Parker Newton
Immanuel Amirtharaj
Friday June 3, 2016
COEN 122

COEN 122 Final Project - Designing a Pipelined CPU

Part 1: Abstract

For this project, we designed a 32-bit pipelined CPU using Verilog HDL. This CPU is compatible with the SCU Instruction set architecture which contains thirteen distinct instructions. After creating the pipeline, we tested it with a sequence of instructions which finds the maximum of a set of given numbers.

Part 2: CPU Design Description

The CPU pipeline consists of five distinct stages as shown in Fig 1. In order to allow for concurrency, we have buffers whose sole purpose is to keep track of the different values in the different stages. In this section we will talk go into more depth about these five stages.

Instruction Fetch Stage

The Instruction fetch is the first stage in our pipelined CPU. In this stage, the PC counter is incremented and the current instruction is loaded into the IF/ID buffer.

Instruction Decode Stage

The main purpose of the instruction decode stage is to take the instruction from the fetch stage and extract the bit sequences necessary in order to carry out the rest of the stages. As shown in Fig. 1 bits 31-28 feed into the control which outputs the values for the WB, MEM, and EX into the ID/EX buffer. Bits 21-16 feed as addresses for register 1, bits 15-10 serve as an address for register 2, and bits 27-22 serve as the addresses for the register write. The R File uses the first two addresses to load values into registers R1 and R2 which are then pushed to the ID/EX buffer. In addition to all this, the PC's value from the IF/ID buffer is forwarded to the next buffer as well,

Execute Stage

In the execution stage, the ALU is used in order to compute a result. The inputs to the ALU as well as the control values both come from the ID/EX buffer. The inputs to the ALU are chosen by the ALUOp values which also choose which ALU operation to execute.

At this stage, the PC value from the previous buffer is taken and incremented by a constant from the previous buffer in the case that the instruction is the PC fetch instruction. This along with the ALU result are passed to the next buffer.

While the result of the ALU as well as the PC fetch address are forwarded to the EX/MEM stage, the N and Z bits are pushed back to the ID/EX buffer. This is because during the execution stage is also in charge of computing values which determine whether or not we need to jump or branch. The control values denoting that a branch/jump instruction is present are provided from the ID/EX buffer while the N and Z bits are used determine whether the branch/jump condition is met.

Memory Stage

The MEM stage is responsible for the access and storage of the data memory. and is written to the MEM/WB buffer. The other values forwarded to the buffer are the ALU computation, the PC value, and the WB control values. If not, the result from the ALU is forwarded to the MEM/WB buffer.

Write Back Stage

In this stage, we choose the correct value (PC value, ALU result, and the value from Data Memory) to write into the register in the second stage by using the control values. After the correct value is selected, it is written to the data write area in the Register memory back ins stage 2.

Miscellaneous Design Choices

- LDPC - Instead of adding muxes, we just used an adder to be more efficient

- AND/XOR Gates at BRZ, BRN, and JUMP - We only expect to have either a BRZ, BRN, and JUMP because we will have only one of those at a time, therefore an XOR coupled with two AND gates will produce the correct selection value

- A and B as inputs for ALU - In our ID/EX stage, we have two muxes which determine which input value goes in each register. We do this because in the subtraction instruction, the order of A and B needs to be switched

- In the register file memory unit, $0 always stores a value of 0.

Part 3: Test/benchmark

To prove that our pipelined CPU functioned correctly, we had to test four distinct instructions in the SCU ISA. In our case, we tested the JM, LDPC, and NEG instructions. In addition, we also tested out our MAX function to ensure that it functioned correctly.

JM Instruction

The JM instruction writes mem[$rs](mem[$1] = mem[17] = 18) back to the PC
As we can see in Fig. 4, the address is written at 17 with the value of 18.

LDPC Instruction

The LDPC instruction writes the value of the sum of the PC and immediate field (7 + 10 = 17) back into register 1 as shown in Fig. 5.

Max function

The max function finds the maximum value of elements in an array. As shown in Fig. 6, an array of size 4 with the element values [5, 2, 7, 1, 8, 5, 92, 100, 0, 4] is passed. The highlighted entry in Fig. 6 shows that the maximum value of 100 is written back to the data memory at address 100 as expected.

NEG Instruction

This instruction calculates the negative (2's complement) value of an input. As shown in Fig. 7, it takes an input of 17 and returns -17.

Part 4: Assembly Code for MAX Instruction

Our assembly code for the MAX function is shown below. In our implementation, this function takes in an array where the first value is the count, the latter values are the numbers for comparing, and the final element holds the maximum value of array which we calculate in the function.

Line #	Instruction	Comments	Line #	Instruction	Comments
0	**ADD 16, $4, $0**		18	**NOP**	
1	**NOP**		19	**ADD $4, $2, $2**	
2	**J $10**		20	**NOP**	
3	**NOP**		21	**NOP**	
4	**NOP**		22	**LD $4, $4**	
5	**NOP**		23	**NOP**	
6	**LD $1, $2, # !1 <- a[0]**	# $1 <- A[0]	24	**NOP**	
7	**LD $3, $4 # $3 <- n**	# $3 <- n	25	**SUB $9, $6, $4**	
8	**ADD $5, $0, $0**		26	**BRN $0**	# $0 <- 0
9	**ADD $6, $1, $0**	#max <- a[0]	27	**NOP**	
10	**NOP**		28	**NOP**	
11	**INC $5, $5**		29	**NOP**	
12	**NOP**		30	**J $10**	# $10 <- 11
13	**NOP**		31	**NOP**	
14	**SUB $4, $3, $5**		32	**NOP**	
15	**BRZ $7**	# $7 <- 34	33	**NOP**	
16	**NOP**		34	**ST $6, $11**	MEM[$11] Write the max here
17	**NOP**				

Stage Delays

For the cycles delays, we are given that the delays of I = 2ns, D = 2ns, R = 1.5ns, and ALU (adders) = 2ns. From this, we found the delay by finding the path with the highest delay in each individual stage.

Cycle	Delay (nanoseconds)
IF	2
ID	1.5
EX	2
MEM	2
WB	1.5

Part 5: Duration of the Max Instruction

To determine the max instruction, we need to compute the number of steps taken and then multiply that by the cycle time given to us in order to find the CPU time. Based on assembly code above, we can easily calculate this. The initialization condition as well as the terminating condition are both 9 steps each. While iterating through the array, if a value is higher than the temporary max register, we need to reassign the max register with the new value which takes up 25 steps, otherwise a normal iteration only takes 23 steps. In this case, our calculations are as shown.

$$\textit{Length of initialization stage} = 23$$
$$\textit{Length of finishing stage} = 23$$
$$\textit{Number of steps without reassignment} = 23$$
$$\textit{Number of steps with reassignment} = 25$$
$$\textit{Total Number of Steps} = 9 + 23 + 25 + 23 + 25 + 23 + 25 + 25 + 23 + 23 + 9 = 230 \textit{ steps}$$

Now that we have the number of steps, we find the CPU time by multiplying the steps by cycle time given to us

$$\textit{Cycle Time} = 2 \textit{ nanoseconds}$$
$$\textit{CPU Time} = 230 * 2 = 460 \textit{ nanoseconds}$$

Therefore, the CPU time of the max instruction is **460 nanoseconds**.

Appendix

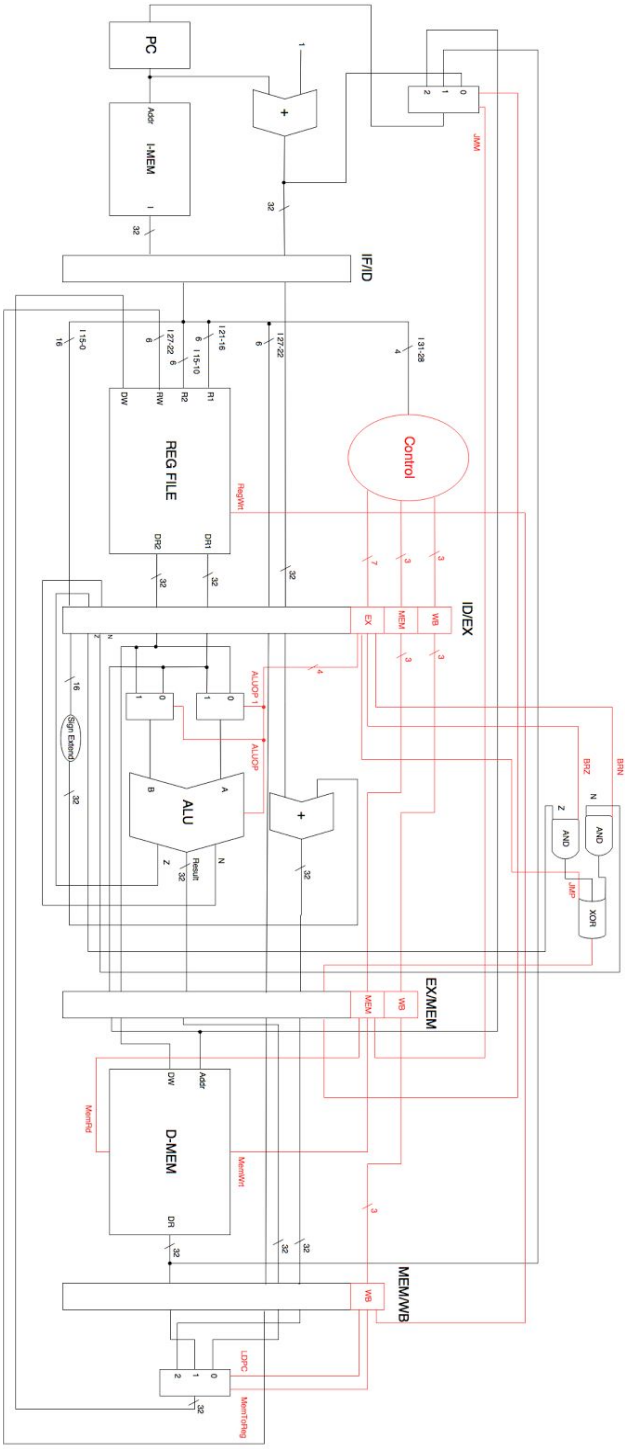

Fig. 1: This is the datapath for our pipelined CPU. It is consisted of 5 stages and supports the SCU Instruction set architecture.

I_{31}	I_{30}	I_{29}	I_{28}	ALUOP(4)	BRN	BRZ	JMP	JMM	MEMWrt	MemRd	RegWrt	LDPC	MemToReg
0	0	0	0	0000	0	0	0	0	0	0	0	X	X
1	1	1	1	0000	0	0	0	0	0	0	1	1	0
1	1	1	0	0000	0	0	0	0	0	1	1	0	1
0	0	1	1	0000	0	0	0	0	1	0	0	X	X
0	1	0	0	1000	0	0	0	0	0	0	1	0	0
0	1	0	1	0100	0	0	0	0	0	0	1	0	0
0	1	1	0	0010	0	0	0	0	0	0	1	0	0
0	1	1	1	0001	0	0	0	0	0	0	1	0	0
1	0	0	0	0000	0	0	1	0	0	1	0	X	X
1	0	0	1	0000	0	1	0	0	0	0	0	X	X
1	0	1	0	0000	0	0	0	1	0	0	0	X	X
1	0	1	1	0000	1	0	0	0	0	0	0	X	X

Fig 2. The truth table we generated for each of the SCU ISA instructions. This was the first step in deciding how to implement the control in our datapath.

$$ALUOP_3 = \overline{I_{31}}\, I_{30}\, \overline{I_{29}}\, \overline{I_{28}}$$
$$ALUOP_2 = \overline{I_{31}}\, I_{30}\, \overline{I_{29}}\, I_{28}$$
$$ALUOP_1 = \overline{I_{31}}\, I_{30}\, I_{29}\, \overline{I_{28}}$$
$$ALUOP_0 = \overline{I_{31}}\, I_{30}\, I_{29}\, I_{28}$$
$$LDPC = I_{31}\, I_{30}\, I_{29}\, I_{28}$$
$$BRN = I_{31}\, \overline{I_{30}}\, I_{29}\, I_{28}$$
$$BRZ = I_{31}\, \overline{I_{30}}\, \overline{I_{29}}\, I_{28}$$
$$JMP = I_{31}\, \overline{I_{30}}\, \overline{I_{29}}\, \overline{I_{28}}$$
$$JMM = I_{31}\, I_{29}\, \overline{I_{28}}$$
$$MemToReg = I_{31}\, I_{30}\, I_{29}\, \overline{I_{28}}$$
$$MemWrt = \overline{I_{31}}\, \overline{I_{30}}\, I_{29}\, I_{28}$$
$$MemRd = I_{31}\, I_{30}\, I_{29}\, \overline{I_{28}}$$
$$RegWrt = I_{31}\, I_{30}\, I_{29} + \overline{I_{31}}\, I_{30}\, ((I_{29} \oplus I_{28}) + \overline{(I_{29} + I_{28})} + I_{29}\, I_{28})$$

Fig. 3: The minterms we generated from the truth table in Fig. 2. These values are used in our implementation of the control in our datapath.

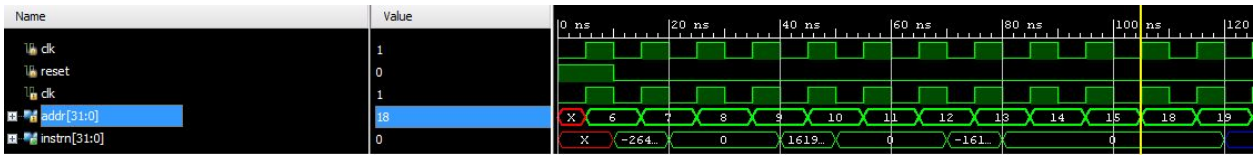

Fig 4: The JM instruction writes mem[$rs] (mem[$1] = mem[17] = 18) back to the PC.

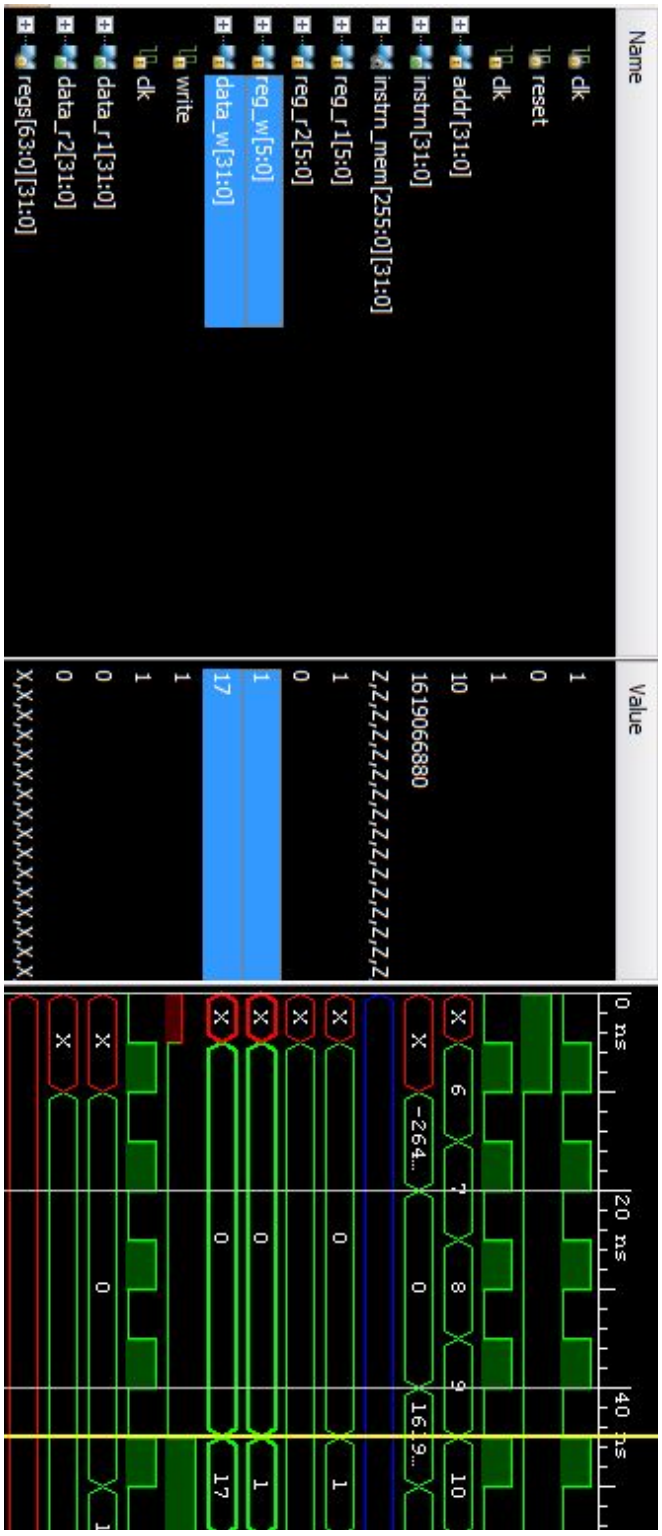

Fig 5: The LDPC instruction writes the value of the sum of the PC and immediate field (7 + 10 = 17) back to register 1.

Fig. 6: A program that finds the maximum value of the elements in an array. The array of size 4 has the following element values: [5, 2, 7, 1, 8, 5, 92, 100, 0, 4]. From the highlighted entry, it is shown that the maximum value of 100 is written back to data memory at address 100.

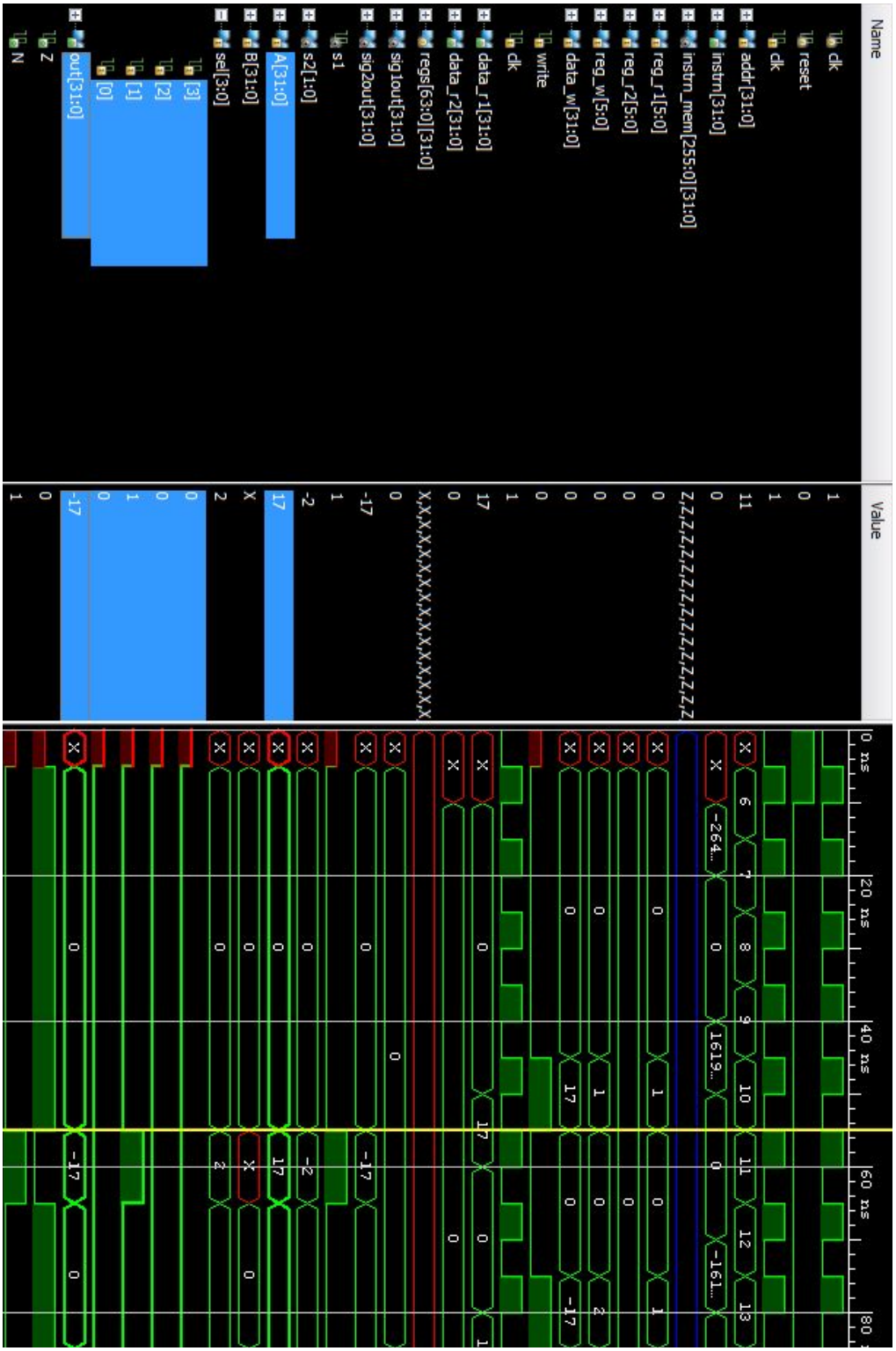

Fig. 7: The NEG instruction calculates the 2's complement negation of $rs (-17). As shown in the highlighed entries, the input A, the output, and the selection values illustrate this.