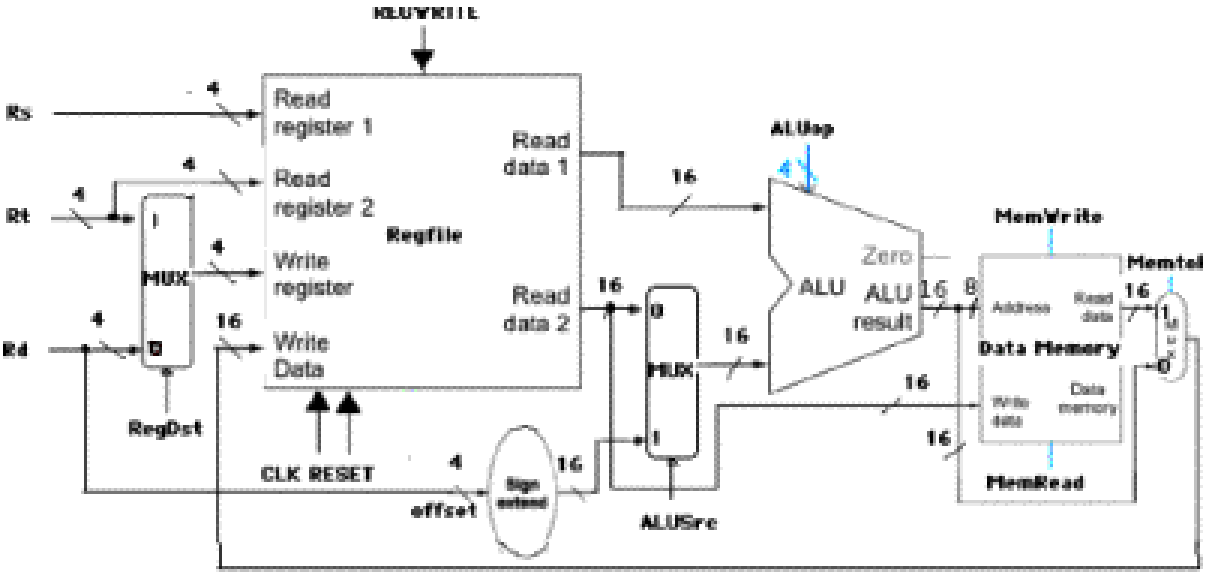
Laboratory 10
Control Path, Single-Cycle and Pipelined CPU

Data Path with Data Memory



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

Control Logic for the ALU

ALU can perform 5 possible operations:

ALUop	ALU function
0	a AND b
1	a OR b
2	a + b (add)
6	a - b
7	set on less than

Need an ALU Control Unit to select the proper operation for each instruction:

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

In binary:

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

Control Unit

Must provide control signals for all other devices in datapath (MUXs, Regfile, Data Memory)

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

Can produce with logic gates or a 4x16 decoder (two 3x8 decoders)

Programming the Single-Cycle CPU

Mini-MIPS program which loops repeatedly to access memory:

Address	Instruction	Meaning	
0:	5002	OR R0 R0 R2	#R2 gets 0
2:	5003	OR R0 R0 R3	#R3 gets 0
4:	1220	SW R2 R2 0	#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 002	#jump to 2*2 (address 4) = beginning of loop

Trace execution:

	R2	R3	Memory Address	Value
0:				
2:				
4:				
6:				
8:				
A:				
4:				
6:				
8:				
A:				
4:				
6:				
8:				
A:				
4:				
6:				
8:				
A:				

Another Example

Assume two values are stored in data memory at address 0 and 2.

Subtract the value at address 2 from the value at address 0, and store the result in address 4:

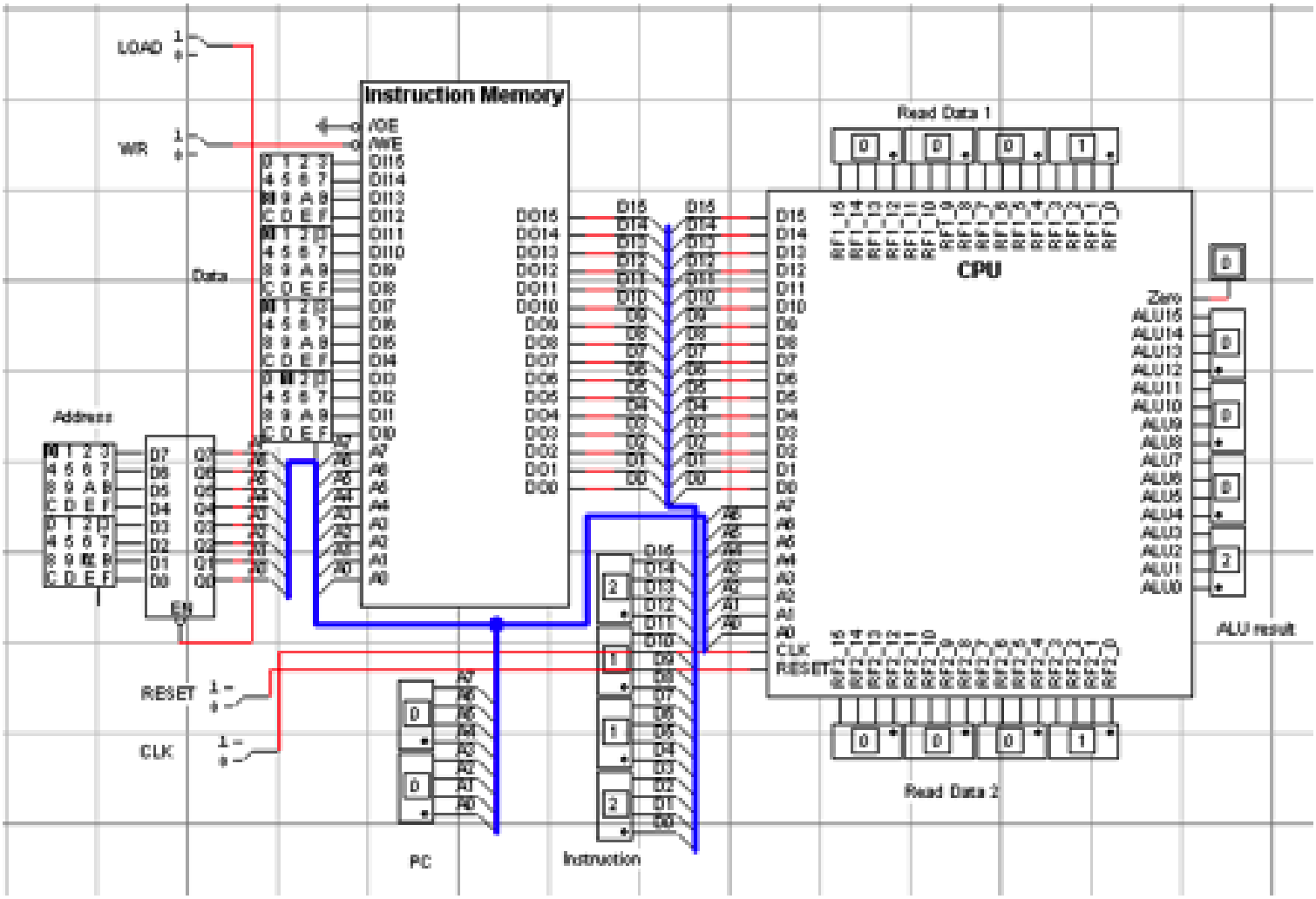
```
0:    LW R0 R2 0

2:    LW R0 R3 2

4:    SUB R2 R3 R2

6:    SW R0 R2 4
```

Mini-MIPS in LogicWorks



Procedure to Load/Execute a New Program

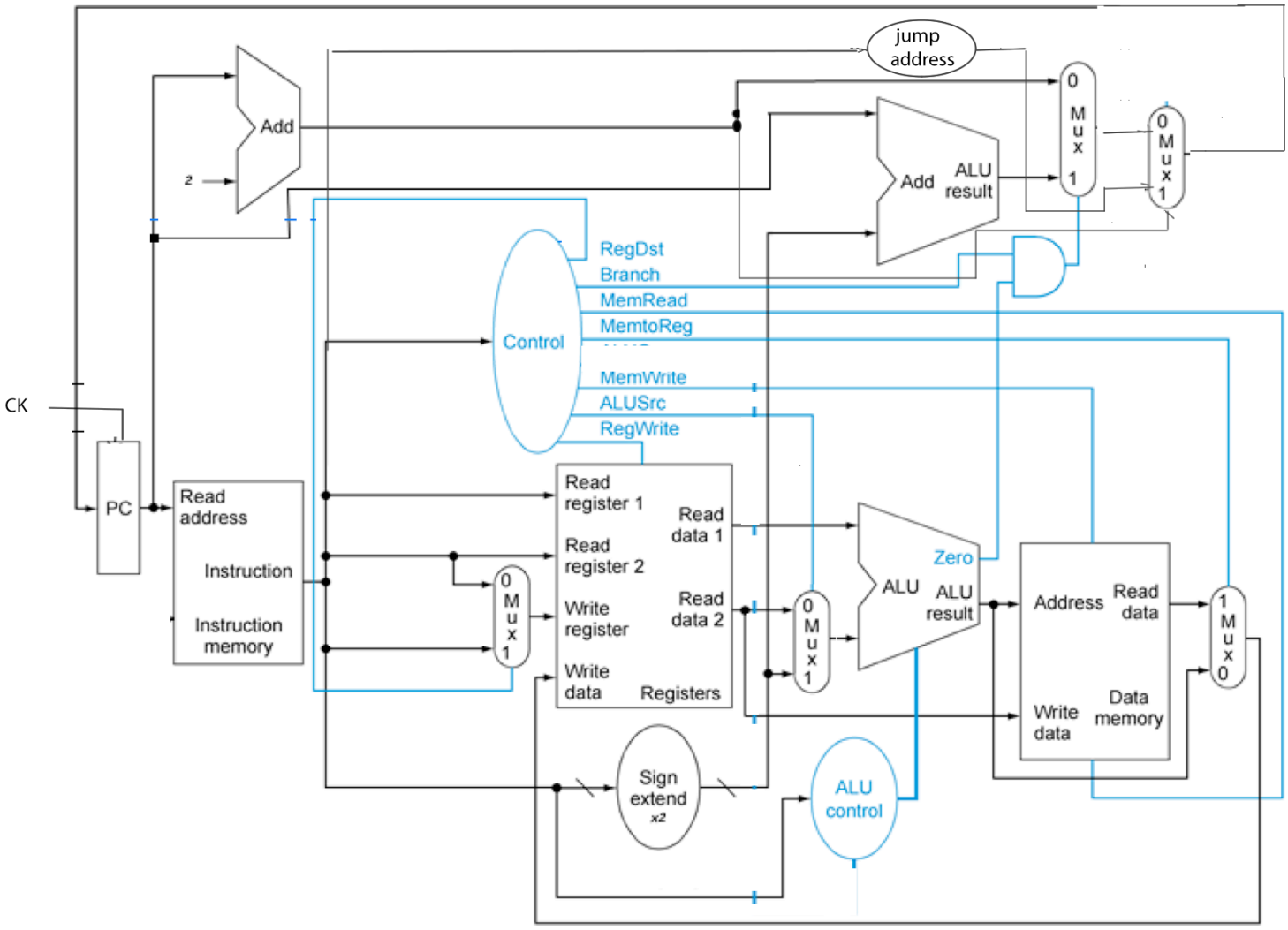
1. Disconnect the address bus of the Instruction Memory from the CPU
2. Set **LOAD** = 0

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

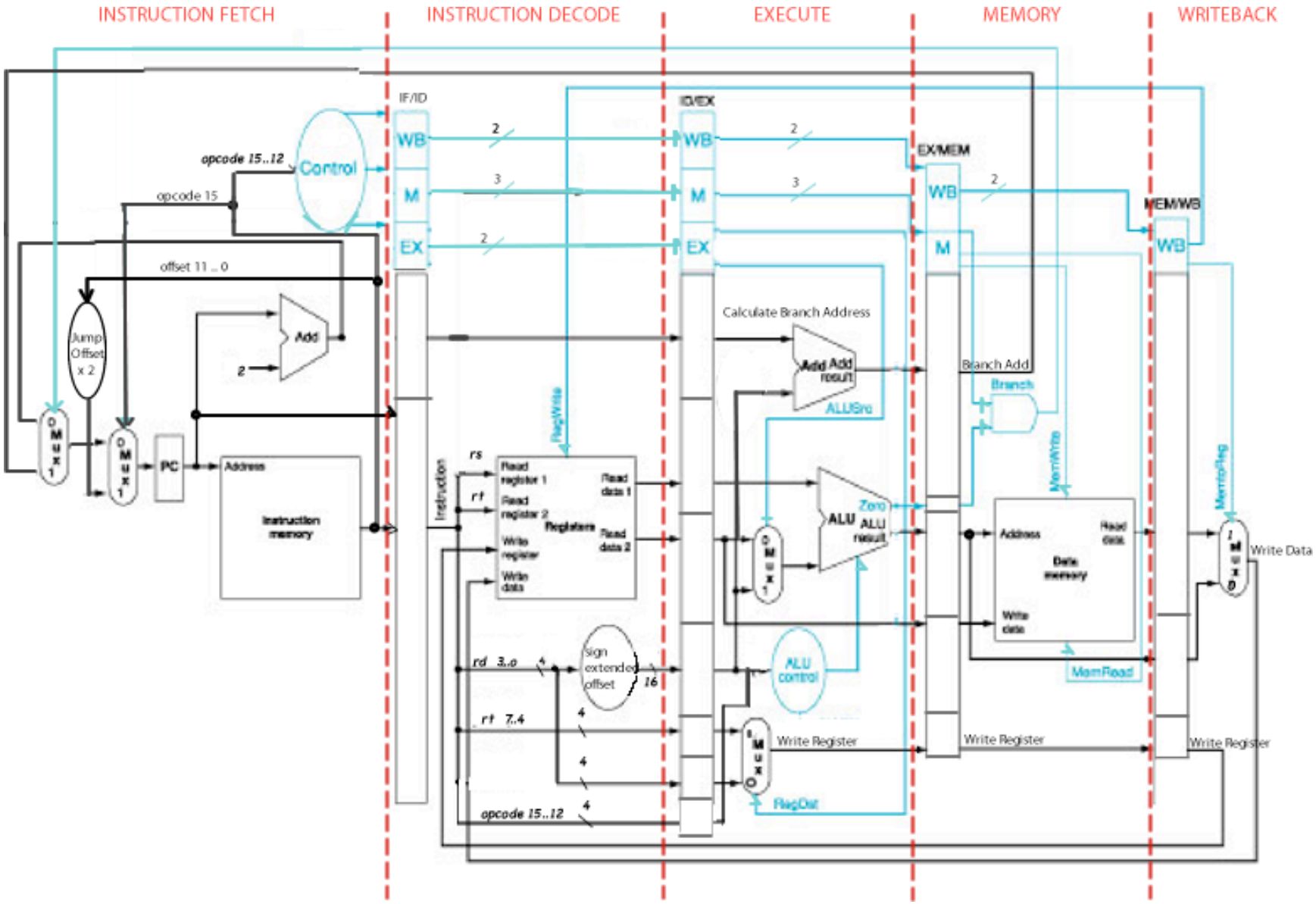
6. Set **LOAD** = 1
7. Reconnect address bus to CPU

8. Set **Reset** = 1, then back to 0
9. Set **CLK** = 1, then back to 0, for each instruction.

Single-Cycle CPU



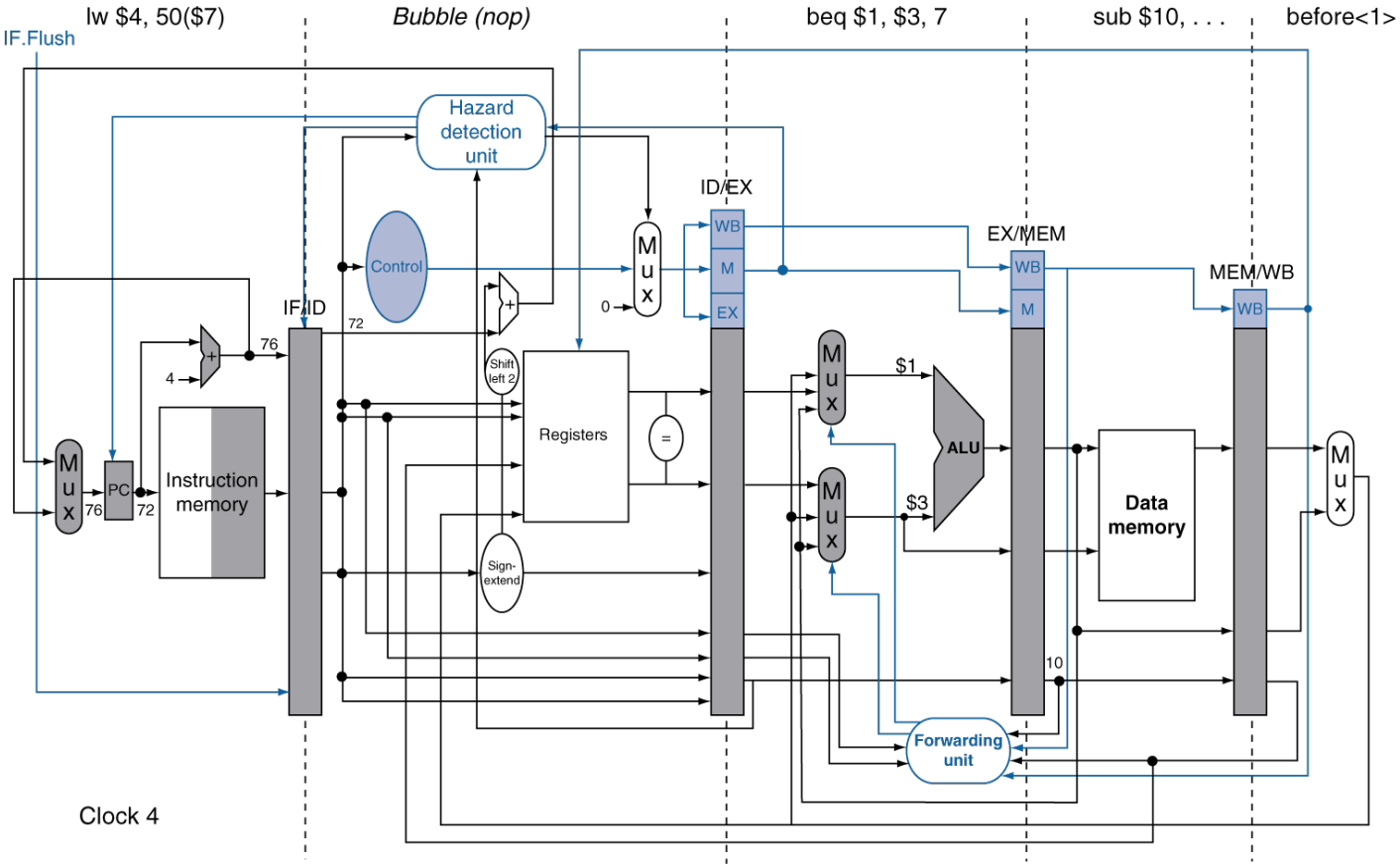
May be made into a pipelined circuit by inserting registers at the pipeline stages:


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

Data Hazards

A data hazard is when an instruction uses a register which is being written to by an earlier instruction which is still in the pipeline. Therefore, the new value of the register is not yet available in the Execute stage of the later instruction which requires it.

The following diagram shows the additions to the pipeline to deal with data hazards:



As an example, there is a data hazard in both the SUB and SLT instruction below because of R2.

ADD R1 R1 R2
SUB R2 R1 R3
SLT R2 R1 R4

There are two types of hazard conditions:

Rs or **Rt** from the current instruction are getting a new value from the **previous instruction**

1a. **EX/MEM.RegisterRd** = **ID/EX.RegisterRs**
1b. **EX/MEM.RegisterRd** = **ID/EX.RegisterRt**

Rs or **Rt** from the current instruction are getting a new value from **2 instructions earlier**

2a. **MEM/WB.RegisterRd** = **ID/EX.RegisterRs**
2b. **MEM/WB.RegisterRd** = **ID/EX.RegisterRt**

We can solve this problem for register-type instructions by *forwarding* the register value back to the Execute stage. An additional MUX in the Execute stage (controlled by a Forward signal) chooses one of the three possible sources for the A and B sides of the ALU:

ForwardA/B	**Source**	**Explanation**
00	ID/EX	Forwarding not required, use Reg from ID/EX (register file)
10	EX/MEM	Forwarding required from previous instruction, use Reg from EX/MEM (ALU)
01	MEM/WB	Forwarding required from data memory or from 2 instructions previous, use result from MEM/WB (value to be written back to register file)

if (**EX/MEM.RegWrite** and
(**EX/MEM.RegisterRd** > 1) and
(**EX/MEM.RegisterRd** = **ID/EX.RegisterRs**)) then

ForwardA = 10

else if (**MEM/WB.RegWrite** and
(**MEM/WB.RegisterRd** > 1) and
(**EX/MEM.RegisterRd** ≠ **ID/EX.RegisterRs**) and
(**MEM/WB.RegisterRd** = **ID/Ex.RegisterRs**)) then

ForwardA = 01

else

ForwardA = 00

The same conditions must be tested for **Rt** , producing **ForwardB**. The *forwarding unit* which produces **ForwardA** and **ForwardB** can be implemented with simple logic

Stalls

Forwarding alone is not effective if an instruction tries to use a register following a **lw** instruction that writes the same register, because the memory access comes so late in the pipeline that the value read from the memory is not yet available to be forwarded back to the Execute stage.

This hazard can be detected by testing the following condition in the ID stage:

if (**ID/EX.MemRead** and
((**ID/EX.RegisterRt** = **IF/ID.RegisterRs**) or (**ID/EX.RegisterRt** = **IF/ID.RegisterRt**)))

When this hazard occurs, a *stall*, or *nop* (no operation) is needed in the ID and IF stage to wait for the result of the **lw** to be written before the next instruction is executed.

This is accomplished by preventing the PC and the IF/ID registers from changing.

It is also necessary to deassert the control bits in the ID stage. This causes the next pipeline stages to "do nothing"; no registers are written and no memory accesses occur if the control bits are not asserted.

Some simple logic and an additional MUX in the ID stage are all that are necessary to implement the *hazard detection unit* and clear the control bits.

Are there any hazards in this code?

```
SW   R0 R1 0
ADD  R1 R1 R2
SLT  R0 R1 R3
SUB  R1 R0 R4
ADD  R1 R1 R5
LW   R0 R6 0
ADD  R1 R1 R7
SLT  R0 R1 R8
SUB  R1 R0 R9
ADD  R1 R1 RA
ADD  R6 R6 RB
```