**MODULE V**

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

# INSTRUCTION PIPELINE DESIGN

**Qn:What is instruction pipeline? Explain in detail.**

- Instruction pipeline processor is processor which executes different instructions belonging to a process in its different stages.
- A stream of instructions can be executed by a pipeline in an overlapped manner.

**Techniques for improving pipelined processor performance**

1. **Instruction Execution Phases(pipelined Instruction processing)**

2. **Mechanisms for Instruction Pipeline**

   **2.1 Pre-fetch Buffers**
   **2.2 Multiple Functional Units**
   **2.3 Internal Data Forwarding**
   **2.4 Hazard avoidance**

3. **Dynamic Instruction scheduling techniques**

   **3.1 Tomasulo's algorithm**
   **3.2 CDC Scoreboarding**

4. **Branch Handling Techniques**

1.**Instruction Execution Phases**

The phases are ideal for overlapped execution of a linear pipeline, each phase may require one or more clock cycles to execute depending on the instruction type and processor/memory architecture.

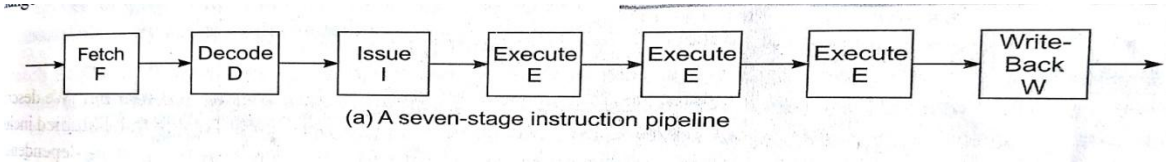
**Fetch Stage** – fetches instr's from a cahe memory, ideally one/cycle

**Decode Stage** – reveals instr function to be performed and identifies the resources needed. Resources include registers, buses, Functional Units…

**Issue Stage** – reserves resources, operands are read from registers

**Execute Stage** – instr's are executed in one or several execute stages.

**Writeback -** stage is used to write results into registers.



(a) A seven-stage instruction pipeline

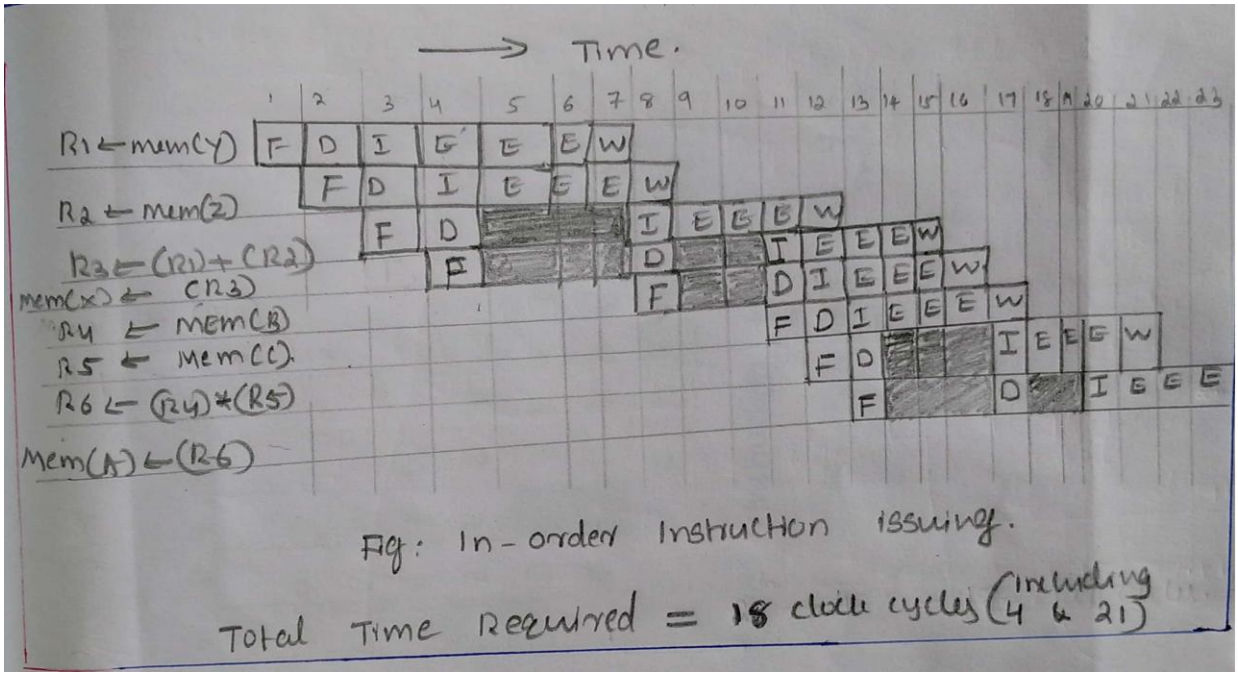


Fig: In-order Instruction issuing.

Total Time Required = 18 clock cycles (Including 4 & 21)

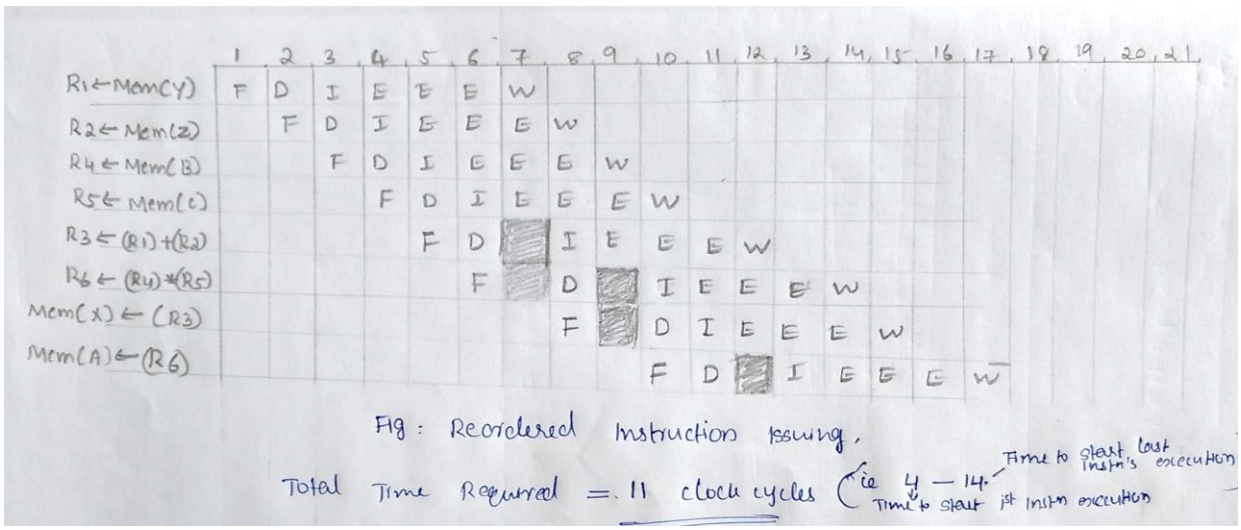


Fig: Reordered Instruction issuing.

Total Time Required = 11 clock cycles (ie 4 — 14. Time to start last instn's execution; Time to start 1st instn execution)

Pipelined execution of X = Y + Z and A = B × C

- Figure above shows the flow of machine instructions through a typical pipeline. These **eight instructions** are for pipelined execution of the high-level language statements
  X = Y + Z and A = B * C.
- Here we have assumed that **load and store instructions take four execution clock** cycles, while floating-point **add and multiply operations take three cycles.**

- Figure b (above)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 dependences.
- The first two load instructions issue on consecutive cycles.
- The **add** is dependent on both loads and must wait three cycles before the data (Y and Z) are loaded in.
- Similarly, the **store** of the sum to memory location X must wait three cycles for the add to finish due to a flow dependence.

There are similar blockages during the calculation of A. The total time required is 18 clock cycles. **This time is measured beginning at cycle 4 when the first instruction starts execution until cycle 21 when the last instruction starts execution.** This timing measure eliminates the undue effects of the pipeline “startup" or “draining” delays.

Figure c (above) 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 multiply 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.

## 2. MECHANISMS FOR INSTRUCTION PIPELINING

**2.1 Pre-fetch Buffers**

**2.2 Multiple Functional Units**

**2.3 Internal Data Forwarding**

**2.4 Hazard avoidance**

### 2.1 . PREFETCH BUFFERS

**Qn: What type of buffers are used to match the instruction fetch rate to the pipeline consumption rate?**

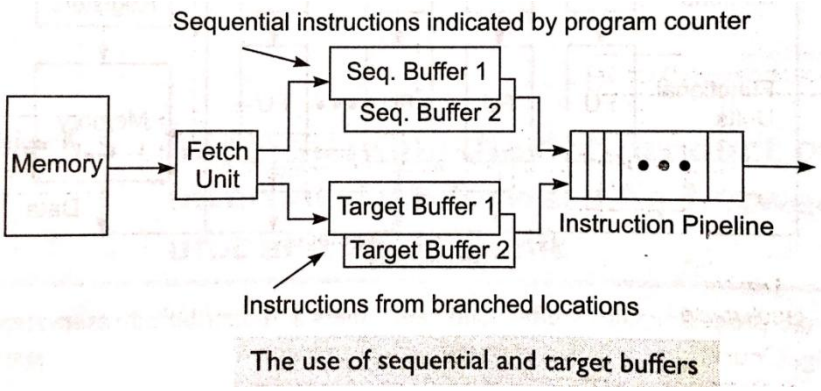
**Qn: Discuss about prefetch buffers?**

3 type’s of buffers are used to match instr fetch rate to pipeline consumption rate.

1. **Sequential buffer**

2. **Target buffer**
3. **Loop buffer**

In a single memory access time, a block of consecutive instr's are fetched into prefetch buffer.




The use of sequential and target buffers

- Sequential inst's are loaded into a pair of **sequential buffers** for in-sequence pipelining.
- Instr's from Branch target are loaded into pair of **target buffers** for out of sequence Pipelining.
- Both buffers operate in FIFO (First In First Out)

- A conditional branch instr causes both sequential buffers and target buffers to fill with instr. After checking branch condition, appropriate instruction will be taken from one of the two buffers and instr in other buffer is discarded.
- From each buffer pair, **one may be used to load instr's from memory and another buffer may be used to feed instr's into pipeline**, thus it prevents collision btw instr flowing IN and OUT of pipeline.

- **Loop Buffer** – holds sequential instr's in a loop. Its maintained by Fetch Stage.
- Pre fetched instr's in a loop body will be executed repeatedly until all iterations complete execution.

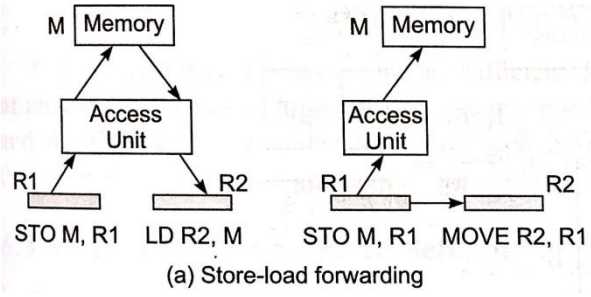
**Loop Buffer operates in 2 steps:**

- First it contains instrs sequentially ahead of current instr, thus saves instr fetch time from memory.
- It recognizes when target of a branch falls within loop boundary. In this case, unnecessary memory accesses can be avoided if the target instruction is already in the loop buffer.

### 2.2. Internal Data Forwarding

- Memory access operations can be replaced by register transfer operations to save time

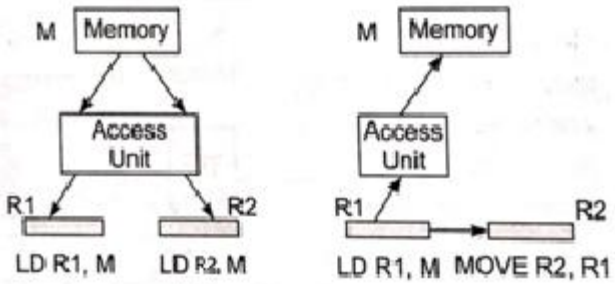
- **Store Load Forwarding**



(a) Store-load forwarding

load operation(LD R2,M) from memory to register R2 can be replaced by move operation (MOVE R2,R1) from register R1 to R2, since register transfer is faster than memory access and will also reduce memory traffic , resulting in shorter execution time.

- **Load-Load Forwarding**
-



(b) Load-load forwarding

We can eliminate 2nd load operation(LD R2,M) and replace it with move operation(MOVE r2, R1)

**Example :problem:**

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

One can feed the output of a multiplier directly to the input of an adder (Fig. 6.14) for implementing the following dot-product operation:

$$s = \sum_{i=1}^{n} a_i \times b_i$$

Without internal data forwarding between the two functional units. the three instructions must be sequentially executed in a looping structure [Fig. a below).

With data forwarding, the output of the multiplier is fed directly into the input register R4 of the adder (Fig. b). At the same time, the output of the multiplier is also routed to register R3. internal

data forwarding between the two functional units thus **reduces the total execution time through the pipelined processo**r.

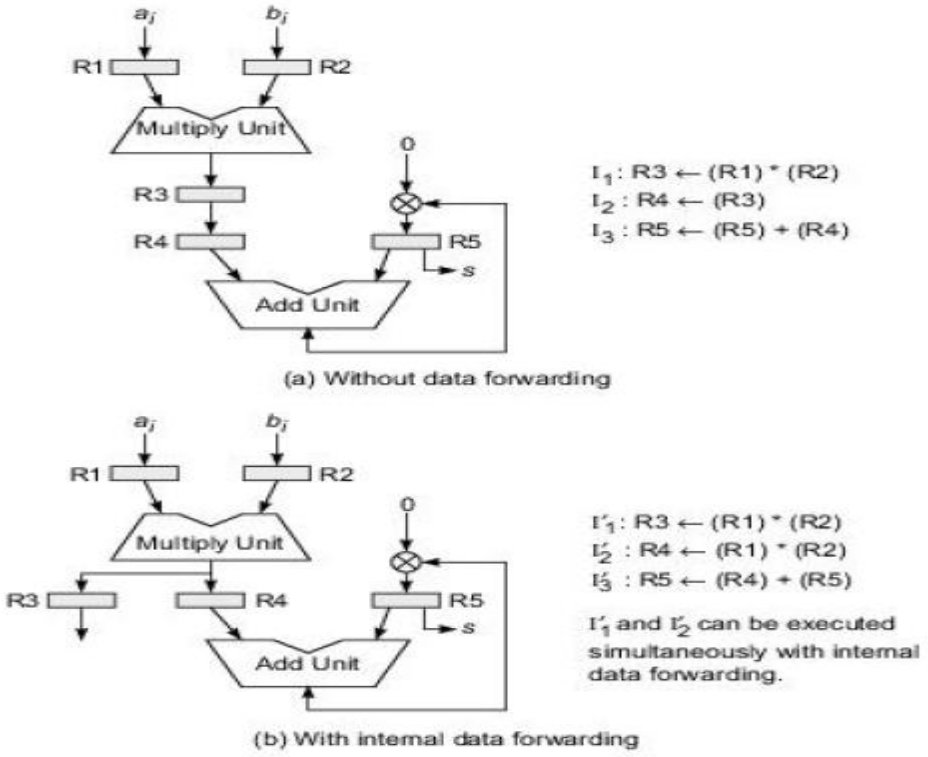



**Fig. 6.14** Internal data forwarding for implementing the dot-product operation

### 2.3.Multiple Functional Units

**Qn: Explain a pipeline processor with multiple functional units?**

- Certain pipeline **stage becomes bottleneck** – **this stage corresponds to row with maximum number of checkmarks in the reservation table**
- Can be solved using **multiple copies of same stage simultaneously ,** ie:- use of multiple execution units in a pipelined processor design.

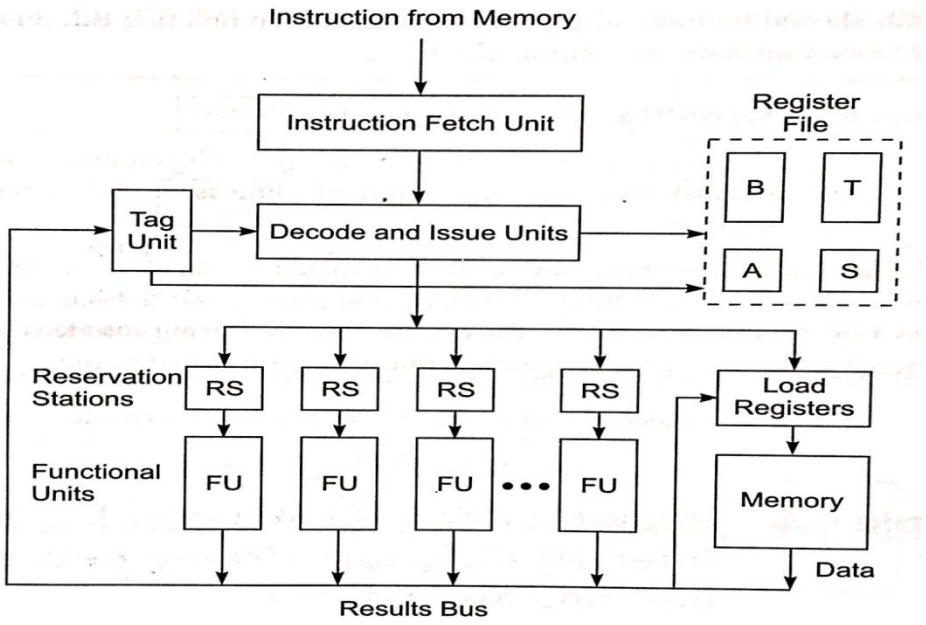



**FIG – A pipelined processor with multiple FU's and distributed RS's supported by Tagging.**

- In order to resolve data or resource dependences among successive instructions entering the pipeline the **Reservation Stations(RS)** are used with each functional unit.
- **Operands wait in RS** until data dependences are resolved.
- Each RS is uniquely identified by a **Tag monitored by tag unit.**
- Tag unit keeps checking the tags from all currently used registers or RS's
- **RS** also serve as **buffers to interface pipelined functional units with decode and issue unit**
- Multiple Functional units operate in parallel once dependencies are resolved.

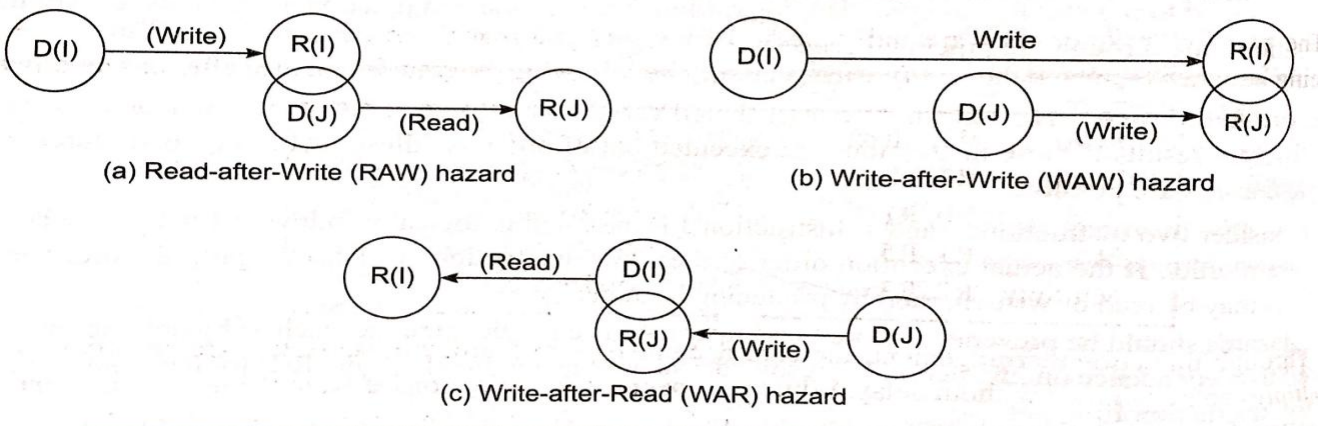
## 2.4.Hazard avoidance

**Qn: Define hazards with respect to pipeline. Describe the various categories of the hazards .**

**Qn:**

- 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**. As shown in figure below:-




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

- Consider instr's I and J, J assumed to logically follow instr 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.
- D is the domain of instr –input set D(I) and D(J)
- R is the range of instr – output set

**Conditions that cause Hazards**

- $R(I) \cap D(J) \neq \phi$ for **RAW hazard- flow dependence**

  Ex:

  I : a = b + c (R(I)∩D(J)→a)

  J : e =a + d

- $R(I) \cap R(J) \neq \phi$ **for WAW hazard – o/p dependence**

  Ex:

  I: a=b+c (R(I)∩R(J) →a)

  J: a=d+e

- $D(I) \cap R(J) \neq \phi$ **for WAR hazard – anti-dependence**

  EX:

  I: a=b+c ((D(I)∩R(J) → b)

  J:b=a+d

- **Resolution of hazard conditions** can be checked by special hardware while instr's are being loaded into prefetch buffer

- **Special tag bit** can be used with each operand register to **indicate safe or hazard.**

**3. DYNAMIC INSTRUCTION Scheduling**

**Qn: What are the methods for scheduling instructions through an instruction pipeline.**

**Qn: What is dynamic scheduling?**

**2 techniques**

1. **Tomasulo's Algorithm**

### 2. CDC scoreboarding

**STATIC Scheduling**

- Data dependencies in sequence of instructions create **interlocked relationships.**
- Inter-locking is resolved through **compiler based static scheduling approach**
- **Compiler or post-processor** can be used to **increase 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($\alpha$) | /R2 ← (Memory ($\alpha$))/ |
| 2 cycles | Load | R3, M($\beta$) | /R3 ← (Memory ($\beta$))/ |
| 3 cycles | Multiply | R2, R3 | /R2 ← (R2) × (R3)/ |

- Considering the execution of code, multiply instr cannot be started until preceding load is complete.
- It stalls pipeline for 3 clock cycles(since 2 loads overlap by one cycle)
- **The 2 loads are independent of add and Move instr.** We can move it ahead to increase spacing btw them and multiply instr.

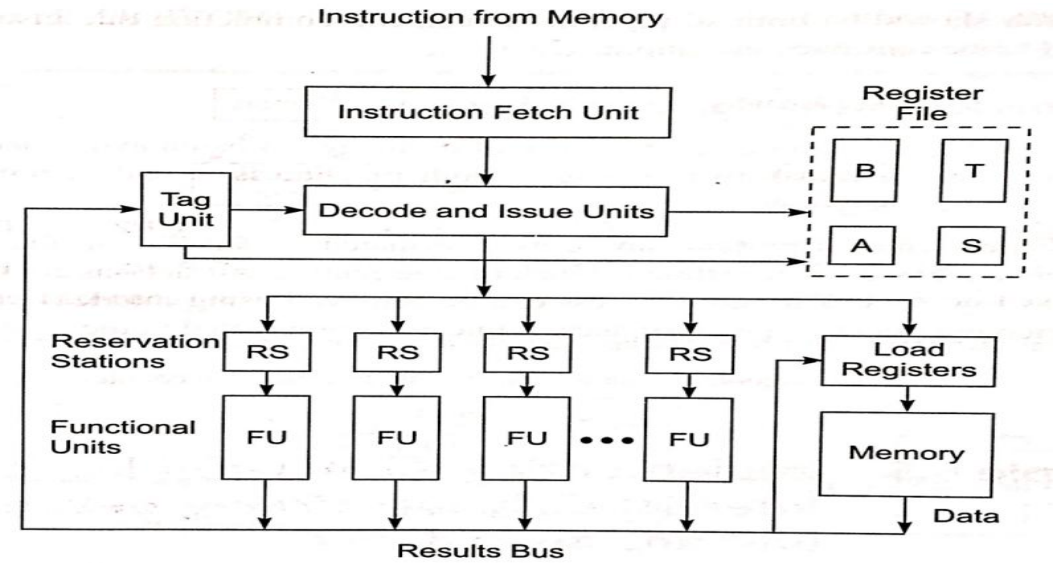
| | | |
|---|---|---|
| Load | R2, M($\alpha$) | 2 to 3 cycles |
| Load | R3, M ($\beta$) | 2 cycles due to overlapping |
| Add | R0, R1 | 2 cycles |
| Move | R1, R5 | 1 cycle |
| Multiply | R2, R3 | 3 cycles |

- Through this code rearrangement data dependences and program semantics are preserved and multiply can be started without delay
- **Compiler based s/w interlocking** is **cheaper** to implement and **flexible** to apply.

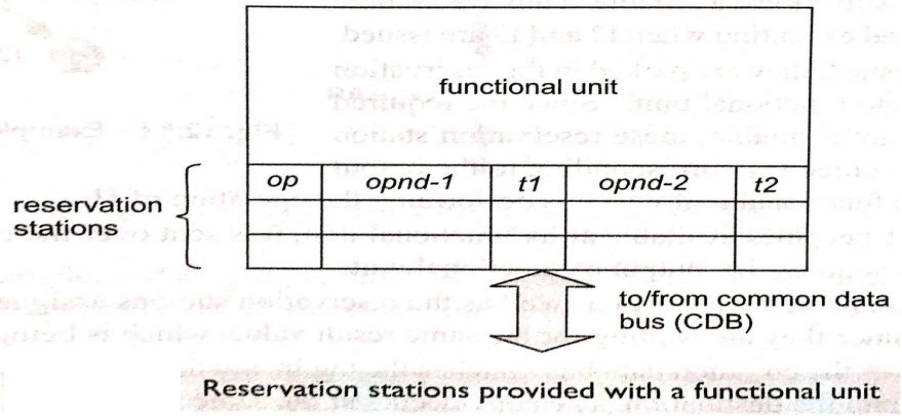
### 3.1. Tomasulo's algorithm scheme

- Named after the chief designer.
- This **hardware dependence –resolution scheme** was first implemented with **multiple floating point units** of the IBM 360/91 processor.
- Functional units are internally pipelined and can complete one operation in every clock cycle, **provided the reservation station(Structure of RS shown below) of the unit is ready with the required input operand values.**

- If source register is busy when an instr reaches issue stage, **tag for source register is forwarded to RS**
- When register becomes available **tag can signal availability.**
- This value is copied into all reservation station which have the matching tag. **Thus operand forwarding is achieved here with the use of tags.**
- All destinations which require a data value receive it in the same clock cycle over the common data bus, by matching stored operand tags with source tag sent over the bus.




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)




Reservation stations provided with a functional unit

- The fig above shows a **functional unit connected to common data bus with three reservation stations provided on it.**

**Op** – operation to be carried out

**OPnd-1 and Opnd-2** – two operand values needed for operation

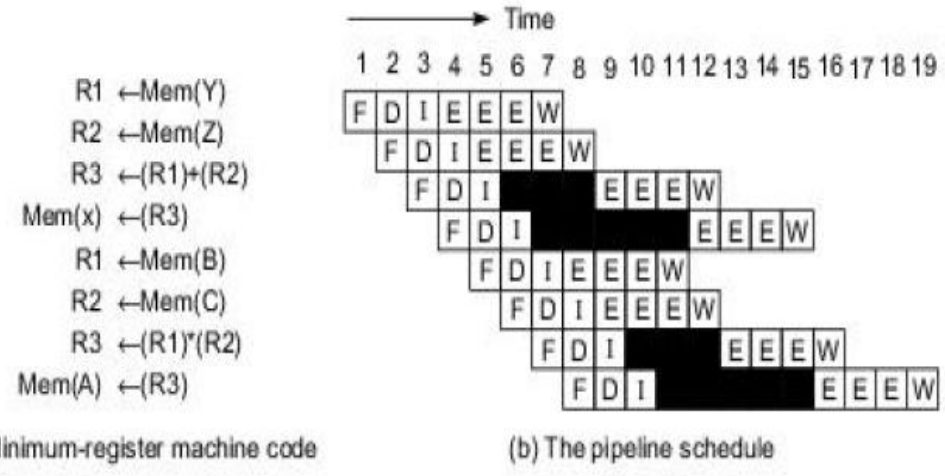
**t1 and t2** – two source tags associated with the operands

- When the needed operand values are available in reservation station, the functional unit can initiate the required operation in the next clock cycle.
- At time of instruction issue the **reservation station** is filled out with the operation code(op).
    - **if an operand value is available** in programmable register it is transferred to the corresponding source operand field in the reservation station. It waits until its data dependencies are resolved and operands become available. Dependence is resolved by monitoring Result bus and when all operands of an instr are available its dispatched to functional unit for execution.

    - **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 is available at its source- typically output of functional unit – the data value is forwarded over the common data bus along with source tag.

## ***Example:Tomasulo's algorithm for dynamic instruction scheduling***

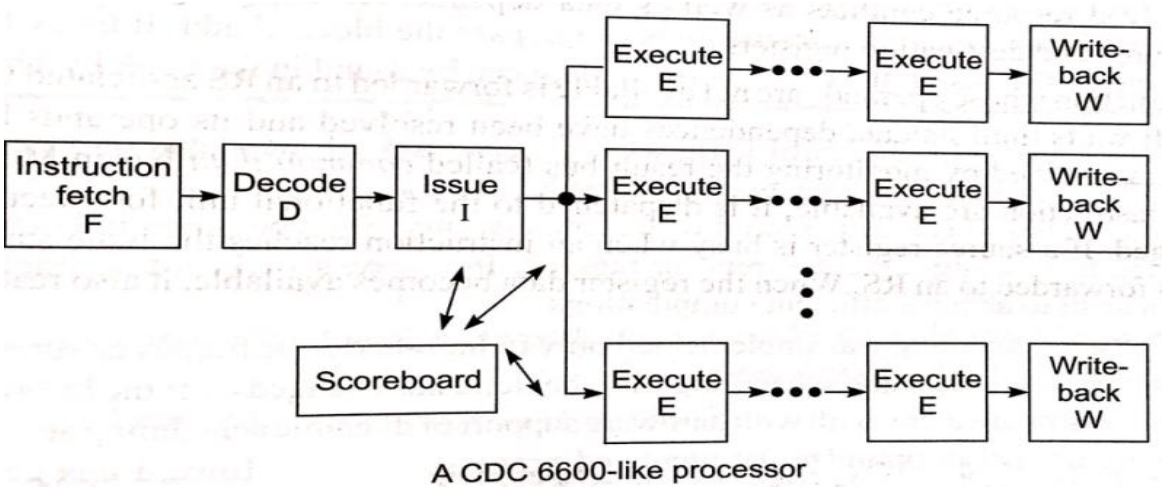
- Tomasulo's algorithm was applied to work with processors having a few floating-point registers. In the ease of Model 91, only four registers were available.
- Fig a below shows a minimum-register machine code for computing X = Y + Z and A = B * C. The pipeline timing with Tomasulo's algorithm appears in Fig.-b.
- Here, the total execution time is **13 cycles**, counting from cycle 4 to cycle 15 by ignoring the pipeline startup and draining times.




5 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)

Memory is treated as a special functional unit. When an instruction has completed execution, the result (along with its tag} appears on the result bus. The registers as well as the RSs monitor the result bus and update their contents (and ready/busy bits) when a matching tag is found.

### 3.2. CDC SCOREBOARDING



A CDC 6600-like processor

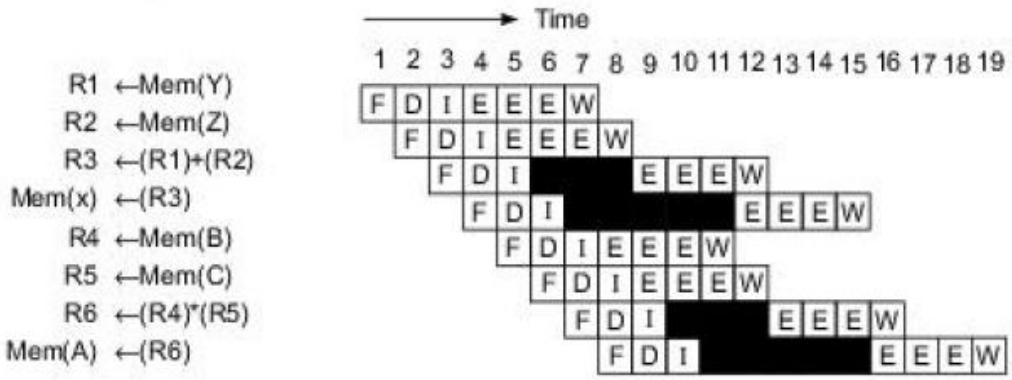
- Figure above shows CDC6600 like processor that uses dynamic instruction scheduling hardware,.
- **Here Multiple Functional units appeared as multiple execution units** pipelines.
- Parallel units allow instr's to complete out of original program order.
- The processor had **instr buffers** for **each execution unit**
- **Instrs** are issued to available FU's regardless of **whether register i/p data** are available.
- To Control correct routing of data btw execution units and registers CDC 6600 used a Centralized Control unit known as **scoreboard.**
- **Scoreboard kept track of registers needed by instrs waiting for various functional units**
- When all registers have valid data scoreboard enables instr execution.
- When a **FU finishes it signals scoreboard to release the resources.**
- **Scoreboard is a Centralized control logic** which keeps track of status of registers and multiple functional units.

***Example: Pipelined operations using hardware scoreboarding***

**Execution of the same machine code for X = Y + Z and A= B * C.**

- The pipeline latencies are the **same as those resulting from Tomasulo's algorithm.**
- The add instruction is issued to its functional unit before its registers are ready.
- lt 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.
- It takes **13 clock cycles** to perform the operations.




(b) The improved schedule from Fig. 6.9b

Hardware scoreboarding for dynamic instruction scheduling (Courtesy of James Smith; reprinted with permission from *IEEE Computer*, July 1989)

## 4. BRANCH HANDLING TECHNIQUES

**Qn:What are the branch handling techniques**

**The performance of pipelined processors is limited by 2 factors**

**Data dependences** — **Branch instructions.**

***(Discussed in previous sections)***

Here we are going to discuss the **effects of branching** on performance of pipelined processors.

TERMS :

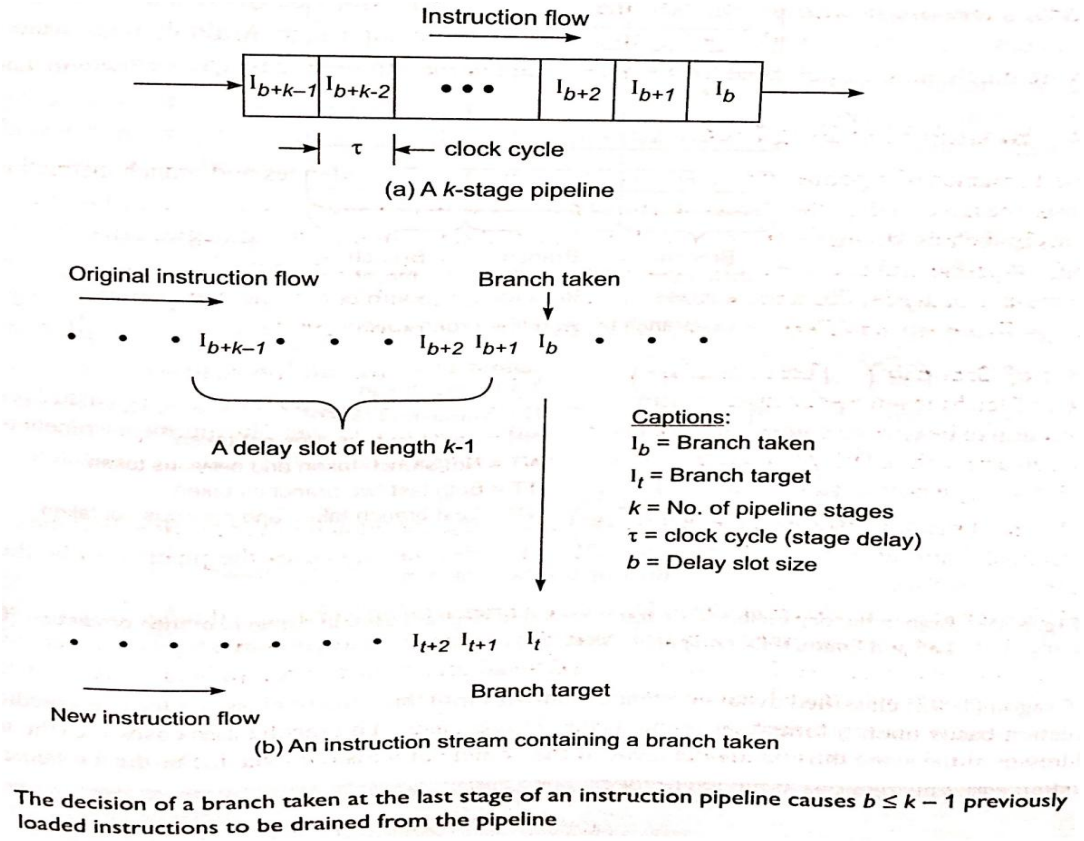
- **Branch Taken** : the action of fetching a non sequential or remote instruction after a branch instruction is called Branch taken.
- **Branch Target** : the instruction to be executed after a branch taken is called Branch target

  **Delay slot** : the no: of pipeline cycles wasted between a branch taken and its branch target is called delay slot, denoted by b

  $0 \leq b \leq k-1$ (K-no of pipeline stages)

**EFFECT OF BRANCHING**

- When a branch is taken all instructions following the branch in the pipeline becomes useless and will be drained from the pipeline. Thus branch taken causes a pipeline to be flushed losing a number of pipeline stages.




The decision of a branch taken at the last stage of an instruction pipeline causes $b \leq k-1$ previously loaded instructions to be drained from the pipeline

**Branch taken causes $I_{b+1}$ through $T_{b+k-1}$ to be drained from pipeline.**

- Let **p** be the probability of a conditional branch in instruction stream
- **q** be the probability of successfully executed conditional branch (ie:-branch taken)
- typical values **p=20% and q=60%** have been observed in some programs
- **Penalty Paid** by branching equal to **pqnbτ** because each branch taken costs **bτ** extra pipeline cycles.

**Calculating Total Execution Time for n instr's including effect of branching ( $T_{eff}$)**

**$T_{eff}$= kτ+(n-1) τ+pqnbτ**

Calculating pipeline throughput ( $H_{eff}$) with influence of branching

**$H_{eff}$ = n / $T_{eff}$ = nf / k+n-1+pqnb**

When n→∞ and tightest upperbound is obtained when b=k-1

**$H^*_{eff}$= f / pq(k-1)+1**

When no branching **p=q=0**

Maximum throughput **$H_{eff}$= f = 1 / τ**

EX: suppose p=.2 and q=.6 and b=k-1=7

**We define Performance Degradation Factor :**

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

(ie **pipeline performance degrades by 46% with branching** when instruction stream is sufficiently long. The above analysis demonstrates the high degree of performance degradation caused by branching in an instruction pipeline.)

**BRANCH PREDICTION ( Static and Dynamic )**

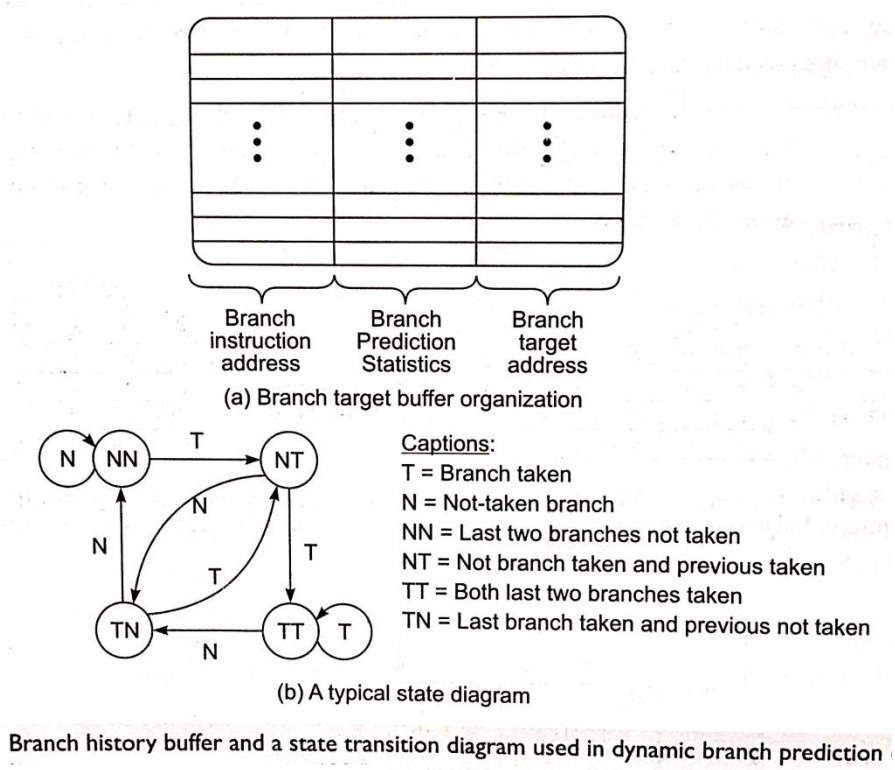
**Static Branch Prediction Strategy**

- Branches are predicted based on branch code types statically or based on branch History during program execution.
- The frequency and probabilities of branch taken and branch types across large no of pgm traces is used to predict a branch.
- Such a Static branch strategy may not be very accurate.
- The static prediction direction (taken or nor rat-en) can even be wired into the processor.
- The wired-in static prediction cannot be changed once committed to the hardware.

**Dynamic Branch Prediction Strategy-** works better than static)

- Uses recent **branch history to predict** whether or not the branch will be taken next time when it occurs.
- To be accurate we may need to use entire histoty of branch to predict future choice – but not practical. Thus we use limited recent history.

- Requires additional hardware to keep track of the past behavior of the branch instructions at run time.
- **Branch Target Buffers** are used to implement branch prediction – BTB holds recent branch information including address of branch target used. The address of branch instr locates its entry in BTB.




Branch history buffer and a state transition diagram used in dynamic branch prediction

- The state transition diagram is used for backtracking the last 2 branches in a given program. The BTB entry contains backtracking information which will guide the prediction.
- BTB can store not only Branch Target address but also **target instruction itself** and few successor instr's, in order to allow zero delay in converting conditional branches to unconditional branches.

**DELAYED BRANCHES**

- The **branch penalty can be reduced** if **delay slot is shortened or minimized to zero penalty.**
- The purpose of delayed branches is to make this Possible, as illustrated in Fig. below
- **A delayed branch of *d cycles allows at most d-1* useful instr's** to be executed following branch taken.
- The execution of these instr's should be **independent of the outcome of branch instruction.**

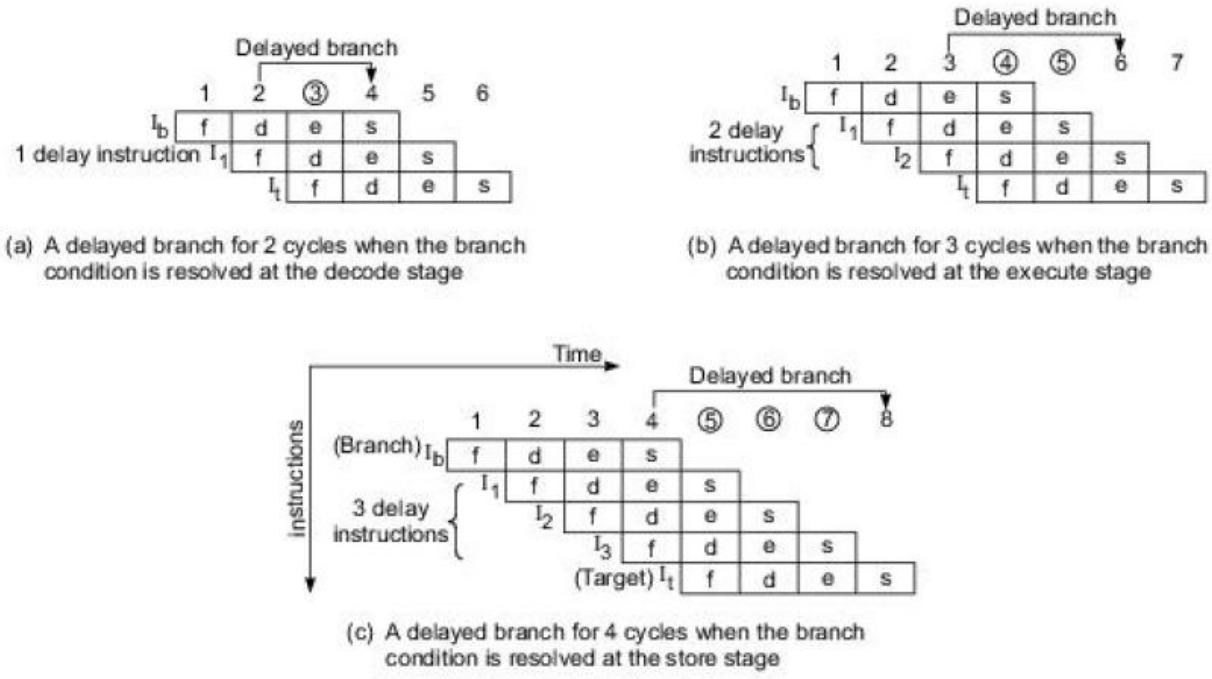



**Fig. 6.20** The concept of delayed branch by moving independent instructions or NOP fillers into the delay slot of a four-stage pipeline

(According to some program trace results.)

- The probability of moving one instruction (if = 2 in Fig. a) into the delay slot is > 0.6.
- The probability of moving two instructions (d = 3 in fig b) is about 0.2,
- and that of moving three instructions (d= 4 in Fig. c) is less than 0.1.
- From the above analysis, **one can conclude that delayed branching may be more effective in short instruction pipelines with about four stages.**

***Example: A delayed branch with code motion into a delay slot***

- Code motion across branches can be used to achieve a delayed branch, as illustrated in fig below.
- Consider the execution of a code fragment in Fig. 6.21 a. The original program is modified by moving the useful instruction Il into the delay slot after the branch instruction I3.

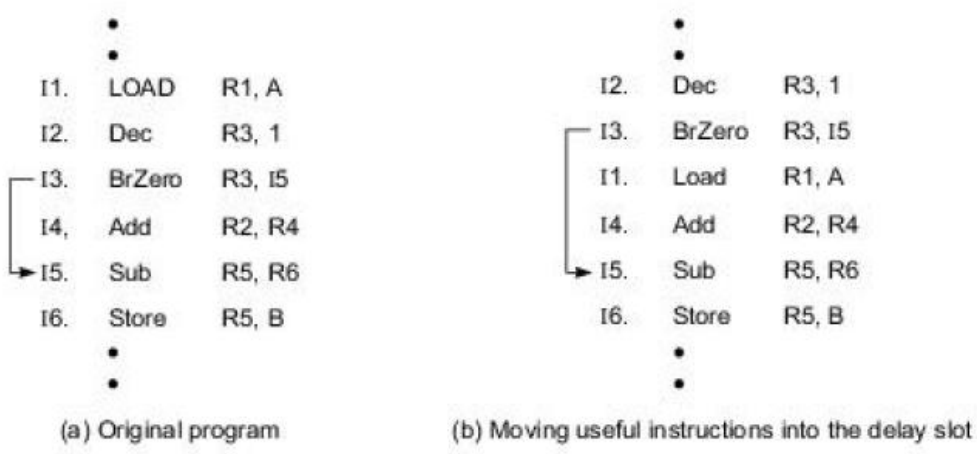



**Fig. 6.21** Code motion across a branch to achieve a delayed branch with a reduced penalty to pipeline performance

- In case branch is not taken , execution of modified pgm produces the same result as original pgm.
- In case branch is taken in modified pgm ( fig b) execution of delayed instr I1 and I5 is needed anyways.

## 5.2 ARITHMETIC PIPELINE DESIGN

**Qn:What is arithmetic pipleline?**

### 5.2.1 Computer arithmetic Principles

- In a digital computer, arithmetic is performed with finite precision due to the use of fixed-size memory words or registers.
- Fixed-point or integer arithmetic offers a fixed range of numbers that can be operated upon. Floating-point arithmetic operates over a much increased dynamic range of numbers.
- In modem processors, fixed-point and floating-point arithmetic operations are very often performed by separate hardware on the same processor chip.

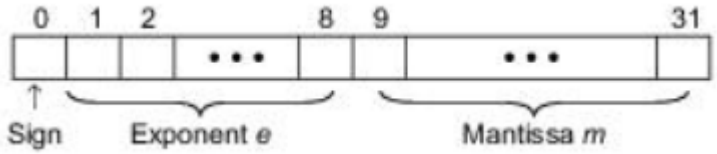
**FIXED-POINT OPERATIONS**

- Fixed-point numbers are represented internally in machines in sign-magnitude, ones compliment or 2's compliment notation.
- Most computers use the two's complement notation because of its unique representation of all numbers (including zero).
- Add, subtract, multiply and divide are the four primitive arithmetic operations.

- For fixed-point numbers, the add or subtract of two n-bit integers (or fractions) produces an n-bit result with at most one carry-out.
- The multiplication of two n-bit numbers produces a 2n-bit result which requires the use of two memory words or two registers to hold the full-precision result.
- Only an approximate result is expected in fixed-point division with rounding or truncation.

**Floating Point Numbers**: A floating-point number X is represented by a pair (m , e) , when m is the mantissa and e is the exponent with an implied base (radix).The algebraic value is represented as **X=m*$r^e$** . The sign of X can be embedded in the mantissa.

**IEEE 754 floating-point standard**




A binary base is assumed with r = 2. The 8-bit exponent ***e*** field uses an excess – 127 code. The dynamic range of e is (-127, 128), internally represented as (0, 255). The sign **s** and the 23-bit mantissa field **m** form a 25-bit sign-magnitude fraction, including an implicit or "hidden" l bit to the left of the binary point. Thus the complete mantissa actually represents the value 1.m in binary.
This hidden bit is not stored with the number . If 0<e< 255 , then a nonzero normalized number represents the following algebraic value:

**X= $(-1)^s$x $2^{e-127}$x (1.m)**

**<u>Floating – Point Operations</u>**

The four primitive arithmetic operations are defined below for a pair of floating point numbers represented by **X = ($m_x$ ,$e_x$) and Y = ($m_y$ ,$e_y$) .** For clarity , we assume $e_x < e_y$

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

The above equations clearly identify the number of arithmetic operations involved in each floating-point function. These operations can be divided into two halves: One half is for exponent operations such as comparing their relative magnitudes or adding/subtracting them; the other half is for mantissa operations,

including four types of fixed-point operations. Arithmetic shifting operations are needed for equalizing the two exponents before their mantissas can be added or subtracted.

### 5.2.2 STATIC ARITHMETIC PIPELINES

The arithmetic /logic units (ALUs) perform fixed-point and floating-point operations separately.

- The **fixed-point unit** is also called the **integer unit**.
- The **floating-point unit** can be built either **as part of the central processor or on a separate coprocessor.**

These arithmetic units perform **scalar operations** involving one pair of operands at a time. And **vector operations.** Scalar and vector arithmetic pipelines differ mainly in the areas of **register files** and **control mechanisms** involved.

- **Vector hardware pipelines** are often built as **add-on options to a scalar processor** or as an attached processor driven by a control processor.

**Arithmetic Pipeline Stages**:

- Since all arithmetic operations (such as add, subtract,multiply,divide,squaring,square rooting ,logarithm etc,) can be implemented with the basic **add and shifting operations**, **the core arithmetic stages require some form of hardware to add and to shift.**

For example, a typical **three stage floating-point adder** includes the following stages:-

1. first stage for **exponent comparison and equalization** which is implemented with an integer adder and some shifting logic;
2. a second stage for **fraction addition** using a high-speed carry lookahead adder;
3. and a third stage for **fraction normalization and exponent readjustment** using a shifter and another addition logic.**(Fraction normalization means, Shifting mantissa to right and incrementing exponent by 1 to get a non zero value after fraction)**

Eg:$X=0.9504*10^3$

$Y=0.8200*10^2$

After Step1: $X=0.9504*10^3$

$Y=0.08200*10^3$

Step2:Add

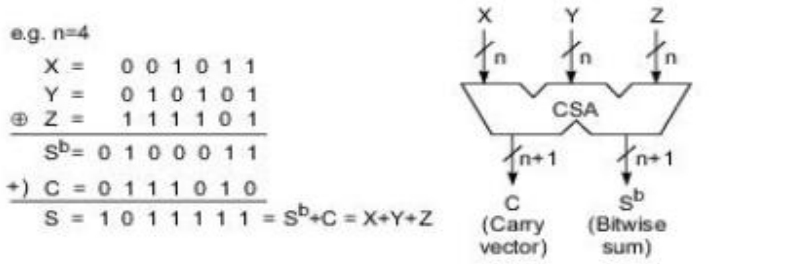
Result=$1.0324*10^3$

Step 3: fraction normalization and exponent readjustment

Result=$0.10324*10^4$

- Arithmetic or logical shifts can be easily implemented with **shift registers**.
- High-speed addition requires either the use of a **carry-propagation adder (CPA**) which **adds two numbers** and **produces an arithmetic sum** as shown in Fig. a below,
    - 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 lookahead technique.
- or the use of a **carry-save adder (CSA)** to **three input numbers** and produce **one sum** output and a **carry output** as exemplified in Fig. b. below
    - 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)

- In general, an n-bit CSA is specified as follows: Let X, Y, and Z be three n-bit input numbers, expressed as X=(Xn-1,Xn-2 …,X1,X0) and so on. The CSA performs bitwise operations simultaneously on all columns of digits to produce two n bit output numbers , carry and sum denoted as $\boldsymbol{S^b = 0, Sn-1, Sn-2, \ldots, S1, S0}$) **and C= (Cn,Cn-1,…..C1, 0).**

Note that the leading bit of bitwise sum $S^b$ is always 0 and the tail bit of the carry vector C is always a 0.

The input output relationships are expressed below:

$$S_i = x_i \oplus y_i \oplus z_i$$
$$C_{i+1} = x_i y_i \vee y_i z_i \vee z_i x_i$$

The arithmetic sum of 3 input numbers i.e S= X+Y+Z is obtained by adding two output numbers ie **S=$S^b$+C** Using a CPA.

**We use the CPA and CSA to implement the pipeline stages of a fixed- point unit as follows:**

**Multiply Pipeline Design** : Consider as an example the multiplication of two 38bit integers .AxB = 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, as shown in Fig. 6.23.

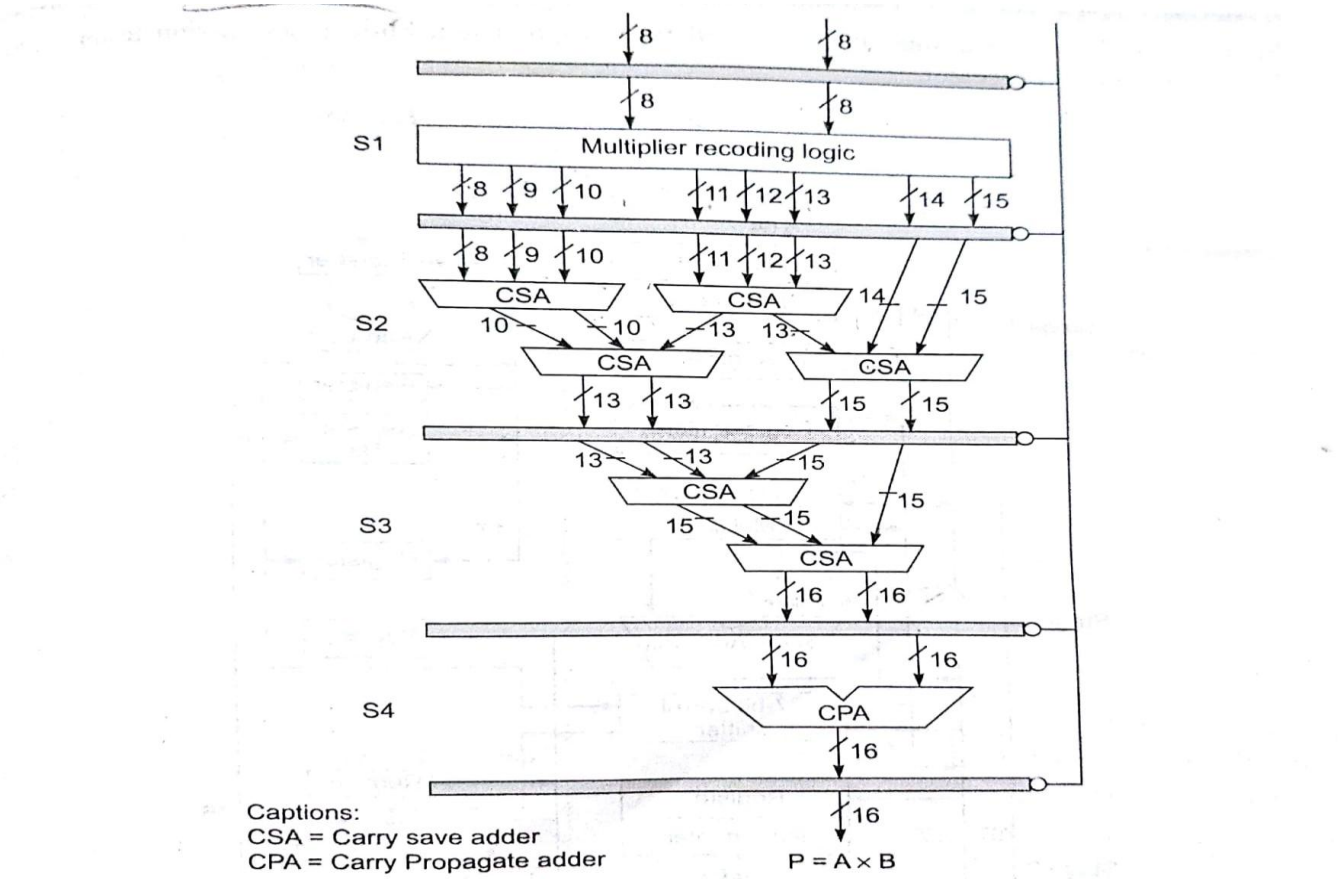


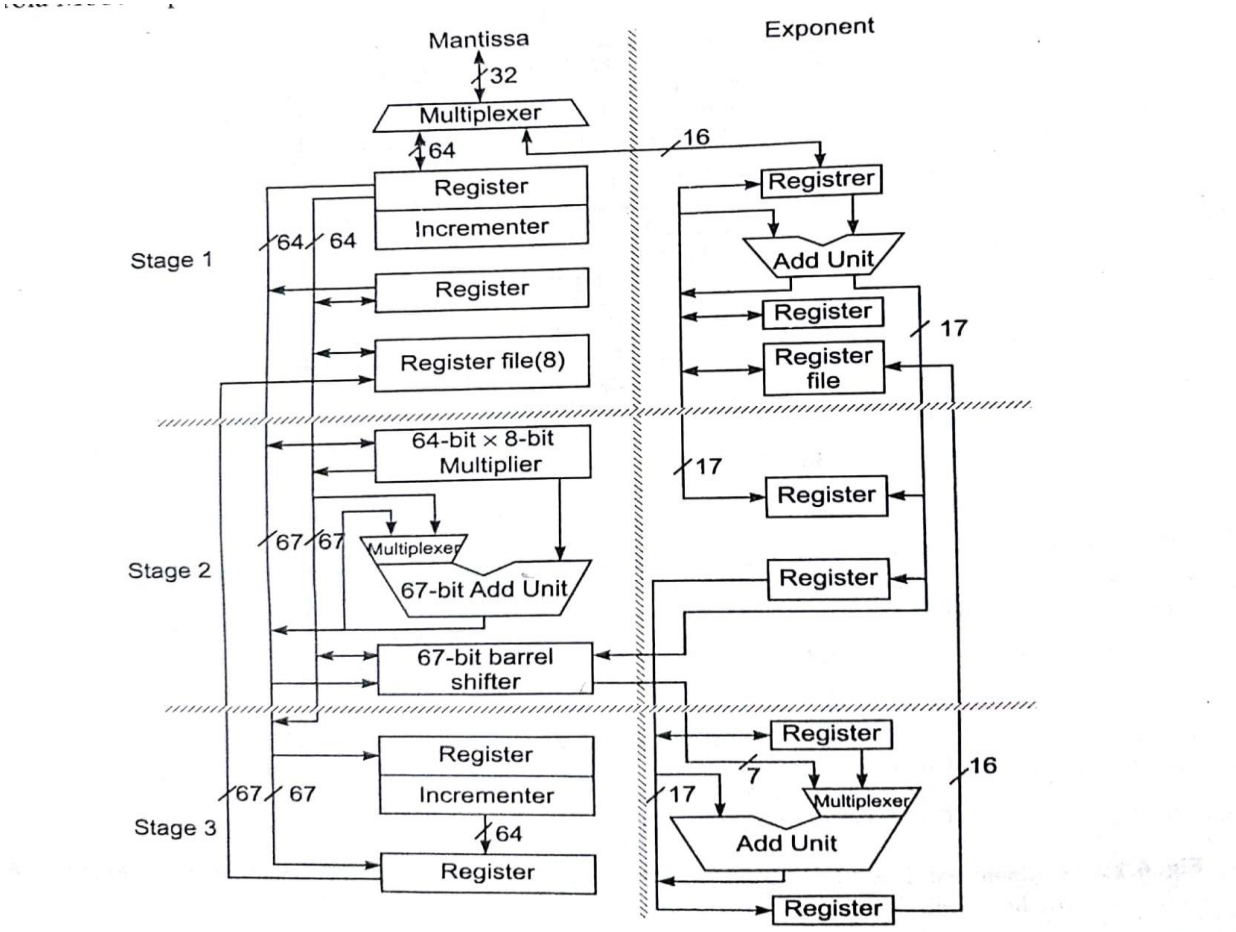

**Fig. 6.23** A pipeline unit for fixed-point multiplication of 8-bit integers (The number along each line indicates the line width.)

- 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 CSAs, and it essentially merges eight numbers into four numbers ranging from 12 to 15 bits.
- The third stage (S3) consists of two CSAs, and it merges four numbers from S1 into two l6-bit numbers.
- The final stage (S4) is a CPA, which adds up the last two numbers to produce the final product P.

## EXAMPLE: THE FLOATING – POINT UNIT IN THE MOTOROLA MC68040

Fig 6.4 below shows the design of a pipelined floating-point unit built as an on-chip feature in the Motorola MC68040 processor.




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

- This arithmetic pipeline has **three stages.**
- The **mantissa section and exponent section** are essentially **two separate pipelines.** 64-bit registers are used in this stage. Note that all three stages are connected to two 64-bit data buses.
    - In the mantissa section, **stage 1** receives input operands and returns with computation results;
    - **Stage 2** contains the array multiplier (64 X 8) which must be repeatedly used to carry out a long multiplication of the two mantissas. The 67-bit adder performs the Addition/subtraction of two mantissas, the **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.

- On the exponent side, a l6-bit bus is used between stages.
  - **Stage 1** has an exponent adder for comparing the relative magnitude of two exponents. The result of stage 1 is used to equalize the exponents before mantissa addition can be performed. Therefore, a shift count (from the output of the exponent adder) is sent to the barrel shifter for mantissa alignment.
  - After normalization of the final result (getting rid of leading zeros), the exponent needs to be 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, ready for subsequent usage.

**Convergence Division**

One technique for division involves repeated multiplications. Mantissa division is carried out by a convergence method . This convergence division obtains the quotient Q = M/D of two normalized fractions $0.5 \leq M < D < 1$ in two's complement notation by performing two sequences of chain multiplications as follows:

$$Q=\frac{M*R1*R2*....*Rk}{D*R1*R2*....*Rk}$$

Where the successive multipliers

$$R_i = 1 + \delta^{2^{i-1}} = 2 - D^{(i)} \quad \text{for } i = 1, 2, \ldots, k \quad \text{and} \quad D = 1 - \delta$$

## 5.2.3 Multifunctional Arithmetic Pipelines

- Static arithmetic pipelines are designed to perform a fixed function and are thus called ***unifunctional.***
- When a pipeline can perform more than one function, it is called **multifunctional.**
  - A 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 :T1/ASC ARITHMENTIC PROCESSOR DESIGN**

**(**a static multifunctional pipeline which was designed into the TI Advanced Scientific Computer (ASC).

- There were **four pipeline arithmetic units** built into the Tl-ASC system, as shown in Fig. 6.26 below.

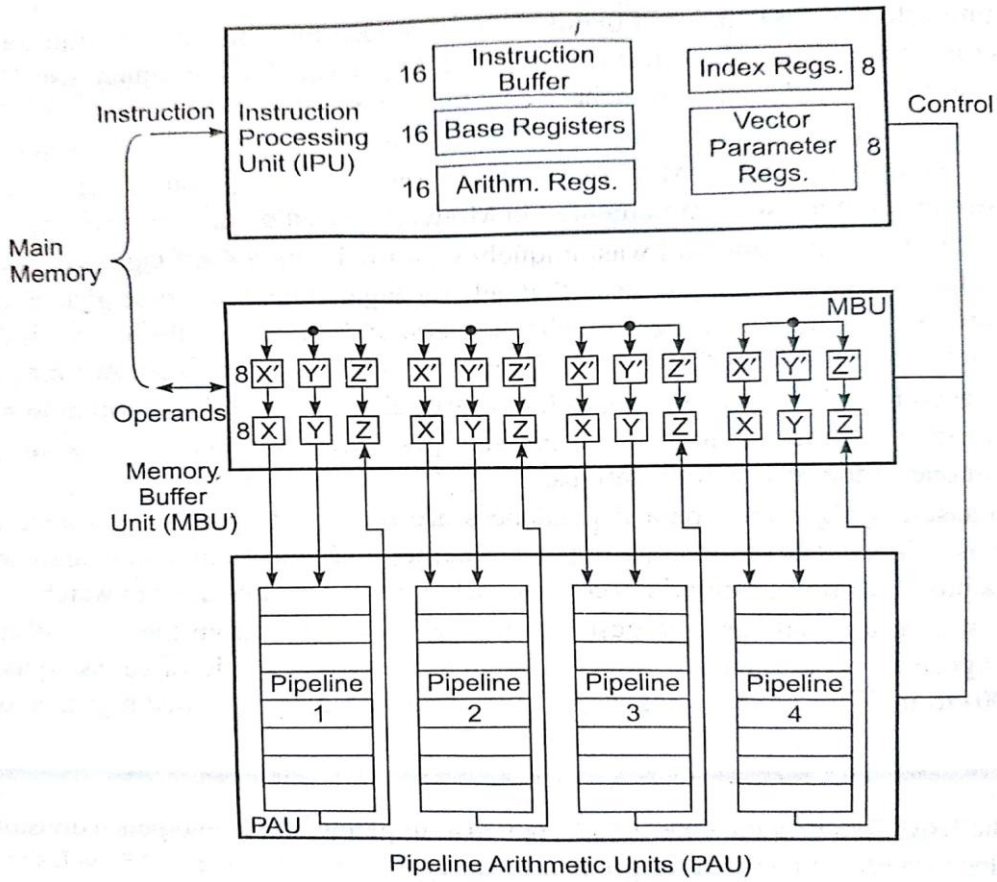



**Fig. 6.26** The architecture of the TI Advanced Scientific Computer (ASC) (Courtesy of Texas Instruments, Inc.)

- The **instruction processing unit** handled the fetching and decoding of instructions.
  - There were a large number of **working registers** in the processor which also controlled the operations of the **memory buffer unit** and of the **arithmetic units.**
- There were two sets of operand buffers, { X , Y , Z} and {X', Y', Z'}, in each arithmetic unit. .
  - X' , X, Y' and Y were used for **input operands**, and Z' and Z were used to **output results**.
  - Note that intermediate results could be also routed from Z-registers to either X or Y registers.
- Both **processor and memory buffers accessed the main memory** for instructions and operands/results, respectively.
- **Each pipeline arithmetic unit** had eight stages as shown in Fig. 6-.27a below. The PAU was a static multifunction pipeline which could perform only one function at a time. Figure 6.27a shows all the possible interstage connections for performing arithmetic, logical, shitting, and data conversion functions.
- 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. It should be noted that different functions required different pipeline stages and different interstage connection patterns.

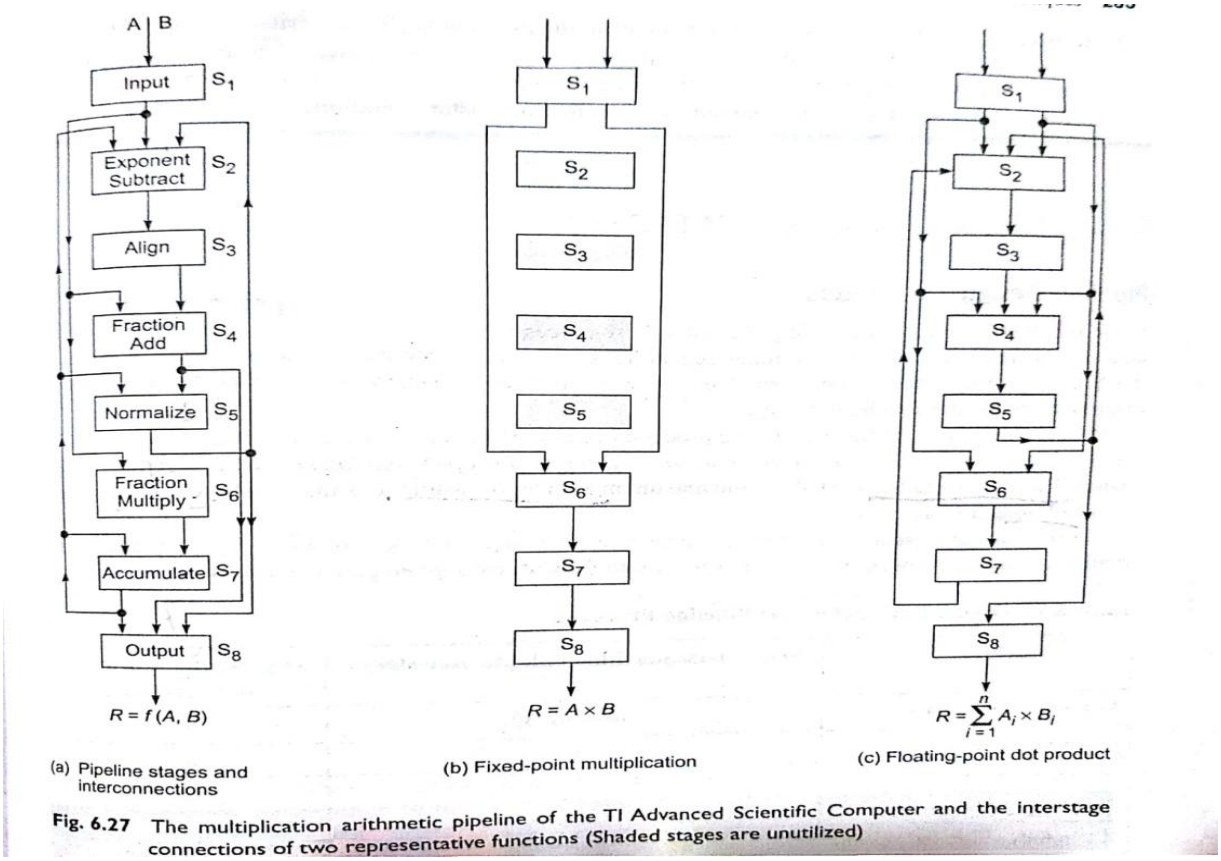



**Fig. 6.27** The multiplication arithmetic pipeline of the TI Advanced Scientific Computer and the interstage connections of two representative functions (Shaded stages are unutilized)

For example, fixed-point multiplication required the use of only segments S1, S6, S7 and S8 as -Shown in Fig. 6.27b. On the other hand, 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 shown in Fig. 6.27c. This dot product was implemented by essentially the following accumulated summation of a sequence of multiplications through the pipeline:

Z<-Ai*Bi+Z

where the successive operands (Ai, Bi) were fed through the X and Y -buffers, and the accumulated sums through the Z-buffer recursively.

Even though the Tl-ASC is no longer in production, the system provided a unique design for multifunction arithmetic pipelines. Today, most supercomputers implement arithmetic pipelines with dedicated functions for much simplified control circuitry and faster operations.

## 5.3 SUPERSCALAR PIPELINE DESIGN

**Qn: What are the pileline design parameters?**

**Pipeline Design Parameters**: Some parameters used in designing the **scalar base processor** and **superscalar processor** are summarized in Table 6.1 for the pipeline processors to be studied below. All pipelines discussed are assumed to have k stages.

The pipeline cycle for the scalar base processor is assumed to be l time unit, called the **base cycle**. We defined the instruction issue rate ,issue latency and simple operation latency. **The instruction level**

**parallelism is the maximum number of instructions that can be simultaneously executed in the pipeline.**

**For the base processor, all of these parameters have a value of 1**. All processor types are designed relative to the base processor. The ILP is needed to fully utilize a given pipeline processor.

**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.

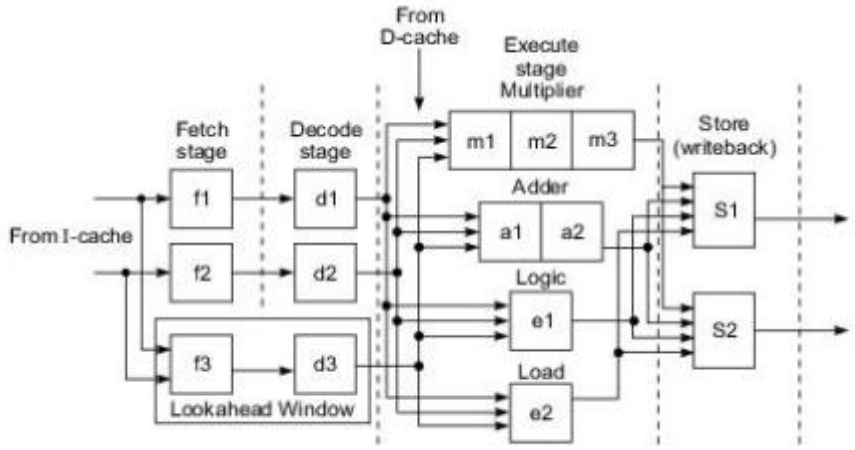
**Super scalar Pipeline Structure**

**Qn: Discuss about the two issue superscalar processor? Or**

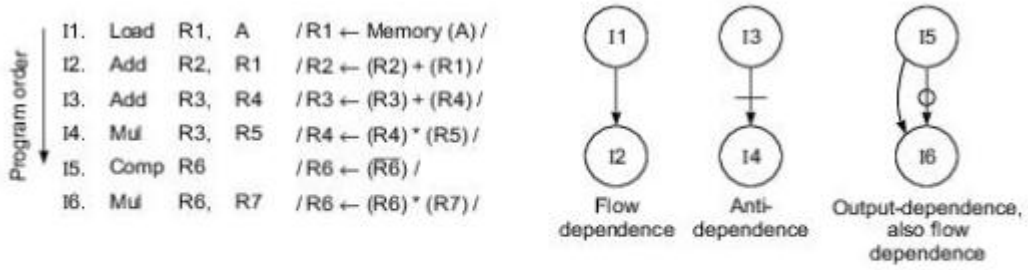
**Qn: Describe the superscalar pipeline structure/design.**

In an **m-issue** superscalar processor, the instruction decoding and execution resources are increased to form effectively **m** pipelines operating concurrently. At some pipeline stages, the functional units may be shared by multiple pipelines.

This resource-shared multiple pipeline structure is illustrated by a design example in Fig. 6.28a.In this design, the processor can issue two instructions per cycle if there is no resource conflict and no data dependence problem. **There are essentially two pipelines in the design**. **Both pipelines have four processing stages labeled fetch, decode, execute, and store, respectively.**




(a) A dual-pipeline, superscalar processor with four functional units in the execution stage and a lookahead window producing out-of-order issues




(b) A sample program and its dependence graph, where I2 and I3 share the adder and I4 and I6 share the multiplier

**Fig. 6.28** A two-issue superscalar processor and a sample program for parallel execution

Each pipeline essentially has its own fetch unit, decode unit and store unit. The two instruction streams flowing through the two pipelines are retrieved from a single source stream (the I-cache). The fan-out from a single instruction stream is subject to resource constraints and a data dependence relationship among the successive instructions.

For simplicity, **we assume that each pipeline stage requires one cycle, except the execute stage which may require a variable number of cycles. Four functional units, multiplier, adder, logic unit, and load unit, are available for use in the execute stage. These functional units are shared by the two pipelines on a dynamic basis. The multiplier itself has three pipeline stages, the adder has two stages, and the others each have only one stage.**

The **two store units (S1 and S2)** can be dynamically used by the two pipelines, depending on availability at a particular cycle. There is a lookahead window with its own fetch and decoding logic. **Lookahad window is used for instruction lookahead in case out-of-order instruction issue is desired to achieve better pipeline throughput.**

It requires complex logic to schedule multiple pipelines simultaneously, especially when the instructions are retrieved from the same source. The aim is to avoid pipeline stalling and minimize pipeline idle time.

**Data dependence:** In fig 6.28 B, 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 b 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. Since both I5 and I6 modify the register R6, and R6 supplies an operand for I6, we have both flow and output dependence: I5 → I6 and I5 ⊶ I6 as shown in the dependence graph.

To schedule instructions through one or more pipelines, these data dependences must not be violated (Eg fig 6.28). Otherwise, erroneous results may be produced.

**Pipeline Stalling**

**Qn:What is pipeline stall?**

**A pipeline stall is** a delay in execution of an instruction in order to resolve a hazard.

This is a problem which may seriously lower pipeline utilization. Proper scheduling avoids pipeline stalling. The problem exists in both scalar and superscalar processors. However, it is more serious in a superscalar pipeline. **Pipeline stalling can be caused by data dependences or by resource conflicts among instructions already in the pipeline or about to enter the pipeline**. We use an example to illustrate the conditions causing pipeline stalling.

Consider the scheduling of two instruction pipelines in a two-issue superscalar processor. Figure 6.29a shows the case of no data dependence on the left and flow dependence (ll -> I2) on the right. Without data dependence, all pipeline stages are utilised without idling.
With dependence, instruction I2 entering the second pipeline must wait for two cycles (shaded time slots) before entering the execution stages. This delay may also pass to the next instruction I4 entering the pipeline.

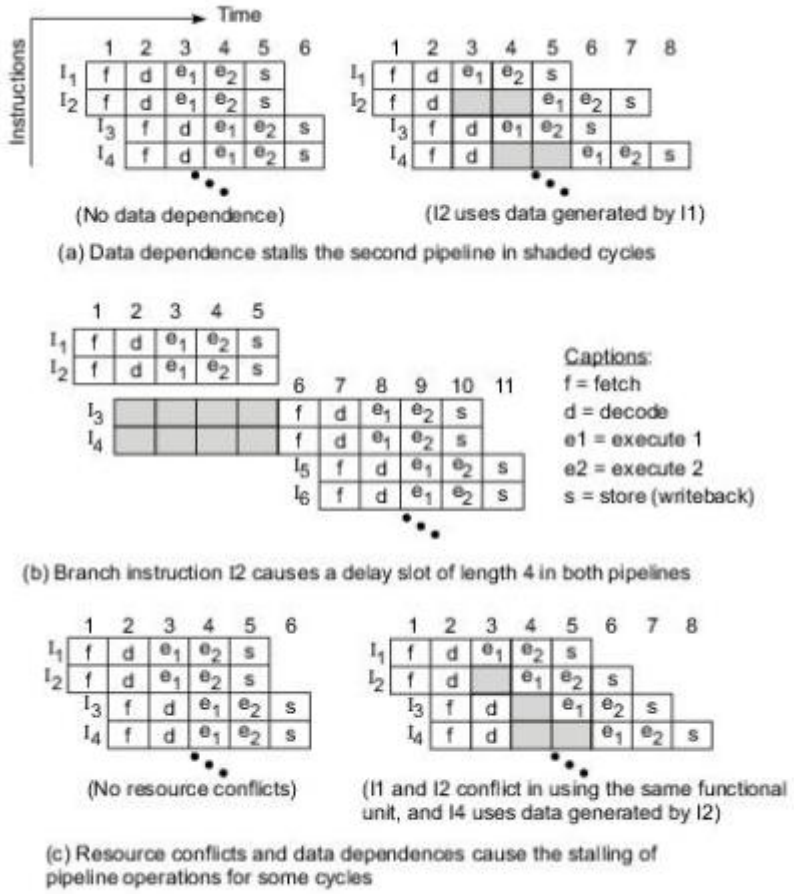



**Fig. 6.29** Dependences and resource conflicts may stall one or two pipelines in a two-issue superscalar processor

In Fig. 6.29b, we show the effect of branching (instruction I2). 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. Here, delayed branch or other amending actions are not taken.

In Fig. 6.29c , we show a combined problem involving both resource conflict and data dependence. Instructions Il and I2 need to use the same functional unit, and also I2 -> I4 exists.
The net effect is that I2 must be scheduled one cycle behind because the two pipeline stages (e1 and e2) of the same functional unit must be used by ll and I2 in an overlapped fashion. For the same reason, I3 is also delayed by one cycle. Instruction I4 is delayed by two cycles due to the flow dependence on I2. The shaded boxes in all the timing charts correspond to idle stages

**Superscalar pipeline Scheduling**

**Qn: Discuss about the in-order and out-of-order issue processor ? or**

**Qn: Describe the superscalar pipeline scheduling.**

Instruction issue and completion policies are critical to superscalar processor performance. **Three scheduling policies** are introduced below.

1. **In order issue & In order completion :** Instructions are issued in program order and instructions are completed in program order.
2. **In order issue & Out of order completion :** Instructions are issued in program order and instructions are completed not in program order**.**
3. **Out of order issue & Out of order completion** : Program order is violated in instruction issue and completion

In-order issue may result in either in-order or out-of-order completion. . ln-order issue is easier to implement but may not yield the optimal performance. The purpose of out of order issue and completion is to improve performance.

These three scheduling policies are illustrated in Fig 6.30 by the execution of the example program in Fig 6.28b on the dual-pipeline hardware in Fig 6.28a. It is demonstrated that performance can be improved from an in-order to out-of-order schedule. Not all programs can be scheduled out of order.

**In-Order Issue**

Figure 6.30a shows a schedule for the six instructions being issued in program order Il, I2, I6. Pipeline 1 receives I1, I3, and I5, and pipeline 2 receives I2, I4, and I6 in three consecutive cycles. Due to I1 —> I2, I2 has to wait one cycle to use the data loaded in by I1.

I3 is delayed one cycle for the same adder used by I2. I6 has to wait for the result of I5 before it can enter the multiplier stages. In order to maintain in-order completion , I5 is forced to wait for two cycles to come out of pipeline 1. In total, nine cycles are needed and five idle cycles (shaded boxes) are observed.

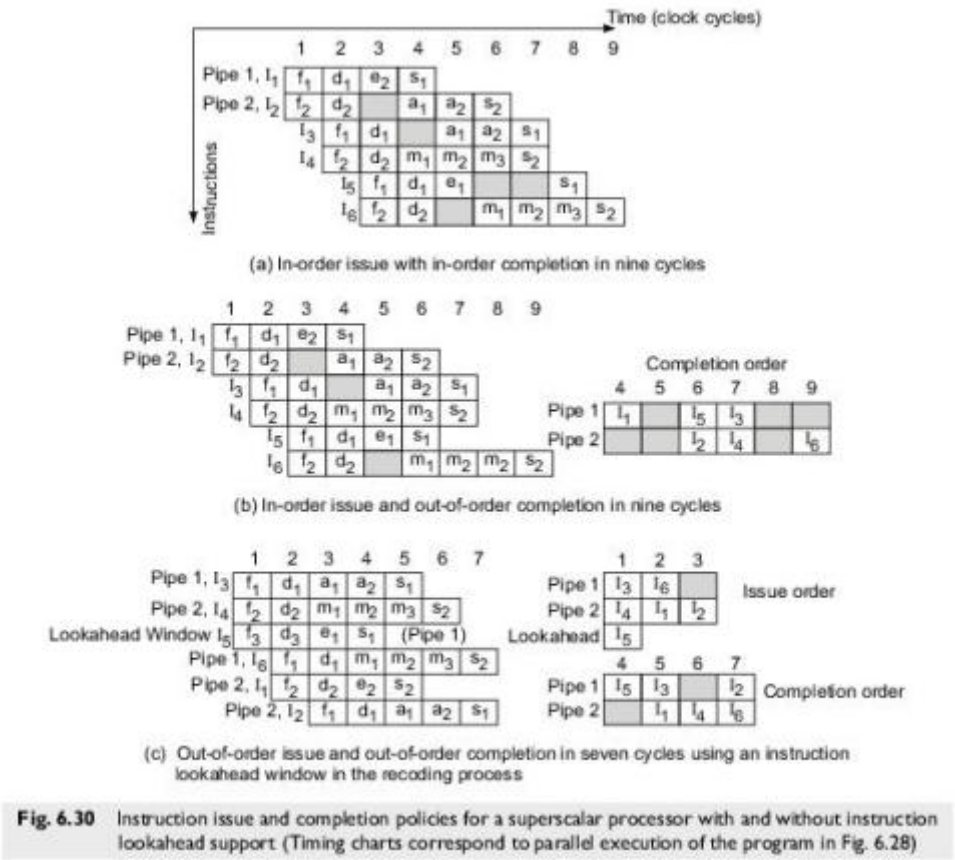



**Fig. 6.30** Instruction issue and completion policies for a superscalar processor with and without instruction lookahead support (Timing charts correspond to parallel execution of the program in Fig. 6.28)

In Fig. 6.30b , out-of-order completion is allowed even if in-order issue is practiced. The only difference between this out-of-order schedule and the in-order schedule is that I5 is allowed to complete ahead of I3 and I4, which are totally independent of I5. The total execution time does not improve. However, the pipeline utilization rate does.

Only three idle cycles are observed. Note that in Figs. 6.29a and 6.29b, we did not use the lookahead window. In order to shorten the total execution time, the window can be used to reorder the instruction issues.

**Out-of-Order Issue**

By using the lookahead window, instruction I5 can be decoded in advance because it is independent of all the other instructions. The six instructions are issued in three cycles as shown: I5 is fetched and decoded by the window, while I3 and I4 are decoded concurrently.

It is followed by issuing I6 and I1 at cycle 2, and I2 at cycle 3. Because the issue is out of order, the completion is also out of order as shown in Fig. 6.30c. Now, the total execution time has been reduced to seven cycles with no idle stages during the execution of these six instructions.

The in-order issue and completion is the simplest one to implement. It is rarely used today even in a conventional scalar processor due to some unnecessary delays in maintaining program order. However, in a multiprocessor environment, this policy is still attractive. Allowing out-of-order completion can be found in both scalar and superscalar processors.

Output dependence and antidependence are the two relations preventing out-of-order completion. Out of order issue gives the processor more freedom to exploit parallelism, and thus pipeline efficiency is enhanced. It should be noted that multiple-pipeline scheduling is an NP-complete problem. Optimal scheduling is very expensive to obtain.
Simple data dependence checking, a small lookahead window, and score-boarding mechanisms are needed along with an optimizing compiler, to exploit instruction parallelism in a superscalar processor.

**Motorola 88710Architecture**

The Motorola 88110 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**, with additional improvements. The 83110 employed advanced techniques for exploiting instruction-level parallelism, including instruction issue, out-of order instruction completion, speculative execution, dynamic instruction rescheduling, and two on-chip caches. The unit also supported demanding graphics and digital signal processing applications.
The 88110 employed a symmetrical superscalar instruction dispatch unit which dispatched two instructions each clock cycle into an array of 10 concurrent units. **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 (8 instructions) per block. The data cache resembled that of the instruction set.

The **88110 employed the MESI cache coherence protocol**. A write-invalidate procedure guaranteed that one processor on the bus had a modified copy of any cache block at any time. **The 83110 was implemented with 1.3 million transistors in a 299-pin package and driven by a 50 MHz clock.**

## Superscalar Performance

**Qn: What is the performance of superscalar processor**

To compare the relative performance of a superscalar processor with that of a scalar base machine, we estimate the ideal execution time of **N** independent instructions through the pipeline.

**The time required by the scalar base machine** is

**T(1,1) = k+N-1 (base cycles)**

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

**T(m,1) = k+ N-m (base cycles)**

**m**

where ***k*** is the time required to execute the first m instructions through the in pipelines simultaneously, and the second term corresponds to the time required to execute the remaining ***N-m*** instructions, ***m*** per cycle, through ***m*** pipelines.

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.

# DEC Alpha 21064 superscalar architecture

As illustrated in Fig. 6.3 l, this was a 64-bit superscalar processor. The design emphasized speed, multiple - instruction issue, multiprocessor applications, software migration from the VAX./VMS and MIPS/OS, and a long list of usable features. The clock rate was 150 MHz, with the first chip implementation.

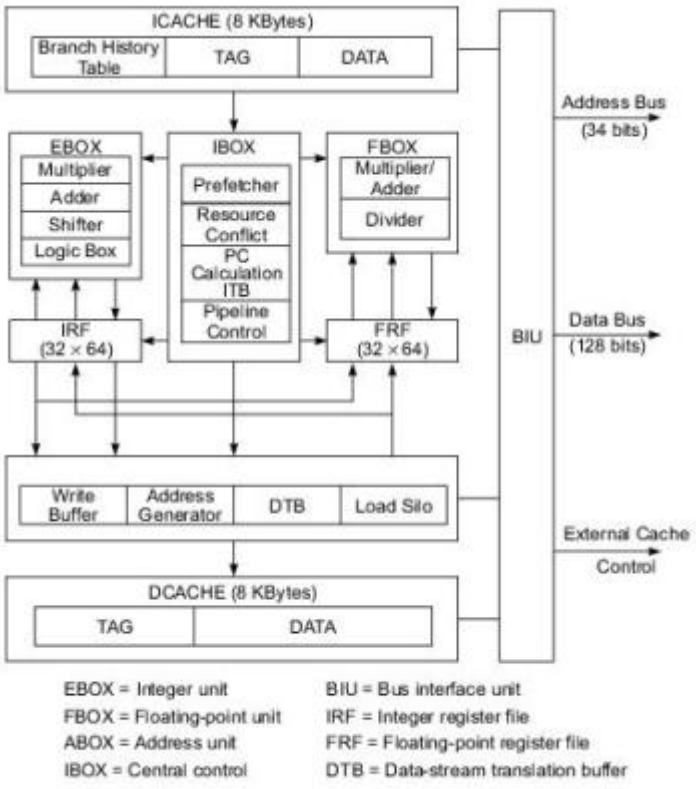



**Fig. 6.31** Architecture of the DEC Alpha 21064 processor (Courtesy of Digital Equipment Corporation)

Unlike others, the Alpha architecture had **thirty-two 64-bit integer registers and thirty-two 64-bit floating point registers.** The **integer pipeline had 7 stages, and the floating-point pipeline had 10**

**stages**. All **Alpha instructions were 32 bits**. The first Alpha implementation issued two instructions per cycle, with larger number of issues in later implementations. Pipeline timing hazards, load delay slots, and branch delay slots were all minimized by hardware support. The Alpha was designed to support fast multiprocessor interlocking and interrupts. The processor was designed to have a **300-MIPS peak and 150 - Mflops peak at 150 MHz.**