MODULE-5

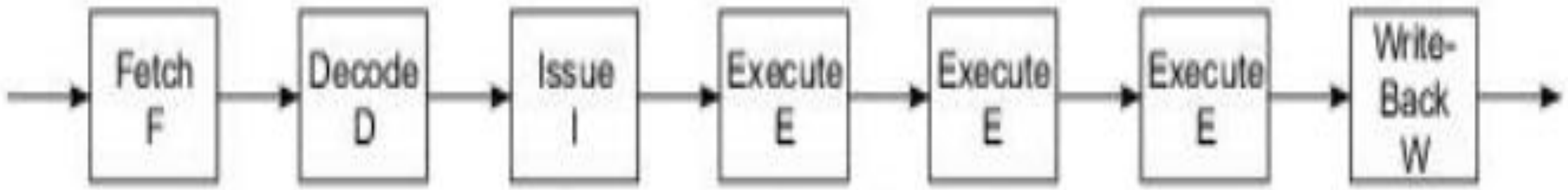
INSTRUCTION PIPELINE DESIGN

INTRODUCTION

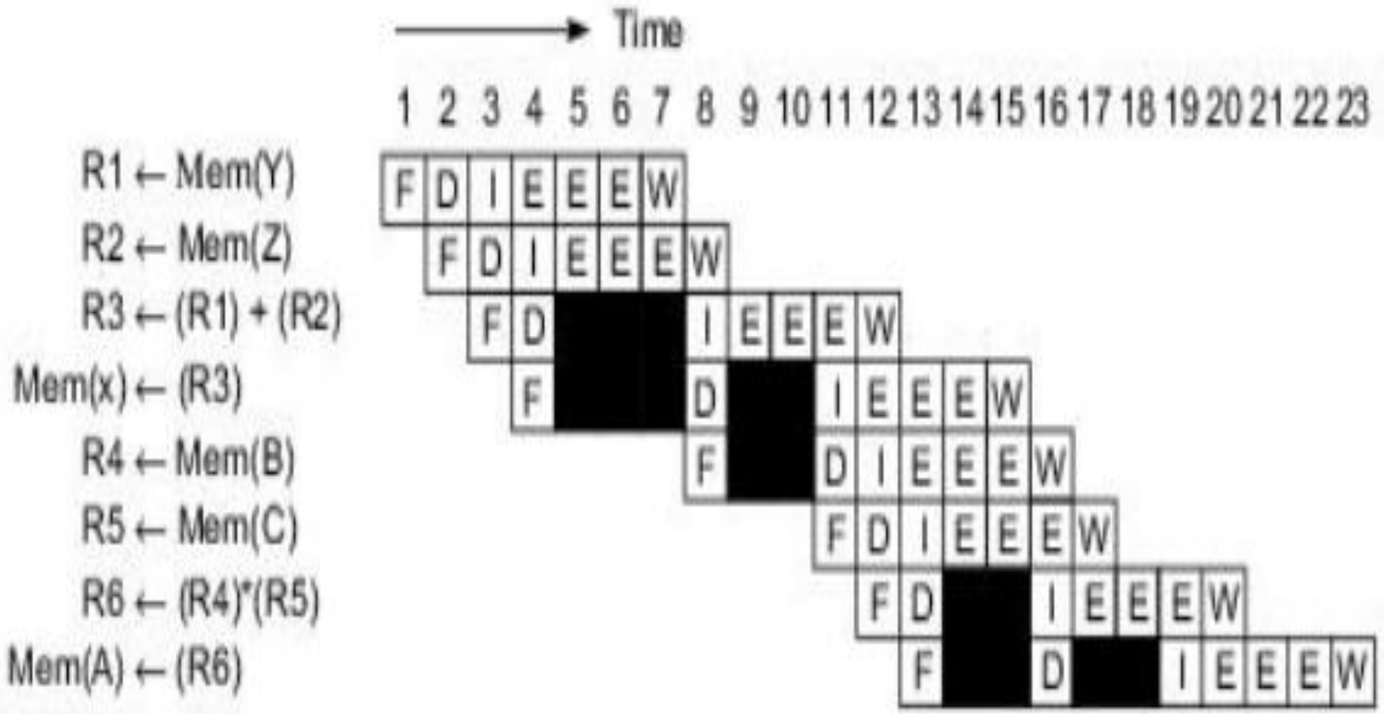
- Instruction execution involves sequence of operations like:-
  - Instruction fetch
  - Decode
  - Operand fetch
  - Execute
  - Write back

PIPELINED INSTRUCTION PROCESSING

- Fetch stage (F)
  - It fetches the instruction from a cache memory
  - Ideally one per cycle
- Decode stage (D)
  - It reveals the instruction function to be performed
  - Identifies the resources needed
  - The resources are:-
    - General purpose registers
    - Buses
    - Functional units
- Issue stage (I)
  - It reserves the resources
  - The operands are read from the registers during the issue stage

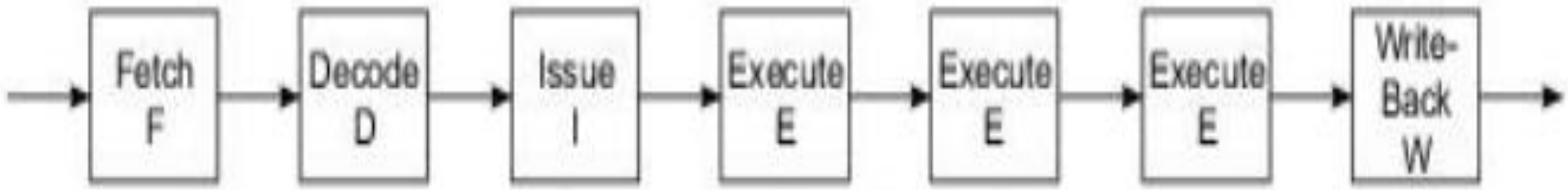


(a) A seven-stage instruction pipeline

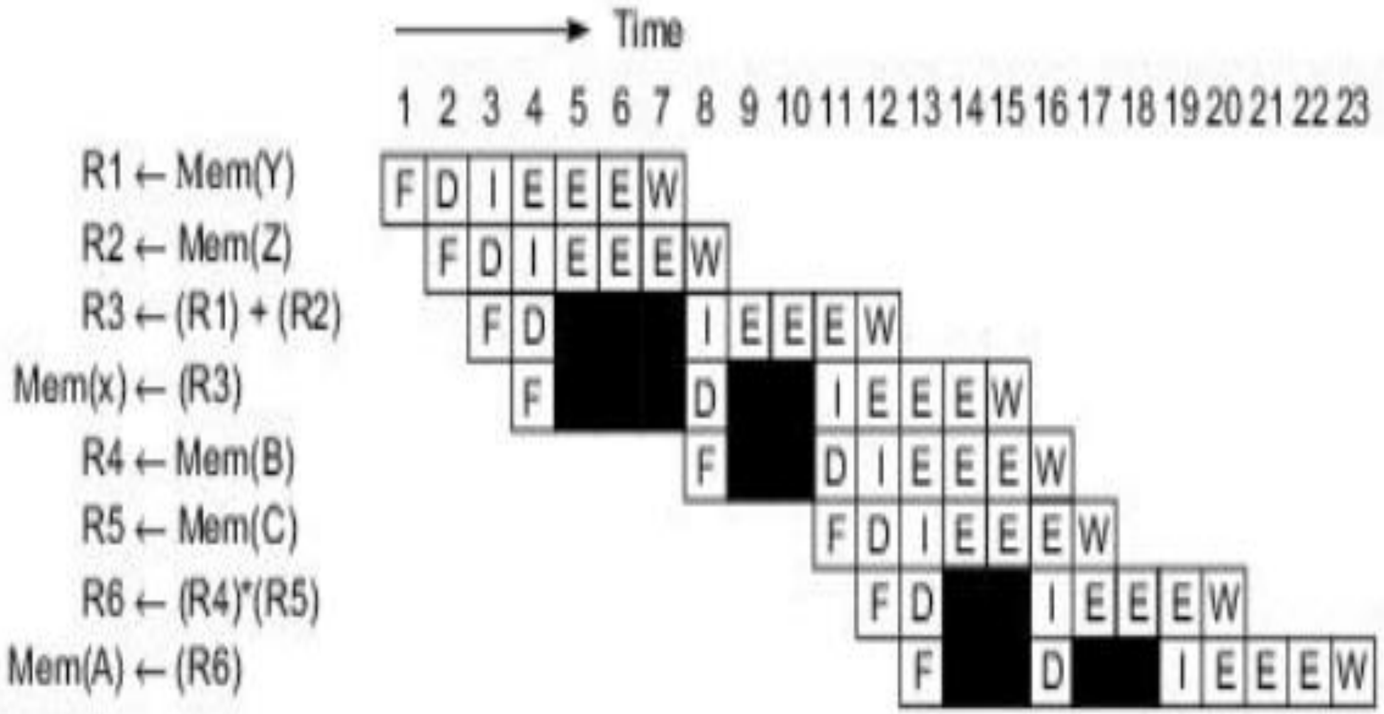


(b) In-order instruction issuing

- Execute stages (E)
  - Instructions are executed in one or several execute stages
- Write back stage (W)
  - This stage is used to write the results into the registers
  - Memory load or store operations are treated as part of execution



(a) A seven-stage instruction pipeline



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
R1 ← Mem(Y)	F	D	I	E	E	E	W																
R2 ← Mem(Z)		F	D	I	E	E	E	W															
R3 ← (R1) + (R2)			F	D	■	■	■	I	E	E	E	W											
Mem(x) ← (R3)				F	■	■	■	D	■	■	I	E	E	E	W								
R4 ← Mem(B)								F	■	■	D	I	E	E	E	W							
R5 ← Mem(C)											F	D	I	E	E	E	W						
R6 ← (R4)*(R5)												F	D	■	■	I	E	E	E	W			
Mem(A) ← (R6)													F	■	■	D	■	■	I	E	E	E	W

(b) In-order instruction issuing

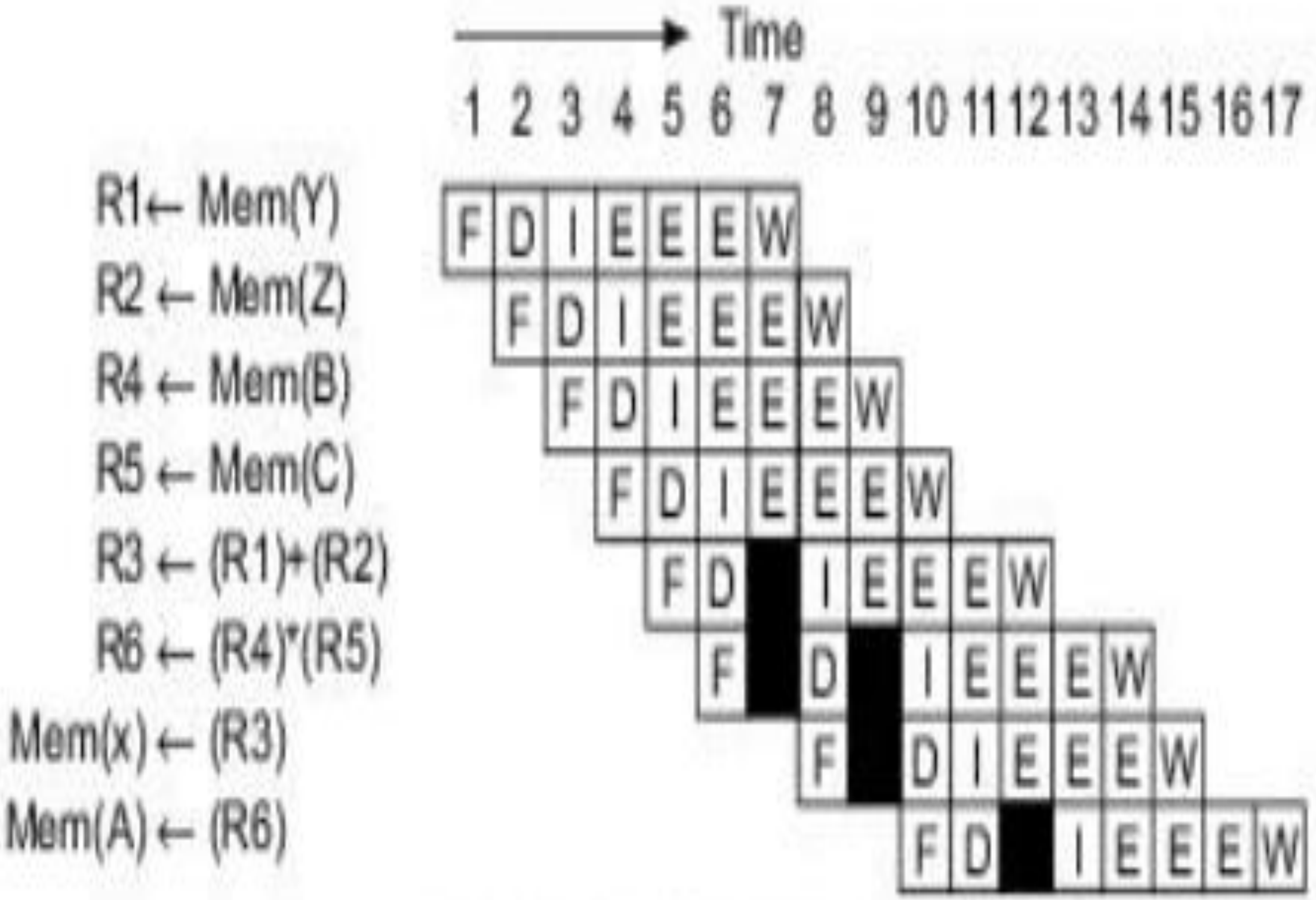
- Aim
  - X=y+z
  - A=b*c
- Shaded boxes corresponds to idle cycles
  - This situation happens when instruction issues are blocked due to
  - Resource latency or
  - Due to conflicts of data dependency
- First 2 load instructions issue on consecutive cycles
- Add is dependent on both loads
  - Hence it must wait 3 cycles before the data Y and Z is loaded

Total time required

- It is measured beginning at cycle 4, when the first instruction starts execution until cycle 20 when the last instruction starts execution
- This timing measure eliminates the undue effects of pipeline startup or draining delays
- Total time= 17 cycles

Instruction reordering

- To improve the timing, instruction issuing order is changed
- This eliminates unnecessary delays due to dependence
- All the load operations are issued first
- This reduces the timing to 11 cycles



(c) Reordered instruction issuing

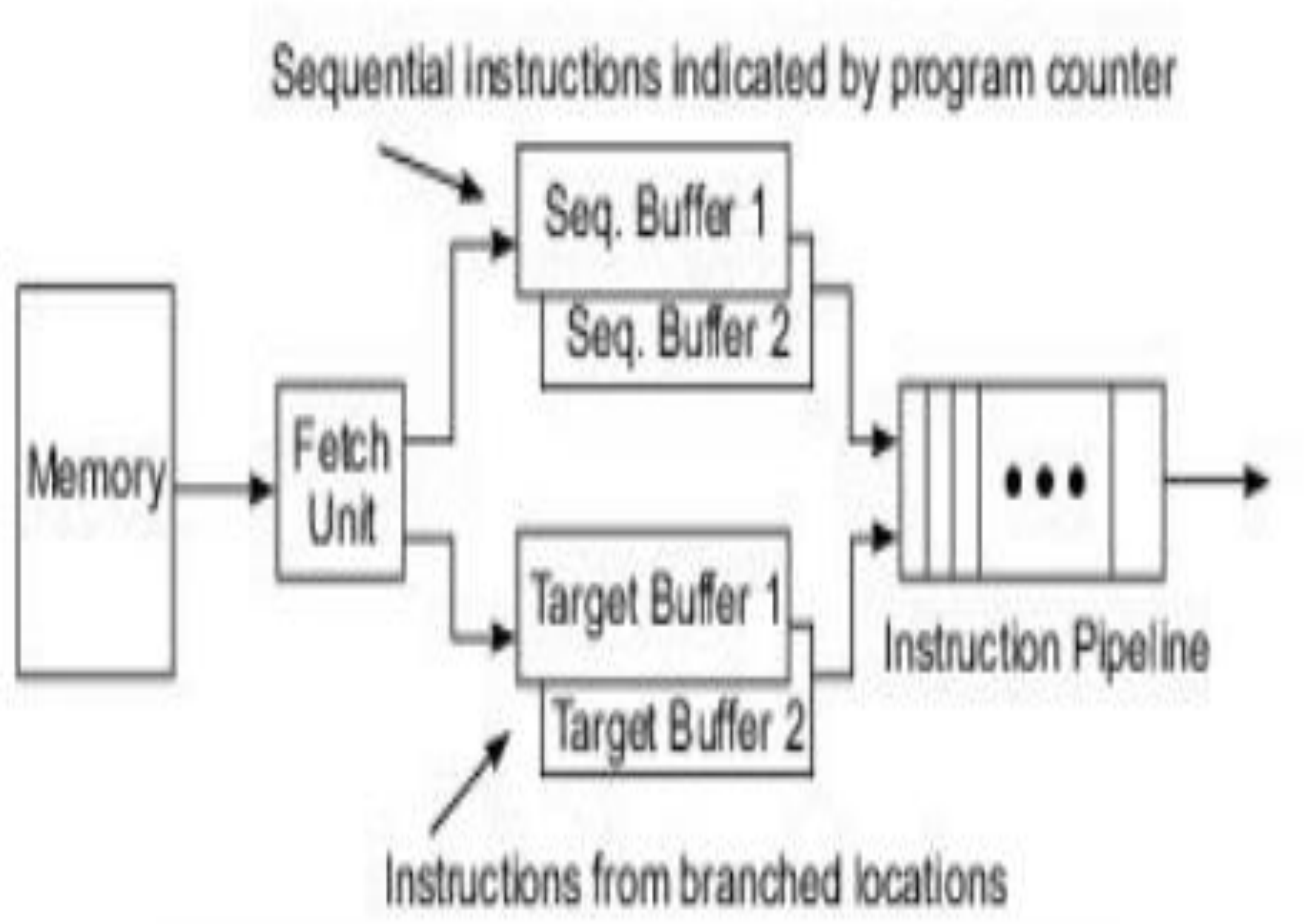
MECHANISM FOR INSTRUCTION PIPELINING

- Techniques are used to smoothen the pipeline flow and to reduce the bottlenecks of unnecessary memory accesses

  - Pre-fetch buffers
  - Multiple functional units
  - Internal data forwarding
  - Hazard avoidance

PRE-FETCH BUFFERS

- A block of consecutive instructions are fetched into a pre-fetch buffer in one memory access time
- 3 types of buffers are used to match the instruction fetch rate to the pipeline consumption rate
  - Sequential buffer
  - Target buffer
  - Loop buffer


Sequential instructions indicated by program counter
Seq. Buffer 1
Seq. Buffer 2
Memory
Fetch Unit
Target Buffer 1
Target Buffer 2
Instruction Pipeline
Instructions from branched locations

- **Sequential buffer**
  - Sequential instructions are loaded into a pair of sequential buffers for in-sequence pipelining
- **Target buffer**
  - Instructions from a branch target are loaded into a pair of target buffers for out of sequence pipelining

- Both buffers operate in FIFO fashion
- Buffers alternate to avoid collision
- Buffers become a part of pipeline as additional stages

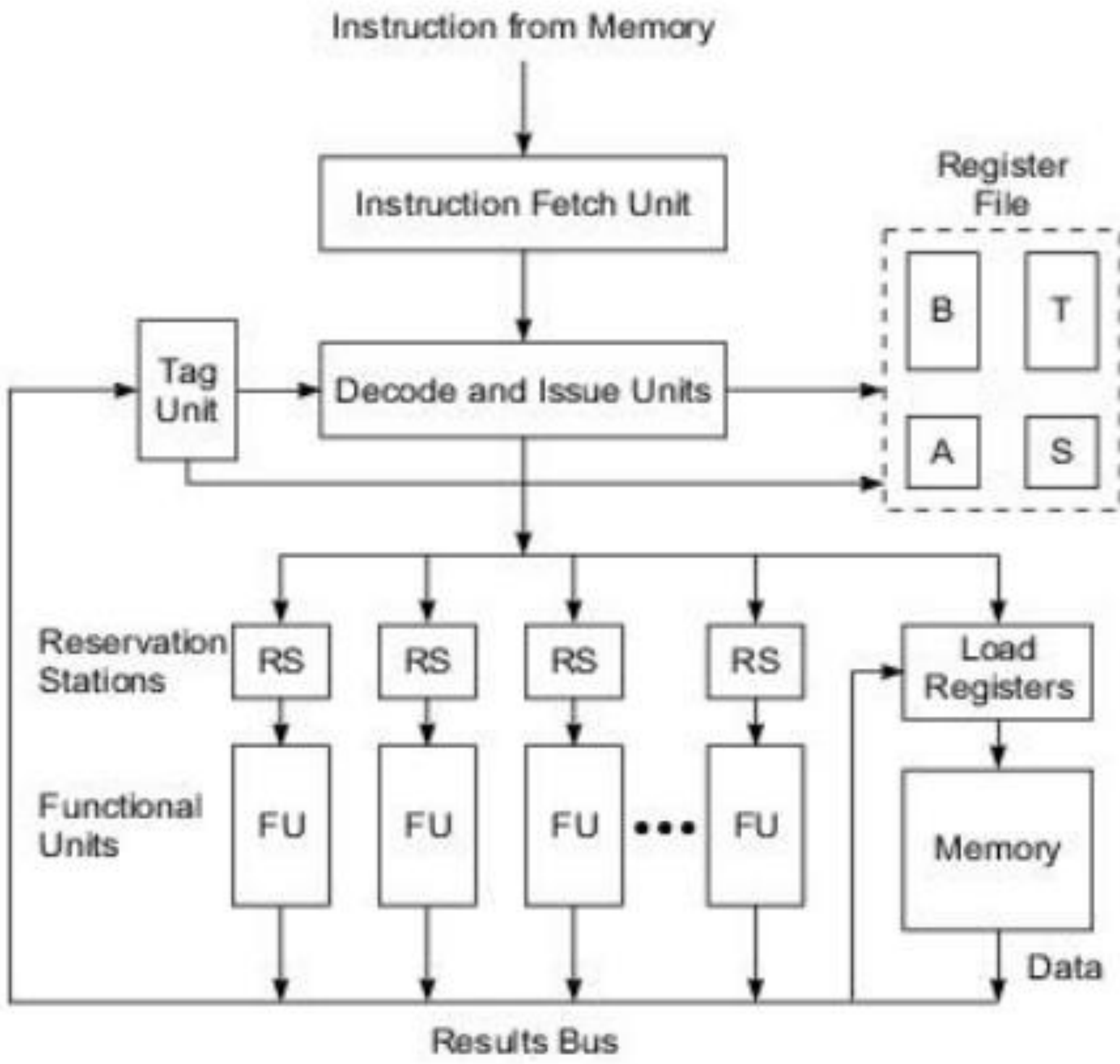
- Conditional branch instruction
  - This causes both sequential buffer & target buffer to fill with instructions
  - After the branch condition is checked, appropriate instructions are taken from one of the 2 buffers
  - Instruction in the other buffer is discarded
- Loop buffer
  - This buffer holds the sequential instructions contained in a small loop
  - Prefetched instructions in the loop body will be executed repeatedly until all iterations complete executions

MULTIPLE FUNCTIONAL UNITS

- Sometimes a particular pipeline stage becomes a bottleneck
  - This may happen to the row which contain max no: of checkmarks in a reservation table
- This bottleneck problem can be alleviated by using multiple copies of same stage simultaneously
- This leads to the use of multiple execution units in a pipelined processor design

Pipelined processor with multiple functional units

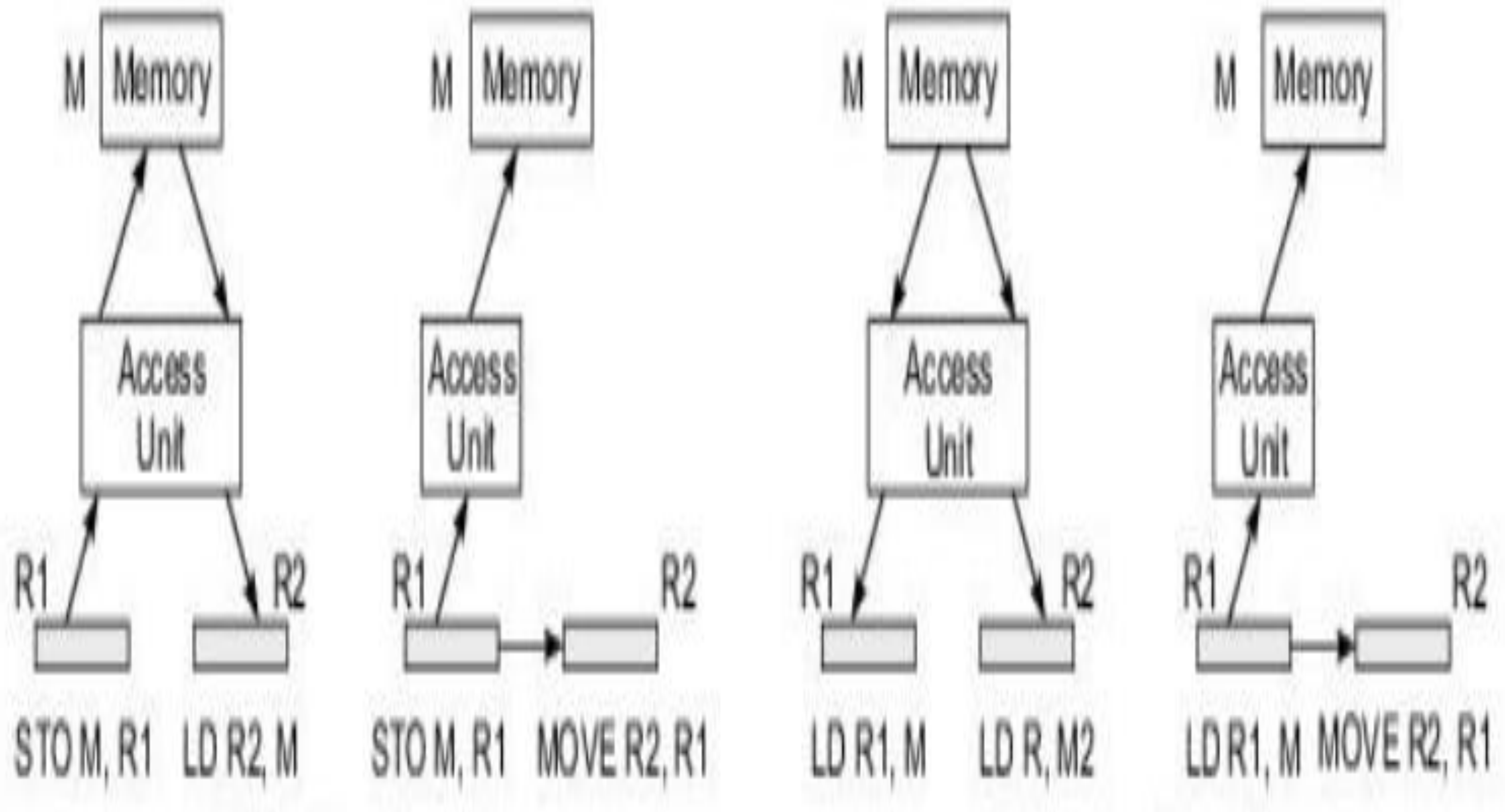
- This architecture consist of a Reservation Station along with each functional unit
  - They are used to resolve the data or resource dependency among successive instructions entering the pipeline
- Operations wait in the RS until their data dependency are resolved
- RS also serve as a buffer to interface the pipeline functional units with decode & issue unit
- RS is uniquely identified using a tag


Instruction from Memory
Instruction Fetch Unit
Register File
B
T
A
S
Tag Unit
Decode and Issue Units
Reservation Stations
RS
RS
RS
RS
Load Registers
Functional Units
FU
FU
FU
FU
Memory
Data
Results Bus

- Tag is monitored by a tag unit
  - They check the tags of all currently used registers or RS
  - This allows to resolve conflicts between source & destination registers assigned for multiple instructions
- When dependencies are resolved, multiple FU execute in parallel
- This reduces the bottleneck in the execution stages of instruction pipeline

INTERNAL DATA FORWARDING

- Throughput of pipelined processor is improves using internal data forwarding among multiple FU
- **A store-load forwarding,** which consist of a load operation (LD R2,M) can be replaced by a move operation from R1 to R2
  - MOVE R2,R1
- Register transfer is faster than memory access
- Hence data forwarding will reduce memory traffic
- This results in short execution time



(a) Store-load forwarding (b) Load-load forwarding

- In load-load forwarding, second load is replaced with move operation
  - LD R2,M➔ MOVE R2,R1

HAZARD AVOIDANCE

- Out of order execution of instructions may lead to certain hazards
- Reason is the read & write operation on shared variables by different instruction in pipeline
- I and J are 2 instructions
  - J follows I logically according to program order
  - If the execution order of these instruction violates the program order, incorrect results may be read or written
  - This leads to hazards

TYPES OF HAZARDS

- 3 types of logical hazards
  - Read after write (RAW)
  - Write after read (WAR)
  - Write after write (WAW)
- Consider 2 instructions I and J
- J follows I in program order

- D(I) (Domain of I)
  - contains the i/p set to be used by the instruction I
- R(I) (range of I)
  - o/p set of instruction I

READ AFTER WRITE HAZARD

- RAW **hazard** refers to a situation where an instruction tries to read a result that has not yet been calculated or retrieved
- J tries to read a source, before I writes to it
- RAW corresponds to flow dependence

- Eg:
  - I: R2← R1+R3
  - J: R4← R2+R3

WRITE AFTER READ HAZARD (WAR)

- This hazard occur when instruction J tries to write to a destination, before it is read by I
- WAR corresponds to antidependence
- Eg:
  - I: R4← R1+R5
  - J: R5← R1+R2

WRITE AFTER WRITE HAZARD

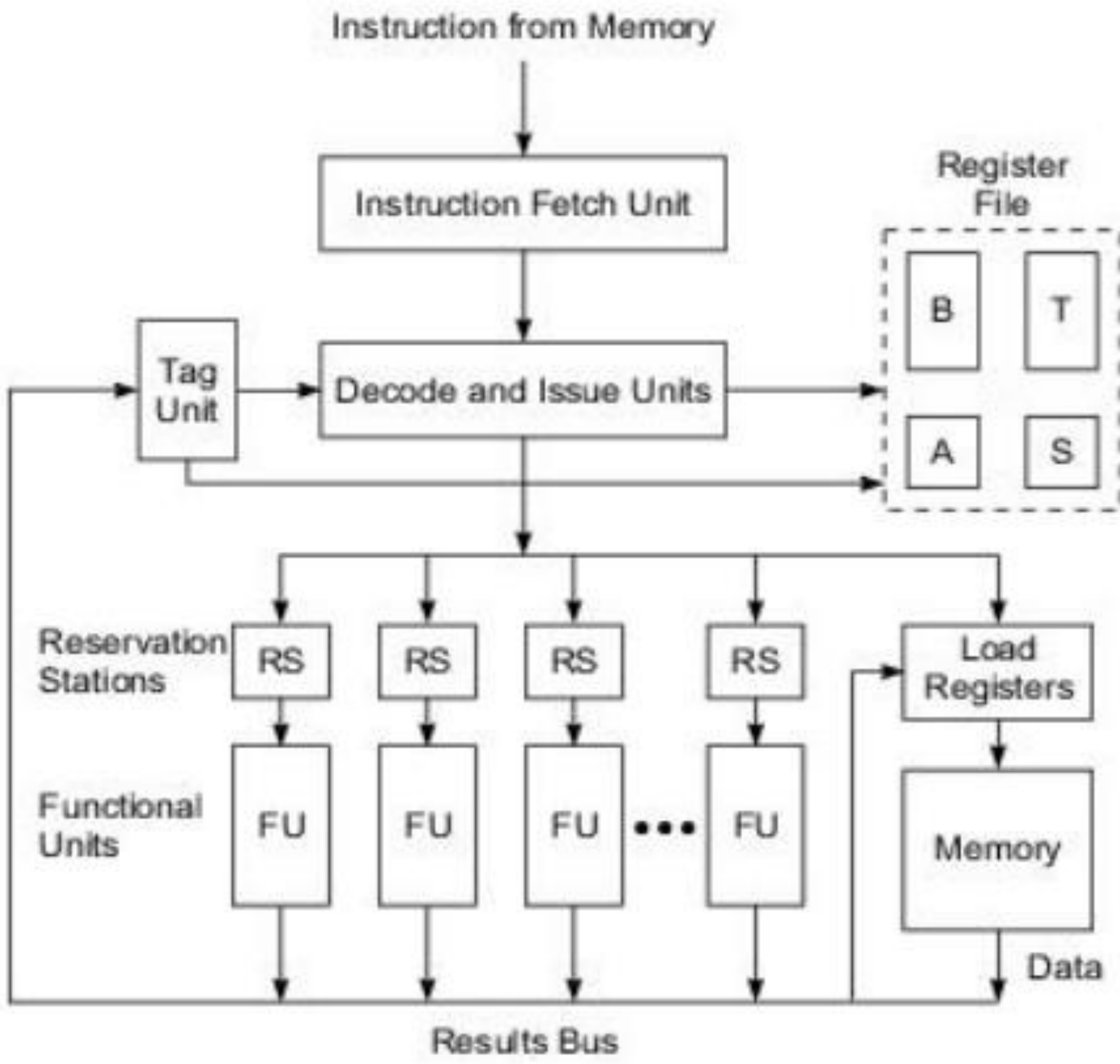
- J tries to write an operand before it is written by I
- WAW corresponds to output dependence
- Eg:
  - I: R2← R4+R7
  - J: R2← R1+R3

- Hazards must be prevented before instructions enter the pipeline
- This is done by holding instruction J, until the dependency on I is resolved

DYNAMIC INSTRUCTION SCHEDULING

TOMASULO'S ALGORITHM

- It is a hardware scheme for dependence resolution
- It was first implemented in floating point units of IBM360/91 processor
- In model91 processor,
  - 3 Reservation stations where used in a floating point adder
    - 2 pairs of RS were used in floating point multiplier
- This scheme resolved data dependencies and resource conflicts using register tagging
  - This helped to allocate or deallocate source & destination registers
- This scheme was applied to processors having few floating point registers


Instruction from Memory
Instruction Fetch Unit
Register File
B
T
A
S
Tag Unit
Decode and Issue Units
Reservation Stations
RS
RS
RS
RS
Load Registers
Functional Units
FU
FU
FU
FU
Memory
Data
Results Bus

WORKING

- An issued instructions whose operands are not available is forwarded to RS associated with the FU it will use
  - It waits until the data dependencies are resolved & the operands become available
- When all operands of an instruction are available,
  - instruction 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 register data becomes available, it also reaches the RS which has the same tag.

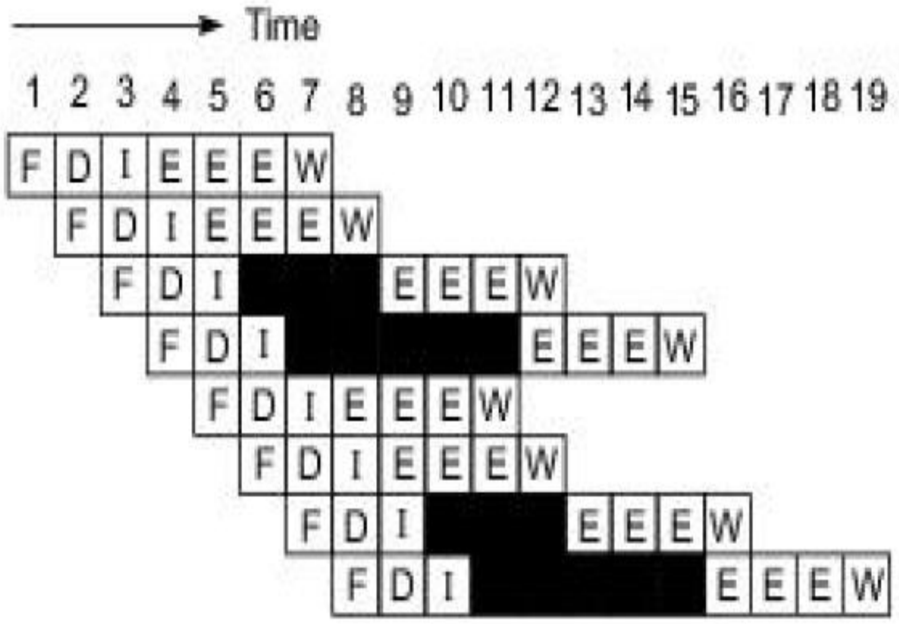
- The dependence is resolved by monitoring the result bus
  - When an instruction has completed execution, the result along with its tag appears on the result bus.
  - The registers as well as the RS’ s monitor the result bus and update their contents when a matching tag is found.

EXAMPLE

- Aim
  - X=Y+Z
  - A=B*C total execution time= 13 cycles

```
    R1 ←Mem(Y)
    R2 ←Mem(Z)
    R3 ←(R1)+(R2)
Mem(x) ←(R3)
    R1 ←Mem(B)
    R2 ←Mem(C)
    R3 ←(R1)*(R2)
Mem(A) ←(R3)
```

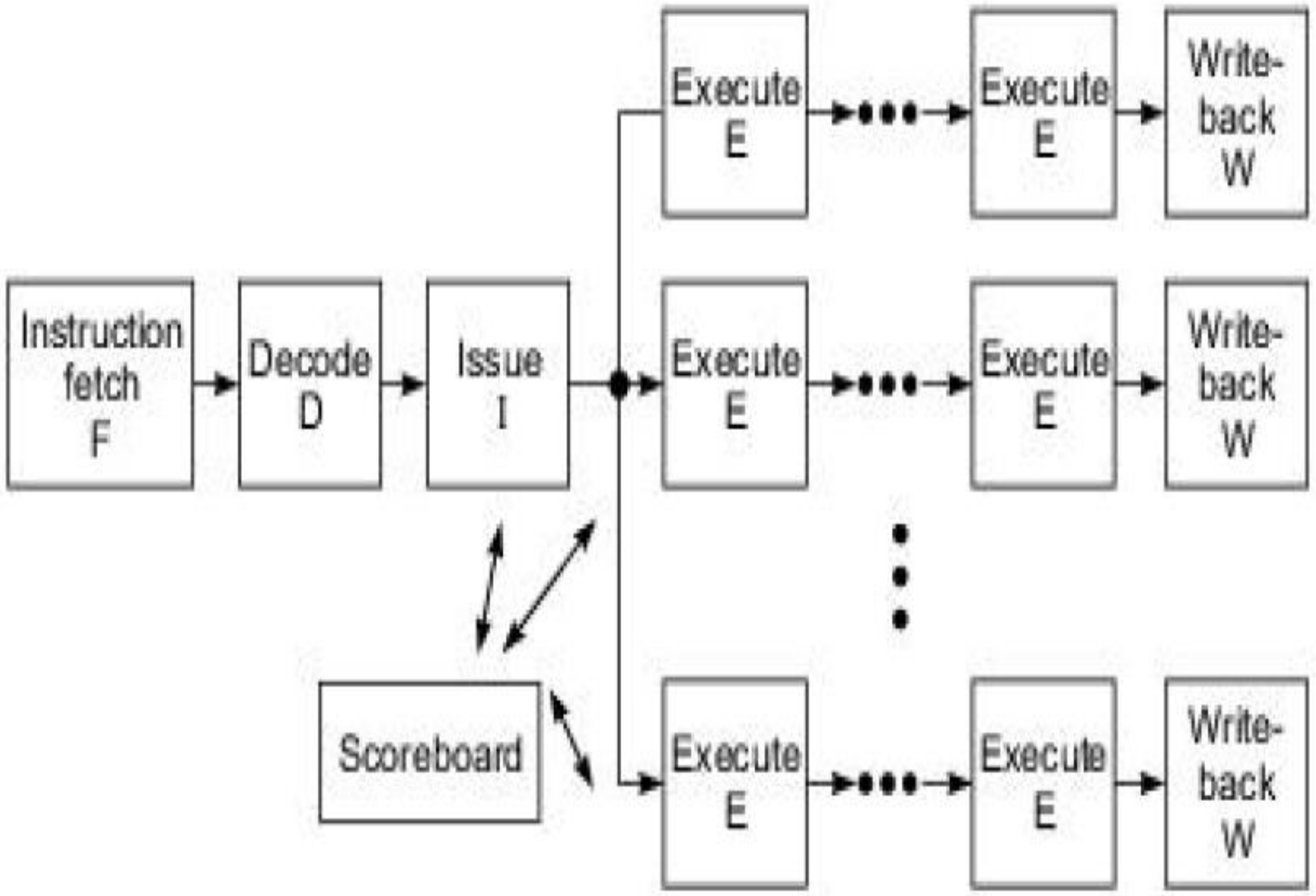
(a) Minimum-register machine code



(b) The pipeline schedule

CDC SCORE BOARDING

- CDC6600 is a high-performance computer that used dynamic instruction scheduling hardware
- In this model, multiple functional units appeared as multiple execution pipelines
- The processor had instruction buffers for each execution unit.
- Instructions were issued to available functional units regardless of whether register input data was available



(a) A CDC 6600-like processor

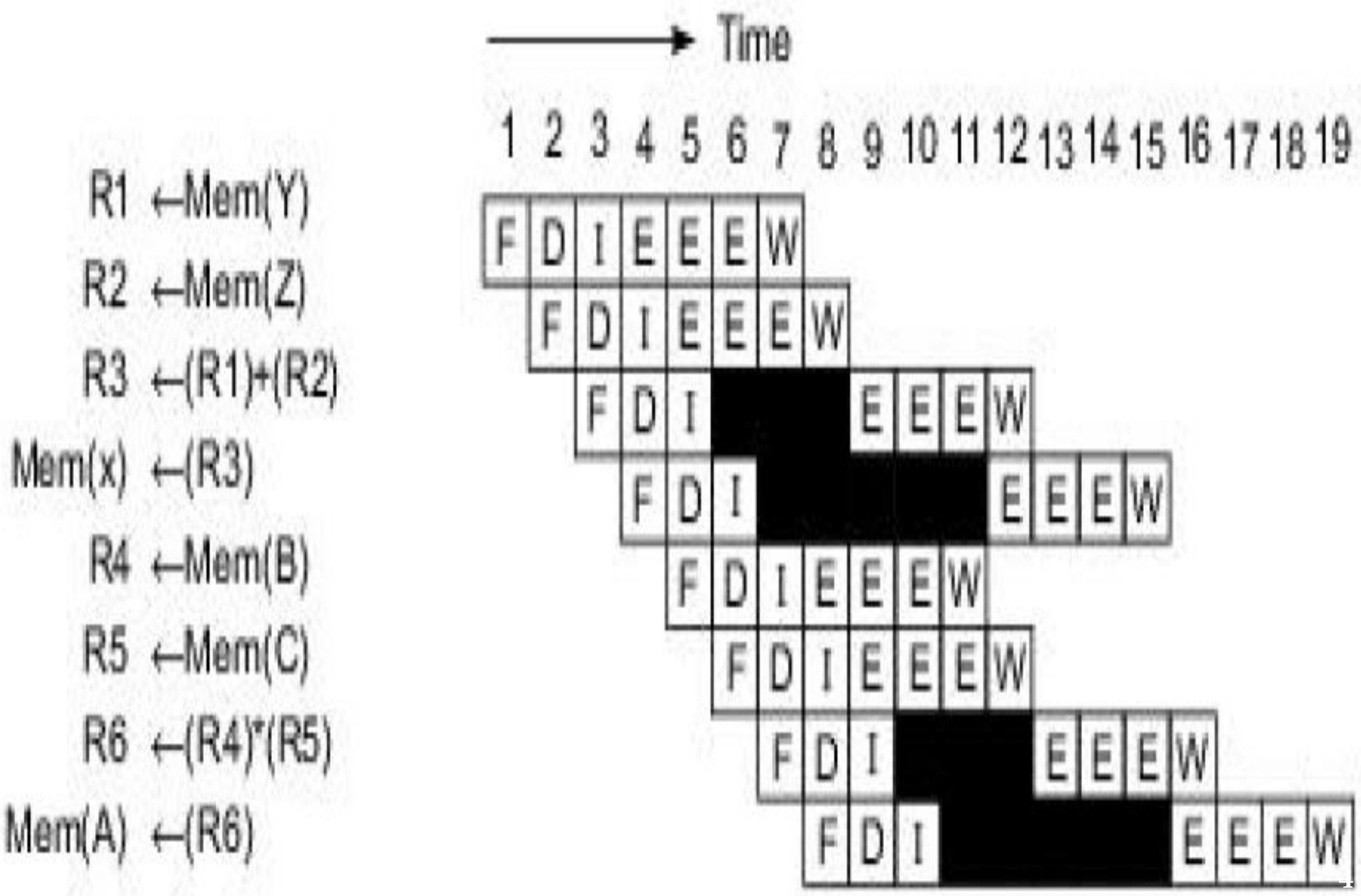
- The instruction waits in a buffer until its data is produced by other instructions.
- To control the correct routing of data between execution units and registers, there is a centralized control unit known as the **scoreboard.**

SCOREBOARD

- The scoreboard is a centralized control logic which keeps track of the status of registers and multiple functional units.
  - It kept track of the registers needed by instructions waiting for the various functional units.
  - When all registers had valid data, the scoreboard enabled the instruction execution.
  - Similarly, when a functional unit finished execution, it signaled the scoreboard to release the resources.

Example

- Aim
  - X=Y+Z
  - A=B*C
- The add instruction is issued to its functional unit before its registers are ready.
- It then waits for its input register operands.
- The scoreboard routes the register values to the adder unit when they become available.
- In the meantime, the issue stage is not blocked, so other instructions can bypass the blocked add.


R1 ←Mem(Y)
R2 ←Mem(Z)
R3 ←(R1)+(R2)
Mem(x) ←(R3)
R4 ←Mem(B)
R5 ←Mem(C)
R6 ←(R4)*(R5)
Mem(A) ←(R6)
Time
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
F D I E E E W
F D I E E E W
F D I E E E W
F D I E E E W
F D I E E E W
F D I E E E W
F D I E E E W
F D I E E E W

CONCLUSION

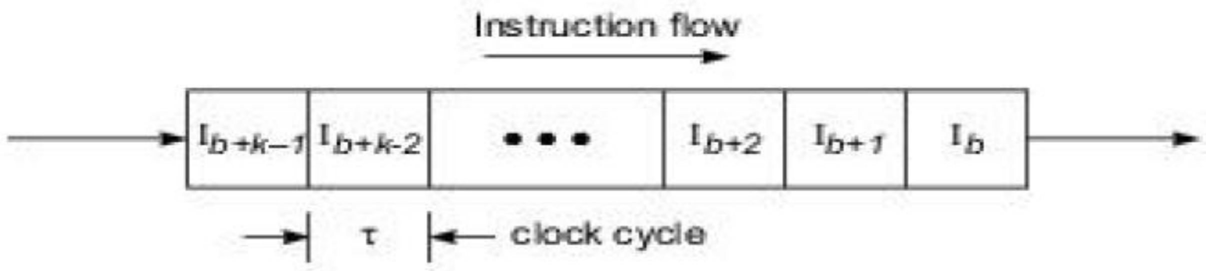
- Dynamic instruction scheduling was implemented only in high-end mainframes or supercomputers in the past.
- Most microprocessors used static scheduling.
- But the trend has changed over the last two decades.
- Today , RI SC and superscalar processors are built with hardware support of dynamic scheduling at runtime.

BRANCH HANDLING TECHNIQUES

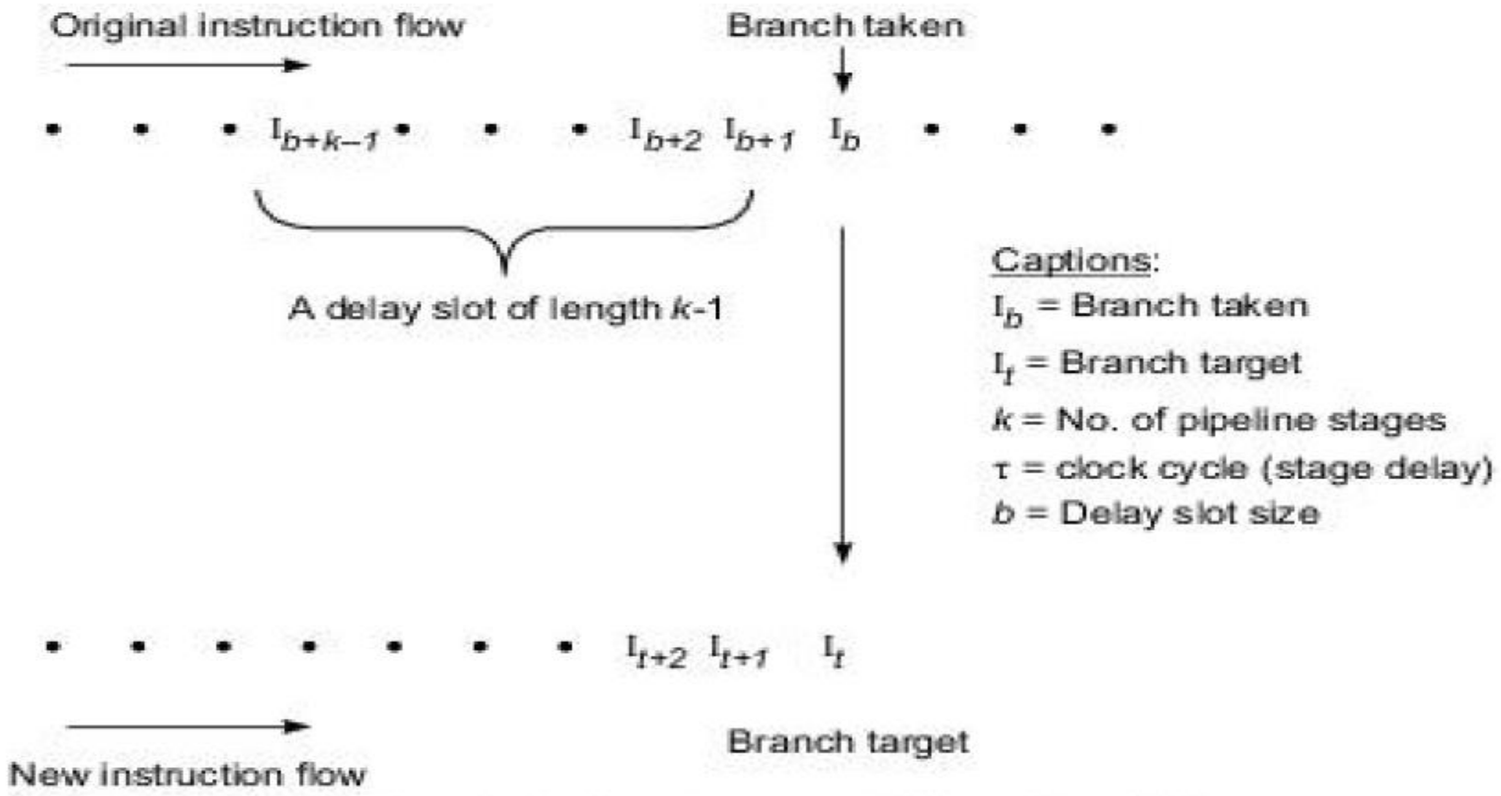
- The performance of pipelined processors is limited by data dependences and branch instructions
- Effects of branching
  - 3 terms used to analyze the effects of branching are:-
    - Branch taken
    - Branch target
    - Delay slot

- Branch taken
  - The action of fetching a non sequential or remote instruction after branch instruction is called branch taken
- Branch target
  - The instruction to be executed after a branch taken is called a branch target.
- Delay slot
  - It is the no: of pipeline cycles wasted between a branch taken and the fetching of its branch target
  - It is denoted by b.
  - Generally $0<=b<=k-1$, where k is the no: of pipeline stages

- When a branch is taken,
  - all the instructions following the branch in the pipeline become useless
  - Hence it will be drained from the pipeline.
  - This implies that a branch taken causes the pipeline to be flushed.
  - This lead to wastage of a no: of useful cycles.
- Branch taken causes l_{b+1} through I_{b+k-1} to be drained from the pipeline.



(a) A *k*-stage pipeline


Original instruction flow
Branch taken
• • • I_{b+k-1} • • • I_{b+2} I_{b+1} I_b • • •
A delay slot of length k-1
Captions:
I_b = Branch taken
I_t = Branch target
k = No. of pipeline stages
τ = clock cycle (stage delay)
b = Delay slot size
• • • • • • • I_{t+2} I_{t+1} I_t
Branch target
New instruction flow

(b) An instruction stream containing a branch taken

TOTAL EXECUTION TIME

- Total execution time of n instruction including the effect of branching
  - $T_{eff}=k\tau+(n-1)\tau+pqnb\tau$
- pqnbτ➔ branch penality
- p➔ probability of a conditional branch instruction in a typical instruction stream
- q➔ probability of a successfully executed conditional branch instruction
- b➔ delay slot size
- k➔ no: of pipeline stages
- n➔ no: of instructions
- τ➔ clock cycle

Effective pipeline throughput

- Effective pipeline throughput including the influence of branching is:-
  - $$H_{eff} = \frac{n}{T_{eff}} = \frac{nf}{k+n-1+pqnb}$$

Branch prediction

- A branch can be predicted using 2 methods
  - Static branch prediction➔ done by compiler
  - Dynamic branch prediction
- In static branch prediction,

Static branch prediction

- Branch is predicted based on branch instruction type
- Static strategy requires the collection of
  - frequency & probabilities of branch taken
  - Branch types across program
- Static branch prediction not very accurate

Dynamic branch prediction

- In dynamic branch prediction, branch is predicted based on the branch history
- Better than static prediction
  - It uses recent branch history to predict whether the branch will be taken next time or not
  - It also specifies when a branch occurs
- To predict this:-
  - Use the entire history of branch
- This is infeasible to implement
  - Hence dynamic prediction is determined with limited recent history

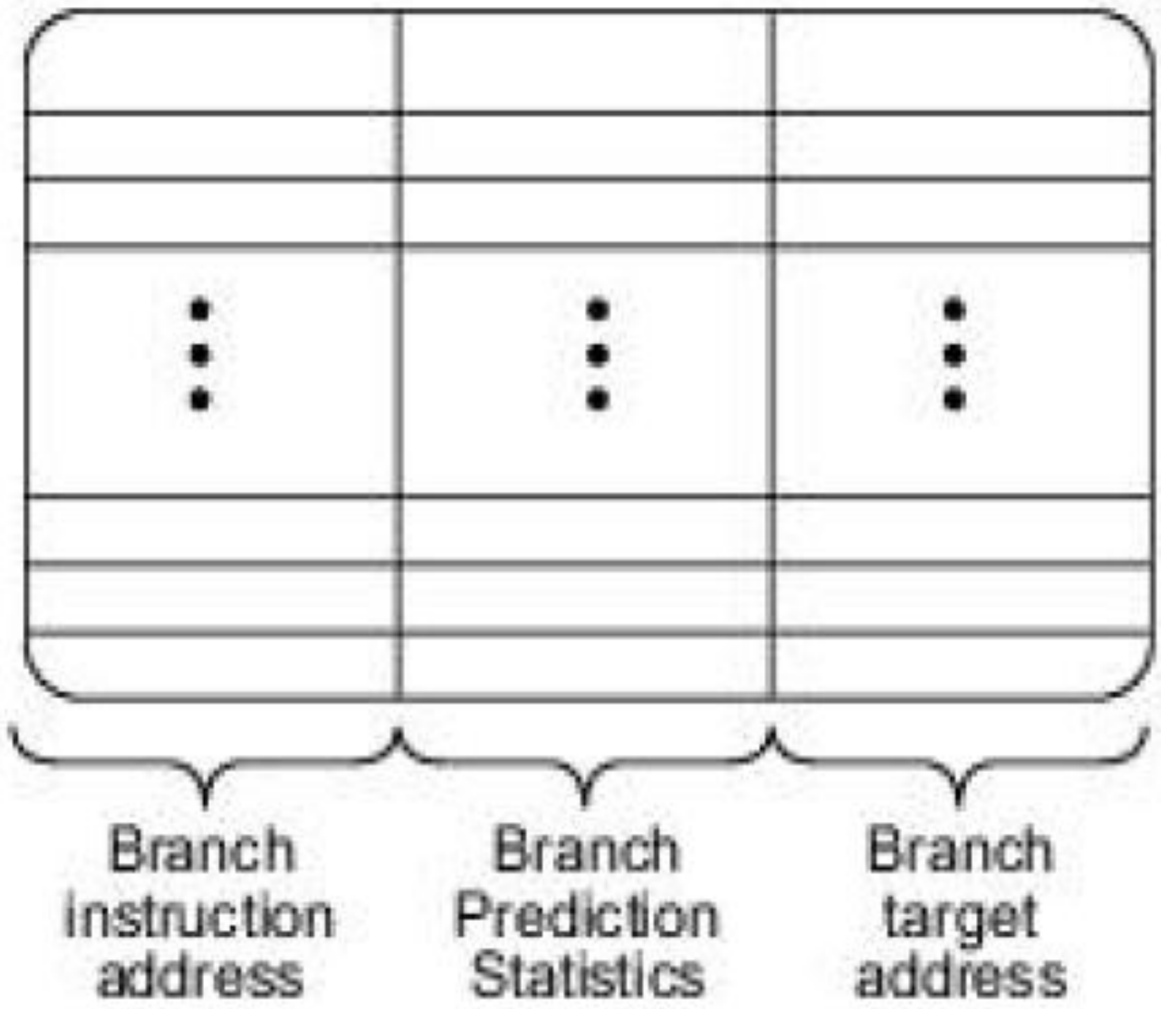
Classes of dynamic branch strategies

- Class1
  - Predicts the branch direction based on information found at the decode stage
- Class 2
  - Predicts the branch direction at the stage when effective address of the branch target is computed
  - It uses a cache to store the target addresses
- Class 3
  - Uses a cache to store the target instructions at fetch stage

- Dynamic prediction requires additional hardware
  - This h/w keep track of the past behavior of the branch instructions at run time.
  - The amount of history recorded should be relatively small.
  - Otherwise, the prediction logic becomes too costly to implement.

Branch target buffer

- Lee and Smith suggested the use of a branch target buffer, to implement branch prediction
- It is used to hold recent branch information
- The BTB entry contains the information which will guide the prediction.
- It includes the following:-
  - address of the branch target
  - Address of branch instruction
  - Branch prediction statistics
- Prediction information is updated upon completion of the current branch
- BTB can be extended to store not only the branch target address but also the target instruction itself
  - This is to allow zero delay in converting conditional branches to unconditional branches.



(a) Branch target buffer organization

ARITHMETIC PIPELINE DESIGN

COMPUTER ARITHMETIC PRINCIPLES

- Pipelining techniques can be applied to speed up numerical arithmetic computations
- In computers arithmetic operations are performed with finite precision
  - Finite precision implies that numbers exceeding the limit must be truncated or rounded
  - This is to provide a precision within the no: of significant bits allowed.
- Fixed-point arithmetic operates on a fixed range of numbers
- Floating-point arithmetic operates over a dynamic range of numbers.

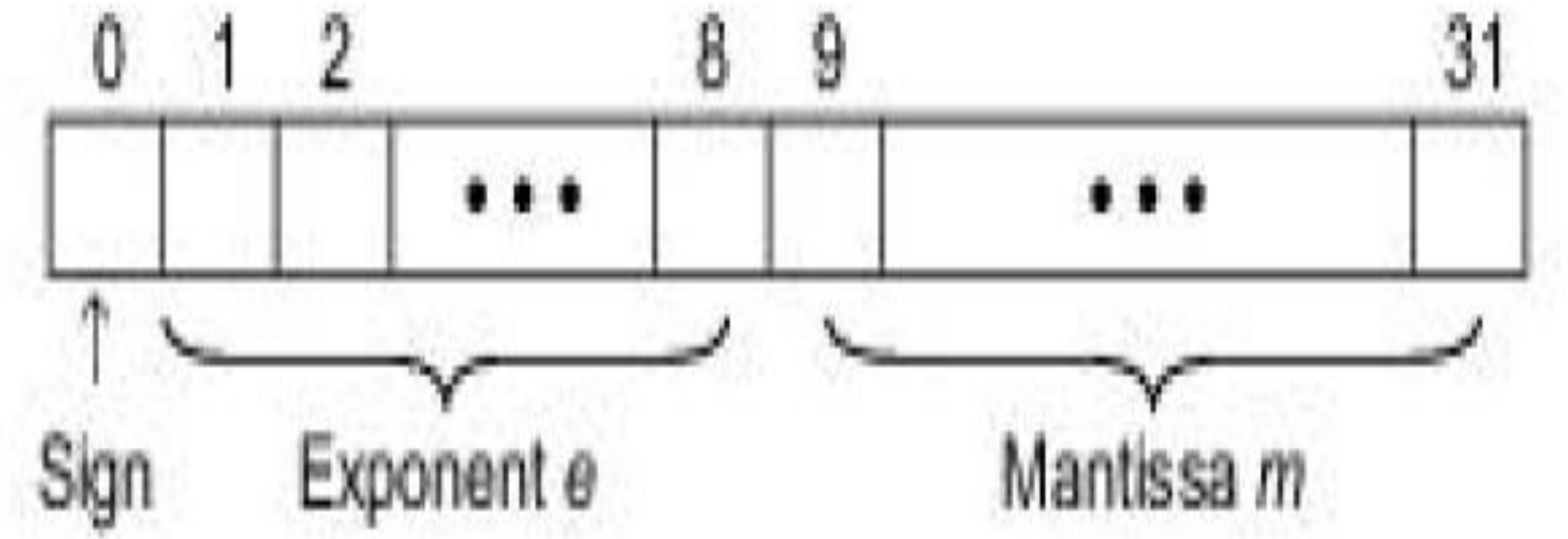
FIXED POINT OPERATIONS

- They are represented internally in machines as
  - sign-magnitude,
  - 1’s complement
  - 2’s complement notation
- Most computers use the two’s complement notation because of its unique representation of all numbers (including zero).
- Four primitive arithmetic operations are:-
  - Add
  - Subtract
  - Multiply
  - Divide

FLOATING POINT OPERATIONS

- Floating -point number X is represented by a pair (m,e)
  - m → mantissa or fraction
  - e→ exponent with an implied base or radix
- Floating point numbers, exceeding the exponent range leads to error conditions, called overflow or underflow
- IEEE has developed standard formats for 32- and 64-bit floating numbers known as the IEEE 754 standard

IEEE 754 STANDARD



STATIC ARITHMETIC PIPELINES

- Static arithmetic pipelines are designed to perform a fixed function
- They are hence called as unifunctional pipelines
- Most of arithmetic pipelines are designed to perform fixed functions.
- ALU perform fixed-point and floating-point operations separately.
  - The fixed-point unit is also called as integer unit
  - The floating-point unit can be built either as part of the central processor or on a separate coprocessor.
- Pipelining in scalar arithmetic pipelines are controlled by software loops

- Scalar and vector arithmetic pipelines differ in the areas of
  - register files
  - control mechanisms
- Vector hardware pipelines are built as
  - add-on options to a scalar processor
  - attached processor driven by a control processor

ARITHMETIC PIPELINE STAGES

- Depending on the function to be implemented, pipeline stages in an arithmetic unit require different hardware logic.
- Arithmetic operations like
  - Add
  - Subtract
  - multiply
  - Divide
  - Squaring
  - square root etc
- are implemented with the basic add and Shifting operations
- Hence the core arithmetic stages require some form of hardware to perform add and shift.

Pipeline stages of a 3 stage floating point adder

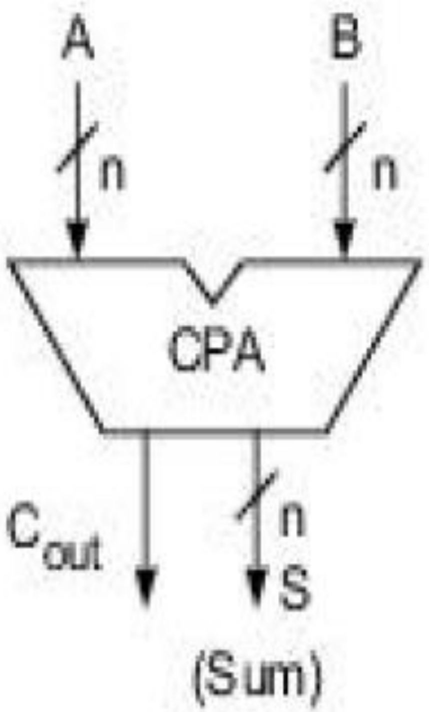
- Stage 1: exponent comparison and equalization
  - It is implemented with an integer adder and some shifting logic
- Stage 2: fraction addition
  - Implemented using a high-speed carry lookahead adder
- Stage 3: Fraction normalization and exponent readjustment
  - Implemented using a shifter and another addition logic

CPA and CSA

- 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)
- CPA adds two numbers and produces an arithmetic sum
  - In a CPA, the carries generated in successive digits are allowed to propagate from the low end to the high end,
  - Propagation is done using
    - ripple carry propagation
    - carry look ahead technique.

e.g. n=4

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

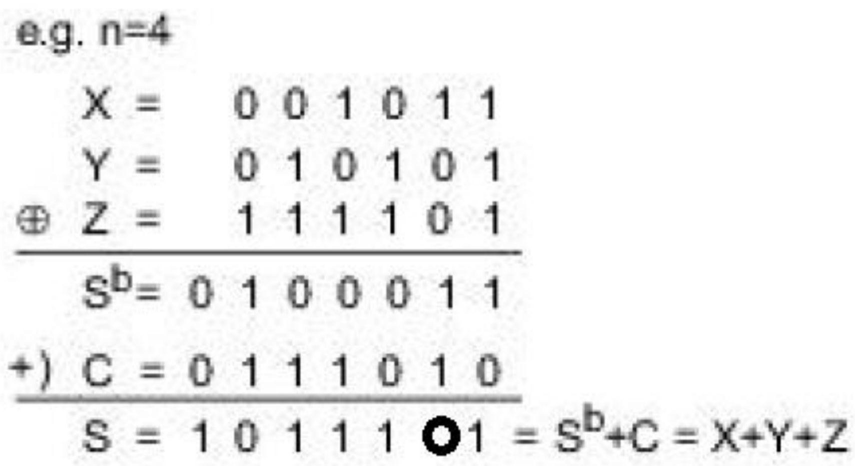


(a) An *n*-bit carry-propagate adder (CPA) which allows either carry propagation or applies the carry-lookahead technique

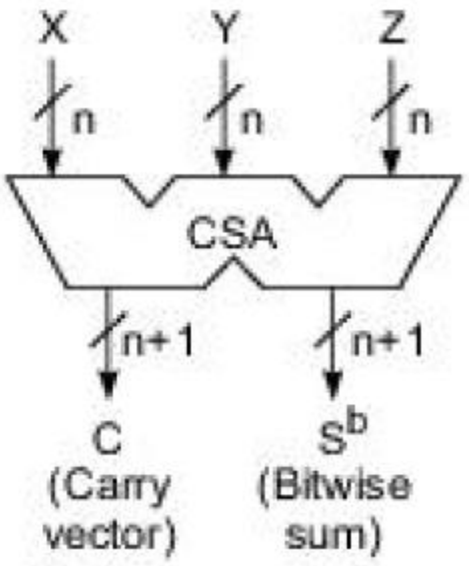
- CSA add three input numbers and produce one sum output and a carry output
  - In a CSA, the carries are not allowed to propagate but instead are saved in a carry vector
  - An n bit CSA is specified as:-
    - Let X, Y, and Z he three n-bit input numbers, expressed as $X=(x_{n-1},x_{n-2},\ldots\ldots x_1,x_0)$
    - The CSA performs bitwise operations simultaneously on all columns of digits
    - It produce two n-bit output numbers, denoted as S^b and C
      - $S^b \rightarrow$ bitwise sum

- $S^b=(0,S_{n-1},S_{n-2},\ldots\ldots S_1,S_0)$
- $C=(C_n,C_{n-1},\ldots\ldots\ldots\ldots C_1,C_0)$
  - Leading bit of S^b is always 0
  - Trail bit of C is always 0

- The i/p o/p relationship is expressed as
  - $S_i = x_i \text{ XOR } y_i \text{ XOR } z_i$
  - $C_{i+1} = x_i y_i \text{ OR } y_i z_i \text{ OR } z_i x_i$



```
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
---------------------------
     Sᵇ= 0 1 0 0 0 1 1
+)   C = 0 1 1 1 0 1 0
---------------------------
     S = 1 0 1 1 1 0 1  = Sᵇ+C = X+Y+Z
```



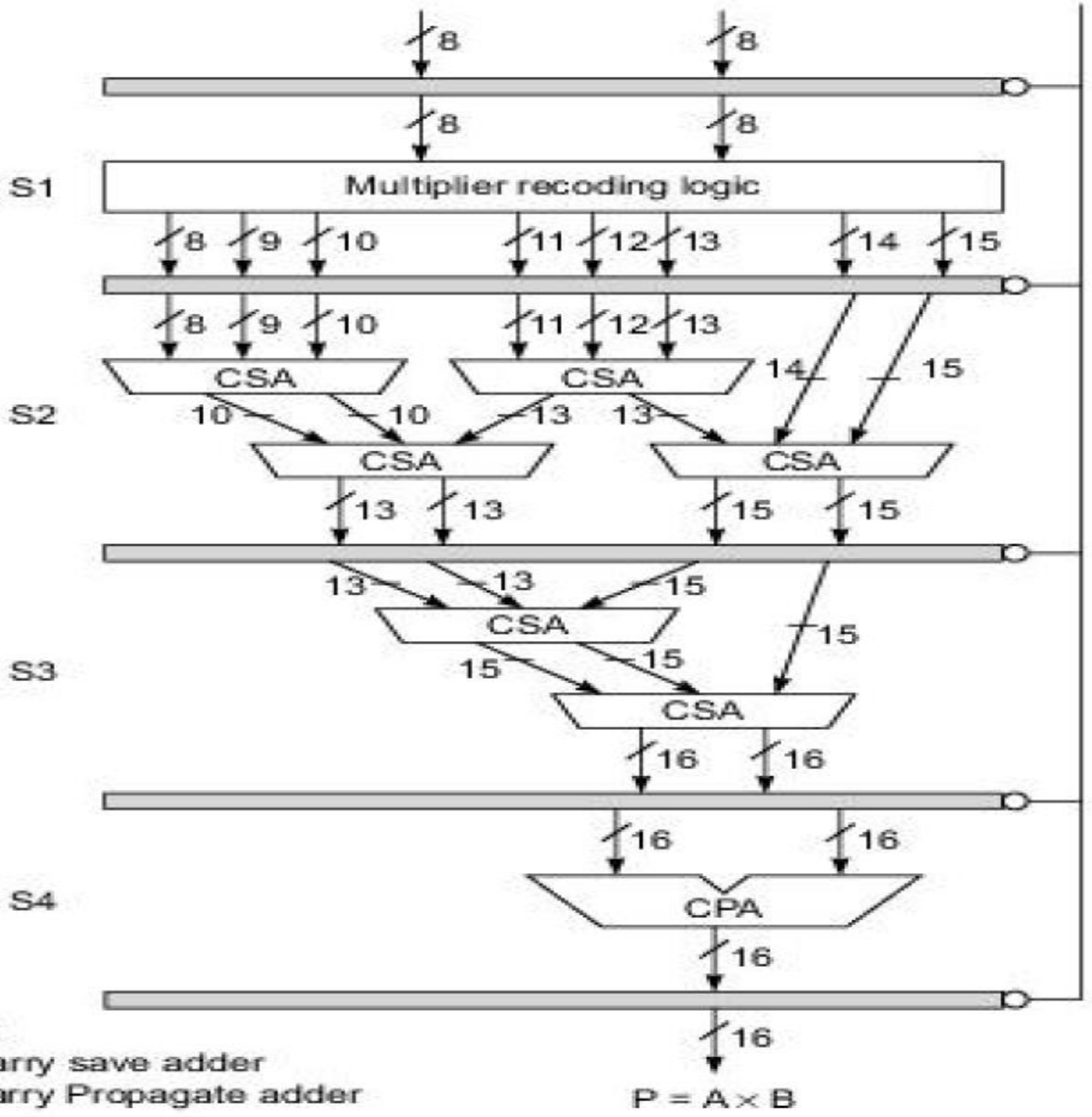
MULTIPLY PIPELINE DESIGN

- Consider following operation
  - A*B= P
- A and B are 8 bit inputs
- P➔16 bit product
- Fixed point multiplication is the summation of 8 partial products
  - P=P0+P1+P2…..P7

$$
\begin{array}{cccccccccccccccccc}
 & & & & & & & & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & = & A \\
 & & & & & & & \times) & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & = & B \\
\hline
 & & & & & & & & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & = & P_0 \\
 & & & & & & & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & = & P_1 \\
 & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = & P_2 \\
 & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = & P_3 \\
 & & & & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & = & P_4 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = & P_5 \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = & P_6 \\
+) & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = & P_7 \\
\hline
0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & = & P
\end{array}
$$

Pipeline unit for fixed point multiplication

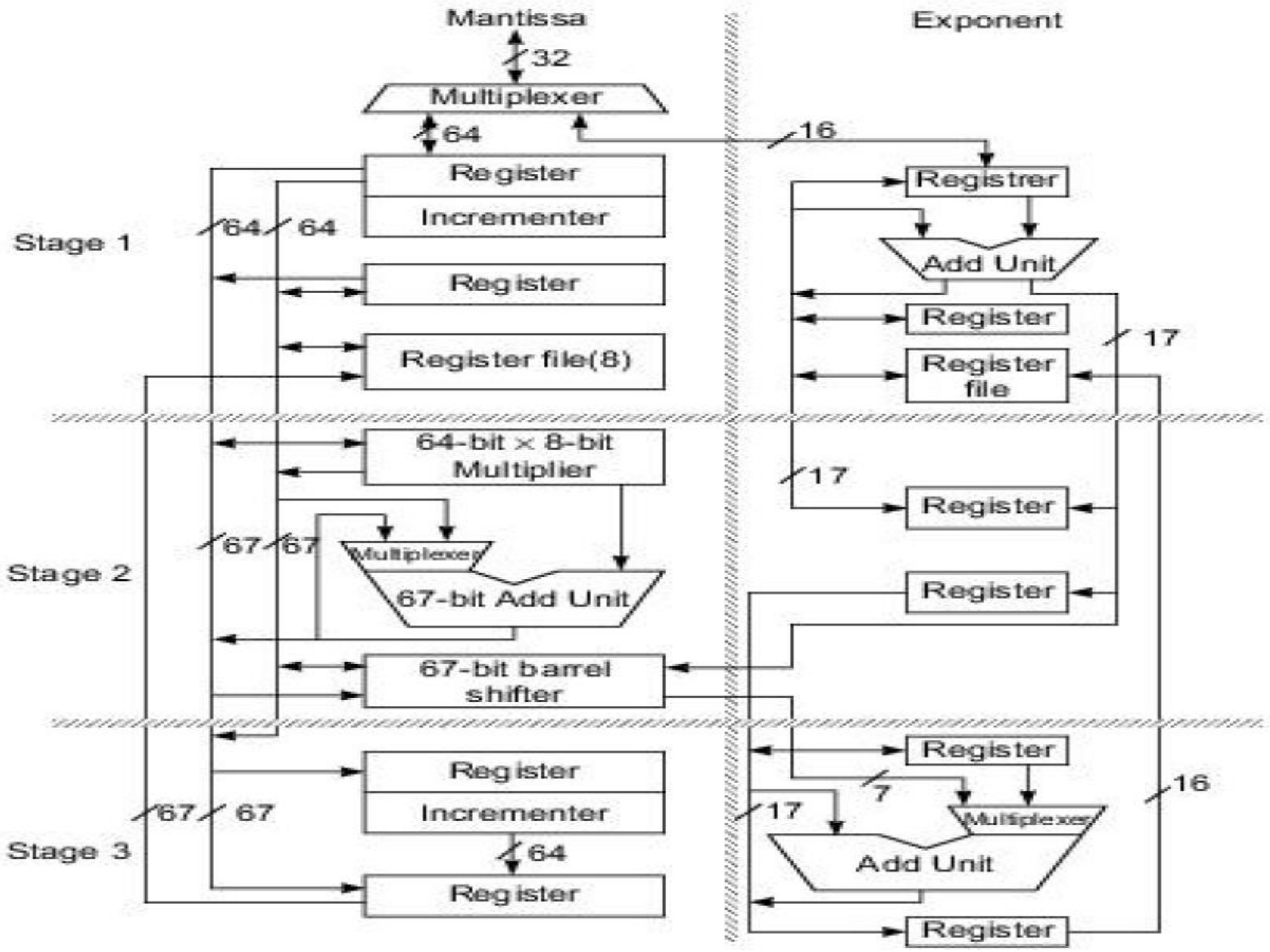
- Stage S1
  - generates all eight partial products, ranging from 8 bits to 15 bits simultaneously.
- Stage S2
  - It is made of two levels of four CSA
  - It merges eight numbers into four numbers
- Stage S3
  - consists of two CSAs,
  - It merges four numbers from S2 into two 16-bit numbers
- Stage S4
  - It is a CPA, which adds up the last two numbers to produce the final product P.



Captions:
CSA = Carry save adder
CPA = Carry Propagate adder

FLOATING POINT UNIT IN MOTOROLA MC68040

- This arithmetic pipeline has three stages.
- The mantissa section and exponent section are implemented as 2 separate pipelines
- The mantissa section can perform floating-point add or multiply operations on 32 bit or 64 bit


Mantissa
32
Multiplexer
64
16
Exponent
Register
Incrementer
Registrer
Stage 1
64
64
Add Unit
Register
Register
17
Register file(8)
Register file
64-bit × 8-bit Multiplier
17
Register
Stage 2
67
67
Multiplexer
67-bit Add Unit
Register
67-bit barrel shifter
Register
Register
Incrementer
7
16
67
67
17
Multiplexer
Stage 3
64
Add Unit
Register
Register

MANTISSA SECTION

- Stage 1
  - receives input operands and returns with computation results
  - 64-bit registers are used in this stage.
  - All three stages are connected using two 64-bit data buses.
- Stage 2
  - contains the array multiplier which is repeatedly used to carry out a long multiplication of the two mantissas.
  - The 67-bit adder performs the addition/subtraction of two mantissas
  - Barrel shifter is used for normalization.
- Stage 3
  - contains registers for holding results before they are loaded into the register file in stage 1 for subsequent use by other instructions.

EXPONENT SECTION

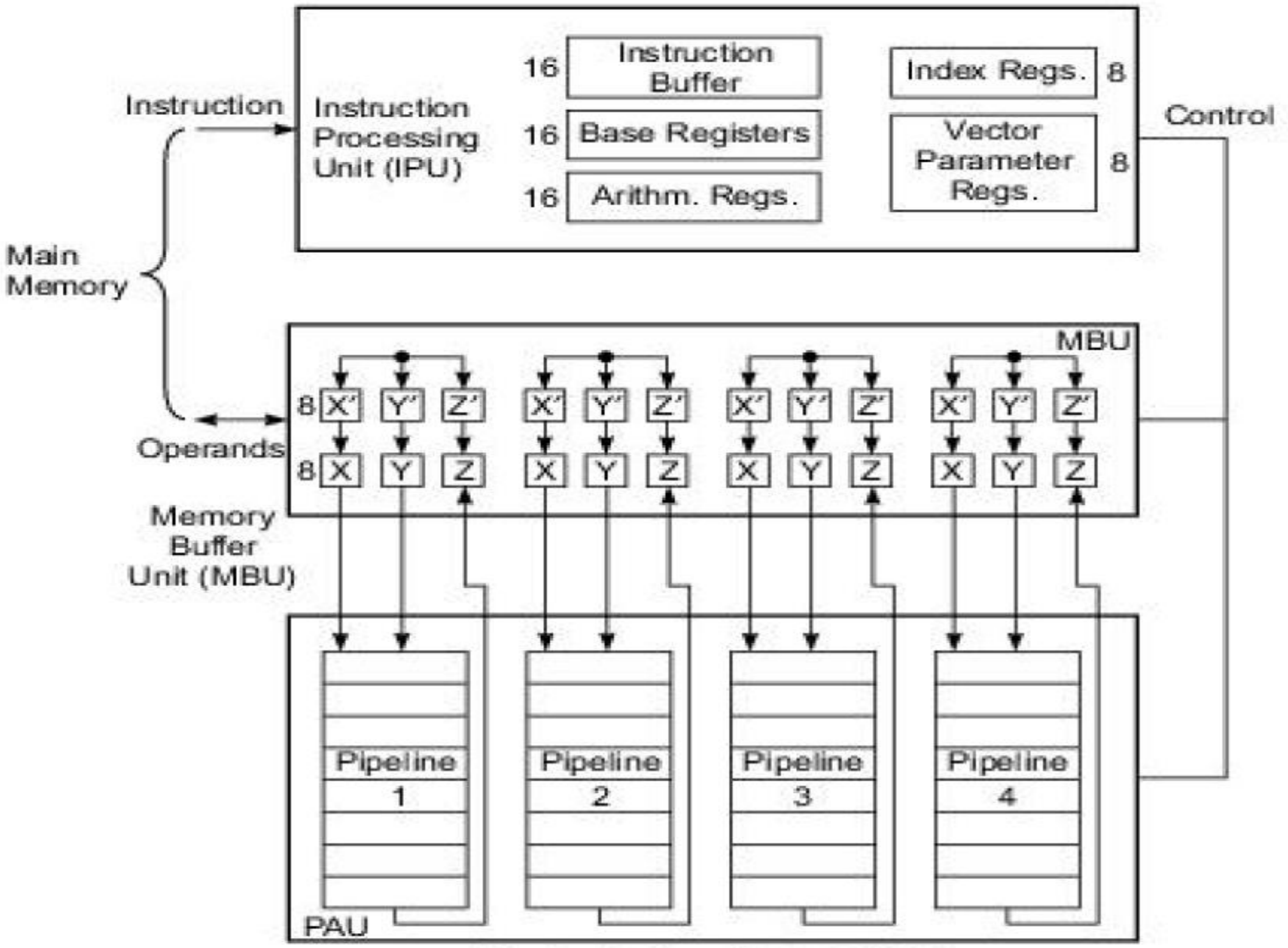
- 16 bit bus is used between stages.
- Stage 1
  - It has an exponent adder for comparing the relative magnitude of two exponents.
- Stage 2
  - The result of stage 1 is used to equalize the exponents before mantissa addition can be performed
  - Therefore, output of the exponent adder is sent to the barrel shifter for mantissa alignment.
- Stage 3
  - After normalization of the final result the exponent is readjusted in stage 3 using another adder.
  - The final value of the resulting exponent is fed from the register in stage 3 to the register file in stage 1

MULTIFUNCTIONAL ARITHMETIC PIPELINE

- A pipeline which can perform more than one function, is called as multifunctional pipeline
- It can be of 2 types
  - Static pipeline
  - Dynamic pipeline
- Static pipelines perform one function at a time
  - 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

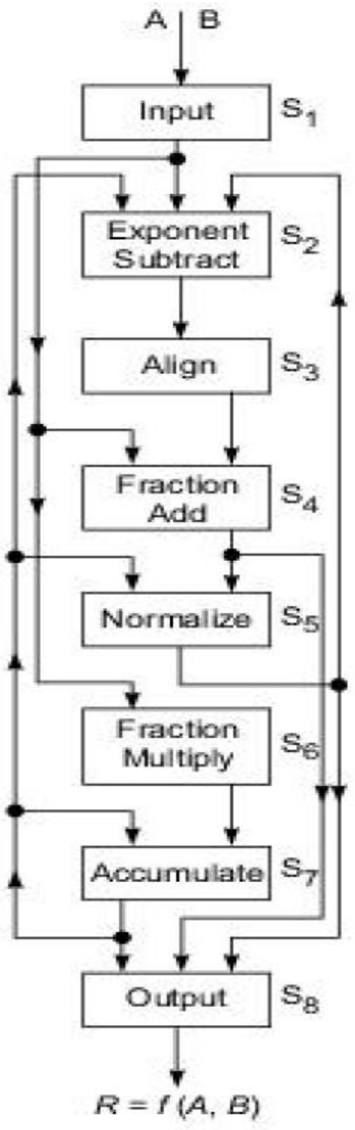
TI/ASC ARITHMETIC PROCESSOR DESIGN

- There are 4 pipeline units in this s/m
- Instruction processing unit
  - handled the fetching and decoding of instructions.
  - working registers in the processor controlled the operations of the memory buffer unit and arithmetic units.
- Operand buffers
  - 2 set of operand buffers in each arithmetic unit
    - {X,Y,Z}
    - {X',Y',Z'}
  - X',X, Y' and Y are used for input operands
  - Z' and Z were used to o/p results
  - intermediate results could are routed from Z-registers to either X or Y registers
- Both processor and memory buffers accessed the main memory for instructions and operands/results

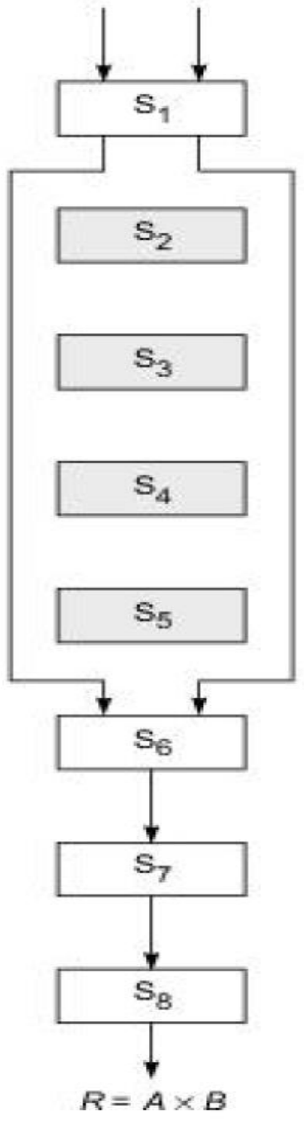


Pipeline Arithmetic Units (PAU)

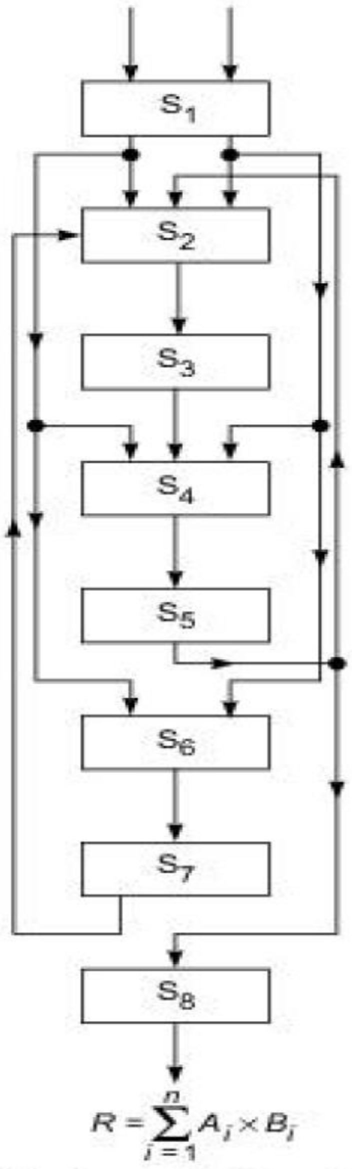
- Pipeline arithmetic unit
  - Each pipeline arithmetic unit had eight stages
  - PAU was a static multifunction pipeline which could perform only one function at a time.
  - Both fixed-point and floating-point arithmetic functions could be performed by this pipeline.
  - The PAU also supported vector in addition to scalar arithmetic operations.
  - Different functions required different pipeline stages and different interstage connection patterns



(a) Pipeline stages and interconnections



(b) Fixed-point multiplication



(c) Floating-point dot product

Example

- fixed-point multiplication required the use of only segments S1, S6, S7 and S8
- The floating-point dot product function, which performs the dot product operation between two vectors, required the use of all segments with the complex connections
  - This dot product was implemented by essentially the following accumulated summation of a sequence of multiplications through the pipeline