Introduction to Computer Architecture

Morteza Biglari-Abhari

Department of Electrical, Computer and Software Engineering
University of Auckland
Email: m.abhari@auckland.ac.nz

Lectures 16 - 17

Review: Our current multi-cycle implementation

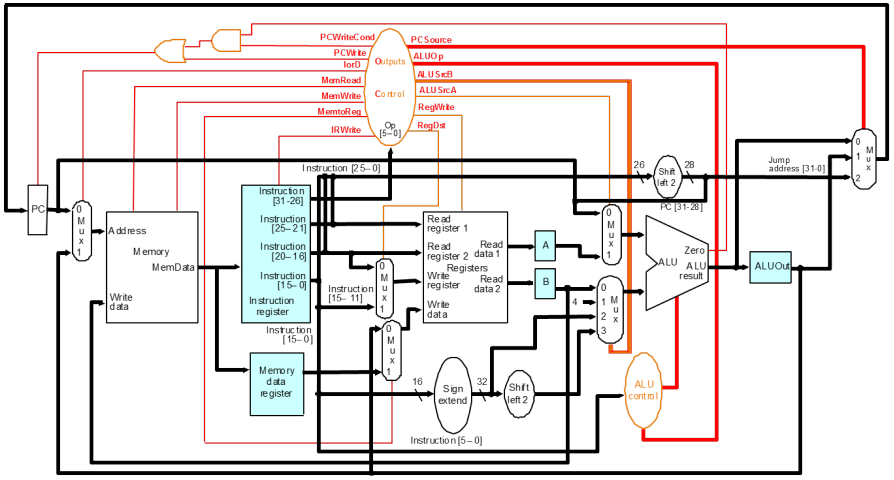


Figure 5.33 of ref3

Required clock cycles for each group of instructions

Step name	Action for R-type instructions	Action for memory reference instructions	Action for branches instructions	Action for jumps instructions
Instruction Fetch	IR ← Memory[PC] PC ← PC + 4			
Decode /register fetch (access)	A ← Reg[IR[25-21]], B ← Reg[IR[20-16]] ALUOut ← PC + (sign-extend(IR[15-0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut ← A *op* B	ALUOut = A + sign-extend(IR[15-0])	If (A == B) then PC ←ALUOut	PC ← PC[31-28] || (IR[25-0])<<2
Memory access or R-type completion	Reg[IR[15-11]] ← ALUOut	**Load:** MDR ← Memory[ALUOut] or **Store:** Memory[ALUOut] ← B		
Memory read completion		**Load:** Reg[IR[20-16]] ← MDR		

Adding Exception Handling to Multi-Cycle Datapath

In our simplified implementation, we only consider:

- two exceptions: undefined instruction and arithmetic overflow.
- special registers **EPC** and **Cause**
  (we assume here these two registers have been added to the CPU while they are coprocessor 0 registers).

For simplicity, we assume that **Cause** register will be written **0** or **1** for undefined instruction and arithmetic overflow exceptions respectively.

When an exception occurs the content of **PC** will be changed to **0x80000180** to point to the first instruction of the ***exception handler***. (This is based on the MIPS-32 architecture and might be different for different architectures).

We need three new control signals: ***EPCWrite***, ***CauseWrite***, ***IntCause*** ('1' for arithmetic overflow, and '0' for undefined instruction).
(Overflow signal from ALU is also used to signal this exception).

Our current multi-cycle implementation

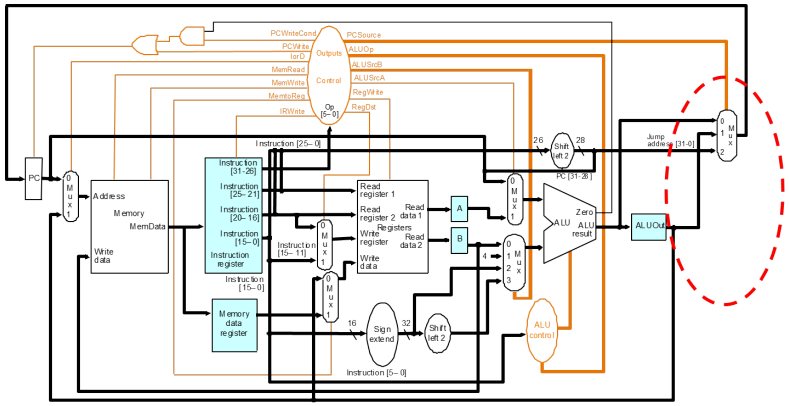


Figure 5.33 of ref3

We need to add two new internal registers ***EPC*** and ***Cause*** to our datapath in addition to some new control signals.

Signal name	Value	Effect
ALUOp	00	The ALU performs an **add** operation.
	01	The ALU performs a **subtract** operation.
	10	The **funct field** of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from **B register**.
	01	The second input to the ALU is the **constant 4**.
	10	The second input to the ALU is **the sign-extended, lower 16 bits of the IR**.
	11	The second input to the ALU is **the sign-extended, lower 16 bits of the IR shifted lift 2 bits**.
PCSource	00	Output of the ALU **(PC + 4)** is sent to the **PC** for writing.
	01	The content of **ALUOut** (the branch target address) is sent to the **PC** for writing.
	10	The jump target address **(IR[25-0]) shifted left 2 bits and concatenated with PC + 4[31-28])** is sent to the **PC** for writing.

ALU control signals

ALUop1, ALUop0	
00	Add (for loads and stores)
01	Subtract (for beq)
10	Depends on ***funct field*** for different R-type instructions
11	Depends on ***opcode field*** for different ALU i-type instructions

The multi-cycle datapath with support for exception handling

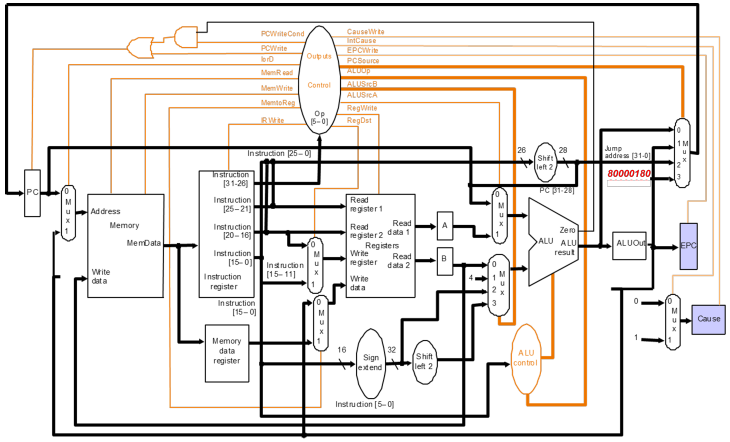
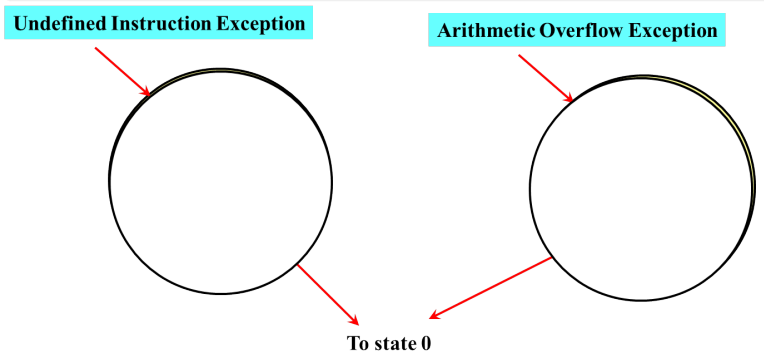


Figure 5.48 of ref3

Each state should provide control for three actions: *setting the cause register, putting the address of the instruction which caused exception into the EPC, and setting the PC to point to the exception vector address* (0x80000080 in our case).



FSM for control unit of the multi-cycle datapath with exception detection

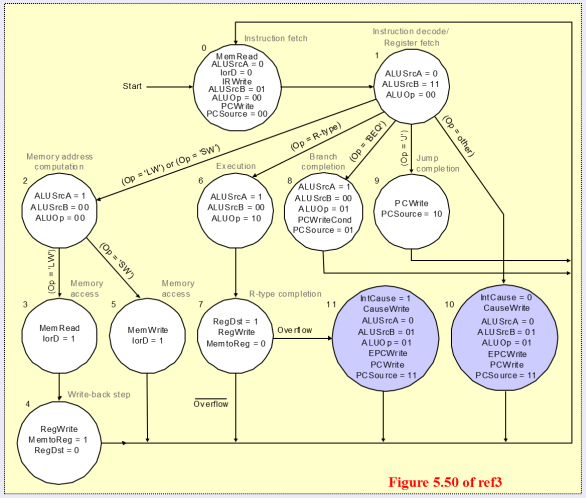
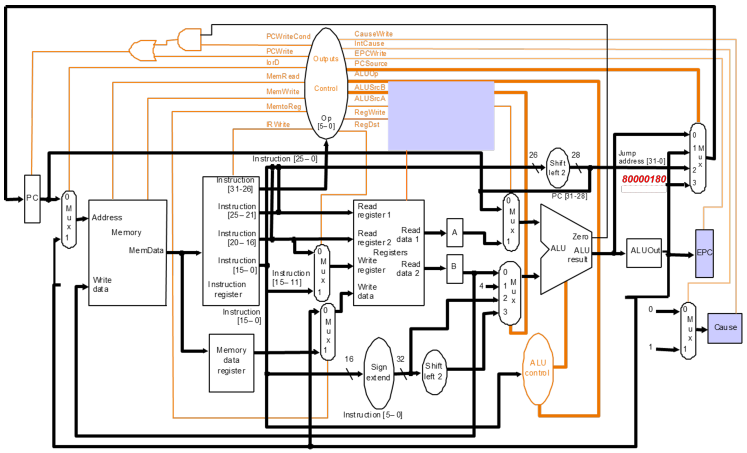


Figure 5.50 of ref3

There is bug(s) in our implementation for exception handling. Can you find it?



Fixing one bug in our implementation. Is there any other bug(s)?

More on performance improvement

CPU execution time = Instruction count * CPI * clock cycle period

- ➢ instruction set architecture
- ➢ compiler optimisations

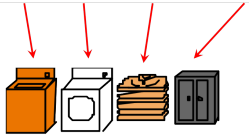
- o processor organisation/implementation

- ❖ integrated circuit technology
- ❖ processor organisation/implementation

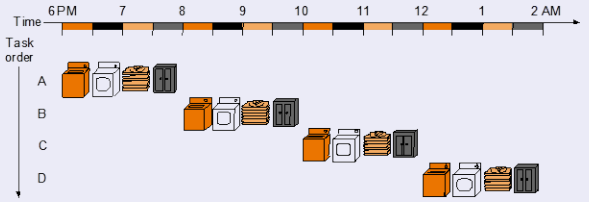
An Overview of Pipelining (an example)

Assume 4 people (A, B, C, and D) plan to clean their clothes. The cleaning process includes:

***washing*, *drying*, *folding* and *storing*.**

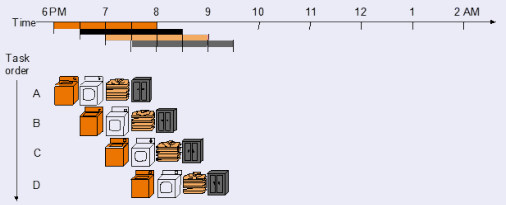


Method 1



It takes 2 hours for each person for cleaning and total time of for 4 people is 8 hours.

Method 2



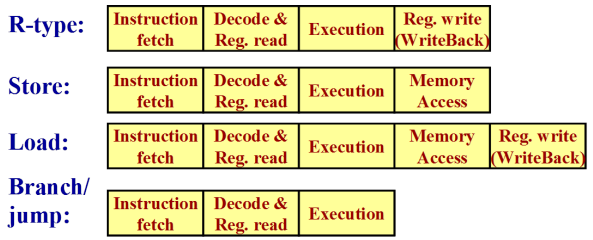
It still takes 2 hours for **each person** for cleaning but the total time for 4 people is now 3.5 hours. ***So pipelining improves the throughput***.

In summary

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. (It does not change the execution time of a single instruction. *However, it improves the throughput*). ***Pipelining is one of the key implementation techniques to make a faster CPU.***

- ***Execution latency:*** The time between the start and completion of an instruction.
- ***Throughput:*** The total amount of work done in a given time.

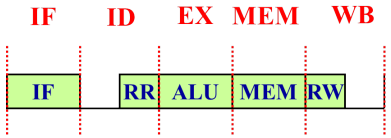
How to change the Multi-Cycle implementation to Pipelined?



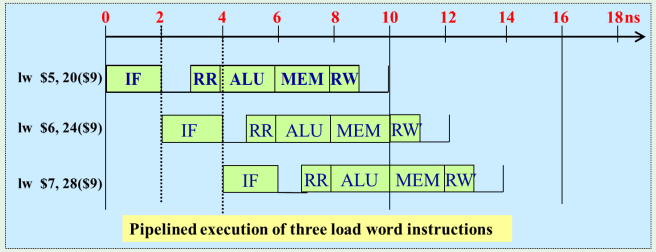
In order to change the multi-cycle implementation to pipelined:

- we notice that some resources cannot be shared as they are needed in every clock cycle for different instructions
- all instructions should have the same execution latency

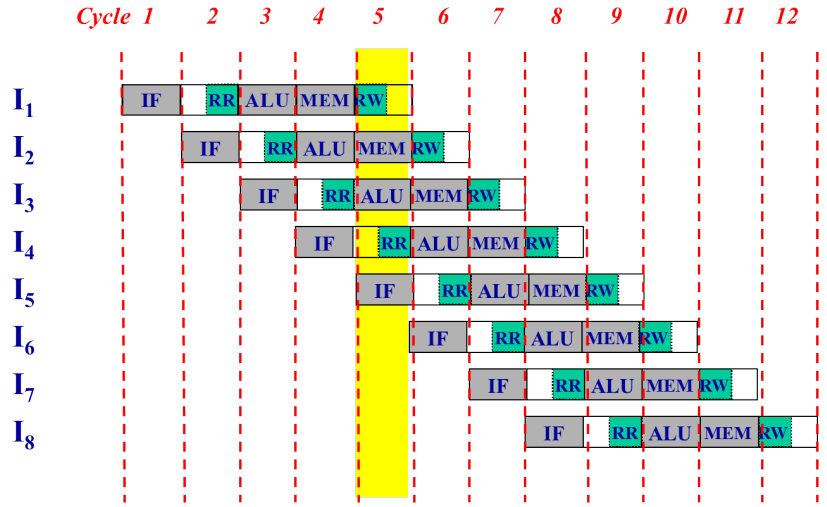
Assuming that the main functional units (ALU and memory access) have a latency of 2 ns, register file read or write takes 1 ns, and ignoring the latency of the other components, we can have a 5-stage pipeline.



Example



Example: Pipelined execution of 8 instructions



Pipelined Datapath

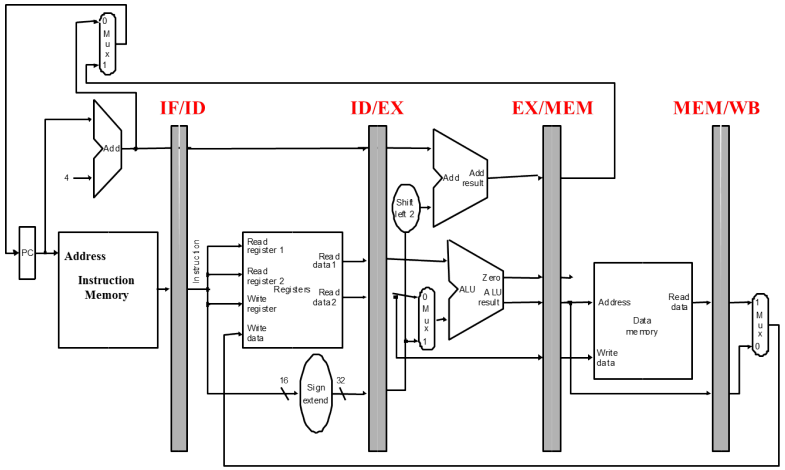


Figure 6.12 of ref3

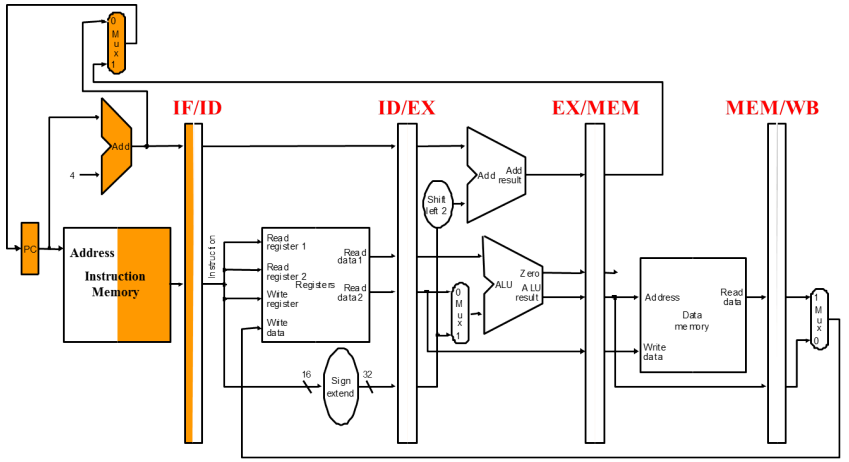


Figure 6.13 of ref3

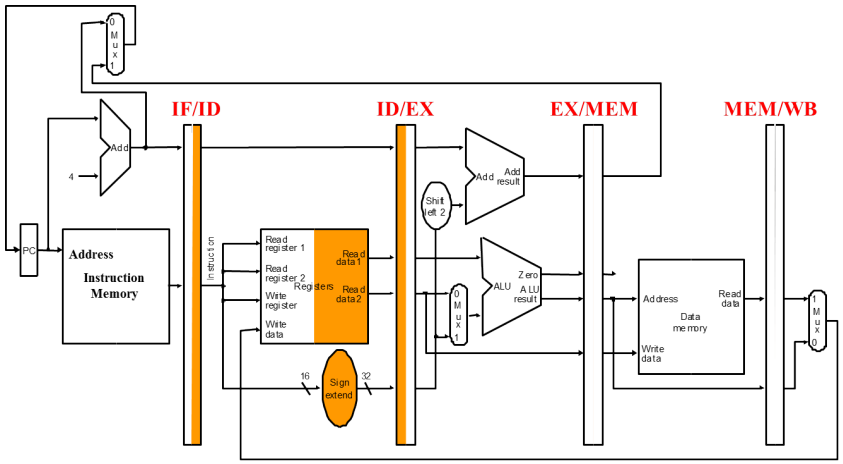


Figure 6.13 of ref3

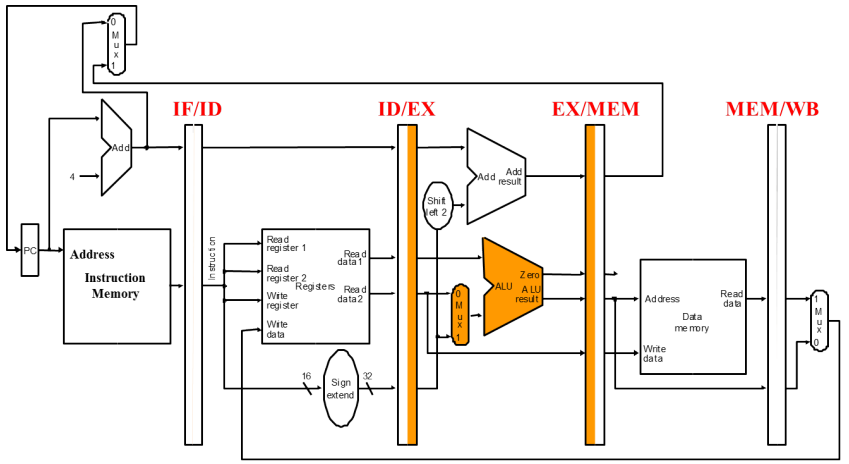


Figure 6.14 of ref3

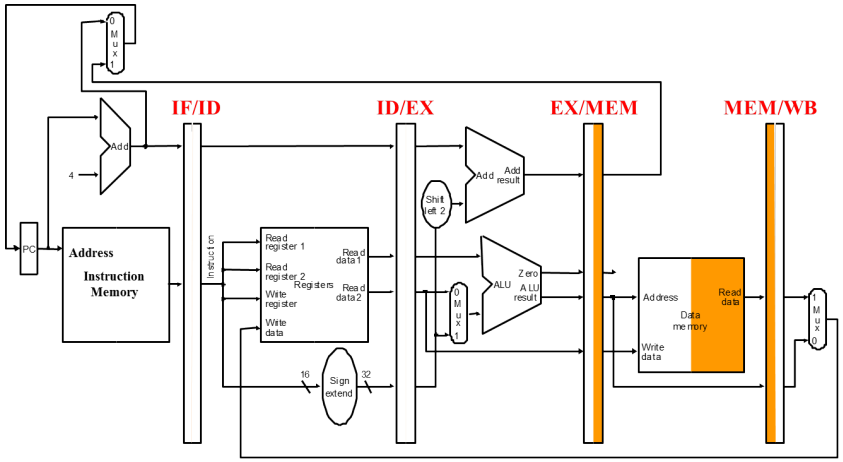


Figure 6.15 of ref3

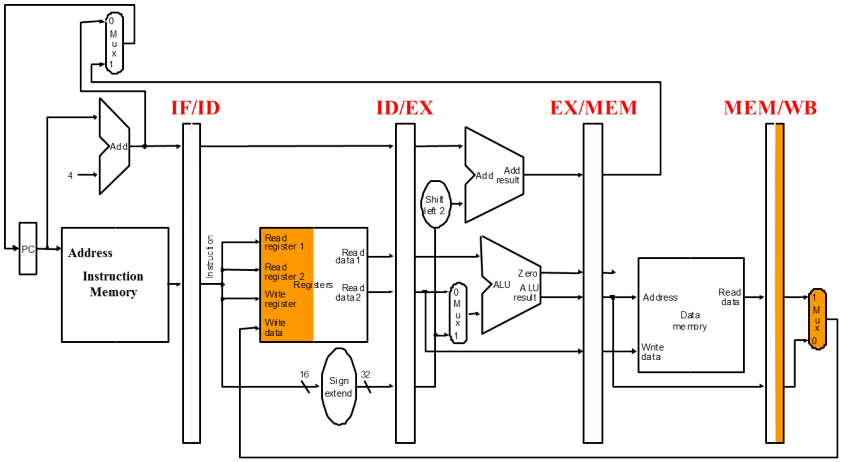


Figure 6.15 of ref3

Control Unit for the Pipelined Implementation

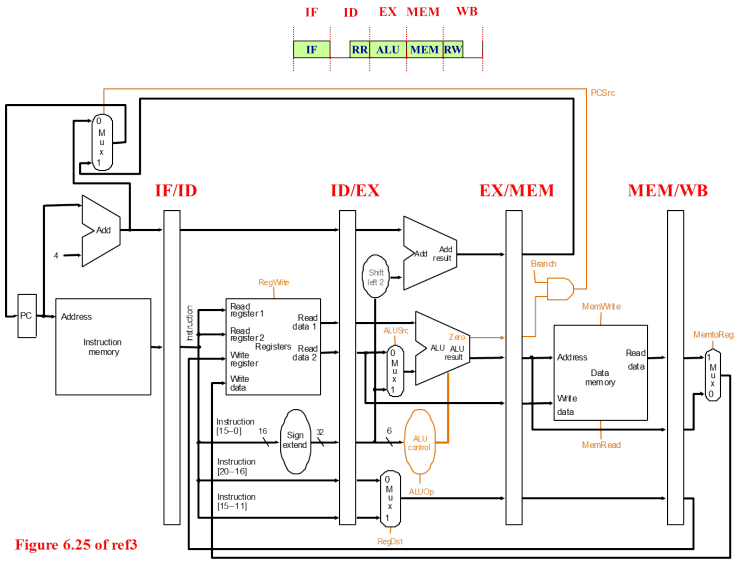


Figure 6.25 of ref3

We need to identify which control signals should be activated in each pipeline stage:

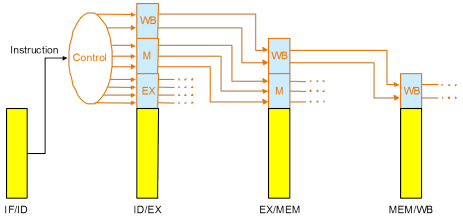
We investigate the need for a control signal in each of our following pipeline stages:

- *Instruction Fetch (IF)*
- *Instruction Decode / register file read (ID)*
- ***Execution / address calculation (EX)***
- ***Memory access (MEM)***
- ***Write back (WB)***

The value of control lines in each pipeline stage is given below. Note that there are no need to activate a specific control line in pipeline stages IF and ID.

Instruction	EX pipeline stage control lines				MEM pipeline stage control lines			WB pipeline stage control lines	
	Reg Dst	*ALU Op1*	*ALU Op0*	*ALU Src*	*Branch*	*Mem Read*	*Mem Write*	*Reg Write*	*Memto Reg*
R-type	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Passing the value of control lines to the appropriate points through the pipelines registers.



The pipelined datapath with its control signals

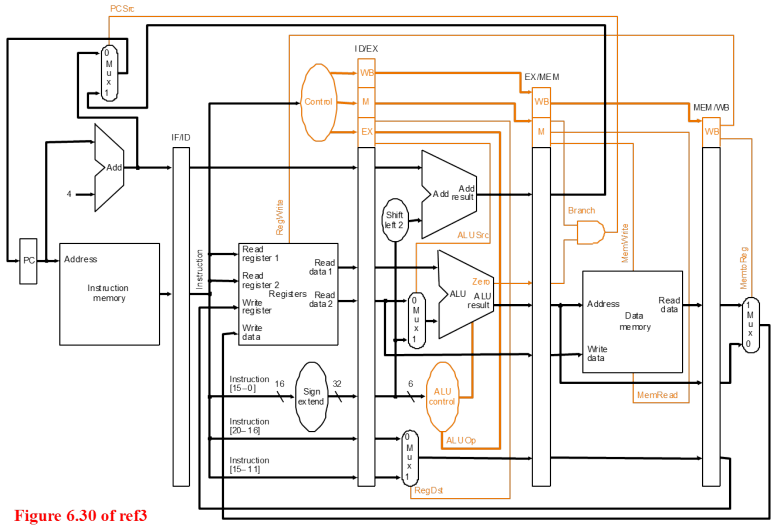
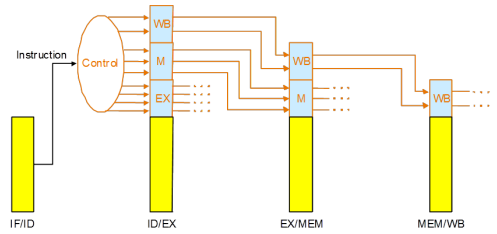
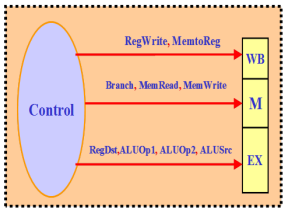


Figure 6.30 of ref3

In summary

Instruction	EX pipeline stage control lines				MEM pipeline stage control lines			WB pipeline stage control lines	
	Reg Dst	*ALU Op1*	*ALU Op0*	*ALU Src*	*Branch*	*Mem Read*	*Mem Write*	*Reg Write*	*MemtoReg*
R-type	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X





Example

Using the following pipelined datapath, determine (as much as you can) the five instructions in the five pipeline stages. (If you cannot fill in a field of an instruction, state why). Use the given appendices in the following slides.

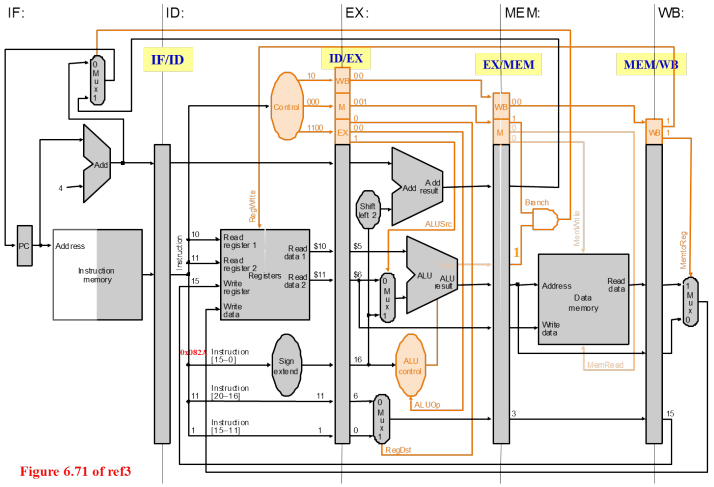


Figure 6.71 of ref3

R-type instruc-tions	Example						
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
	OP	Rs	Rt	Rd	shamt	funct	
add	0	2	3	1	0	32	*add $1, $2, $3*
addu	0	2	3	1	0	33	*addu $1, $2, $3*
sub	0	2	3	1	0	34	*sub $1, $2, $3*
subu	0	2	3	1	0	35	*subu $1, $2, $3*
mfc0	16	0	1	14	0	0	*mfc0 $1, $epc*
mult	0	2	3	0	0	24	*mult $2, $3*
multu	0	2	3	0	0	25	*multu $2, $3*
div	0	2	3	0	0	26	*div $2, $3*
divu	0	2	3	0	0	27	*divu $2, $3*
mfhi	0	0	0	1	0	16	*mfhi $1*
mflo	0	0	0	1	0	18	*mflo $1*
and	0	2	3	1	0	36	*and $1, $2, $3*
or	0	2	3	1	0	37	*or $1, $2, $3*

R-type instruc-tions (Cnt.)	Example						All values in decimal
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
	OP	Rs	Rt	Rd	shamt	funct	
sll	0	0	2	1	10	0	*sll $1, $2, 10*
srl	0	0	2	1	10	2	*srl $1, $2, 10*
slt	0	2	3	1	0	42	*slt $1, $2, $3*
sltu	0	2	3	1	0	43	*sltu $1, $2, $3*
jr	16	0	1	14	0	0	*jr $31*

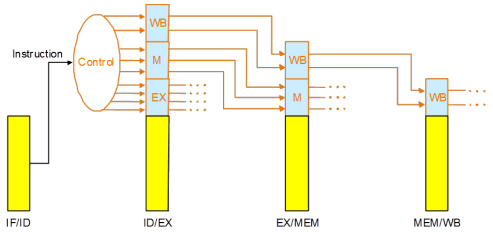
J-type instruc-tions	Example						
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
	OP	target address					
j	2	2500					*j 10000*
jal	3	2500					*jal 10000*

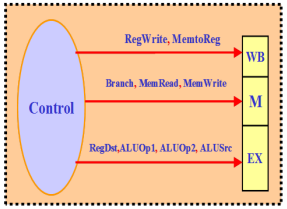
i-type instruc-tions	Example						
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
	OP	Rs	Rt	Immediate/offset			
addi	8	2	1	100			*addi $1, $2, 100*
addiu	9	2	1	100			*addiu $1, $2, 100*
andi	12	2	1	100			*andi $1, $2, 100*
ori	13	2	1	100			*ori $1, $2, 100*
lw	35	2	1	100			*lw $1, 100($2)*
sw	43	2	1	100			*sw $1, 100($2)*
lui	15	0	1	100			*lui $1, 100*
beq	4	1	2	100			*beq $1, $2, 100*
bne	5	1	2	100			*bne $1, $2, 100*
slti	10	2	1	100			*slti $1, $2, 100*
sltiu	11	2	1	100			*sltiu $1, $2, 100*

Instruction Opcode	ALUop	Instruction Operation	Funct field Inst[5-0]	Desired ALU action	ALU control input
LW	00	load word	xxxxxx	add	010
SW	00	store word	xxxxxx	add	010
BEQ	01	branch equal	xxxxxx	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	Set on less than	101010	set on less than	111

How the ALU control bits are set depends on the ALUop control bits and the different function codes for the R-type instruction. **Figure 5.14 of ref3**

Instruction	EX pipeline stage control lines				MEM pipeline stage control lines			WB pipeline stage control lines	
	Reg Dst	*ALU Op1*	*ALU Op0*	*ALU Src*	*Branch*	*Mem Read*	*Mem Write*	*Reg Write*	*MemtoReg*
R-type	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X





IF stage:

ID stage:

EX stage:

MEM stage:

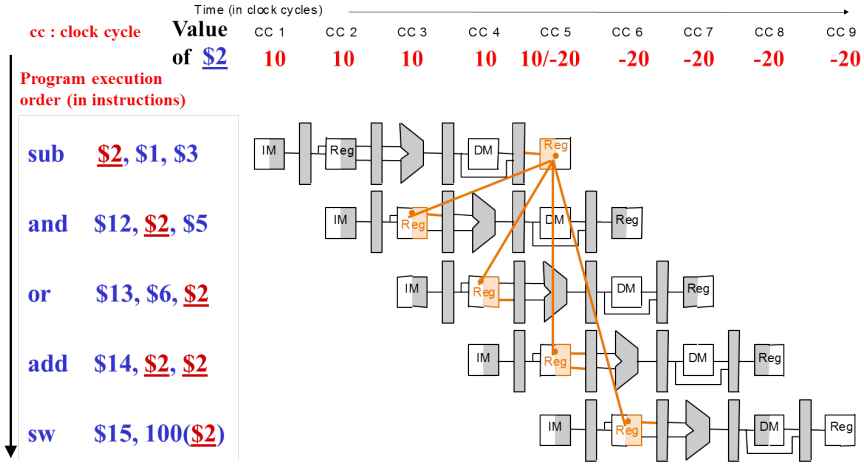
WB stage:

Example

Consider a non-pipelined processor (machine M1), which has a 1 ns clock cycle period. It takes 4 cycles for ALU operations and branches, and 5 cycles for memory operations to execute. Assume that the relative frequencies of these operations are 40%, 20%, 40% respectively. Suppose we designed a pipelined processor (machine M2), which has a 1.2 ns clock cycle period (0.2 ns is added to 1 ns non-pipelined cycle due to pipeline implementation issues such as clock skew, setup/hold times, ...).
How much speedup can be achieved through using machine M2?

Pipeline Hazards

Are the results of the following code correct (using our pipelined implementation)? (Assume the initial values in regs. $1, $2 and $3 are 10, 10 and 30 respectively).



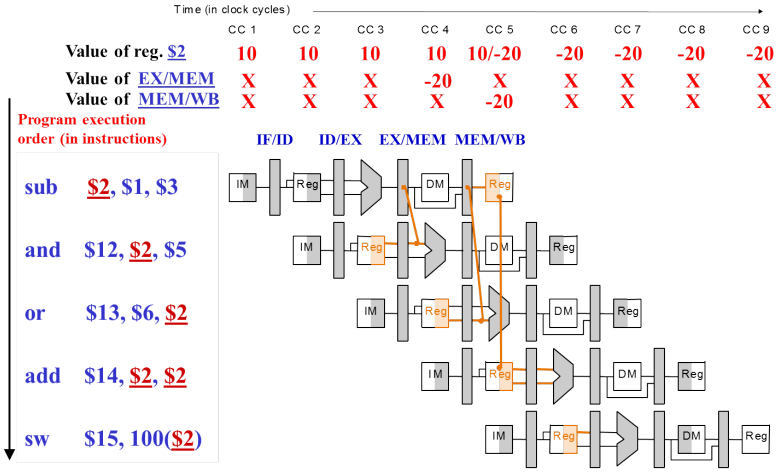
There are situations, called ***pipeline hazards***, that prevent the next instruction in the instruction stream to execute properly during its designated clock cycle. ***Hazards*** reduce the performance and if not handled properly wrong results may be generated. So, pipeline might be ***stalled*** or some other ways should be used to prevent wrong results .

Three different types of pipeline hazards:

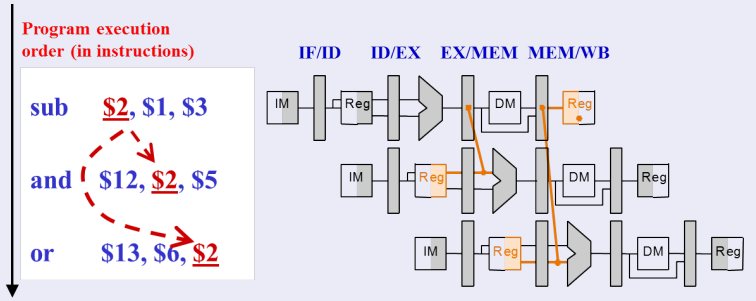
- ***Structural hazards***, which are present when resource conflicts occur when instructions simultaneously access the same resource.
- ***Data hazards***, which occur when an instruction depends on the results of a previous instruction which is needed to be ready for the execution of the current instruction.
- ***Control hazards*** may occur when branches are in the pipeline with other instructions and the PC value is to be changed.

Detecting and resolving Data Hazards in the pipeline

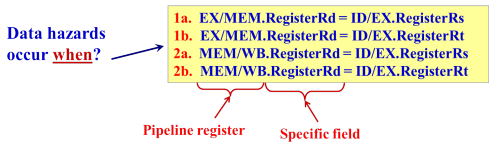
What happens in our pipeline stages in the following case:



How to detect data hazards?



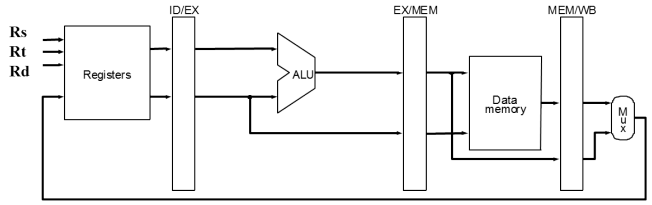
Data hazards can be detected through checking the proper pipeline registers.



The above is valid for ALU instructions. So, *RegWrite* control signal should be checked.

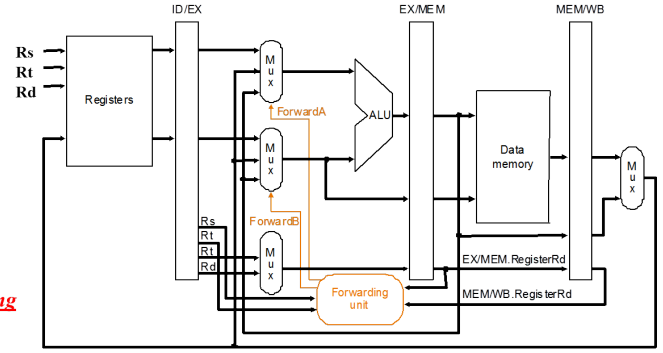
part of datapath

No forwarding



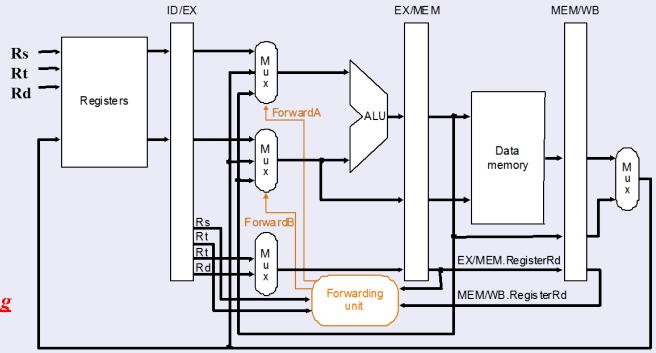
part of datapath

with forwarding



part of datapath

with forwarding



Detecting data hazard in EX stage and forwarding:

if (EX/MEM.RegWrite & (EX/MEM.RegisterRd != 0) & (EX/MEM.RegisterRd = ID/EX.RegisterRs))

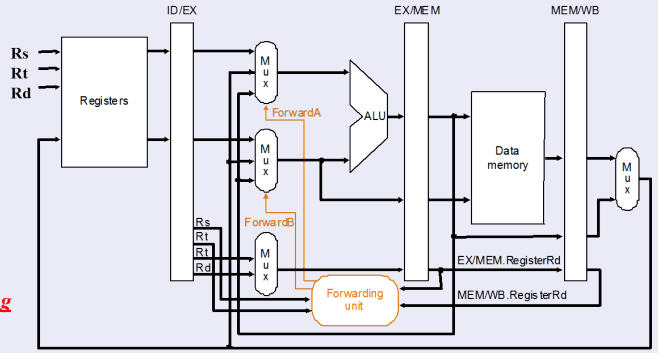
ForwardA = 10

if (EX/MEM.RegWrite & (EX/MEM.RegisterRd != 0) & (EX/MEM.RegisterRd = ID/EX.RegisterRt))

ForwardB = 10

part of datapath

with forwarding



Detecting data hazard in MEM stage and forwarding:

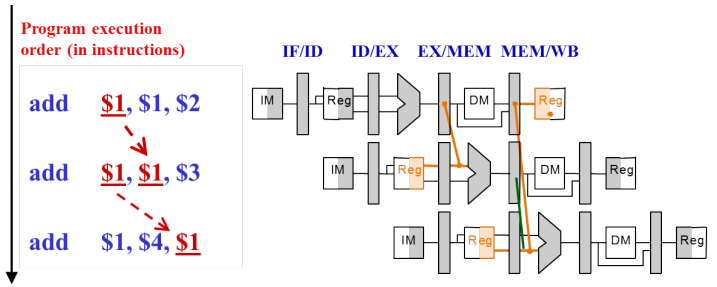
if (MEM/WB.RegWrite & (MEM/WB.RegisterRd != 0) & (MEM/WB.RegisterRd = ID/EX.RegisterRs) & (EX/MEM.RegisterRd != ID/EX.RegisterRs))

ForwardA = 01

if (MEM/WB.RegWrite & (MEM/WB.RegisterRd != 0) & (MEM/WB.RegisterRd = ID/EX.RegisterRt) & (EX/MEM.RegisterRd != ID/EX.RegisterRt))

ForwardB = 01

Summary of forwarding mechanism (for Data Hazard):



The above data hazard can be resolved through result **forwarding** (for ALU instructions).

- We check at the EX stage if one of the source registers of the instruction in ID stage is the same as the destination register of the pervious instruction(s) either in the EX/MEM or MEM/WB pipeline registers. If that is the case, then the appropriate value is forwarded from the pipeline registers (EX/MEM or MEM/WB) to the input of ALU.

Control values for the forwarding multiplexors:

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

References and Recommended Readings

1. David A. Patterson and John L. Hennessy, *Computer Organization & Design – The Hardware/Software Interface*, 4th Edition Morgan Kaufmann Publishers 2009. **(Chapters 1 to 4) and Appendix A (HP_AppA.pdf)**
2. John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd Edition, Morgan Kaufmann Publishers (Elsevier Science) 2003. **(Chapters 2)**
3. David A. Patterson and John L. Hennessy, *Computer Organization & Design – The Hardware/Software Interface*, 2nd Edition Morgan Kaufmann Publishers 1998. **(Chapters 4, 5 and 6)**