# MODULE 5

## 5.1 INSTRUCTION PIPELINE DESIGN

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

### 5.1.1 Instruction Execution Phases

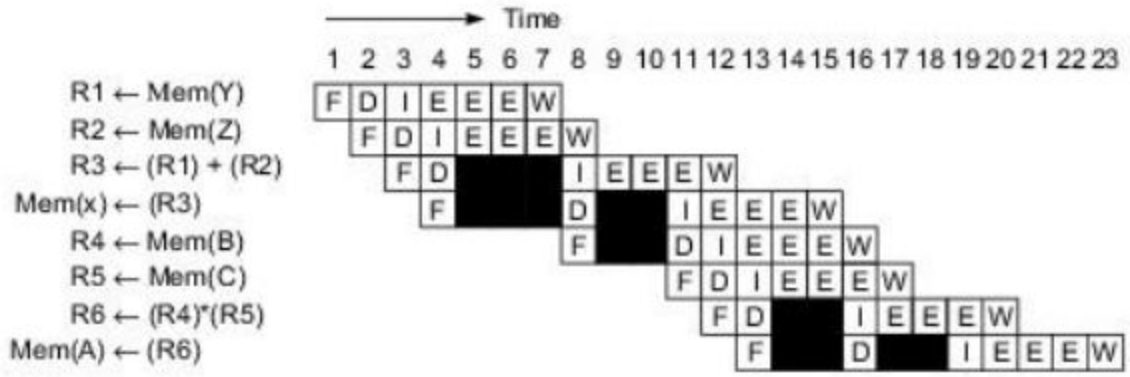
A typical instruction execution consists of a sequence of operations, including instruction fetch, decode, operand fetch, execute, and write-back phases.

***Pipelined Instruction Processing***

A typical instruction pipeline is shown below

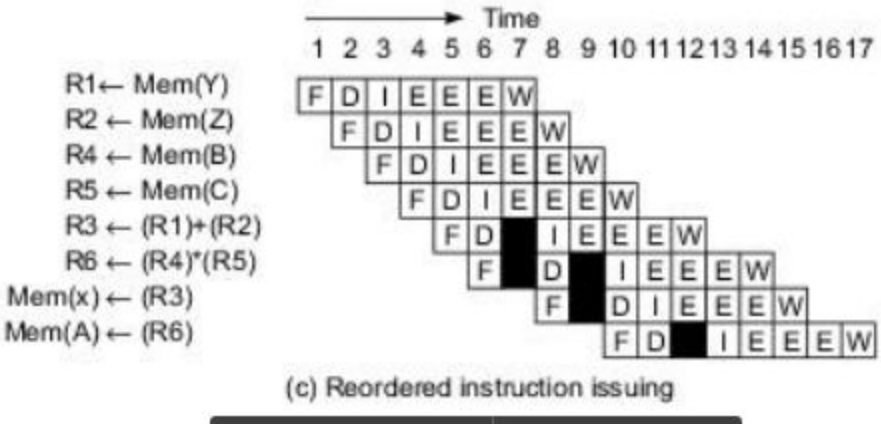

(a) A seven-stage instruction pipeline

- 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. Resources include general-purpose registers buses, and functional units.
- 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)*. Three execute stages are shown in Fig.
- The last *writeback stage (W)* is used to write results into the registers. Memory load or store operations are treated as part of execution.




(b) In-order instruction issuing

- Figure (b): illustrates the issue of instructions following the original program order.
- The shaded boxes correspond to *idle cycles* when instruction issues are blocked due to resource latency or conflicts or due to data dependencies.
- The total time required is 17 Clock cycles. This time is measured beginning at cycle 4 when the first instruction starts execution until cycle 20 when the last instruction starts execution.




- Figure (c) shows an improved timing after the instruction issuing order is changed to eliminate unnecessary delays due to dependence.
- The idea is to issue all four load operations in the beginning.
- Both the add and mutiply- instructions are blocked fewer cycles due to this data prefetching.
- The reordering should not change the end results.
- The time required is being reduced to 11 cycles, measured from cycle 4 to cycle I4.

### 5.1.2 Mechanisms for Instruction Pipelining

***Prefetch Buffers***

- In one memory-access time, a block of consecutive instructions are fetched into a prefetch buffer as illustrated in Fig.

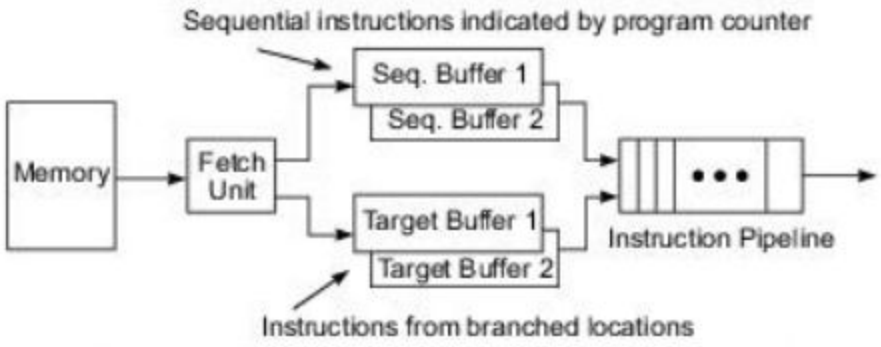



**Fig. 6.11** The use of sequential and target buffers

- Three types of buffers can be used to match the instruction fetch rate to the pipeline consumption rate.
    1. Sequential instructions are loaded into a pair of *sequential buffers* for in-sequence pipelining.
    2. Instructions from a branch target are loaded into a pair of *target buffers* for out-of-sequence pipelining.
- Both buffers operate in a first-in-first-out fashion.

- A conditional branch instruction causes both sequential buffers and target buffers to fill with instructions. After the branch condition is checked, appropriate instructions are taken from one of the two buffers, and instructions in the other buffer are discarded.
- Within each pair, one can use one buffer to load instructions from memory and use another buffer to feed instructions into the pipeline. The two buffers in each pair alternate to prevent a collision between instructions flowing into and out of the pipeline.

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

***Multiple Functional Units***

- 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.
- In order to resolve data or resource dependences among the successive instructions entering the pipeline, the *reservation stations (RS)* are used with each functional unit.
- Operations wait in the RS until their data dependences have been resolved. Each RS is uniquely identified by a *tag*, which is monitored by a *tag unit.*
- Besides resolving conflicts, the RSs also serve as buffers to interface the pipelined functional units with the decode and issue units.
- The multiple functional units operate in parallel, once the dependences are resolved.

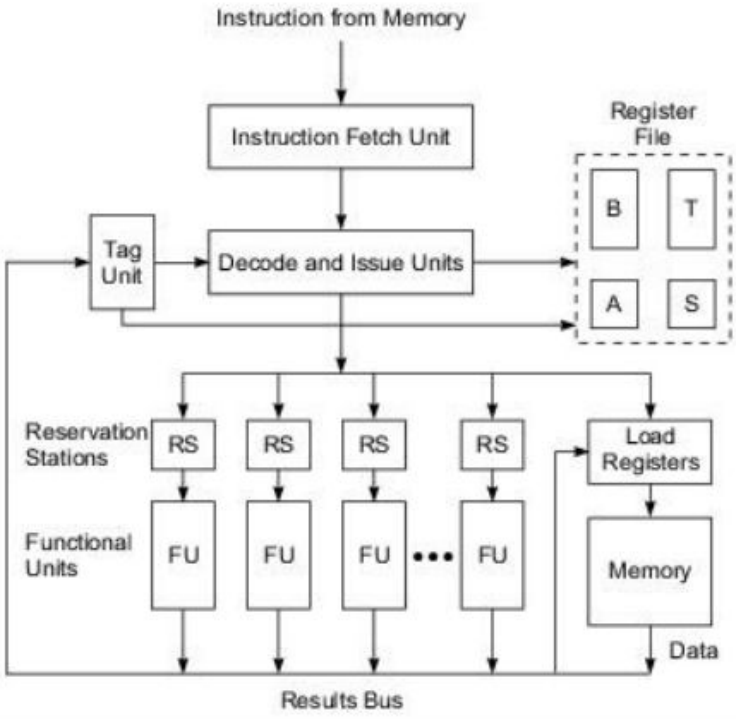



**Fig. 6.12** A pipelined processor with multiple functional units and distributed reservation stations supported by tagging (Courtesy of G. Sohi; reprinted with permission from *IEEE Transactions on Computers*, March 1990)

***Internal Data Forwarding***

- The throughput of a pipelined processor can be further improved with internal data forwarding among multiple functional units.
- In some cases, some memory-access operations can be replaced by register transfer operations.
- A *store-load forwarding* is shown in Fig. 6.13a in which the *load operation* (LD R2,M) from memory to register R2 ean be replaced by the move operation (MOVE R2, R1) from register R1 to register R2. Since register transfer is faster than memory access, this data forwarding will reduce memory traffic and thus results in a shorter execution time.
- Similarly, *Load-load forwarding* (Fig. 6.13b) eliminates the second *load* operation (LD R2, M) and replaces it with the *move* operation (MOVE R2, R1).

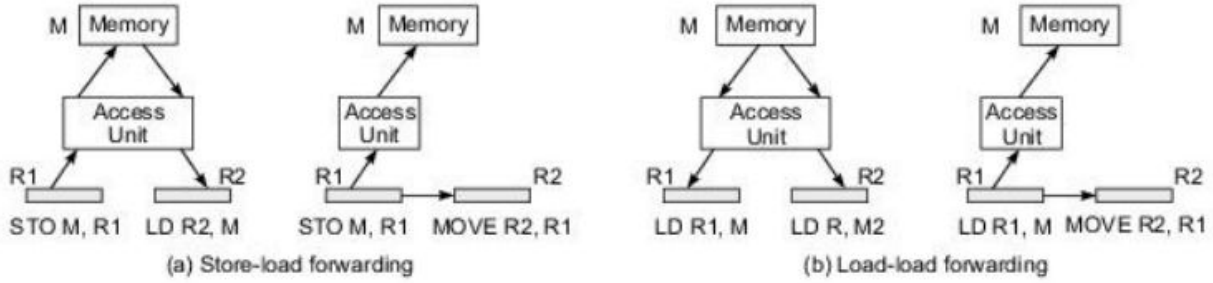



**Fig. 6.13** Internal data forwarding by replacing memory-access operations with register transfer operations

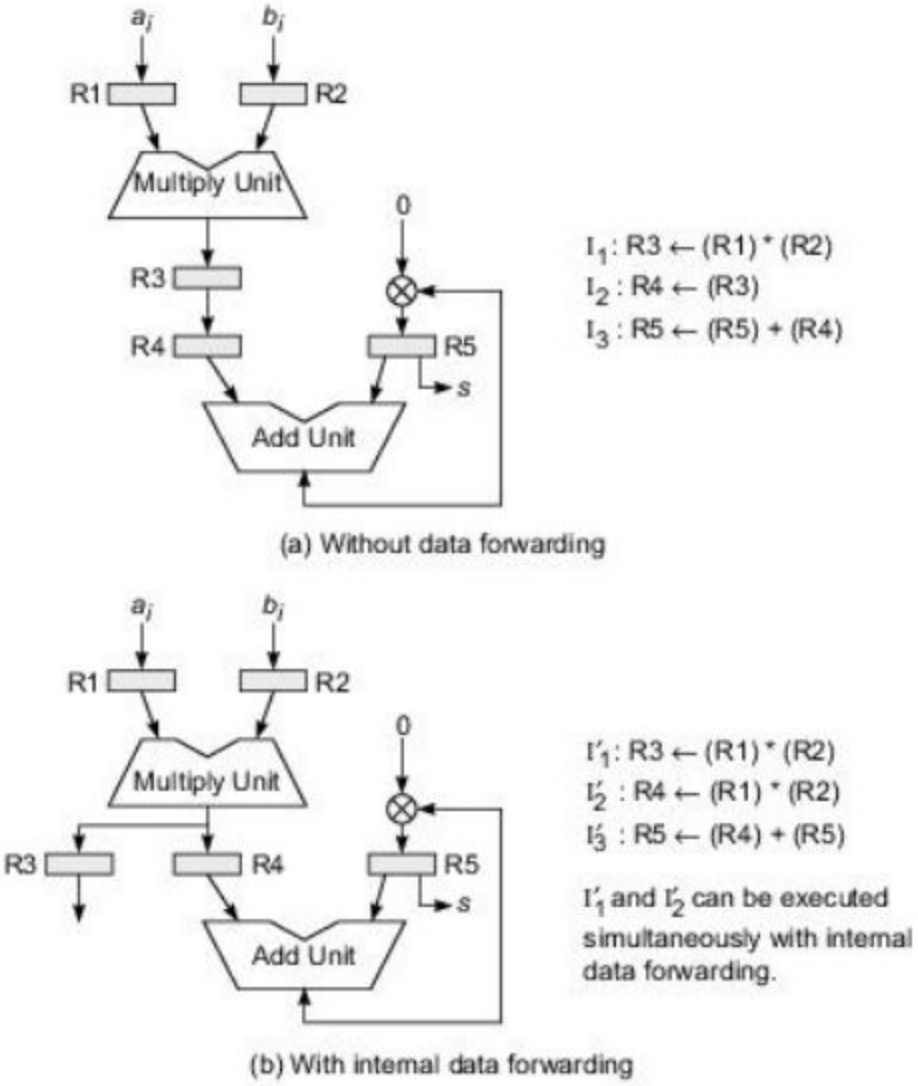



**Fig. 6.14** Internal data forwarding for implementing the dot-product operation

***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 I according to program order. If the actual execution order of these two instructions violates the program order. Incorrect results may be read or written, thereby producing hazards.
- Hazards should be prevented before these instructions enter the pipeline, such as by holding instruction J until the dependence on instruction I is resolved.
- We use the notation D(I) and R(I) for the *domain* and *range* of an instruction I.

- The domain contains the input set (such as operands in registers or in memory) to be used by instruction I. The range corresponds to the output set of instruction I.
- Listed below are the conditions under which possible hazards can occur:
    1. $R(I) \cap D(J) \neq \phi$ for RAW hazard
    2. $R(I) \cap R(J) \neq \phi$ for WAW hazard
    3. $D(I) \cap R(J) \neq \phi$ for WAR hazard
- This means the hazard may not appear even if one or more of the conditions exist.

**RAW hazard**

- When J attempts to read some data object that has been modified by I.
  Eg: I: Inc R0
  J: Mul Acc,R0

**WAW hazard**

- I & J attempt to modify the same data object
- Eg: I: Mul R1,R2
  J: add R1,R2

**WAR hazard**

- J attempts to modify some data object read by I
- Eg: I: Add R1,R2
  J: Mul R2,R3

**Note**

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

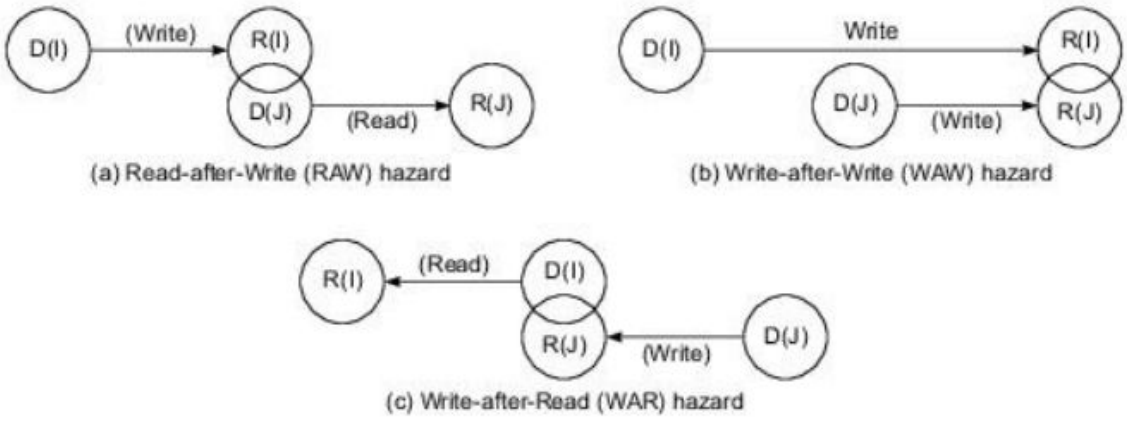



**Fig. 6.15** Possible hazards between read and write operations in an instruction pipeline (instruction I is ahead of instruction J in program order)

### 5.1.3 Dynamic Instruction Scheduling

- In this section, we describe three methods for scheduling instructions through an instruction pipeline.
- The *static scheduling* scheme is supported by an optimizing compiler.

- *Dynamic scheduling* is achieved using a technique such as *Tomasulo's register-tagging* scheme, or the *scoreboarding* scheme.

***Static Scheduling***

- Consider the execution of the following code fragment.

| Stage delay: | Instruction: | | |
|---|---|---|---|
| 2 cycles | Add | R0, R1 | /R0 ← (R0) + (R1)/ |
| 1 cycle | Move | R1, R5 | /R1 ← (R5)/ |
| 2 cycles | Load | R2, M($\alpha$) | /R2 ← (Memory ($\alpha$))/ |
| 2 cycles | Load | R3, M($\beta$) | /R3 ← (Memory ($\beta$))/ |
| 3 cycles | Multiply | R2, R3 | /R2 ← (R2) × (R3)/ |

- The multiply instruction cannot be initiated until the preceding load is complete. This data dependence will stall the pipeline for three clock cycles since the two loads overlap by one cycle.
- The two *loads,* since they are independent of the add and move, can be moved ahead to increase the spacing between them and the multiply instruction. The following program is obtained after this modification:

| | | |
|---|---|---|
| Load | R2, M($\alpha$) | 2 to 3 cycles |
| Load | R3, M ($\beta$) | 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.
- While the operands are being loaded from memory cells $\alpha$ and $\beta$ into registers R2 and R3, the two instructions add and move consume three cycles and thus pipeline stalling is avoided.

***Tomasulo's algorithm***

- This hardware dependence-resolution scheme was first implemented with multiple floating-point units of the IBM 360/91 processor.
- The algorithm has since become known as Tomasulo's Algorithm, after the name of its chief designer
- Let us assume that the functional units are internally pipelined and can complete one operation in every clock cycle.
- Therefore each functional unit can initiate one operation in every clock cycle—provided of course that a reservation station of the unit is ready with the required input operand value or values.
- Figure 12.12 shows such a functional unit connected to the common data bus, with three reservation stations provided on it.

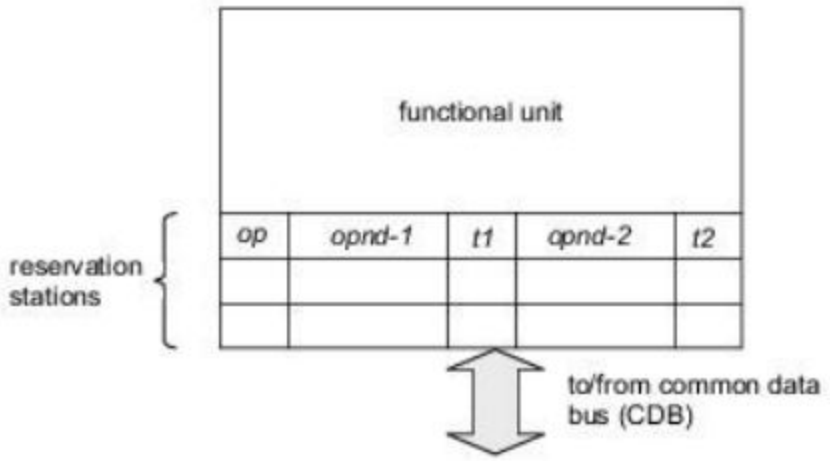



**Fig. 12.12** Reservation stations provided with a functional unit

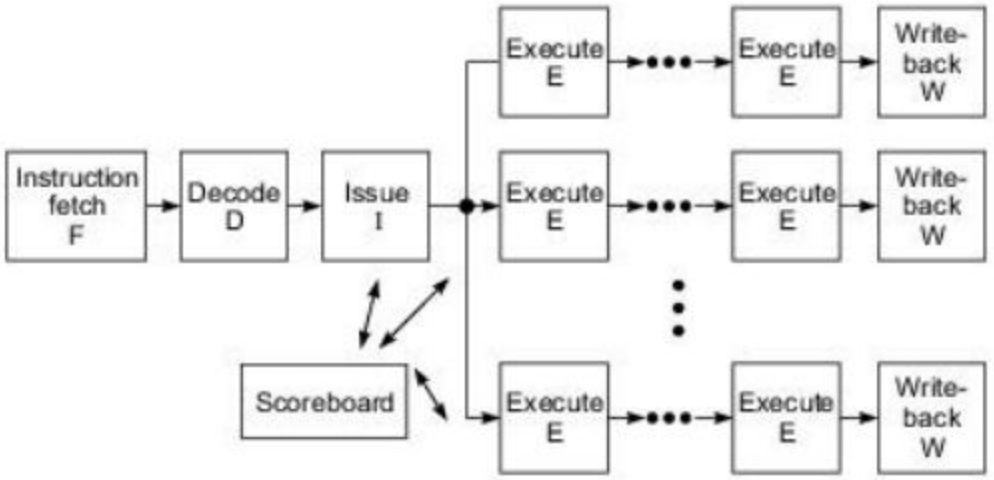
The various fields making up a typical reservation station are as follows:

| | |
|---|---|
| *op* | operation to be carried out by the functional unit |
| *opnd-1* & *opnd-2* | two operand values needed for the operation |
| *t1* & *t2* | two source tags associated with the operands |

- When the needed operand value or values are available in a reservation station, the functional unit can initiate the required operation in the next clock cycle.
- At the time of instruction issue, the reservation station is filled out with the operation. code {op}.
- If an operand value is available, for example in a programmable register, it is transferred to the corresponding source operand field in the reservation station.
- However, if the operand value is not available at the time of issue, the corresponding source tag (t1 and/or t2) is copied into the reservation station.
- The source tag identifies the source of the required operand.
- As soon as the required operand value is available at its source which would be typically the output of a functional unit—the data value is forwarded over the -common data bus, along with the source tag.
- This value is -copied into all the reservation station operand slots which have the matching tag.Thus operand forwarding is achieved here with the use of tags.
- All the destinations which require a data value receive it in the same clock cycle over the common data bus, by matching their stored operand tags with the source tag sent out over the bus.

***CDC Scoreboarding***

- The CDC 6600 was a nearly high-performance computer that used dynamic instruction scheduling hardware.



(a) A CDC 6600-like processor

- Figure 6.17a shows a CDC 6600-like processor, in which multiple functional units appeared as multiple execution pipelines.
- Parallel units allowed instructions to complete out of the original program order.
- Instructions were issued to available functional units regardless of whether register input data was available.
- The instruction would then wait in a buffer for its data to be produced by other instructions.
- To control the correct routing of data between execution units and registers, the CDC 6600 used a centralized control unit known as the scoreboard.
- This unit 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, it signaled the scoreboard to release the resources.
- The scoreboard is a centralized control logic which keeps track of the status of registers and multiple functional units.
- When functional units generate new results, some data dependences can be resolved and thus a higher degree of parallelism can be explored with scoreboarding.

### 5.1.4 Branch HandlingTechniques

- The performance of pipelined processors is limited by data dependences and branch instructions.

**Branch Prediction**

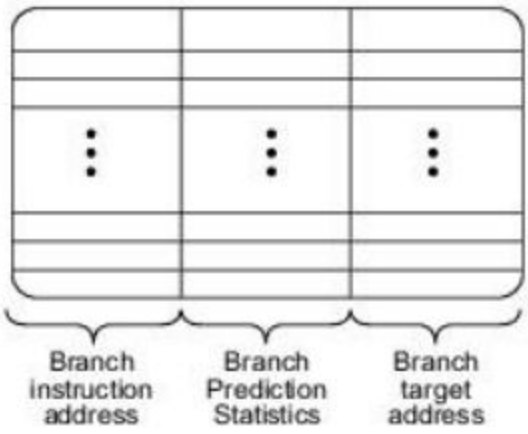
- A branch can be predicted using 2 methods
  - Static branch prediction➔ done by compiler
  - Dynamic branch prediction
- In static branch prediction,
  - Branch is predicted based on branch instruction type
  - Static strategy requires the collection of

        - frequency & probabilities of branch taken
        - Branch types across program
    - Static branch prediction not very accurate
- 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**



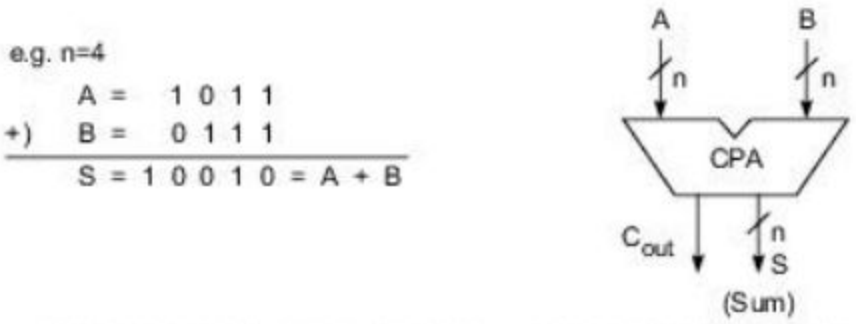
(a) Branch target buffer organization

- 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.

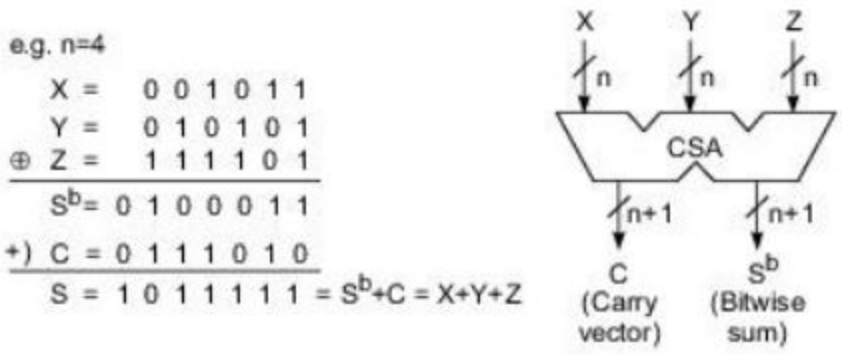
- It includes the following:-
    - address of the branch target
    - Address of branch instruction
    - Branch prediction statistics
- Prediction information is updated upon completion of the current branch
- BTB can be extended to store not only the branch target address but also the target instruction itself
    - This is to allow zero delay in converting conditional branches to unconditional branches.

## 5.2 ARITHMETIC PIPELINE DESIGN

- Pipelining techniques can be applied to speed up numerical arithmetic computations.
- Depending on the function to be implemented, different pipeline stages in an arithmetic unit require different hardware logic.
- High-speed addition requires either the use of a *carry-propagation adder (CPA*) which adds two numbers and produces an arithmetic sum, or the use of a *carry-save adder (CSA)* to “add” three input numbers and produce one sum output and a carry output.
- In a CPA, the carries generated in successive digits are allowed to propagate from the low end to the high end, using either ripple carry propagation or some carry looka-head technique.
- In a CSA, the carries are not allowed to propagate but instead are saved in a carry vector.




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




(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)

- We use the CPA and CSA to implement the pipeline stages of a fixed-point multiply unit as follows.
- ***Multiply Pipeline Design*** Consider as an example the multiplication of two 8-bit integers A x B = P, where P is the 16-bit product.
- This fixed-point multiplication can be written as the summation of eight partial products as shown below: P = A x B = P0 + P1 + P2 +.... + P7, where x and + are arithmetic multiply and add operations, respectively.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | = | $A$ |
| | | | | | | ×) | 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 | = | $P_7$ |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | = | $P$ |

- The summation of the eight partial products is done with a *Wallace tree* of CSAs plus a CPA at the final stage,
- The first stage (S1) generates all eight partial products, ranging from 8 bits to 15 bits. simultaneously.
- The second stage (S2) is made up of two levels of four CSA.s, and it essentially merges eight numbers into four numbers ranging from I3 to I5 bits.
- The third stage (S3) consists of two CSAs, and it merges four numbers from S1 into two 16-bit numbers.

- The final stage (S4) is a CPA, which adds up the last two numbers to produce the final product P.

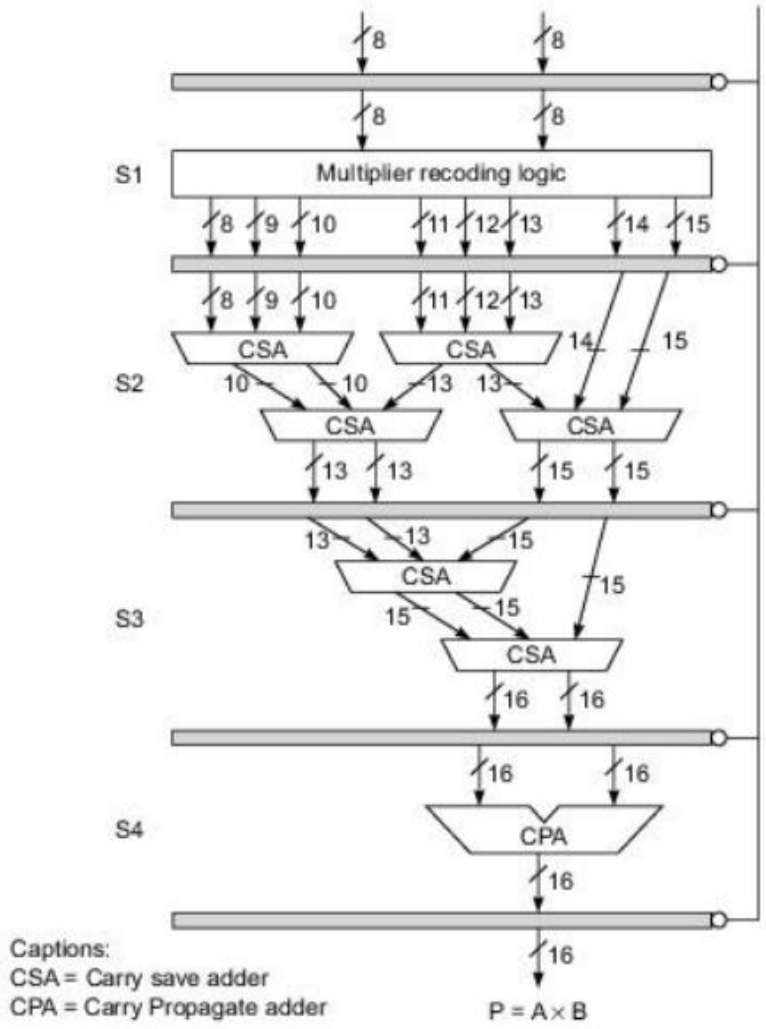



**Fig. 6.23** A pipeline unit for fixed-point multiplication of 8-bit integers (The number along each line indicates the line width.)