CompArch Lab 4

Vivien Chen, Lauren Pudvan, Samantha Young

November 16, 2018

0.1 Schematic

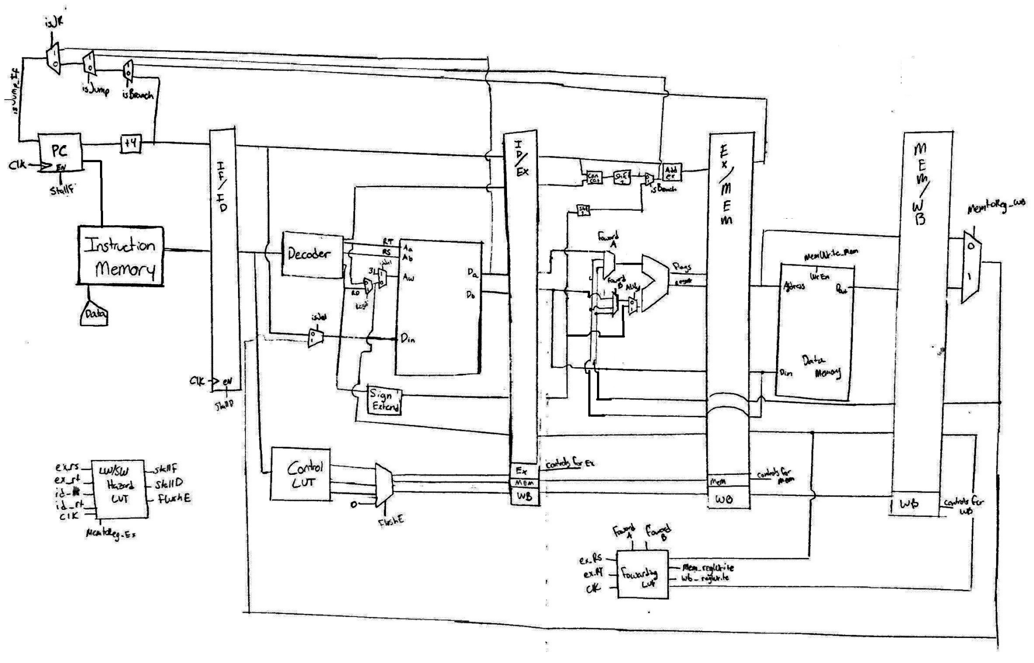


Figure 1: This is a diagram of our pipelined CPU Schematic.

0.2 Description

The pipelined CPU is divided into five stages: instruction fetch (IF), instruction decode (ID), execution (EX), memory (MEM), and write back (WB). A register exists between stages to hold the values of wires and control lines for a single clock cycle before the next stage, thereby allowing instructions to be processed in parallel at each stage, with the second instruction being fetched as the first instruction is being decoded, and so on.

IF stage: At the first PC count on the first clock cycle, the first instruction is fetched from instruction memory, which is fed into the IF/ID register to be processed at the ID stage in the next clock cycle. PC plus 4, which drives the counter to fetch the next instruction, is calculated at this stage. It is also fed into the IF/ID register to be used at a later stage to calculate the branch and jump addresses. There are three muxes to choose which PC count to look at: PC plus 4, branch address for branching, jump address for jumping, or register address for jump register. Since our CPU currently does not have branching or jumping capabilities, these muxes are always set to 0 and take PC plus 4 as the next PC count. If there is a LW hazard, calculated in the next stages, the PC and IF/ID registers are stalled to allow the LW hazard to continue for a cycle with the next instruction held.

ID stage: On the second clock cycle, the instruction decoder decodes the first instruction. Depending on if the instruction type is J- (currently unsupported), I-, or R-type, the appropriate registers are read from the register file and the immediates are sign extended. These, along with register destination used for the WB stage, are fed into the ID/EX register to be processed later. The control LUT also takes in the OP and FUNCT codes from the decoded instruction to output the appropriate control lines, also fed into the ID/EX register. If a LW hazard is detected, all the control lines are flushed or set to 0 before they reach the ID/EX register to generate a no-op between the LW instruction and the next instruction.

EX stage: On the third clock cycle, the ALU calculates the appropriate data or address using the appropriate ALU function, depending on the ALUsrc and ALUctrl lines previous looked up in the control LUT for the instruction. The control lines needed for this stage are used here and the rest are fed into the EX/MEM register to be used at the next two stages. The output of the ALU, as well as the register destinations previously determined, are also pushed forward. In addition, there are two muxes that determine the operands of the ALU, which take the values from the MEM or WB stages when a forwarding hazard is determined by the forwarding LUT. Otherwise, the normal operands from this stage's instruction are used. LW hazards are also determined from the Rs and Rt registers in this stage, as well as from the ID stage. The LUT outputs the appropriate stalling and flushing control lines to stall the PC and IF/ID register and flush the ID/EX register to introduce the no-op.

MEM stage: On the fourth clock cycle, the control lines for the MEM stage are used here and the final control lines for the WB stage are fed into the MEM/WB register. Data may be written or read from data memory, which exists in the same memory module as instruction memory, depending on the control lines previously determined for this stage, i.e. whether the instruction is SW or LW. The data read from memory is fed into the intermediate register, as is the output of the ALU and the register destination previously determined.

WB stage: On the fifth and final clock cycle for a single instruction, a mux determines whether the data written to the register, if at all, depending on the RegWr control line, is from memory (LW) or from the ALU (ALU MIPS functions). The forwarding LUT takes the register destinations and control lines from the WB and the MEM stage, as well as the registers from the EX stage, to determining the forwarding necessary.

When thinking through different implementations of MIPS code on the above pipeline CPU schematic we ran into some problems. We broke these hazards into 3 types, Data Hazards, Structual Hazards and Control/Branch hazards. In order to mitigate data hazards we took advantage of "forwarding". This is done in hardware by allowing future instructions to access a value computed by the ALU before it is stored in memory. To deal with structural hazards we are adding stalling.

Forwarding is utilized when we need to access a register that is in the process of being being written to, this is called a data hazard. Instead of waiting for the CPU to finish writing to this register, we pull the value that will be written to it from the section of the CPU the writing instruction is computing. This is controlled by a the Forwarding LUT (section 0.3.2) and some muxs. Stalling utilizes a LUT to stall all the controls of the CPU to stall some instructions to add time between instructions that would cause a hazard. This is called a structural hazard because it affects the control of the CPU.

Our pipelined CPU can run the following MIPS instructions: LW, SW, XORI, ADDI, ADD, SUB, and SLT.

0.3 LUTs

0.3.1 Instruction LUT

	RegDst	RegWr	MemWr	MemToReg	ALUctrl	ALUsrc	IsJump	IsJAL	IsJR	IsBranch
LW	0	1	0	1	000	1	0	0	0	0
SW	0	0	1	0	000	1	0	0	0	0
XORI	0	1	0	0	010	1	0	0	0	0
ADDI	0	1	0	0	000	1	0	0	0	0
ADD	1	1	0	0	000	0	0	0	0	0
SUB	1	1	0	0	001	0	0	0	0	0
SLT	1	1	0	0	011	0	0	0	0	0

*If the zero flag is 1 and overflow is 0, IsBranch is set to 1 for BEQ and 0 for BNE; otherwise, it is 0 for BEQ and 1 for BNE.

0.3.2 Forwarding LUT

mem_regAw	mem_regWrite	wb_regAw	wb_regWrite	ex_rs	ForwardA
!00000	1			mem_regAw	10
!ex_rs		!00000	1	wb_regAw	01
					00

mem_regAw	mem_regWrite	wb_regAw	wb_regWrite	ex_rt	ForwardB
!00000	1			mem_regAw	10
!ex_rt		!00000	1	wb_regAw	01
					00

ForwardA controls the first ALU operand and ForwardB controls the second ALU operand. 01 is when the operand needs to be pulled from the data memory or an earlier ALU result. 10 is when the operand is from the prior ALU result. Lastly, 00 occurs when the operand is from the register file. The full MUX mappping for these two forwarding muxes are as follows:1.

Mux Control	Source	Explanation
ForwardA = 00	EX	The first ALU operand comes from the register file.
ForwardA = 10	MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	EX	The second ALU operand comes from the register file.
ForwardB = 10	MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Table 1: Forwarding sources

0.3.3 Hazard Detection LUT

id_rs	id_rt	MemToReg_Ex	FlushE	StallD	StallF
ex_rt	ex_rt	1	1	0	0
ex_rt	!ex_rt	1	1	0	0
!ex_rt	ex_rt	1	1	0	0
!ex_rt	!ex_rt	1	0	1	1
ex_rt	ex_rt	0	0	1	1
ex_rt	!ex_rt	0	0	1	1
!ex_rt	ex_rt	0	0	1	1
!ex_rt	!ex_rt	0	0	1	1

1 Testing Plans

From our single cycle CPU we learned to incrementally test our CPU through its various stages of development in order to catch errors with each new level of complexity. We built our CPU first to handle no hazards at all, just converting the single cycle CPU to pipeline in order to increase the throughput of the instructions that did not cause errors. We wrote a test bench to test the functionality of all of the ALU operations using unique registers. This would confirm that our CPU was functioning properly because it would run into no hazards.

aluTest.asm test the basic non hazard funcitonalities of the pipeline cpu. This tests the BEQ, ADDI, XORI, and SLT functionalities of the CPU. Where the final

register looks like:
$t0$ = 32h’3
$t1$ = 32h’2
$t2$ = 32h’3
$t3$ = 32h’2
$t4$ = 32h’5
$t5$ = 32h’6

We then added forwarding capabilities which dealt with data hazards.

forwarding2.asm tests each forwarding scenario in the pipeline CPU. This tests the ADD instructions and each forwarding instruction. The forwarding test bench ensured that we could compute instructions that required data from the traditional path directly from the execute stage, the forwarded from the memory stage and forwarded from the write back stage. From this test bench we realized we had to change our logic for the forwarding look up table because the code implementation did not output the intended logic.

This test bench test forwarding from one stage prior, forwarding from two stages prior, not forwarding, and then renabling of forwarding.

The final register looks like:
$t0$ = 32h’3
$t1$ = 32h’a
$t2$ = 32h’8
$t3$ = 32h’4
$t4$ = 32h’5
$t5$ = 32h’6
$t6$ = 32h’8

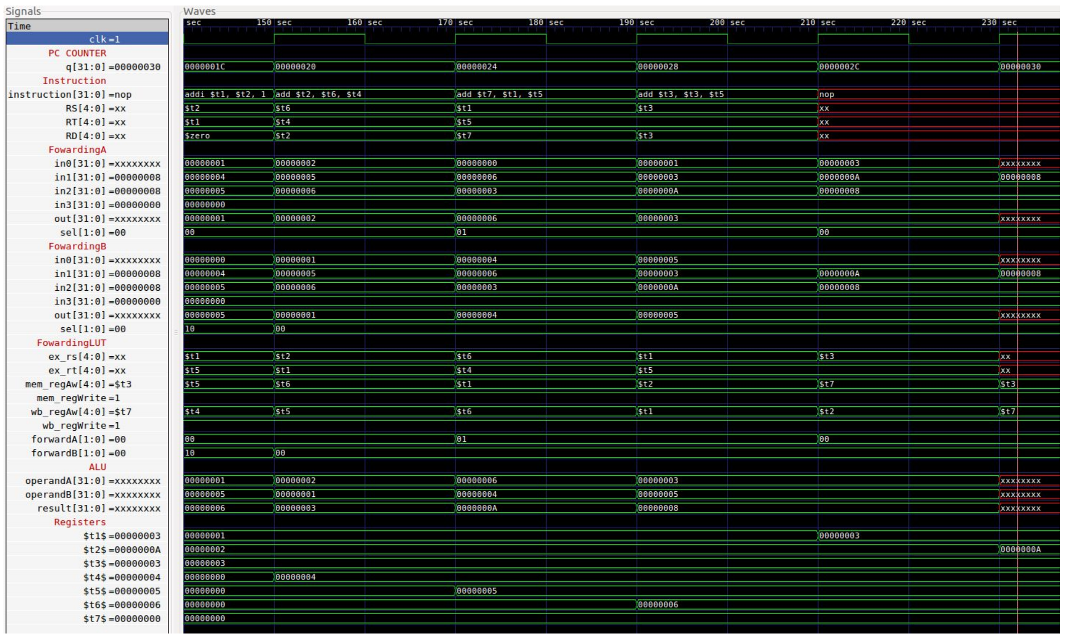


Figure 2: GTKwave of the forwarding test. The outputs of the Forwarding look up table change based on what data from what stage needs to be used in computation. (A larger version of this figure is on the last page.)

Next we looked at load/store structural hazards. We added stalling functionality to our CPU in order to give instructions enough time to prevent the next instruction from needing the same hardware. This was evident in LW instructions were information needed to be retrieved from memory and the next instruction had to be stalled in order to use that newly loaded data.

The assembly file for the load/store test is LWHazard.asm. It sets a stack pointer. Stores a word, loads it and does instructions that use that data.

The final register looks like:

\$*t*0\$ = 32h'7

\$*t*1\$ = 32h'0

\$*t*2\$ = 32h'2

\$*t*3\$ = 32h'3

\$*t*4\$ = 32h'5

\$*s*0\$ = 32h'2

This test allowed us to find problems in our forwarding schematic logic. Originally we had the two forwarding muxes right before the ALU and after the mux to determine if sign extend was needed. However in this test case we found when we loaded a word and needed it forwarded back this logic did not output the desired answer. By swapping the location of these muxes we were able to both take in the properly forwarded data and do ALU operations on the necessary data.

Each module used in the CPU also had its own test bench to ensure it worked as intended.

The strategy of making small test benches that test specific errors helped us debug our verilog faster because we were able to know where the error was.

2 Performance Analysis

A pipelined CPU will have a faster throughput (rate of instruction execution). The latency (time it takes to compute a single instruction) will increase slightly because of the addition of muxs and LUTs. Through pipelining the CPU we are not improving the time it takes to complete a single instruction. Instead we are allowing multiple instructions to be processing at once. This creates the opportunity for a faster throughput.

With our current pipelined CPU. We have the potential to have almost a 4 time greater throughput compared to our CPU from Lab 3. This is because our pipelined CPU has 4 stages for instructions to be passed through. The worst case is that there are many hazards and our pipelined CPU stalls with every instruction. In this case, the pipelined CPU operates with almost same throughput as our Lab 3 CPU (just a little slower).

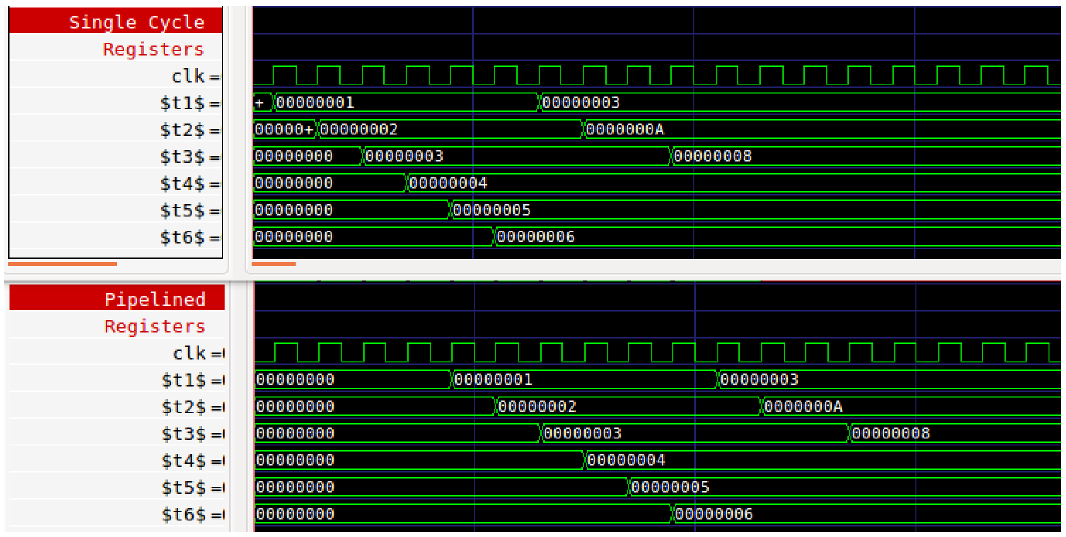


Figure 3: Single-cycle vs pipelined CPU performance on forwarding test

3 Resources Utilized

We spent a lot of time going through the companion readings. We utilized Patterson 4.6 and Harris 7.5 to learn about hazards. We also utilized Dr. Benjamin Hill Ph.D. during his office hours. We took advantage of Ariana Olson's NINJA hours to make a make file.

4 Work Plan Reflection

Planning	Expected Time	Actual Time
Research	0 hours	4 hours
Schematic	3 hours	3 hours
Written LUTs	1 hour	2 hours
Check in w/ Ben	0.5 hour	0.25 hour
Create Organized Repository	0.5 hour	0.5 hour
Total	5 hours	9.75 hours

Verilog CPU	Expected Time	Actual Time
Top Level Modules	3-4 hours	5 hours
LUT	1 hour	2 hours
Sub-modules	1-2 hours	1 hour
Verilinting	1 hour	2 hours
Forwarding and Hazard Control	0 hours	6 hours
Total	6-8 hours	16 hours

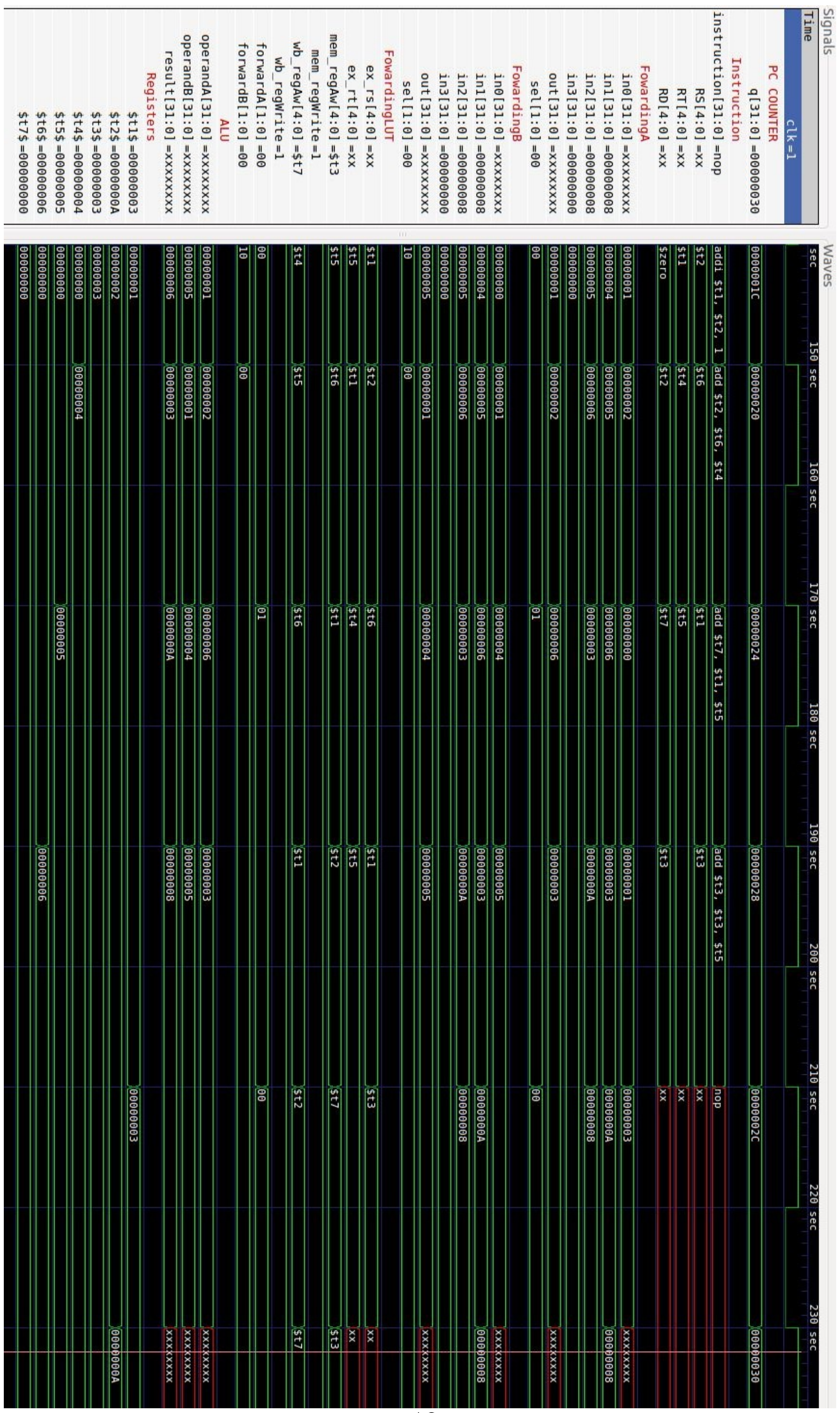
Testing	Expected Time	Actual Time
Implement Test Benches	3 hours	4 hours
Design Dynamic Test Benches	1 hour	0 hours
Debugging	3 hours	6 hours
Total	7 hours	10 hours

Final Countdown	Expected Time	Actual Time
Write Make Files	2 hours	0.5 hour
Performance Analysis	2 hours	0.5 hour
Write Report	2 hours	4 hours
Total	6 hours	5 hours

The total time we expected this lab to take was 26 hours, it actually took 40.75 hours. The time it took to plan was scoped well for the times we allotted time for. We failed to plan time for researching hazard and implementing and this is one of the tings that threw off our scoping. This shortsight was because we made this gameplan before we entirely understood how pipeline CPUs worked. The time budgeting for verilog was very under scoped because we originally did not intend to do forwarding and stalling. Implementing these took a lot of time. Our testing and debugging time was much more than expected, since we added on forwarding and stalling. Since those took more time we decided to not have dynamic test benches. Overall we are much happier that we got to learn more about pipelining a CPU, but this did come at the cost of not learning how to make dynamic test benches. The time it took for polishing off this lab is roughly what we planned. Overall the error in our scoping was not allotting time for some tasks that took a lot of time to execute.

We to attempt adding branching to our pipelined CPU. We did not succeed at this in the time budgeted so we moved onto writing the report and making the cleaning the repository for future use.These three hours of work are not shown in the tables above but we thought it would be valuable to mention our efforts.

5 Larger Image forwarding testbench GTK Wave



```
Time
clk =1
PC COUNTER
q[31:0] =00000030
Instruction
instruction[31:0] =nop
RS[4:0] =xx
RT[4:0] =xx
RD[4:0] =xx
FowardingA
in0[31:0] =xxxxxxxx
in1[31:0] =00000008
in2[31:0] =00000008
in3[31:0] =00000000
out[31:0] =xxxxxxxx
sel[1:0] =00
FowardingB
in0[31:0] =xxxxxxxx
in1[31:0] =00000008
in2[31:0] =00000008
in3[31:0] =00000000
out[31:0] =xxxxxxxx
sel[1:0] =00
FowardingLUT
ex_rs[4:0] =xx
ex_rt[4:0] =xx
mem_regAw[4:0] =$t3
mem_regWrite =1
wb_regAw[4:0] =$t7
wb_regWrite =1
forwardA[1:0] =00
forwardB[1:0] =00
ALU
operandA[31:0] =xxxxxxxx
operandB[31:0] =xxxxxxxx
result[31:0] =xxxxxxxx
Registers
$t1$ =00000003
$t2$ =0000000A
$t3$ =00000003
$t4$ =00000004
$t5$ =00000005
$t6$ =00000006
$t7$ =00000000
```