UNIT II – Chapter 1

PIPELINING

(Reference text book: Patterson Appendix A)

What is pipelining?

- Pipelining is one of the implementation techniques of overlapping the execution of instructions.
- One of the techniques that takes advantage of parallelism and is considered as one of the key feature to enhance CPU performance at present.
- Pipelining technique is made of different steps were each step completes the part of execution in parallel and each of these steps are called **pipe stage** or **pipe segment**.

Some basic pipeline concepts:

1. **Throughput**: Throughput of an instruction pipeline is time taken to complete the execution of single instruction. Basically it tells how often an instruction exits the pipeline.

2. **Processor Cycle**: Time required in moving an instruction from one stage to another stage in a pipeline is called **processor cycle** of a pipeline.

→ In ideal pipeline each stage has to complete in one clock cycle and is chosen equal to time required for processing longest stage. **(For ex if there is a pipeline of three stages s1, s2 and s3 having cycle time equal to 10, 20 and 30ns respectively. then cycle time is chosen equal to 30ns which is longest of all.)**

3. **Speed up of pipeline:** speed up of pipeline is defined as the ratio of CPU time in un pipelined processor to CPU time in pipelined processor which is given by the expression:

$$\textbf{Speed up} = \frac{\textbf{CPU time un pipelined}}{\textbf{CPU time pipelined}}$$

Basics of RISC (Reduced Instruction Set Computer) Instruction set:

→Basic pipelining methods are implemented on RISC instruction set. The typical RISC architecture chosen for our study here is MIPS (Microprocessor without Interlocked Pipeline Stages) under RISC.

Some of the characteristics of RISC architectures are:

1. All ALU operation on data applies to data in registers which are of 32/64 bits in length.

2. Only operation that affect memory are load and store operations.

3. Uses very set of instruction formats.

4. All RISC architecture irrespective of versions provide 32 registers for operations which are called register files.

Classes of instruction in MIPS architecture:

MIPS architecture has three different classes of instruction based on the operations performed by them:

1. **ALU instructions:** As name indicates these instructions are used to perform various ALU operations. For operation, these instructions take either two registers or a single register with an immediate value, operate on them and store the result in third register.

→Typical operations of these include add, subtract, or etc. usage of immediate versions are specified by using the mnemonics I as suffix to the instructions (Ex ADDI). It can even specify the signed and unsigned form of data. By default all the operations are signed form and unsigned form is specified by suffix u for the instruction (Ex ADDU).

→Extended 64 bit operations are specified by adding D as suffix or prefix to the instruction (Ex DADD, LD etc)

Example format:

a. ADD $1,$2,$3 [$1←[$2] +[$3]]

Where $2 and $3 are source registers and $1 is a destination register.

b. ADDI $1, $2, 4 [$1←[$2]+4]

2. **Load and Store instructions:**

→ Takes one value from the register source called **base register** and another immediate value called **offset value** and calculates the sum called **Effective address** which points to the address of memory location.

Effective address (EA) = [base register] + offset

→When it is load operation effective address is used to read the data contents from memory location and load into register.

Example: lw $1, 4($3)

EA = [$3] + 4 and $1 is destination register. lw means load word

→When it is store operation effective address is used to write the contents of register into memory location specified in effective address calculation.

Example: sw $1,4($3)

Stores the contents of $1 into the memory location [$3] + 4, sw means store word

3. **Branches and jump instructions:**

→Branches are conditional transfer instructions and jumps are unconditional transfer instructions. Condition can be checked either by conditional flags or by register comparisons.

→Branch destination is obtained by adding an immediate value called offset value to the program counter value.

A simple implementation of RISC instruction set

Implementation of every RISC instruction takes at most 5 clock cycles were each clock cycle correspond to the section of execution.

The 5 clock cycles are:

1. **Instruction Fetch cycle (IF):** operations performed in this stage are:

   i**.** Send PC to memory and fetch current instruction from the memory for the execution.

   ii. Update the value of PC by adding 4 to PC.

2. **Instruction Decode/ Register Fetch cycle (ID/RF):** operations are:

   i. Decodes the instruction and reads the register corresponding to the source operands from the register file.

   ii. Decoding and reading of registers are done in parallel, the process is called **fixed field decoding.**

3. **Execution/ Effective address cycle (EX):** In this stage, ALU operates on the operands prepared in the previous cycle were one of the three functions are performed on them:

   *a. **Memory reference:*** ALU adds base register value and offset value and calculates effective address which is meant for reading contents from memory location or writing the contents into the memory location.

*b. **Register- Register ALU operation:*** ALU unit performs arithmetic or logic operations specified by opcode on register files that is read in previous cycle were in both operands come from the register contents.

*c. **Register – immediate ALU operation:*** Here one operand will be read from register and another value is an immediate value and ALU unit operates on this based on the opcode.

4. **Memory Access cycle (MEM):** This is the only cycle which has access to the memory location. Based on the access type, it can perform two operations: load and store operations.

   →If the instruction is load, memory does read using EA value computed in the prior cycle.

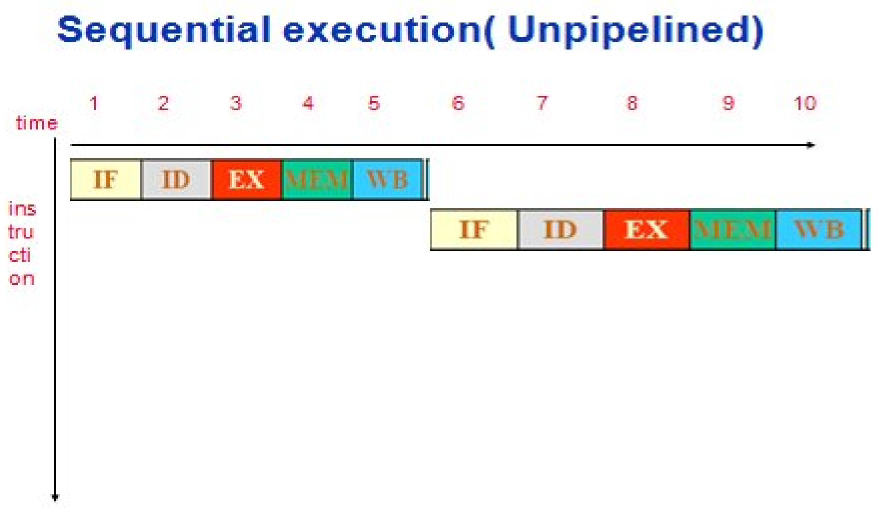
   →If the instruction is store, memory writes data into from second source (operand 2) into the EA address calculated in the prior cycle.

5. **Write-Back cycle (WB):** Final cycle which is used only for ALU and load instructions.

   →If the instruction is load instruction, then data value read in the memory access cycle is written into the register specified.

   →If the instruction is ALU operation, then the computed result in EX cycle is written into the register file.

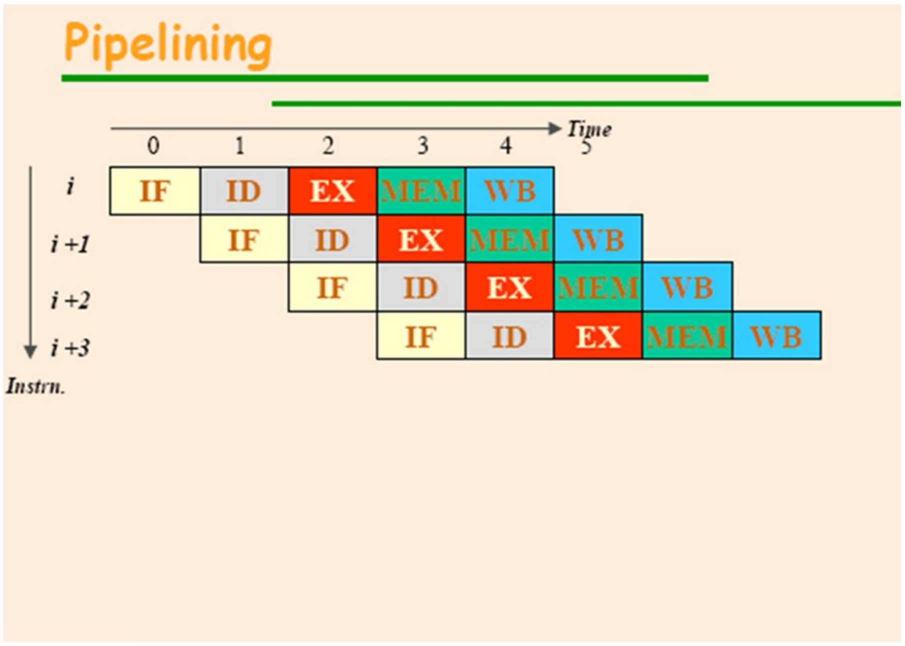
When the following stages are executed sequentially, it is executed in the way shown below through a schematic representation.



The Classic Five Stage pipeline for a RISC processor

→ Pipelining of the execution in RISC processor can be done almost without any changes to the sequential system just by overlapping the sequential stages (Clock cycles) as shown in the figure.

→In the pipeline implementation, each clock cycle becomes a **pipe stage**.



Working of the above pipelined implementation as follows:

→In Every clock cycle H/W initiates the execution of the new instruction

→When the stages are overlapped, the key factor to be considered is that, it is required to make sure not to try two different operation on same H/W resource in same clock cycle, which causes resource conflict.

For example: A single ALU unit cannot do add operation and target address calculation at same time.

Problems arise due to above pipeline implementation technique:

→In the diagram shown above, the memory unit is getting clashed with IF stage and MEM stage, were in both stage requires memory access operation.

→Apart from that, there is one more conflict, were in single register file is used for reading in ID stage and for writing purpose in WB stage.

→Another problem is that in every clock cycle new instruction is fetched and stages do not keep any result with them more than one clock cycle. The condition is called **instruction interference**. (Example: Result of ALU is loaded only in the WB stage but ALU will be occupied by new instruction when write back stage is executed.)

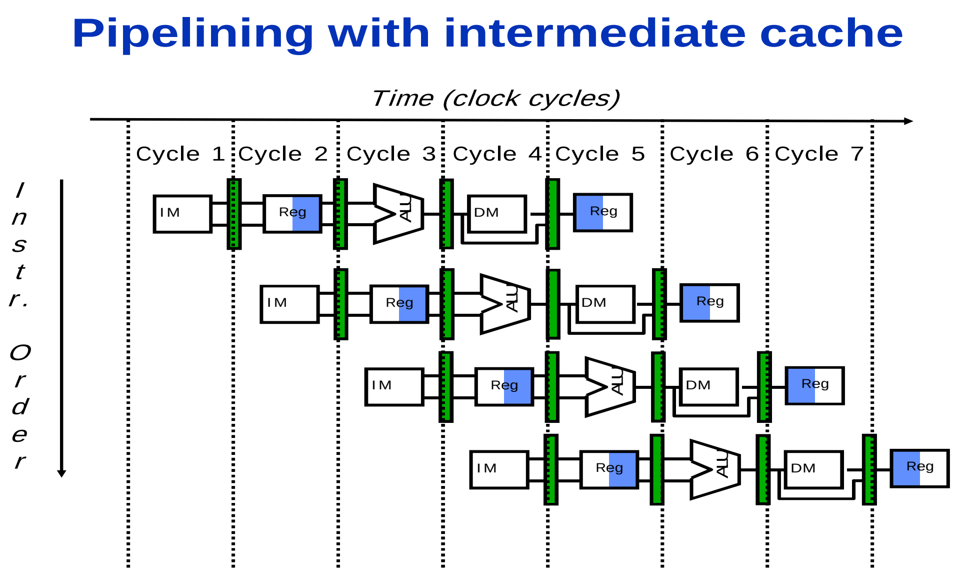
Techniques to overcome above problems: The above problems can be overcome by following enhancements to pipeline stages:

1. Usage of separate memory unit for instruction and data called instruction memory (IM) and Data memory (DM) respectively to overcome memory unit conflict for fetch and memory access.

2. To overcome register file conflict in ID stage and WB stage, first half cycle is used for writing purpose and another half cycle is used for reading purpose of the register.

3. To overcome the problem of interference, register caches are included between each and every stages of the pipeline to store the previous stage outcome.

With these enhancements, the data path for the typical pipeline looks as shown below in the figure:



Performance Issues in Pipelining

Performance limitations for pipelining arise due to following factors:

1. **Limitations from pipeline latency**: Pipeline latency is the delay incurred because of the delay in response from the H/W unit.

2. **Imbalance from pipeline stages:** There is a marginal effect in performance due to the imbalance in pipeline stages. I.e. cycle time for the pipeline is chosen equal to time taken for processing the longest stage. There is wastage of clock cycles for the stages requiring less amount of time.

3. **Limitations due to pipelining overhead:** Pipelining overhead is the combination of pipeline register delay and clock skew.

   →Register delay occurs due to the response of registers for a particular clock edges only.

   →Clock skew is the delay caused by the unit producing clock cycles which may not occur in a specified time.

A MAJOR HURDLE OF PIPELING – Pipeline Hazards

Pipeline hazards are the one that prevent next instruction in the instruction stream from executing in a designated clock cycle.

Hazards are classified into three different classes:

1. **Structural Hazard:** Arises due to resource conflicts when Hardware cannot support the combination of overlapped instruction.
2. **Data Hazard:** Arises when a subsequent instruction depends on the result of previous instructions.
3. **Control Hazard:** Arises from pipelining of branches and other instructions that basically change the value of **program counter.**

Impact of any of the above hazards in pipeline can cause **stalls (bubbles)** which are nothing but the wastage of clock cycles in the pipeline execution. Whenever stall occurs the later instructions are issued after the stall elimination.

Structural Hazards:

→If some combination of instruction execution in pipeline cannot be accommodated because of resource conflicts, they are called as **Structural hazards.**

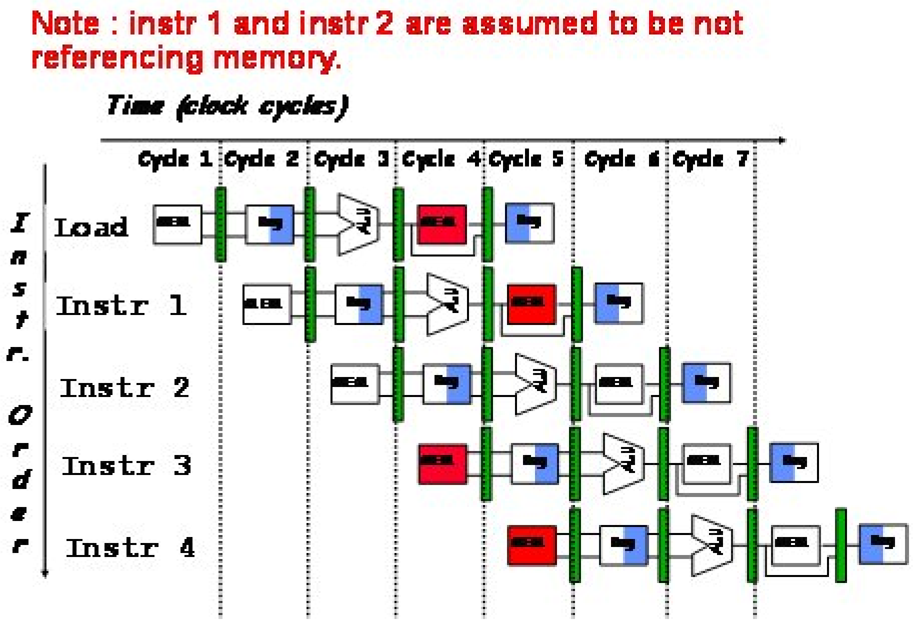
→**These hazards basically arise in two cases:**

1. If the functional unit used in pipelining is not supporting the pipeline operation.

   Example: If a serial adder is used in ALU in place of Carry look ahead adder, then pipeline execution cannot be implemented.

2. When resource is not duplicated enough.

Example: When a single memory unit is used for both Instruction fetch and Memory access stage, it leads to resource conflict.

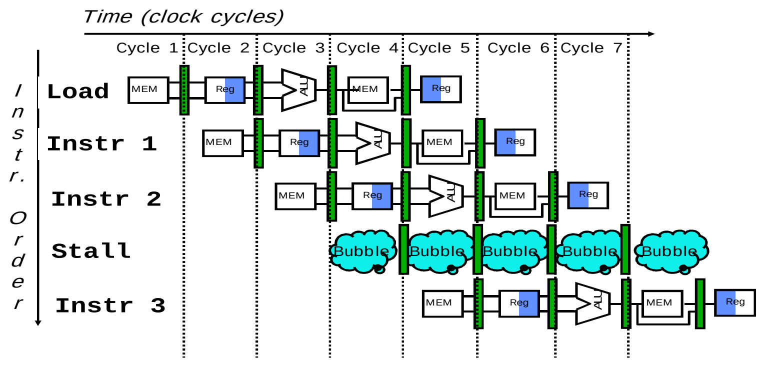


In the above example, the pipeline system uses only one memory unit for both **IF** and **MEM** stage. I.e. there is no resource duplication in pipelining. In the above figure, there is a resource conflict in **cycle 4** with **MEM** stage of load instruction and **IF** stage of Instr 3.

Methods to overcome Structural Hazard:

When these types of hazards occur in a pipeline, to resolve this pipeline will stall until a unit is available free. Stall is also called **bubble or pipeline bubble.** The stall makes a negative impact on performance of the pipeline by increasing CPI value. The approach is shown below:

One Memory Port/Structural Hazards



Data Hazards:

→These hazards occur when a subsequent instruction depend on the result of the prior instructions. I.e. in pipelining the read /write access order will be changed when compared to sequential execution.

→As an example, consider the following set of instructions:

I1: add r1, r2,r3

I2: sub r4, r1,r3

I3: and r6, r1,r7

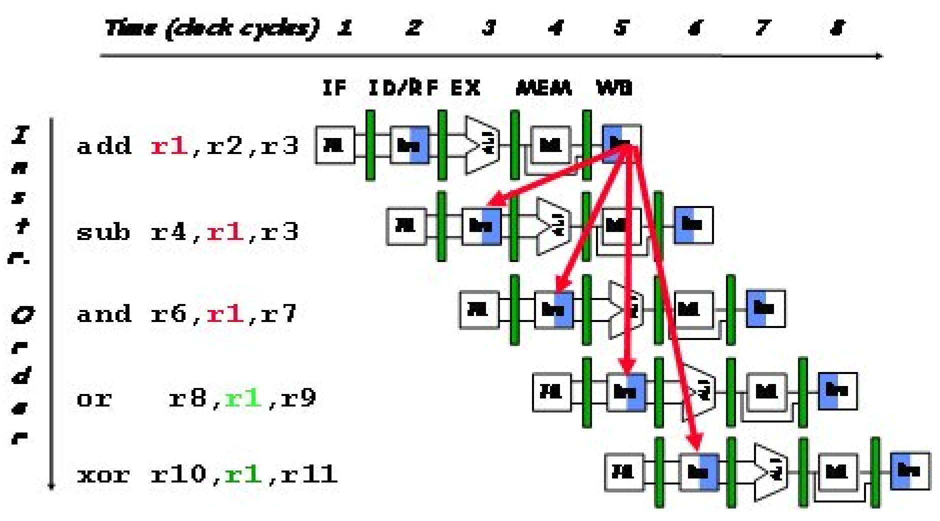
I4: or r8, r1,r9

I5: xor r10,r1,r11

Following data dependencies occur in the above instruction set:

1. In instruction I2, r1 is a source register, which is destination register in instruction I1.
2. In instruction I3, r1 is a source register which is again destination register in I1.
3. In instruction I4, r1 is a source register which is again destination register in I1.
4. In instruction I5, r1 is a source register which is again destination register in I1.

When above instructions are implemented in a pipeline, we will see how these data dependencies cause the hazard in the program as shown schematically:



In the above figure above, hazards are caused in two conditions:

1. Instruction 2 requires the value of register r1 which is destination register in add instruction and is available only at the end of WB stage in clock cycle 5. But register fetching is done in ID stage of add instruction in clock cycle3. Because of which it reads a wrong value which imposes **Data hazard.**
2. In second condition, consider the third instruction which is doing and operation were again r1 is used as source register, which is a destination register in very first instruction. Even in this case, the value of register is written in WB stage in clock cycle 5 and register reading is done in IF stage of clock cycle 4 which again leads to **Data hazard.**

To conclude this, whenever data hazard occurs in the 5 stage pipeline program, it can occur only in connection to previous two instructions from the current execution.

Three generic Data hazards

Data hazards are classified into three different categories:

1. Read After Write (RAW)

2. Write After Read (WAR)

3. Write After Write (WAW)

We will discuss each one of them very briefly

Read After Write (RAW):

→A type of data hazard called as **dependence hazard**.

→Consider the following instructions:

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

→If instruction J tries to read the operand before instruction I writes it, then the hazard is called as **Read After Write hazard**.

Write After Read (WAR):

→This type of hazard is called as **anti-dependence hazard**.

→Consider the following instructions:

```
I: sub r4,r1,r3
J: add r1,r2,r3
```

In this case data hazard occurs when instruction J writes operand before instruction I reads it, then the hazard is called as **Write After Read hazard.**

Write After Write (WAW)

→This type of hazard is called as **Output-dependence hazard.**

→Consider the following instructions:

```
I: sub r1,r4,r3
J: add r1,r2,r3
```

In this case data hazard occurs when instruction J writes the operand before instruction I writes it which leads to a wrong final value.

Techniques for minimizing Data Hazard Stalls: Forwarding

→Stalls occurring in pipelining can be reduced by using a simple Hardware technique called **forwarding or bypassing or short circuiting.**

→To get a clear picture of this technique consider the following instructions:

I1: add r1, r2,r3

I2: sub r4, r1,r3

I3: and r6, r1,r7

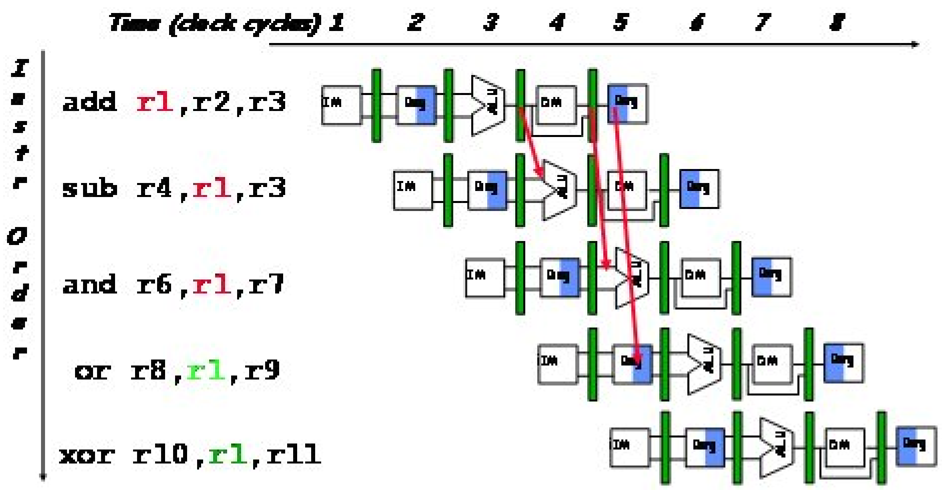
I4: or r8, r1,r9

I5: xor r10,r1,r11

→In the above set of instructions, instruction I2 rather than requiring the value of r4, it requires the result of operation r2+r3 computed by ALU in EX stage. Same in the case of instruction I3 also.

→Based on the above observation made, forwarding is implemented accordingly: ALU results computed are fed back to ALU as a source operand in next cycle. There is hardware to detect the dependencies and implements forwarding accordingly in the next clock cycle.

→Figure shown below demonstrates the implementation of operand forwarding technique:



In the above example second instruction ALU unit require the result of r2 and r3 in cycle 4 which was calculated at the end of cycle 3 by ALU unit. So the result of ALU is feedback to ALU in the next cycle. Another condition is that in cycle 5, even instruction 3 requires the result of the ALU operation of first instruction performed in cycle 3 which will be moved into DM unit in cycle 4 and value from DM/WB register is forwarded into ALU in cycle 5.

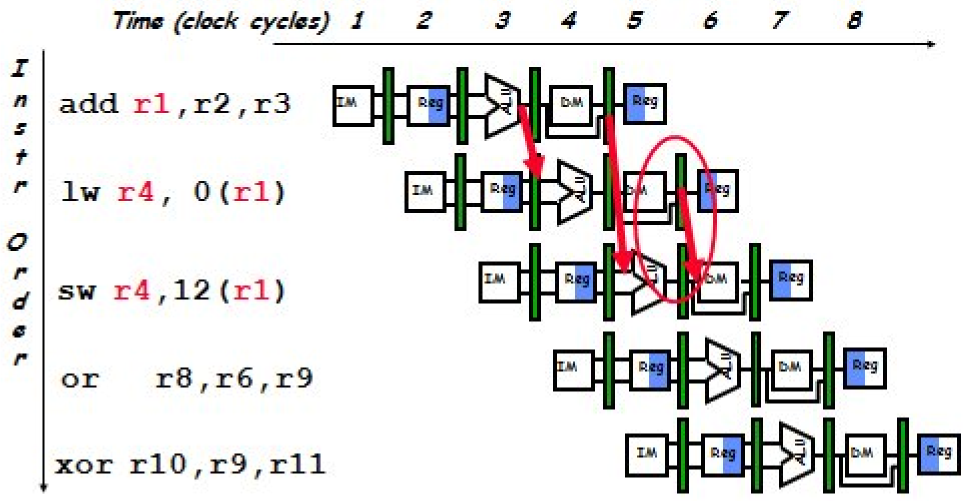
On conclusion, whenever there is a data hazard with the very previous instruction, then operand is forwarded from ALU back to ALU. If the data hazard lies in connection with previous to previous instruction, then value is forwarded from DM back into ALU.

Data Hazards occurrence and avoidance in load/ store instructions

Data hazards occurring due to load/store instruction are bit different compared to other ALU instructions one of which we have seen just in a last section. Hazards in these types of instructions can also occur due the requirement of source operand by memory element. To see how exactly it occurs and how to recover the type of hazard, consider the following instructions:

```
add r1,r2,r3
lw r4, 0(r1)
sw r4,12(r1)
or   r8,r6,r9
xor r10,r9,r11
```

→When the above instructions are implemented in pipeline, we can get a clear picture on how data hazards occur due to load/store instructions as shown below:



→Looking into the pipeline implementation shown above, there data dependencies occurring at different level:

→Second instruction (lw) requires the ALU result of first instruction (add) as the source operand. I.e. register r1 is required to calculate the Effective Address (EA). This can be resolved by operand forwarding into ALU unit which is being discussed in last section.

→Another condition is that the store instruction requires the value of r1, result of add instruction for calculating the Effective Address (EA), which is available by operand forwarding from DM in clock cycle 4.

→One more case of data dependency is r4 is the source register for storing the contents into the memory location, which is the destination register in load instruction which is a previous instruction. This is the dependency in **load/store** instructions. In this case r4 is available in the WB stage of load instruction which is completed in clock cycle 6. But even value of r4 is required as source operand in clock cycle 6, which causes a resource conflict. To overcome again the operand forwarding technique is used in a different method. Store instruction actually requires the contents of memory location 0+ [r1], which is read in MEM stage in cycle 5 of load instruction. So operand forwarding is done from DM to DM in next clock cycle.

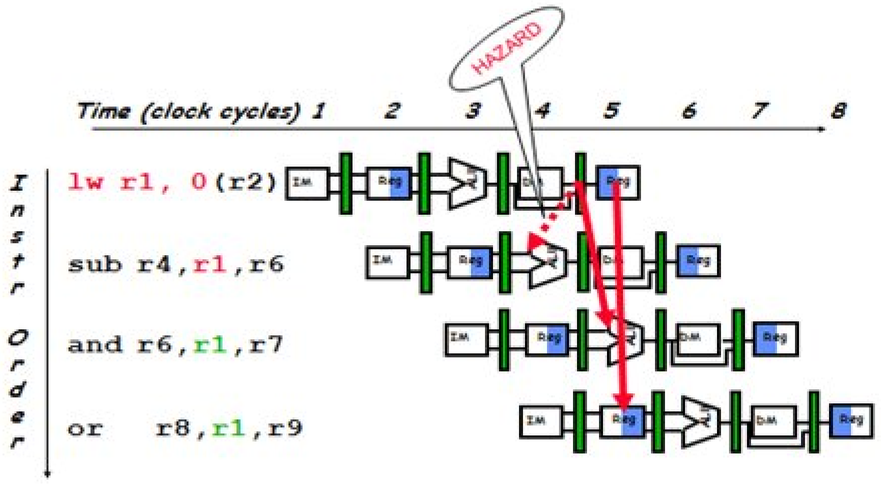
To conclude this, whenever there is data dependency in load/store instructions, the operand forwarding technique has to feedback the result of DM into DM in the next cycle.

Data Hazards Requiring Stalls

There are Data hazards occurring in some set of instructions which can never be eliminated without using the **stalls** in the pipeline. One such condition is shown in set of instructions shown below:

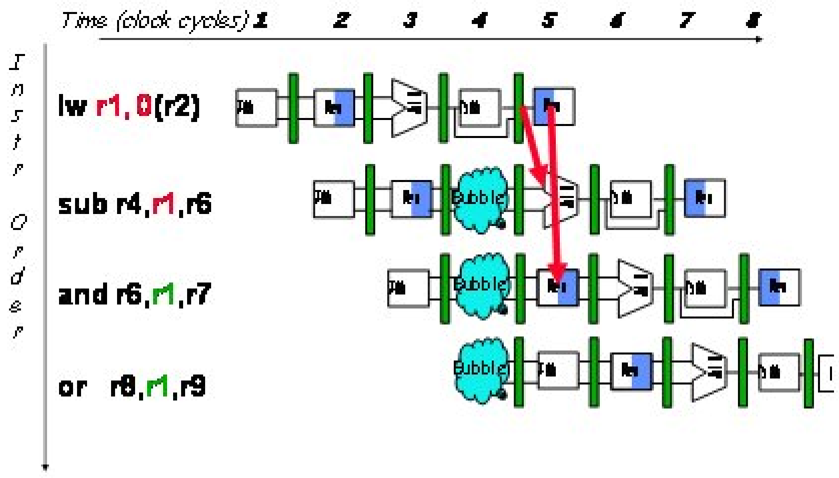
```
 lw r1, 0(r2)
sub r4,r1,r6

and r6,r1,r7

or   r8,r1,r9
```

The pipeline implementation for these instructions is as shown below in the figure:



Consider instruction 1, load instruction and instruction 2, sub instruction. In load instruction r1 is destination register and in sub instruction r1 is source register. At the best case even if you implement forwarding technique, result of load instruction is available only at the end of clock cycle 4 I.e. DM stage of load instruction and sub instruction requires the result of load instruction as a source operand in the same clock cycle I.e. at starting of cycle 4 which causes the **hazard** that cannot be overcome even with any of the **forwarding technique** discussed so far.

→These types of data hazards can never be eliminated, only way to guarantee correct execution among them is to introduce **pipeline stalls** wherever the data hazard has occurred as shown below:



→To introduce a pipeline stalls whenever hazard of this kind occurs, there is a special hardware unit called **pipeline interlock** which detects the hazard of this kind and introduces stalls until the hazard is cleared wherever required.

Branch Hazards

→These kinds of hazards are also called as **control hazards.**

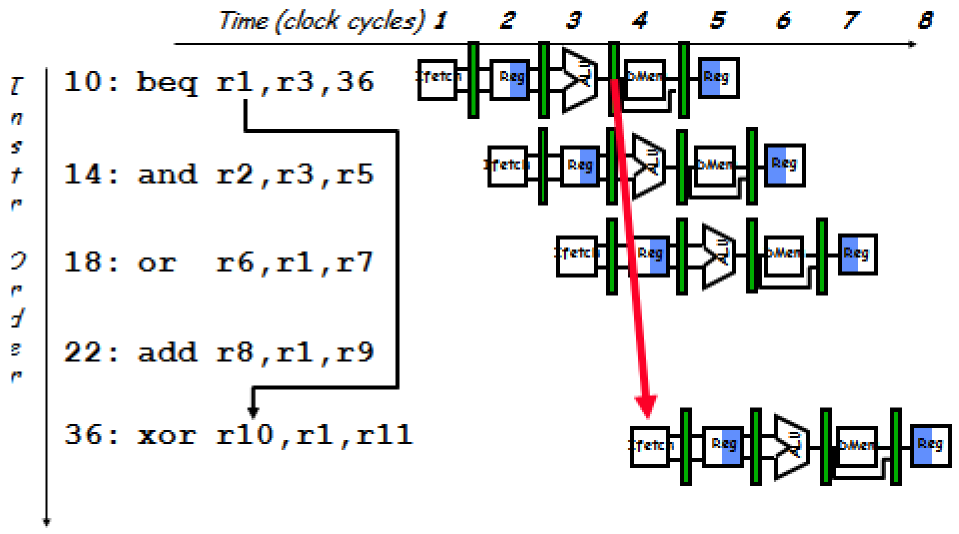
→When a branch is executed, based on the outcome it may or may not change the contents of **PC** value other than normal practice which points to next sequential address. I.e. when branch is evaluated to true, it changes the PC value to the **target address** specified in the instruction.

→Before looking into branch hazards in detail, we will look into some of the important terms related with it:

1. **Taken Branch:** If the instruction updates the contents of PC to target address specified, then it is called as a **taken branch.** I.e. in this condition branch is evaluated to true.

2. **Not taken Branch:** If the instruction does not change the PC to target address specified, then it is called as **Not taken Branch.** I.e. in this condition branch will be evaluated to false.
3. **Branch Penalty:** Wastage of clock cycles occurring due to branch hazards is called Branch Penalty**.**
4. **Branch Delay Slot(s):** Wrong set of instructions executed due to branch hazards are called as Branch Delay Slots.

→Consider the following pipeline implementation to get a clear picture of Branch hazard:



→In the above implementation, instruction 10 is a branch instruction and is evaluated only at the end of clock cycle 3 by the EX stage. I.e. until then it is not known if branch is taken or not taken and contents of PC will not be updated until then if it is taken.

→In the above case, it is assumed to be taken and it updates the PC to target address I.e. address 36 at the end of clock cycle 3 and instruction at address 36 will be fetched only at clock cycle 4 which should have been fetched at clock cycle 2. Because of this there is wastage of two clock cycles **cycle 2** and **cycle 3** fetching the unwanted instructions which is considered as **Branch penalty** in this case.

→Due to delay in branching, two wrong instructions at address 14 and at address 18 will be executed and this slot is called as **Delay slot** in the figure shown above.

Methods to reduce Pipeline Branch Penalties

→Pipeline branch penalties can be reduced by implementation of different methods. There are four basic methods discussed here:

1. Flush pipeline – Redo fetch
2. Predicted Not Taken
3. Predicted Taken
4. Delayed Branching

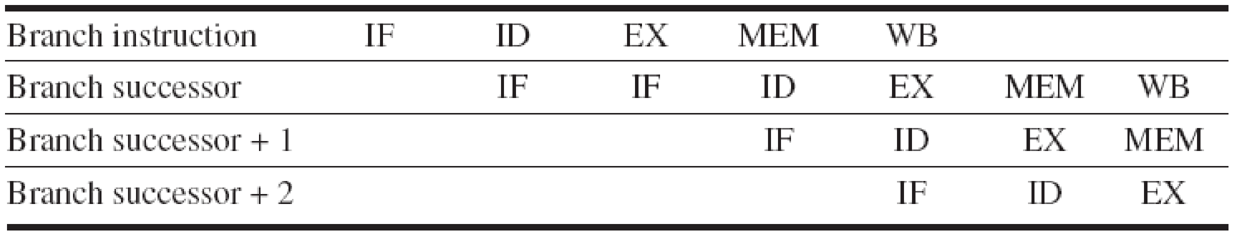
In all the above mentioned methods, actions are static I.e. they are done at compile time itself. We will discuss the required one in detail.

Flush Pipeline

→One of the simplest methods to reduce pipeline branch penalties. It is also called as **redo fetch** method. The method works for the pipeline system were the branch instruction is evaluated at the end of ID stage.

→As the name indicates, the fetching operation is repeated for a particular situation. I.e. whenever there is a branch instruction fetched for a particular clock cycle, for the instruction following the branch instruction carry out instruction fetching operation two times. I.e. repeat the IF stage one more time instead of entering the ID stage.

→Example shown below gives a clear picture on how it is done exactly



Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

→In this example after the fetching of branch instruction, for the next branch successor instruction, fetching operation is done two times because if the branch is taken PC will be updated to branch target address at the end of ID stage of branch instruction. So calling the IF second time for branch successor instruction will fetch Branch Target Address instruction if the branch is taken. If it is not taken then it will fetch already fetched instruction another time. So there will be no wrong instruction executed in the slot.

→Another point is that, if the branch is taken, first IF becomes redundant and if the branch is taken, second IF becomes redundant. Finally this method forces at least one IF to be redundant.

Disadvantage: There will be minimum one clock cycle stall for every branch instruction irrespective of the condition whether it is taken or not taken as one of the IF becomes redundant which leads to wastage of that cycle.

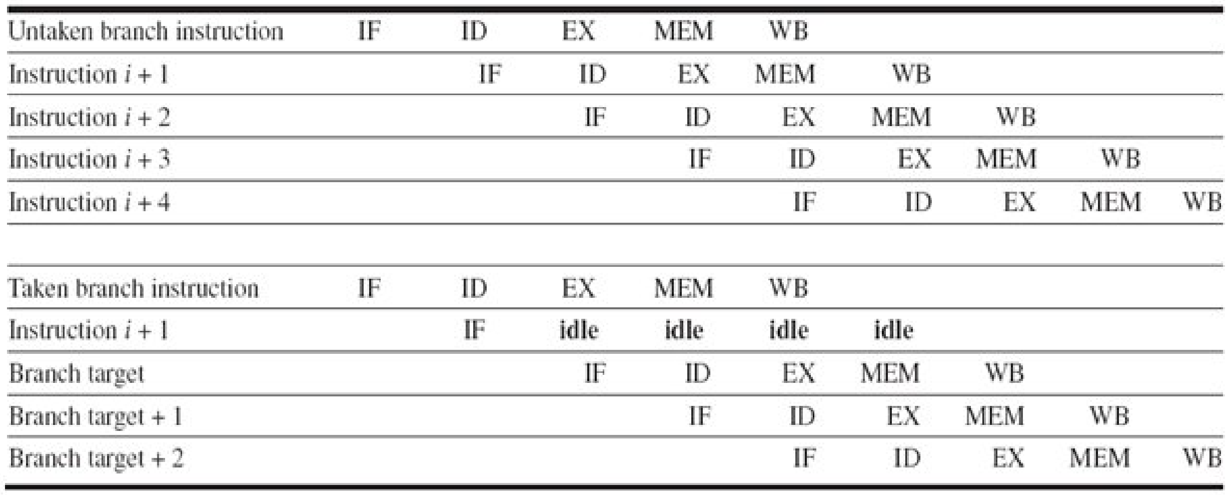
Predicted Not Taken

→In this scheme, all the branch instructions to be executed in program are predicted as **Not Taken.** I.e. execution continues in a sequence whether it is take or not taken.

→If a branch is Not Taken after evaluation, the prediction becomes correct and there will be no hazard taking place.

→If a branch is taken after evaluation, the wrong instruction fetched is turned into a no-op and restart the fetch at the target address.

→Figure below shows both the situation of taken and not taken for predicted not taken scheme



Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

→As shown in the figure, top section shows the untaken branch and according to the prediction execution continues in sequence and there is no effect of any hazards by branch

→Bottom part shows the taken branch scheme were the prediction goes wrong and a wrong instruction (instruction i+1) in sequence is fetched which is turned to idle states after evaluation and a stall of one clock cycle is induced in a pipeline.

Predicted Taken Scheme

→This scheme is exactly opposite to what we have discussed in the last section. All the branch instruction occurring in a program is predicted as taken and instruction in **branch target address** is fetched for execution.

→Anyways this scheme cannot be implemented for the 5 stage pipeline scheme, since the Branch Target Address is available only after the end of second stage. I.e. At the end of ID stage of all instruction.

Delayed Branch

→One of the best techniques that works reasonably well in a 5 stage pipeline scheme.

→Main principle of this method is using valid and useful instruction under **delay slot.** I.e. the instruction has to be executed and useful whether the branch is taken or not taken.

→To get a clear picture of this method, consider the following sequence of instructions:

branch instruction
sequential successor
branch target if taken

→In the above sequence of instruction, branch instruction is executed and is evaluated to taken or not taken at the end of clock cycle 2 which takes a delay of one clock cycle for the evaluation. In this delay time, a sequential instruction (sequential successor) will be executed which is called as **delay slot**. Above sequence is an example for branch delay of one was in there is only one delay slot.

→If the branch is **taken**, the delay slot (sequential successor) is an invalid instruction fetched and executed.

→The idea of delayed branching is to insert a valid instruction into delay slot such that even if the branch is taken or not taken, it should be a valid execution which does not depend on the outcome of branch instruction.

→**Compiler** takes care of inserting a valid and useful instruction into delay slot statically at compile time and the technique is called **Scheduling delay slot.**

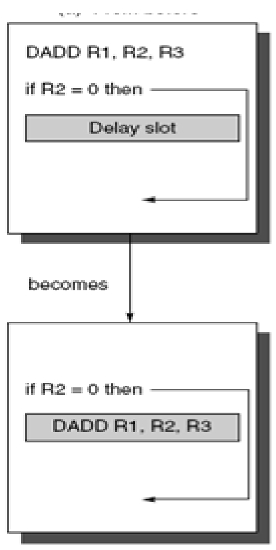
→There are three different methods of scheduling the delay slot:

1. From Before

2. From Target

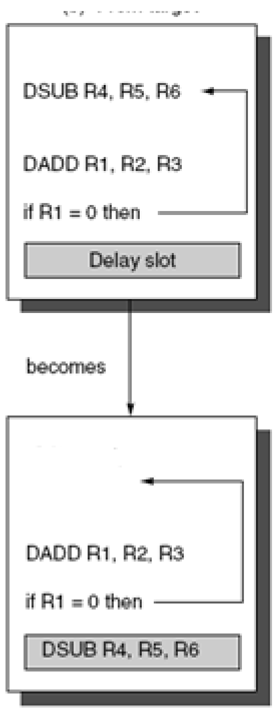
3. From Fall-through

1. **From Before:** consider the example shown below in the figure



→In this method, as you can notice from the above figure, **delay slot** is scheduled with an independent instruction which is just before the branch instruction. It's a sequential instruction that does not depend on the outcome of branch instruction and it has to be executed even if the branch is taken or not taken.

2. From Target: Consider the example shown below:



→In this method, Branch Target address instruction is inserted into delay slot.

→Here for the instruction format shown above, **from before** method cannot be executed because branch instruction depends on the result of previous instruction which means there is a data dependency and previous instruction has to be executed first and then branch instruction is executed.

→The instruction resembles the action of do while loop as shown below:

do

{

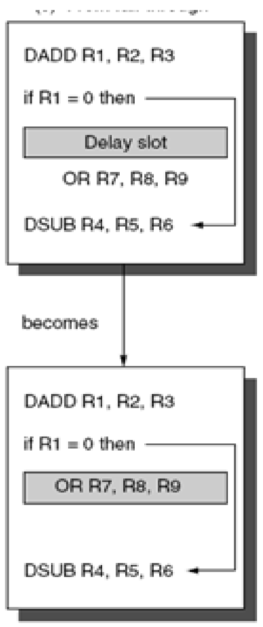
R4=R5+R6;

R1=R2+R3;

}while(R1= =0)

→If the condition is taken the control has to go back to the target address. That means the instruction inside the do-while loop is executed at least once whether branch is taken or not taken. So shifting branch address into the delay slot will execute the instruction at least once before the branch is evaluated to true or false and the number of times when the control goes back to the loop, so many times the instruction inside in the delay slot will be executed.

3. From Fall Through: Consider the example shown below:



→A complex and not an efficient method were in the delay slot is filled with instruction from the not-taken fall through condition. Here the instruction inserted into delay slot is not a useful instruction but even if it is not useful, the program is executed correctly. The work is wasted but doesn’t harm the result.

→In this example, instruction **OR R7,R8,R9** which is inside the branch is assumed to be using an register R7 which is nowhere used in any of the instructions as the source. So executing this instruction even if the branch goes in other ways will not harm the end result.

Conclusion: First method **from before** scheduling method is the best technique and is given the highest priority. When this cannot be applied, the next best is considered as the **from target** scheduling method. When both of this cannot be applied only then the last method **from fall through** scheduling method is chosen when there is no option.

How Is Pipelining Implemented For R, I & J Type Of instructions (MIPS Architecture)

In this section we will know about how pipelining is actually implemented for MIPS architecture, were in we’ll come to know about different Hardware units used in each and every stage and how exactly they will transfer control and data between them through different instructions.

Before actually knowing the exact implementation of pipeline, first let us see how does a sequential MIPS architecture is implemented for processing the R, I & J form of instructions in different stages. Then we will see how the same sequential architecture is converted into pipeline system by imposing small changes to the system. First we will look into what are R, I and J types of instruction very briefly

What are R, I and J types of instruction?

→Based on the type of operations and addressing modes, instructions in MIPS architecture is classified into fixed formats called: R, I and J types were R represents **Register mode**, I represents **Immediate mode** and J represents **Jump instructions.**

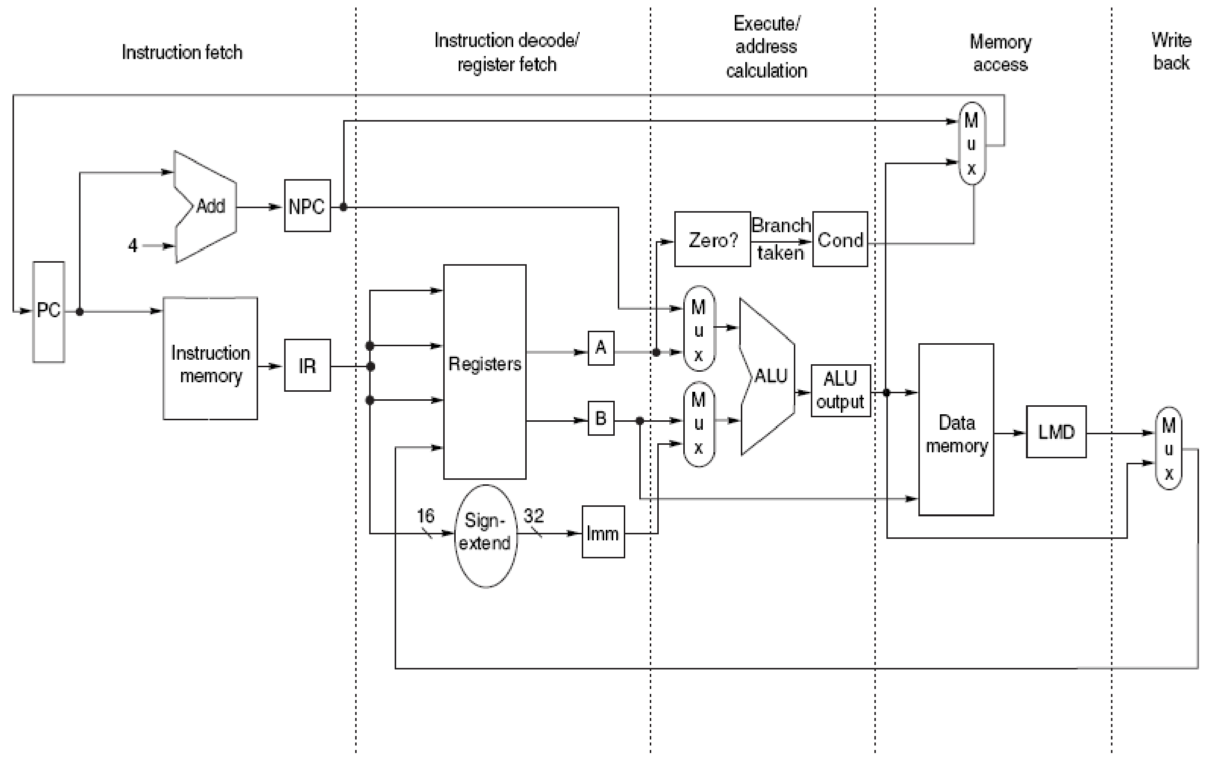
→Since they are classified according to fixed formats, it helps to implement **fixed field decoding technique** were in common fields in any format of the instructions arrive in similar sequence and with same length as shown in the figure:

Type	-31-	format (bits)		-0-		
R	opcode (6)	rd(5)	rs1 (5)	rs2 (5)	shamt (5)	funct (6)
I	opcode (6)	rd (5)	rs1(5)	immediate (16)		
J	opcode (6)	address (26)				

A Simple Implementation of MIPS for R, I and J instructions

→We will see how exactly a simple MIPS architecture is implemented for R, I and J types of instructions in connection to how exactly data and control are processed in each and every stage of MIPS.

→Figure below shows the implementation of typical MIPS data path architecture along with the different hardware units used in each and every stage of the architecture.



Every MIPS instruction can be implemented in at most 5 clock cycles. The 5 clock cycles along with its processing in each clock cycle are as follows:

I. **Instruction Fetch cycle (IF):**

Operation:

1. Send out PC and fetch the instruction from memory into the Instruction Register (IR) **for** execution.

IR← Mem [PC];

2. Next after fetching the instruction from memory, increment the PC value by 4 to point to the next sequential instruction address and save the value in NPC (Next Program Counter)

NPC←PC+4;

II. **Instruction Decode / Register fetch cycle (ID):** In this cycle, instruction decoding and reading of source registers is done in parallel.

Operations:

1. Decode the instructions and access the register file to read the source operands.

2. Source operands can occur in two different forms:

a. ***Both source operands are register values:*** Access both the operands from register file and save the values in temporary register A and register B which is given by:

A← regs[rs];

B← regs[rt];

Were A & B are temporary registers and rs and rt are operand 1 and operand 2 respectively.

b. ***One of the source operand is register value and another operand is immediate value:***

When one of the source operand is an immediate value, immediate value is read from immediate field and stored in a temporary register **imm** and the other operand is read from register file.

imm←immediate field of IR

III. **Execution/ Effective address cycle:**

In this cycle, ALU operates on the operands prepared in the prior cycle. One of the 4 functions is performed in this cycle depending on the type of instruction (R, I, J)

i. ***Memory reference:*** Memory reference constitutes to access of memory for load/store operations. Effective address required for memory access is calculated by ALU unit were in it takes one operand as base register and immediate value as offset adds them and places the result in temporary register ALUOutput.

ALUOutput🡐 A + Imm;

ii. ***Register – Register ALU instruction:*** ALU performs the operation specified by the opcode on source operands fetched from the register file and places the result in temporary register ALUOutput.

ALUOutput 🡐 A func B;

Were A- source operand1, B- source operand2 and func is the ALU operation code.

iii. ***Register – immediate ALU instruction:*** ALU performs the operation specified by the opcode on one value of register **A** and the other one on value in **Imm** register and places the result in temporary register ALUOutput.

ALUOutput 🡐 A op Imm;

iv. ***Branch instruction:*** If the instruction fetched is a branch instruction, ALU unit performs two operations:

1. First it calculates the Branch Target Address by taking **NPC** value as one operand and displacement value from **Imm** register as another operand and stores the result in ALUOutput

ALUOutput 🡐 NPC + Imm;

2. Next it will update the value of temporary register **Cond** either to 1 or 0 based on the condition evaluation. If condition is evaluated to true, it will update the value of **Cond** to 1 or else it will update the value to 0.

Cond 🡐 (A= =0)

[Here for the simplicity, only one condition branch equals to zero is considered.]

IV. **Memory access/ Branch completion cycle:** This stage uses the result computed by ALU in the previous cycle which is stored in **ALUOutput** register**.** The stage operates on two types of instructions: memory reference instructions (load/store) and branch instructions.

i. ***Memory Reference:*** In this cycle, the effective address calculated in the previous cycle is used to access the address of the memory unit and also updates the value of **PC** to value **NPC** value. When the instruction is memory reference, it can be either a load or store instruction:

PC 🡐 NPC;

a. Load instruction: If the operation is load operation, the data value is read from the memory location pointed by the effective address value and is stored in a temporary register **LMD.**

LMD ← Mem[ALUOutput];

b. Store instruction: If the operation is store operation, the data from the register **B** is written into the memory location pointed by the effective address value.

Mem[ALUOutput] ← B;

ii. **Branch instruction:** If the instruction is branch instruction, it will check the status of **Cond** register and if the value of Cond register is 1, it will update the value of PC to Branch Target address computed in the previous cycle by ALU.

if (Cond) PC ← ALUOutput;

V. Write – back cycle: The cycle is used to write the result into register file. This stage is used only by ALU instructions and load instructions. Operations performed based on instruction types are:

1. **ALU instruction:** Computed register – register ALU operation is written into destination register specified in the instruction.

   Regs[rd] ← ALUOutput;

2. **Load instruction:** If the instruction is load instruction, value read from the memory location is written into the destination register specified in the instruction.

   Regs[rd] ← LMD;

Pipeline Implementation for MIPS data path (5 stage pipelined MIPS data path)

By making small changes to the simple MIPS data path, five stage MIPS data path can be implemented. Some of the changes implemented to simple MIPS data path are:

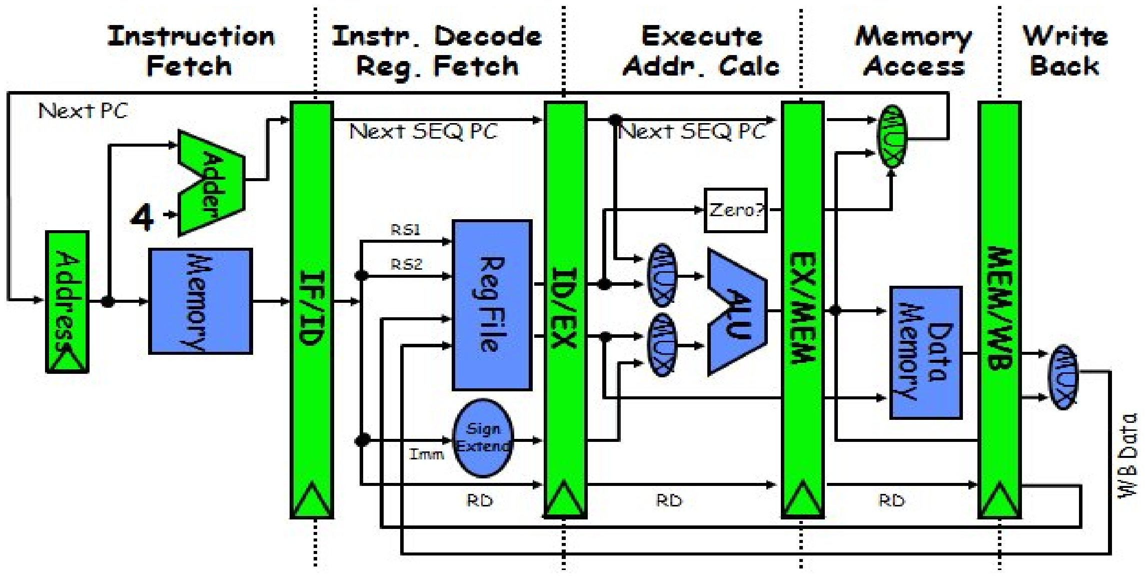
1. Each cycle of a simple MIPS pipeline is converted into a pipeline stage were in each and every stage is completed in only one clock cycle.

2. Stages of pipeline are overlapped such that in every clock cycle of the pipeline there will be a new instruction fetched from the memory unit and every clock cycle has a different units operating.

3. Computation done in each and every stage lasts only for one clock cycle and in the next cycle it will be acquired by the other instruction. Hence it is required to preserve the result computed in every stage. So between each stage, a register called **pipeline registers or latches** is used.

4. All the temporary registers used in simple MIPS data path are embedded within the pipeline registers and all data and control information into different stages are transferred via pipeline registers.

5. Instructions fetched remains only for one clock cycle and in each cycle the stage will be acquired by a new instruction. So while moving into next stage even the copy of instruction is carried and placed in the subsequent pipeline registers.
Complete MIPS pipeline implementation is shown below with changes implemented:



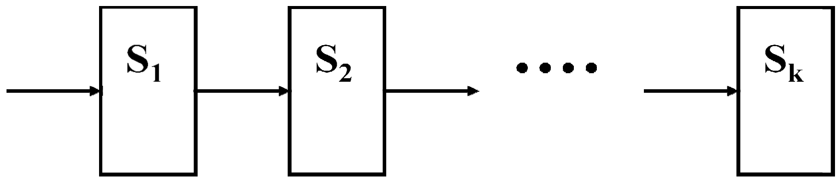
UNIT II – Chapter 2

LINEAR AND NON LINEAR PIPELINE PROCESSORS

(Reference: KAI HWANG 6th Chapter)

Linear Pipeline Processor

→It is a cascade of processing stages which are linearly connected to perform a fixed function over a stream of data flowing from one end to another end. A typical structure of linear pipeline is shown below which is also called as **static pipeline**



→Pipeline is constructed of K number of stages. External inputs are fed into the pipeline at a very first stage (S1) and processed result are passed onto subsequent stages and final result emerges from the last stage (Sk)

→Depending on the control of data flow among the subsequent stages, linear pipeline is classified into two types:

1. Asynchronous linear pipeline model.

2. Synchronous linear pipeline model

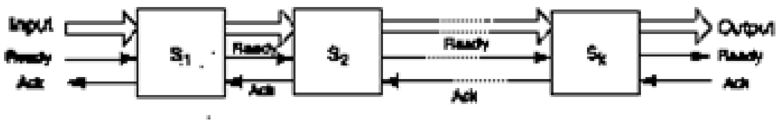
.

Asynchronous model

→In the Asynchronous model, dataflow between adjacent stages of pipeline is controlled by a **Handshaking protocol.**

→When a particular stage S_i is ready to send data into $S_{i+1,}$ S_i will send **Ready** signal into $S_{i+1.}$ On receiving ready signal, if S_{i+1} is ready to receive the data it will send back **Ack** back to S_i with which stage S_i sends data into S_{i+1}.

→Representation of linear asynchronous pipeline is shown below:



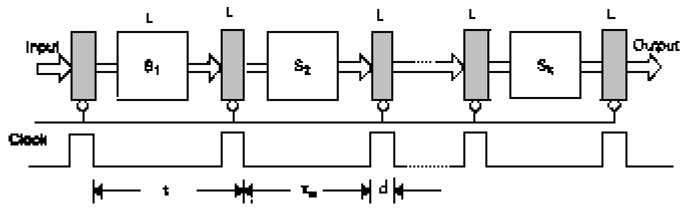
→The pipeline system is used in multicomputer systems to implement **Message passing** function.

Synchronous model

→In this model, dataflow between adjacent stages in pipeline is controlled by centralized **clock signal.** I.e. all units in a pipeline respond simultaneously upon the arrival of a clock signal.

→To facilitate simultaneous dataflow between stages, clocked latches are used to interface between the stages. Latches are made up of **master – slave** flip-flops which can isolate read operation from write operation.

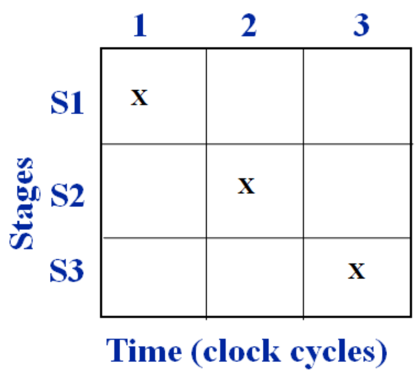
→Typical representation of this model is shown below:



Reservation Table for Linear Synchronous Pipeline

→Utilization pattern for a particular task with respect to successive stages is specified by a two dimensional table with stages represented along the rows and clock cycles represented along the columns.

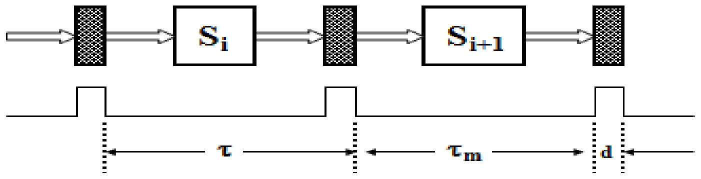
→Reservation table is also called as **Space- time diagram**. Reservation table for a synchronous linear pipeline follows a diagonal path for the task processing. A typical three stage linear synchronous pipeline reservation table is shown below:



Stages	1	2	3
S1	X		
S2		X	
S3			X

Time (clock cycles)

Determining cycle time for pipeline



→Cycle time for pipeline is chosen such that it is equal to the time taken for processing of a longest stage.

→Let τ cycle time for a pipeline which is given by the expression:

$$\tau = \max \{\tau_m\} + d$$

were τ_m time taken for processing each stage, d is the latch delay used between intermediate registers and **max** $\{\tau_m\}$ is the time taken for processing the longest stage.

Pipeline frequency (clock rate) is given by the expression

$$f = 1 / \tau$$

Calculating Speedup, efficiency and throughput

Speedup: Equation for speed up is given by

$$\text{speedup} = \frac{\text{cpu time unpipelined}}{\text{cpu time pipelined}}$$

***k-stage* pipeline processes *n* tasks in k + (n-1) clock cycles:**

Total time to process *n* tasks : $T_k = [\, k + (n-1)]\, \tau$

For the non-pipelined processor $T1 = n\, k\, \tau$

Speedup factor

$$S_k = \frac{T_1}{T_k} = \frac{n\, k\, \tau}{[\, k + (n-1)]\, \tau} = \frac{n\, k}{k + (n-1)}$$

Efficiency and Throughput of a pipeline

Calculating efficiency of a pipeline:

Efficiency of a pipeline is given by the ratio of speed to the number of stages of pipeline given by the expression:

$$E_k = \frac{S_k}{k} = \frac{n}{k + (n-1)}$$

Calculating pipeline throughput:

→Pipeline throughput is defined as the number of tasks performed by a pipeline in a unit time which is given by the ratio of number of tasks in a pipeline to CPU time for performing n tasks

→It is also defined as the product of efficiency of pipeline and frequency of a pipeline, which is given by the expression:

$$H_k = \frac{n}{[k + (n-1)]\,\tau} = \frac{nf}{k + (n-1)}$$

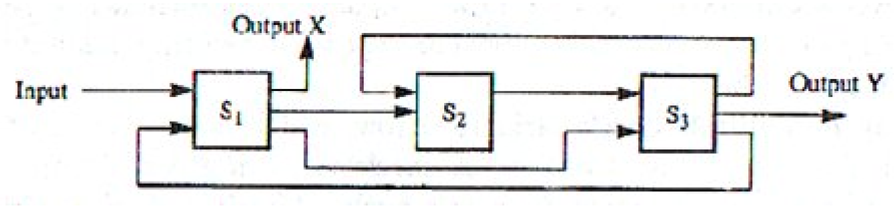
Non Linear Pipeline

→A non linear pipeline is a **dynamic pipeline** that can perform variable function at different time units.

→Apart from having a data stream flowing from one end to another end, it even allows **feed forward** and **feed backward** connections.

→In linear pipeline the output is achieved only in the last stage. But the non linear pipeline can produce output in any of the intermediate stage also.

→One more difference in linear and non linear pipeline is that, a linear pipeline performs only one function but a non linear pipeline can perform any number of functions. A high level representation for a three stage non linear pipeline is given as shown below:



Reservation Table for Non linear pipeline

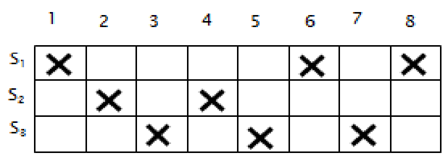
→As mentioned in the earlier section a reservation table for a non linear pipeline is also a two dimensional table for plotting the utilization pattern for a particular task with respect to successive stages with stages represented along the rows and clock cycles represented along the columns.

→But there is a small difference in the plotting pattern of these two tables, were in for a linear pipeline plotting for a particular function follows a diagonal pattern which means that each and every stage for a particular task is visited only once in a sequential pattern

→For a linear pipeline, there is only one reservation table as it is having only one fixed function for the operation. But for a non linear pipeline, it can perform n number of functions and for n different functions there will be a n different reservation table.

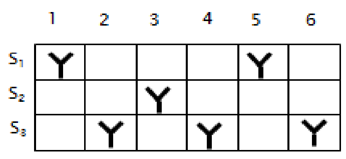
→Number of columns in a reservation table indicates the clock cycles acquired for performing specific task.

→As an example, Reservation table for function X and Y for a pipeline is shown below



	1	2	3	4	5	6	7	8
S_1	X					X		X
S_2		X		X				
S_3			X		X		X	

Reservation table for function X



	1	2	3	4	5	6
S_1	Y				Y	
S_2			Y			
S_3		Y		Y		Y

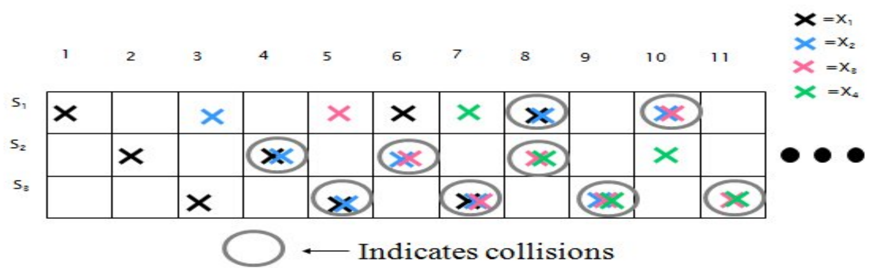
Reservation table for function Y

Latency Analysis: Some of the terms under non linear pipeline are

***Latency**:* number of time units (clock cycles) between two initiations of a pipeline.

***Collision**:* an attempt by two or more initiations to use the same pipeline stage at the same time

- Consider the same pipeline, lets see initiations in the pipeline for function X, with latency=2



Forbidden latency: These latencies are the one that causes collision in the initiation.

Permissible latency: These latencies are the one that does not cause collision in the initiation.

In the previous example 2 and 5 were forbidden latencies.

Collision-free Scheduling:

It is the technique of selecting the minimum latency value for the initiation of tasks such that the latency value does not lead to collision in the reservation table.

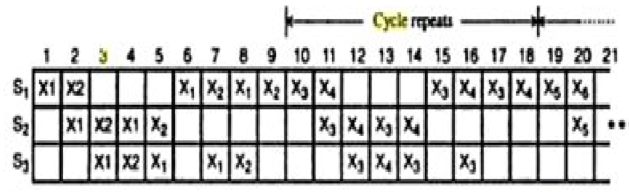
Let us first look at some basic terms:

Latency sequence*:* a latency sequence is a sequence of permissible latencies between successive tasks initiations.

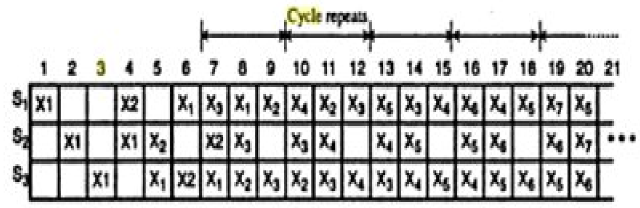
Latency cycle: It is a latency sequence which repeats itself.

Constant cycle*:* This contains a single latency value in the latency cycle.

A sample reservation table for latency cycle with value (1, 8) and (3) is shown below:



(a) Latency cycle (1,8) = 1,8,1,8,1,8,..., with an average latency of 4.5



(b) Latency cycle (3) = 3,3,3,3,..., with an average latency of 3

(**For problems of both the chapters, please do refer the problems solved in class)**