MODULE V

- Instruction pipeline design, Arithmetic pipeline deign - Super Scalar Pipeline Design

INSTRUCTION PIPELINE DESIGN

- A stream of instructions can be executed by a pipeline in an overlapped manner

Instruction Execution Phases

- Instruction fetch
- Instruction decode
- Operand fetch
- Execute and
- Write-back

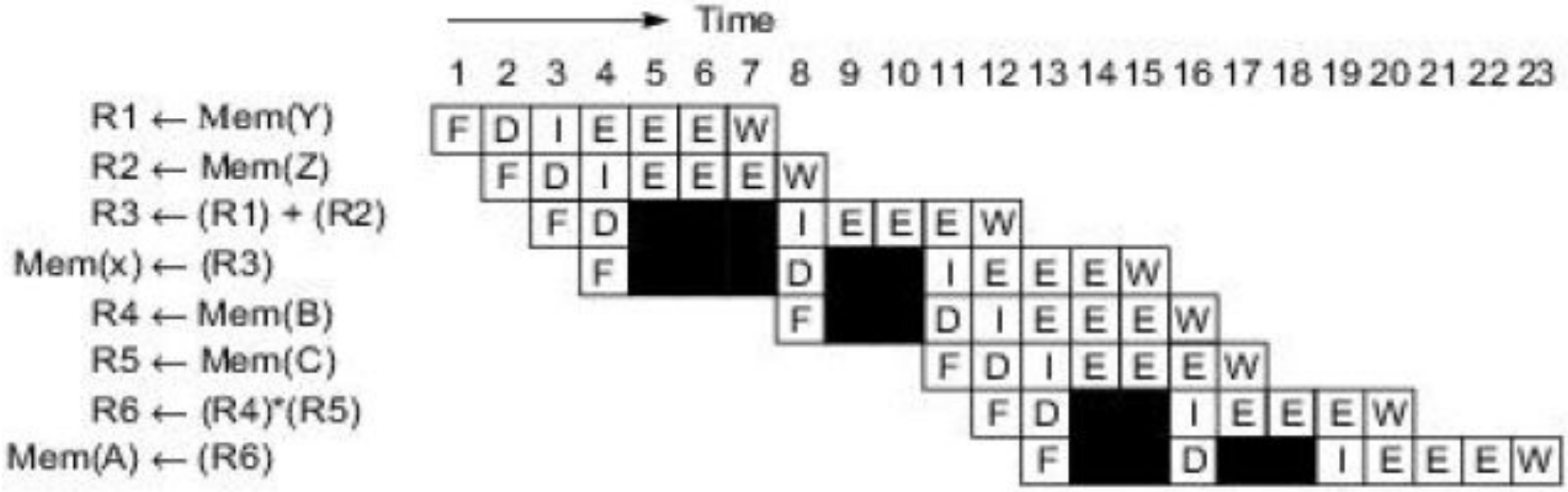
Pipeline

(a) A seven-stage instruction pipeline

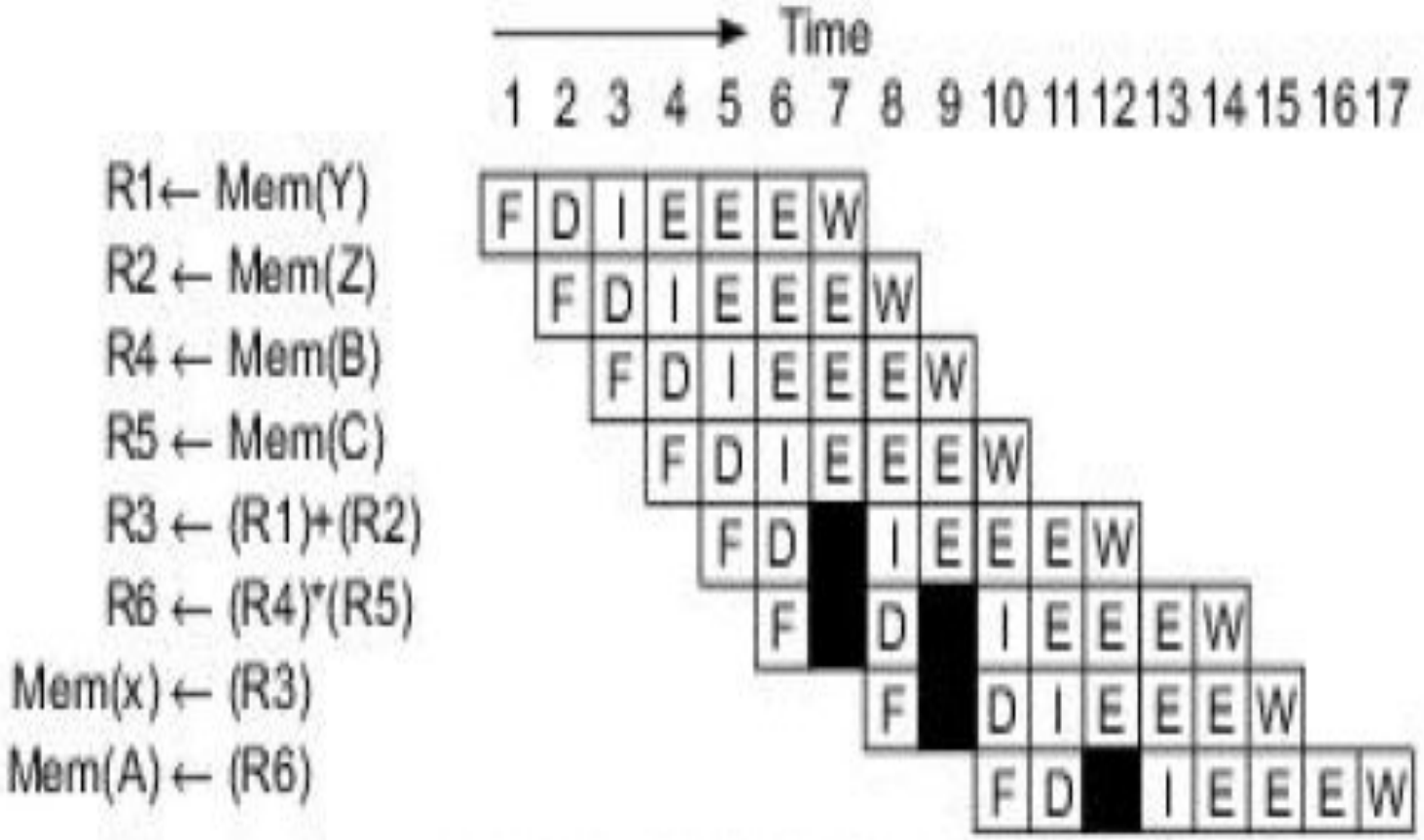
Pipe-lined Instruction Processing

- The fetch stage (F)fetches instructions from a cache memory, ideally one per cycle.
- The decode stage (D) reveals the instruction function to be performed and identifies the resources needed.
- The issue stage (I) reserves resources. The operands are also read from registers during the issue stage.
- The instructions are executed in one or several execute stages (E)
- The last write back stage (W) is used to write results into the registers.
- Memory load or store operations are treated as part of execution.

(a) A seven-stage instruction pipeline

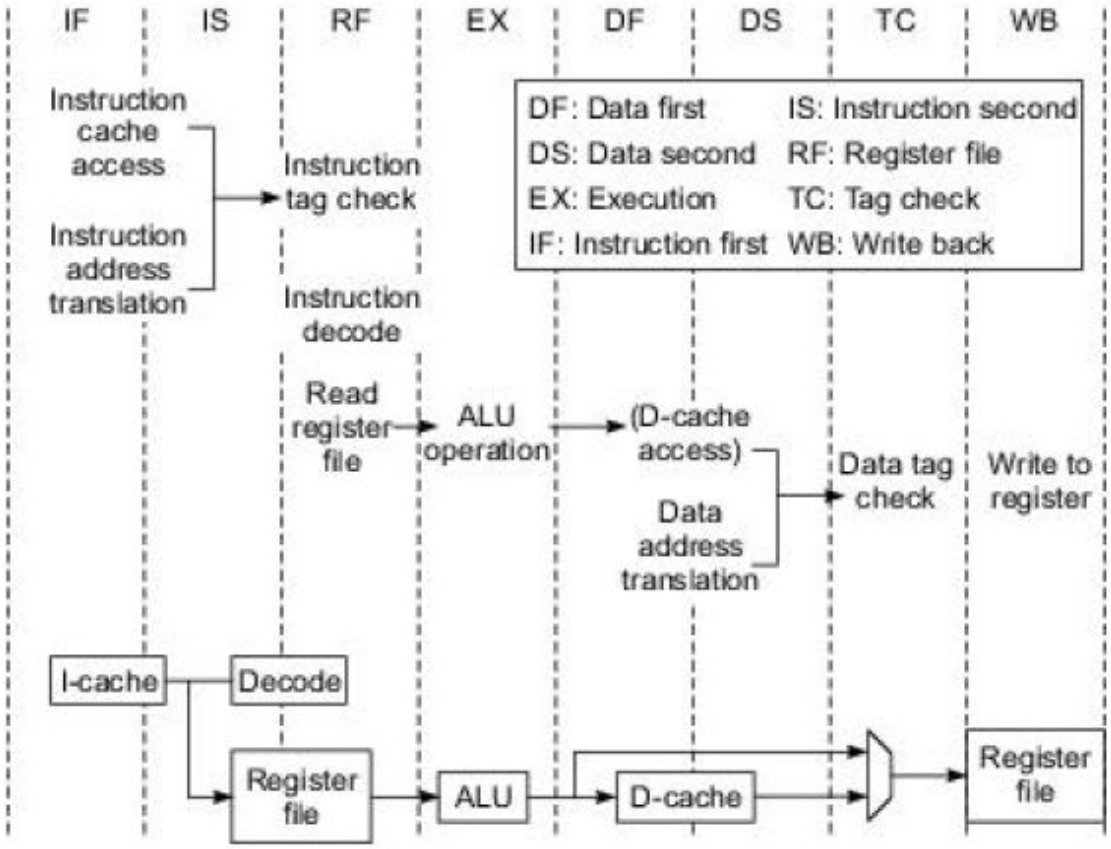


(b) In-order instruction issuing

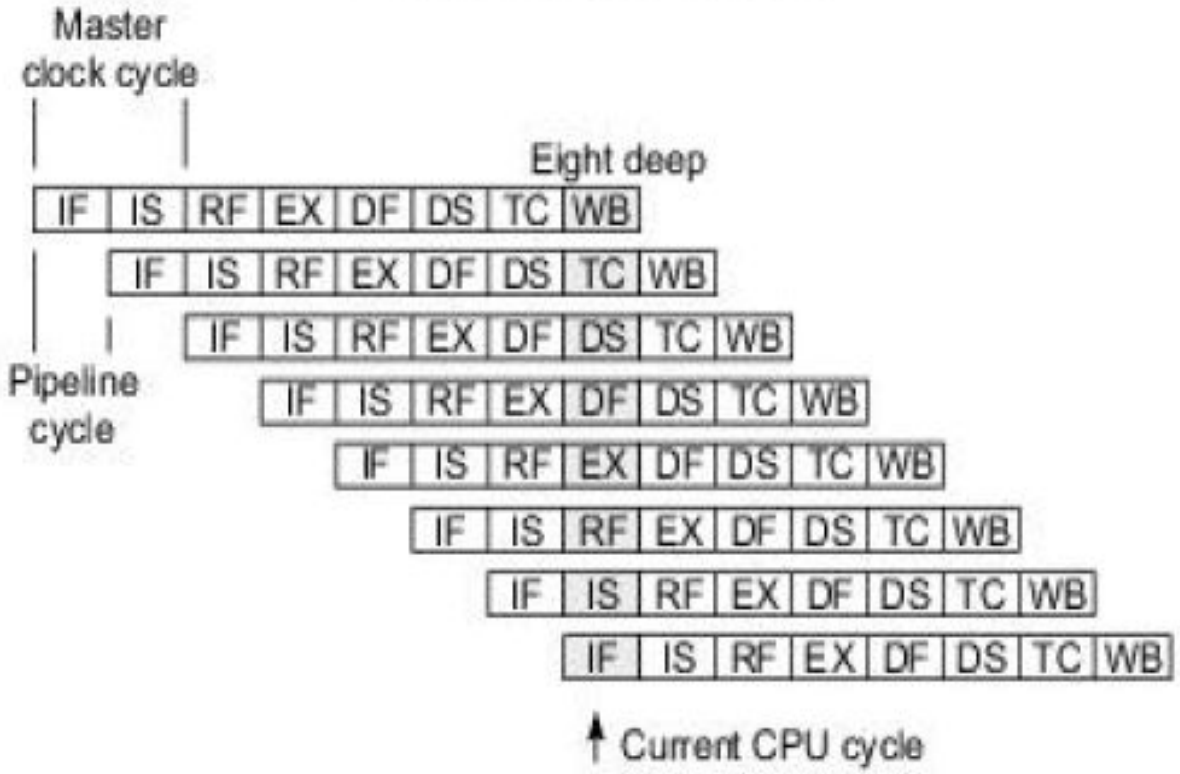


(c) Reordered instruction issuing

Example : The MIPS R4000 instruction pipeline



(a) R4000 pipeline stages



(b) R4000 instruction overlapping in pipeline

Mechanisms for Instruction Pipelining

- **Prefetch Buffer**
- **Multiple Functional Unit**
- **Internal Data forwarding**
- **Hazard Avoidance**

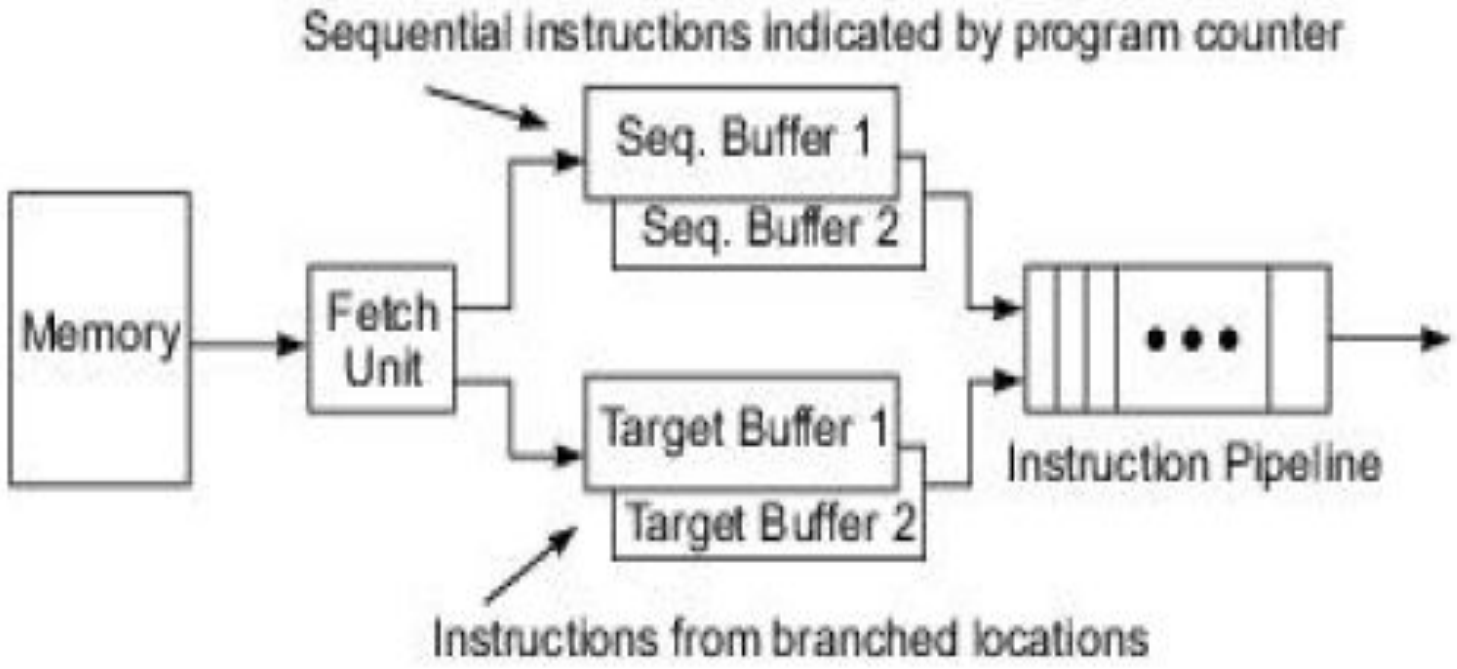


Fig. 6.11 The use of sequential and target buffers

Mechanisms for Instruction Pipelining

- **Prefetch Buffer**

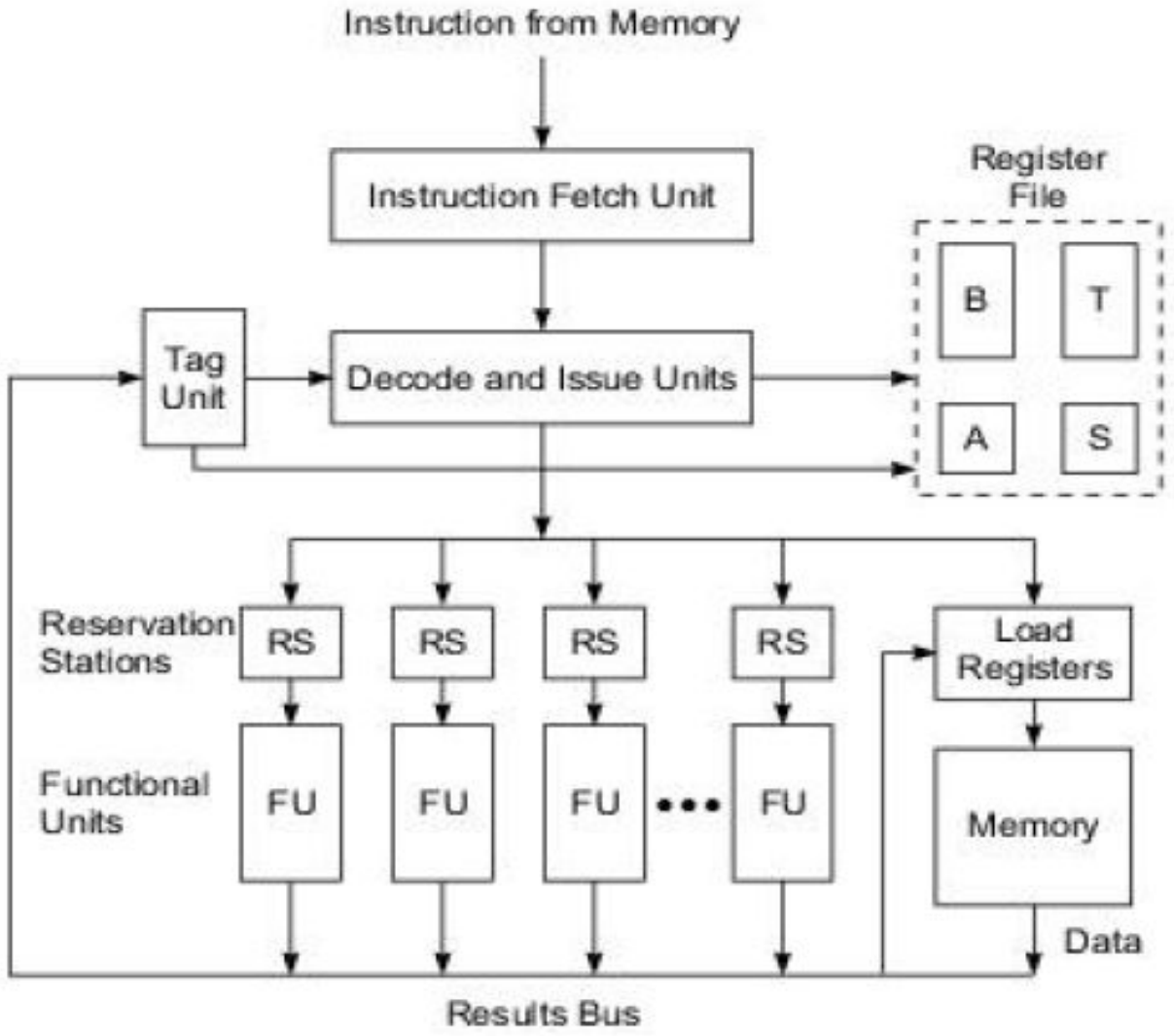
✔ Three types of buffers can be used to match the instruction fetch rate to the pipeline consumption rate.

- Sequential instructions are loaded into a pair of **sequential buffers** for in-sequence pipelining.
- Instructions from a branch target are loaded into a pair of **target buffers** for out-of-sequence pipelining.
- Both buffers operate in a FIFO fashion

- A third type of prefetch buffer is known as **loop buffer**
- This buffer holds sequential instructions contained in a small loop

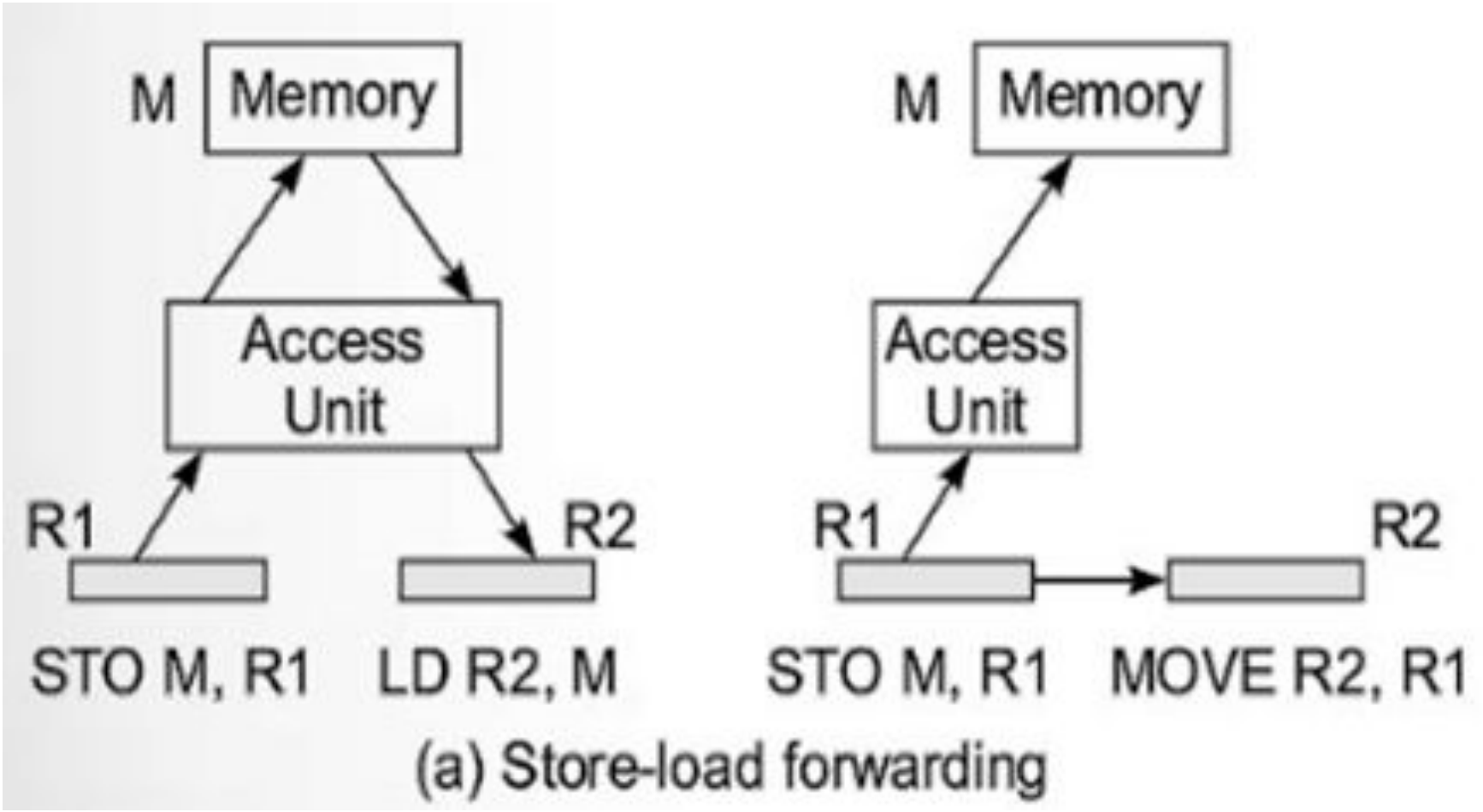
Multiple Functional Unit

- Sometimes a certain pipeline stage becomes the bottleneck.

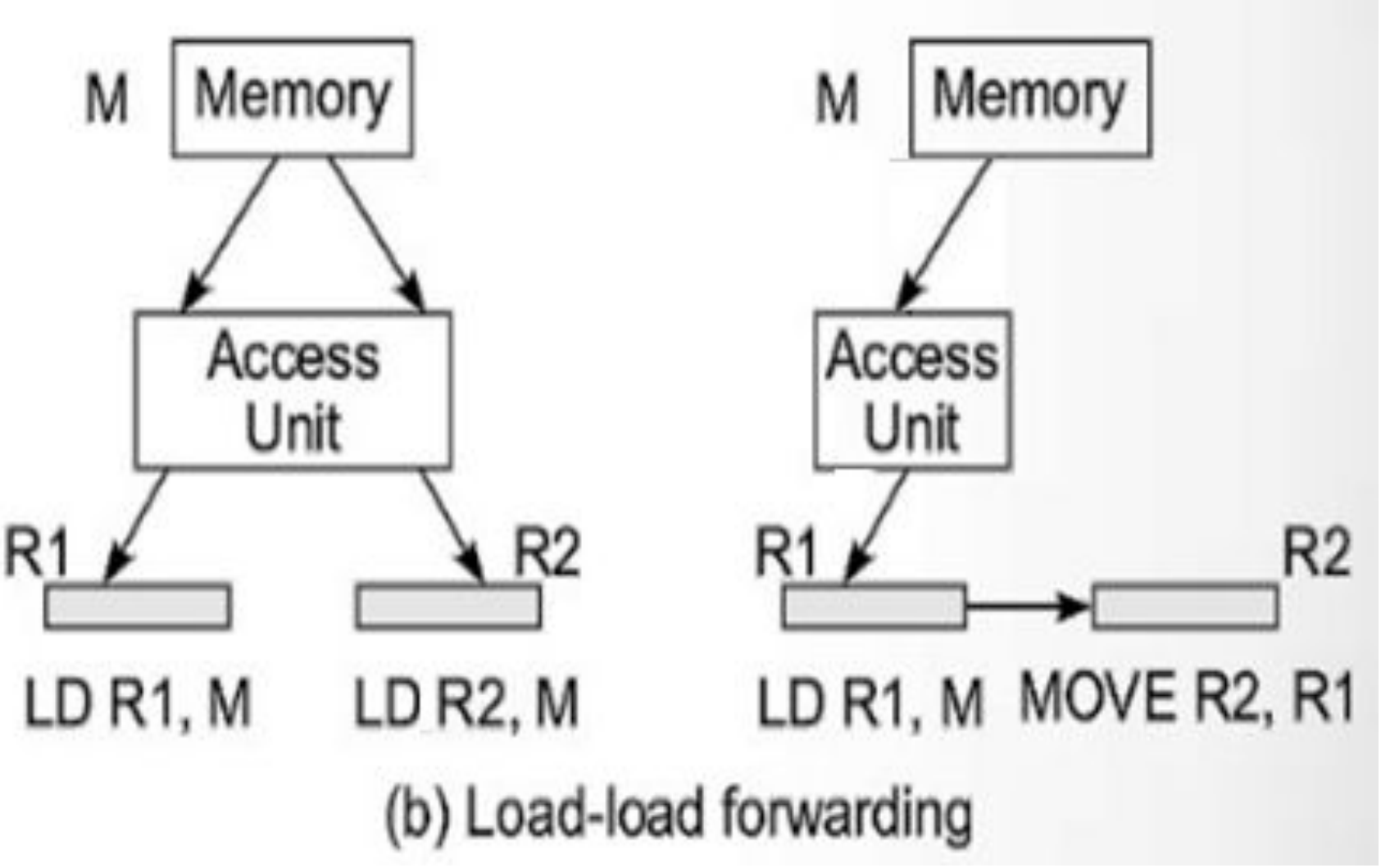


A pipelined processor with multiple functional units and distributed reservation stations supported by tagging (Courtesy of G. Sohi; reprinted with permission from *IEEE Transactions on Computers*, March 1990)

Internal Data Forwarding

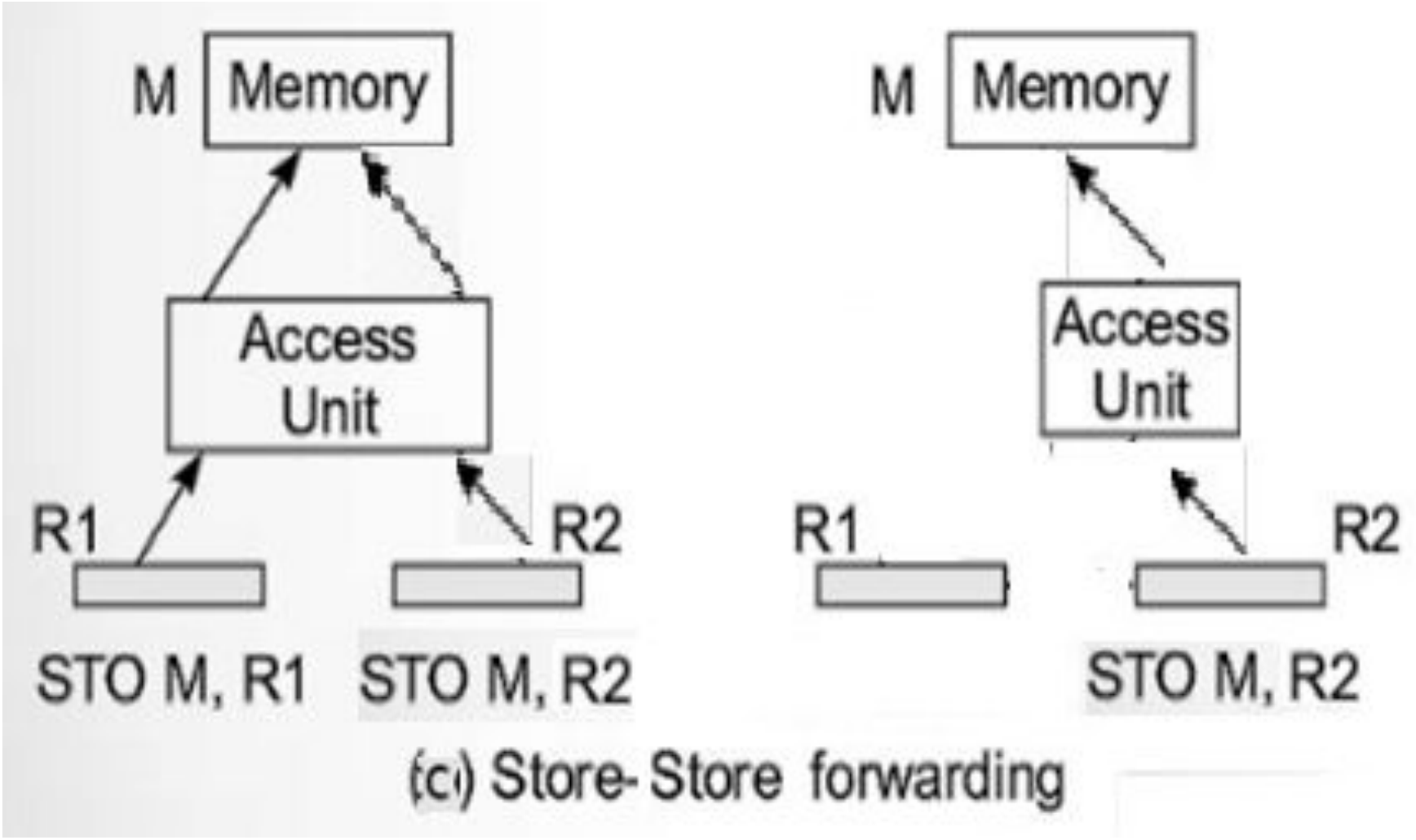


Internal Data Forwarding

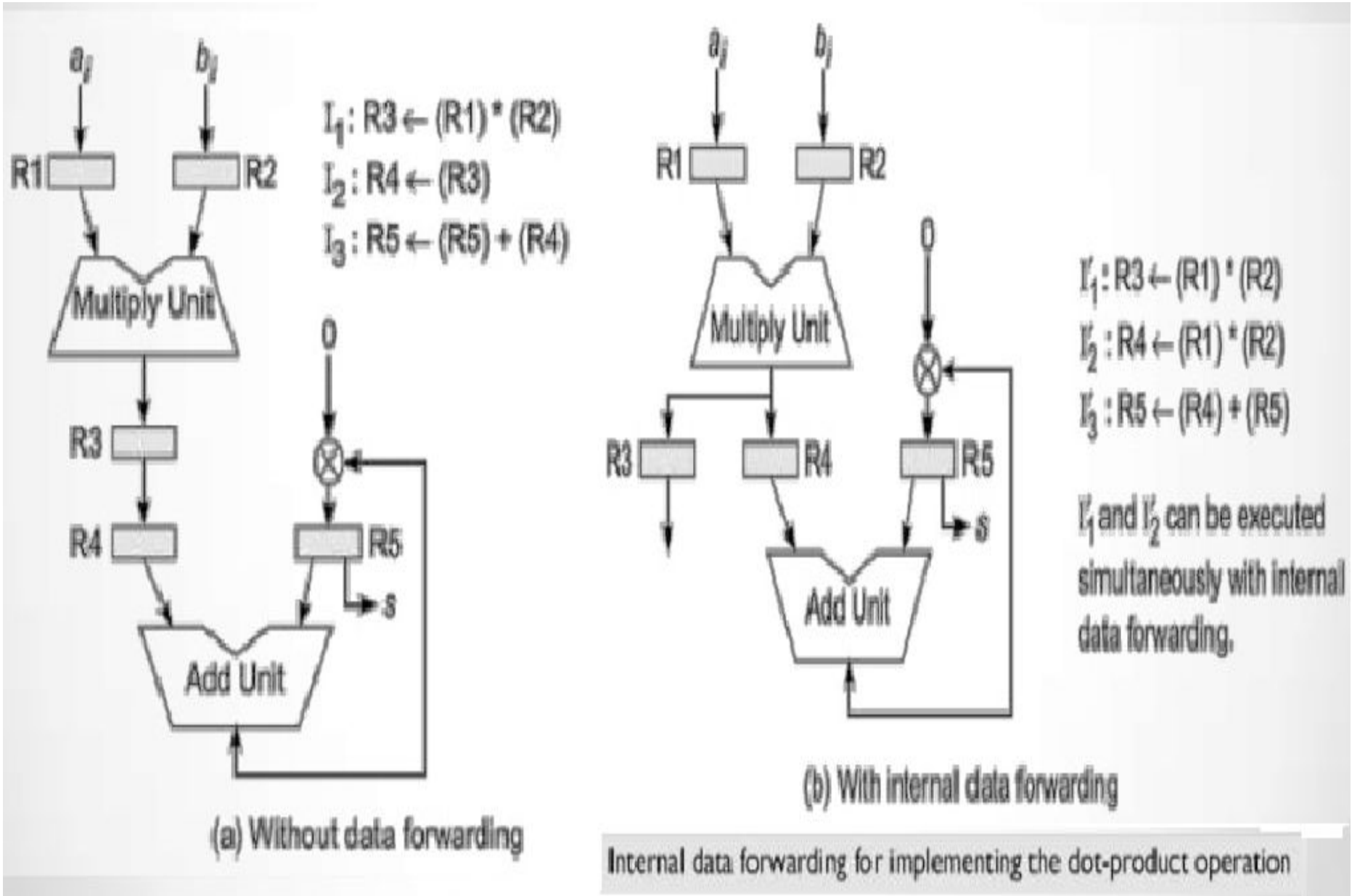


(b) Load-load forwarding

Internal Data Forwarding



Example :Implementing the dot-product operation with internal data forwarding between a multiply unit and an add unit



Internal data forwarding for implementing the dot-product operation

Hazard Avoidance

- The read and write of shared variables by different instructions in a pipeline may lead to different results if these instructions are executed out of order
- Three types of logic hazards are possible
- Consider two instructions I and J
- Instruction J is assumed to logically follow instruction l according to program order.

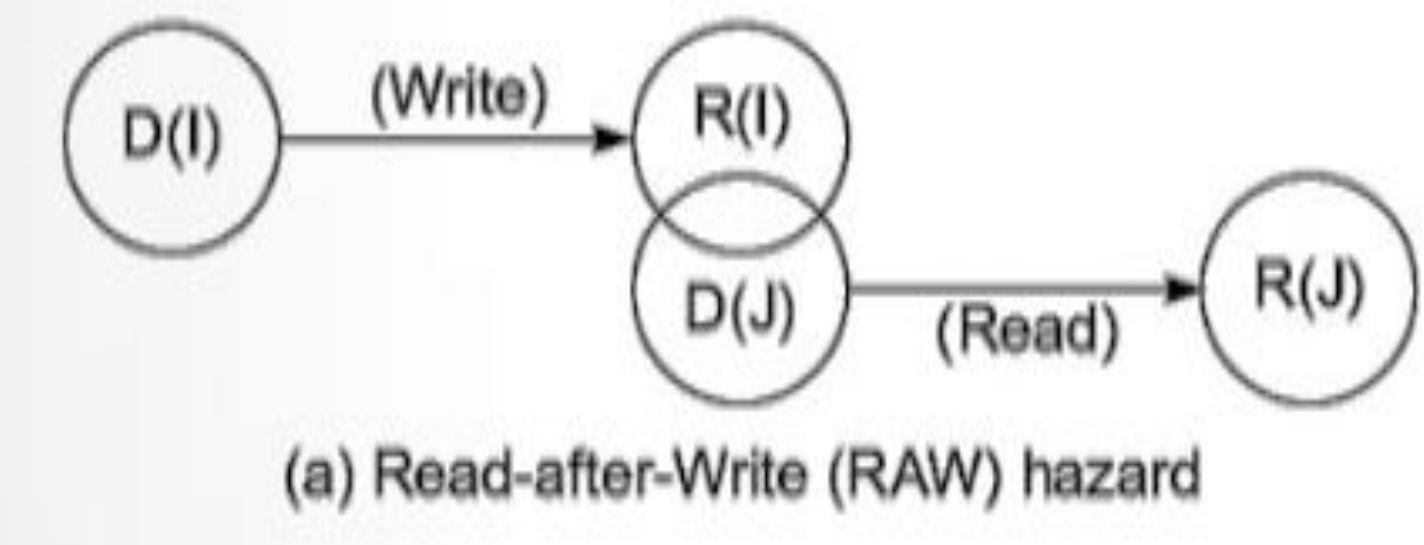
RAW (Read After Write)

j tries to read a source before i writes to it , so j incorrectly gets the old value;

Eg:

I: R2 <- R1 + R3

J: R4 <- R2 + R3



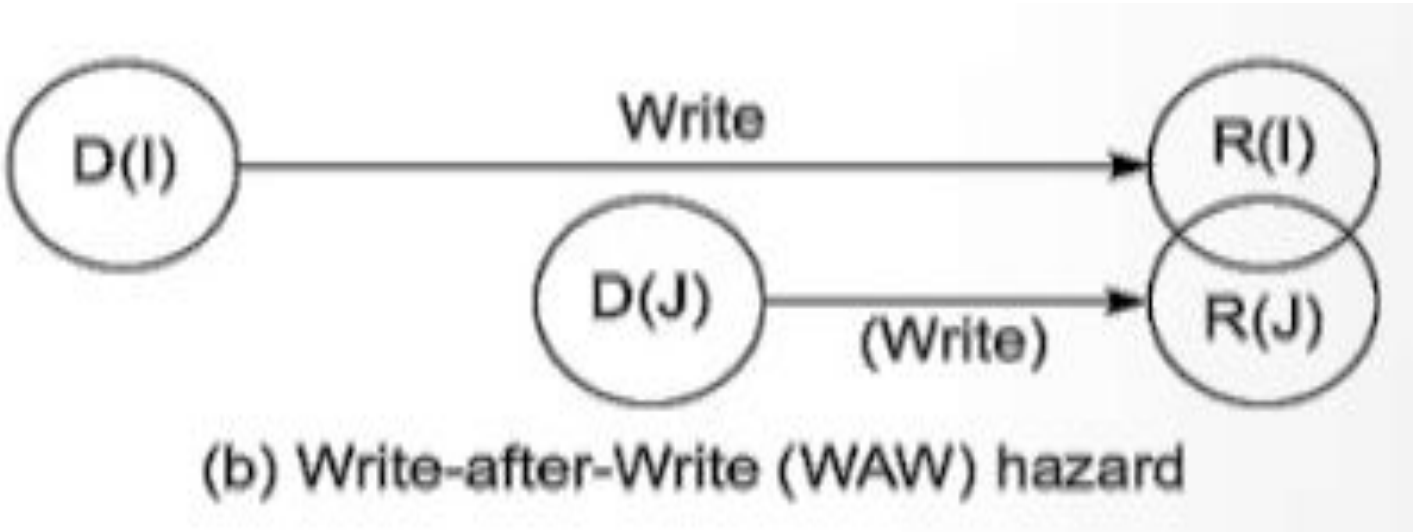
WAW (Write After Write)

j tries to write an operand before is written by I

Eg:

I: R2 <- R1 + R3

J: R2 <- R4 + R5



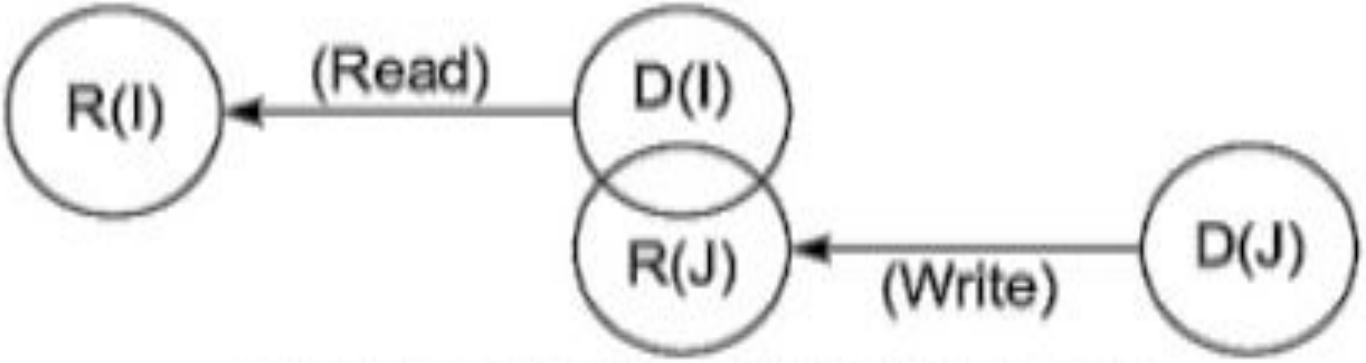
(b) Write-after-Write (WAW) hazard

WAR (Write After Read)

Eg:

I: R2 <- R1 + R3

J: R3 <- R4 + R5



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

- We use the notation D(I) and R(I) for the domain and range of an instruction I.
- The domain contains the input set.
- The range corresponds to the output set of instruction I

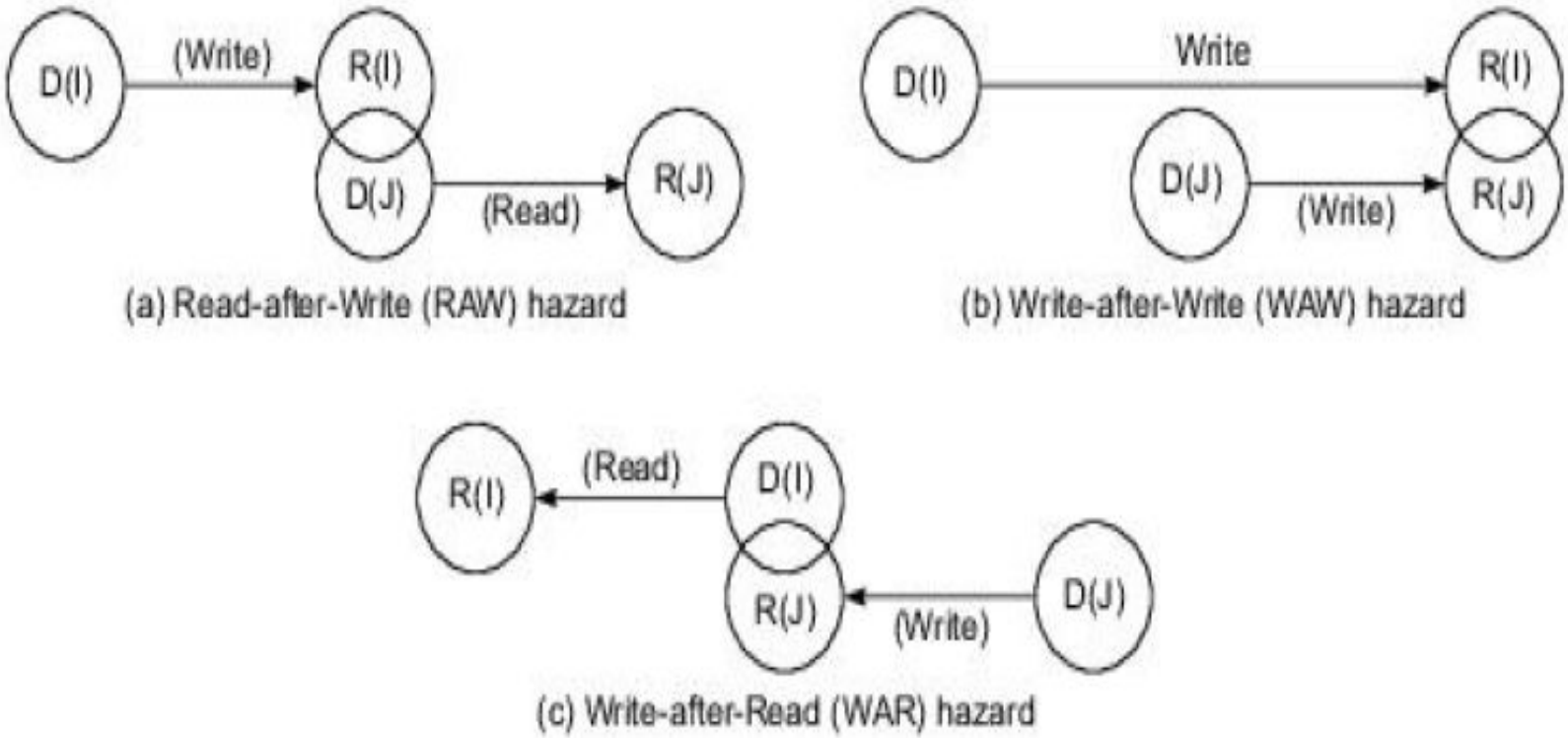


Fig. 6.15 Possible hazards between read and write operations in an instruction pipeline (instruction I is ahead of instruction J in program order)

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

- The RAW hazard corresponds to the flow dependence
- WAR to the antidependenee,
- WAW to the output dependence

Dynamic Instruction Scheduling

- Two types: static and dynamic

Static scheduling

- Interlocking can be resolved through a compiler-based static scheduling approach.
- A compiler or a postprocessor can be used to increase the separation between interlocked instructions

Stage delay:	Instruction:		
2 cycles	Add	R0, R1	/R0 ← (R0) + (R1)/
1 cycle	Move	R1, R5	/R1 ← (R5)/
2 cycles	Load	R2, M(α)	/R2 ← (Memory (α))/
2 cycles	Load	R3, M(β)	/R3 ← (Memory (β))/
3 cycles	Multiply	R2, R3	/R2 ← (R2) × (R3)/

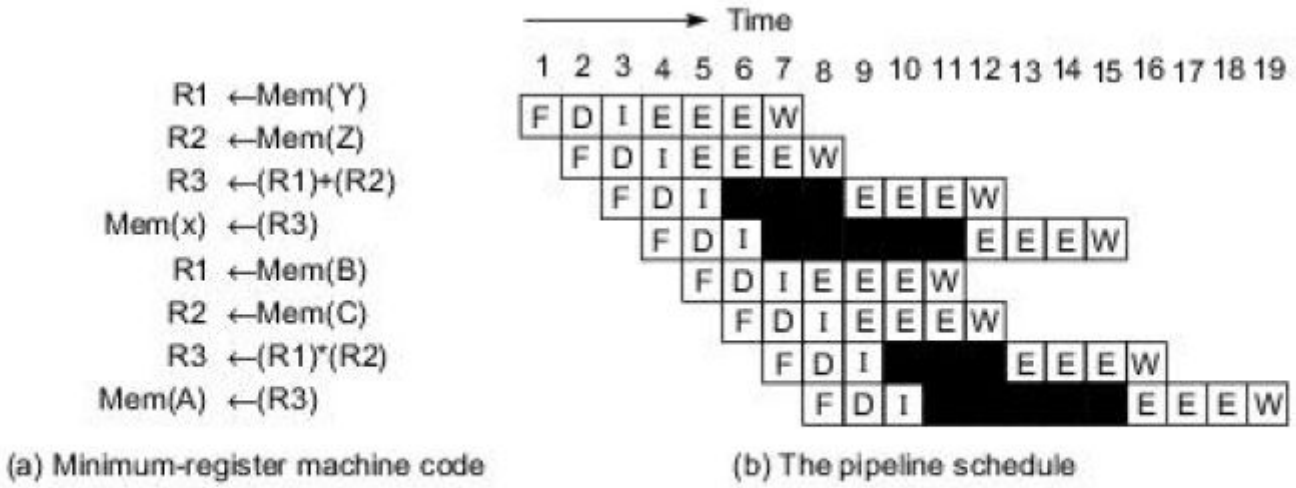
Load	R2, M(α)	2 to 3 cycles
Load	R3, M (β)	2 cycles due to overlapping
Add	R0, R1	2 cycles
Move	R1, R5	1 cycle
Multiply	R2, R3	3 cycles

- Through this code rearrangement, the data dependences and program semantics are preserved, and the multiply can be initiated without delay

Dynamic Instruction Scheduling

- Dynamic scheduling
  - **Tomasulo's algorithm**
  - **CDC Scoreboarding**

Example: Tomasulo's algorithm for dynamic instruction scheduling

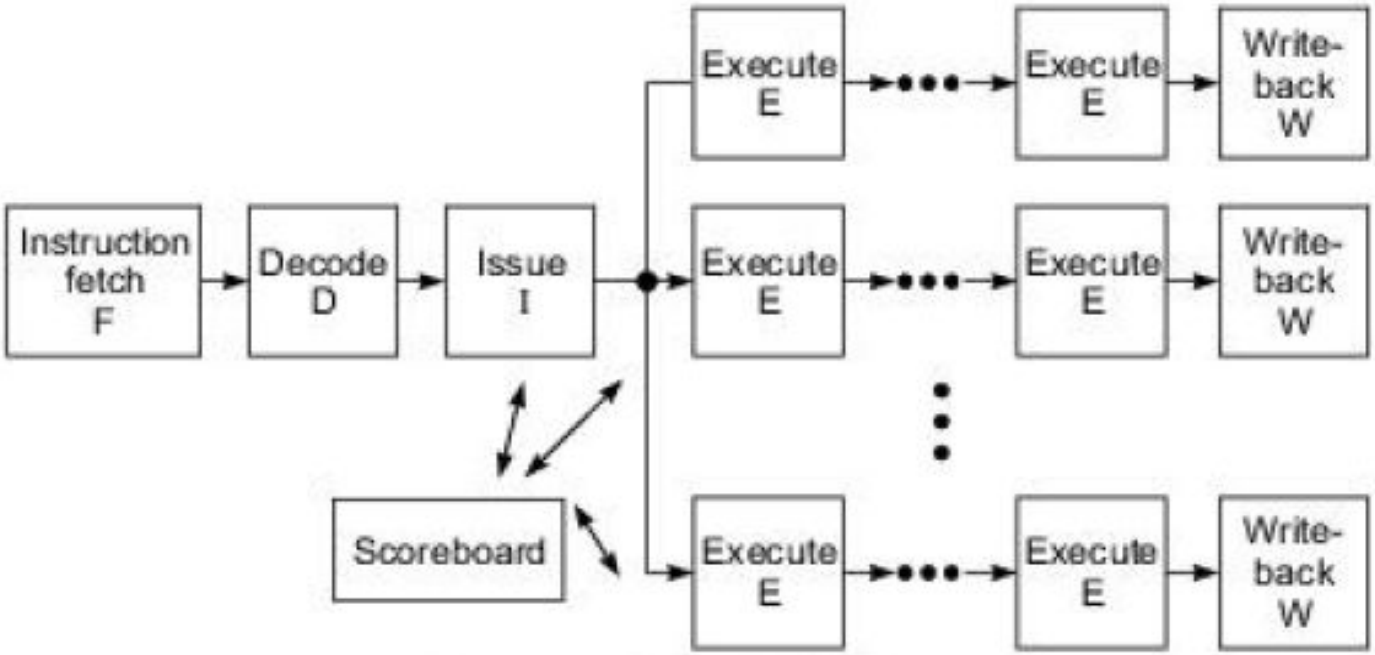


(a) Minimum-register machine code

(b) The pipeline schedule

Fig. 6.16 Dynamic instruction scheduling using Tomasulo's algorithm on the processor in Fig. 6.12 (Courtesy of James Smith; reprinted with permission from *IEEE Computer*, July 1989)

CDC Scoreboarding



(a) A CDC 6600-like processor

$$T_{eff} = k\tau + (n-1)\,\tau + pqnb\,\tau$$

Modifying Eq. 6.9, we define the following *effective pipeline throughput* with the influence of branching:

$$H_{eff} = \frac{n}{T_{eff}} = \frac{nf}{k+n-1+pqnb} \tag{6.12}$$

When $n \to \infty$, the tightest upper bound on the effective pipeline throughput is obtained when $b = k-1$:

$$H_{eff}^{*} = \frac{f}{pq\,(k-1)+1} \tag{6.13}$$

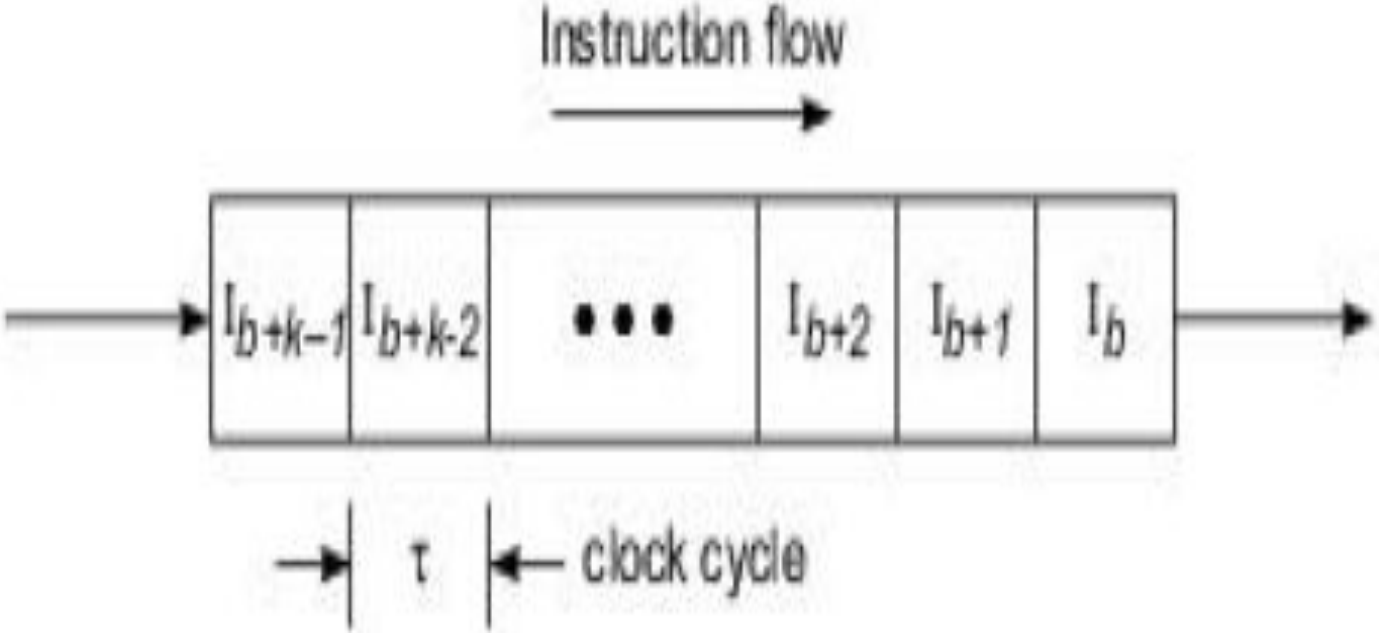
When $p = q = 0$ (no branching), the above bound approaches the maximum throughput $f = 1/\tau$, same as in Eq, 6.2. Suppose $p = 0.2$, $q = 0.6$, and $b = k-1 = 7$. We define the following *performance degradation factor*:

$$D = \frac{f - H_{eff}^{*}}{f} = 1 - \frac{1}{pq(k-1)+1} = \frac{pq(k-1)}{pq(k-1)+1} = \frac{0.84}{1.84} = 0.46 \tag{6.14}$$

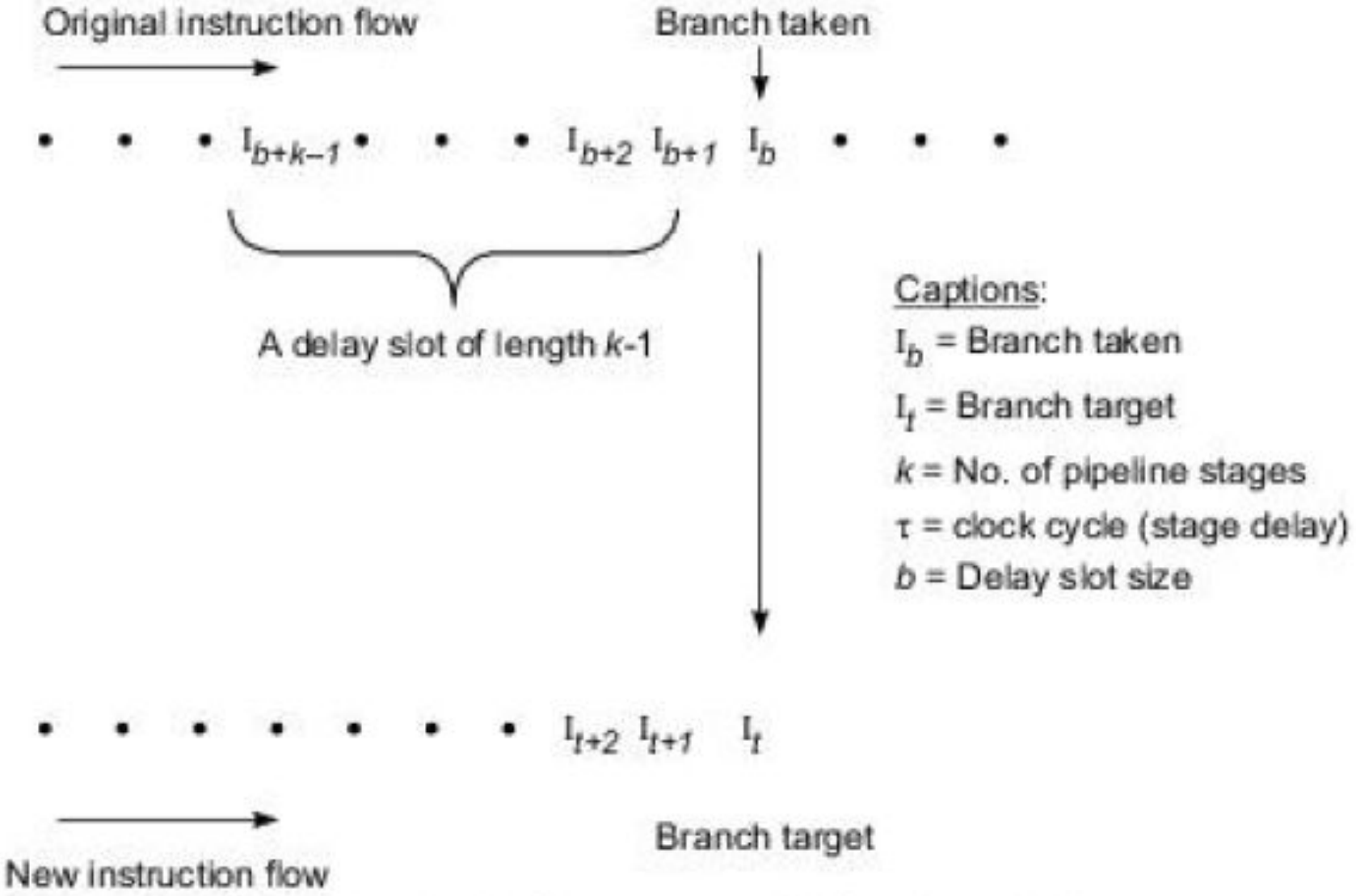
Branch Handling Techniques

Effect of Branching

- The action of fetching a non sequential or remote instruction after a branch instruction is called **branch taken**.
- The instruction to be executed after a branch taken is called a **branch target**
- The number of pipeline cycles wasted between a branch taken and the fetching of its branch target is called the **delay slot** denoted by b
- $0 \leq b \leq k - 1$, where k is the number of pipeline stages



(a) A *k*-stage pipeline



(b) An instruction stream containing a branch taken

Branch Prediction

- Branch can be predicted either based on branch code types statically or based on branch history during program execution
- The static prediction direction (taken or not taken) can even be wired into the processor

- A dynamic branch strategy works better because it uses recent branch history to predict whether or not the branch will be taken next time when it occurs.

Branch Prediction

Loop

…..

…..

Bne $1,$2,loop

exit

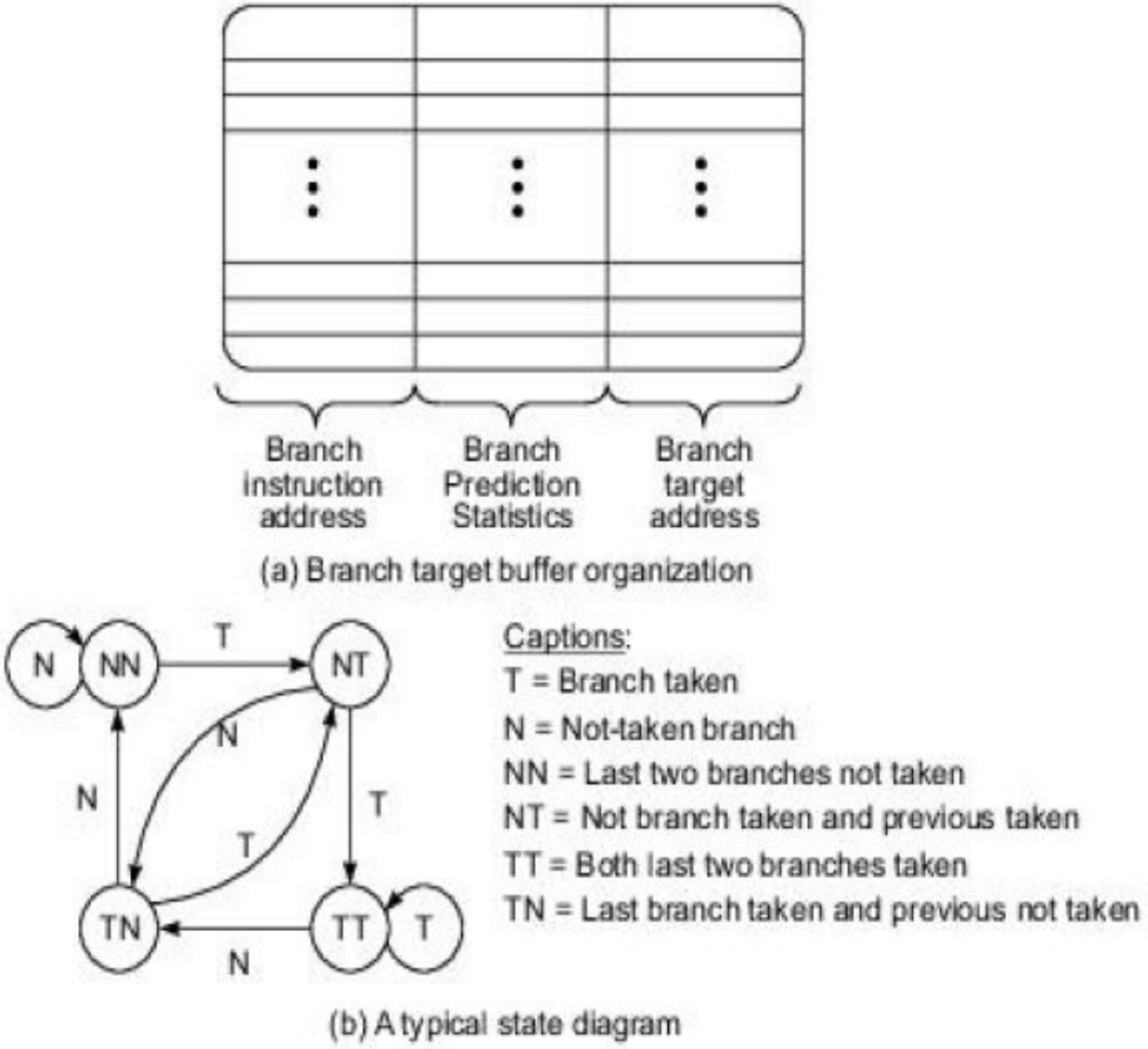
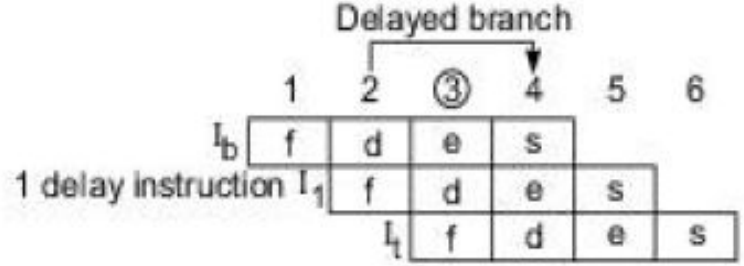


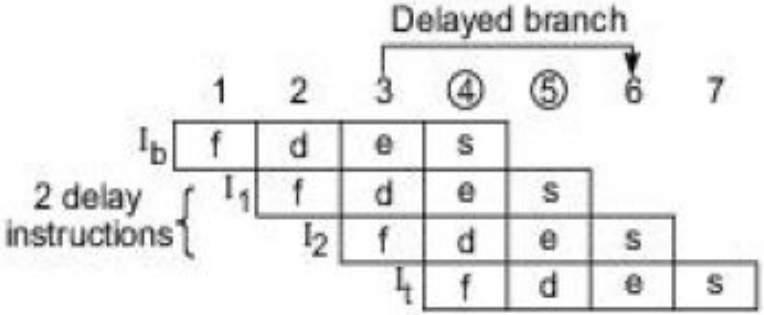
Fig. 6.19 Branch history buffer and a state transition diagram used in dynamic branch prediction (Courtesy of Lee and Smith, *IEEE Computer*, 1984)

Delayed Branches

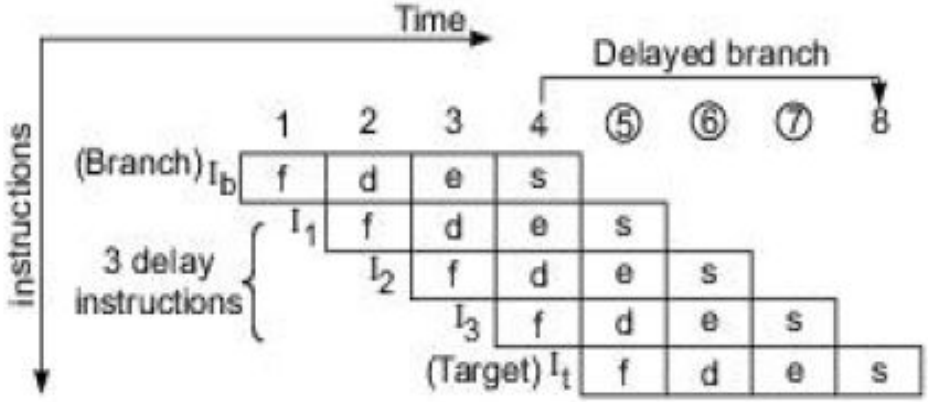
- The idea was originally used to reduce the branching penalty in coding microinstructions
- A delayed branch of d cycles allows at most d-1 useful instructions to be executed following the branch taken.
- The execution of these instructions should be independent of the outcome of the branch instruction.
- Otherwise, a zero branching penalty cannot be achieved.



(a) A delayed branch for 2 cycles when the branch condition is resolved at the decode stage



(b) A delayed branch for 3 cycles when the branch condition is resolved at the execute stage



(c) A delayed branch for 4 cycles when the branch condition is resolved at the store stage

Fig. 6.20 The concept of delayed branch by moving independent instructions or NOP fillers into the delay slot of a four-stage pipeline

- Code motion across branches can be used to achieve a delayed branch

ARITHMETIC PIPELINE DESIGN

- Pipelining techniques can be applied to speed up numerical arithmetic computations
- Finite precision
- Overflow or underflow

Fixed-Point Operations

- Fixed-point numbers are represented internally in machines in sign-magnitude, one's complement or in two's complement notation.
- Add, subtract , multiply, divide are the four primitive arithmetic operations.

- **Floating Point Number**:

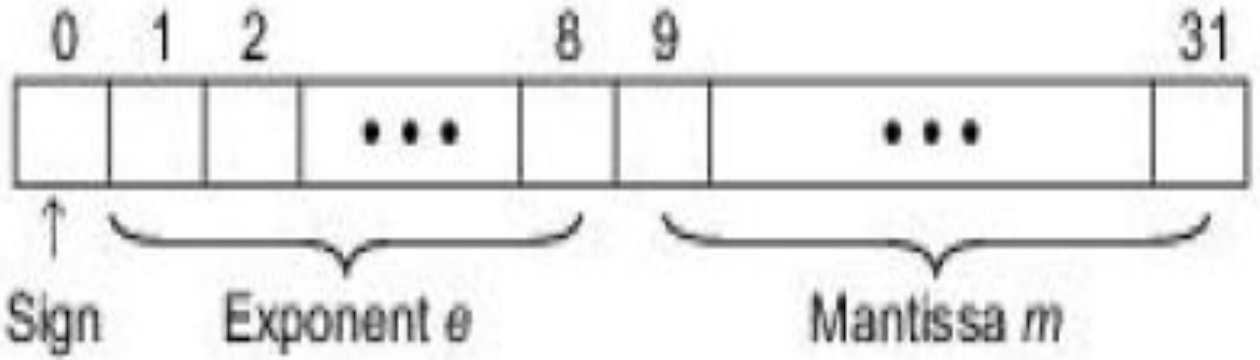
- A floating-point number X is represented by a pair (m,e)

  m:mantissa (or fraction)

  e:exponent with an implied base (or radix)

- The algebraic value is represented as X = m X (r^e)

Example : The IEEE 154 floating-point standard



Floating-Point Operations

$$X + Y = (m_x \times 2^{e_x - e_y} + m_y) \times x^{e_y}$$

$$X - Y = (m_x \times 2^{e_x - e_y} - m_y) \times x^{e_y}$$

$$X \times Y = (m_x \times m_y) \times 2^{e_x + e_y}$$

$$X \div Y = (m_x \div m_y) \times 2^{e_x - e_y}$$

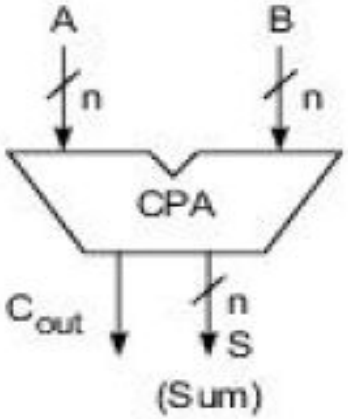
Static Arithmetic Pipelines

- The arithmetic/logic units perform fixed-point and floating-point operations separately
- The fixed-point unit is also called the integer unit

- Arithmetic or logical shifts can be easily implemented with shift registers
- High-speed addition requires either the use of a carry-propagation adder (CPA) or carry save adder(CSA)

e.g. n=4

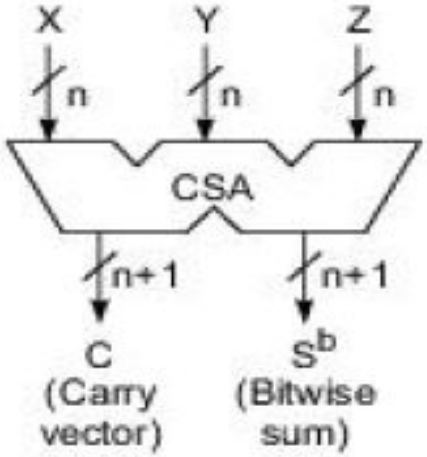
```
    A =   1 0 1 1
+)  B =   0 1 1 1
-------------------
    S = 1 0 0 1 0 = A + B
```



(a) An *n*-bit carry-propagate adder (CPA) which allows either carry propagation or applies the carry-lookahead technique

e.g. n=4

```
    X =   0 0 1 0 1 1
    Y =   0 1 0 1 0 1
⊕   Z =   1 1 1 1 0 1
-------------------------
    S^b = 0 1 0 0 0 1 1
+)  C = 0 1 1 1 0 1 0
-------------------------
    S = 1 0 1 1 1 1 1 = S^b+C = X+Y+Z
```



(b) An *n*-bit carry-save adder (CSA), where S^b is the bitwise sum of X, Y, and Z, and C is a carry vector generated without carry propagation between digits

Fig. 6.22 Distinction between a carry-propagate adder (CPA) and a carry-save adder (CSA)

Multiply Pipeline Design

- Consider as an example the multiplication of two 3-bit integers A X B = P, where P is the 16- bit product

$$P = A \times B = P_0 + P_1 + P_2 + \cdots + P_7,$$

where X and + are arithmetic multiply and add operations

							1	0	1	1	0	1	0	1	=	A
						x)	1	0	0	1	0	0	1	1	=	B
							1	0	1	1	0	1	0	1	=	P_0
						1	0	1	1	0	1	0	1	0	=	P_1
					0	0	0	0	0	0	0	0	0	0	=	P_2
				0	0	0	0	0	0	0	0	0	0	0	=	P_3
			1	0	1	1	0	1	0	1	0	0	0	0	=	P_4
		0	0	0	0	0	0	0	0	0	0	0	0	0	=	P_5
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	=	P_6
+) 1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	=	P_7
0 1	1	0	0	1	1	1	1	1	1	1	0	1	1	1	=	P

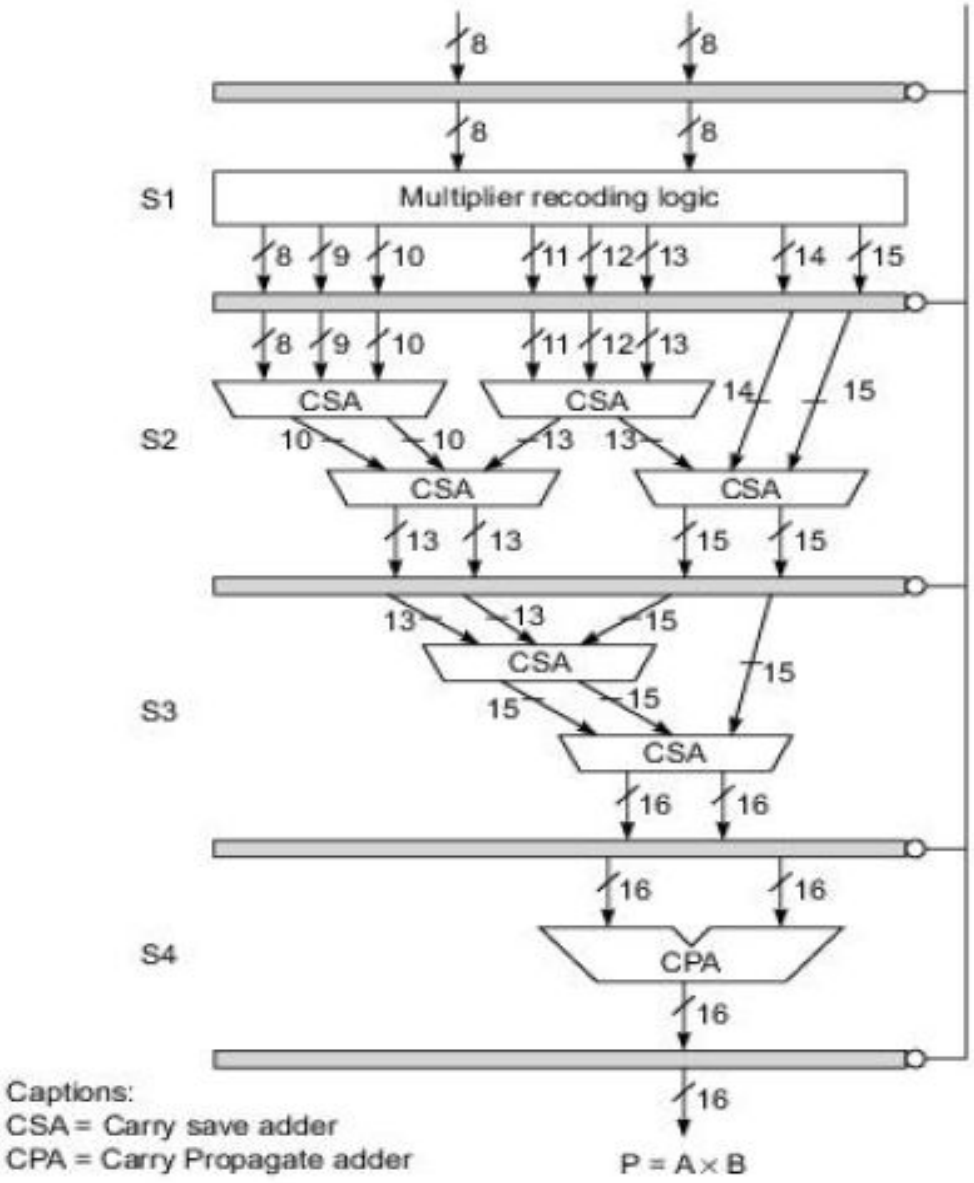


Fig. 6.23 A pipeline unit for fixed-point multiplication of 8-bit integers (The number along each line indicates the line width.)

Example : The floating-point unit in the Motorola MC68040

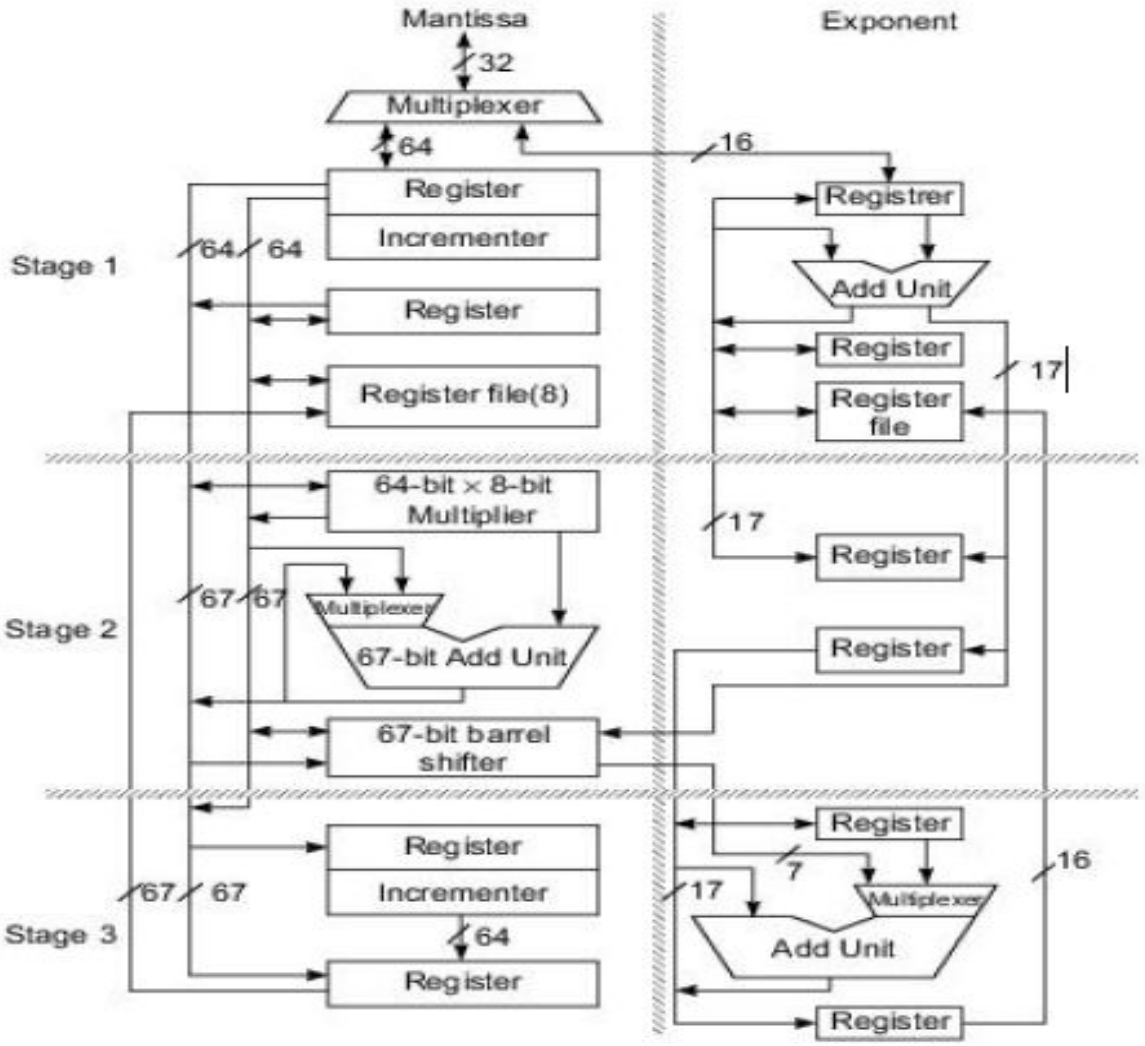


Fig. 6.24 Pipelined floating-point unit of the Motorola MC68040 processor (Courtesy of Motorola, Inc., 1992)

Example :The IBM 360iModel 91 floating-point unit design

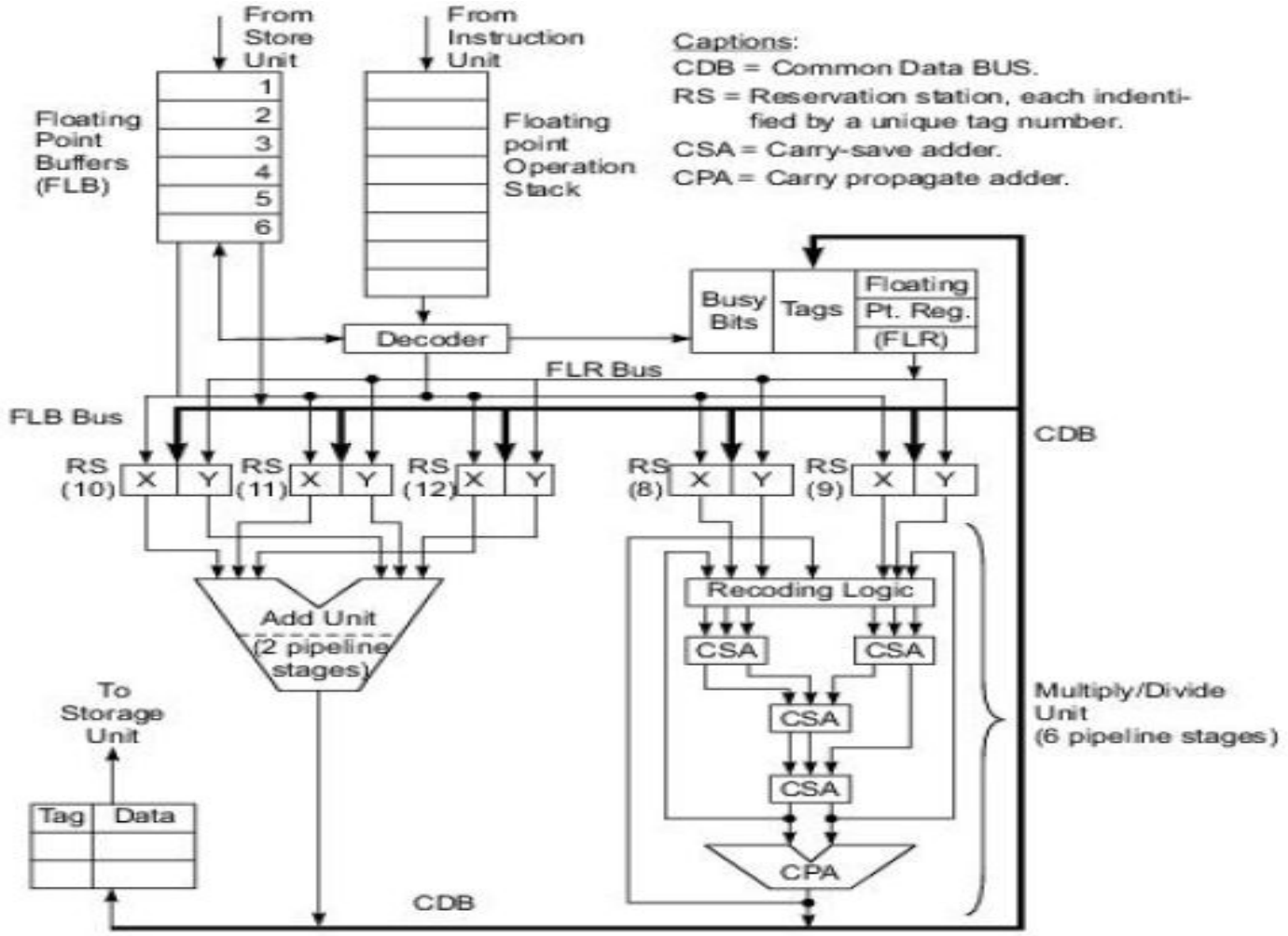


Fig. 6.25 The IBM 360 Model 91 floating-point unit (Courtesy of IBM Corporation, 1967)

- Floating-point execution unit in Model 91 consisted of two separate functional pipelines the add unit and the multiply /divide unit
- Operands may also come from the floating-point registers which were connected via the common data bus to the output bus
- The add unit allowed three pairs of operands to be loaded into three reservation stations.
- Only one pair could be used at a time.
- The use of these reservation stations made the add unit behave like three virtual functional units.

- Every source of an input operand was uniquely identified with a 4-bit tag.
- Every destination of an input operand had an associated tag register that held the tag naming the source of data if the destination was busy.
- Through this register tagging technique, operands / results could be directly passed among the virtual functional units
- When data is generated by a source, it passes its identification and the data onto the common data bus.
- When the source tag matches, the destination takes in the data from the bus

Multifunctional Arithmetic Pipelines

- Static arithmetic pipelines are designed to perform a fixed function and are thus called unifunctional.
- Multifunctional-pipeline can perform more than one function
- Multifunctional pipeline can be either static or dynamic
- Static pipelines perform one function at a time, but different functions can be performed at different times.
- A dynamic pipeline allows several functions to be performed simultaneously through the pipeline, as long as there are no conflicts in the shared usage of pipeline stages.

Example :The TI/ASC arithmetic processor design

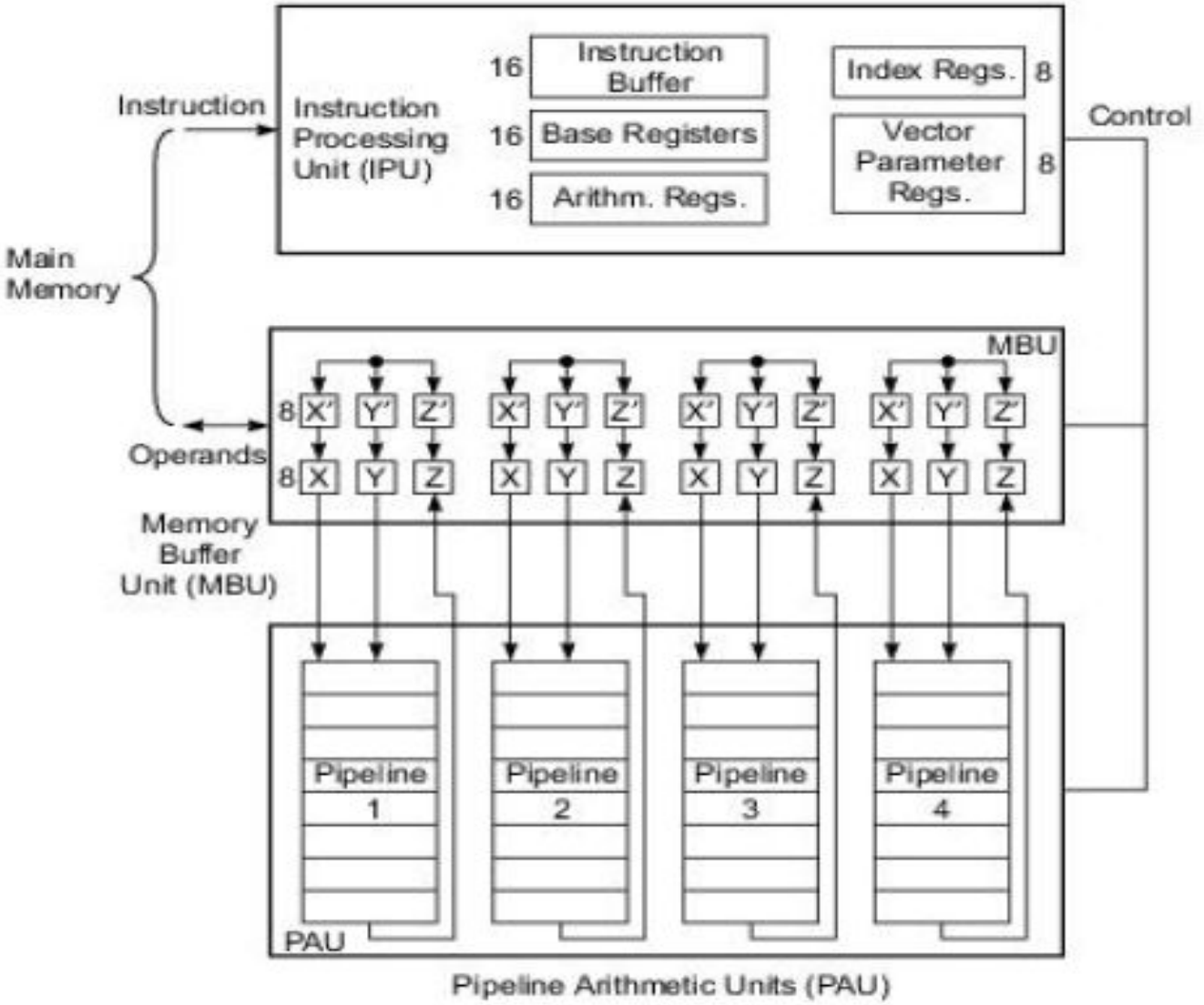
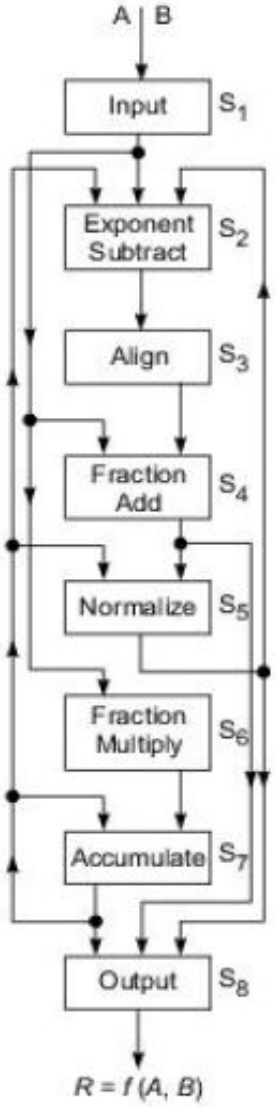
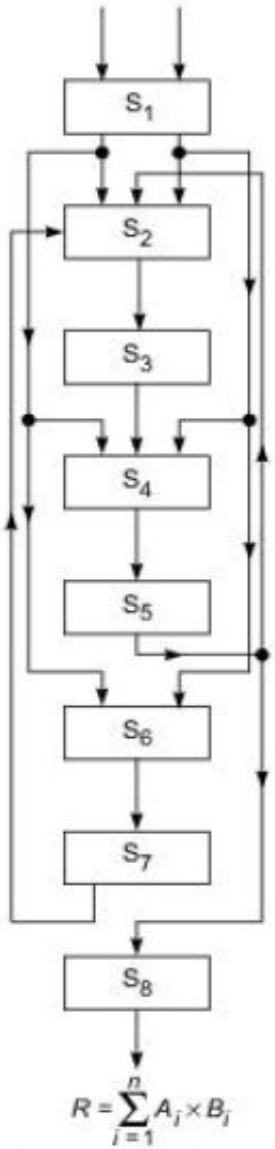


Fig. 6.26 The architecture of the TI Advanced Scientific Computer (ASC) (Courtesy of Texas Instruments, Inc.)



(a) Pipeline stages and interconnections

(b) Fixed-point multiplication



(c) Floating-point dot product

SUPERSCALAR PIPELINE DESIGN

- **Pipeline Design Parameter**

The instruction level parallelism(ILP) is the maximum number of instructions that can be simultaneously executed in the pipeline.

SUPERSCALAR PIPELINE DESIGN

Table 6.1 *Design Parameters for Pipeline Processors*

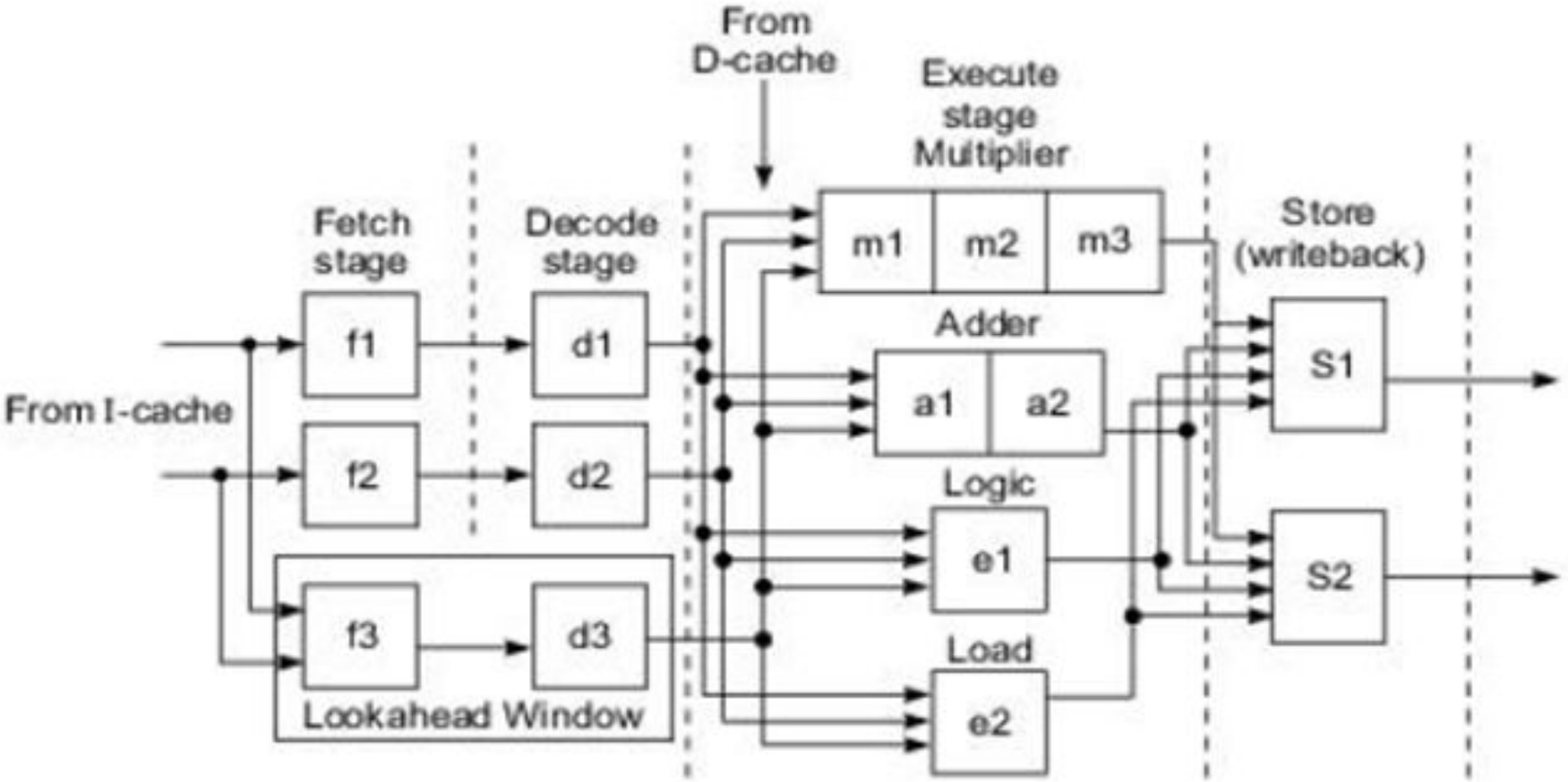
Machine type	*Scalar base machine of k pipeline stages*	*Superscalar machine of degree m*
Machine pipeline cycle	1 (base cycle)	1
Instruction issue rate	1	*m*
Instruction issue latency	1	1
Simple operation latency	1	1
ILP to fully utilize the pipeline	1	*m*

Note: All timing is relative to the base cycle for the scalar base machine. ILP: Instruction level parallelism.

Superscalar Pipeline Structure

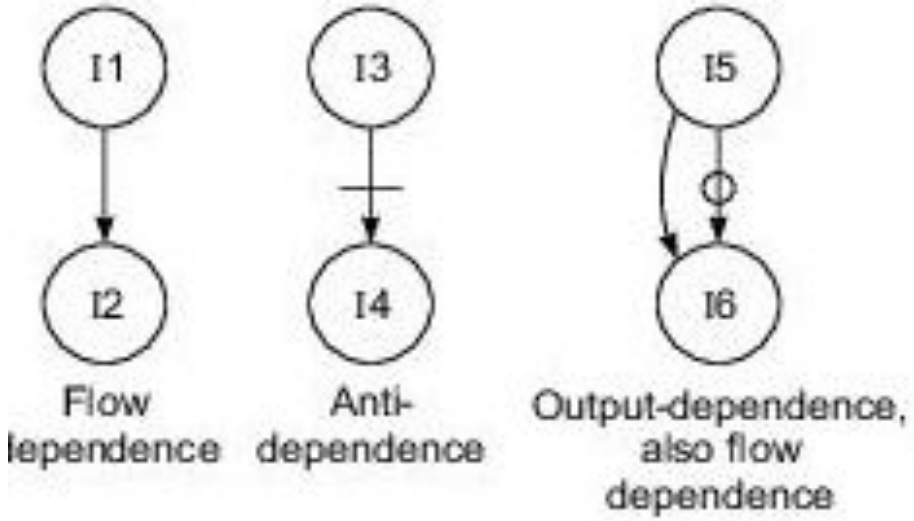
- In an *m*-issue superscalar processor, the instruction decoding and execution resources are increased to form effectively *m* pipelines operating concurrently.

Dual-pipeline superscalar processor



Program order

```
I1.  Load  R1,  A     /R1 ← Memory (A)/
I2.  Add   R2,  R1    /R2 ← (R2) + (R1)/
I3.  Add   R3,  R4    /R3 ← (R3) + (R4)/
I4.  Mul   R3,  R5    /R4 ← (R4) * (R5)/
I5.  Comp  R6         /R6 ← (R6)/
I6.  Mul   R6,  R7    /R6 ← (R6) * (R7)/
```



Data Dependences

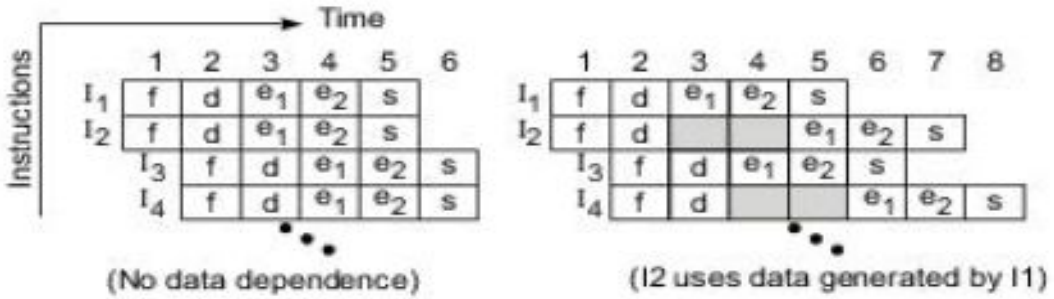
- A dependence graph is drawn to indicate the relationship among the instructions.
- Because the register content in R1 is loaded by I1 and then used by I2, we have flow dependence: I1—> I2
- Because the result in register R4 after executing I4 may affect the operand register R4 used by I3, we have antidependence:

I3 |—> I4

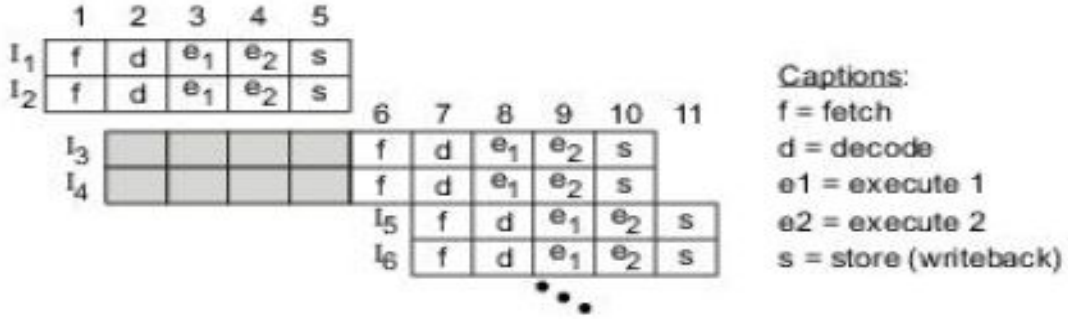
- To schedule instructions through one or more pipelines, these data dependences must not be violated.

Pipeline Stalling

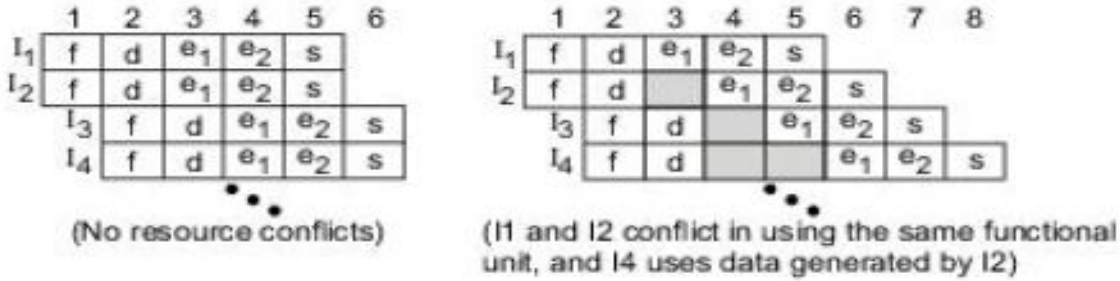
- This is a problem which may seriously lower pipeline utilization.
- Proper scheduling avoids pipeline stalling.
- Stalling can be caused by data dependencies or by resource conflicts among instructions already in the pipeline or about to enter the pipeline.



(a) Data dependence stalls the second pipeline in shaded cycles



(b) Branch instruction I2 causes a delay slot of length 4 in both pipelines



(c) Resource conflicts and data dependences cause the stalling of pipeline operations for some cycles

Fig. 6.29 Dependences and resource conflicts may stall one or two pipelines in a two-issue superscalar processor

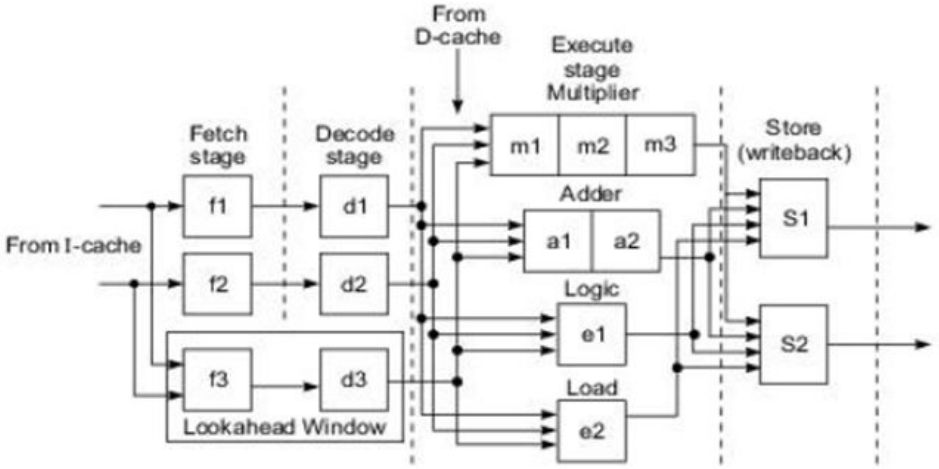
- A delay slot of four cycles results from a branch taken by I2 at cycle 5
- Therefore, both pipelines must be flushed before the target instructions I3 and I4 can enter the pipelines from cycle 6
- we show a combined problem involving both resource conflict and data dependence.
- Instructions Il and I2 need to use the same functional unit, and I2 -> I4 exists.

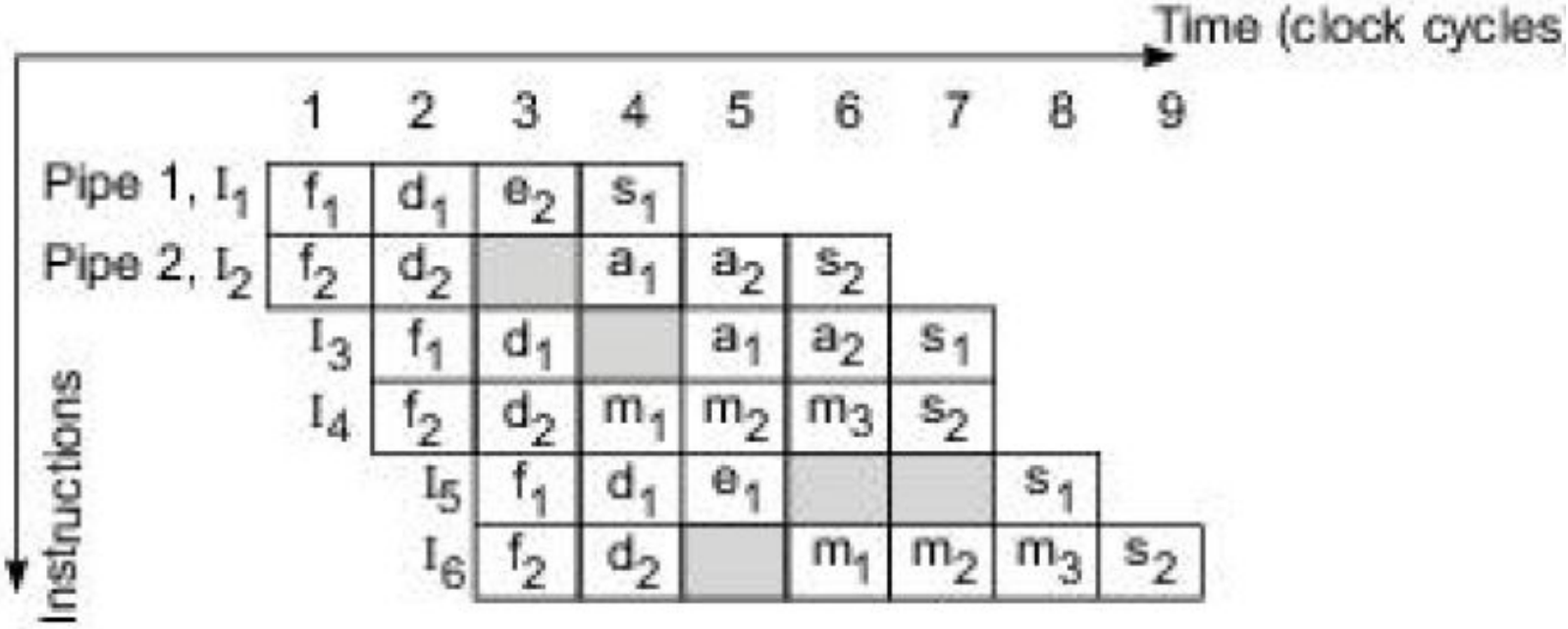
Superscalar Pipeline Scheduling

- Three scheduling policies are there.
- When instructions are issued in program order, we call it in-order - issue.
- When program order is violated, out-of-order issue is being practiced.
- If the instructions must be completed in program order, it is called in-order -completion.
- In-order issue may result in either in-order or out-of-order completion.

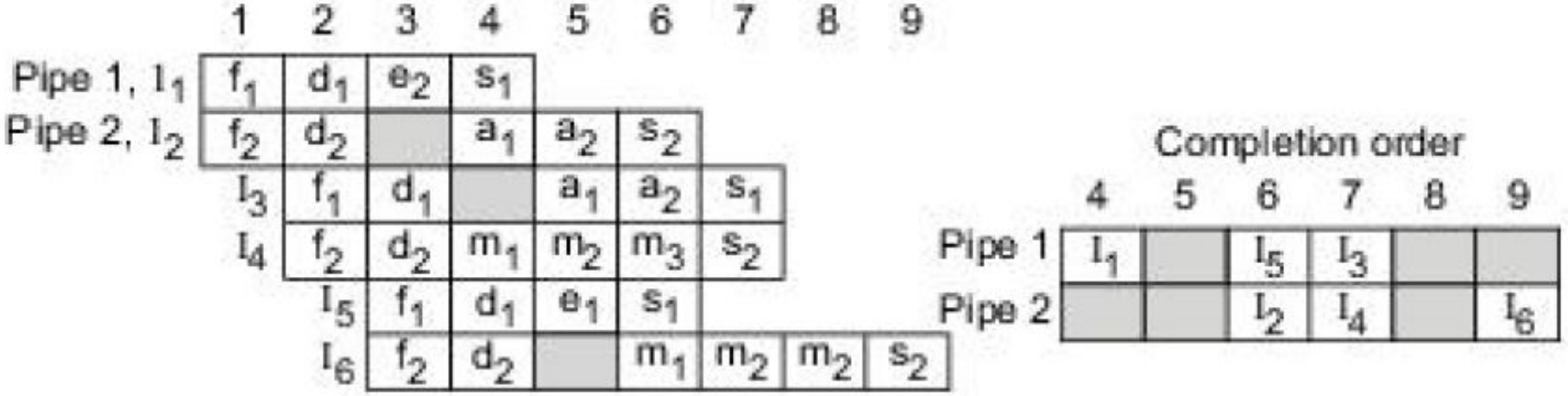
Program order

I1.	Load	R1,	A	/R1 ← Memory (A)/
I2.	Add	R2,	R1	/R2 ← (R2) + (R1)/
I3.	Add	R3,	R4	/R3 ← (R3) + (R4)/
I4.	Mul	R3,	R5	/R4 ← (R4) * (R5)/
I5.	Comp	R6		/R6 ← $(\overline{R6})$/
I6.	Mul	R6,	R7	/R6 ← (R6) * (R7)/

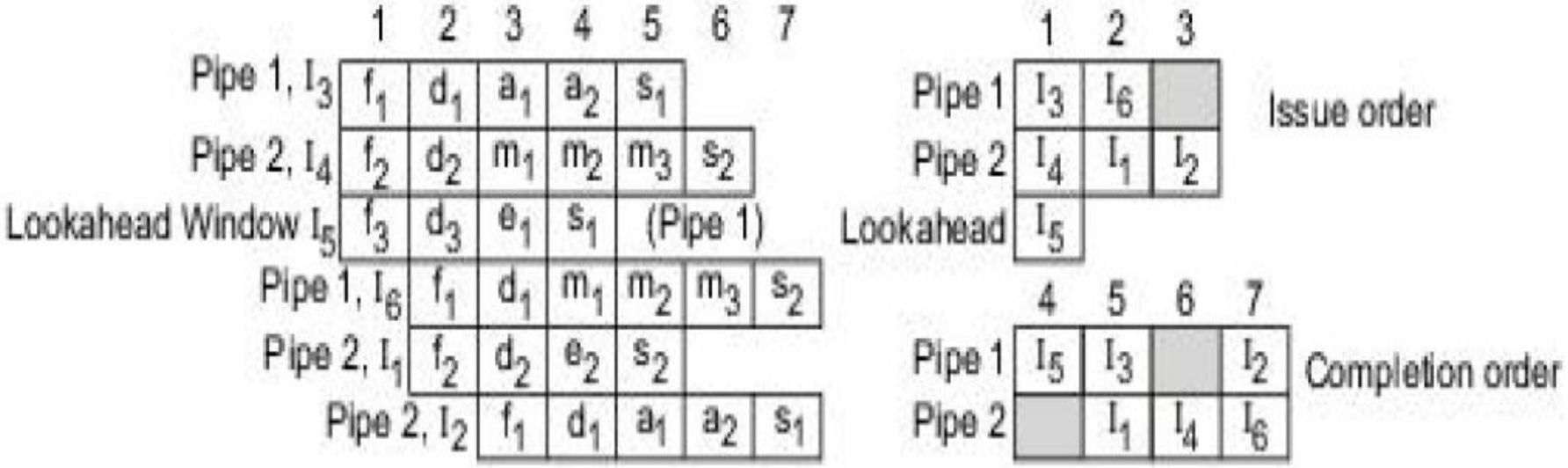




(a) In-order issue with in-order completion in nine cycles



(b) In-order issue and out-of-order completion in nine cycles



(c) Out-of-order issue and out-of-order completion in seven cycles using an instruction lookahead window in the recoding process

Motorola 88710 Architecture

- The Motorola 881 I0 was an early superscalar RISC processor.
- It combined the three-chip set, one CPU (B8100) chip and two cache (33200) chips, in a single-chip implementation
- The 88110 employed a symmetrical superscalar instruction dispatch unit.
- It allowed out-of-order instruction completion and some out-of-order instruction issue, and branch prediction with speculative execution past branches.

- The instruction set of the 88110 extended that of the 83100 in integer and floating-point operations.
- It added a new set of capabilities to support 3-D color graphics image rendering
- The 88110 had separate, independent instruction and data paths, along with split caches for instructions and data.
- The instruction cache was 8K-byte, 2-way set-associative with 128 sets, two blocks for each set, and 32 bytes per block.

Example : DEC Alpha 21064 superscalar architecture

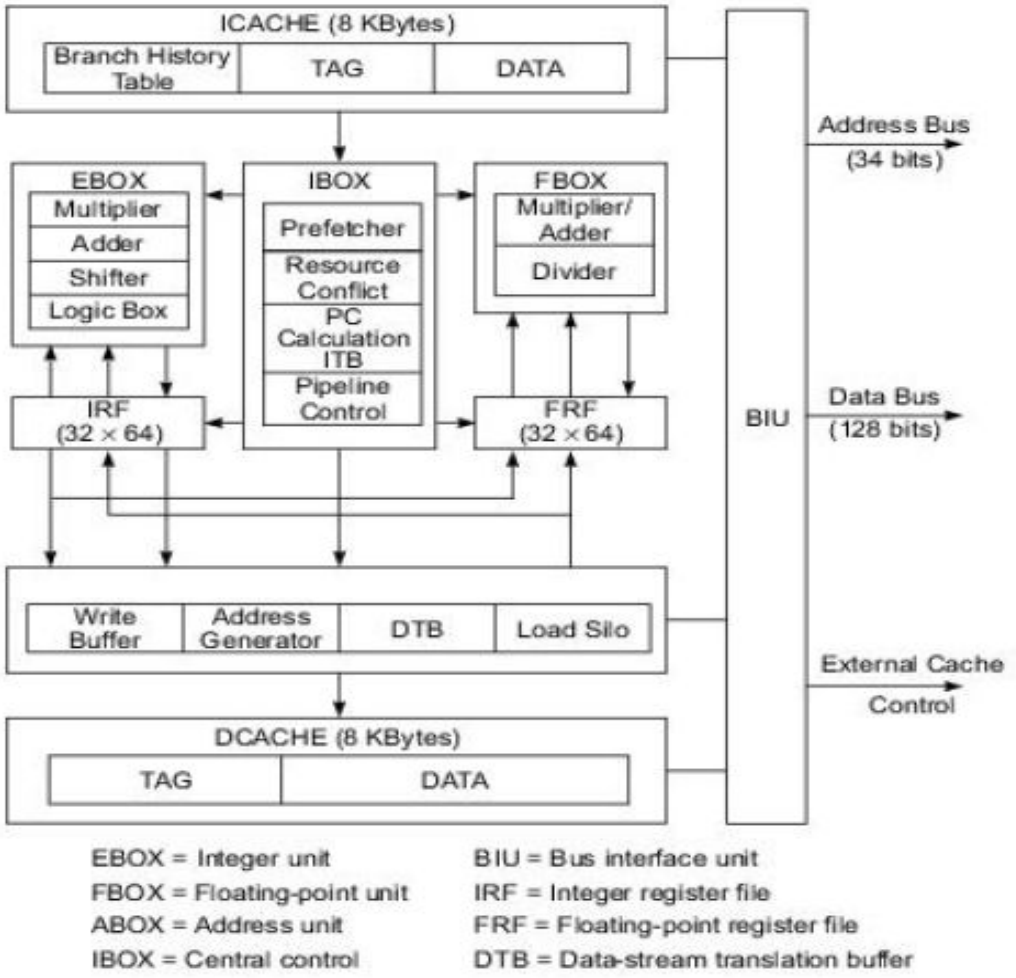


Fig. 6.31 Architecture of the DEC Alpha 21064 processor (Courtesy of Digital Equipment Corporation)

Superscalar Performance

- Estimate the ideal execution time of N independent instructions through the pipeline
- The time required by the scalar base machine is:

$$\mathbf{T(1,1) = K + N - 1 \text{ (Base cycles)}}$$

- The ideal execution time required by an m-issue superscalar machine is:

$$\mathbf{T(m,1) = K + ((N-m)/m) \text{ (Base cycles)}}$$

- The ideal speedup of the superscalar machine over the base machine is:

$$S(m,1) = \frac{T(1,1)}{T(m,1)} = \frac{N+k-1}{N/m+k-1} = \frac{m(N+k-1)}{N+m(k-1)}$$

As $N \to \infty$, the speedup limit $S(m, 1) \to m$, as expected.