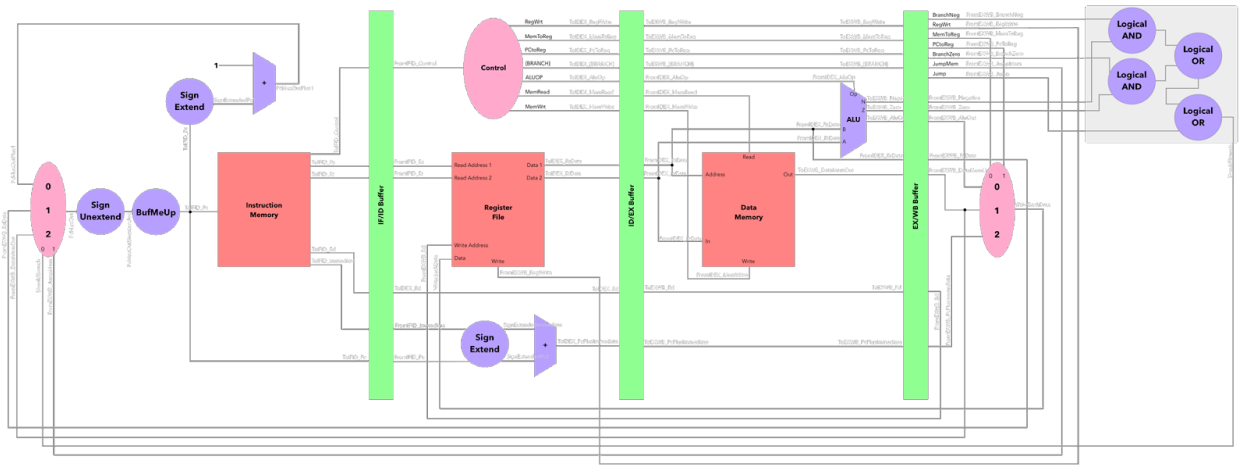
Ronak Gajrawala
Ryan Ku
Yutong Li

COEN 122L: Project, Version with MAX
Design a Structural Model of a Pipelined CPU

1. Abstract

In this project, we designed a datapath and truth table for a 32-bit pipelined CPU using a 13-instruction set architecture (SCU-ISA) based on RISC V. We implemented the CPU design by programming in Verilog HDL using Xilinx's Vivado IDE.

2. **Detailed description of the CPU design including the datapath and the truth table of the control.**



CPU Datapath Design

	DON'T CARE IF:						MemRead: 0						PCToReg: 0 & MemToReg: 0 || RegWrt: 0					
	Instruction	Op3	Op2	Op1	Op0	RegWrt	MemToReg	PCToReg	BranchNeg	BranchZero	Jump	JumpMem	ADD	INC	NEG	SUB	MemRead	MemWrt
0	NOP	0	0	0	0	0	X	X	0	0	0	0	X	X	X	X	0	0
1	MAX	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0
2		0	0	1	0													
3	ST	0	0	1	1	0	X	X	0	0	0	0	X	X	X	X	0	1
4	ADD	0	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0
5	INC	0	1	0	1	1	0	0	0	0	0	0	0	1	0	0	0	0
6	NEG	0	1	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0
7	SUB	0	1	1	1	1	0	0	0	0	0	0	0	0	0	1	0	0
8	J	1	0	0	0	0	X	X	0	0	1	0	X	X	X	X	0	0
9	BRZ	1	0	0	1	0	X	X	0	1	0	0	X	X	X	X	0	0
10	JM	1	0	1	0	0	0	0	0	0	0	1	X	X	X	X	1	0
11	BRN	1	0	1	1	0	X	X	1	0	0	0	X	X	X	X	0	0
12		1	1	0	0													
13		1	1	0	1													
14	LD	1	1	1	0	1	1	0	0	0	0	0	X	X	X	X	1	0
15	SVPC	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0

Truth Table of the Control

For the CPU datapath design, we followed the example given in the lab for the pipelined version of the RISC V architecture ignoring the hazard detection and forwarding units. This made sense to us because our SCU-ISA architecture was a slightly modified version of the RISC V architecture.

To optimize CPI, we decided that buffers would flush on the positive edge of the clock while components between buffers would update on the negative edge of the clock. So in one clock cycle, the pipeline would move forward by one stage. There are four stages in our datapath: PC/instruction, register, data, and write-back.

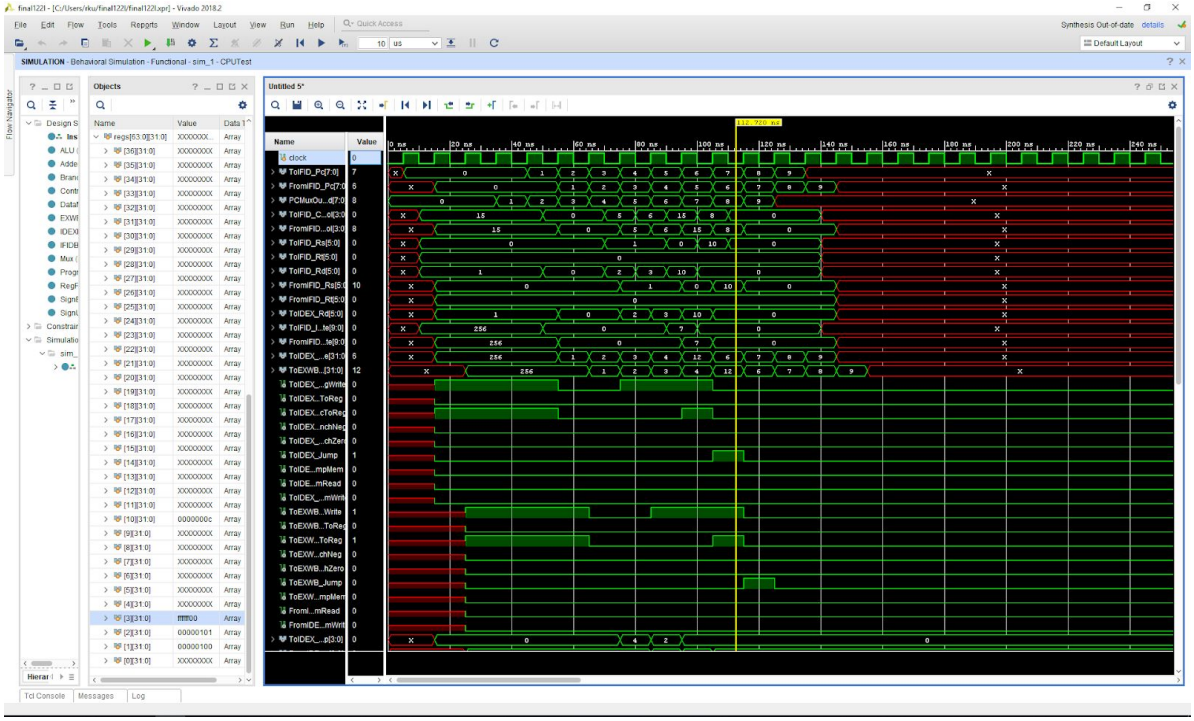
In the PC/instruction stage, PC is set depending on if the instruction branches or not. If the instruction is a non-branching one, then the PC is incremented by one. Otherwise, depending on the instruction, PC is set to the output from data memory or the output of adding previous PC and the immediate. We had to add a buffered component here, *BufMeUp*, to update PC only on negative clock edge. This was because the *PC + 1* path was short-circuiting the multiplexer that chooses what PC should be set to. Next, PC is sent to the instruction memory component, which fetches the instruction associated with PC and sends the broken-down instruction (opcode, rd, rs, rt, and immediate) to the IF/ID buffer.

In the register stage, the opcode is analyzed by the Control unit, which sets flags accordingly. The register file reads the data at register locations rs, rt and outputs the register data to the next buffer. If the write-back flag is set, the register file will also write the specified data back to the specified register location. Finally, we add PC and the immediate together in this stage.

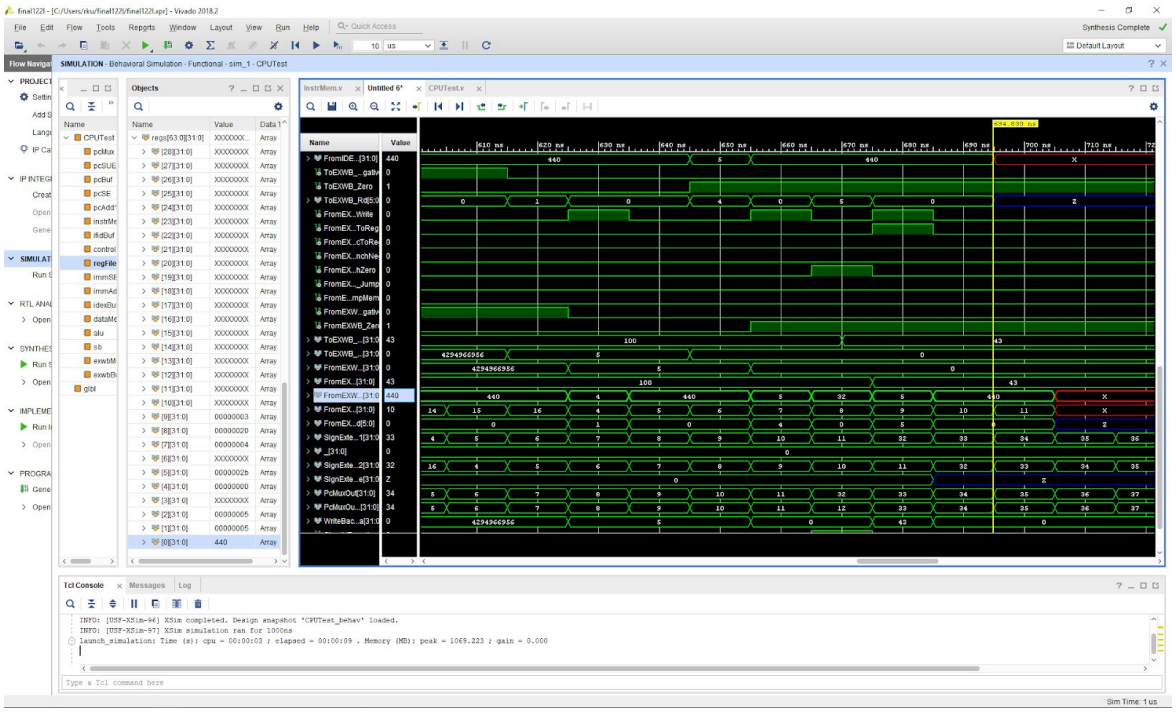
In the data memory stage, the data memory component will read the memory at the specified location if the read flag is set. The data memory component will also write to the memory at the specified location if the write flag is set. The ALU also lives in this stage and computes arithmetic needed to update PC and check if the conditions are right for conditional branching.

Finally, the write-back stage is where computations are made for data that is written or used in previous stages of the pipeline. For example, the register write-back mux determines what data should be written back to the register file. The logic at the top right of the datapath determines if the CPU should branch. The *shouldBranch* data is then sent all the way back to the PC/instruction stage to choose what PC should be updated to.

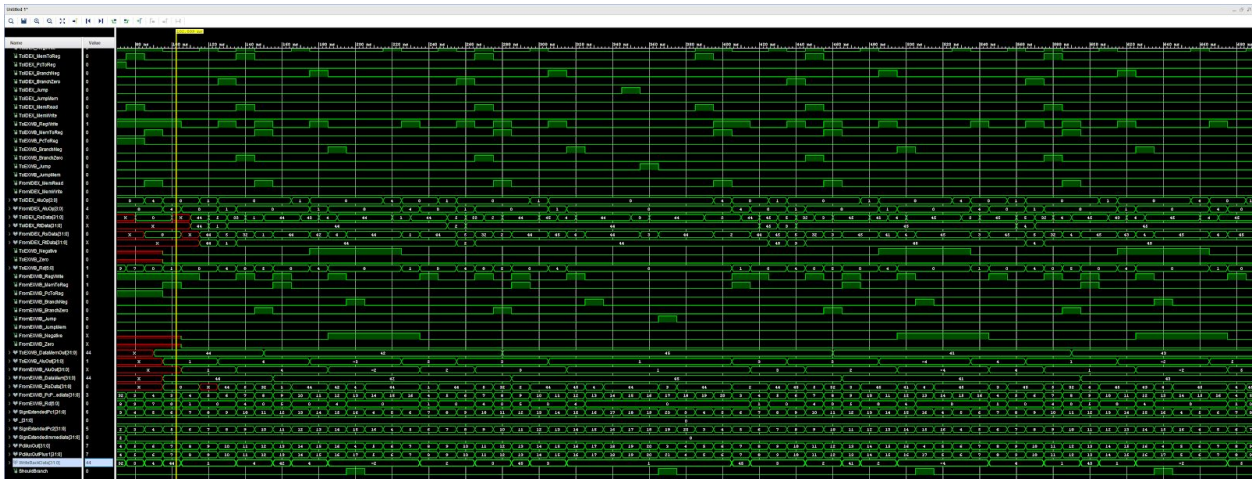
3. Test benchmarks/waveforms verifying the functions.



Demo Part 2



Max (Easy to See)



Max (Full Run)

In the waveform for the Max test, you can verify that the correct maximum value was written by following the *FromEXWB_RegWrite*, *FromEXWB_Rd*, and *WriteBackData* wires. A new max is written in the instruction *LD R0,R1* (R0=M[R1]). At this time, *FromEXWB_RegWrite* should be in a high state since we are writing register data in this instruction. In addition, *FromEXWB_Rd* should be 0 since *R0* is *rd*. Finally, *WriteBackData* should contain the data being written back, i.e. the new max element. In the waveform, for the array *{44, 42, 45, 41, 43}*, you can see that there are two new maxes being written: the first element, 44, and the largest element, 45. The first element is written to initialize the max element. The elements that are less than the max element are only written to the temp register, but are not saved in *R0* (max). This means that *R0* will be 45 by the end of the program.

4. **Assembly code for calculating the max in (1).**

```
SVPC    R8,32       // R8 <-- address of loop exit
SVPC    R9,2        // include LD R0,R1
SVPC    R7,2        // R7 <-- address of next instruction for looping
LD      R0,R1       // R0 = M[R1]
INC     R1,R1       // increment RS (address) to check next element
NOP                 // read after write on R1
NOP
SUB     R4,R2,R1    // R4 <-- count (last - first)
BRZ     R8          // branch to loop exit (R7) if count = 0 --> last element already
                    // checked in prev. iteration
LD      R5,R1       // R5 = M[R1]
NOP                 // read after write on R5
NOP
SUB     R4,R5,R0    // R4 <-- R5 - R0; R4 will be negative if temp max is greater
BRN     R7          // if temp max is greater than new element, check next element
NOP                 // stall 3 cycles for BRN
NOP
NOP
J       R9          // check next element
NOP                 // stall 4 cycles for J
NOP
NOP
NOP
```

5. **Estimate the time need to execute your code for (1) based on your CPU design, and verify your estimate with simulation/waveform.**

The longest be 2ns per cycle at most. The most confusing parts are the register file buffer zone and the data memory buffer zone, but since the adders/ALUs are in parallel with the register file and data memory components, they are still only limited to 2ns. As such, since this is the maximum cycle length needed for any given buffer, limiting each cycle length to 2ns works in this case, resulting in an 8ns code execution time (per instruction). This leads to a $22 \times 2 + 1 \times (8 - 2) = 54$ ns execution time for the max instruction, assuming the the array only has 1 element. Should an array has n elements, the formula for execution time should be as follows: $3 \times 2 + 19 \times 2 \times n + 1 \times (8 - 2) = 12 + 38n$.