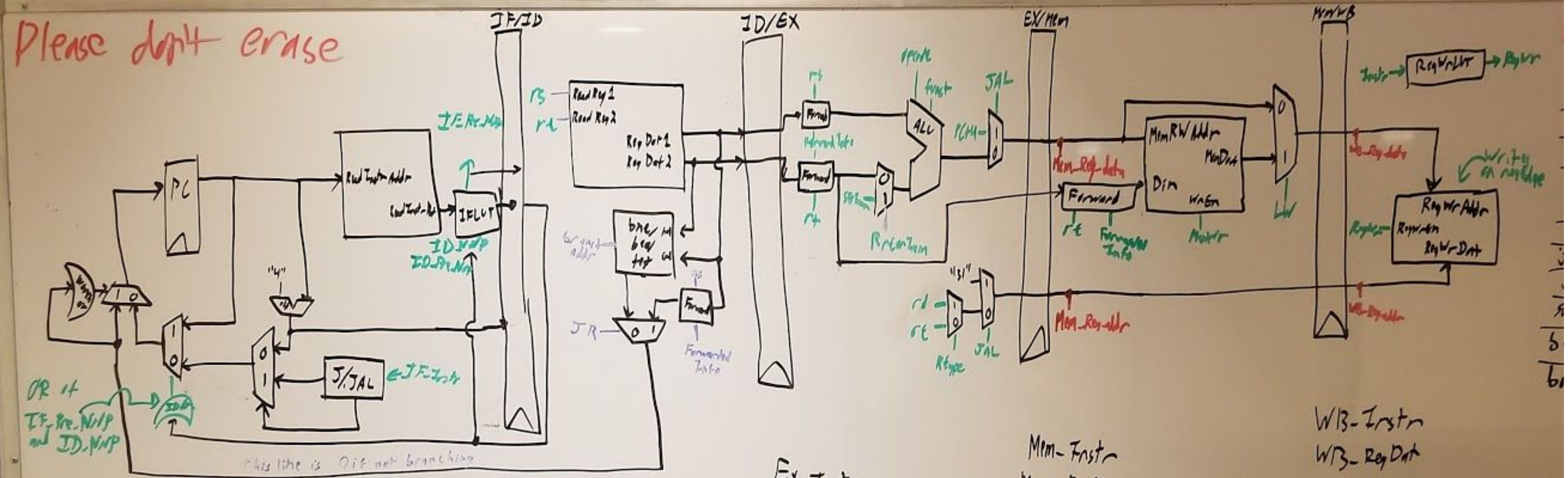
We had a couple more issues with forwarding where we were not forwarding things that we needed to forward, but those were much easier to find and fix.

Our Work Plan - Sort Of

We did not actually create a work plan for this project. We fell a bit behind on lab 3, and worked on lab 3 until Thursday 11/8. We then met on that Thursday to start lab 3. We broke up the pieces of the CPU and each wrote individual components as well as made a more detailed diagram of the CPU by Sunday 11/11. We then put the CPU together, started writing the report, and started debugging. Our CPU was fully put together on Tuesday and we were able to get it most of the way working. We then met for another hour on Wednesday and finished the CPU and this report.

After having a bit of a rough time with the first CPU lab due to illness, this lab went a lot more smoothly. Even though we started late, we were able to finish on time without straining ourselves too much.

Please don't erase



NOP: 001000 00000 00000 0000000000000000
addi r0 zero r0 zero Imm=0

NOP after bne, beq, jr, lw

IF LUT

```
In: Instr, NoOp
Out: Instr', NoOp'

if NoOp:
    NoOp' = 0
    Instr' = NOP
if Instr = lw, jr, bne, beq && NoOp = 0
    Instr' = Instr
    NoOp' = 1
else:
    Instr' = Instr
    NoOp' = 0
```

Please Don't Erase

ID_NoOP
ID_Instr
ID_PC+4
ID_Pre NoOP

ID_rs = ID_Instr[25:21]
ID_rt = ID_Instr[20:16]
ID_branchaddr = ID_PC+4 + {14{ID_Instr[15]}, ID_Instr[15:0], 2'b0}
ID_JR = ID_Instr[3]
(funct[3])

Ex_Instr
Ex PC+4
Ex_Rrs
Ex_Rrt

Ex_rs = Ex_Instr[25:21]
Ex_rt = Ex_Instr[20:16]
Ex_SEImm = {16{Ex_Instr[15]}, Ex_Instr[15:0]}
Ex_opcode = Ex_Instr[31:26]
Ex_funct = Ex_Instr[5:0]
Ex_rd = Ex_Instr[15:11]
Ex_JAL = Ex_opcode[$\overline{5}$, $\overline{3}$, 1:0]
Ex_Rt or Imm = Ex_opcode[3]
Ex_Rtype = NOR(Ex_opcode)

Mem_Instr
Mem_RegDat
Mem_RegAddr
Mem_Rrt

Mem_rt = Mem_Instr[20:16]
Mem_MemWr = Mem_Instr[5, 3, 1:0]
Mem_LW = Mem_Instr[5, $\overline{3}$, 1:0]

WB_Instr
WB_RegDat
WB_RegAddr

Mem_RegWr = Output from RegWrite

```
Forward:
In: target Reg, target Dat, Forward Reg 1, For
                            Mem_Reg_addr   WB
Out: Dat
case (target Reg)
    Forward Reg 1:
        Dat = Forward Dat 1
    Forward Reg 2:
        Dat = Forward Dat 2
    default:
        Dat = target Dat
```

Please don