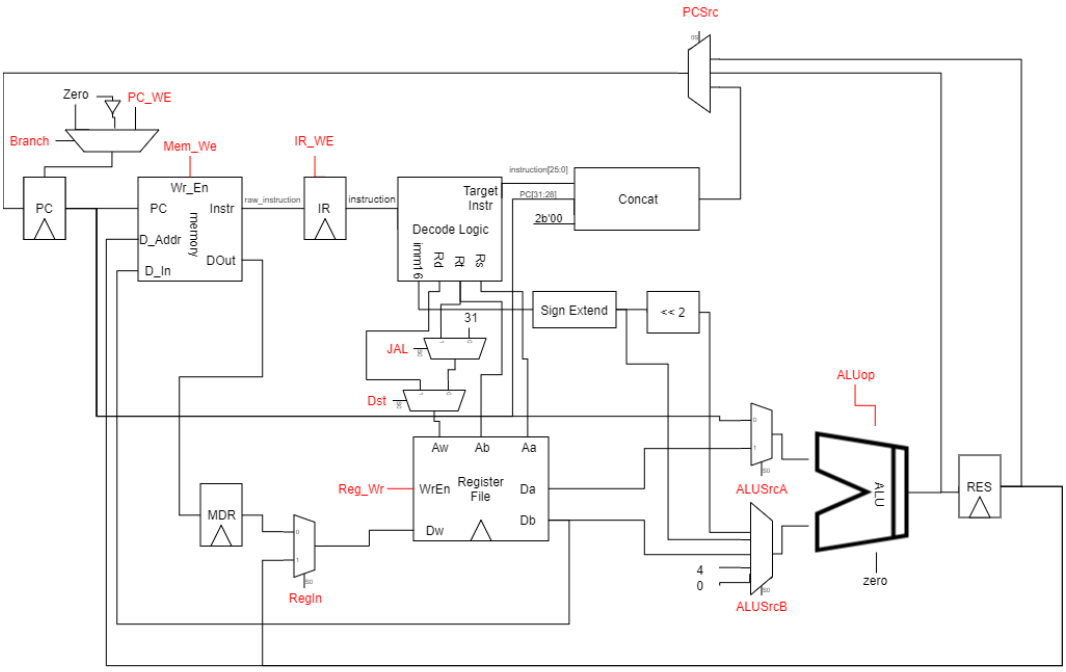
Lab b0100 Write-up

Mark Goldwater and Nathan Estill

November 15, 2019

1 Data Path and Finite State Machine



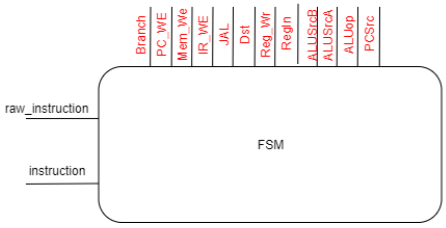


Figure 1: This figure shows the schematic for our multi-cycle CPU design. It is more compact and connected than the single-cycle design, for, as a result of the multi-cycle control flow, we are able to reuse hardware for different tasks. Note that for simplicity some muxes in the diagram do not show unattached inputs

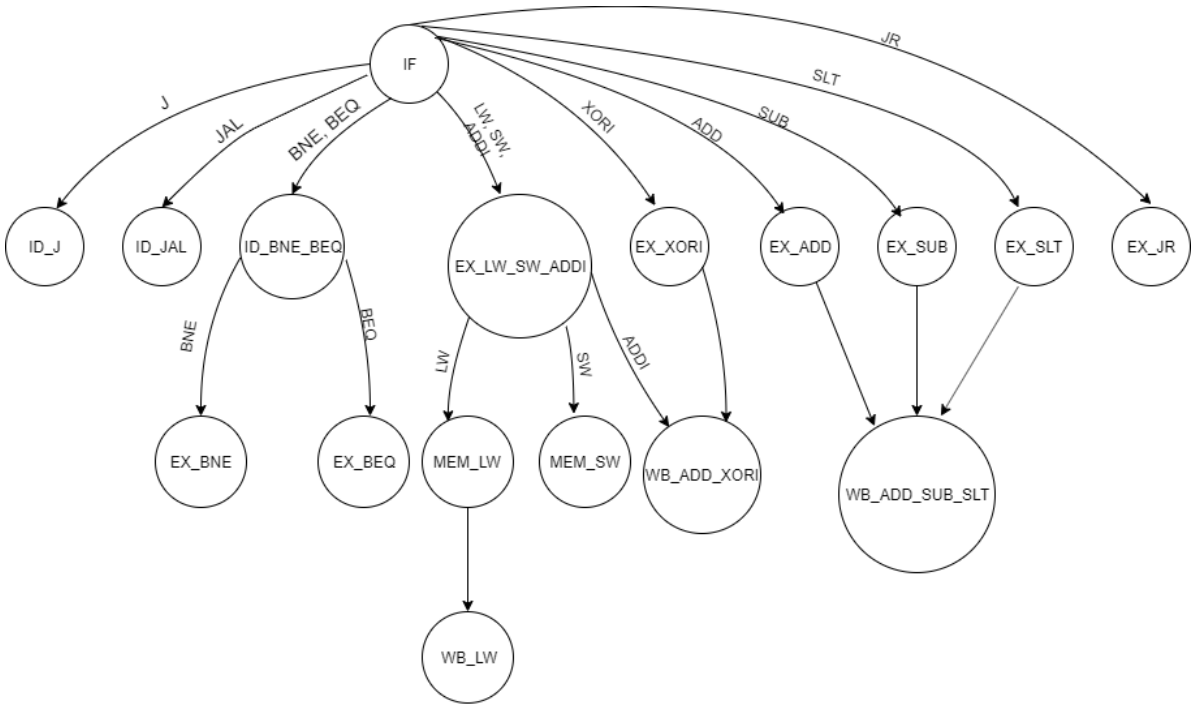


Figure 2: This figure shows the finite state machine used to guide the control signals for the different phases of each instruction. Note that each "leaf" of this FSM "tree" goes back to the IF state, but these arrows were omitted to make the diagram more clear.

There are five basic phases used to carryout the LW, SW, JR, J, JAL, BEQ, BNE, XORI, ADDI, ADD, SUB, and SLT instructions. These phases are instruction fetch (IF), instruction decode (ID), execute (EX), memory (MEM), and write-back (WB). Below, we describe how each instruction utilizes these phases, or a subset of these phases, in order to execute on our multi-cycle CPU. Note that each phase for a given instruction corresponds to a state in the FSM for the multi-cycle CPU

1.1 Load Word (LW)

The RTL which we want the data path to handle for the load word operation is R[rt] = M[R[rs] + SignExtImm]. In other words we want the CPU to load the contents of the data memory at address **R[rs] + SignExtImm** into the register file at address **rt**. In the IF phase PC is inputted into the memory unit, causing the current instruction to come out of the **Instr** port and goes into the **d** port of the IR register to be loaded on the rising edge of the clock. PC is also added to four to get the address of the next instruction which is led back to the PC register and loaded in on the clocks rising edge.

Next, we enter the EX_LW_SW_ADDI state, for the EX phase. Here the output of the **Da** port is added to the output of the SignExtend module (the sign extended immediate) and then stored in the ALU RES register on the next rising clock edge.

In the MEM_LW phase the MDR is then loaded with the data from **D_out** which is elicited by the input of the calculated memory location being fed into **D_Addr** from the ALU RES register.

Lastly, in the WB_LW phase, this value is then fed into the Dw port of the register file as **rt** is fed into the **Aw** port from the instruction decoder unit.

1.2 Store Word (SW)

The RTL for the execution of a SW instruction is M[R[rs] + SignExtImm] = R[rt]. The phases to execute this instruction are the exact same until the MEM phase. At this point the data address coming out of the ALU RES register is still input into the **D_Addr** port of the memory unit, but the data that is desired to be stored goes into the the **D_in** port of the memory unit from the **Db** port of the register file. Note that we skip the WB phase with this instruction.

1.3 Jump Register (JR)

The RTL for the JR instruction is simply PC = R[rs]. In order to make this happen, the instruction executes the same IF phase as previously described, but differs in the EX phase where **Da** (R[rs]) is added to zero through the ALU and, before the RES register, is fed back into the program counter, completing this instruction.

1.4 Jump (J)

The RTL for the J instruction is PC = JumpAddr. Our design accomplishes this by first executing the global IF phase and then, in its ID phase, concatenate the address to jump to (the last 26 bits of the current instruction on the **q** port of the IR register) to the first 4 bits of the program counter, and then to 2'b00 on the end. This is then fed back to the PC register in this phase.

1.5 Jump and Link (JAL)

The RTL for this instruction is R[31] = PC + 4; PC = JumpAddr (with no branch delay). This instruction first executes the global IF phase and then moves to the ID_JAL phase where the result of PC + 4 is fed into the **Dw** port of the register file, to be stored in R[31] where 31 was muxed into the **Aw** port of the register file. At the same time PC was set to JumpAddr in the same way it was for the Jump instruction.

1.6 Branch Equal (BEQ)

The RTL for this instruction is if (R[rs]==R[rt]), then PC = PC + 4 + BranchAddr. This is done by first executing the global IF phase and then moving into the ID_BNE_BEQ phase. In this phase we add the sign extended immediate to the current PC and store it in the ALU RES register. This result is then fed into the **d** port of the PC register. Then in the next phase (EX_BEQ) then **enable** of the PC register is controlled by the zero flag of the ALU while it subtracts R[rs] from R[rt]. If this is true, the desired instruction to branch to is let into the PC register on the rising clock edge. Special easter egg: Hey Ben, how was your day? Email us with the answer, we'd love to hear from you. Go take a break from grading. Get a snack or go on a walk or play with your kid.

1.7 Branch Not Equal (BNE)

The RTL for the BNE instruction is if(R[rs]!=R[rt]), then PC = PC + 4 + BranchAddr. This instruction functions in almost the same way as the BEQ instruction does in our datapath except the **enable** port of the PC register is controlled by zero rather than zero.

1.8 XOR Immediate (XORI)

The RTL for the XORI instruction is R[rt] = R[rs] $\oplus$ SignExtImm. To execute this instruction, first the global IF phase is executed. Next, in the EX_XORI phase the ALU calculates the bit-wise xor of R[rs] with the sign extended immediate and is loaded into the ALU RES register on the rising clock edge. Next, in the WB_ADD_XORI phase, rt is put into the **Aw** port on the register file and the value in the ALU RES register is loaded into the **Dw** port of the register file and then into the correct register address on the rising clock edge.

1.9 Add Immediate (ADDI)

The RTL for the ADDI instruction is R[rt] = R[rs] + SignExtImm. This instruction is executed in an identical way to the XORI instruction with our datapath, but the ALU calculates the addition of R[rs] and SignExtImm rather than the bit-wise XOR.

1.10 Add, Subtract, and Set Less Than (ADD, SUB, SLT)

The RTL for ADD, SUB, and SLT are R[rd] = R[rs] + R[rt], R[rd] = R[rs] - R[rt], and R[rd] = (R[rs] < R[rt]) ? 1 : 0 respectively. After completing the global IF phase, each of these instructions jump to the EX_ADD, EX_SUB, and EX_SLT phases respectively. In each one of these phases the ALU is set to calculate the correct operation on R[rs] and R[rt] and the result is stored in the ALU RES register. Lastly, all of these operations then enter the WB_ADD_SUB_SLT phase where rd is connected to the **Aw** port of the register file and the result contained in the ALU is loaded into the **Dw** register file port and then stored into the correct register on the rising clock edge.

2 Testing

2.1 Failures (Learning Opportunities™)

- Forgot to make Rtor31 a 32 bit wire, so it always treated it as a zero or one.
- Our simulation would be initialized without knowing what instruction would be running, so the IR register would have nothing in it when the state machine needed it. To fix it, we put more time for initialization before the first positive clock edge.
- After running one instruction, when the state machine needed the next instruction, it would not yet be in the IR register yet, so would act as if the previous instruction were still running. So we made it so that only the IF state uses the instruction prior to the IR register. We also reverted the previous fix.
- For some random operations, it detected the wrong instruction. This was because we were only checking the last six bits for the R type instructions, so some I and J instructions would be misinterpreted. Once we started checking for the first six and last six, the error went away.

2.2 Testing Plan

Tests for the ALU, register file we incorporated the original test benches into the repository and made them able to be run from the Makefile. In testing our multi-cycle CPU, we used the same test bench. We tested our CPU with a program which calculated the Tower of Hanoi recurrence relation and tested the instructions ADDI, BEQ, LW, ADD, J, and SW. We also ran a program

which recursively calculated the spinout recurrence relation for an input of 5 and used the following instructions: ADDI, JAL, J, BNE, ADD, JR, SW, and LW. Next, we tested a multiplication program which multiplied 7 by 5 using repeated addition and used the following instructions: ADDI, JAL, J, SLT, BNE, and ADD. At this point we were just missing the test of XORI and SUB, so we wrote a short assembly program which tests these two instructions.

We also reused a the Python script we wrote which dumps the memory and register file contents into a file with which it checks whether registers and memory locations contain the correct values based on given input when the cpu test is run (check the Makefile with instructions on how to do this).

2.3 Assembly Code For Tests

2.3.1 Tower of Hanoi

```
    # Save the 0th address of the array into $t4
    la $t4, our_array # addi, $t4, $zero,8192

    # Save the length of the array into t3
    addi, $t3, $zero, 3

    # Label which iterates through the array
    ARRAY:

    # Check if we've finshed with last element of array
    beq $t3, $zero, FINISH

    # Get value from specified location in array and
    # set counter for tower of Hanoi recurrence relation to zero
    lw, $t0, 0($t4)
    addi $t1, $zero, 0

    # Label marking Hanoi calculation loop
    START:

    # Check if register containing N is 0
    beq $t0, $zero, END

    # Double current value
    add $t1, $t1, $t1
    # Add 1 to current value
    addi $t1, $t1, 1
    # Subtract 1 from register containing N
    addi $t0, $t0, -1

    # Jump back start until desired N calculated
    j START

    END:

    # Load result in $t1 into memory address where N came from
    sw $t1, 0($t4)
    # Iterate array index by 4
    addi $t4, $t4, 4
    # Subtract 1 from array length value
    addi $t3, $t3, -1

```

```
    # Keep iterating through array
    j ARRAY

    FINISH:
    #j FINISH (infinite loop for running in verilog simulation)

    # Initialize array in memory
    .data
    our_array:
    0x00000010
    0x00000009
    0x0000000A
```

2.3.2 Spinout Recurrence Relation

```
    # Single Cycle CPU Assembly Test
# Liv Kelly, Jamie O'Brien, Sabrina Pereira

main:
# Set up arguments for call to spin
addi  $a0, $zero, 5
jal   fib


# Jump to "exit", rather than falling through to subroutines
j     program_end

#-----------------------------------------------------------------------
# Recursive Spinout Function.
# Tells you the minimum number of moves necessary to solve the Spinout puzzle with n
    gates
# Uses recurrence relation: F(n) = F(n-1) + 2F(n-2) + 1
# Initial conditions: F(0) = 0; F(1) = 1
#   Equivalent C code:
#   int Spinout(int n)
#   {
#       if (n==0) return 0;
#       if (n==1) return 1;
#       int spin_1 = Spinout(n-1);
#       int spin_2 = Spinout(n-2);
#       return spin_1 + 2*spin_2 + 1
#   }
fib:
# Test base cases. If we're in a base case, return directly (no need to use stack)
bne   $a0, 0, testone
add   $v0, $zero, $zero     # a0 == 0 -> return 0
jr    $ra
testone:
bne   $a0, 1, spin_body
add   $v0, $zero, $a0       # a0 == 1 -> return 1
jr    $ra

spin_body:
# Create stack frame for fib: push ra and s0
addi  $sp, $sp, -8  # Allocate two words on stack at once for two pushes
```

```
sw     $ra, 4($sp)   # Push ra on the stack
sw     $s0, 0($sp)   # Push s0 onto stack

# Call spin(n-1), save result + 1 in s0
add $s0, $zero, $a0     # Save a0 argument
addi    $a0, $a0, -1    # a0 = n-1
jal fib
add     $a0, $s0, -2    # a0 = n-2
add     $s0, $zero, $v0 # s0 = Spin(n-1)
addi    $s0, $s0, 1     # s0 = Spin(n-1) + 1


# Call spin(n-2), save 2*result in s0
jal fib
add     $s0, $s0, $v0   # s0 = Spin(n-1) + 1 + Spin(n-2)
add     $v0, $v0, $s0   # v0 = Spin(n-1) + 2*Spin(n-2) + 1

# Restore registers and pop stack frame
lw     $ra, 4($sp)
lw     $s0, 0($sp)
addi   $sp, $sp, 8

jr     $ra   # Return to caller

#-------------------------------------------------------------------------------
# Jump loop to end execution
program_end:
```

2.3.3 Multiplication Program

```
# Arithmetic Test
# Nah, Alex, Anna

# memory configure memory
addi $sp, $zero, 0x3ffc



# Main function
main:
addi $v0, $zero, 0
# intial values in a0 and a1
addi $a0, $zero, 5
addi $a1, $zero, 7

# call multiply function
jal multiply

j exit

# Multiply Function
multiply:

# initalize t0, counter, as 1
addi $t0, $zero, 1
```

```
26
27 # add a0 to itself a1 times
28 loop:
29     # loop a1 times
30     bgt $t0, $a1, exit
31
32     # adding one to the counter
33     addi $t0, $t0, 1
34
35     # add a0 to v0 which holds the previous result
36     add $v0, $v0, $a0
37
38     # jump back to beginning of the loop
39     j loop
40
41 exit:
42 j exit
```

2.3.4 XORI and SUB Tests

```
1  # Load numbers
2  addi $t0,$zero,30
3  addi $t1,$zero,2
4  addi $t2,$zero,20
5  addi $t3,$zero,10
6
7  # Test Subtract
8  sub $s0,$t0,$t1 #s0 = 30 - 2
9  sub $s1,$t0,$t2 #s1 = 30 - 20
10 sub $s2,$t2,$t1 #s2 = 20 - 2
11
12 # Test xori
13 xori $s3,$t2,21 # s3 = 20^21
14 xori $s4,$t3,5 #s4 = 10^5
15 xori $s5,$t2,10 #s5 = 20^10
```

3 Performance Analysis

3.1 Cycle Per Instruction (CPI) Analysis

To approximate the CPI of our multi-cycle CPU, we ran two sample programs and analysed the cycles per instruction. From the Instructions Statistics tool of MARS, we got the frequency of each operation of two of our programs. From there, we calculated the theoretical CPI based upon the states per each instruction (The two memory states are different, so an average was used as an approximation). Then we counted the actual number of cycles the program took to run with GTKWave and compared the two. They were almost identical, for the only difference was due to the discrepancy in the memory. Both of our programs have a similar CPI, which leads us to believe that this CPI will hold for programs of similar complexity and composition.

Operation	# States	Frequency	Clocks
ALU	3	116	348
Jump	2	38	76
Branch	3	42	126
Memory	3.5	6	21
Total	572	CPI	2.83
GTKWave Counting	570	CPI	2.821

Table 1: **Table of Cycles for Hanoi.**

Operation	# States	Frequency	Clocks
ALU	3	99	297
Jump	2	31	62
Branch	3	27	81
Memory	3.5	28	98
Total	538	CPI	2.856
GTKWave Counting	537	CPI	2.90

Table 2: **Table of Cycles for Spinout.**

Operation	# States	Frequency	Clocks
ALU	3	27	81
Jump	2	8	16
Branch	3	8	24
Memory	3.5	0	0
Total	121	CPI	2.81
GTKWave Counting	121	CPI	2.81

Table 3: **Table of Cycles for Manual Multiplication.**

3.2 Comparison to Single-Cycle CPU

While it would take a long time to find the clock period for each of the kinds of CPU, we can estimate that the Single-Cycle CPU will have a clock period of about five times that of the multi-cycle, as it performs all five phases in one clock period. Taking that into account, we can calculate the time per instruction for the multi-cycle (Equation 1) and the single-cycle (Equation 2) CPU as shown below:

$$1\frac{\text{Time Units}}{\text{Cycle}} \times 2.8\frac{\text{Cycles}}{\text{Instruction}} = 2.8\frac{\text{Time Units}}{\text{Instruction}} \tag{1}$$

$$5\frac{\text{Time Units}}{\text{Cycle}} \times 1\frac{\text{Cycles}}{\text{Instruction}} = 5\frac{\text{Time Units}}{\text{Instruction}} \tag{2}$$

As a back-of-the-envelope calculation, we can take into account that the average time per instruction for the multi-cycle is about 2.8, while the average time per instruction for the single-cycle as about 5. Therefore the multi-cycle CPU is about 1.75 times faster than the single-cycle CPU.