Dillon Kanai
Matthew Kiyama
Ryan Summers

COEN 122 Data Path Report

Abstract

We were tasked to create an advanced 32-bit datapath which then acts as a pipelined CPU. We were given a list of instructions with which to input into our path to test its functionality. We started out by creating assembly code for the given instructions, this was important as our entire Vivado code was based on this code. During the first few labs we learned how to create certain individual modules of the final datapath and we used that knowledge to create the modules used in our project. Once the modules were created, we wired the modules together and coded a test bench with which to ensure our datapath functioned properly.

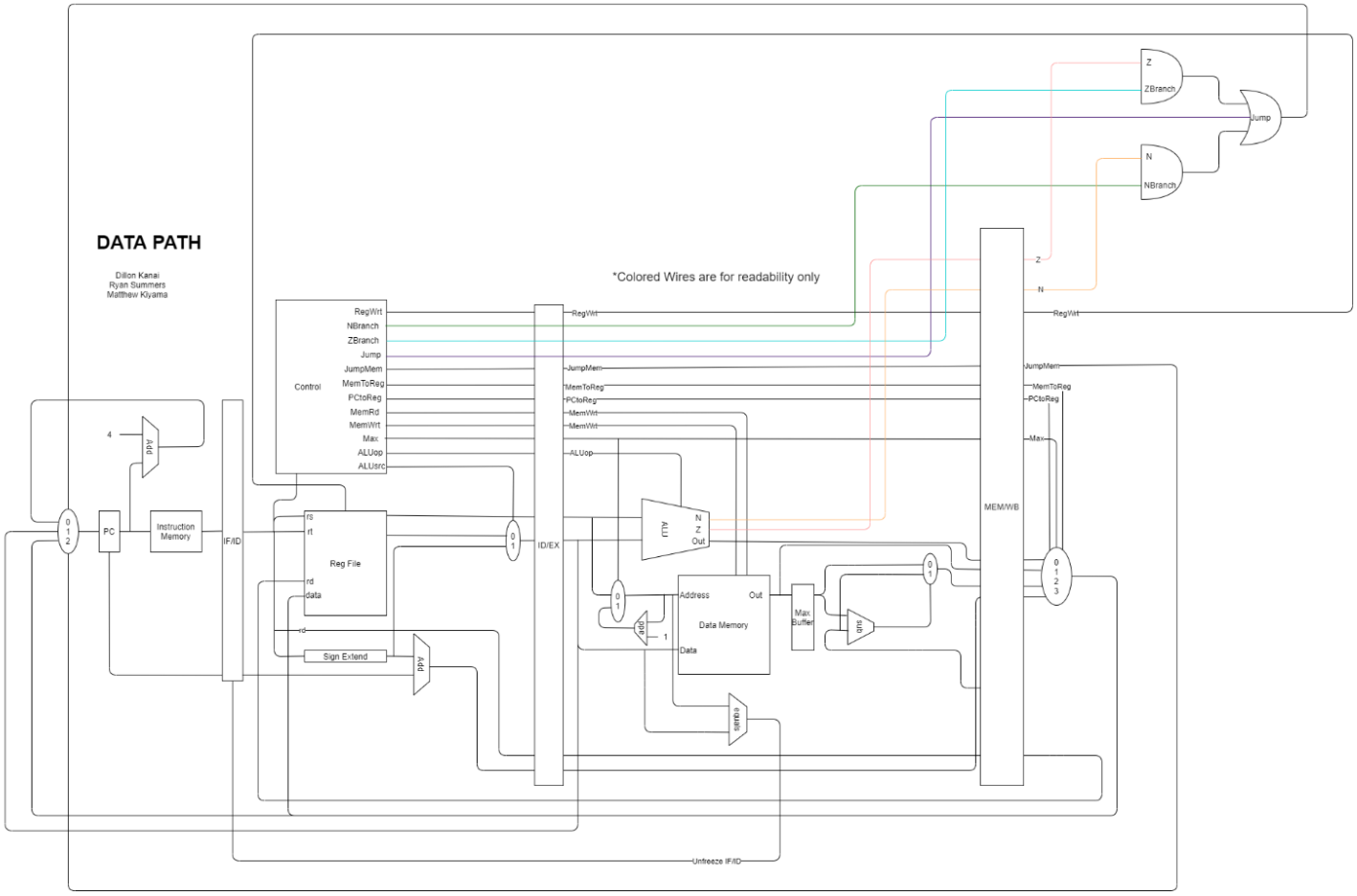
Description of CPU

(Description of hardware MAX not included)

Our datapath is controlled by a clock, using both the positive edge and the negative edge. The positive edge controls the modules while the negative controls the buffers. By alternating clock edges, we are able to pass along values through buffers in between each positive edge. This made it possible to sync up our data path. Our data path begins with the Program Counter (PC) module that keeps track of where we are at in the instruction memory. The PC then inputs into the instruction memory, the memory that holds all of the instructions we have to perform. After selecting the appropriate instruction based on the PC, we then output an instruction that is coded in binary. We then parse the whole instruction into parts and insert them into the control module and register module. The control module sets flags for future operations and the register module checks our registers and outputs their values. We also use sign extension for instructions that deal with constants and the adder for instructions that deal with PC. After passing these values through the ID/EX buffer, we do mathematical operations using the ALU and the ALUop flag. This sets the N and Z flag of the ALU and outputs the result of the mathematical operations. At the same time, we also access our data memory here. Based on the flags MemRd and MemWrt, the DataMemory will either write to memory or read from memory. There is also a wire branching off the rs wire right before the ALU, which is necessary for the branch and jump instructions to work. After passing through the MEM/WB buffer, the remaining flags will determine what will get outputted through the 3 to 1 mux: the ALU result, the information from data memory, or the PC + constant. Two more flags will determine what the 3 to 1 mux on the left will output to the PC: the next instruction PC, new jump/branch location, or the output of the 3 to 1 mux on the right.

Dillon Kanai
Matthew Kiyama
Ryan Summers

Datapath



Control Truth Table

Instruction	RegWrt	NBranch	ZBranch	Jump	JumpMem	MemToReg	PCtoReg	MemRead	MemWrt	MAX	ALUSrc	ALUOp
No Op	0	0	0	0	0	0	0	0	0	x	0	0011
Save PC	1	0	0	0	0	0	1	x	0	x	1	0000
Load	1	0	0	0	0	1	0	1	0	x	x	0011
Store	0	0	0	0	0	0	0	x	1	x	x	0011
Add	1	0	0	0	0	0	0	x	0	x	0	0000
Increment	1	0	0	0	0	0	0	x	0	x	1	0000
Negate	1	0	0	0	0	0	0	x	0	x	x	0001
Subtract	1	0	0	0	0	0	0	x	0	x	0	0010
Jump	0	0	0	1	0	x	0	x	0	x	x	0011
Branch If Zero	0	0	1	x	0	x	0	x	0	x	x	0011
Branch if Neg	0	1	0	x	0	x	0	x	0	x	x	0011
Jump Memory	0	0	0	1	1	x	0	1	0	x	x	0011
MAX	1	x	x	x	x	x	x	x	x	1	x	0011

Dillon Kanai
Matthew Kiyama
Ryan Summers

Simulation Verification

To prove that our datapath processes the last instruction necessary to complete MAX instruction.

The red circle shows the values of x4 (58324) and x6 (14800) registers in the “SUB x21 x4 x6” instruction. This iteration of this instruction happens near the end of the waveform. This picture proves that the code recognizes 58324 as the highest value. This also proves that the code pulls the last value of the 20 numbers in data memory, thus completing all of the iterations necessary for the MAX instructions.

Dillon Kanai
Matthew Kiyama
Ryan Summers

To prove that 58324 was passed to x4, which is our best so far register:

The two red circles show two instructions, the “LD x6 x0” and “INC x4 x6 0” instructions. x0 holds the value of the current index (the current number we are looking at in data memory). x4 is our best so far. In the first circle, we are accessing data memory and getting the next number, which is 58324. We are then passing this on to x6, which is shown in the row WB_rd. Since this is a higher number than our best so far, we then arrive at “INC x4, x6, 0”, where we will be replacing the value in x4 with the 58324, which is stored in x6. As we can see, 58324 is being passed into register x4, which can be seen in row WB_rd.

Dillon Kanai
Matthew Kiyama
Ryan Summers

Assembly Code

Assembly Version:

```
1    SVPC    x29, 3
2    SVPC    x30, 10
3    SVPC    x31, 7
4    SUB     x20, x0, x1
5    BRN     x30
6    NOP
7    NOP
8    LD      x6, x0
9    NOP
10   NOP
11   SUB     x21, x4, x6
12   BRN     x31
13   NOP
14   NOP
15   INC     x4, x6, 0
16   INC     x0, x0, 1
17   J       x29
18   NOP
19   NOP
20   NOP
```

Vivado Version:

```
//SVPC x29, 3
assign instructions[0] = 32'b11110111010000000000110000000000;

//SVPC x30, 20
assign instructions[1] = 32'b11110111100000000101000000000000;

//SVPC x31, 14
assign instructions[2] = 32'b11110111110000000011100000000000;


//SUB x20, x0, x1
assign instructions[3] = 32'b01110101000000000000010000000000;
```

Dillon Kanai
Matthew Kiyama
Ryan Summers

```
//BRN x30
assign instructions[4] = 32'b10110000000111100000000000000000;
assign instructions[5] = 32'b00000000000000000000000000000000;
assign instructions[6] = 32'b00000000000000000000000000000000;

//LD x6, x0
assign instructions[7] = 32'b11100001100000000000000000000000;
assign instructions[8] = 32'b00000000000000000000000000000000;
assign instructions[9] = 32'b00000000000000000000000000000000;

//SUB x21 x4, x6
assign instructions[10] = 32'b01110101010001000001100000000000;

//BRN x31
assign instructions[11] = 32'b10110000000111110000000000000000;
assign instructions[12] = 32'b00000000000000000000000000000000;
assign instructions[13] = 32'b00000000000000000000000000000000;

//INC x4, x6, 0
assign instructions[14] = 32'b01010001000001100000000000000000;
assign instructions[15] = 32'b00000000000000000000000000000000;

//INC x0, x0, 1
assign instructions[16] = 32'b01010000000000000000010000000000;
assign instructions[17] = 32'b00000000000000000000000000000000;

//J x29
assign instructions[18] = 32'b10000000000111010000000000000000;
assign instructions[19] = 32'b00000000000000000000000000000000;
assign instructions[20] = 32'b00000000000000000000000000000000;

//NOP
//assign instructions[21] = 32'b00000000000000000000000000000000;
```

Dillon Kanai
Matthew Kiyama
Ryan Summers

Estimate of Running Time

Given: "Delay of memory (I and D memory): 2 ns., delay of register file: 1.5 ns., delay of ALU (adders): 2 ns."

We have a clock size of 1ns and we execute 20 instructions total, 9 of which are no operation instructions and 11 of which are other. Our following is the estimate of running time using the given information.

Delay for 3 SVPC = 3*(2ns) = 6ns

Delay for each loop = 2(SUBs) + 2(BRNs) + LD + 2(INC) + 9(NOP) + J =

2*(5.5ns) + 2(3.5ns) + 5.5ns + 2(5.5ns) + 9(2ns) + 3.5ns = 56 ns per loop

Total delay = SVPC delay + loop delay * 20 = 6ns + 56ns * 20 = 1126ns

Our actual run time was 706ns.