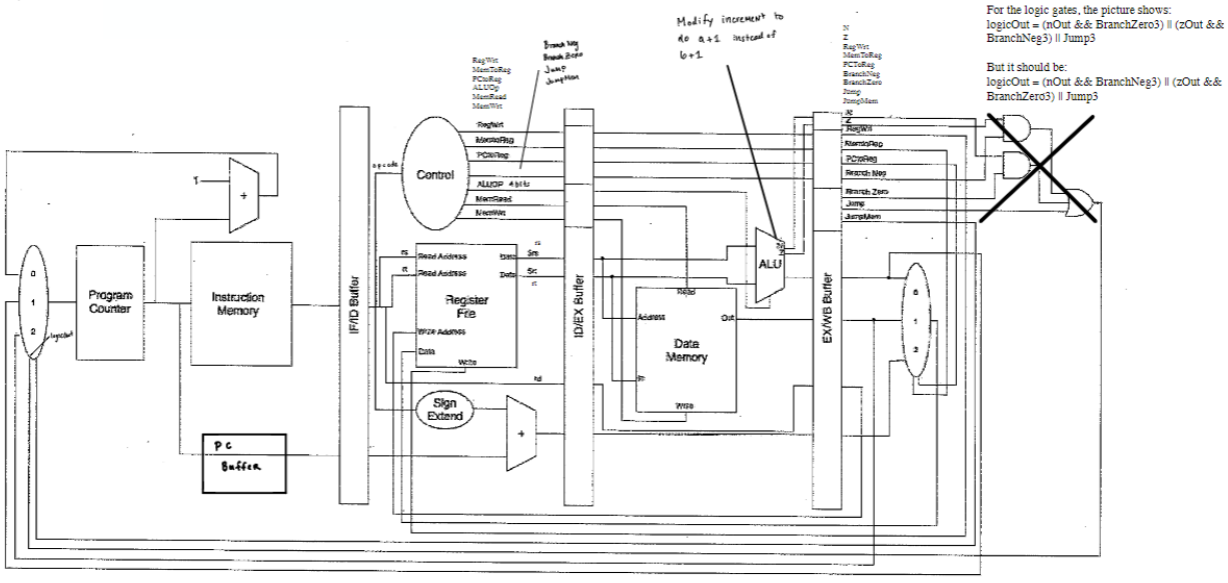
COEN 122 Final Lab Report

Rohan Nair
4/17/20

Abstract

We were tasked with constructing a gate-level implementation of a fully pipelined CPU with 13 opcodes and 32-bit instruction width in Verilog HDL. Our CPU was split into 4 stages with 3 buffers. We also wrote an assembly program using this SCU ISA to find the max value of an array of integers.



Instruction Fetch Stage

In the IF stage, we are first using a mux to determine the address that will be sent to our instruction memory. The 3 inputs to this mux are PC + 1, output from data memory, and ALU output. By default, we want to simply increment PC by 1, but if we encounter a jump, jump mem, branch neg, or branch zero instruction, we need to either update our PC with data from memory or from ALU operation. Once we have fetched the appropriate instruction from the I-MEM, the instruction along with the current PC are sent to the IF/ID buffer.

Instruction Decode Stage

In the ID stage, we are extracting bit-fields corresponding to opcode, rs, rt, rd, and literal from our 32-bit instruction. Opcode gets sent to the control block, which sets all of our flags (see truth table below), rs and rt are sent to the register file in order to obtain the contents of those register locations, and rd is sent directly to the ID/EX buffer. In this stage, we are also sign-extending our 12-bit literal and adding it to the PC for the “Save PC” instruction.

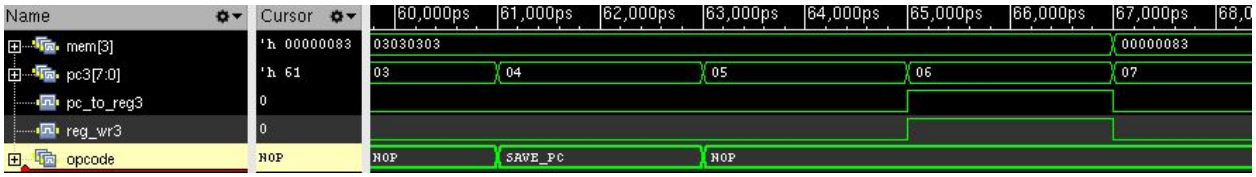
Truth Table for Control:

	RegWrt	Memto Reg	PCto Reg	ALUOp	Mem Read	Mem Wrt	Branch Neg	Branch Zero	Jump	Jump Mem
NOP	0	0	0	0000	0	0	0	0	0	0
SavePC	1	0	1	1111	0	0	0	0	0	0
Load	1	1	0	1110	1	0	0	0	0	0
Store	0	0	0	0011	0	1	0	0	0	0
Add	1	0	0	0100	0	0	0	0	0	0
Increment	1	0	0	0101	0	0	0	0	0	0
Negate	1	0	0	0110	0	0	0	0	0	0
Subtract	1	0	0	01111	0	0	0	0	0	0
Jump	0	0	0	1000	0	0	0	0	1	0
JumpMem	0	0	0	1010	1	0	0	0	0	1
BRZ	0	0	0	1001	0	0	0	1	0	0
BRN	0	0	0	1011	0	0	1	0	0	0

Execute Stage

In the EX stage, we are simultaneously passing the contents of rs and rt into our ALU and generating the appropriate result depending on the opcode of our instruction, and also reading from/writing to our data memory — depending on whether the MemWrt or MemRead signals are set. The ALU result along with the Z and N flags generated by the ALU, and the contents of any location we read from data memory are passed into the EX/WB buffer. We are also passing along the control signals from the previous stage.

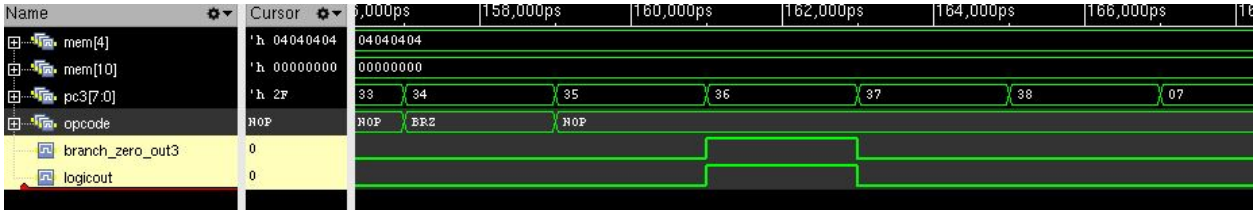
Waveform for ADD R6, R4, R5



Waveform for SAVE_PC function (0x07F+PC(4) =='h83 into register 3)

Write-Back Stage

In the WB stage, we are first determining which data gets written back into our register file. We have a three-to-one mux with three inputs: PC + sign extended literal (for Save PC instruction), output from ALU operation, and data output from the data memory. The selector for this three-to-one mux is determined by 2 bits - the MemToReg and PCtoReg signals.

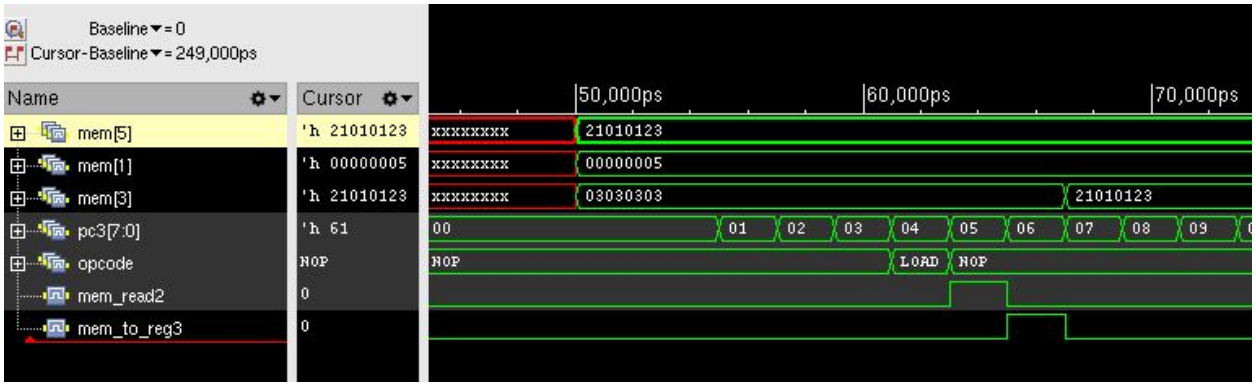


Waveform for Branch Zero (BRZ R10)

We also have a second mux to determine the next PC. This three-to-one mux has the following inputs: PC + 1, data output from data memory, and ALU result. This selector is determined by the LogicOut and JumpMem signals. The JumpMem signal is generated by the control block and LogicOut is the result of the following boolean expression:

LogicOut = (nOut && BranchZero) || (zOut && BranchNeg) || Jump

This simply means that if we have a BRZ instruction and the Z flag is set, or if we have a BRN instruction and the N flag is set, or if we have an unconditional jump instruction, we want to set LogicOut to be high.



Waveform for LOAD instruction (LOAD R3, R1)

Assembly code for MAX function

// initialize data memory

mem[10] = 32'd30
mem[11] = 32'd8
mem[12] = 32'd11
mem[13] = 32'd18

//initialize registers
//register[0] = 32'd0; //initialize max to 0
register[1] = 32'd10; //base address of A
register[2] = 32'd4; //# elements in A
register[3] = 1 //i
register[20] = 32'd20; //loop label
register[21] = 32'd50; //end label
register[22] = 32'd60; //if label
register[30] = 1;
register[31] = 0;
LOAD R0, R1 // initialize max to A[0]

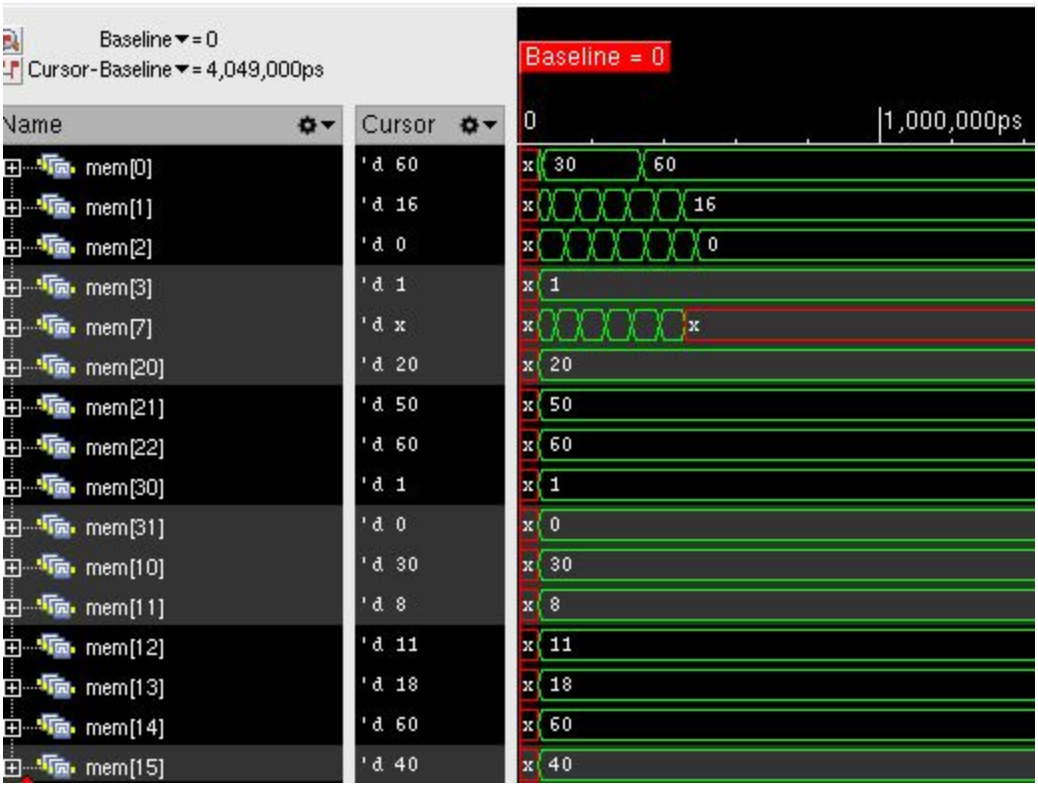
//LOOP
BRZ R21 // end label

ADD R1, R1, R30 // &A[i]
LOAD R7, R1 // A[i]
SUB R9, R0, R7 // max - A[i]

BRN R22
SUB R2, R2, 1 // n = n -1
J R3 // go to beginning of loop

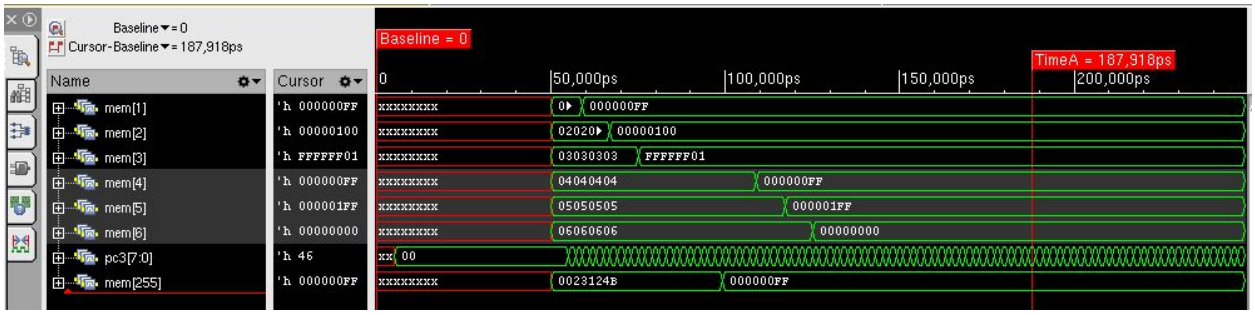
//IF
ADD R0, R7, R31 // max = A[i]
SUB R2, R2, R30 // n = n -1
J R3

//END



Waveform for max function (calculating the max of [30, 8, 11, 18, 60, 40])

Waveform for MAX instruction (MAX R1, R10, R11)



Waveform for demo program

Runtime Calculations

In order to obtain an estimate of the total runtime we took into account the longest delay at each stage of the pipeline: 2ns in the IF stage (corresponding to instruction memory), 2ns in the ID stage (corresponding to addition of PC and 12-bit literal), 2ns in the execute stage (ALU) and 1.5ns in the WB stage (register file). This gives us a maximum cycle time needed of 2ns. We can find total runtime by multiplying cycle time by the number of instructions (including added NOPs). For each element in our array, we have to make a comparison, meaning we have to branch to our LOOPlabel and execute instructions 20 - 54 (35 instructions). Each time we encounter a number that is larger than our current max, we must branch to the IF label and execute instructions 60 - 74 (15 instructions).

This means our total number of instructions is:

10 + 35*n + 15*m, where n is the number of elements in our array A, and m is the number of elements greater than A[0].

For my sample input of A = [30, 8, 11, 18, 60, 40], n = 6 and m = 2
(10 + 35*6 + 15*2 = 250)*2ns = 500ns. I ran my max function with these inputs to verify and got a total runtime of 435ns, which was fairly close.