CS405
Computer System Architecture

MODULE 5

Syllabus

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

INSTRUCTION PIPELINE DESIGN

- A stream of instructions can be executed by a pipeline in an overlapped manner.
- Topics to be studied include
  - instruction pre-fetching,
  - data forwarding,
  - hazard avoidance,
  - interlocking for resolving data dependences,
  - dynamic instruction scheduling, and
  - branch handling techniques for improving pipelined processor perfomance

Instruction Execution Phases

- A typical instruction execution consists of a sequence of operations, including
  - instruction fetch,
  - decode,
  - operand fetch,
  - execute, and
  - write-back phases.
- These phases are ideal for overlapped execution on a linear pipeline.

Pipelined Instruction Processing

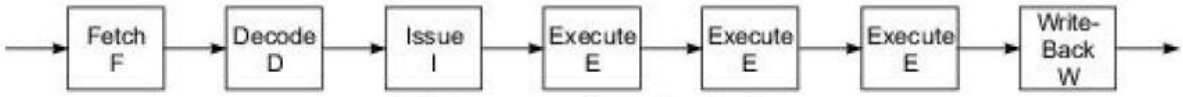
- fetch stage (F)
  - fetches instructions from a cache memory, ideally one per cycle.
- decode stage (D)
  - reveals the instruction function to be performed and identifies the resources needed.
  - Resources include general-purpose registers, buses, and functional units.
- issue stage (I)
  - reserves resources. The operands are also read from registers during the issue stage.

- execute stages (E)
  - The instructions are executed in one or several execute stages (E).
- writeback stage (W)
  - is used to write results into the registers.
  - Memory load or store operations are treated as part of execution

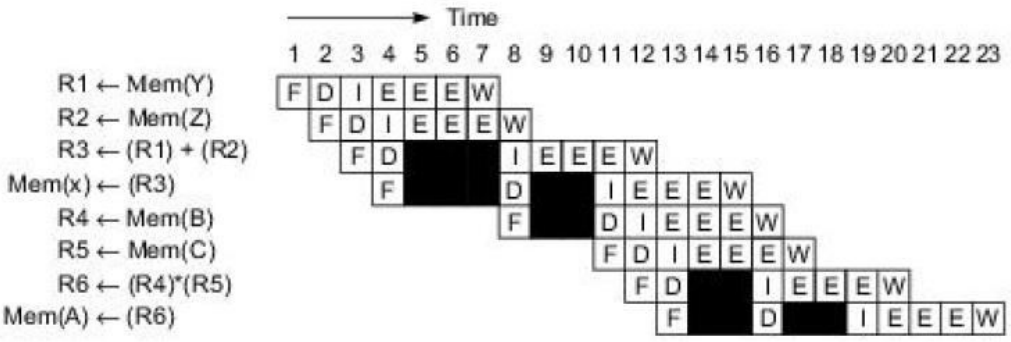
(a) A seven-stage instruction pipeline

- With in-order instruction issuing,
  - if an instruction is blocked from issuing due to a data or resource dependence, all instructions following it are blocked.

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



(a) A seven-stage instruction pipeline



(b) In-order instruction issuing

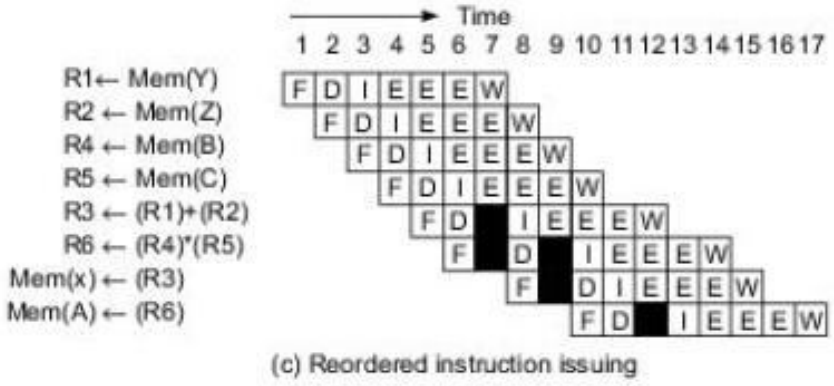
- Aim
  - X=Y+Z
  - A=B*C
- Shaded boxes corresponds to idle cycles
  - This situation happens when instruction issues are blocked due to
  - Resource latency or
  - Due to conflicts of data dependency
- First two load instructions issue on consecutive cycles
- Add is dependent on both loads
  - Hence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

Total Time Required

- It is measured beginning at cycle4, 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=17cycles

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

1. Pre-fetch buffers
2. Multiple functional units
3. Internal data forwarding
4. Hazard avoidance

Pre-fetch buffers

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

1. Sequential buffer
2. Target buffer
3. Loop buffer

- Sequential buffer
  - Sequential instructions are loaded for in-sequence pipelining
- Target buffer
  - Instructions from a branch target are for out-of-sequence pipelining
- Both buffers operate in a first-in-first-out fashion
- Buffers alternate to avoid collision
- Buffers become a part of pipeline as additional stages

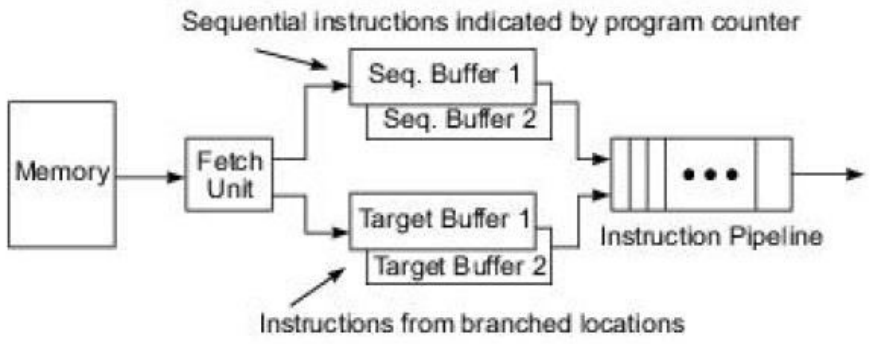


Fig. 6.11 The use of sequential and target buffers

- 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.

❑Loop buffers

- maintained by the fetch stage of the pipeline.
- This buffer holds the sequential instructions contained in a small loop

- Pre-fetched instructions in the loop body will be executed repeatedly until all iterations complete execution.

- The loop buffer operates in two steps.
  - First, it contains instructions sequentially ahead of the current instruction.
    - This saves the instruction fetch time from memory.
  - Second, it recognises when the target of a branch falls within the loop boundary.
    - unnecessary memory accesses can be avoided if the target instruction is already in the loop buffer.

Multiple Functional Unit:

- 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 bottle neck 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 Unit

- 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

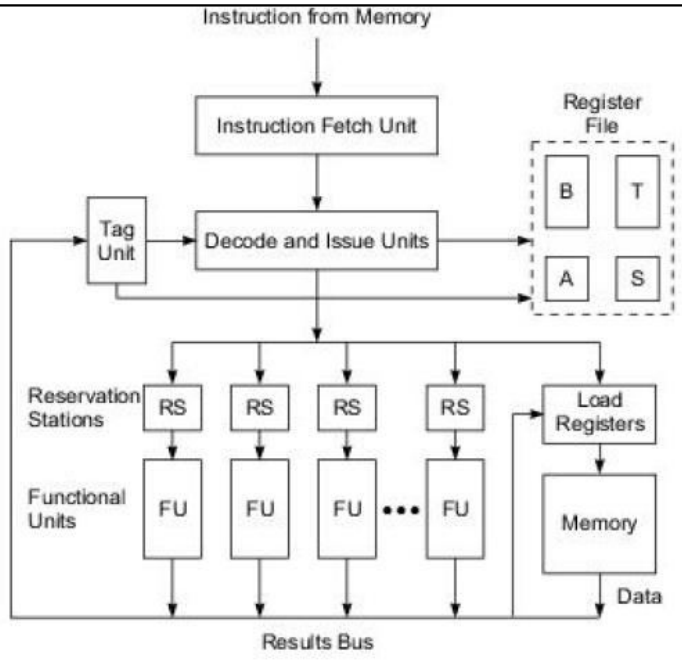
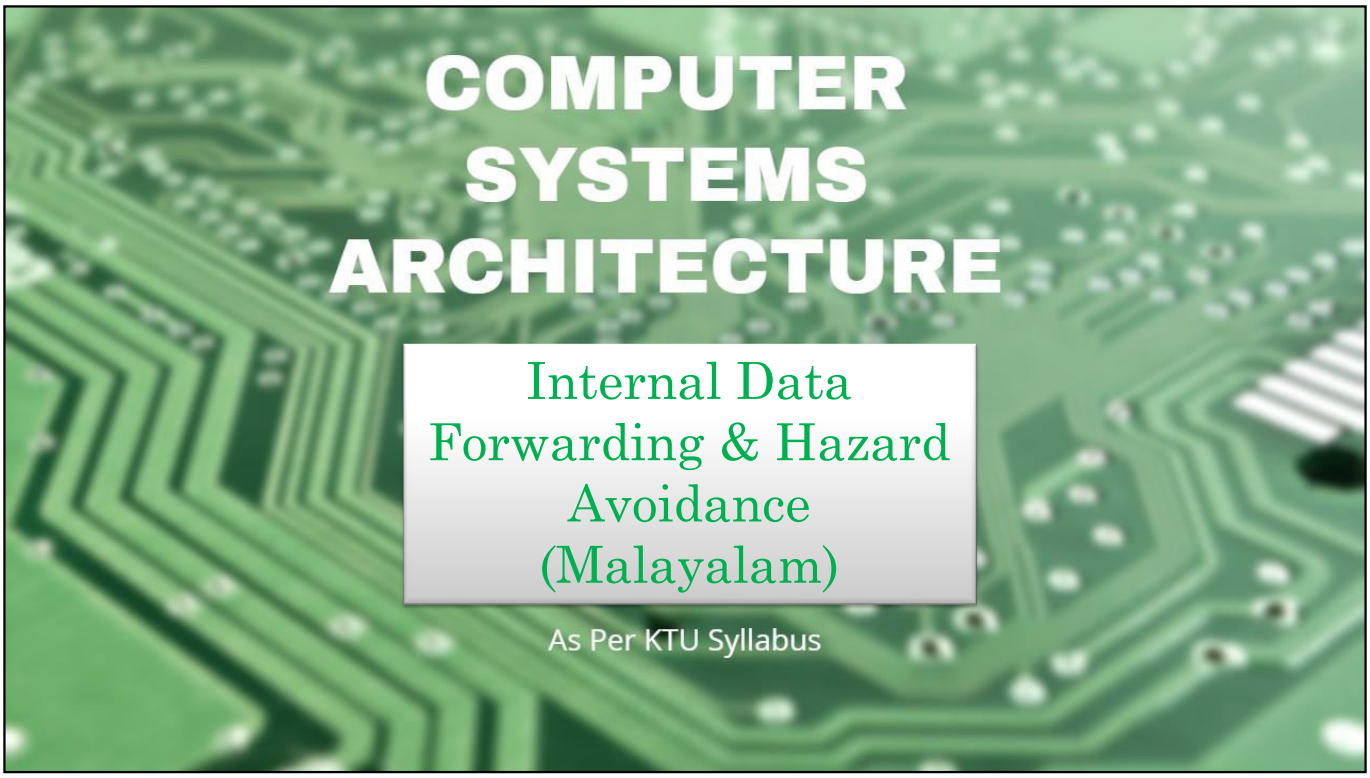


Fig. 6.12 A pipelined processor with multiple functional units and distributed reservation stations supported by tagging (Courtesy of G. Sohi; reprinted with permission from *IEEE Transactions on Computers*, March 1990)

- 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 bottle neck in the execution stages of instruction pipeline



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

3. Internal data forwarding
4. Hazard avoidance

Internal Data Forwarding

- Throughput of pipelined processor is improved using internal data forwarding among multiple FU
- A store-load forwarding, which consist of a load operation (LDR2,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

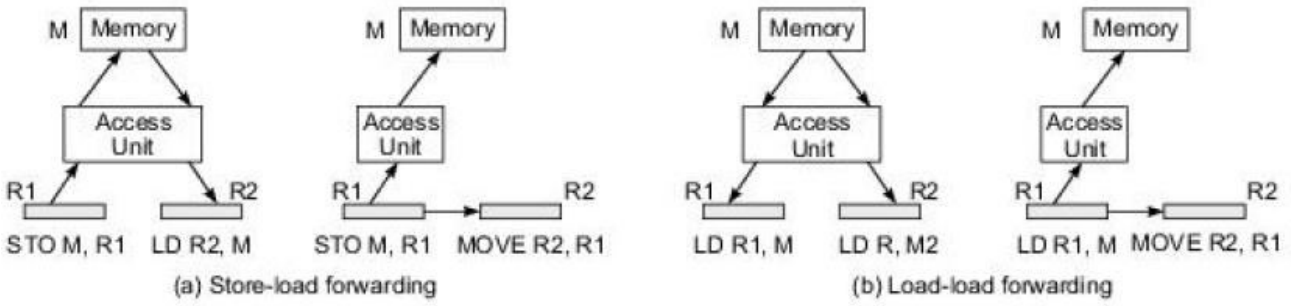
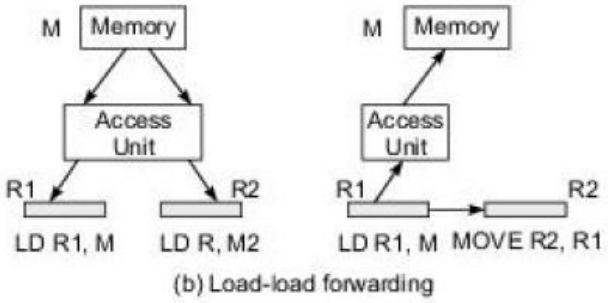


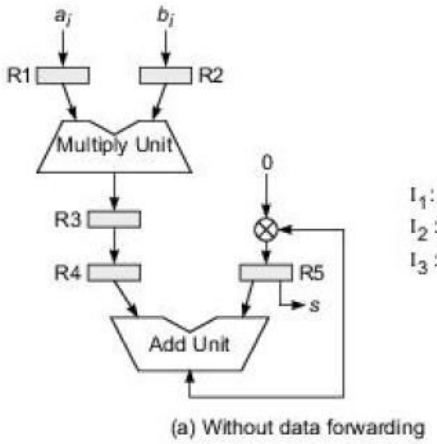
Fig. 6.13 Internal data forwarding by replacing memory-access operations with register transfer operations

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



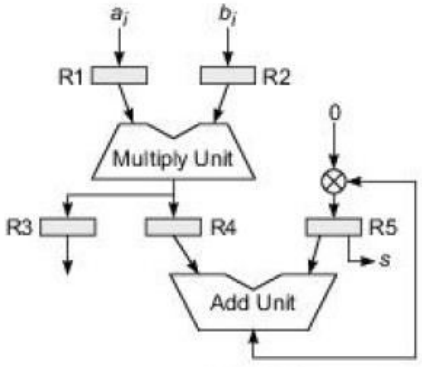
Hazard Avoidance

- 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



(a) Without data forwarding

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



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

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

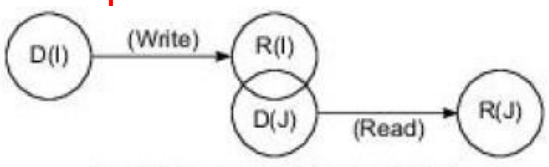
(b) With internal data forwarding

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(RAW)

- 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 , so J incorrectly gets the old value
- RAW corresponds to flow dependence
- Solve via forwarding



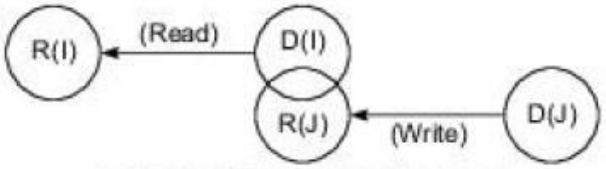
(a) Read-after-Write (RAW) hazard

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

Write after read(WAR)

- This hazard occur when instruction J tries to write to a destination, before it is read by I
- WAR corresponds to antidependence
  - j tries to write a destination before it is read by i, so i incorrectly gets the new value
  - For this to happen we need a pipeline that writes results early in the pipeline, and then other instruction read a source later in the pipeline

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

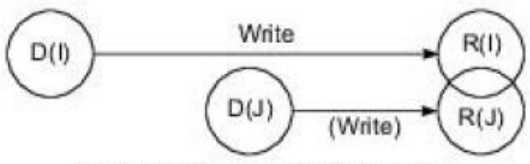


(c) Write-after-Read (WAR) hazard

Write after write(WAW)

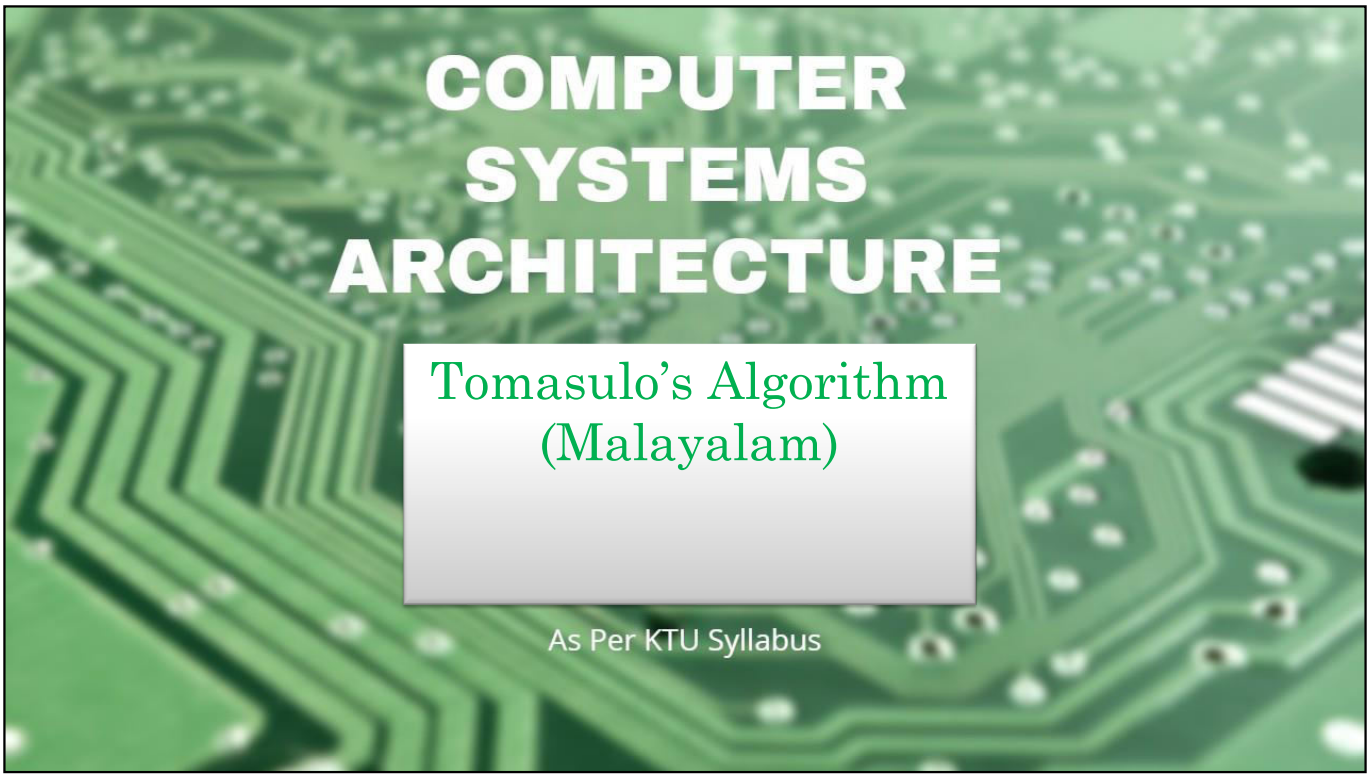
- WAW corresponds to output dependence
  - j tries to write an operand before it is written by i, so we end up writing values in the wrong order
  - Only occurs if there are writes in multiple stages
    - Not a problem with single cycle integer instructions

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



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

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



Instruction Scheduling

- Three methods for scheduling instructions through an instruction pipeline

1. Static scheduling scheme
   - is supported by an optimizing compiler.
2. Dynamic scheduling
   - is achieved using a technique such as **Tomasulo's register-tagging** scheme built in the IBM 360/91 , or
3. Score boarding scheme
   - built in the CDC 6600 processor.

Static Scheduling

- Data dependences in a sequence of instructions create interlocked relationships among them.
- Interlocking can be resolved through a compiler-based static scheduling approach.
- A compiler or a postprocessor can be used to increase the separation between interlocked instructions.

- 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

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

- The two loads since they are independent of the odd and move can be moved ahead to increase the spacing between them and the multiply instruction

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

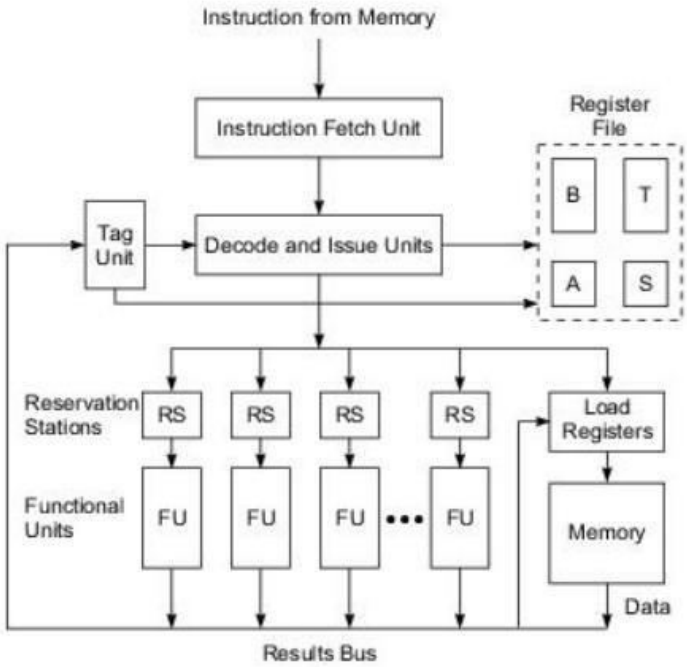
- Through this code rearrangement, the data dependences and program semantics are preserved, and the multiply can be initiated without delay.
- While the operands are being loaded from memory cells a and b into registers R2 and R3, the two instructions add and move consume three cycles and thus pipeline stalling is avoided.

Tomasulo's Algorithm

- It is a hardware scheme for dependence resolution
- It was first implemented in floating point units of IBM 360/91 processor
- In model 91processor,
  - 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

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.

- Aim
  - X=Y+Z
  - A=B*C

total execution time= 13 cycles

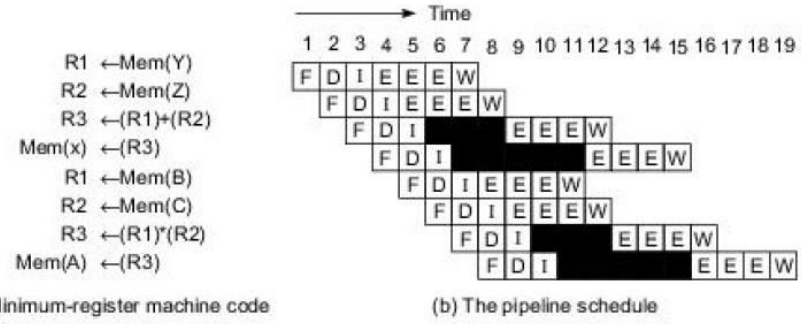


Fig. 6.16 Dynamic instruction scheduling using Tomasulo's algorithm on the processor in Fig. 6.12 (Courtesy of James Smith; reprinted with permission from *IEEE Computer*, July 1989)

COMPUTER SYSTEMS ARCHITECTURE

CDC Score boarding (Malayalam)

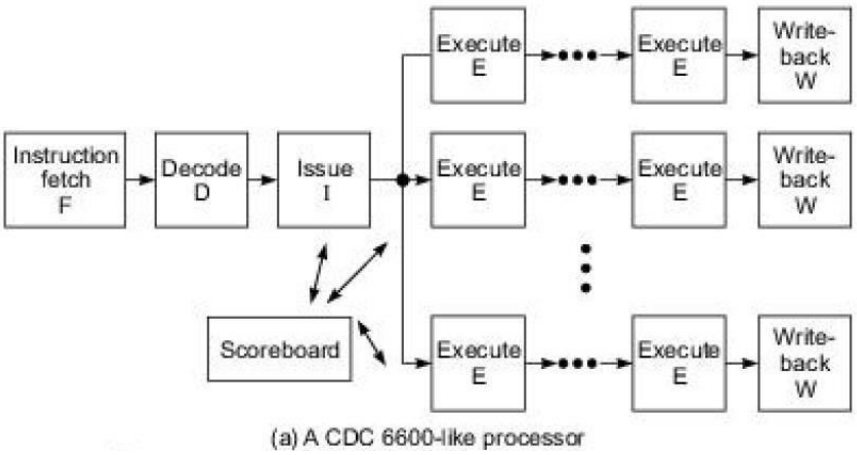
As Per KTU Syllabus

CDC Score boarding

- CDC 6600 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

- 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.**

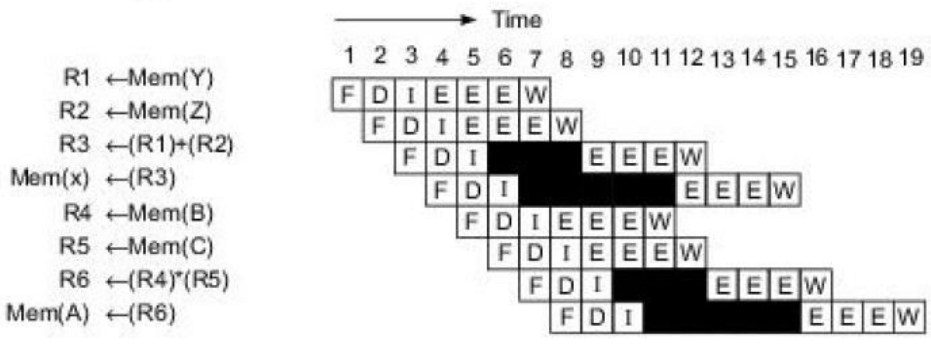
- The scoreboard is a centralized control logic which keeps track of the status of registers and multiple functional units.
- It kept track of the registers needed by instructions waiting for the various functional units.
- When all registers had valid data, the scoreboard enabled the instruction execution.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



(a) A CDC 6600-like processor

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 mean time, the issue stage is not blocked, so other instructions can by pass the blocked add.

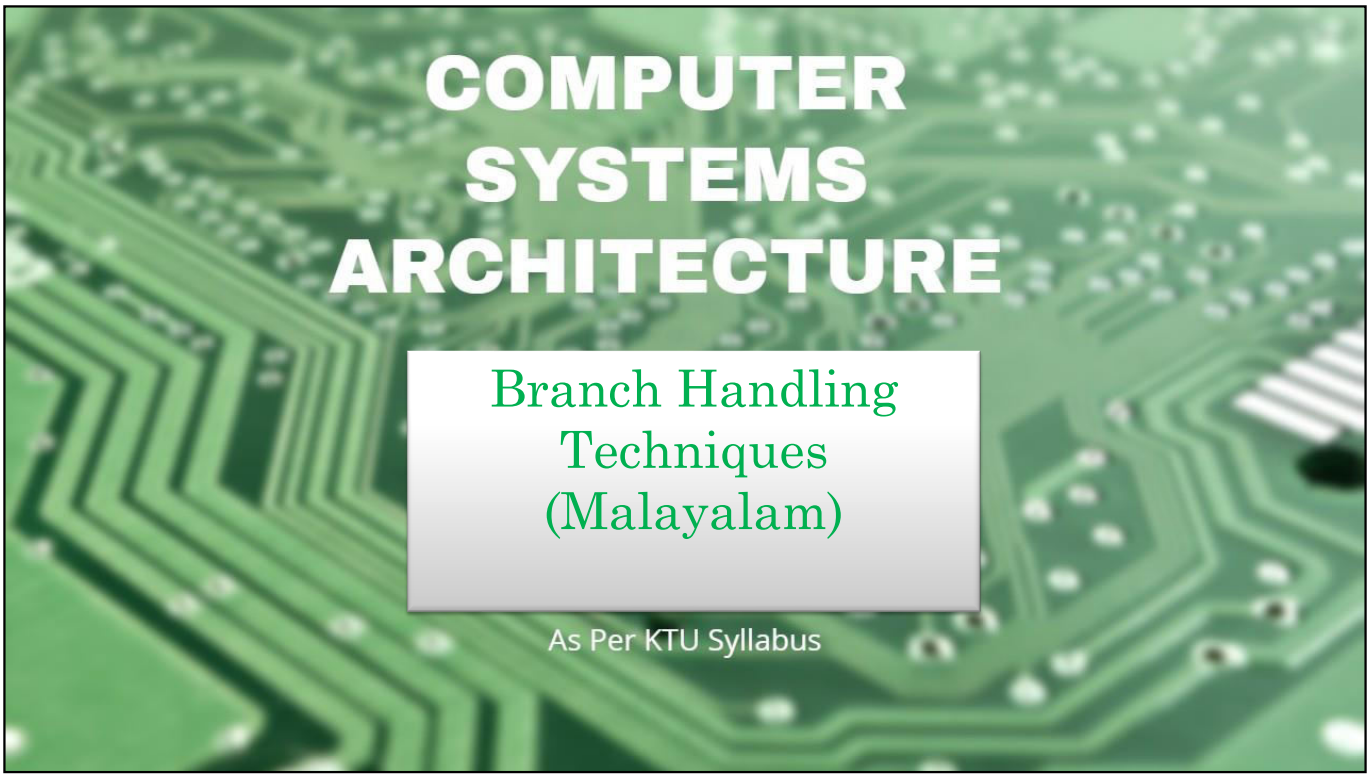


(b) The improved schedule from Fig. 6.9b

Fig. 6.17 Hardware scoreboarding for dynamic instruction scheduling (Courtesy of James Smith; reprinted with permission from *IEEE Computer*, July 1989)

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, RISC and superscalar processors are built with hardware support of dynamic scheduling at runtime.

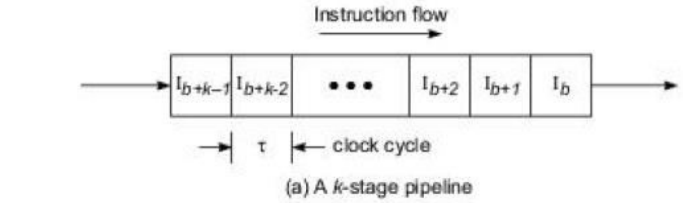


Branch Handling Techniques

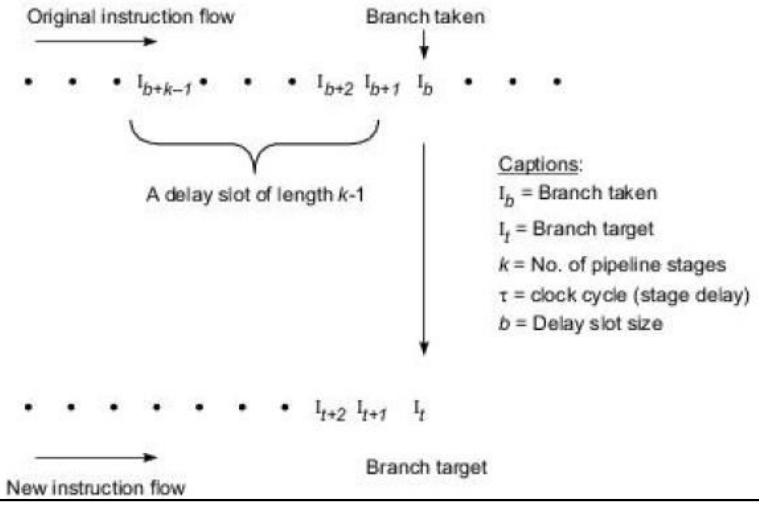
- The performance of pipelined processors is limited by data dependences and branch instructions
- Effects of branching
- 3 terms used to analyze the effects of branching are:-
- Branch taken
- Branch target
- Delay slot

- 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 I_{b+1} through I_{b+k-1} to be drained from the pipeline.



(a) A *k*-stage pipeline



Total Execution Time

- Total execution time of n instruction including the effect of branching $T_{eff} = k\tau + (n-1)\tau + pqnb\tau$
- $pqnb\tau$ → 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:- $H_{eff} = \frac{n}{T_{eff}} = \frac{nf}{k+n-1+pqnb}$
- When $n \to \infty$, the tightest upper bound on the effective pipeline throughput is obtained when b = k -1:

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

- When p = q = 0 (no branching), then we have maximum throughput

- Suppose p = 0.2, q = 0.6, and b = k-1 = 7
- Performance Degradation Factor(D):

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

- The above analysis implies that the pipeline performance can be degraded by 46% with branching when the instruction stream is sufficiently long.
- This analysis demonstrates the high degree of performance degradation caused by branching in an instruction pipeline

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
- According to past experience the best performance is given by branch taken
- This results from the fact that most conditional branch instructions are taken in program execution.
- The wired-in static prediction cannot be changed once committed to the hardware
- Static branch prediction is not very accurate

Dynamic branch prediction

- In dynamic branch prediction, recent branch history to predict whether or not the branch will be taken next time when it occurs
- 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

Cragon's Classes of Dynamic branch Stratergies

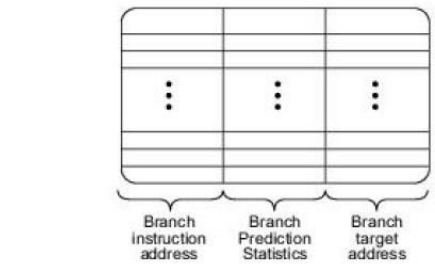
- Class 1
  - Predicts the branch direction based on information found at the decode stage
- Class2
  - Predicts the branch direction at the stage when effective address of the branch target is computed
  - It uses a cache to store the target addresses
- Class3
  - Uses a cache to store the target instructions at fetch stage

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

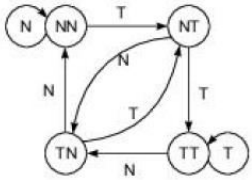
Branch Target Buffer

- 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



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

(b) A typical state diagram

- A state transition diagram has been used by Lee and Smith for tracking the last two outcomes at each branch instruction in a given program.
- The BTB entry contains the information which will guide the prediction.
- Prediction information is updated upon completion of the current branch.
- The BTB can be extended to store not only the branch target address but also the target instruction itself, in order to allow zero delay in converting conditional branches to unconditional branches.
- The taken (T) and not-taken (N) labels in the state diagram correspond to actual program behavior

ARITHMETIC PIPELINE DESIGN

- Pipelining techniques can be applied to speed up numerical arithmetic computations.

Computer arithmetic Principles

- 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 Operation

- 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 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 64bit floating numbers known as the IEEE 754 standard