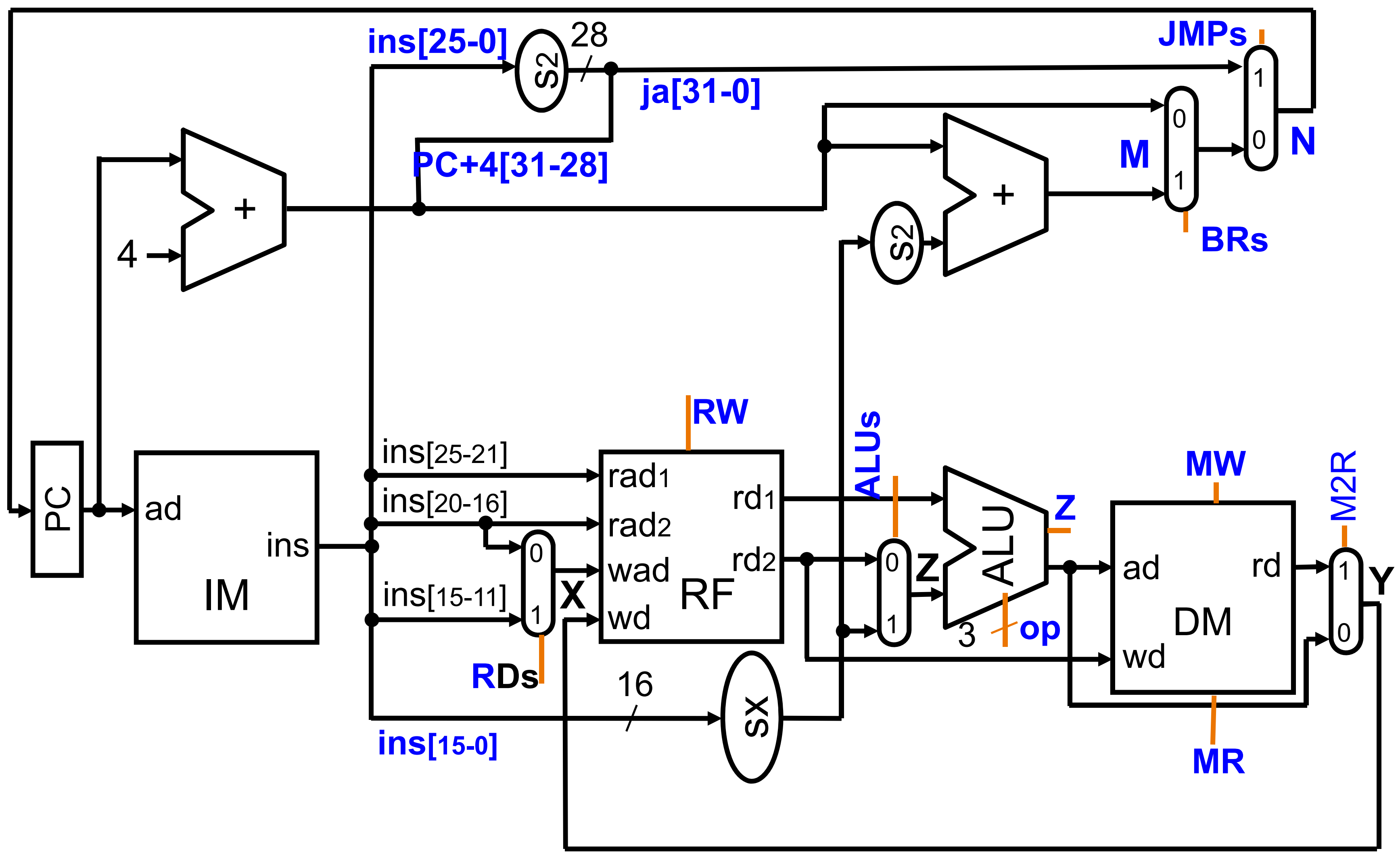
COMPUTER ORGANIZATION AND ARCHITECTURE (IT 2202)

Pipelining-II

MIPS Pipeline Implementation

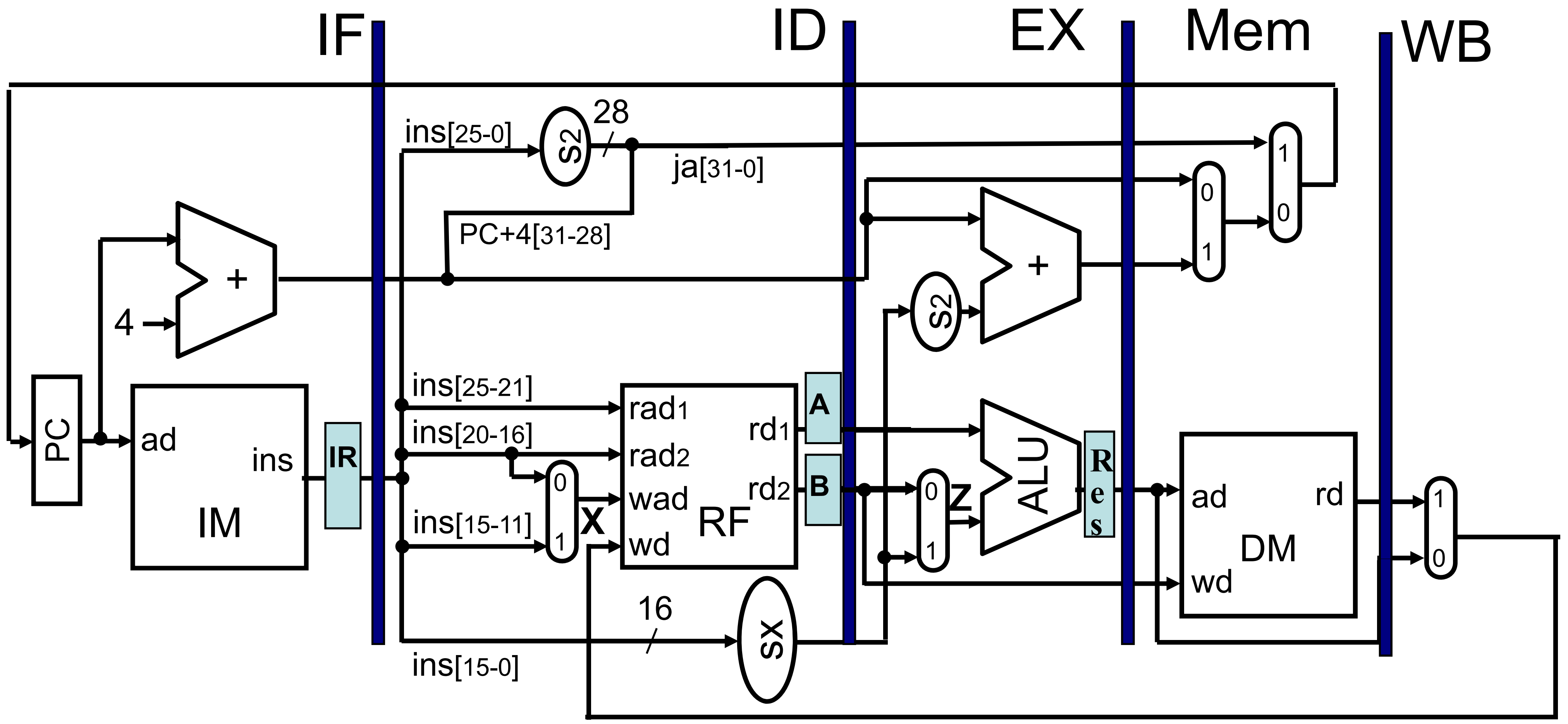
- We have already presented the implementation of non-pipelined (Single Cycle) data path for the MIPS32 instruction set architecture.
- MIPS32 architecture provides simplicity of the instruction set, the regularity, the simple instruction encoding and the required hardware is very simple.
- MIPS32 architecture needs very simple hardware in the data path with very regular interconnections.
- Now, we shall discuss the implementation of the pipelined version of the MIPS32 data path.

MIPS Pipeline Implementation



Single Cycle MIPS Architecture

MIPS Pipeline Implementation

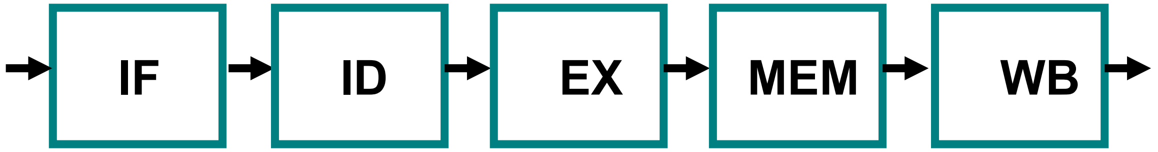


- **In MIPS32 non-pipelined datapath, there are 5 steps to execute an instruction: instruction fetch, instruction decode, execute, memory operation and write back.**

MIPS Pipeline Implementation

- We will discuss the implementation of the MIPS pipeline data path.
- In Pipeline datapath, a new instruction will be initiated in every clock cycle.
- In MIPS pipeline , it has 5 stages: IF (Instruction Fetch), ID (Instruction Decode), MEM (Memory), WB (Write Back).
- Each stage must its operation within one clock cycle.
- Since execution of several instructions are overlapped, there may arise of conflict during the execution.
- The conflict in execution will be avoided as MIPS32 is very simple architecture.

MIPS Pipeline Implementation



5 stages Pipeline

- In pipeline, every instruction is divided into 5 subset that can be implemented at most 5 clock cycles. Each of the clock cycles becomes a *pipe stage*— a cycle in the pipeline. The 5 clock cycles are given as follows.

Instruction fetch cycle (IF): PC points to memory and the current instruction is fetched from memory and PC is incremented by adding 4 for pointing the next instruction.

Instruction decode/register fetch cycle (ID): The instruction is decoded and the registers corresponding to register source specifiers are read from the register file.

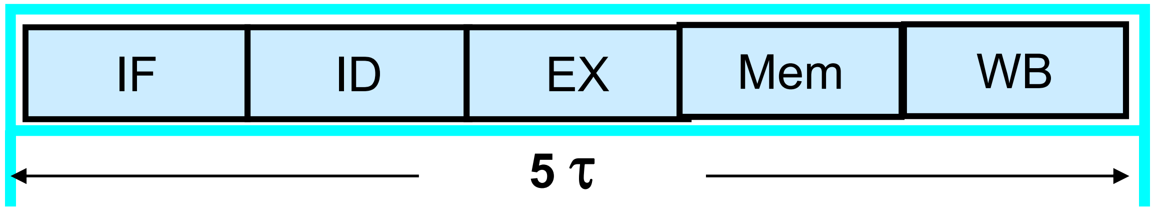
Execution/effective address calculation cycle (EX): ALU operates on the operands prepared in the prior cycle, performing one of three functions depending on the instruction type.

MIPS Pipeline Implementation

- R-Type (Register-Register ALU instruction): ALU performs operation specified by ALU opcode on the values read from the register file.
- Memory reference: ALU adds the base register and offset to form the effective address. ALU performs operation specified by ALU opcode on first value read from register file and sign-extended immediate.
- In a load-store architecture, effective address and execution cycles can be combined into a single clock cycle, since no instruction needs to simultaneously calculate a data address and perform an operation on the data.

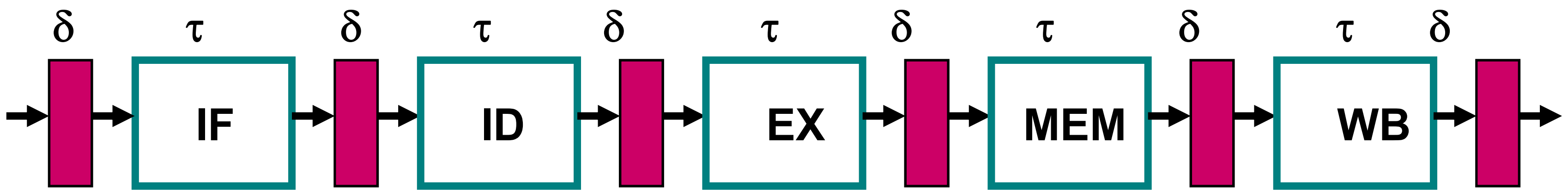
- Memory access (MEM): If instruction is a load, memory does a read using effective address computed in previous cycle. If it is a store, then memory writes data from second register read from register file using effective address.
- Write-back cycle (WB): R-type instruction or lw instruction: Write the result into register file, whether it comes from the memory system (for a load instruction) or from the ALU (for R-type instruction).

MIPS Pipeline Implementation



Non-pipeline Datapath

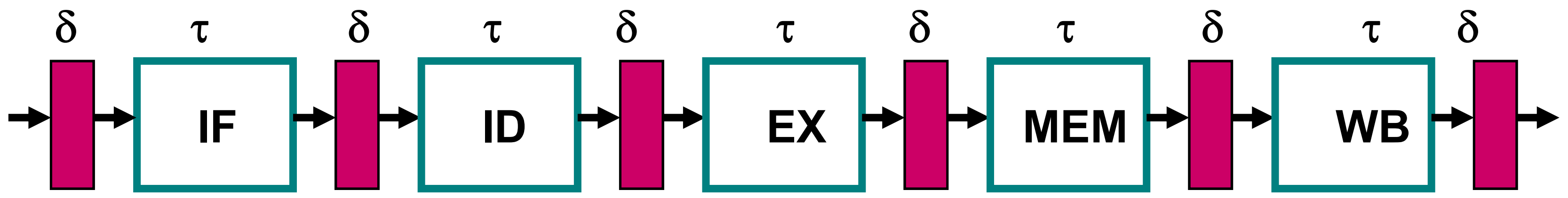
- Time of Execution of a single Instruction in Non-pipeline Datapath= 5τ
- Time of Execution for n Instructions in Non-pipeline Datapath (T1)= 5 .n.τ
- Actually, Pipeline will be implemented with 5 stages: IF (Instruction Fetch), ID (Instruction Decode), MEM (Memory), WB (Write Back).
- In between stages, at input and output, latches will be inserted.



We use a latch or a register between successive stages, because one stage finishes some calculations or operation and the output of the this stage will move to the next stage provided next stage finishes current job, otherwise, it will waits may be, still the next stage finishes. So, that value is temporarily kept in the latch, so that the next stage whenever it is free can take it from the latch.

MIPS Pipeline Implementation

Speedup of Pipeline Datapath



T_2 =Time required to execute *n* Instruction = $\{5 (\tau+ \delta) +(n-1) (\tau+ \delta)\}$ if each stages takes $(\tau+ \delta)$ unit time.

$T_2 = [\{5 +(n-1)\}. (\tau+ \delta)] = [(4 +n). (\tau+ \delta)]$

$=(4+n). \tau$, If $\tau >> \delta$,

Ideal Speedup= T1/T2 = $\{5. \tau.n/\{(4+n). \tau\}$ for large n.

$= 5n/4+n$

$= 5/(4/n+1) = 5/(0+1)$ as $n \rightarrow \propto$

$\cong 5$

Due to the different pipeline conflicts (hazards), it is very difficult to achieve ideal speedup.

MIPS Pipeline Implementation

Example: Consider the 5-stage MIPS32 Pipeline, with the following features

- Pipeline clock rate of 1 GHz (i.e., 1 ns clock cycle time)
- For non-pipeline implementation, ALU operations and branches take 4 cycles, while memory operation take 5 cycles.
- Relative frequencies of ALU operations, branches, and memory are 50%, 15%, and 35%, respectively.
- In the pipelined implementation, due to clock skew and setup time, the clock time increases by 0.25 ns.
- Calculate the estimated speedup of the of pipeline?

Solution:

a) For non=pipelined Datapath:

- Average instruction execution time = Clock cycle time X Average CPI

  = 1 ns X (0.5x4+0.15 x 4+ 0.35 X 5) =4.35 ns

b) For pipelined Datapath:

- Clock Cycle time = 1+0.25 =1.25 ns
- In the steady state, one instruction will get executed every clock cycle.
- Speedup=4.35/1.25=3.48

MIPS Pipeline Implementation (Working of Pipeline)

Here, the movement of instructions from one stage to other under different the time steps or clock cycles. Assume, the instructions are coming one by one.

In clock cycle 1, first instruction *i* enters the IF stage; instruction i, is fetched. After it goes to the ID stage in cycle 2. While it is being decoded, next instruction can be fetched. After this is done, in 3 cycle, instruction ‘i’ will go to EX stage, instruction i+1 moves to ID stage, and instruction (i+2) (3rd) can go into IF. In 4th cycle, instruction i moves to MEM stage, (i+1) moves to EX stage, (i+2) moves to ID stage and instruction (i+3) enters in IF stage. In 5th cycle, instruction i moves to WB stage, (i+1) moves to MEM stage, (i+2) moves to EX stage, (i+3) moves to ID stage and instruction (i+4) enters into IF stage.

After 5 cycle instruction ‘i’ will finish, after cycle 6, instruction (i+1) will finish time, after cycle 7, (i+2) will finish, after cycle 8, (i+3) will finish and after cycle 9, (1+4) will finish . Now, the initial delay, pipe gets filled up. In the ideal pipeline case, one instruction will be completed every clock cycle.

Instr. i+1 finishes

Instr. i+2 finishes

	Clock Cycles or time steps								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction i	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

Instr. i+4 finishes

Instr. i finishes

Instr. i+3 finishes

MIPS Pipeline Implementation (Working of Pipeline)

	Clock Cycles or time steps								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	$\boxed{\text{MEM}}$	$\boxed{\text{WB}}$				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				$\boxed{\text{IF}}$	$\boxed{\text{ID}}$	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

The various kinds of conflicts occur in the pipe.

Case 1: There is a conflict between IF & MEM indicated in 4 cycle. Suppose, the first instruction was a load or store kind of instructions, load and store means during the MEM stage it will be accessing memory. But at the same time, instruction (i+3) is also trying to read from memory for an instruction fetch. So, both instructions *i* and (*i+3*) are trying to access memory at the same time.

Possible Solution: Use of separate instruction and data caches. IF will use the instruction cache and MEM will the data cache. So, the conflict may be removed or avoided.

MIPS Pipeline Implementation (Working of Pipeline)

Instruction number	Clock Cycles or time steps								
	1	2	3	4	5	6	7	8	9
Instruction *i*	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

Case 2:

In cycle 5, first instruction, i.e., instruction i (suppose, a register type instruction, say ADD). After execution of ADD instruction, the result will be written into the register in the WB phase. Again, in the *ID* phase, the register operands are pre-fetched and here, in cycle 5, register will be read in *ID* Phase in 5 cycle. There is a conflict in cycle 5. The instruction (*i*+3) is trying to read from the register bank and instruction '*i*' is trying to write into the register.

***Possible Solution*: Reading from Register and Writing into the register in the same cycle.**

We can read from the registers also you can write into the register in the same clock cycle. Basically, write operation will be performed first half cycle and then read operation will be performed in next half cycle of the same cycle.

MIPS Pipeline Implementation (Summary)

- **In the non-pipelined version of the MIPS, the execution time of a instruction is equal to the combined delay of the 5 stages which is 5T, but in a pipeline version in the ideal case, one instruction gets executed after every time T, assuming that all stage delays are equal, and equal to T, and we are neglecting the delays of the latches.**

- **But practically, it is not simple to execute each instruction at every T time, because various conflicts commonly known as hazards can arise between instructions. We cannot achieve the ideal performance, because in the ideal case the speedup can approach 5, but because of hazards it can be less than 5. We will discuss various techniques to improve the performance to some extent.**
- **For the non-pipelined version there will be 2 memory accesses maximum in 5 cycles.**

- **For the pipeline version, In the worst case in every cycle there can be potentially 2 memory accesses. In non-pipeline version, there are 2 memory accesses in 5 cycles, whereas there can be maximum 2 memory accesses in one cycle in the pipeline version. Therefore, in pipeline operation, the memory bandwidth requirement is 5 times.**

MIPS Pipeline Implementation (Working of Pipeline)

- To support overlapped execution in the pipeline stages, the peak memory bandwidth should be increased by 5 times compared to that required by non-pipelined version.

Solution: In the L1 cache, we use separate instruction and data caches. Actually most of the time, due to the locality of reference, the data reference by the CPU is found in the caches. So, there will be no conflict. One instruction can fetch it from Instruction cache, other instruction can read or write from the Data cache. This is a simple solution. However, when there is a cache miss, there will be delay. This is why, we need separate instruction and data caches.

Register Access Issue.

In pipeline version, Some instruction can read a register during ID stage and some other instruction can write into a register during WB. In the ID stage, for some instructions, we are pre-etching both the register operands, source 1 and source 2. So, ID requires 2 register reads and we require at most one register write in WB stage. So, in a pipeline, in every cycle, datapath should support 2 register reads and one register write.

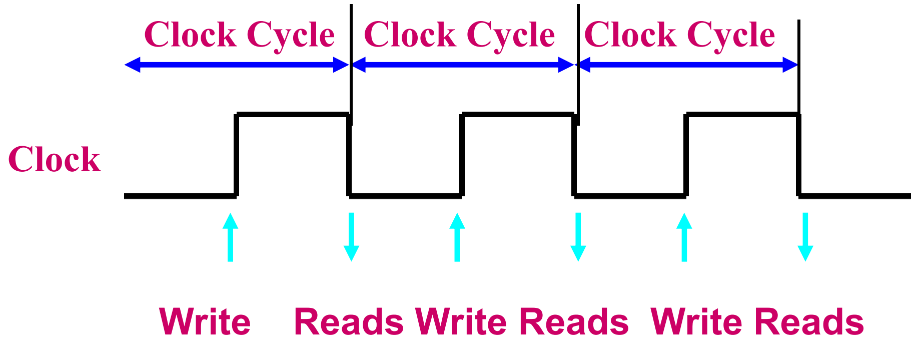
Solution: If the register bank has 2 separate read ports, then reading 2 registers concurrently is not problem. However, the conflict will occur, if someone is trying to read register (say r1) and someone other is trying to write into register r1 at the same time. Simultaneous reading and writing will result in clash if same register is used.

MIPS Pipeline Implementation

Solution: **In the MIPS pipeline, we divide the cycle time into two halves as shown. Each clock cycle has two clock edges: leading (rising) edge indicated by the up-arrow and falling edge indicated by down arrow as shown in Fig.**

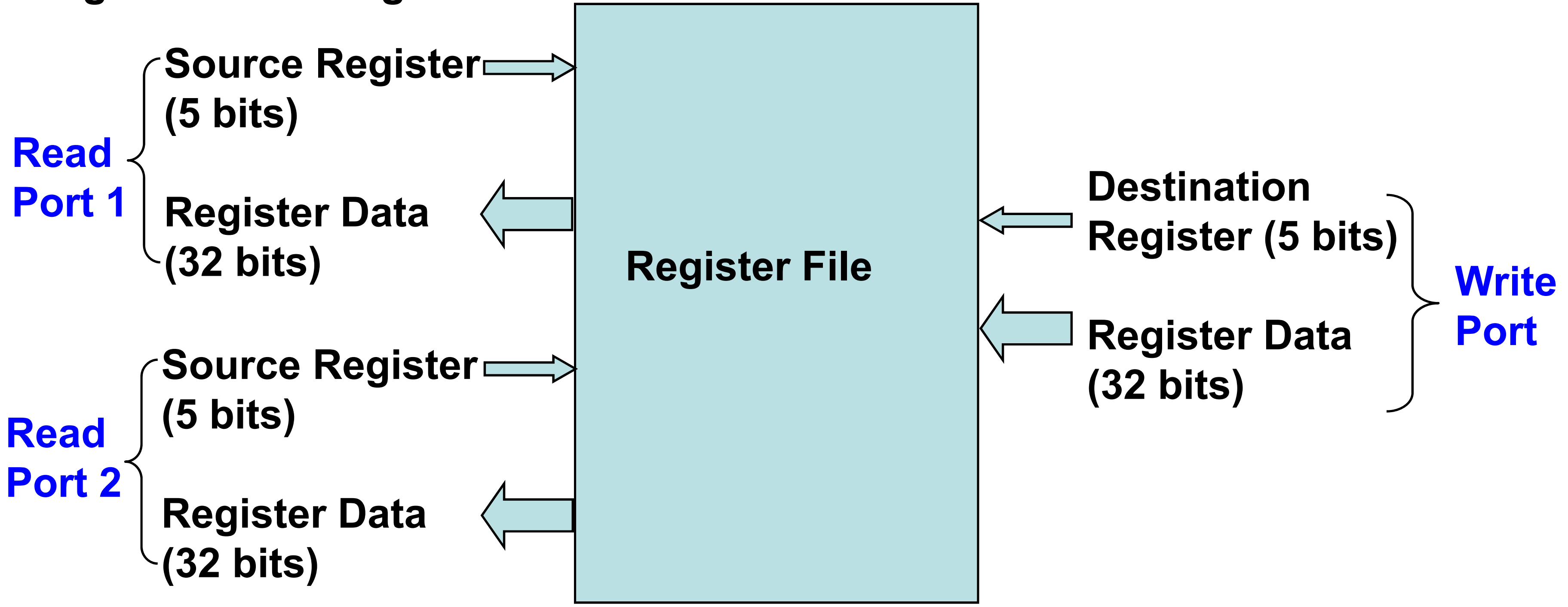
In the leading edge, all the register writes will be done, and in falling edge, all the register reads will be performed because register access is fast compared to clock cycle time as it is long. The clock cycle time is dividing into 2 halves; in the first half we are doing the write, in the second half we are doing the reads.

In any case, someone is writing into r1 and some other instruction is reading from r1, then first, write will happen then the read of new data will be performed.



MIPS Pipeline Implementation (Working of Pipeline)

Register Bank (2 Read Ports and one Write port and 32 number 32 bit Registers): Each read port consists of one 5 bit source address port and one 32 bit Data Port and the write port has one 5 bit destination address port and one 32 bit data port. 5 bit register address comes from the Instruction through the 5 bit Source Register Port to identify the 32 bit Data register in the Register File from where 32 bit operand will be accessed. Through two 32 bit Register ports two 32 bit operands will be accessed just like memory reading. Similarly, 5 bit register address comes from the Instruction through the 5 bit destination Register Port to identify the 32 bit Data register in the Register File in which 32 bit data will be written.



As it has already mentioned that “write” will happen in the first half and the 2 reads will happen during the second half of the clock. Using this simple convention and the modification to the register bank, this conflict can be avoided.

MIPS Pipeline Implementation (Working of Pipeline)

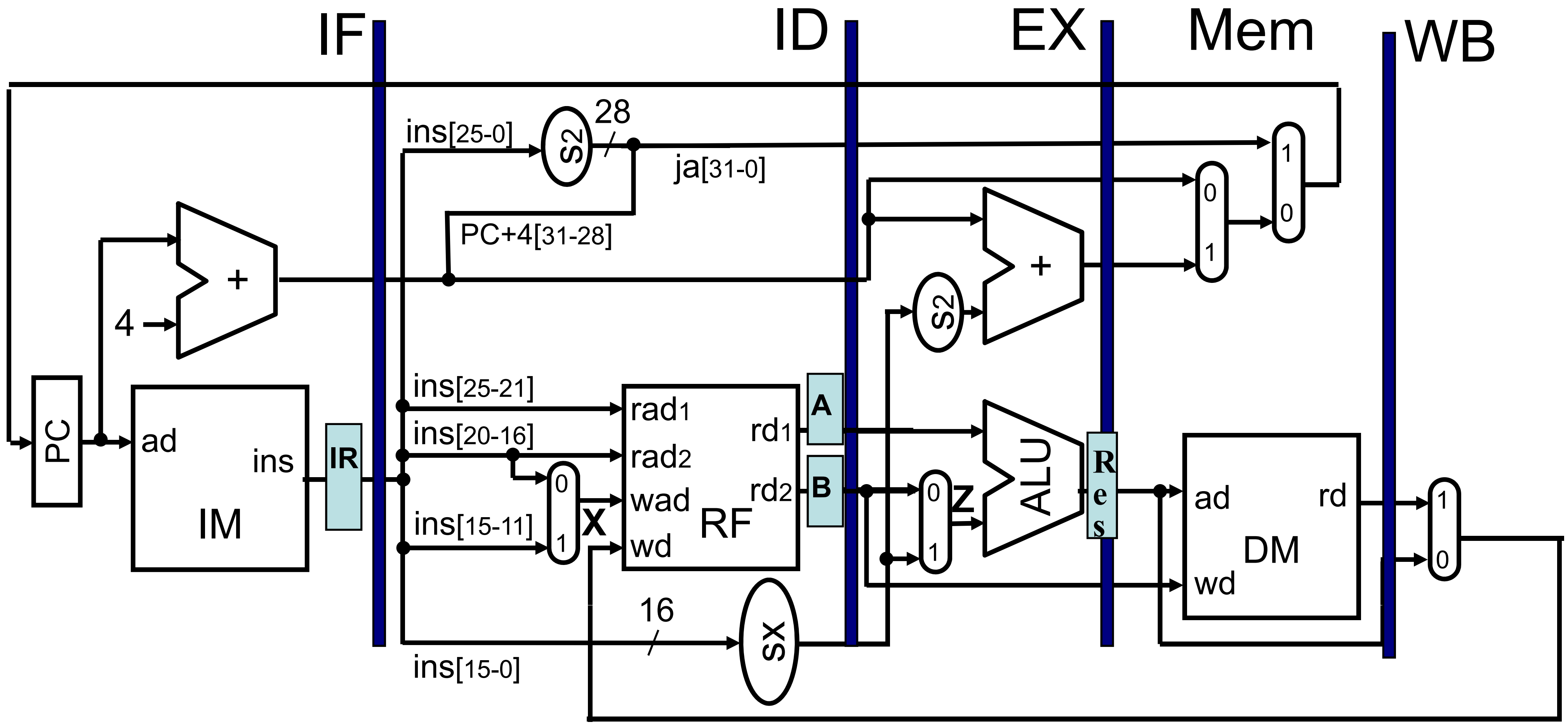
Issue on PC Operation: In the non-pipelined MIPS, (PC+4) is not implementing in the IF stage. (PC + 4) result is transferring into PC if it was not a branch.

In a pipeline, we need to fetch an instruction in every clock. PC should be ready with the address of the next instruction at the end of every clock. So, at the end of every clock, the PC will be incremented in the IF stage.

PC updating has to be done during the IF stage, otherwise, we cannot fetch the next instruction in the next clock. In the non-pipelined version this was done in MEM.

MIPS Pipeline Implementation

Non-Pipeline MIPS Datapath: This Figure shows the operation of non-pipeline MIPS Datapath. In the non-pipelined MIPS, (PC+4) is not implementing in the IF stage. (PC + 4) result is transferring into PC in MEM stage. This non-pipeline Datapath will be modified for the operation as the Pipeline Datapath.



MIPS Pipeline Implementation

The pipeline datapath has five stages: IF, ID MEM, EX and WB.

Some latches along with some registers are required in between the stages. The latch stage placed between IF and ID is called as IF/ID. The latch stage between ID and EX is called as ID/EX. The latch stage between EX and MEM is called as EX/MEM. The latch stage between MEM and WB is called as MEM/WB.

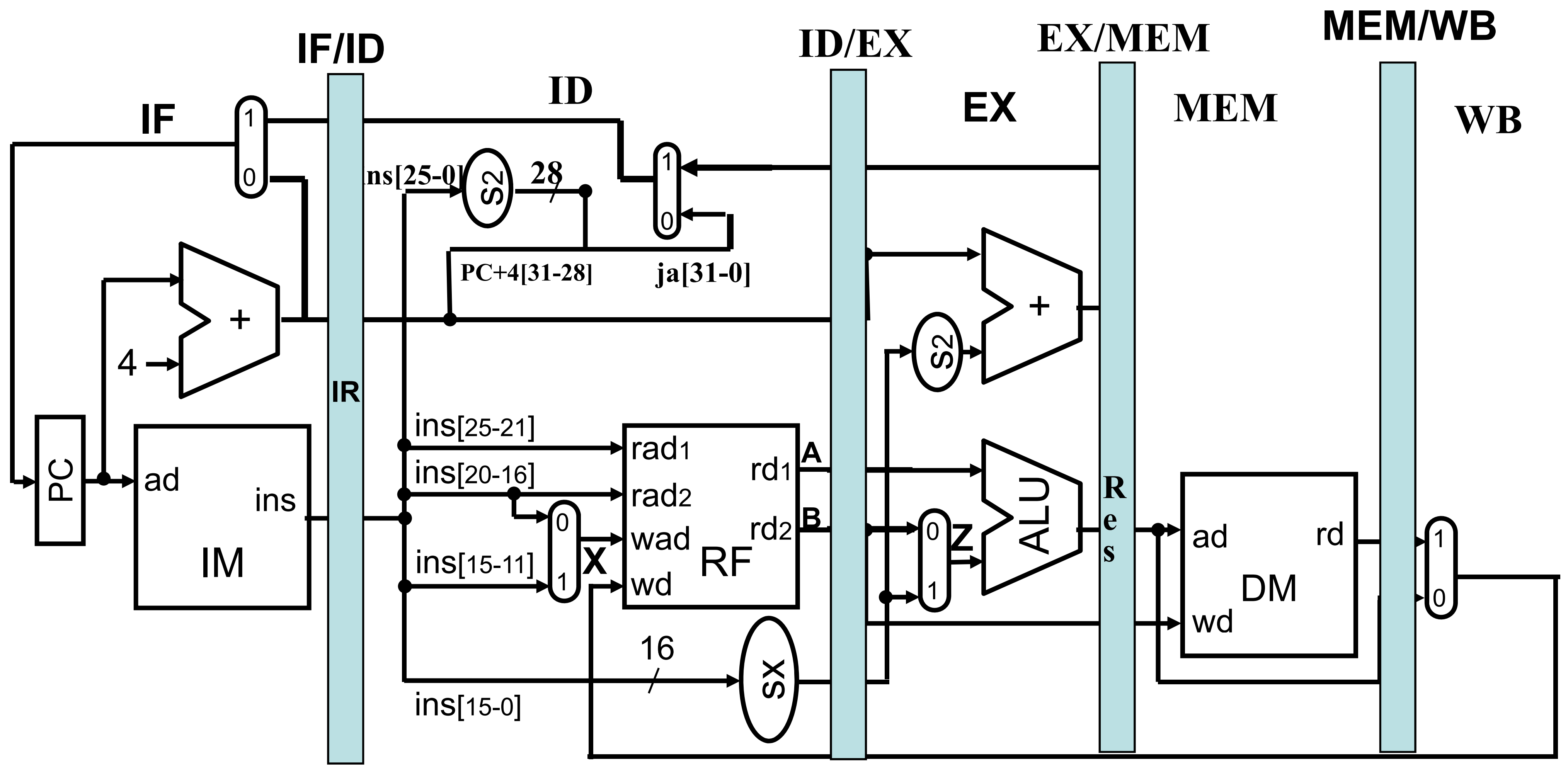
In the non-pipelined in the IF stage, an instruction memory is being accessed by some address from PC; and some data is read and is loaded into instruction register (IR).

In the pipeline, we need not use separate registers like IR in IF/ID stage. IR will be included IF/ID latch stage (IE/ID.IR). Actually, we shall not use the registers separately between these stages; rather we are integrating these registers along with these latches.

We are integrating register file's two output registers (A and B) within ID/EX latch stage (ID/EX.A.B), this means that the registers (A and B) are implemented as part of the ID/EX latch. Similarly, ALU output register (Res) is integrated with the EX/MEM (EX/MEM.Res).

MIPS Pipeline Implementation (Working of Pipeline)

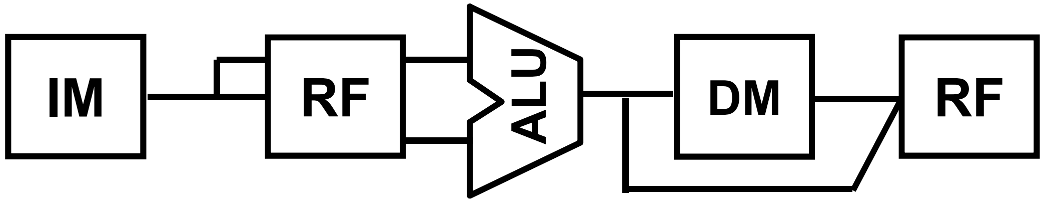
Pipeline datapath has been implemented in the Fig.



Instruction Pipeline

Example: ADD R1, R2, R3

Stages



Actions

IF	ID	EX	MEM	WB

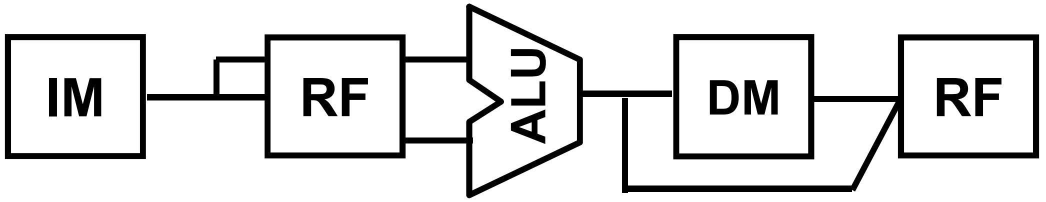
- **IF stage fetches this instruction from IM.**
- **ID stage decodes the instruction and R2 and R3 are read from RF stage.**
- **In ALU, Addition of R2 and R3 is performed in execute stage**
- **Output of (R2+R3) is available at the ALU output which will pass to the multiplexer inserted after Data memory.**
- **In 5th stage, the data will be written back into register R1**

Instruction Pipeline

Design of reservation table for this instruction

- ADD R1,R2,R3

Stages



	t0	t1	t2	t3	t4
IF	X				
ID		X			X
EX			X		
MEM					
WB					X

- At time step T0, IF stage is used.
- At T1, instruction will make use of ID stage is used
- During T2, EX stage is used.
- DuringT3, Writeback (WB) cannot be done because the data is available at the MUX after DM. Memory operation is not needed. So during T3, no stage is used.
- DuringT4, WB stage is used.

- During T4, ID and WB stages are used together because data will be written back to RF (R1) taking address of R1 from ID stage.

Assignments

Problem 1: Draw the 5 stage non-pipeline MIPS32 Datapath.

Problem 2: What are the different stages of MIPS32 Pipeline Datapath? Explain the operation of each stage?

Problem 3: Derive the Speedup expression for MIPS32 Pipeline Datapath.

Problem 4: Consider the 5-stage MIPS32 Pipeline, with the following features

- **Pipeline clock rate of 1 GHz (i.e., 1 ns clock cycle time)**
- **For non-pipeline implementation, ALU operations and branches take 4 cycles, while memory operation take 5 cycles.**
- **Relative frequencies of ALU operations, branches, and memory are 50%, 15%, and 35%, respectively.**
- **In the pipelined implementation, due to clock skew and setup time, the clock time increases by 0.2 ns.**
- **Calculate the estimated speedup of the of pipeline?**

Assignments

Problem 5: How do the 5 stages work in MIPS32 Pipeline Datapath?

Problem 6: How do you solve the memory related conflict in 5 stages work in MIPS32 Pipeline Datapath?

Problem 7: How do you solve the Register file related conflict in 5 stages work in MIPS32 Pipeline Datapath?

Problem 8: Draw and explain the abstract model of the 5 stages MIPS32 Pipeline Datapath?

Problem 9: Derive the reservation table for the instruction (ADD R1,R2,R3)

Pipelining

MIPS Pipeline Hazard

- Implemented the ideal pipeline structure for the MIPS32 processor.
- An instruction pipeline in the ideal sense should complete the execution of one instruction every clock cycle. But, in practical cases, some instructions can not be executed in single clock cycle.
- Hazards are circumstances that prevent the next instruction in the instruction stream from executing during the designated clock cycle. The pipeline can not be operated at its maximum possible speed. These kinds of conflicts or hazards can impact or degrade the performance on a pipeline to a great extent.
- Actually, several instructions have entered the pipe already and several instructions are in various stages of execution and there can be some dependencies between the instructions. They may be trying to access some common resources because of which there can be a conflict. For branch instructions we may have to wait till we know the outcome of the branch and the target address. There are several such instances that may prevent a pipeline to work with its maximum possible capability or speed.

MIPS Pipeline Hazard

Three major types: Broadly speaking hazards can be classified into 3 types;

Structural hazard: arises due to resource conflicts if two instructions require same resource in the same cycle
Example: In memory access, IF and MEM, two instructions may be trying to access instructions and data leading to structural hazard. If there was a single memory module then one of the instructions will have to wait, but because we have use separate instruction and data caches they can proceed together.

Data hazards: Data dependencies - one instruction needs data which is yet to be produced by another instruction. It arises due to data dependencies between instructions.
Because of data dependencies between instructions, say one instruction is producing a result that the next instruction is using, data hazards may arise.

Control Hazards: Arise due to branch and other instructions that change the PC. Decision about next instruction needs more cycles. Control hazard may arise due to branch and other instructions like interrupts that change the program counter.

MIPS Pipeline Hazard

The hazards prevent a pipeline from operating at its maximum possible clock speed, which means that we cannot feed an Instruction every clock cycle. Such hazards can prevent some instructions from executing during its designated clock cycle.

Suppose, an instruction was supposed to be entering the pipeline now, it may have to wait for one clock cycle.

MIPS Pipeline Hazard

- **We can use some hardware and control logic circuits to avoid the clash or conflicts that arise due to hazard.**
- **Another alternative cheaper solution is to simply insert stall cycles without using special hardware.**
- **Suppose, there is a hazard, if the next instruction is allowed to enter the pipe, there will be an clash/conflict in pipe stages.**
- **In this case, we hold the instruction back and instruction may be fed in the next clock cycle, not in the current clock cycle.**
- **A stall cycle has been inserted for one cycle and no new instruction will be fed in the pipeline. After stalling the pipeline for one cycle, again we are feeding the next instruction after that. Then the hazard will not occur.**

MIPS Pipeline Hazard

- **If this principle is followed and one instruction is stalled, then all the next instructions following this stalled instruction will also get stalled because unless this instruction enters the pipe, the next instructions will not be able to enter the pipe. The depending on the criticality of the hazard, the number of stall cycles can vary.**

MIPS Pipeline Hazard

- The instructions that have already entered, the pipeline run in the process of execution, they can proceed normally; there is no problem,

- but all the next instructions following stalled instruction will be held back by inserting one or more stall cycles. The execution of these instructions will be delayed, and after the stall cycles, they will enter the pipeline.

- The instruction before the stall instruction can continue, but no new instructions can be fetched during the duration of the stall.

- Stalling will result in performance degradation, because we are not allowing instructions to be fed or entering the pipe in every clock cycle. Therefore, we are not able to get the full pipeline performance which is achievable in the ideal case.

MIPS Pipeline Hazard

Calculation of Performance Degradation:

We can estimate the speedup in a pipeline with respect to a non-pipelined version.

Simply, speedup = (the execution time of the non-pipelined version) ÷ (the execution time of the pipelined version).

Execution time of the non-pipelined version (ET_{nopipe}) = CPI (Cycles Per Instruction) X C_{nopipe} (clock cycle time) X I_{count} (Number of Instructions)

Execution time of the pipelined version (ET_{pipe}) = CPI (Cycles Per Instruction) X C_{pipe} (clock cycle time) X I_{count} (Number of Instructions).

$$\text{Pipeline Speedup} = \frac{ET_{nonpipe}}{ET_{pipe}} = \frac{CPI_{nopipe} \times C_{nopipe}}{CPI_{pipe} \times C_{pipe}} = \frac{C_{nopipe}}{C_{pipe}} \times \frac{CPI_{nopipe}}{CPI_{pipe}}$$

Actually, pipelining techniques reduce CPI or Clock cycle time.

The ideal CPI of an instruction pipeline can be written as $CPI_{ideal} = \dfrac{CPI_{nopipe}}{Pipeline\ Depth}$

$$\text{Speedup} = \frac{C_{nonpipe}}{C_{pipe}} \times \frac{CPI_{ideal} \times Pipeline\ Depth}{CPI_{ideal} + (Pipeline\ stall\ cycles\ /Instruction)}$$

MIPS Pipeline Hazard

Performance Analysis with Stall Cycles:

CPI for Ideal Pipeline = 1, but practically, It is not possible to get CPI=1. Suppose, we are using the method of inserting stall cycles for removing hazards. The actual CPI of the pipeline will obviously be greater than the ideal CPI. It will be ideal CPI plus average pipeline stall cycles per instruction.
Here, we are only considering stall only.

$$CPI_{pipe} = CPI_{ideal} + (\text{Pipeline Stall cycles per Instruction})$$

We can write the Speedup of pipeline:

$$\text{Speedup} = \frac{C_{nonpipe}}{C_{pipe}} \times \frac{CPI_{ideal} \times \text{Pipeline Depth}}{CPI_{ideal} + (\text{Pipeline stall cycles /Instruction})}$$

Ignoring any increase in pipeline clock cycle i.e. [Pipeline Clock cycle (C_{pipe})= Non-pipeline clock cycles ($C_{nonpipe}$)]

$$\text{Speedup} = \frac{CPI_{ideal} \times \text{Pipeline Depth}}{CPI_{ideal} + (\text{Pipeline stall cycles /Instruction})}$$

Pipeline Depth = No. of Pipe Stages

MIPS Pipeline Hazards: Structural

Structural hazard arises due to the resource conflicts if two instructions require same resource in the same cycle. Major functional units are used in different cycles. Basically, overlapping of execution of multiple instructions may introduce conflicts when the hardware can support overlapped execution. If the hardware does not support concurrent or overlapped execution, the structural hazards will occur.

Problem Area:

Structural Hazards occur:-

- When a datapath consists of single memory. In that case, in the same time while a instruction is being fetched from memory, some other instruction is trying to read data from memory or write data in the memory. If Datapath does not have separate instruction and data, then when an instruction is being fetched and some other instruction is trying to read or write, the next instruction will have to wait. We will have to insert a stall cycle.

MIPS Pipeline Hazards: Structural

- When an instruction is trying to read data from Register file (ID Stage), while some other instruction is trying to write into a register (WB stage). If “Write” operation is performed in the first half of the clock and “Read” operation is performed in the second half of the clock cycle, then this structural Hazards will be avoided.

- When some functional units are not fully pipelined (example floating point ADD or Multiply). In this case, same instruction is trying to use same stage for more than one cycles and a sequence of instructions that are trying to use the same functional stage will be stalled.

- The functional units like floating point adder or multiplier are not fully pipelined. The instructions consist of floating point add or floating point multiply consume more than one cycles. These will result in stall cycles because when the instruction with floating point ADD/Multiply will be using multiple clock cycles during the EX stage, because the multiply cannot finish in one cycle. So, it is using the EX stage for more than one cycle. If the next instruction is also requiring floating point multiplication, next instruction will have to wait and stall cycles will be inserted.

MIPS Pipeline Hazards: Structural

Example shows structural hazards due to use of single memory system, which stores both instructions and data.

Instruction	Clock Cycles or time steps								
	1	2	3	4	5	6	7	8	9
LW R1, 70(R2)	IF	ID	EX	$\boxed{\text{MEM}}$	WB				
ADD R3,R4,R5		IF	ID	EX	MEM	WB			
SUB R10,R2,R9			IF	ID	EX	MEM	WB		
AND R5,R7, R7				$\boxed{\text{IF}}$	ID	EX	MEM	WB	
ADD R2, R2,R5					IF	ID	EX	MEM	WB

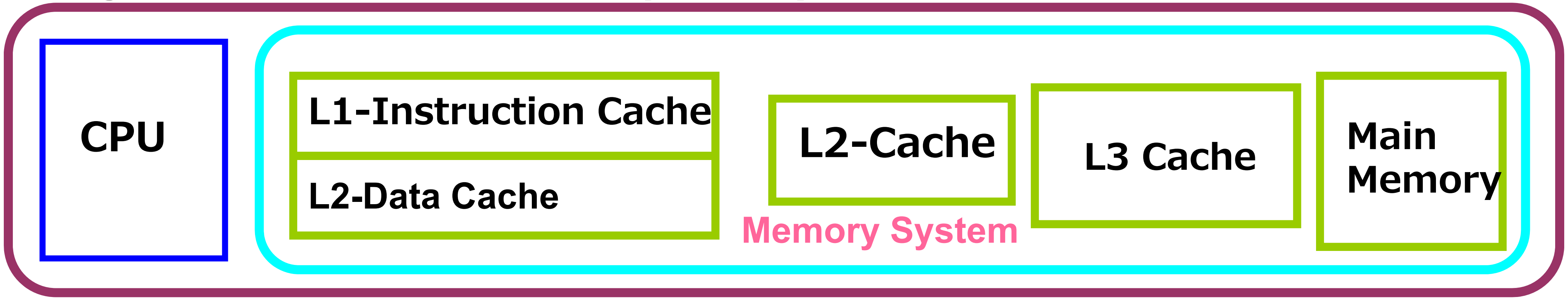
There is a conflict between IF & MEM indicated in 4 cycle. LW instruction is trying to access memory and fetch data from memory for loading Data in the register File. But at the same time, instruction AND (i+3) is also trying to read from memory for an instruction fetch in IF cycle. There will be Structural Hazards as Datapath does not have separate instruction memory and data memory. In this case, we can insert a stall cycle. Hardware will automatically detect that there is a conflict as both instruction LW and AND Instructions are trying to access memory. HW will automatically insert a stall cycle and Datapath will resume the instruction fetch in the next cycle. All instructions that follow will also incur a one cycle delay.

MIPS Pipeline Hazards: Structural

Based on Amdahl's law, the functional unit appears very rarely for example, floating point unit for this unit, it is not wise to invest anything to improve this unit to remove structural hazard. In this case, the stall cycles is usually inserted to avoid the conflict.

But the functional units that are frequently used, are modified or replicating additional hardware. In these cases, adding stall cycles will be expensive.

The structural hazards due to Memory access is frequent. Structural Hazard due to memory Access can be avoided by using separate instruction and data caches. The IF will use the instruction cache and MEM will the data cache. So, the conflict may be removed or avoided. Generally, L1-Data cache and Instruction cache are incorporated for holding Data and Instruction separately.



MIPS Pipeline Hazards: Structural

Example shows structural hazards due to read from Register File and write operation into Register File in the same clock cycle.

Instruction	Clock Cycles or time steps								
	1	2	3	4	5	6	7	8	9
LW R1, 70(R2)	IF	ID	EX	MEM	WB				
ADD R3,R4,R5		IF	ID	EX	MEM	WB			
SUB R10,R2,R9			IF	ID	EX	MEM	WB		
AND R5,R1, R7				IF	ID	EX	MEM	WB	
ADD R2, R2,R5					IF	ID	EX	MEM	WB

In cycle 5, first instruction, i.e., instruction i (LW R1, 70(R2)) is trying to write R1 in the Register. After execution of LW instruction, the Data accessed from the memory indicated by the calculated effective address will be written into the R1 in the WB phase.

Again, in the ID phase, the register operands are pre-fetched and here, in cycle 5, R1 and R7 registers will be read in ID Phase in 5 cycle. There is a conflict in cycle 5. The instruction {AND (R5,R1,R7)} is trying to read R1 from the register bank and instruction {LW R1, 70(R2) is trying to write into the register in the same cycle in WB stage.

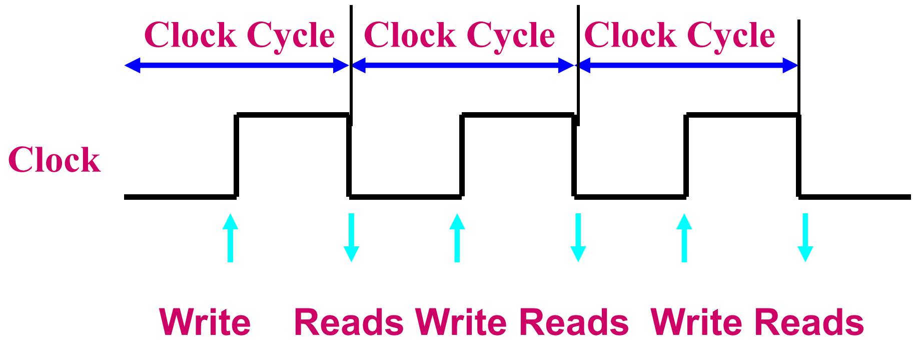
MIPS Pipeline Hazards: Structural

- As the register bank has 2 separate read ports, the reading 2 registers concurrently is not problem.

- In the MIPS pipeline, we divide the cycle time into two halves as shown. Each clock cycle has two clock edges: leading (rising) edge indicated by the up-arrow and falling edge indicated by down arrow as shown in Fig.

- In the leading edge, all the register writes will be done, and in falling edge, all the register reads will be performed because register access is fast compared to clock cycle time as it is long. The clock cycle time is dividing into 2 halves; in the first half we are doing the write, in the second half we are doing the reads.

- In this case, (LW R1, 70(R2)) is writing into R1 and {AND (R5,R1,R7)} instruction is reading from R1, then first, write will happen and then the read of new data will be performed.



MIPS Pipeline Hazards: Data

- **Data hazards occur due to data dependencies between instructions that are in various stages of execution in the pipeline.**

R1 written in WB

Instruction	Clock Cycles or Time Steps								
	1	2	3	4	5	6	7	8	9
LW R1, 70(R2)	IF	ID	EX	MEM	WB				
ADD R3,R1,R5		IF	ID	EX	MEM	WB			
SUB R10,R2,R9			IF	ID	EX	MEM	WB		
AND R5,R1, R7				IF	ID	EX	MEM	WB	
ADD R2, R2,R5					IF	ID	EX	MEM	WB

R1 read in ID

The LW instruction is writing data in R1 in WB stage in Cycle 5 and in the 3rd cycle, ADD instruction will be pre-fetching all register operands in ID stage (Cycle 3). ADD instruction is getting old value of R1 before the value of R1 has been updated as new value of R1 will be written in WB stage (cycle 5). Data hazard occurs due to the data dependency among instructions. The correct execution of ADD instruction depends on the new value of R1 resulted from the execution of LW instruction.

MIPS Pipeline Hazards: Data

Solution to remove Data Hazards:

- **In this case, Data Hazards can be avoided by inserting stall cycles. After the ADD Instruction is decoded and the control unit determines that there is data dependency because this instruction uses the R1 operand that will come from the output of the LW instruction and ADD instruction will be waiting till WB is completed. So three stall cycles will be inserted to avoid the conflicts.**
- **In the simple implementation, 3 clock cycles are stalled.**
- **3 clock cycles are wasted.**

Instruction	Clock Cycles or Time Steps								
	1	2	3	4	5	6	7	8	9
LW R1, 70(R2)	IF	ID	EX	MEM	WB				
ADD R3,R1,R5		IF	ID	STALL	STALL	ID	EX	MEM	WB
SUB R10,R2,R9			IF	STALL	STALL	IF	ID	EX	MEM
AND R5,R1, R7				STALL	STALL		IF	ID	EX
ADD R2, R2,R5					STALL			IF	ID

MIPS Pipeline Hazards: Data

Techniques to reduce the number of stall cycles

- Increased number of stall cycle insertion degrades the performance of the pipeline Datapath. The following two techniques may be used to reduce the number of stall cycle insertion.

  a) Data Forwarding/bypassing: By using extra hardware consisting of multiplexers, the data needed can be forwarded as soon as they are computed, instead of waiting for the result to be written into the register.

  b) Concurrent Register Access: By splitting a clock cycle into two halves, register read and write can be carried out in the two halves of the a clock cycles (register write in first halves and register read in second half.

  The 1st instruction (LW) which is generating the result R1, is writing the result (R1) in WB, but it is seen that the value of the result (R1) after execution of LW is already calculated at the end of EX stage. But the value will be written into the register (R1) in WB stage. So, if we take the output of the ALU directly that is already calculated, and forward the value to next instruction by using some additional MUXes. In this case, we will not have to wait for the WB stage and we will get the value earlier. This is called data forwarding or bypassing, by using additional hardware consisting of MUXes. The data required can be forwarded as soon as they are computed, instead of waiting for the result to be written into the register file.

MIPS Pipeline Hazards: Data

Data Forwarding/bypassing to reduce the Data Hazards:

- **The result computed by the previous instruction is stored in some register (generally ALUout) just output of the ALU within the datapath. The value will be directly taken from ALUout without waiting till WB, and will be forwarded to the instruction that require the result.**

- **To implement the above technique, some additional connections, data transfer paths and some additional MUXes are needed.**

- **The control circuit will be a little complex because it will have to analyze the source and the destination registers of consecutive instructions, and it will automatically activate or deactivate this kind of forwarding paths whenever required. Basically the control circuit identifies the data dependencies and selects the multiplexers in a suitable manner.**

MIPS Pipeline Hazards: Data (Example: Forwarding)

Instruction	Clock Cycles or Time Steps								
	1	2	3	4	5	6	7	8	9
LW R1, 70(R2)	IF	ID	EX	MEM	WB				
ADD R3,R1,R5		IF	ID	EX	MEM	WB			
SUB R6,R4,R1			IF	ID	EX	MEM	WB		
AND R8,R1, R7				IF	ID	EX	MEM	WB	
ADD R9, R1,R5					IF	ID	EX	MEM	WB

- The 1st Instruction LW generates R1 and the register R1 is used in all consecutive next four instructions and the value generated by LW will be written in WB stage in 5th clock cycle in the normal case. The LW computes R1 that is required by next subsequent three instructions and the dependencies are shown by red arrows. The last instruction (ADD R9, R1, R5) is using ID after WB. So, it is not affected by data dependencies.
- Data dependency problem for 3rd instruction (AND R8, R1, R7) has already been solved by splitting the register access. "Writing" is done in the first half of the clock cycle and "Reading" is done in the second half of the clock cycle. So, the data dependencies conflict is already resolved for 3rd instruction (AND R8, R1, R7).
- Now, we will remove two conflicts for the instruction 2nd and 3rd. The 1st instruction is calculating R1 in the EX stage and the result is available at this stage. We need not have to wait till WB to be written into R1, we can directly take from output of EX stage of the 1st instruction and forward it to the input of EX stage of 2nd instruction (ADD R3, R1, R5) and 3rd Instruction (SUB R6, R4, R1).

MIPS Pipeline Hazards: Data (Example: Forwarding)

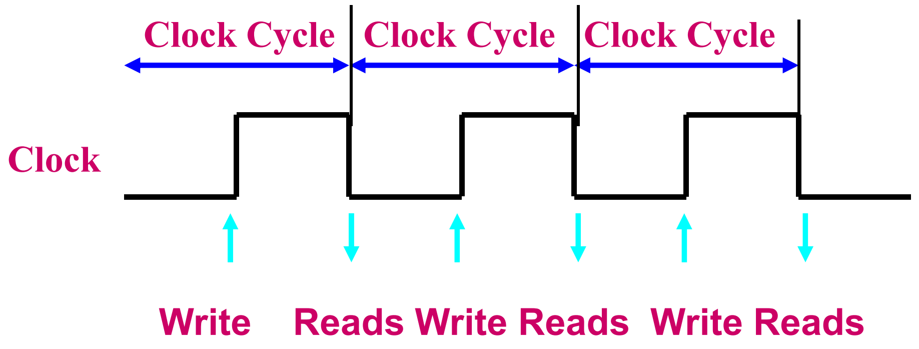
Instruction	Clock Cycles or Time Steps								
	1	2	3	4	5	6	7	8	9
LW R1, 70(R2)	IF	ID	$\boxed{\text{EX}}$	MEM	WB				
ADD R3,R1,R5		IF	ID	EX	MEM	WB			
SUB R6,R4,R1			IF	ID	EX	MEM	WB		
AND R8,R1, R7				IF	ID	EX	MEM	WB	
ADD R9, R1,R5					IF	ID	EX	MEM	WB

- The 1st instruction {LW R1, 70(R2)} completes the result at the end of EX stage shown by Blue box, but normally it will be written into R1 only in WB.
- We need to forward the result directly from the output of the ALU in EX stage of 1st instruction to the appropriate ALU input Registers of the following instructions.
- When the next instruction enters the EX stage, 1st instruction will already be in the MEM stage and the result has to be fed back to the input of the next instruction by using some multiplexer and additional paths.

MIPS Pipeline Hazards: Data

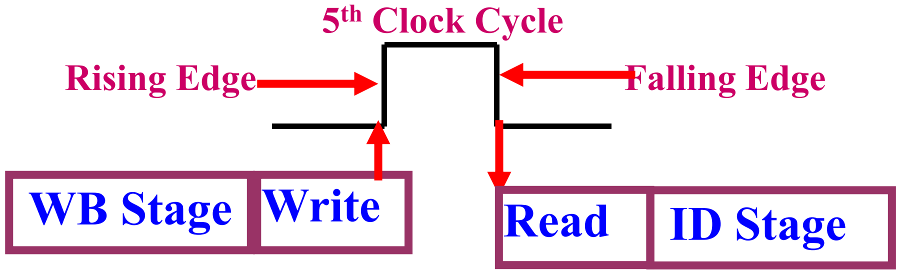
Data Hazard Reduction by Splitting Register Write/Read

- The data forwarding technique can solve the data hazards between ALU instructions.
- By splitting register read/write, we can reduce the number of data forwarding to avoid or reduce the dependency.
- In the earlier example, we need to forward 3 instructions.
- We can reduce number to 2 by avoiding the conflict between the 1st and 4th instruction, where WB and ID are accessed in the same cycle.
- We perform register write in WB stage during the 1st half of the clock cycle, and the register reads in ID stage during the 2nd half of the clock cycle. So the hazard between 1st and 4th instruction will be eliminated.



MIPS Pipeline Hazards: Data [Example]

- Performing register "Write" in WB stage during the 1st half (Rising Edge) of the 5th clock cycle, and the register "Read" in ID stage (Falling Edge) during the 2nd half of the 5th clock cycle, the data hazard between 1st and 4th instruction has been eliminated.
- Data forwarding has been reduced to 2 from 3 by accessing WB of 1st instruction and ID stage of 4th instruction during the 5th cycle. The data forwarding has been performed only for 2nd instruction and 4th instruction.



Write/Read Cycle in same cycle

Instruction	Clock Cycles or Time Steps								
	1	2	3	4	5	6	7	8	9
LW R1, 70(R2)	IF	ID	EX	MEM	WB				
ADD R3,R1,R5		IF	ID	EX	MEM	WB			
SUB R6,R4,R1			IF	ID	EX	MEM	WB		
AND R8,R1, R7				IF	ID	EX	MEM	WB	
ADD R9, R1,R5					IF	ID	EX	MEM	WB

Performance Issues in Pipelining

- Pipelining increases the CPU instruction throughput - the number of instructions completed per unit of time.
- It does not reduce the execution time of an individual instruction. In fact, it usually slightly increases the execution time of each instruction due to overhead in the pipeline control.
- Increase in instruction throughput means that a program runs faster and has lower total execution time.

Performance Issues in Pipelining

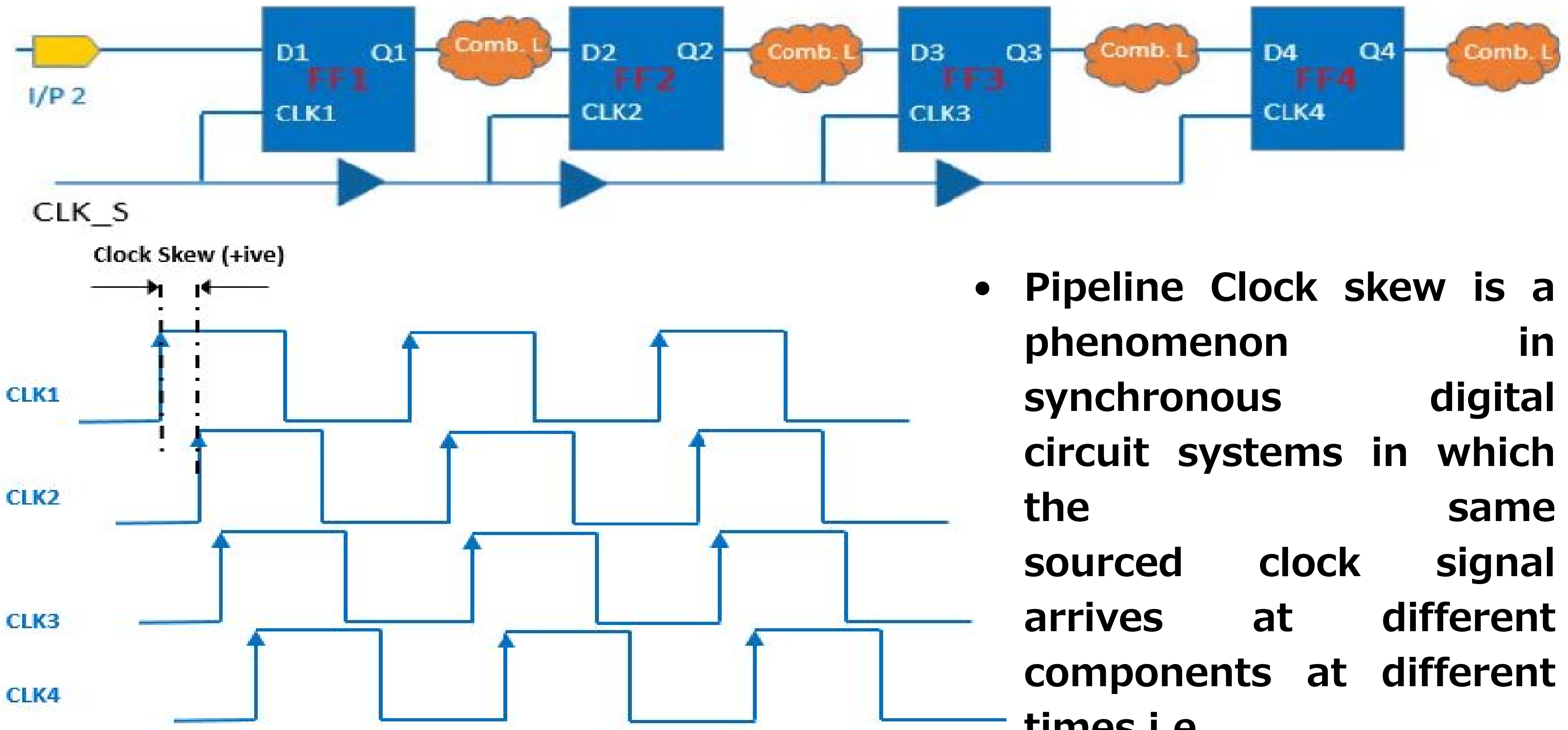
Limitations on practical depth of a pipeline arise from:

- **Pipeline latency: Execution time of each instruction does not decrease which puts limitations on pipeline depth.**

- **Imbalance among pipeline stages: Imbalance among the pipe stages reduces performance since the clock can run no faster than the time needed for the slowest pipeline stage.**

- **Pipeline overhead: Pipeline overhead arises from the combination of pipeline register delay (setup time plus propagation delay) and clock skew.**

Performance Issues in Pipelining



- **Pipeline Clock skew is a phenomenon in synchronous digital circuit systems in which the same sourced clock signal arrives at different components at different times i.e.**

the instantaneous difference between the readings of any two clocks is called their skew.

Performance Issues in Pipelining

Exercise: **Consider a non-pipelined machine with 6 execution stages of lengths 50ns, 50ns,60 ns, 60 ns, 50 ns, and 50ns.**

a) Find the instructional latency on this machine.
b) How much time does it take to execute 100 instructions?

Suppose, pipelining is introduced on this machine and assume that the clock skew of 5ns is added to each execution stage.

c) - What is the instruction latency on the pipelined machine?
d) How much time does it take to execute 100 instructions?
e) What is the speedup obtained from pipelining?

Solution:

a) Instruction latency = 50+50+60+60+50+50= 320 ns

b) Time to execute 100 instructions = 100*320 = 32000 ns

c) In pipelined implementation, length of all the pipe stages must be same, i.e. speed of slowest stage plus overhead. With 5ns overhead, it comes to:

Length of pipelined stage = MAX (lengths of un-pipelined stages) + clock skew
= 60 + 5 = 65 ns

Performance Issues in Pipelining

d) Instruction latency = 65 ns

Time to execute 100 instructions = 65*6*1 + 65*1*99 = 390 + 6435
= 6825 ns

Speedup is ratio of average instruction time without pipelining to the average instruction time with pipelining.

Average instruction time not pipelined = 320 ns

Average instruction time pipelined = 65 ns

Speedup for 100 instructions = 32000 / 6825 = 4.69

Assignments

Problem 1: What are the pipeline Hazards? What are the different types of Pipeline Hazards?

Problem 2: Derive the expression for the speedup for Pipeline Datapath.

Problem 3: (a) What are the causes of the structural Hazards? (b) How do you remove the Structural Hazards in the pipeline Dataparth?

Problem 4: (a) What are the causes of the structural Hazards? (b) How do you remove the Structural Hazards in the pipeline Dataparth?

Problem 5: (a) How do you remove the Structural Hazards occurred due to use of single memory System? (b) (a) How do you remove the Hazards occurred due to the Write/ Read operation in the register file in the same cycle?

Problem 6: (a) What are the causes of the Data Hazards? (b) How do you remove the Data Hazards in the pipeline Dataparth?

Assignments

Problem 7: Consider a non-pipelined machine with 6 execution stages of lengths 50ns, 50ns,60 ns, 60 ns, 50 ns, and 50ns.

a) Find the instructional latency on this machine.
b) How much time does it take to execute 100 instructions?

Suppose, pipelining is introduced on this machine and assume that the clock skew of 5ns is added to each execution stage.

c) - What is the instruction latency on the pipelined machine?
d) How much time does it take to execute 100 instructions?
e) What is the speedup obtained from pipelining?

Problem 8: Consider the un-pipelined processor and assume that it has a 1ns clock cycle and that it uses 4 cycles for ALU operations and branches and 5 cycles for memory operations. Assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively. Suppose, that due to clock skew and setup, pipelining the processor adds 0.2 ns of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?

Pipeline Hazards: Branch/Control Hazard

Control hazard comes due to the fact that some decision has to be taken based on the output of some instruction while other instructions are already in the pipeline.

Branch instruction: beg R1, R2, OFFSET

When R1 and R2 are same, then we have to take the branch and the branch address will be the OFFSET value away from the current PC value.

- Branch has to be taken or branch is not to be taken that depends upon the output of this instruction [beg R1, R2, OFFSET].
- R1 and R2 are equal or not that decision will come to know only after the EX Stage i.e. in MEM Stage.
- Next PC (PC+4) value will be taken only after EX Stage, that means, for three clock periods, the pipeline will not be able to take new instructions.
- This is the problem which is called pipeline stalling.
- This hazard is known as *control hazard*.

Pipeline Hazards: Branch/Control Hazard

- **Three stall cycles have inserted to avoid the pipeline Hazards. Therefore, there is a huge loss in performance by Control Hazard.**
- **3 cycles are wasted for every branch.**

Example: Assume that branch frequency = 30% and CPI_{Ideal}=1.
CPI due to pipeline stalls = CPI_{Ideal}+ No. of Stall cycles =1+3=4
Actual CPI=0.7 x1+0.3x4=1.9
Performance degradation is 90% as Speed becomes almost half due to doubling of the Actual CPI.

Instruction	Clock Cycles or Time Steps										
	1	2	3	4	5	6	7	8	9	10	11
beg R1, R2, OFFSET	IF	ID	EX	MEM	WB						
Instruction (i+1)		Stall	Stall	Stall	IF	ID	EX	MEM	WB		
Instruction (i+2)			Stall	Stall	Stall	IF	ID	EX	MEM	WB	
Instruction (i+3)				Stall	Stall	Stall	IF	ID	EX	MEM	WB
Instruction (i+4)					Stall	Stall	Stall	IF	ID	EX	MEM
Instruction (i+5)						Stall	Stall	Stall	IF	ID	EX
Instruction (i+6)							Stall	Stall	Stall	IF	ID

Pipeline Hazards: Branch/Control Hazard

Reduction of Branch Penalty

There are two issues to achieve the reduction of branch Penalty.

a) We have to determine whether the branch is taken or not,
b) We have to compute the branch target address earlier in the pipeline

- **Both of these have to be calculated early enough so that the stall cycles can be reduced. This is easier in MIPS32 datapath because of the simplicity of the instructions. For computing whether a branch is taken or not, we have to check whether the comparison result of two registers is 0 or non-zero, that is, the branch depends on zero or non-zero result while comparing two registers [beg R1, R2, Offset].**
- **Registers are already fetched in ID, and a simple comparator can be added that hardly takes any time and (R1-R2) will be save in a register; whenever we are fetching that register we also check whether it is 0 or non-zero. The decision is already known at the end of ID itself. So, in MIPS32 the branches either require testing for 0 or comparing two registers because we are fetching all the registers in the ID. So, we have to add some special comparison logic.**
- **Branch target address can be calculated earlier by using a separate adder. At the end of the ID cycle, the decision “branch taken” or “not taken” will be known and the branch target address will also be known. In this case, we will have to wait for single cycle only, not for three cycles.**

Pipeline Hazards: Branch/Control Hazard

Reduction of Branch Penalty

- In the earlier technique, decision on branch taken or not taken was known at the end of the MEM stage
- But in the new approach, because of the simplicity of the MIPS datapath, this decision will be taken at the end of ID stage and we will come to know that whether the branch is taken and what is the address. We can start fetching after that decision. So one stall cycle will be required.

Instruction	Clock Cycles or Time Steps										
	1	2	3	4	5	6	7	8	9	10	11
beg R1, R2, OFFSET	IF	ID	EX	MEM	WB						
Instruction (i+1)		IF	IF	ID	EX	MEM	WB				
Instruction (i+2)			Stall	IF	ID	EX	MEM	WB			
Instruction (i+3)				Stall	IF	ID	EX	MEM	WB		
Instruction (i+4)					Stall	IF	ID	EX	MEM	WB	
Instruction (i+5)						Stall	IF	ID	EX	MEM	WB
Instruction (i+6)							Stall	IF	ID	EX	MEM

Pipeline Hazards: Branch/Control Hazard

Reduction of Pipeline Branch Penalty

We will discus four techniques to avoid the control hazards in MIPS32 pipeline implementation.

a) **Simplest scheme to handle branches is to *freeze the pipeline and insert (one) stall cycles.* pipeline, holding or deleting any instructions after the branch until the branch destination is known. Attractiveness of this solution lies primarily in its simplicity both for hardware and software.**

In this case, the branch penalty is fixed i.e., one cycle and cannot be reduced by software.

Instruction	Clock Cycles or Time Steps										
	1	2	3	4	5	6	7	8	9	10	11
beg R1, R2, OFFSET	IF	ID	EX	MEM	WB						
Instruction (i+1)		IF	IF	ID	EX	MEM	WB				
Instruction (i+2)			Stall	IF	ID	EX	MEM	WB			
Instruction (i+3)				Stall	IF	ID	EX	MEM	WB		
Instruction (i+4)					Stall	IF	ID	EX	MEM	WB	
Instruction (i+5)						Stall	IF	ID	EX	MEM	WB
Instruction (i+6)							Stall	IF	ID	EX	MEM

Pipeline Hazards: Branch/Control Hazard

Reduction of Pipeline Branch Penalty

Next, we will discuss two scheme (b) Branch Not Taken, or untaken, c) Branch Taken.

- **Control hazards can cause a greater performance loss for the pipeline compared to the data hazards. When a branch is executed, it may or may not change the PC to something other than its current value plus 4.**

- **If a branch changes the PC to its target address, then it is a *taken* branch. If it falls through, it is *not taken,* or *untaken.* If instruction *i* is a taken branch, then the PC is normally not changed until the end of ID, after the completion of the address calculation and comparison.**

Pipeline Hazards: Branch/Control Hazard

(b) Branch Not Taken, or untaken

This scheme is to treat every branch as not taken, simply allowing the hardware to continue as if branch was not executed.

- **In this five-stage pipeline, this *predicted-not-taken* or *predicted untaken* scheme is implemented by continuing fetching instructions as if the branch were a normal instruction. The hardware will be assuming that the branch is not taken, and it will fetch the consecutive instruction without any stall. For such cases there will be no penalty.**

Instruction	Clock Cycles or Time Steps										
	1	2	3	4	5	6	7	8	9	10	11
Branch Instr. not taken	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		
Instruction (i+5)						IF	ID	EX	MEM	WB	
Instruction (i+6)								IF	ID	EX	MEM

Predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top)

Pipeline Hazards: Branch/Control Hazard

Pipeline looks as if nothing out of the ordinary is happening. If the branch is taken, we need to turn the fetched instruction into a no-op and restart the fetch at the target address. If the branch is actually taken, then at the end of the ID stage, we will come to know that the prediction was wrong and will be incurring a 1-cycle penalty.

Instruction	Clock Cycles or Time Steps										
	1	2	3	4	5	6	7	8	9	10	11
Untaken Branch Instr.	IF	ID	EX	MEM	WB						
Instruction (i+1)		IF	IF	ID	EX	MEM	WB				
Instruction (i+2)			Stall	IF	ID	EX	MEM	WB			
Instruction (i+3)				Stall	IF	ID	EX	MEM	WB		
Instruction (i+4)					Stall	IF	ID	EX	MEM	WB	
Instruction (i+5)						Stall	IF	ID	EX	MEM	WB
Instruction (i+6)							Stall	IF	ID	EX	MEM

- If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall 1 clock cycle.

Pipeline Hazards: Branch/Control Hazard

c) Branch Taken: Next scheme is to treat every branch as taken. As soon as the branch is decoded and target address is computed, we assume the branch to be taken and begin fetching and executing at the target.

- But unfortunately for MIPS32, we will see that this prediction does not help because if we predict that the branch is taken; then we will always be fetching the next instruction from the target address, and the target address is known only at the end of ID. Only after ID stage, we can start the fetch. So, one cycle will be lost, irrespective of whether it is taken or not taken branch for MIPS32. This one cycle penalty will always be there.
- In case of MIPS32, we know the branch outcome and the target address both together. So, there is no advantage in this approach because in both cases there is one cycle penalty.

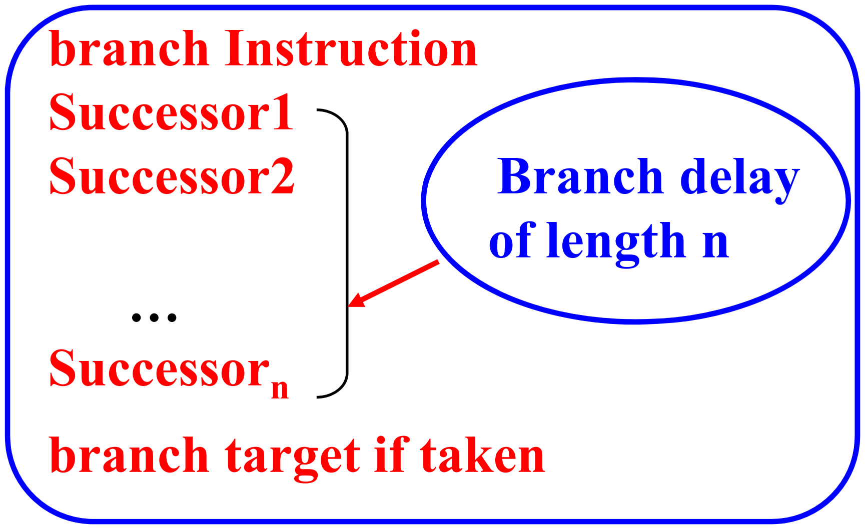
Taken or not taken Branch (1 cycle Penalty)

Instruction	Clock Cycles or Time Steps										
	1	2	3	4	5	6	7	8	9	10	11
BEQ Instruction	IF	ID	EX	MEM	WB						
Instruction (i+1)		IF	IF	ID	EX	MEM	WB				
Instruction (i+2)			Stall	IF	ID	EX	MEM	WB			
Instruction (i+3)				Stall	IF	ID	EX	MEM	WB		
Instruction (i+4)					Stall	IF	ID	EX	MEM	WB	
Instruction (i+5)						Stall	IF	ID	EX	MEM	WB
Instruction (i+6)							Stall	IF	ID	EX	MEM

Pipeline Hazards: Branch/Control Hazard

d) Delayed branch

- Delayed branch technique makes the hardware simple but puts responsibility on the compiler. This scheme was heavily used in early RISC processors and works reasonably well in five-stage pipeline.
- In practice, almost all processors with delayed branch have a single instruction delay; other techniques are used if the pipeline has a longer potential branch penalty.
- If a branch instruction incurs a penalty of n stall cycles, the execution of a branch instruction is defined as follows

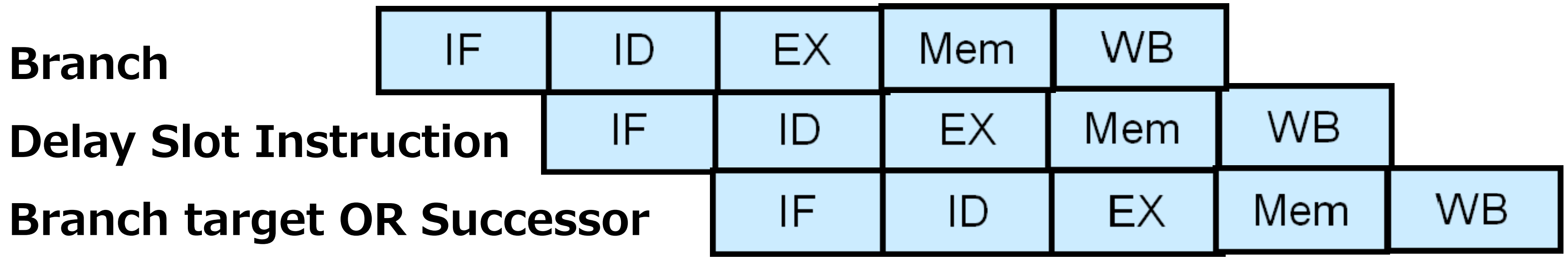


- The compiler will try to fill up the branch-delay slots with meaningful instructions.
- Instructions in the branch-delay slot are always executed irrespective of the outcome of the branch.
- For MIPS Datapath, n=1

Pipeline Hazards: Branch/Control Hazard

Simple idea: Put an instruction that would be executed anyway right after a branch

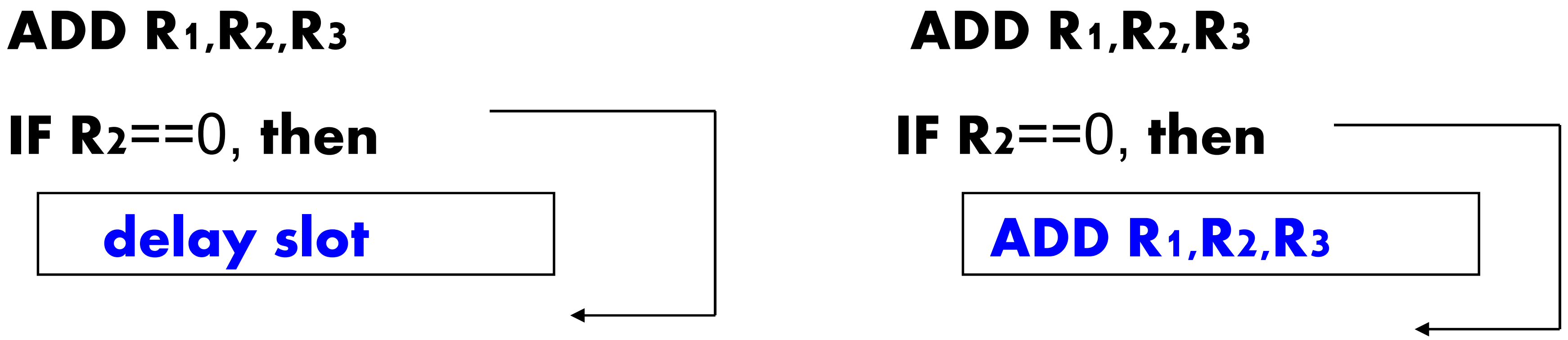
The first instruction is the branch instruction, the second instruction is the delay slot instruction and third instruction is the branch target or the successor of the instruction.



- **Basically, we have to put an instruction in the delay slot, that can safely be executed. No matter what the branch does that means whether the branch is taken or not taken that instruction can be executed. It will not lead to be converted into a nop.**
- **The compiler has to decide which instruction to be put in this delay slot.**

Pipeline Hazards: Branch/Control Hazard

Example: Possibility-1: an instruction from before: An instruction can be taken from before the branch. So, Here ADD R1, R2, R3 is a delay slot.



- ADD R1, R2, R3 instruction is appeared before this branch instruction. if R2 = 0, then it will jump to target location or targeted address. Here, ADD R1, R2, R3 is the targeted delay slot and this will be the normal instruction execution. So, ADD R1, R2, R3 is executed, after that this branch instruction encountered. So, this instruction will be executed whether this branch instruction is taken or not Because it is executed before the branch
- We are filling up this delay slot with ADD R1, R2 R3 and after that it will go to the target.
- Here, we are able to put an instruction which is useful and we do not have to convert into an NOP if prediction is wrong. So obviously, the movement of ADD R1,R2,R3 to the delay slot is the possible solution

Pipeline Hazards: Branch/Control Hazard

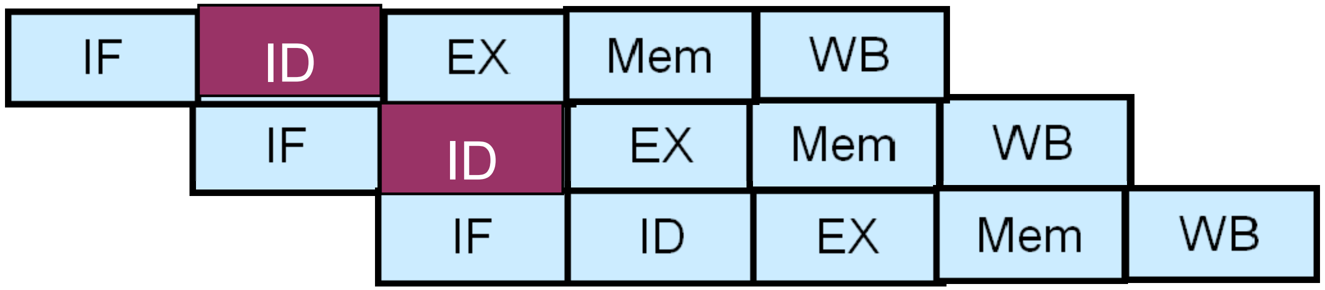
- We get to execute the ADD execution for free. Here, as if we are getting this instruction executed free of cost. Free of cost means that some instructions are supposed to be executed in this slot to avoid the branch penalty.
- So, as we go to the next instruction by that time we will know where the branch is taken or not taken and also the target address. So this instruction can be either it will can be the PC plus 4 or it can be that PC plus that immediate value which is available as the part of the instruction. As the branch is taken, so we find that this is the best solution.

(Before) : ADD R1,R2,R3

(Branch) : IF R2==0, then

(Delay Slot): ADD R1,R2,R3

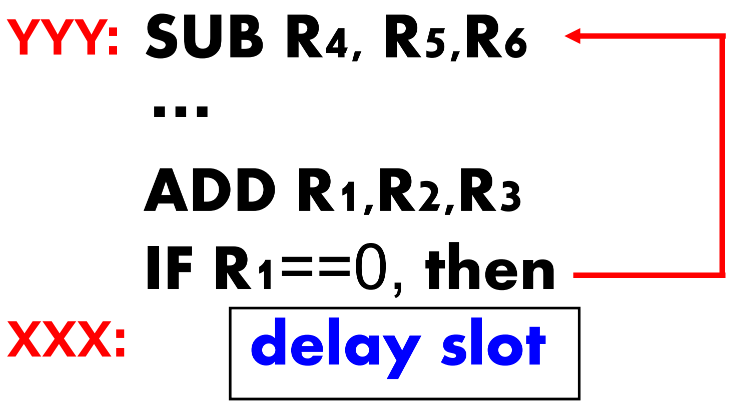
Branch target OR Successor



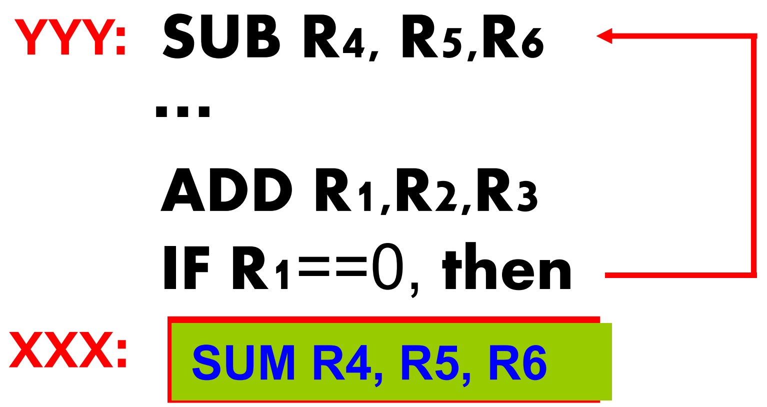
Pipeline Hazards: Branch/Control Hazard

Possibility-2: An instruction from the target will be put in the delay slot

- **In the example, if R1 = 0, then it is jumping to the instruction "SUM R4,R5,R6". As in the branch (If R1==0, then..), R1 depends on the execution of ADD R1,R2,R3, ADD R1,R2,R3 will not be put in the delay slot. An instruction from the target will be put in the delay slot.**
- **SUM R4, R5, R6 will be put in the delay slot. Then the branch target can be changed to (XXX) from YYY. If the prediction is that the branch will be taken then this is very advantageous. So, for filling up the delay performance by an instruction from the target and by doing this, we are able to improve the performance. But improvement of performance takes place when branch is taken.**



```
YYY: SUB R4, R5,R6
     ...
     ADD R1,R2,R3
     IF R1==0, then
XXX:     delay slot
```

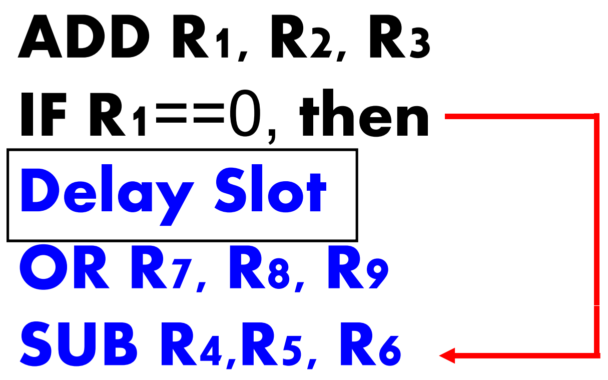


```
YYY: SUB R4, R5,R6
     ...
     ADD R1,R2,R3
     IF R1==0, then
XXX: SUM R4, R5, R6
```

Pipeline Hazards: Branch/Control Hazard

Possibility-3: An instruction from inside the taken path will be put in delay slot

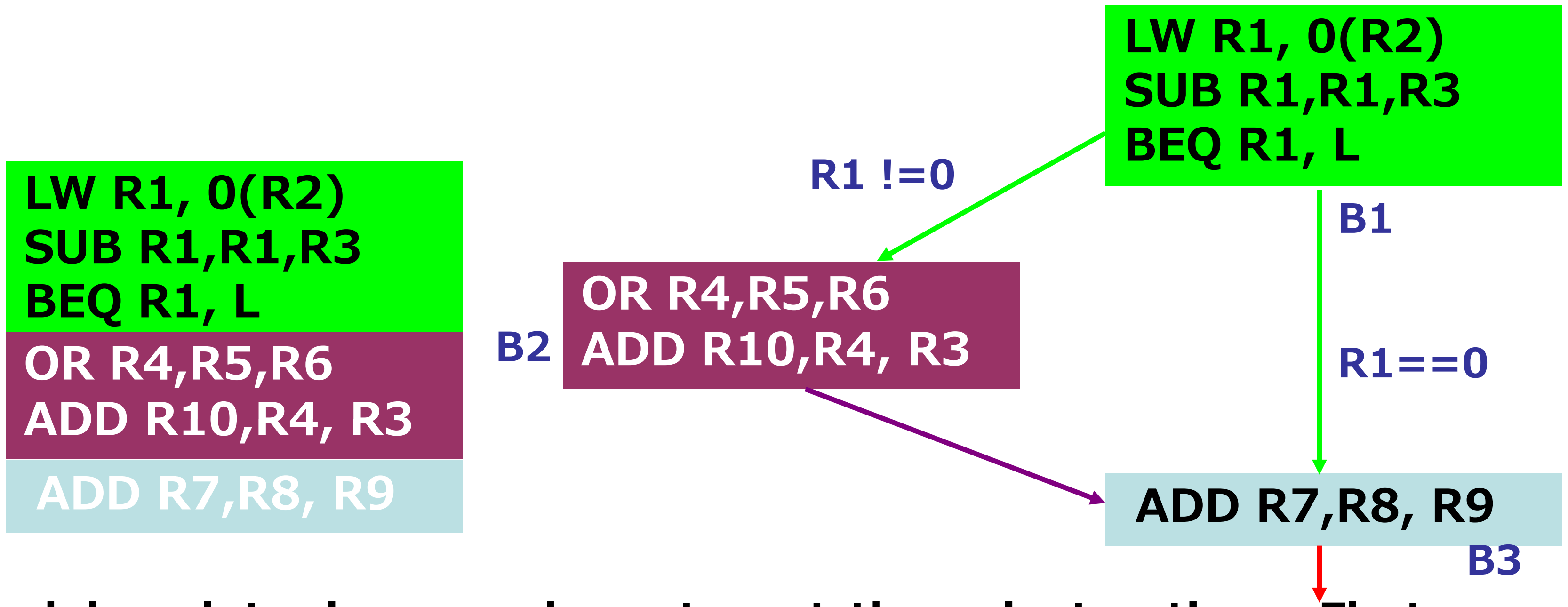
An instruction is taken from inside the "taken path" and put in the delay slot. This particular instruction can be moved into the delay slot only if its execution does not disrupt the program execution. Therefore, from the taken path, the delay slot is filled up by using the instruction (OR R7, R8,R9).



```
ADD R1, R2, R3
IF R1==0, then
Delay Slot
OR R7, R8, R9
SUB R4,R5, R6
```

Pipeline Hazards: Branch/Control Hazard

BEQ is dependent
on SUB on LW



We have to fill up the delay slot where we have to put these instruction. First, we take the instruction SUB R1,R1,R3. Unfortunately, we can not execute it in the delay slot because BEQ is dependent on this, because of this dependency we cannot move this instruction after this branch instruction.

If we know that branch is taken within a high probability, then ADD R7,R8, R9 could be moved into the block B1 that means the target address is moved after branch Instruction since it has no dependencies on B2. Since there is no dependency, it can be moved without any problem, but this solution will be good whenever branch is taken with high probability.
If the branch is not taken then " OR R4,R5,R6" could be moved into block B1, since it does not affect anything in B3. So, we can see we have got 3 possibilities whenever we can fill out by depending on different situations