BCS-29
Advanced Computer Architecture

Pipelined Processing

INSTRUCTION PIPELINE DESIGN

Pipeline Hazards

- ***Pipeline Hazards***
  - The situations that prevent the next instruction in the instruction stream from executing in its designated clock cycle.
  - Hazards reduce the performance from the ideal speedup gained by pipelining
- Three types of hazards
  - ***Structural hazards***
    - Arise from resource conflicts when the hardware can't support all possible combinations of overlapping instructions
  - ***Data hazards***
    - Arise when an instruction depends on the results of a previous instruction in a way that is exposed by overlapping of instruction in pipeline
  - ***Control hazards***
    - Arise from the pipelining of branches and other instructions that change the PC (Program Counter)

Structural Hazards

- If certain combination of instructions can't be accommodated because of resource conflicts, the machine is said to have a *structural hazard*
- It can be generated by:
  - Some functional unit is not fully pipelined
  - Some resources has not been duplicated enough to allow all the combinations in the pipeline to execute
  - For example: a machine may have only one register file write port, but under certain conditions, the pipeline might want to perform two writes in one clock cycle – this will generate structural hazard
    - When a sequence of instructions encounter this hazard, the pipeline will stall one of the instructions until the required unit is available
    - Such stalls will increase the Clock cycle Per Instruction from its ideal 1 for pipelined machines

Structural Hazards

Instruction Number	Clock number									
	1	2	3	4	5	6	7	8	9	10
load	IF	ID	EX	MEM	WB					
Instruction i+1		IF	ID	EX	MEM	WB				
Instruction i+2			IF	ID	EX	MEM	WB			
Instruction i+3				stall	IF	ID	EX	MEM	WB	
Instruction i+4						IF	ID	EX	MEM	WB
Instruction i+5							IF	ID	EX	MEM

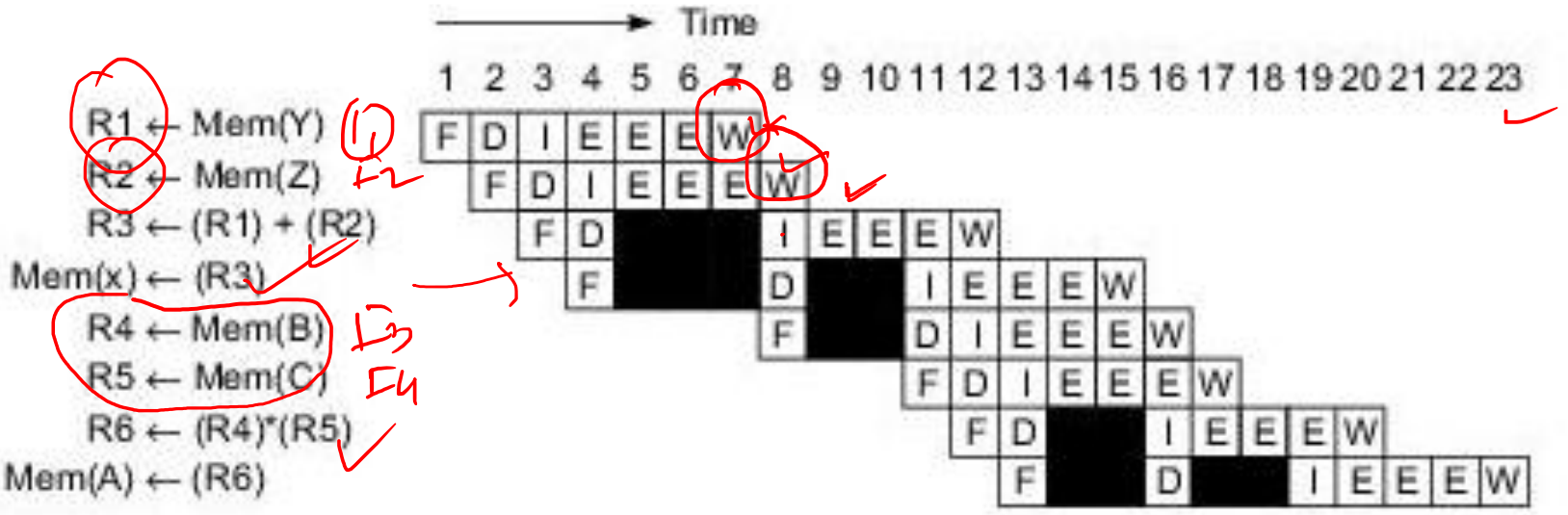
- A machine with structural hazard will have lower CPI
- Why a designer allows structural hazard?
  - To reduce cost
    - Pipelining all the functional units or duplicating them may be too costly
  - To reduce latency
    - Introducing too many pipeline stages may cause latency issues

INSTRUCTION PIPELINE DESIGN

Pipelined Instruction Processing

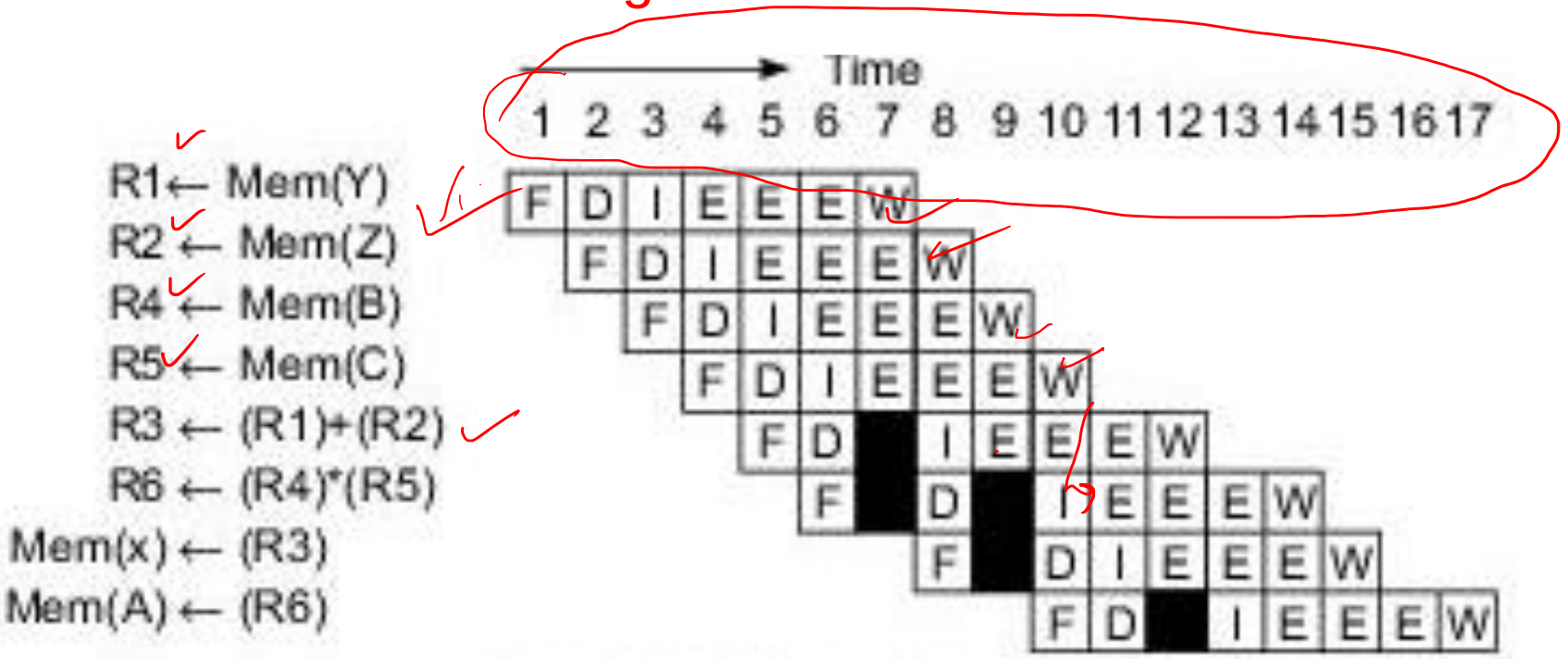
X = Y + Z and A = B X C

In-order Instruction issuing



INSTRUCTION PIPELINE DESIGN

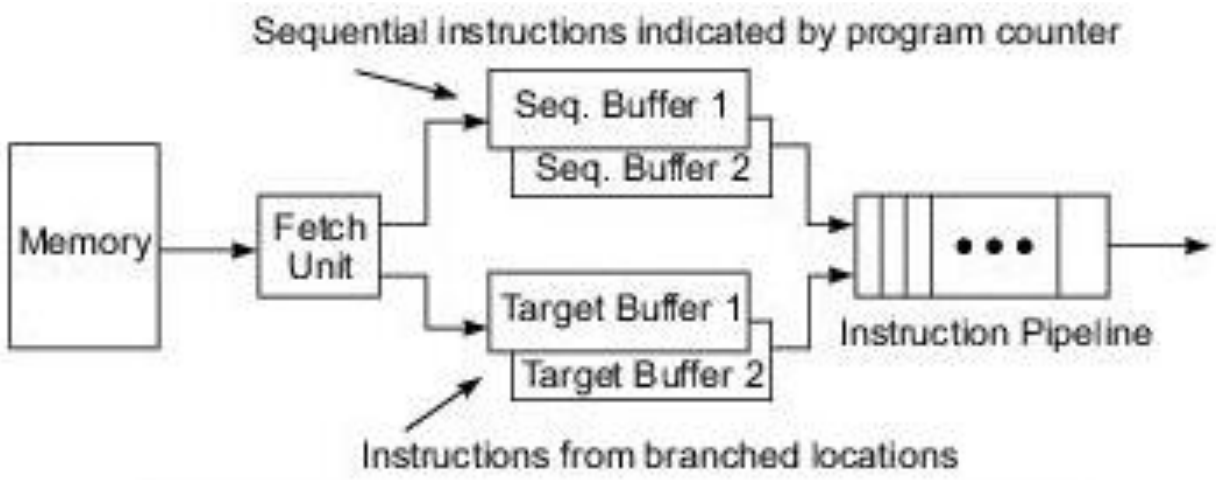
Reordered Instruction issuing



Mechanisms for Instruction Pipelining

Prefetch Buffers:

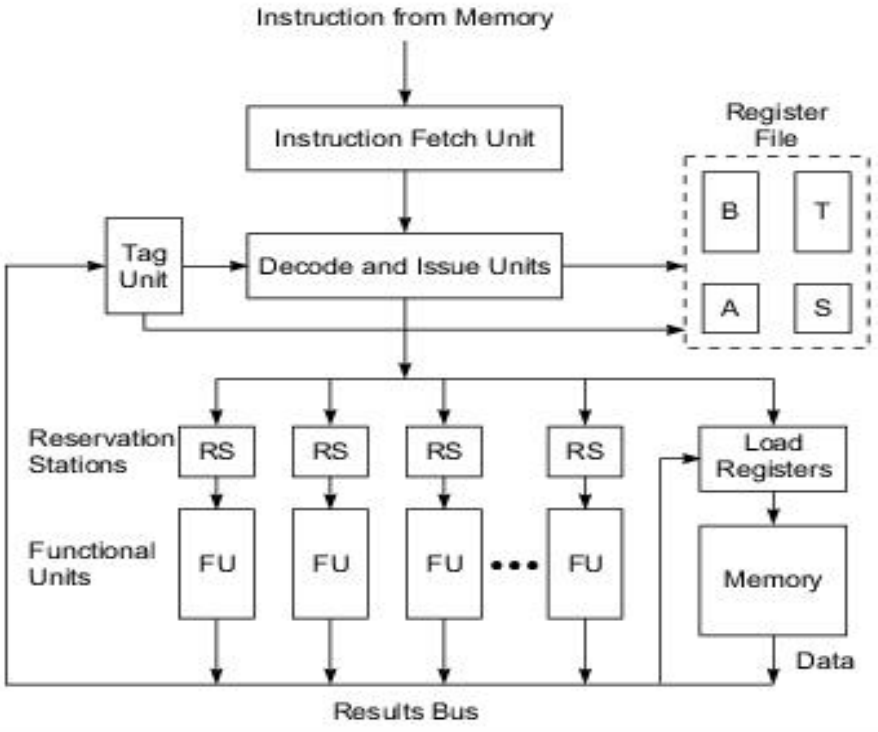
- Three types of buffers can be used to match the instruction fetch rate to the pipeline consumption rate.
- In one memory-access time, a block of consecutive instructions are fetched into a prefetch buffer.
  - **Sequential Buffer:** Sequential instructions are loaded into this Buffer for in-sequence pipelining.
  - **Target Buffer:** Instructions from a branch target are loaded into this buffer for out-of-sequence pipelining.
  - **Loop Buffer:** This buffer holds sequential instructions contained in a small loop. The loop buffers are maintained by the fetch stage of the pipeline.



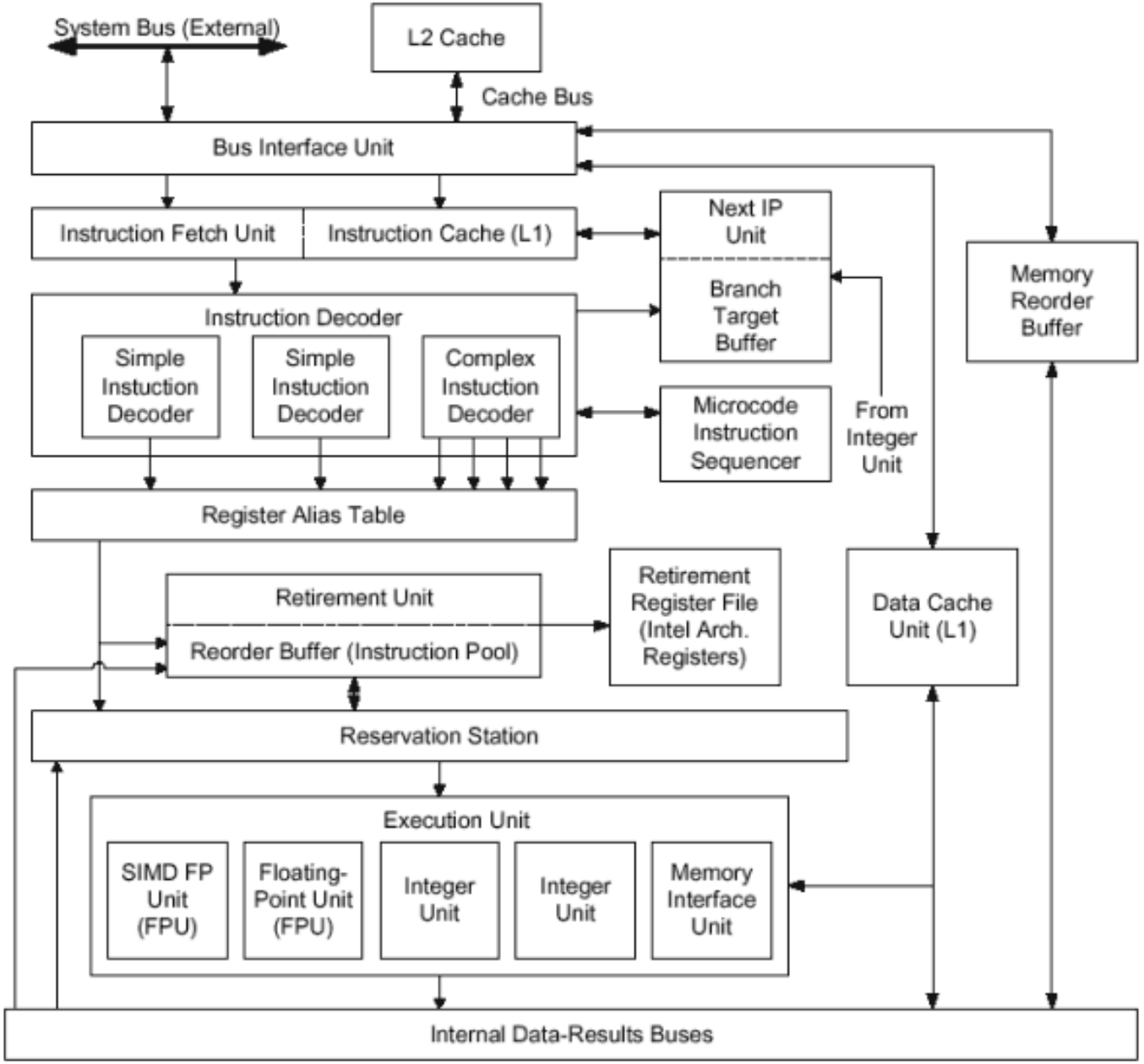
Mechanisms for Instruction Pipelining

Multiple Functional Unit:

- Sometimes a certain pipeline stage becomes the bottleneck. This stage corresponds to the row with the maximum number of checkmarks in the reservation table. This bottleneck problem can be alleviated by using multiple copies of the same stage simultaneously. This leads to the use of multiple execution units in a pipelined processor design.



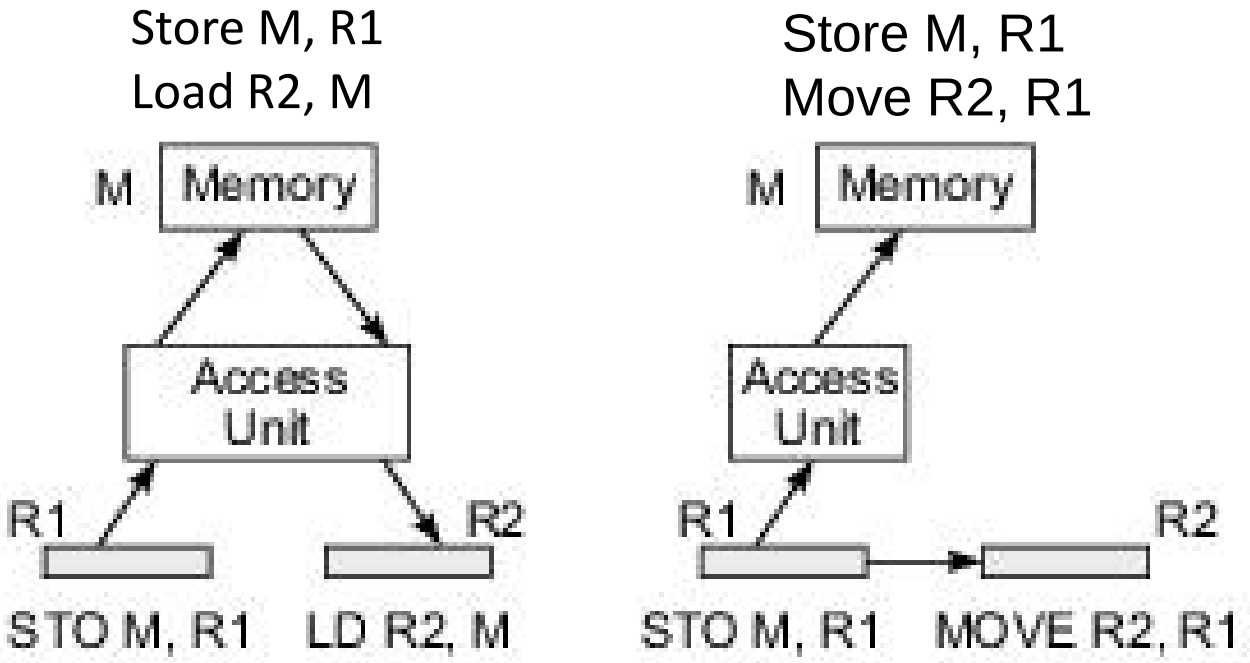
P6 Microarchitecture



Mechanisms for Instruction Pipelining

- Internal Dara Forwarding:

The throughput of a pipelined processor can be further improved with internal data forwarding among multiple functional units. In some eases, some memory-access operations can be replaced by register transfer operations.



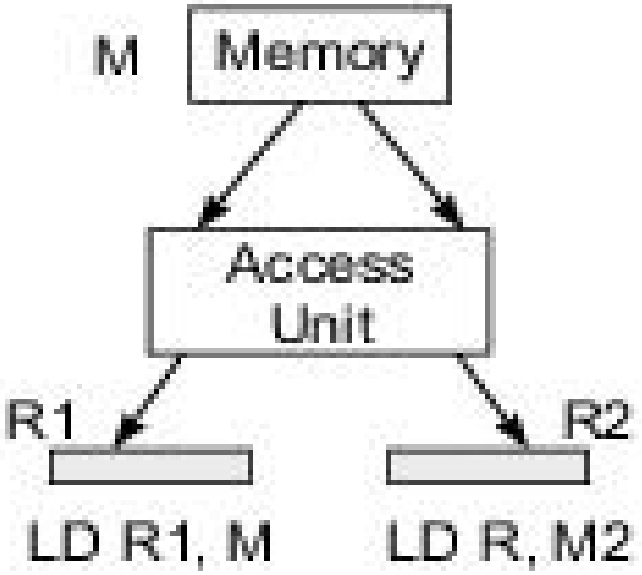
Store-Load forwarding

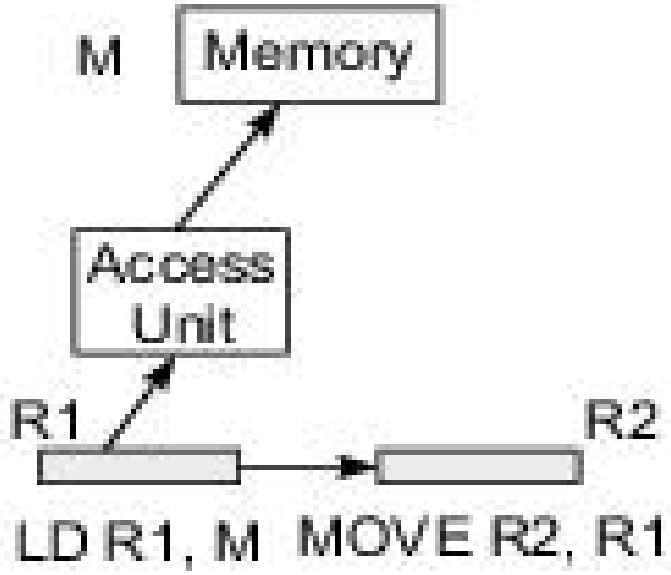
Mechanisms for Instruction Pipelining

Internal Dara Forwarding

Load R1, M
Load R2, M

Load R1, M
Move R2, R1

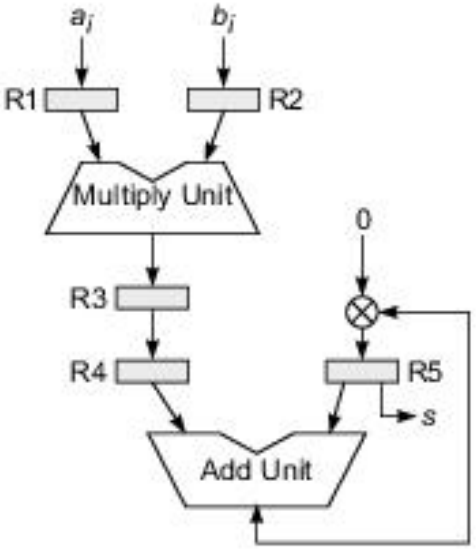




Load-Load forwarding

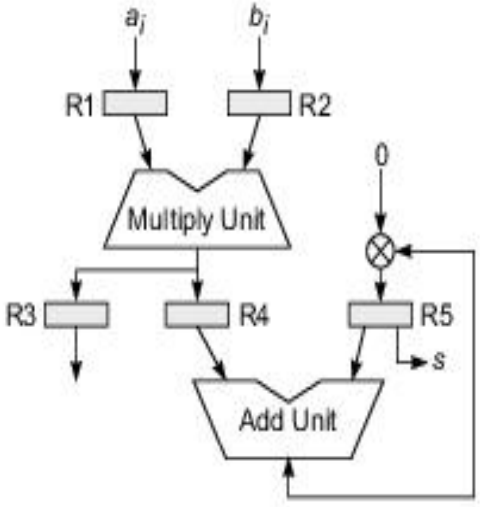
Mechanisms for Instruction Pipelining

Example of Internal Dara Forwarding



(a) Without data forwarding

I_1: R3 ← (R1) * (R2)
I_2 : R4 ← (R3)
I_3 : R5 ← (R5) + (R4)

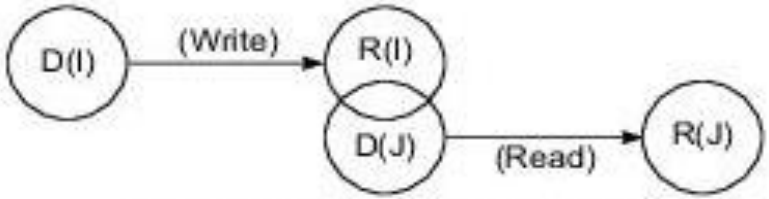


I'_1: R3 ← (R1) * (R2)
I'_2 : R4 ← (R1) * (R2)
I'_3 : R5 ← (R4) + (R5)

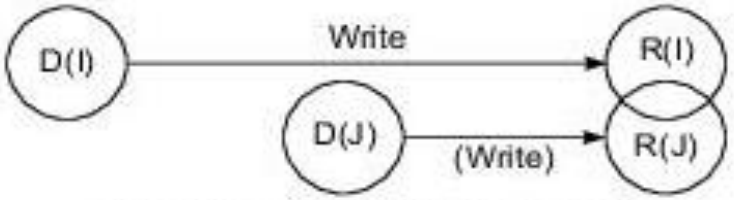
I'_1 and I'_2 can be executed simultaneously with internal data forwarding.

Pipeline Hazards

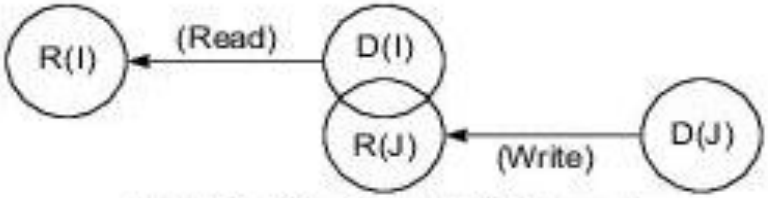
- The read and write of shared variables by different instructions in a pipeline may lead to Hazard.
- Hazards should be prevented before these instructions enter the pipeline.



(a) Read-after-Write (RAW) hazard



(b) Write-after-Write (WAW) hazard



(c) Write-after-Read (WAR) hazard

$$R(I) \cap D(J) \neq \phi \text{ for RAW hazard}$$

$$R(I) \cap R(J) \neq \phi \text{ for WAW hazard}$$

$$D(I) \cap R(J) \neq \phi \text{ for WAR hazard}$$

Dynamic Instruction Scheduling

Static Scheduling:

- This scheme is supported by an optimizing compiler.
- Data dependences in a sequence of instructions create interlocked relationships among them.
- Interlocking can he resolved through a compiler-based static scheduling approach.
- A compiler can be used to increase the separation between interlocked instructions by re sequencing.

Dynamic Instruction Scheduling

- Tomasulo's Algorithm
  - Register tagging
  - Hardware based Dependence resolution
- Scoreboarding Technique
  - Scoreboard: The centralized control Unit
  - A type of Data Driven Mechanism
- Enables out-of-order execution and allows out-of-order completion.
- Split the ID pipe stage of pipeline into 2 stages:
  - Decode instructions, check for structural hazards
  - Wait until no data hazards, then read operands

Dynamic Instruction Scheduling

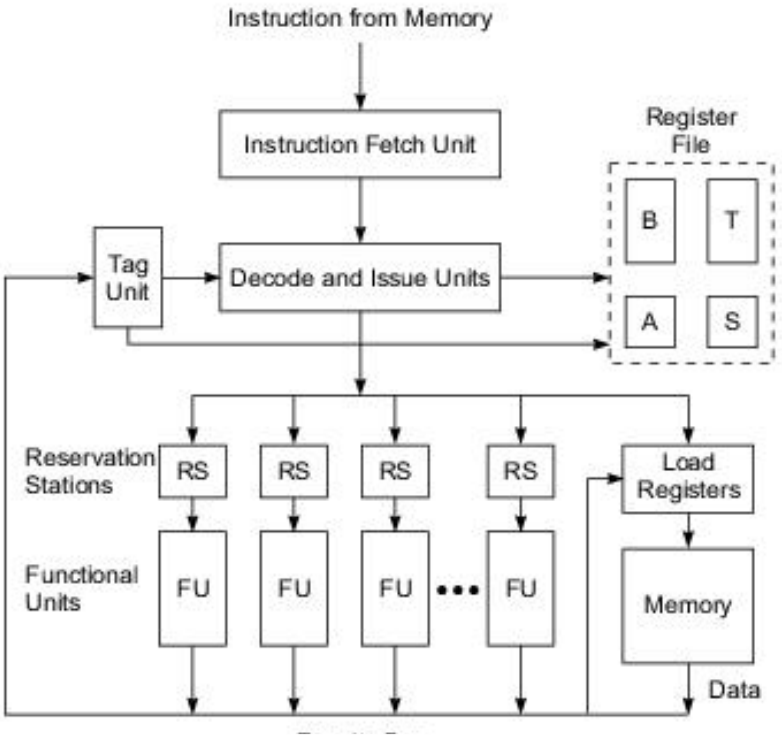
Tomasulo's Algorithm

- Control & buffers distributed with Function Units (FU)
  - FU buffers called "reservation stations"; have pending operands
- Registers in instructions replaced by values or pointers to reservation stations (RS);
  - form of register renaming ;
  - avoids WAR, WAW hazards
  - More reservation stations than registers, so can do optimizations compilers can't
- Results to FU from RS, not through registers, over Common Data Bus that broadcasts results to all FUs
- Load and Stores treated as FUs with RSs as well

Dynamic Instruction Scheduling

Tomasulo's Algorithm

- An issued instruction whose operands are not available is forwarded to an RS associated with the functional unit it will use.
- It waits until its data dependences have been resolved and its operands become available.
- The dependence is resolved by monitoring the result bus, when all operands for an instruction are available, it is dispatched to the functional unit for execution
- All working registers are tagged. If a source register is busy when an instruction reaches the issue stage, the tag for the source register is forwarded to an RS.
- When the register data becomes available, it also reaches the RS which has the same tag.



Dynamic Instruction Scheduling

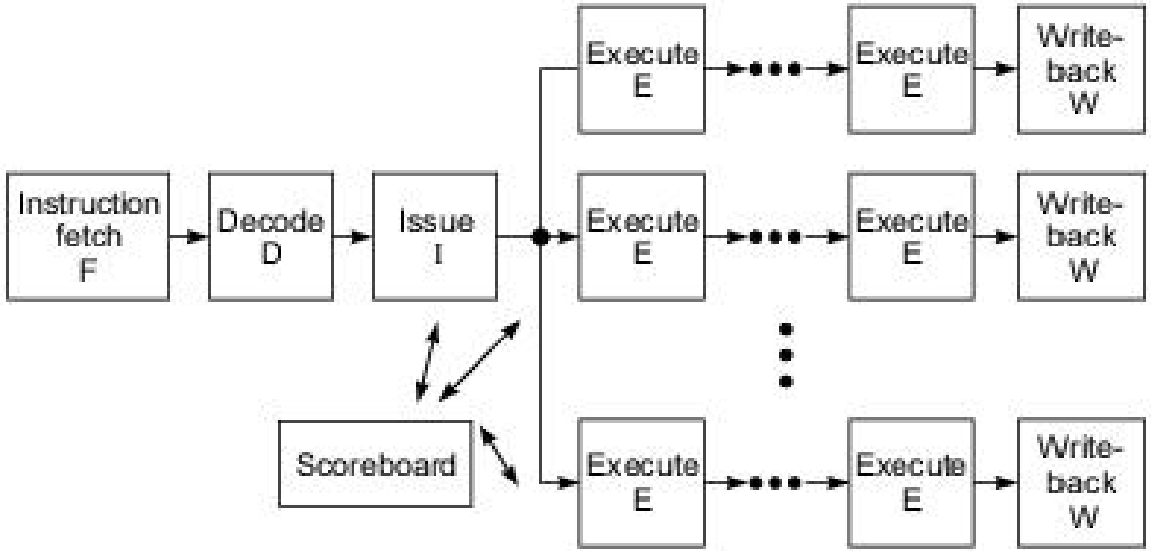
Scoreboarding –

- Technique for allowing instructions to execute out of order when there are sufficient resources and no data dependencies.
- Scoreboard keeps track of dependencies, state or operations
- Scoreboard replaces ID, EX, WB with 4 stages
  - ID1: Issue — decode instructions & check for structural hazards
  - ID2: Read operands — wait until no data hazards, then read operands
  - EX: Execute — operate on operands; when the result is ready, it notifies the scoreboard that it has completed execution
  - WB: Write results — finish execution; the scoreboard checks for WAR hazards. If none, it writes results. If WAR, then it stalls the instruction

The Scoreboard

Three Parts of the Scoreboard

- Instruction status—which of 4 steps the instruction is in
- Functional unit status—Indicates the state of the functional unit (FU).
- Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions will write that register



Branch Handling Techniques

Source code segment

```
for ( i = 0; i < 100; i++)
        if ( a[i] <= 50)
                j = j + 1;
        else
k = k +1;
```

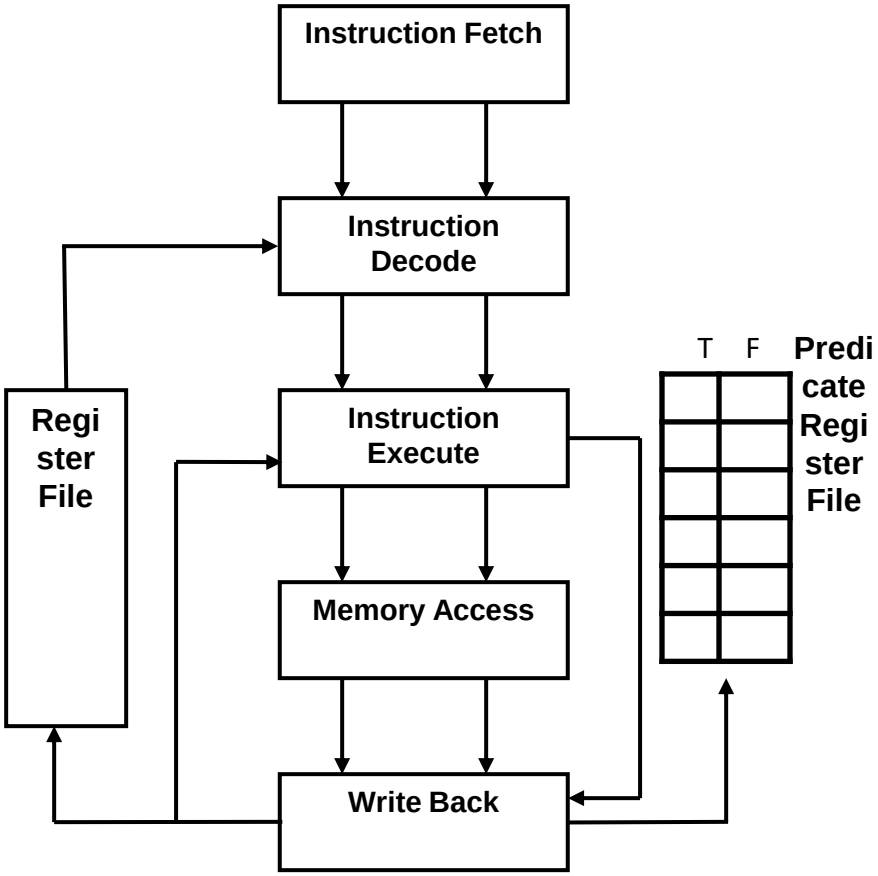
Assembly code segment

```
        mov r1,0
        mov r2,0
        ld_i r3, A, 0
        L1:
        Ld_i r4,r3,r2
        Bgt r4, 50, L2
        Add r5, r5, 1
        Jmp l3
        L2:
        add r6,r6,1
        L3:
        add r1,r1,1
        add r2,r2,4
         blt r1,100,L1
```

assembly code segment after if-conversion

```
        mov r1,0
        mov r2,0
        ld i r3,A,0
        L1:
        ld i r4,r3,r2
        pgt p1(U),p2(U),r4,50
        add r5,r5,1       (p2)
        add r6,r6,1       (p1)
        add r1,r1,1
        add r2,r2,4
         blt r1,100,L1
```

Branch Handling Techniques



Branch Handling Techniques

Delayed branches:

- A delayed branch of d cycles allows at least d-1 useful instructions to be executed after the branch instruction.
- Execution of these instructions must be independent of branch instruction to achieve the zero branch penalty.
- Code motion across branches can be used to achieve a delayed branch and on non availability of useful instructions, NOPs can be used as fillers.

```
        •
        •
I1.   LOAD     R1, A
I2.   Dec      R3, 1
I3.   BrZero   R3, I5
I4,   Add      R2, R4
I5.   Sub      R5, R6
I6.   Store    R5, B
        •
        •
```

```
        •
        •
I2.   Dec      R3, 1
I3.   BrZero   R3, I5
I1.   Load     R1, A
I4.   Add      R2, R4
I5.   Sub      R5, R6
I6.   Store    R5, B
        •
        •
```

Branch Handling Techniques

Branch Prediction:

Branch can be predicted either based on branch code types statically or dynamically, based on branch history during program execution.

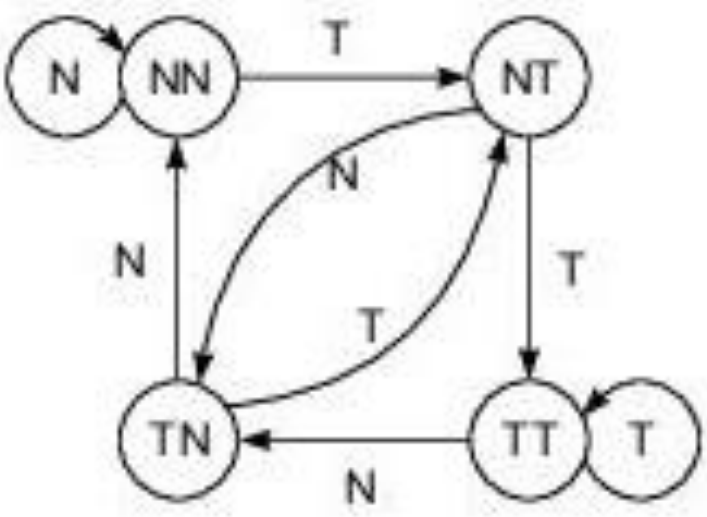
Static Branch Prediction:

- The static prediction direction (token or not taken) can even be wired into the processor. According to past experience, the best performance is given by predicting taken.
- The wired-in static prediction cannot be changed once committed to the hardware.
- However, the scheme can be modified to allow the compiler to select the direction of each branch on a **semi-static** prediction basis.

Branch Handling Techniques

Dynamic Branch Prediction:

- The dynamic branch prediction strategy works better because it uses recent branch history to predict whether or not the branch will be taken next time when it occurs.
- Dynamic prediction demands additional hardware to keep track of the past behavior of the branch instructions at run time.
- Following state transition diagram may used for tracking the last two outcomes at each branch instruction in a given program.



Captions:
T = Branch taken
N = Not-taken branch
NN = Last two branches not taken
NT = Not branch taken and previous taken
TT = Both last two branches taken
TN = Last branch taken and previous not taken