Laboratory 10
Control Path,Single-Cycle and Pipelined CPU

Computer Science 240

In the last lab, you used LogicWorks to design and implement all the main components of the datapath for the Mini-MIPS machine. In this lab, you will also investigate how to produce the control signals for the machine. You will then be able to observe the operation of the completely connected single-cycle CPU, by writing and executing some small programs.

ALU Control Unit

Exercise 1: The **ALUop** bits (which control the operation of the ALU) are not the same as the **opcode** (which specifies which instruction is being executed). It is necessary to design an ALU Control Unit to translate the opcode to the proper ALU operation for each instruction.

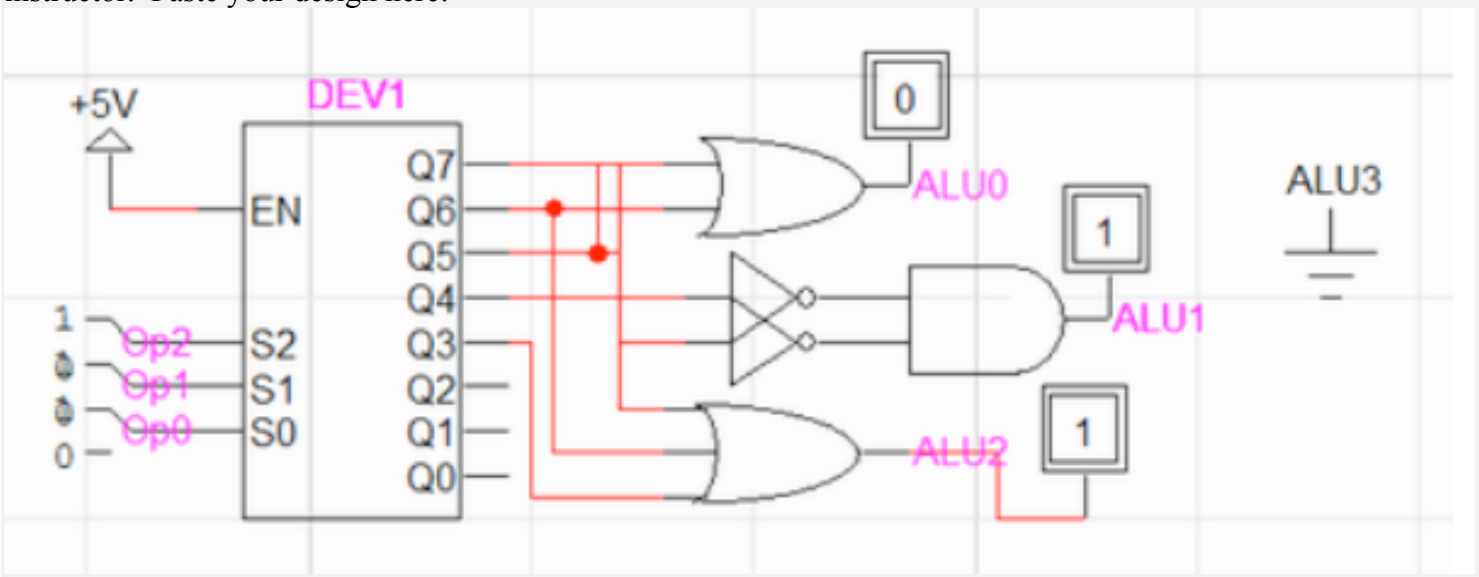
The ALU Control Unit for the Lab machine is somewhat simpler than the one discussed in lecture. Here is the truth table:

Instruction	Opcode	ALU operation	ALUop
LW	0	add	2
SW	1	add	2
ADD	2	add	2
SUB	3	sub	6
AND	4	and	0
OR	5	or	1
SLT	6	slt	7
BEQ	7	sub	6
JMP	8	don't care	don't care

So, you must design a circuit for the following:

Op3	Op2	Op1	Op0	ALUop3	ALUop2	ALUop1	ALUop0
0	0	0	0	0	0	1	0
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	0
0	0	1	1	0	1	1	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	1	1	1
0	1	1	1	0	1	1	0

Use a 3x8 decoder to produce the ALUop bits. Implement in LogicWorks , test, and demonstrate to the instructor. Paste your design here:

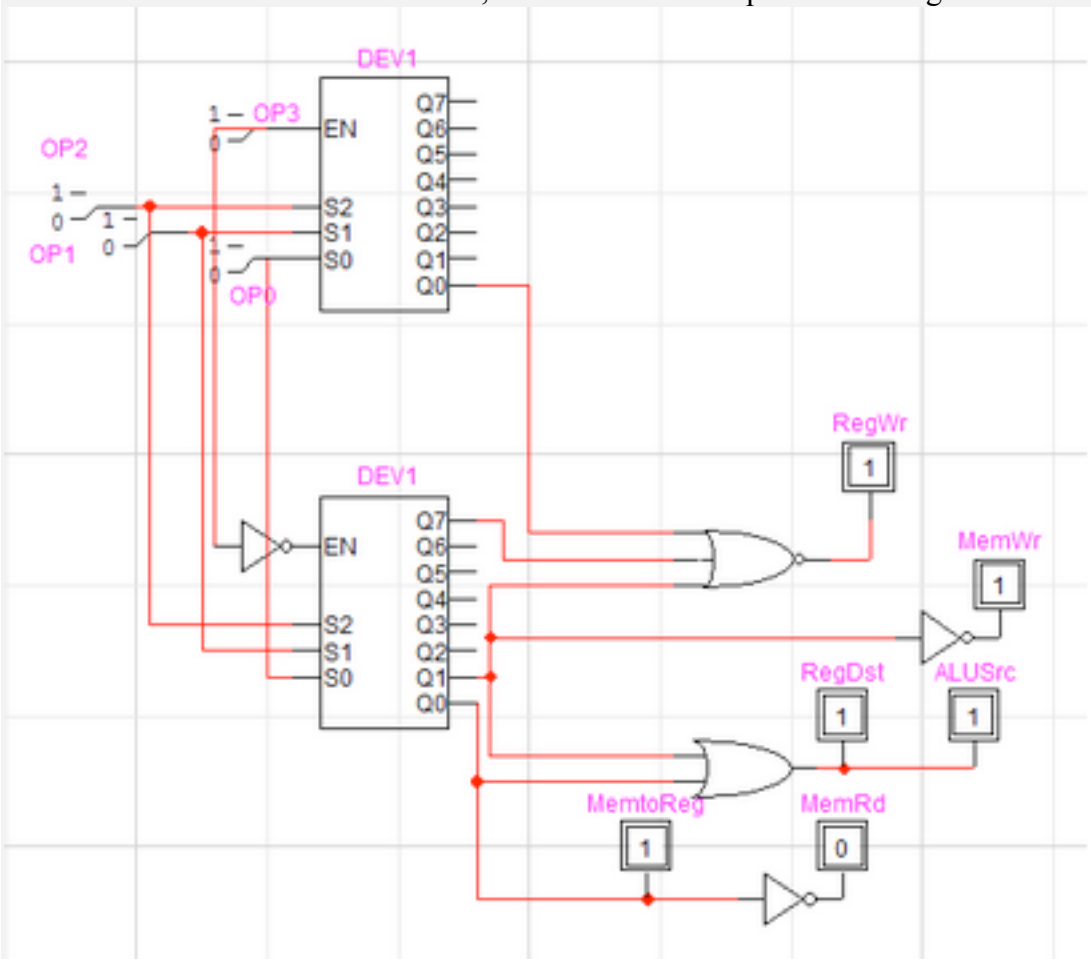


Control Unit

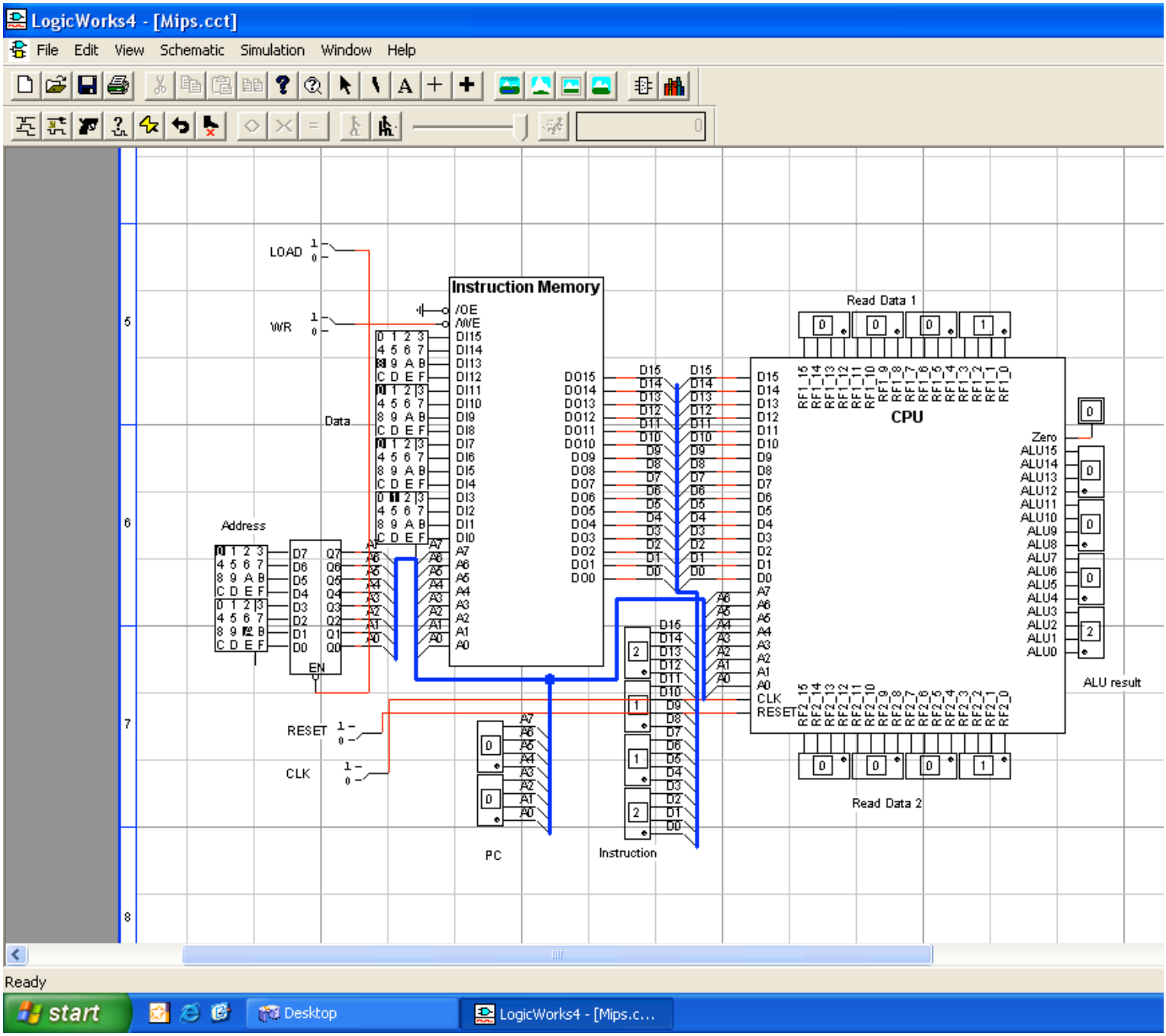
Exercise 2: The control lines can be produced as a simple function of the opcode, as defined in the following table:

Instruction	Opcode	RegDst	RegWr	ALUSrc	MemRd	MemWr	MemtoReg
LW	0000	1	1	1	0	1	1
SW	0001	1	0	1	1	0	0
ADD	0010	0	1	0	1	1	0
SUB	0011	0	1	0	1	1	0
AND	0100	0	1	0	1	1	0
OR	0101	0	1	0	1	1	0
SLT	0110	0	1	0	1	1	0
BEQ	0111	0	0	0	1	1	0
JMP	1000	0	0	0	1	1	0

Write a function for each control line, or use a decoder to produce the signals. Paste your design here:



Exercise 3: Open the *datapath1.cct* circuit sent to you prior to lab. Close the timing window and parts palette to view the full circuit:



The CPU has been abstracted to a black box, and is connected to the Instruction Memory, where programs must be loaded prior to execution by the CPU. You may recall from an earlier lab that the **LOAD, WR, data**, and **address** switches on the left are used to load a program into Instruction Memory.

The **PC** and **Instruction** LED displays on the bottom left show the address in Instruction Memory currently being executed, and the instruction itself.

The **Read Data 1** and **Read Data 2** register file ports and the **ALU result** from the CPU are also displayed. These can be used to verify correct execution of instructions.

The **Reset** and **CLK** switches on the bottom left are used to control execution of a program in the CPU.

The circuit you uploaded contains a short program you have seen before in binary form. Fill in the table below for each instruction in the program, explaining what the instruction does :

Address	Instruction	operation	Rs	Rt	Rd/offset	Purpose
0:	5002	OR	R0	R0	R2	#R2 gets 0
2:	5003	OR	R0	R0	R3	#R3 gets 0
4:	1220	SW	R2	R2	R0	#address n: gets n (start of loop!)
6:	0230	LW	R2	R3	0	#R3 gets n
8:	2122	ADD	R1	R2	R2	#R2 gets R2 + 1
A:	8002	JUMP	0002			#jump to 2*2 (address 4)= beginning of loop

Does this program ever stop? No

Execute the program by performing the following steps:

1. Set the **Reset** switch to 1 and back to 0.

   Notice that the PC goes to 00. What does this mean?
   Start at the very beginning of instruction memory.

   What value is displayed on the Instruction LED? What is this value?
   5002 is the first instruction (see above).

   What values are displayed on Read Port 1, Read Port 2, and the ALU Result? What do these mean?
   Read Port 1 = 0000, Read Port 2 = 0000, ALU Result = 0000
   Rt and Rs from the instruction are both R0 which has the value 0000. ALU Result is 0000 because we have not yet executed an instruction.

2. Set the **CLK** switch to 1 and back to 0. You just executed the first instruction, and stored the ALU Result to the destination register (in other words, R0 OR R0 = 0 -> R2).

   What is the value of **PC** now? Why?
   PC is 2 to move to the next instruction.

   What value is displayed on the Instruction LED? What is this value?
   5003 is the second instruction, at address 2

3. Set the **CLK** switch to 1 and back to 0 to execute this next instruction. You have now cleared both R2 and R3.

   What is the value of **PC** now?
   PC is now 4

You are now at the start of the loop in the program For the next 16 instructions, toggle the **CLK** switch and record the values you observe in the table below (for each instruction, **only** record the values that change for that instruction):

PC	Inst.	R2	R3	Addr. 0	Addr. 1	Addr. 2	Addr. 3
		0	0	?	?	?	?
4	SW R2 R2 0	0	0	0	?	?	?
6	LW R2 R3 0	0	0	0	?	?	?
8	ADD R1 R2 R2	1	0	0	?	?	?
A	JUMP 002	1	0	0	?	?	?
4	SW R2 R2 0	1	0	0	1	?	?
6	LW R2 R3 0	1	1	0	1	?	?
8	ADD R1 R2 R2	2	1	0	1	?	?
A	JUMP 002	2	1	0	1	?	?
4	SW R2 R2 0	2	1	0	1	2	?
6	LW R2 R3 0	2	2	0	1	2	?
8	ADD R1 R2 R2	3	2	0	1	2	?
A	JUMP 002	3	2	0	1	2	?
4	SW R2 R2 0	3	2	0	1	2	3
6	LW R2 R3 0	3	3	0	1	2	3
8	ADD R1 R2 R2	4	3	0	1	2	3
A	JUMP 002	4	3	0	1	2	3

Exercise 4: Assume two values are stored in data memory at address 0 and 2. The following program will subtract the value at address 2 from the value at address 0, and store the result in address 4 (the first five instructions place values into address 0 and address 2 in the data memory).

Fill in the hexadecimal value for each instruction in the table below:

Address	Instruction	op	Rs	Rt	Rd/offset	Purpose
0:		ADD	R1	R1	R2	; R2 = 2
2:		ADD	R2	R2	R3	; R3 = 4
4:		ADD	R3	R3	R3	; R3 = 8
6 (LOOP):		SW	R0	R3	0	; data address 0: 8
8:		SW	R0	R2	2	; data address 2: 2
A:		LW	R0	R5	0	; R5 <- 8 from address 0
C:		LW	R0	R4	2	; R4 <- 2 from address 2
E:		SUB	R5	R4	R5	;R5<- 8 – 2 = 6
10:		SW	R0	R5	4	;data address 4: 6
12:		LW	R0	R15	4	;R15 <- contents of address 4
14:		OR	R15	R15	R15	; displays contents of R15 at ALU result
16:		BEQ	R2	R15	LOOP	; repeat the instructions starting at label loop
18:		J	END			; jump to end of program

Fill in the table below with the **predicted** values for the registers and data memory after each step (for each instruction, only record the values that change for that instruction):

PC	Inst.	R2	R3	R4	R5	R15	Addr. 0	Addr. 2	Addr. 4
		0	0	0	0?	0	?	?	?
0	2112	2							
2	2223		4						
4	2333		8						
6	1030						8		
8	1022							2	
A	0050				8				
C	0042			2					
E	3545				6				
10	1054								6
12	00F4					6			

To load the Instruction Memory with this program, follow these steps:

1. Disconnect the address bus of the Instruction Memory from the CPU (ask the instructor to demonstrate how to do this).
2. Set **LOAD** = 0
3. Set **address** and **data** switches for instruction
4. Set **WR** = 0, then back to 1
5. Repeat steps 3 and 4 until all instructions are loaded to memory

To execute the program, follow these steps:

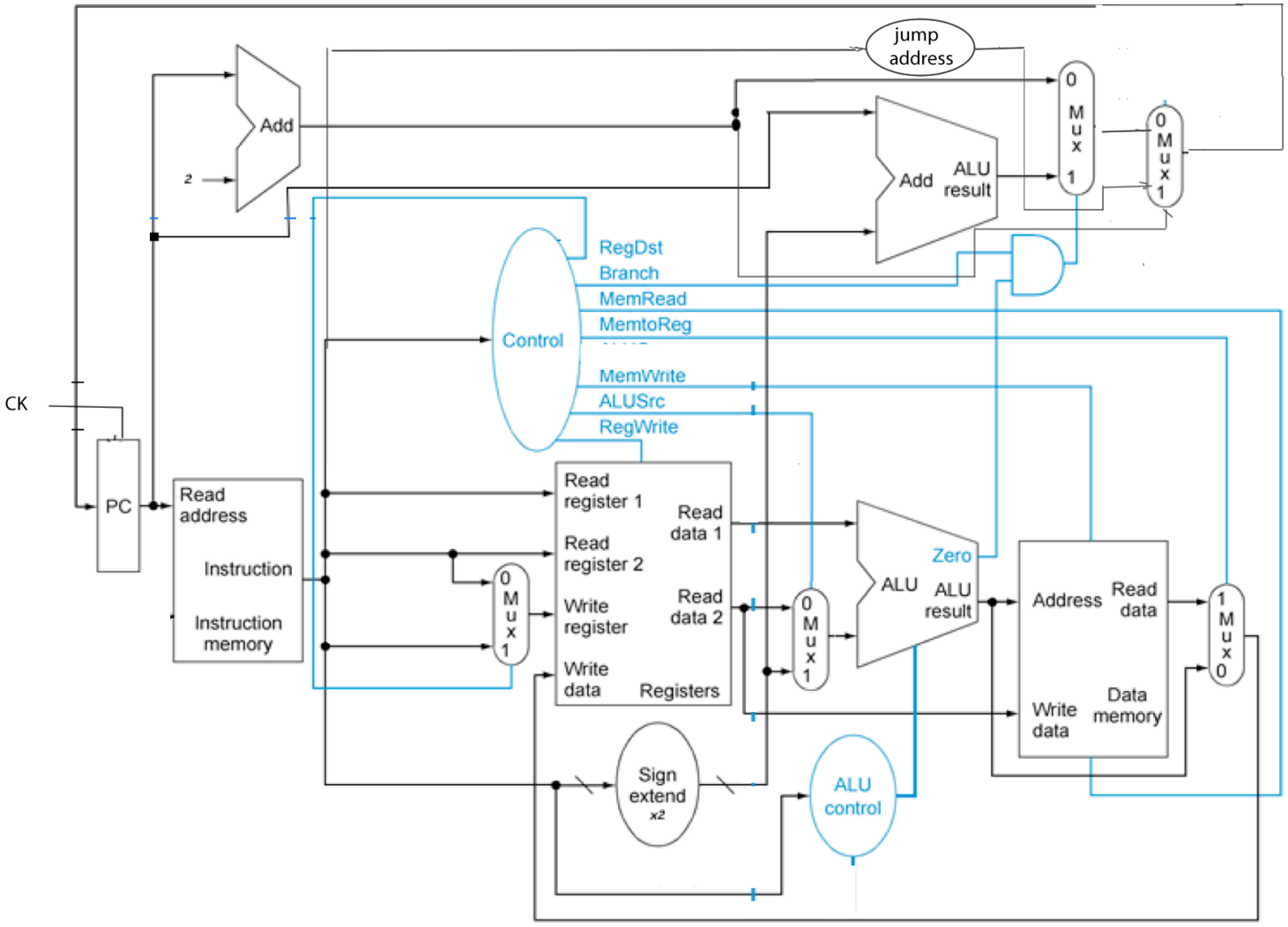
1. Set **LOAD** = 1
2. Reconnect address bus to CPU
3. Set **Reset** = 1, then back to 0
4. Set **CLK** = 1, then back to 0, for each instruction. Stop when the PC = 14.

Do you see the value of 6 displayed at the ALU Result? Demonstrate to the instructor.

Close the circuit before the next exercise.

Pipelining

Exercise 5: The following diagram shows the single-cycle datapath for the mini-MIPS lab machine:



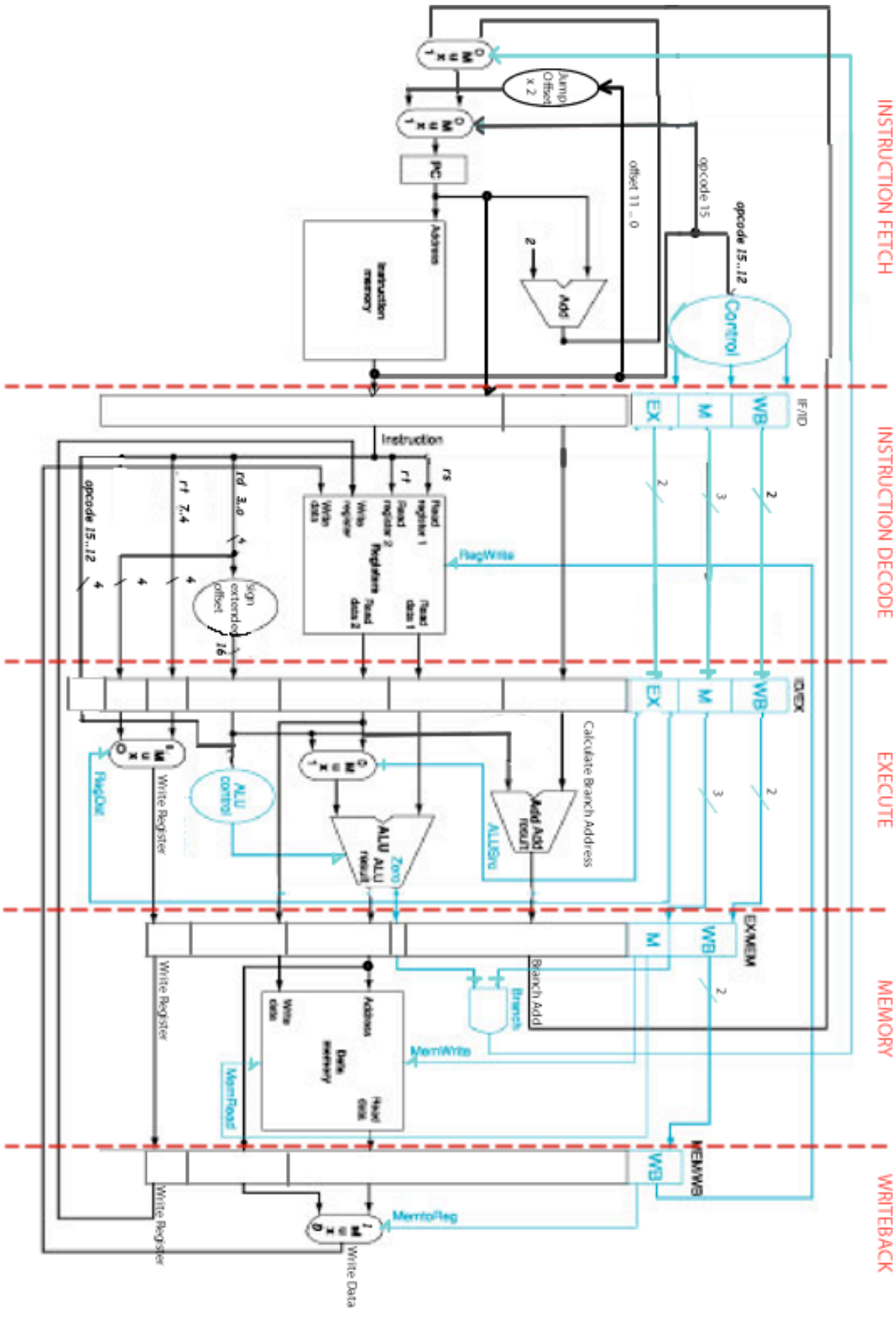
As you have learned in lecture, this machine executes one instruction per clock cycle. The clock cycle must be long enough for the instruction with the longest path (*critical path*) , which is the **lw**. Since many of the instructions executed in a typical program have a shorter path than the **lw**, the single-cycle datapath is inefficient (it wastes time during the shorter instructions).

To increase efficiency, it is possible to divide the datapath into stages, and execute multiple instructions in parallel. Assuming a large number of instructions are executed, this results in a speed-up proportional to the number of stages, which is a big improvement in efficiency. This is called a *pipelined datapath*.

In the pipelined Mini-MIPS machine, the following stages are used:

1. Instruction fetch (get the instruction from memory)
2. Instruction decode/register file access (use the instruction to get the control signals and operands)
3. Execution/ALU (use the operands and control signals to perform an operation in the ALU)
4. Data memory access (access the memory if necessary)
5. Register file write-back (write back the result to the register file)

Each stage is separated from the next by a set of registers, which store the results and control signals from the stage and send them to the next stage on each clock pulse. The following page shows the mini-MIPS pipelined datapath (hazards are not taken into consideration in this design). This diagram is similar to the MIPS pipeline from the textbook, with a few modifications to correspond to the differences between the text and lab machine:


INSTRUCTION FETCH
INSTRUCTION DECODE
EXECUTE
MEMORY
WRITEBACK
M u x
PC
Address
Instruction memory
Add
Control
opcode 15
opcode 15..12
offset 11..0
IF/ID
EX
M
WB
Instruction
rs
rt
rd
rt 7..4
rd 3..0
opcode 15..12
Register file
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
RegWrite
Sign extend
16
ID/EX
ALU control
ALU
Zero
Add
Add result
ALUSrc
Calculate Branch Address
RegDst
Write Register
EX/MEM
Branch
Branch Add
MemWrite
Data memory
Address
Write data
Read data
MemRead
MEM/WB
MemtoReg
Write Data

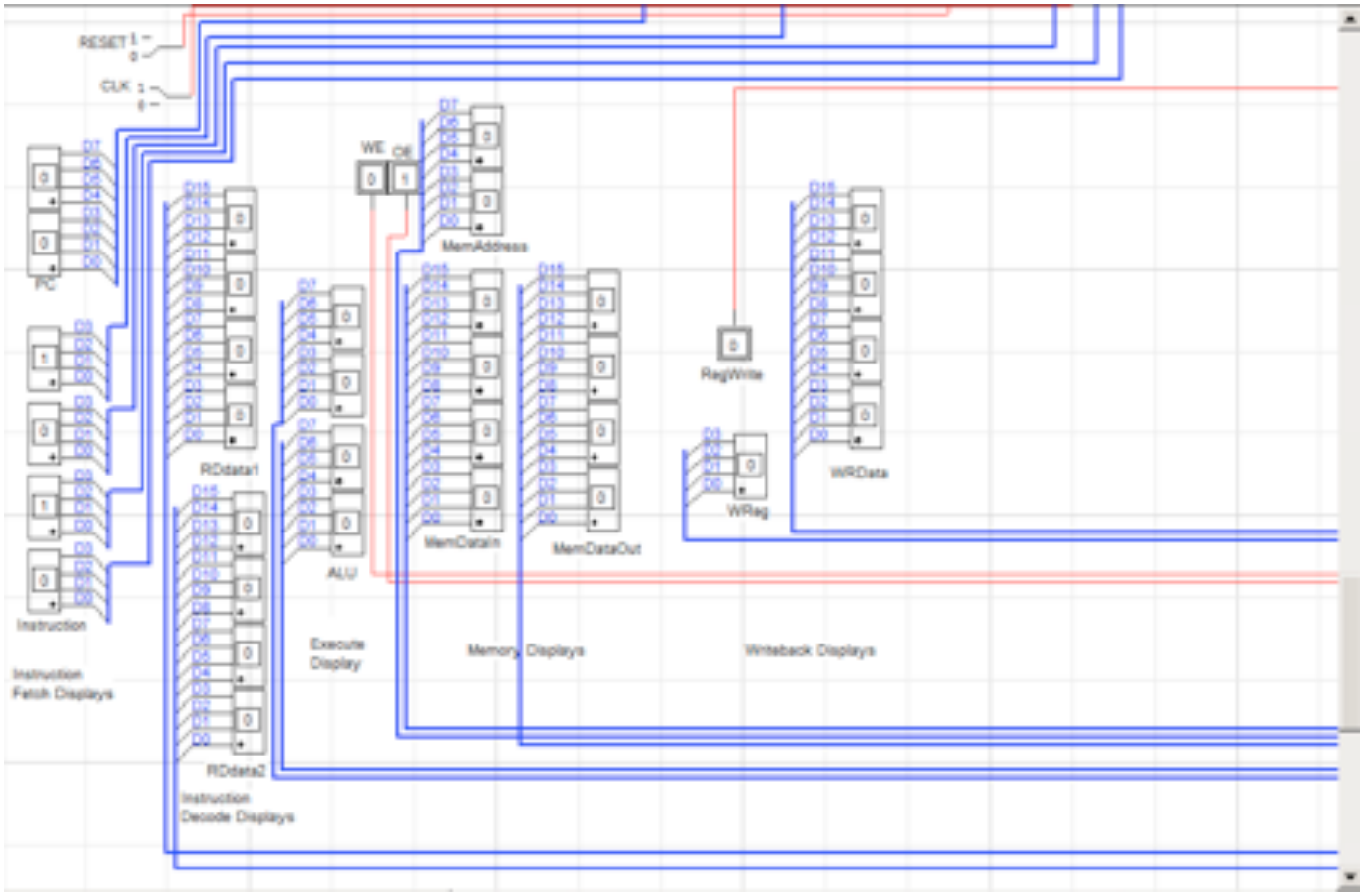
Open the *pipeline.cct* circuit from the Google group. Examine the circuit, to verify that you understand the modifications which have been made to implement pipelining.

The Instruction Memory is preloaded with a testing program which does not contain any hazards.

1. For each instruction, predict the value of the ALU and any registers or data memory locations that are modified by the instruction:

PC/Address	Instruction		ALU	Register	Data Memory Address: Value
00:	SW	R0 R1 0	0	N/A	0: 1
02:	ADD	R1 R1 R2	2	2:2	N/A
04:	SLT	R0 R1 R3	1	3:1	N/A
06:	SUB	R1 R0 R4	1	4:1	N/A
08:	ADD	R1 R1 R5	2	5:2	N/A
0A:	LW	R0 R6 0	0	6:1	N/A
0C:	ADD	R1 R1 R7	2	7:2	N/A
0E:	SLT	R0 R1 R8	1	8:1	N/A
10:	SUB	R1 R0 R9	1	9:1	N/A
12:	ADD	R1 R1 RA	2	A:2	N/A
14:	ADD	R6 R6 RB	2	B:2	N/A
16:	BEQ	R0 R0 7	0	N/A	N/A
18:	OR	R3 R3 R3	1	3:1	N/A
1A:	OR	R4 R4 R4	1	4:1	N/A
1C:	OR	R5 R5 R5	2	5:2	N/A
1E:	OR	R6 R6 R6	0	6:1	N/A
20:	.				
22:	. no instructions stored here				
24:	.				
26:	JMP	00 00 00			

You will run the program (don't do it yet!) by pulsing the **reset** to 1 and back to 0, and then pulsing the **clk** to 1 and back to 0 to execute a stage of the pipeline. You will be able to understand the progress of your program by monitoring the displays at the bottom left of the circuit. Each set of displays represents a stage of the pipeline, and is labeled appropriately.



2. **Predict** (don't operate the circuit yet) values of the pipeline stages for the first instruction SW R0 R1 0:

Instruction Fetch (reset)	**Instruction Decode (1st clk)**	**Execute (2nd clk)**	**Memory (3rd clk)**	**Writeback (4th clk)**
PC = 00 **Inst** = 1010				
	RDd1 = **00** **RDd2** =**01**			
		ALU=**0000**		
			MemAdd=**00** **MemDin**=**01** **MemDout**=**xx**	
				RegWrite=**0** **WReg**=**xx** **WRData**=**xx**

Does it matter what **MemDataOut** is for this instruction?

Does anything happen in the **Writeback** stage for this instruction (in other words, does a value get written back to a register in the register file?)

3. Also **predict** the values of the pipeline stages for the second instruction (ADD R1 R1 R2). Notice that the table of values here is offset by one clock cycle from the first instruction (which is how the pipeline works).

PC = 00 **Inst** = 1010				
	RDd1 = **0000** **RDd2** = **0001**			
	PC =02 **Inst** = 2112	**ALU**= **0000**		
		RDd1 = **01** **RDd2** = **01**	**MemAdd**= **00** **MemDin**= **01** **MemDout**= **xx**	
			ALU =**02**	**RegWrite**= **0** **WReg**= **xx** **WRData**= **xx**
				MemAdd=**xx** **MemDin**= **xx** **MemDout**=**xx**
				RegWrite=**1** **WReg**=**02** **WRData**=**02**

Does it matter what the values are in the **Memory** stage for this instruction? How about the **Writeback** stage?
The memory stage does not matter, because it is not a memory access instruction. The writeback stage does matter, because a value is being written back to register 2 as a result of the add.

Exercise 6: Execute the program in memory, and record the values on the displays after each clock cycle (use a small font, it is important that the cells stay aligned!):

PC = 0 Inst = 1010	RD1=0 RD2 = 1	ALU=0	MemDin=1 MemAdd=0 MemDataO ut=1	WReg=1 WRData=1 RegWrite=0

PC = 2 Inst = 2112	RD1=1 RD2=1	ALU=2	MemDin=1 MemDout=0 Memadd=2	WReg=2 WRData=2 RegWrite=1

PC = 4 Inst = 6013	RD1=0 RD2=1	ALU=1	MemDin=1 MemDout=0 Memadd=1	WReg=3 WRData=1 RegWrite=1

PC=6 Inst=3104	Rd1=1 Rd2=0	ALU=1	MemDin=0 MemDout=0 Memadd=1	WReg=4 WRData=1 RegWrite=1

PC=8 Inst=2115	RD1=1 RD2=1	ALU=2	MemDin=1 MemDout=0 Memadd=2	WReg=5 WRData=2 RegWrite=1

PC=AIn st=0060	RD1=0 RD2=0	ALU=0	MemDin=0 MemDout=1 Memadd=0	WReg=6 WRData=1 RegWrite=1

PC=C Inst=2117	RD1=1 RD2=1	ALU=2	MemDin=1 MemDout=0 Memadd=2	WReg=7 WRData=2 RegWrite=1

PC=E Inst=6018	RD1=0 RD2=1	ALU=1	MemDin=1 MemDout=0 Memadd=1	WReg=8 WRData=1 RegWrite=1

PC=10 nst=3109	RD1=1 RD2=0	ALU=1	MemDin=0 MemDout=0 Memadd=1	WReg=9 WRData=1 RegWrite=1

PC=12 Inst=210A	RD1=1 RD2=0	ALU=1	MemDin=0 MemDout=0 Memadd=1	W W Re

PC=14 Inst=266B	RD1=1 RD2=1	ALU=2	Me Me Me

PC=16 Inst=7007	RD1=0 RD2=0

*****PC=18 Inst=5333

At this point, you should be at the **JMP 0 0 0** instruction, which should take you back to the beginning of the program.

2. Did you notice that for the **BEQ R0 R0 7** instruction, even though the branch condition was met (**R0 = R0**), two more instructions entered the pipeline before the branch was taken? Examine your output to verify this, and mark the two cycles where these instructions entered the pipeline.

For this particular program, this does not cause a problem, since those two instructions make no changes to any of the registers. But, normally, this is a problem called a *branch hazard*, which you learned about in lecture. There are several modifications which deal with branch hazards, including calculating the branch address and evaluating the condition earlier in the pipeline.

Data Hazards

Exercise 7: The test program you are using does not have any *data hazards*, which are when one instruction needs the result of an instruction either one or two cycle earlier. In this case, the instruction is still in the pipeline (is not yet complete, so that the value has not yet been written back to the register which is needed).

1. For example, for the first five instructions:

```
00:     SW    R0 R1 0
02:     ADD   R1 R1 R2
04:     SLT   R0 R1 R3
06:     SUB   R1 R0 R4
08:     ADD   R1 R1 R5
```

changing the third and fourth instructions to:

```
04:     SLT   R0 R2 R3
06:     SUB   R2,R0,R4
```

would both cause a data hazard and incorrect final results for **R3** and **R4**. Why?
Because R2 is used as an operand in each of the two instructions (but will not yet have its new value from the instruction at addr. 02)

However, changing the fifth instruction to:

```
08:     ADD   R1 R2 R5
```

(which also uses **R2**) would work correctly. Why is this okay?
Because it is three instructions (pipeline stages) later than the instruction at addr. 2, so R2 will have its final value.

2. Modify the three instructions as indicated in the previous step, and reset the circuit. Clock until you see that the result in R3 is incorrect (it will take 6 clock pulses to view the incorrect result be written to R3, although you can see by reading the displays in the earlier stages that R2 has not yet been updated to 2 when it is used in the SLT instruction).

3. Clock again to view the incorrect result for **R4**.

4. Clock once more to view the correct result for **R5**.

To solve this problem, you can *forward* the value as soon as it is available after the Execute stage or Memory stage, so that subsequent instructions have access to the updated value earlier than the final Writeback stage.

5. A similar data hazard is caused by trying to read a register following a load instruction that writes that same register. Given the following sequence in your program:

```
0A:     LW    R0 R6 0
0C:     ADD   R1 R1 R7
0E:     SLT   R0 R1 R8
10:     SUB   R1 R0 R9
```

Modify the following instructions so that they use **R6** after it is loaded in the **LW** instruction:

```
0C:     ADD   R1 R6 R7
0E:     SLT   R6 R1 R8
10:     SUB   R6 R0 R9
```

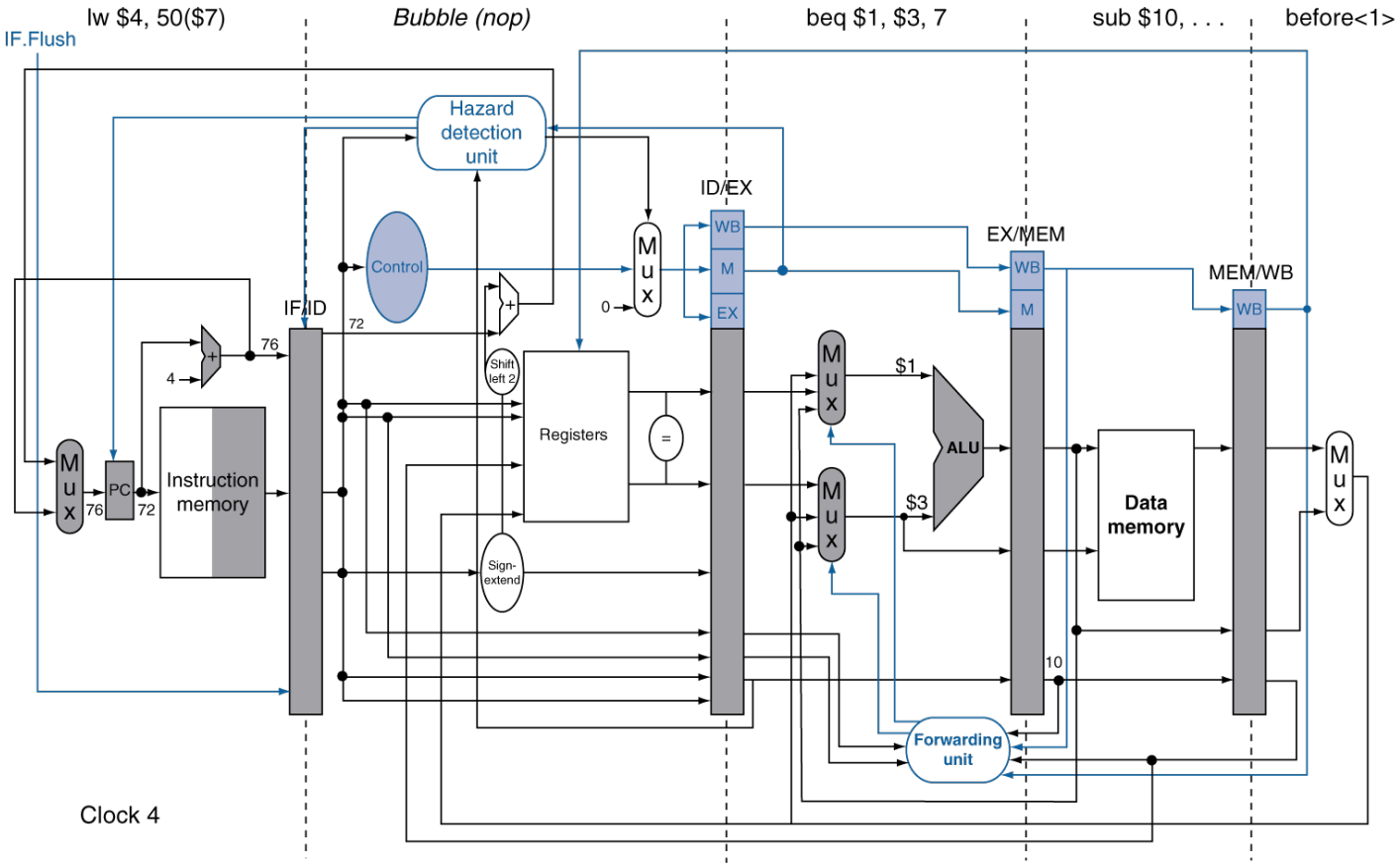
6. Clock until you see that the result in **R7** is incorrect.

7. Clock again to view the incorrect result for **R8**.

8. Clock once more to view the correct result for **R9**.

Since reading from memory comes so late in the pipeline, the value can not be forwarded back to an earlier stage in time to avoid the hazard, as it could for R-type instructions. So, in this case, the pipeline has to be *stalled* until the value is available, which means that several cycles are executed without fetching any new instructions into the pipeline (these cycles are called *nops*, or *bubbles*). Nops are accomplished by setting the control bits to all 0s for several stages (which is like executing an instruction that doesn't access memory or write back to any registers).

9. The following diagram in the text and in lecture explains the modifications needed in the pipeline to handle hazards:



Examine the diagram, refer to your textbook and lecture notes, and explain how the forwarding unit and hazard detection units work: