CS257 Advanced Computer Architecture

Topic 9: Processor Performance and Pipelining

Graham Martin and Matthew Leeke

{grm, matt}@dcs.warwick.ac.uk

Department of Computer Science

University of Warwick

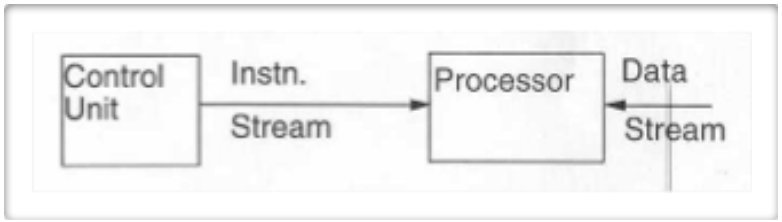
WARWICK

Processor Architecture

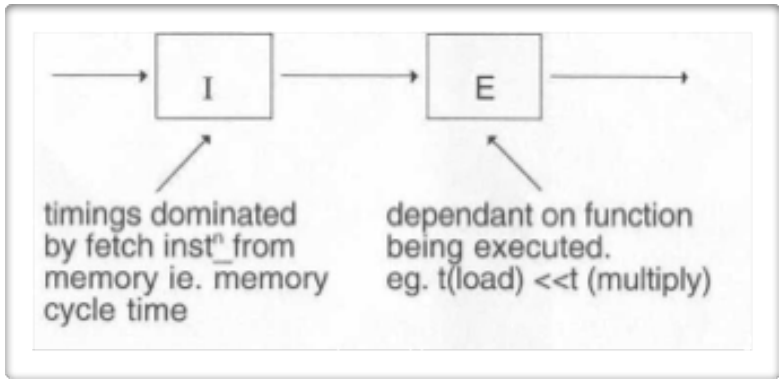
- Processor Organisation
- CPU Control and Sequencing
- **Processor Performance and Pipelining**
- Superscalar Processors
- Exploiting Instruction Level Parallelism

Architecture and Performance

- **Von Neumann Machine**
  - M J Flynn (1966) defined a simple means of classifying machines
  - SISD (Single Instruction, Single Data stream)



  - To introduce parallelism, consider the CPU instruction cycle as comprising two sub-cycles
    - I cycle: Fetch & decode
    - E cycle: Execution (function performed)



Processor Performance

- A simple measure of performance is the number of instructions executed per second

  Normally expressed as MIPS (Millions of Instructions Per Second)

- The performance of parallel systems is very difficult to measure, as it depends on:
  - (i) System architecture, especially:
    - Degree of parallelism
    - Interconnection structure
    - Main memory architecture
  - (ii) Degree of parallelism in the programs to be executed
- The simple performance measures of ***speedup*** and ***processor utilisation*** are good indicators

Processor Performance

- The performance of a single processor (or processing element, PE) can be measured by the Instruction Execution Rate, or 'Instruction Bandwidth', b_I, in units of MIPS
  - Corresponding measure of ***data bandwidth***, b_D, typically measured in MFLOPS (Millions of Floating Point Operations per Second)
- Consider n processors, each with a performance p

  The maximum possible performance = np

  This maximum performance is never achieved because of a number of factors, including:
  - Non-highly parallel algorithms
  - Inter-processor communication delays

Processor Performance

- Speedup is a useful measure that takes into account the degree of parallelism (for some class of tasks)

$$Speedup, S(n) = \frac{\text{Execution time on 1 (sequential) machine, } T(1)}{\text{Execution time on a parallel machine, } T(n)}$$

- We can assume: $T(1) \leq nT(n)$, therefore $S(n) \leq n$
- A closely related performance measure is ***efficiency***, E_n, or ***processor utilisation***

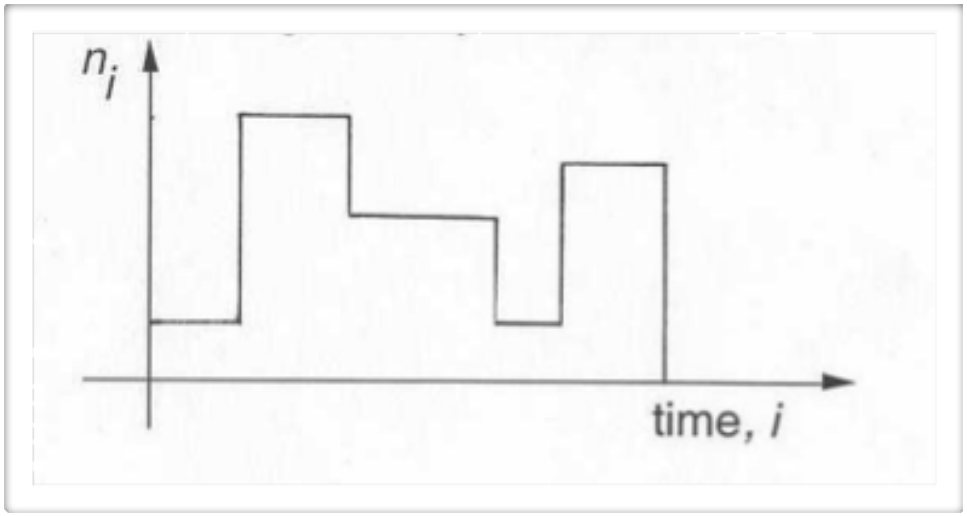
$$E_n = \frac{S(n)}{n}$$

- Hence the speedup and efficiency measures give an indication of performance, but they must be used with caution

  They depend on the parallelism of the algorithm

Processor Performance

- A program or algorithm, Q, may be characterised by its degree of parallelism, n_i, which is the value of n for which $S(n)$ and E_n reach minimum values at time i
  - Therefore n_i = maximum level of parallelism that can be exploited by Q at time i
- As an example, consider a program Q which shows some degree of parallelism:



Processor Performance

- Let us now consider the influence of program parallelism on parallel computer performance
- Example:
  - Assume all computations are floating point, and of two types:
    - Vector operations of some fixed length, N
    - Scalar operations where all operands are scalar N = 1
  - Let f be the total proportion of scalar ops, hence the total proportion of vector ops is (1-*f*)

    (1-*f*) is a measure of the parallelism in the program
  - Let b_v be the throughput of vector ops in (MFLOPS) and b_s be the throughput of scalar ops in (MFLOPS), then the average throughput is:

$$\frac{1}{b} = \frac{f}{b_s} + \frac{(1-f)}{b_v} \qquad (1)$$

Processor Performance

- Execution time for 1 N-component vector operation is:

$$T_v = \frac{N}{b_v}$$

- Execution time for 1 scalar operation is:

$$T_s = \frac{1}{b_s}$$

- These are related by:

$$T_v = T_0 + \frac{NT_s}{n}$$

Where T_0 is some fixed set up time, independent of vector length, and n is the processor parallelism degree

Processor Performance

- When N is large, we can ignore T_0, then:

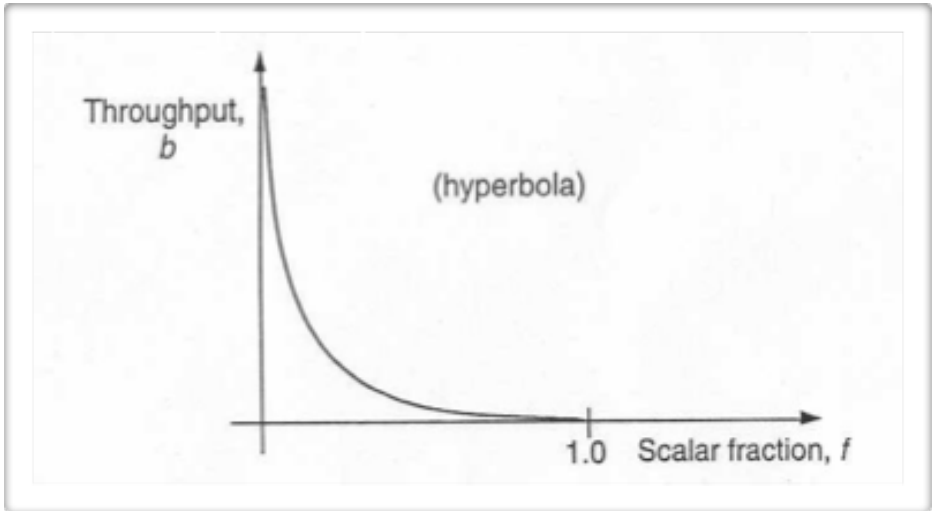
$$T_v = \frac{NT_s}{n} \tag{2}$$

- Average throughput can be obtained by substituting (2) into (1):

$$b = \frac{nb_s}{1 + (n-1)f} \tag{3}$$

Processor Performance

- As b_s (scalar throughput) and n (processor parallelism) are assumed constant, we can plot equation (3)
- If we assume speedup, $S(n) = b/b_s$, then:



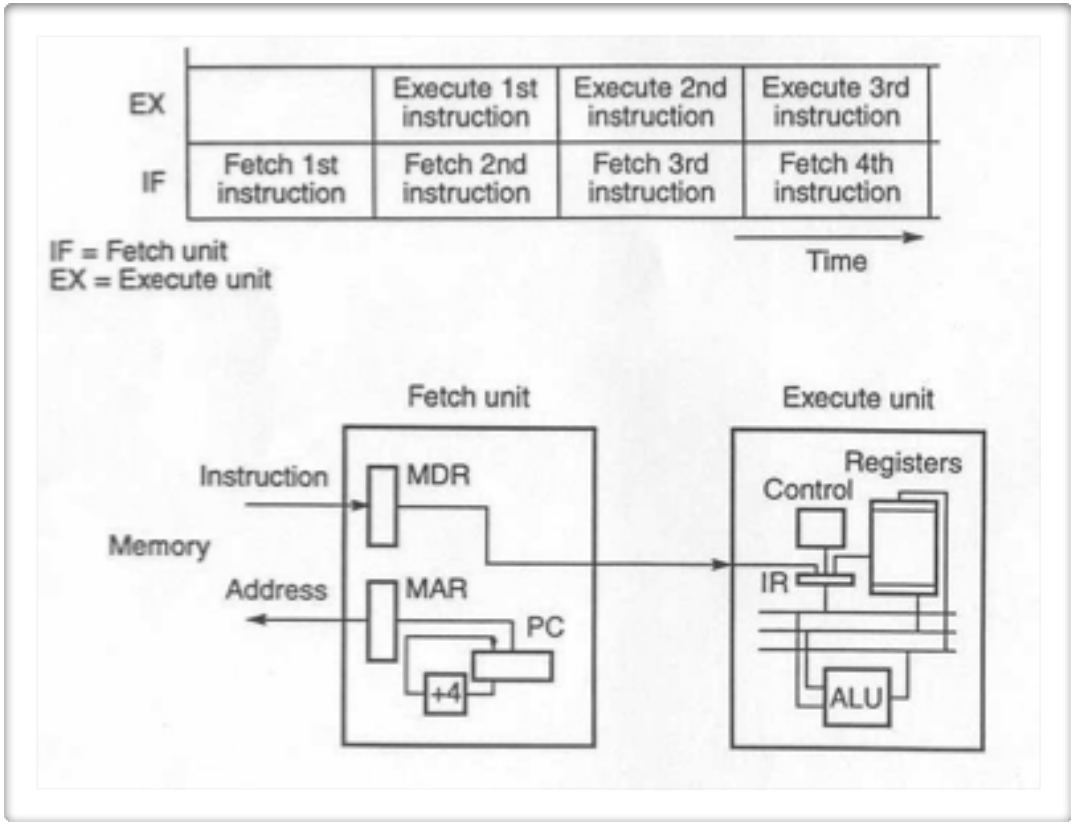
$$S(n) = \frac{n}{1 + (n-1)f} \qquad (4)$$

Where *f* is interpreted as the fraction of non-parallelisable operations

Equation (4) is known as 'Amdahl's Law' (after G. Amdahl, designer of many early computers, e.g., IBM/360)

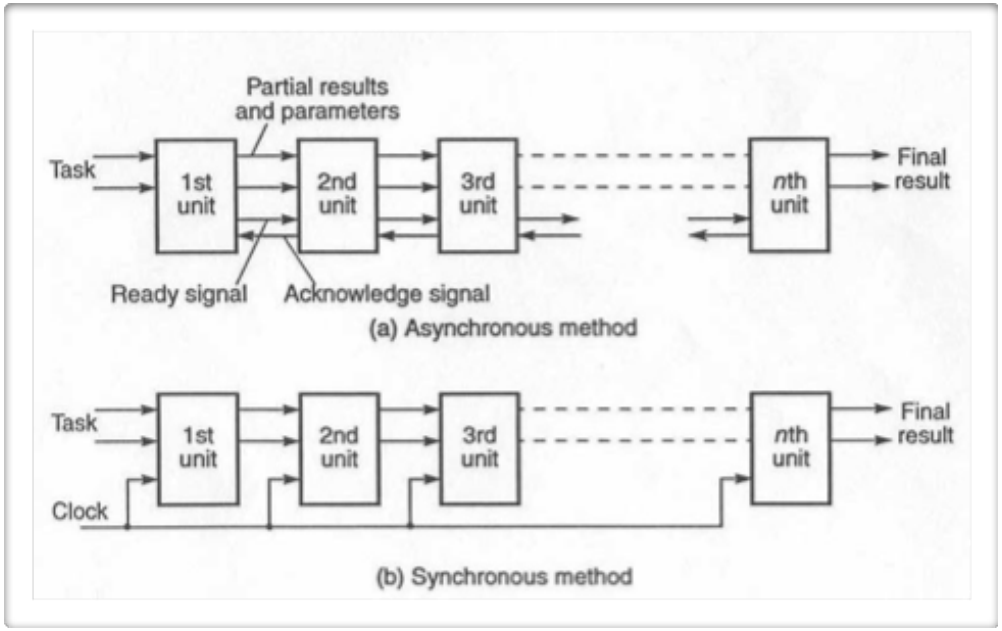
Pipelining

- ***Instruction Pipelining (I/E only)***
- The problem with the I-E pipeline is memory contention (both units attempting to access memory at same time)
- Possible solution:
  - Use banked/interleaved memory, separating instruction and data areas
- The space-time diagram and a possible implementation are shown opposite



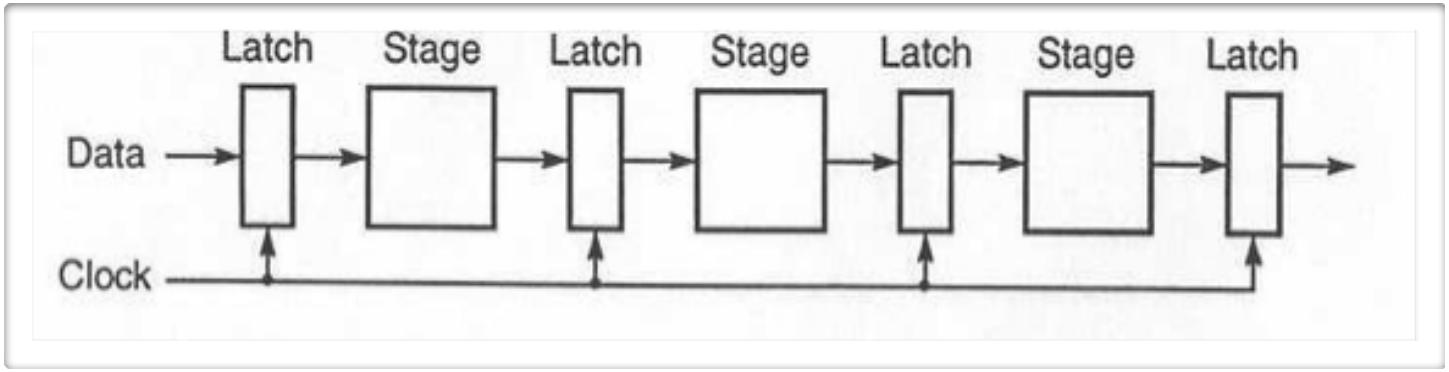
Pipelining – Pipeline Principles

- 2 methods of controlling the transfer of information between stages:
  - ***(i) Asynchronous***
    - Uses pairs of 'handshake signals'
    - Most flexible, allowing variable length FIFO buffers between stages, though, the maximum speed is determined by slowest stage



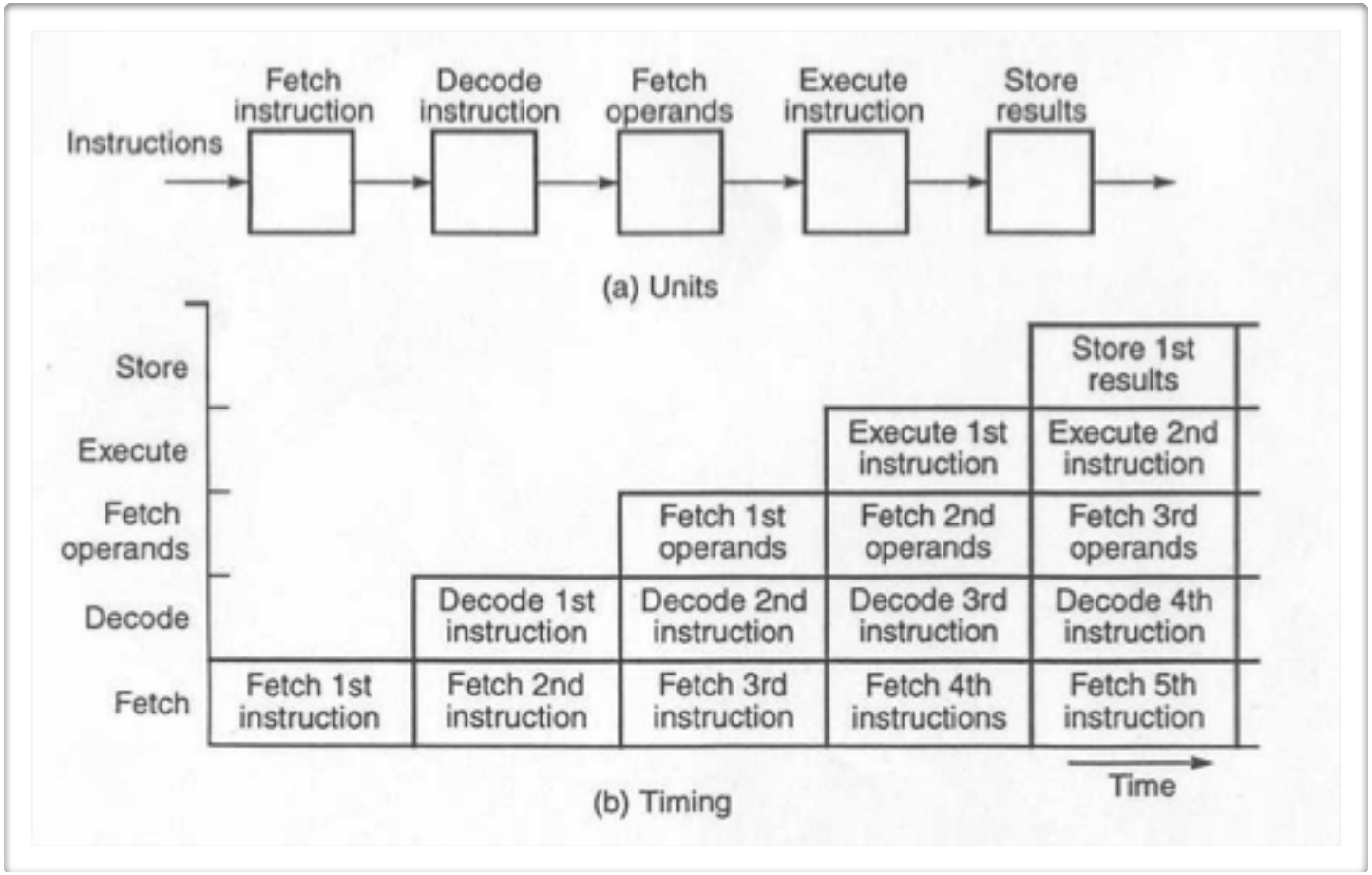
Pipelining – Pipeline Principles

- ***(ii) Synchronous***
  - One timing signal, determined by the slowest stage
  - Most common
  - Analogous to a car production line
  - Instruction and execution unit (arithmetic) pipelines always use synchronous timing, simplifying timing and implementation
  - Synchronous pipelines incorporate latches between stages, often known as ‘staging registers’



Pipelining – Instruction Pipelining

- Further elaboration of the I/E unit system separates the instruction cycle into more stages
- Example: 5-stage pipeline



Pipelining - Pipeline Hazards

- Pipelining assumes the only interaction between stages is the passage of information

  However, we can identify 3 major causes of breakdown of an instruction pipeline

- Known as '***pipeline hazards***' - situations that prevent the next instruction from executing during its designated clock period

  This reduces performance

  - ***(a) Structural Hazards (resource conflicts)***

    Example: 2 stages wishing to use the same memory port in the same clock cycle

    Possible solutions:

    - Use interleaved memory, stages having separate banks
    - Fetch more than one data item at a time, freeing the use of memory at other times

Pipelining - Pipeline Hazards

(b) Control Hazards (procedural dependencies)

Change in order of execution of instructions, e.g., branch and jump instructions (typically account for 10-20% of instructions)

Example: If a 5-stage pipeline, with a 10ns clock period is stalled (emptied) after every 10 instructions, then the average time to process 10 instructions is:

$$\frac{9 \times 10\text{ns} + 1 \times 50\text{ns}}{10} = 14\text{ns}$$

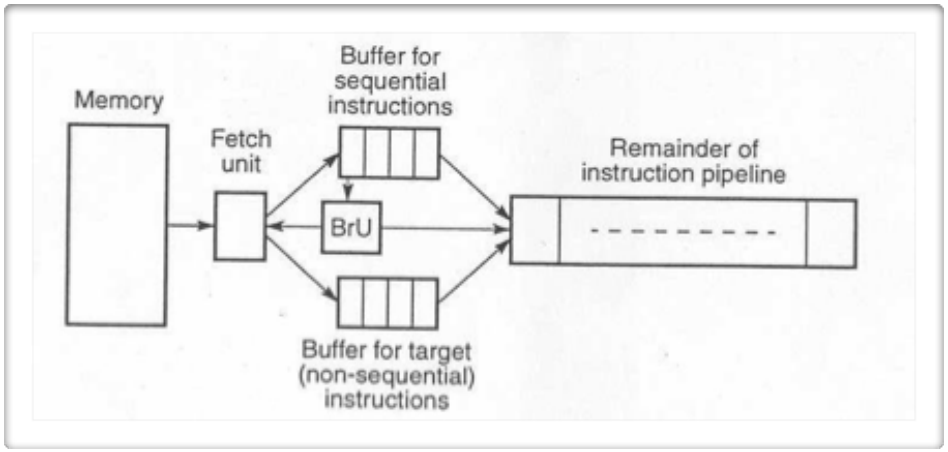
Implying a 40% increase in instruction processing time

- Strategies exist to reduce pipeline failures due to conditional branches

NB. Unconditional branches less of a problem as they are 'predictable'

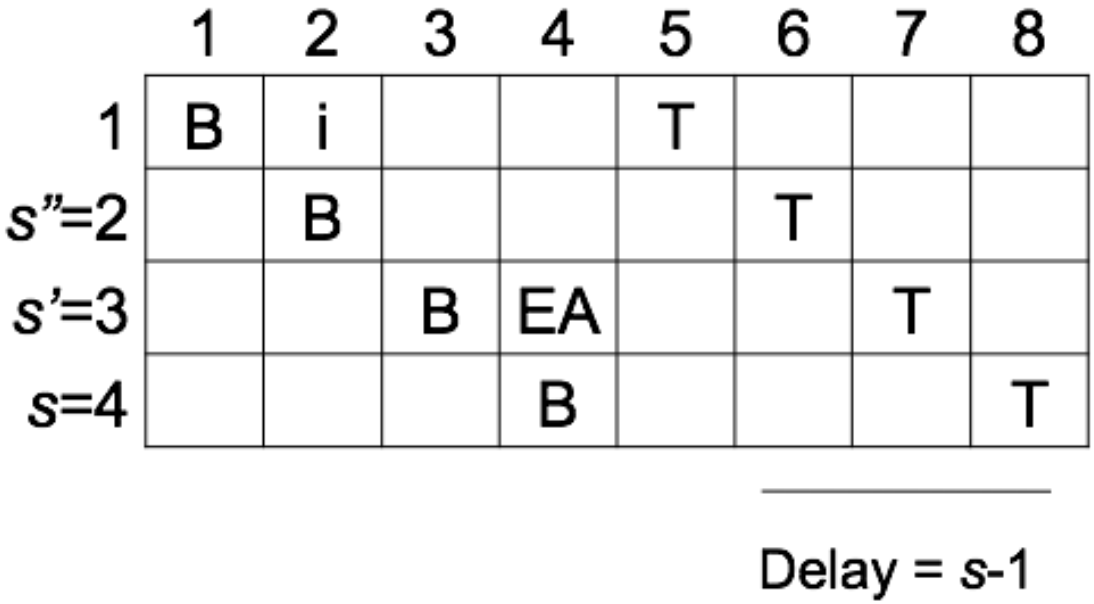
Pipelining - Pipeline Hazards

- ***(i) Instruction Pre-fetch Buffers*** - assumes both addresses of possible sequences known immediately after fetching branch instruction
  - Conditional branches cause both buffers to fill with instructions, assumes use of interleaved memory
  - Separate 'Branch Unit' (BrU) used to execute branch instruction as soon as fetched. When actual 'next address' discovered, instructions taken from appropriate buffer and others discarded
  - Technique known as 'multiple prefetching' or 'branch bypassing'
  - Problem when more than 1 conditional branch is in the instruction stream



Pipelining - Pipeline Hazards

- ***(ii) Pipeline Freeze Strategy***
  - Freeze pipeline on receipt of branch instruction
  - Simple implementation, but poor performance



Pipelining - Pipeline Hazards

- ***(iii) Static Prediction***
  - On receipt of Branch instruction, predict always either Take or Not Take ('hard-wired')
  - 60% of all branches are taken, therefore to predict Take may appear to be the best policy
  - To predict Not Take may be better as pipeline cycles are not lost, and overall (average) performance may be better.
- ***(iv) Dynamic Prediction***
  - Predict on-the-fly for each branch instruction

    Three general forms:

    (1) Predict on branch instruction characteristics - provide a bit in the branch instruction (set by programmer or compiler) to indicate the most likely outcome of branch, e.g., if loop is being programmed, set bit to 'Take Branch'

Pipelining - Pipeline Hazards

(2) Predict on Target Address Characteristics - short branches and backward branches are characteristics of loops in a program and are usually taken, while long branches are used for major changes in program control and are usually not taken

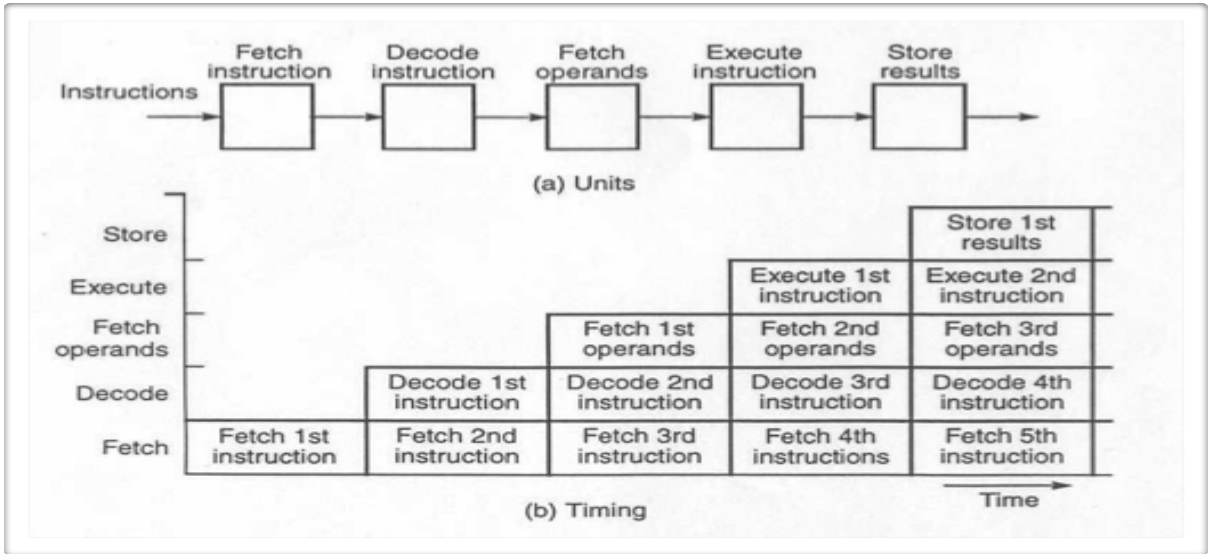
(3) Predict on Branch History - if bit is set or reset on the outcome of an execution, that bit is a good predictor of the next execution

Problem: The bit really needs to be in the instruction resulting in self-modifying code

Solution: have a field of history bits in instruction cache but not actually part of the instruction

Research shows the more bits in the history field, the better the prediction (2 bits common)

Pipelining - Pipeline Hazards



- ***(c) Data Hazards (resource conflicts)***

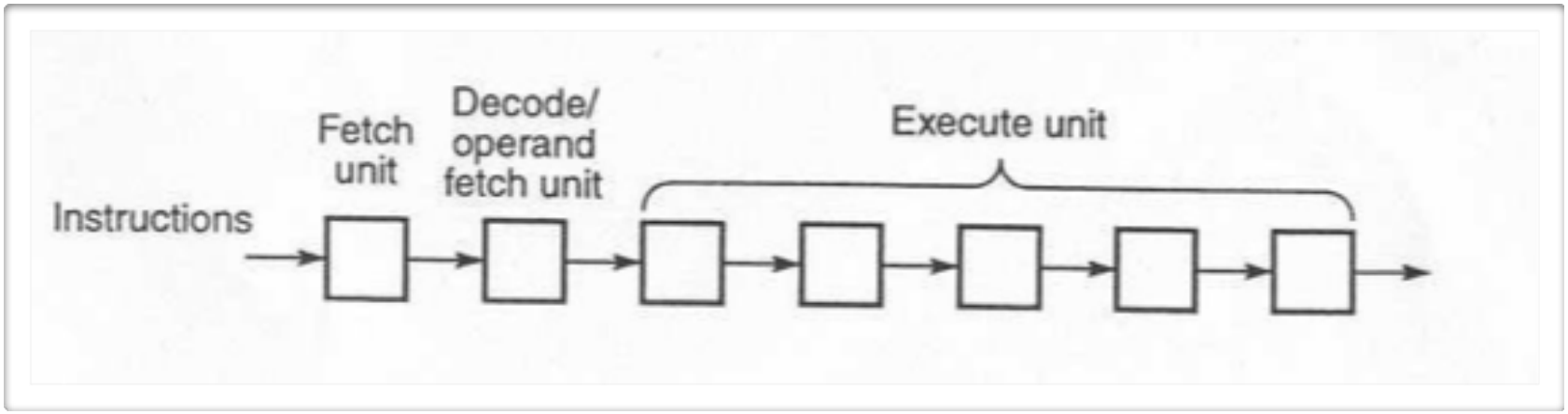
Often the data that instructions use, depend upon data created by other instructions

- Example: R3 = R2 + R1
  R4 = R3 – 1

Reading R3 in instruction 2 would be wrong before instruction 1 has written the new value of R3 to the storage register

Pipelining - Pipelined Execution Units

- Synchronous instruction pipeline clock period is determined by the slowest stage
  - Slowest stage is normally the ‘execution stage’, especially complex floating point operations
- Hence, to gain maximum performance increase, it may be preferable to pipeline the execution unit or have multiple execution units



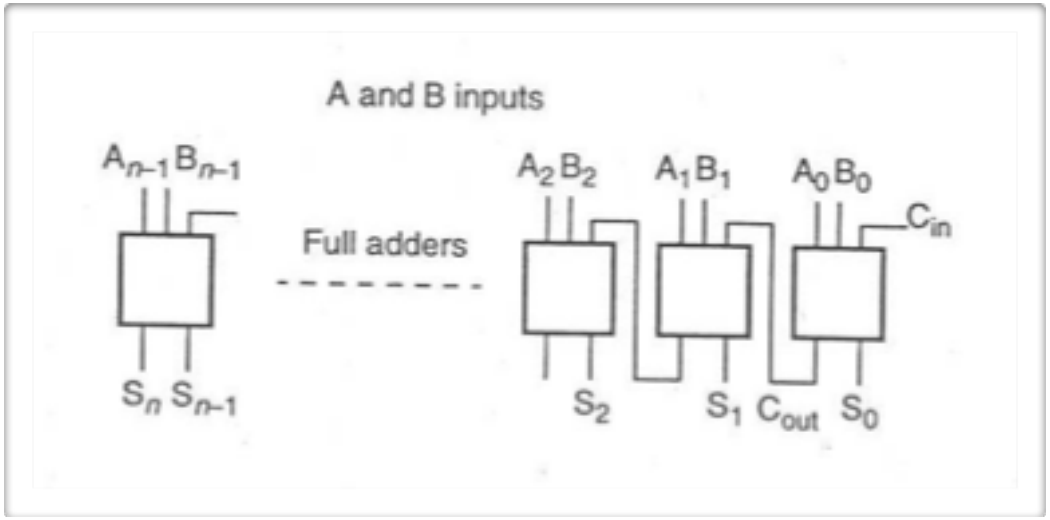
Pipelining - Pipelined Execution Units

- Simplest pipeline linkage is a linear array

- A 2D array permits propagation of information in more than one direction
- Example: binary addition

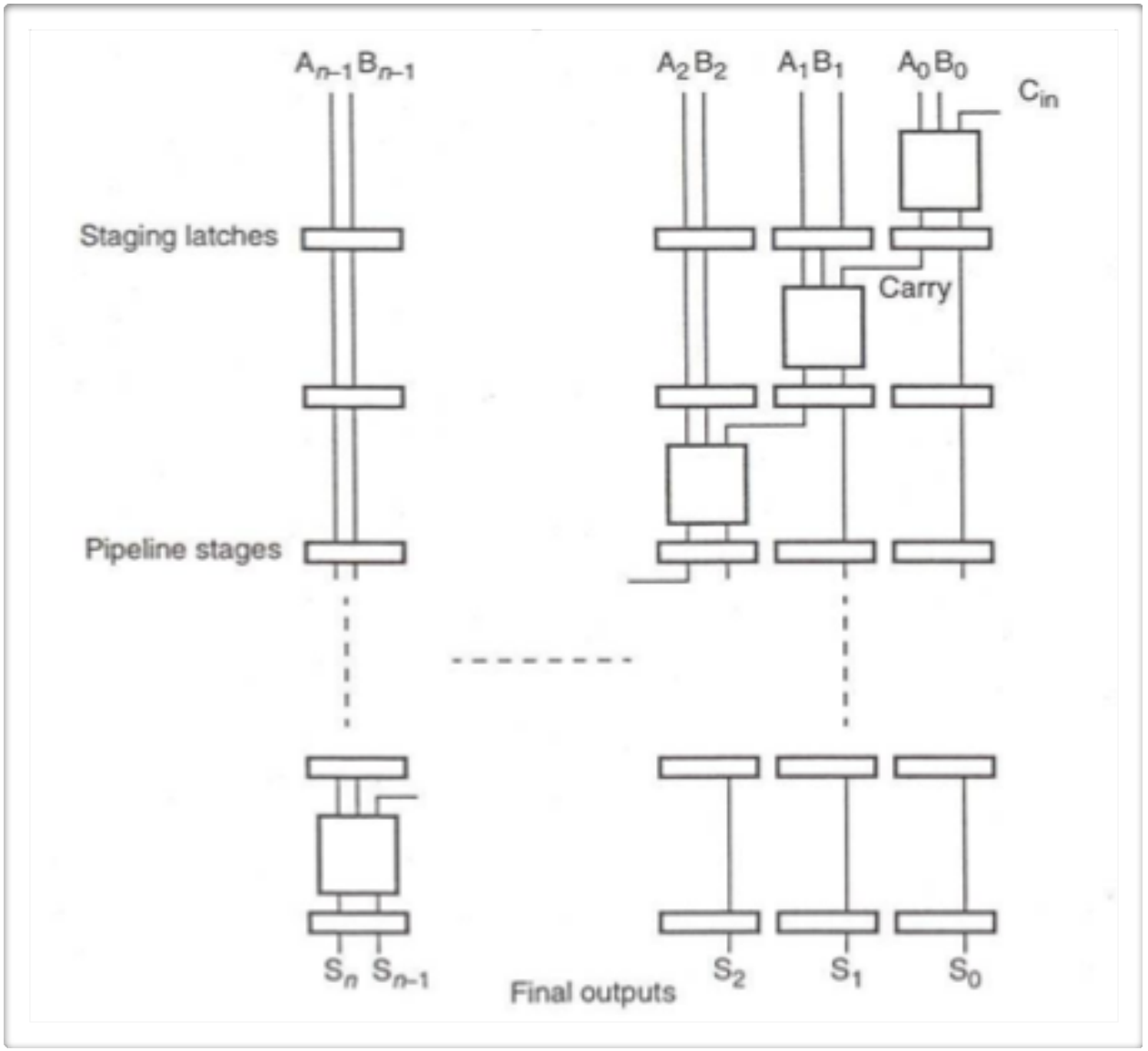
A: 01011
B: 00101
Carry: *01110*

Result: 10000



- Pipelined binary adder is essentially a linear array with staging latches

Pipelining - Pipelined Execution Units



Pipelining - Pipelined Execution Units

- It is sometimes useful to add feedback between stages (recursion)

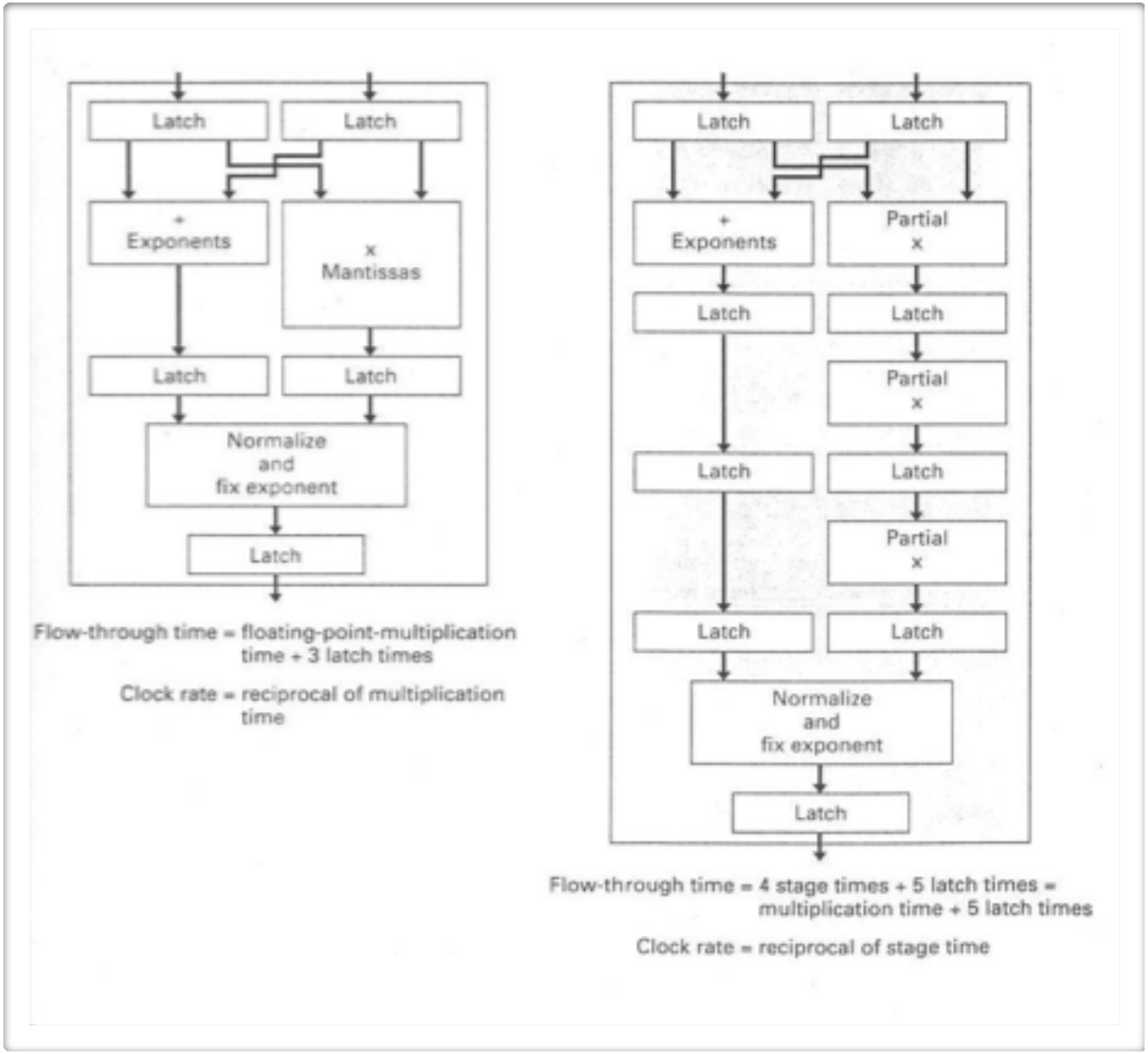
  Output of one stage becomes the input to a previously used stage, e.g., in accumulation

- Alternative designs of a pipeline implementation are often possible, which may lead to performance tradeoffs

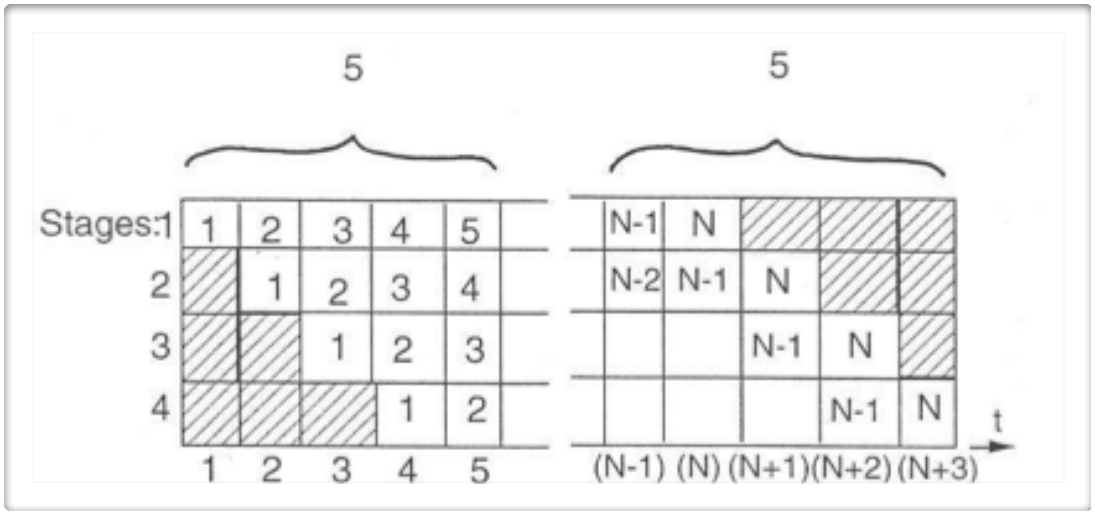
  - Example: Two designs for reciprocal floating-point multiplication

    The highly-pipelined unit (with more stages) produces vector results at a higher rate but takes more time to perform a single multiplication

Pipelining - Pipelined Execution Units



Pipelining - Space-time Diagrams



- The above diagram depicts a 4-stage linear pipeline
- All stages utilised except at the beginning and end of the sequence

$$\text{Efficiency} = E_n = \frac{\text{Busy Area}}{\text{Total Area}} = \frac{N}{(N+3)} = E_4$$

$$\text{Speedup} = S(n) = nE_n, \text{ Therefore: } S(4) = \frac{4N}{N+3}$$

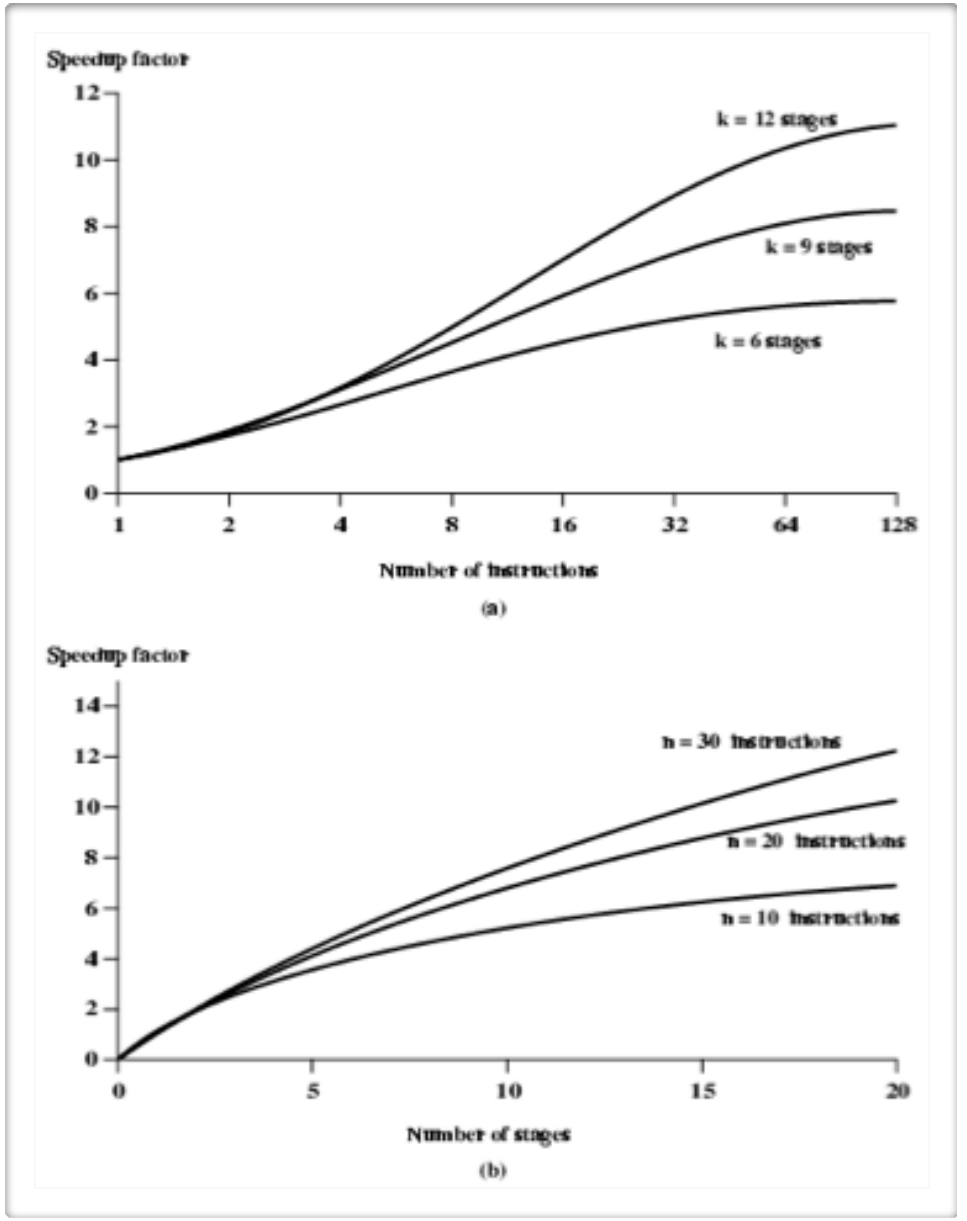
Pipelining

- In general, Speedup is given as:

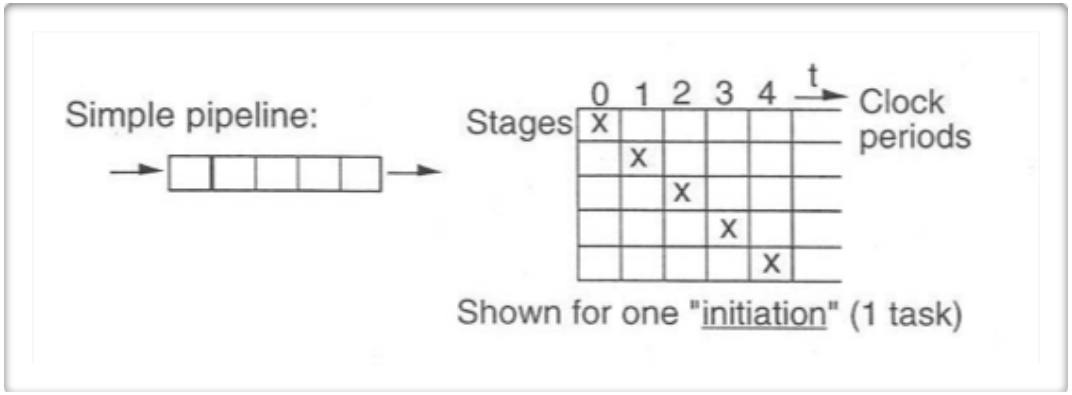
$$S(n) = \frac{Nn}{n + (N - 1)}$$

  Where n is the number of stages, and N is the number of instructions executed

- As might be expected, as $N \rightarrow \infty$ so Speedup $\rightarrow$ n



Logical Design of Pipelines

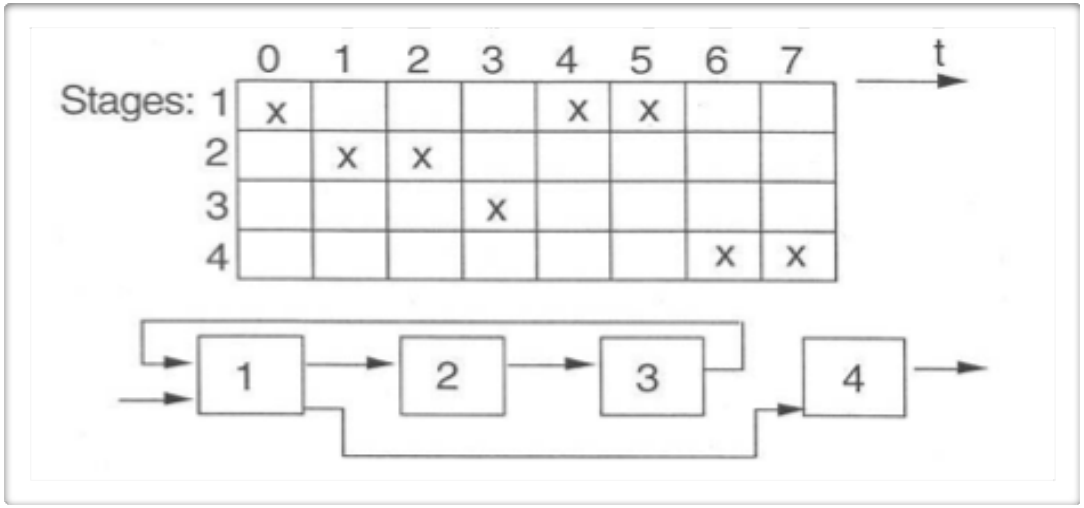


- Complex pipelines with feedback and unequal clocked stages can be difficult to design and optimise

  Useful to use techniques based on ***reservation tables*** (example shown above)

- It is possible that:
  - (i) Stages operate for more than 1 time period: X's in adjacent columns of the same row in the reservation table
  - (ii) Feedback: more than one X in a row, but not in adjacent columns
  - (i) and (ii) indicate that pipelines may not accept initiations at the start of every clock period, otherwise ***collisions*** occur

Logical Design of Pipelines

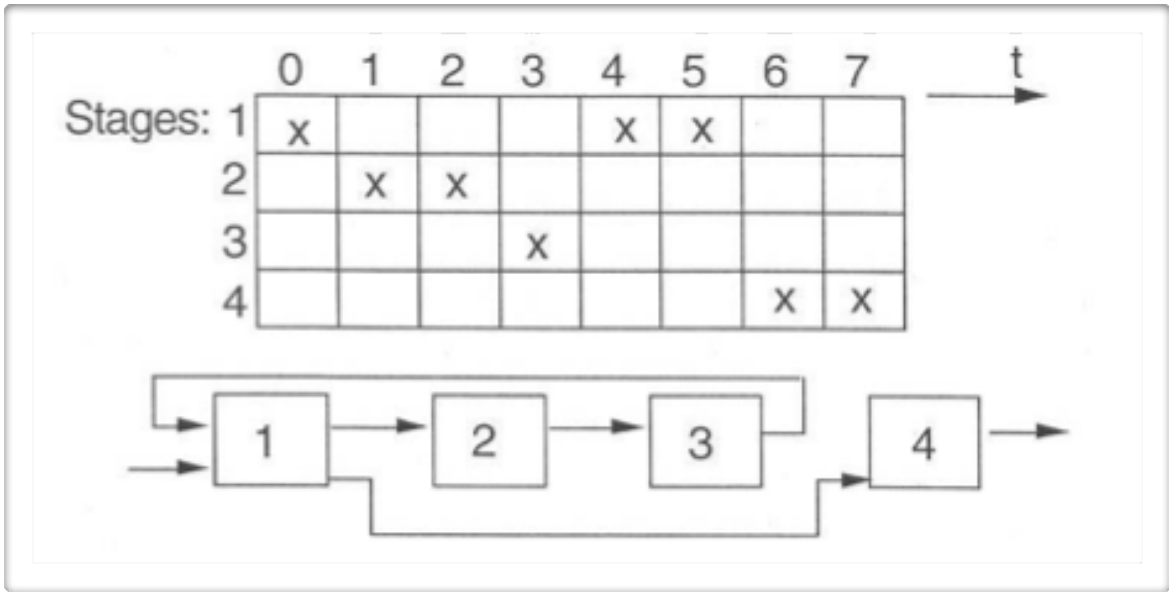


- Consider the above example:
  - Collision occurs in time slots 1 & 2 if there are two consecutive initiations
- Potential collisions identified by noting the distance in time slots between X's in each row – this provides data for the '***collision vector***'
- Collision Vector, $C = C_{n-1}\ C_{n-2}\ \ldots\ C_2\ C_1\ C_0$
- $C_i = 1$ if a collision would occur with an initiation *i* cycles after a previous initiation, otherwise $C_i = 0$ (Note: $C_0 = 1$ always)

Logical Design of Pipelines

- The '***initial collision vector***' is the collision vector representing the state of the pipeline after the first initiation
- To compute the initial collision vector, we need only consider the distances between all pairs of X's in each row
- For previous example (shown below) the distances between all pairs are: 5, 4, 1

  This gives an Initial Collision Vector of 110011 (including C_0)



Logical Design of Pipelines

- We need a control/scheduling mechanism to determine when new initiations can be accepted without a collision occurring

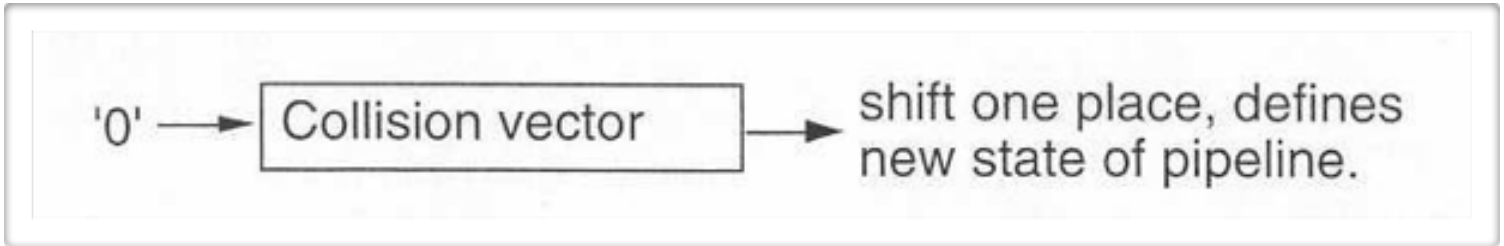
- First, let us consider some terms
  - ***Latency*** - The number of clock periods between two initiations
  - ***Average Latency*** (AL) - The average number of clock periods between initiations over some specified repeating cycle
  - ***Minimum Average Latency*** (MAL) - The smallest possible Latency considering all possible sequences of initiations

    MAL may well be the goal for optimum design

Logical Design of Pipelines

- ***A Scheduling Strategy*** (Davidson, 1971)
- A pipeline changes from one state to another as a result of initiations, therefore it seems logical to represent the activity in a pipeline by a '***state diagram***'
- State diagram: diagram of linked states, identified by status vectors. NB. Initially, with an empty pipeline, the status vector is 00..00
- After the first initiation, the status vector becomes the initial collision vector, and C_1 will indicate whether another initiation is permitted in the next cycle
- Hence the algorithm:

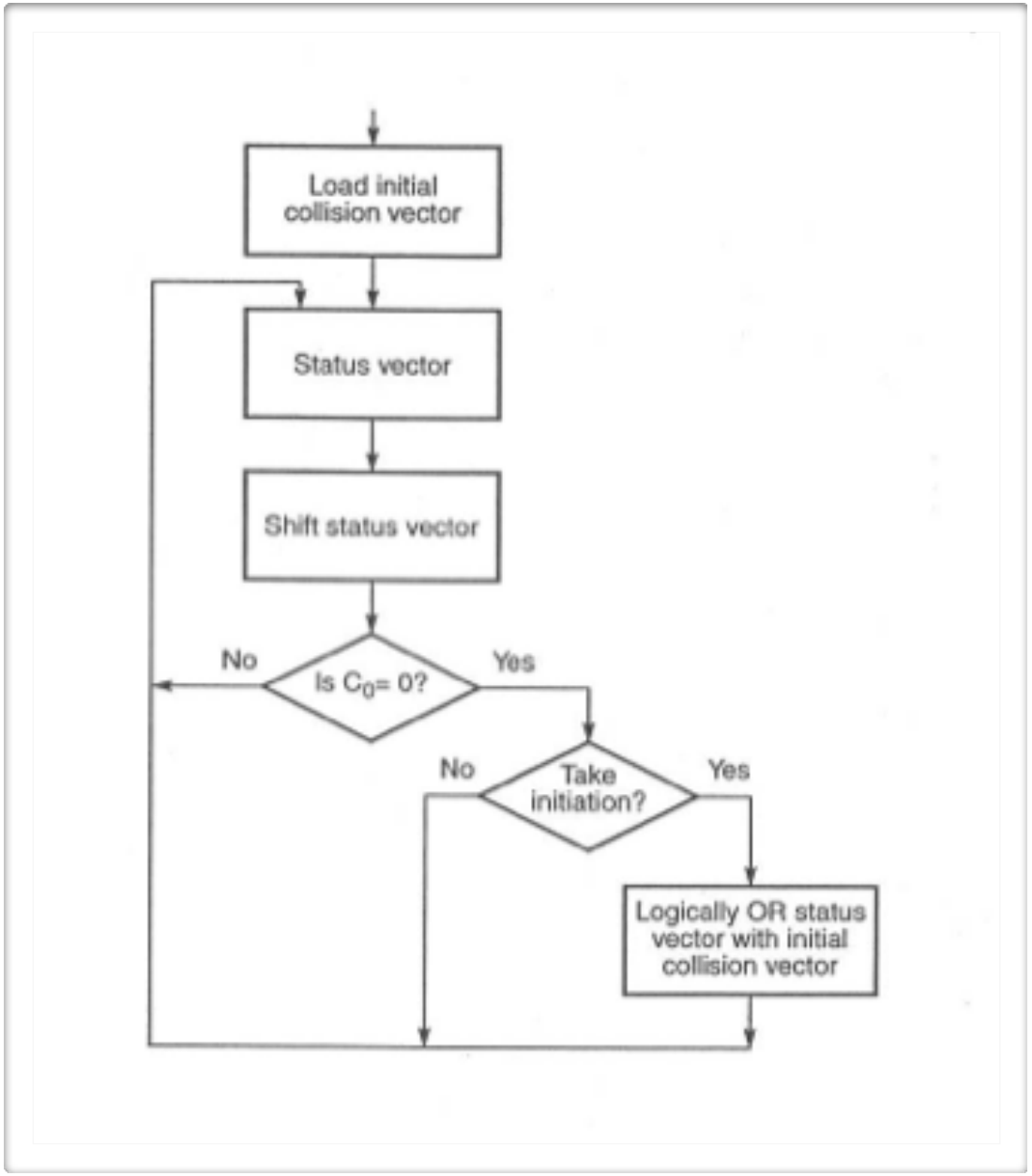
Logical Design of Pipelines



- If the shifted status vector has $C_0 = 1$, no new initiation is possible.

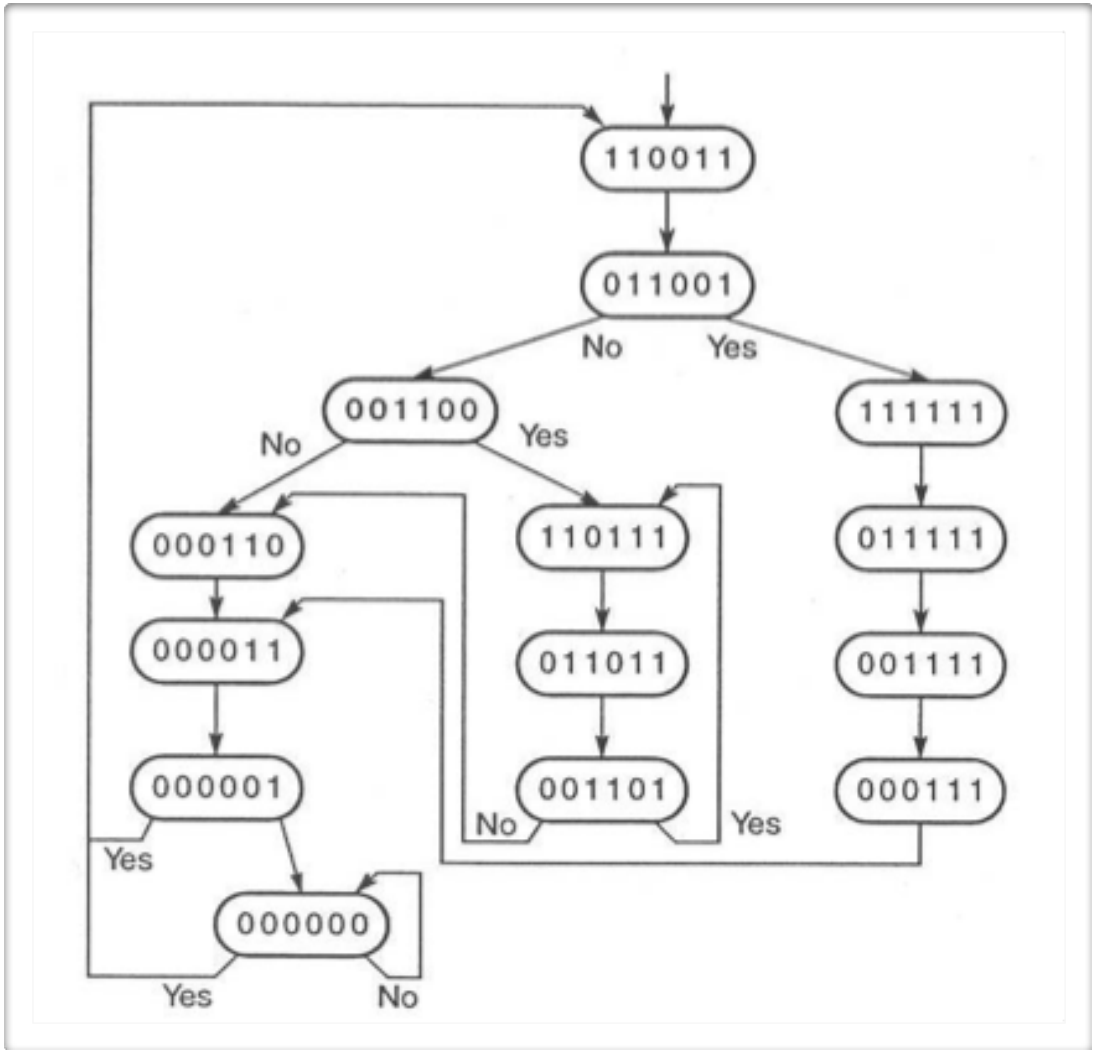
- If the shifted status vector has $C_0 = 0$, there are two possibilities:
  - (i) Initiation is not taken (i.e., as for $C_0 = 1$)
  - (ii) Initiation is taken, hence the new status vector is given by the bit-wise logical OR of the shifted vector and the initial collision vector

Logical Design of Pipelines

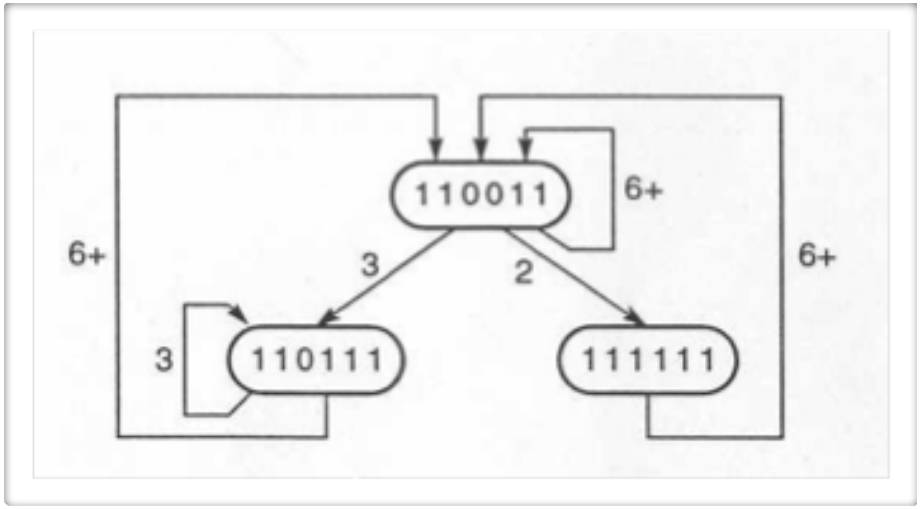


Logical Design of Pipelines

- Initial collision vector: 110011 - used to generate the state diagram



Logical Design of Pipelines



- The state diagram is reduced to show only those changes when initiations are taken (those states that result from 'Yes'), as shown above

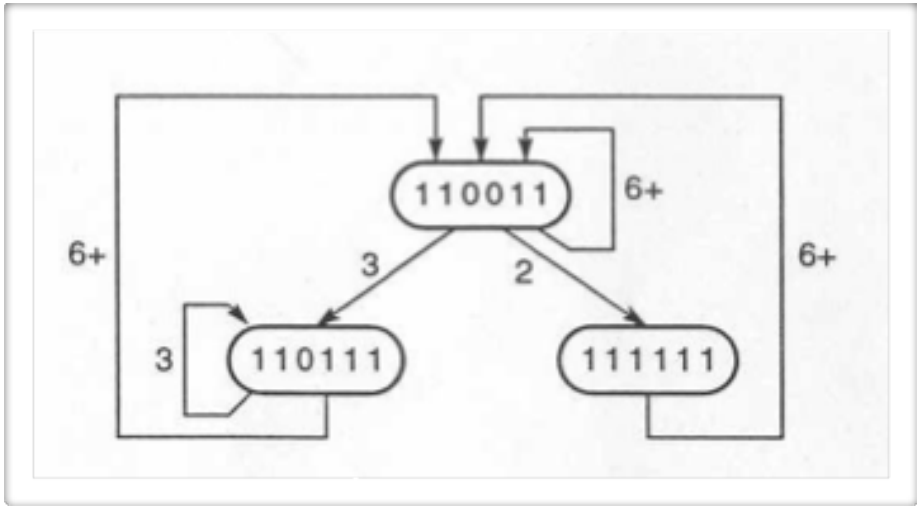
  The numbers indicate the number of clock periods to reach the next state shown

- We can identify possible closed '***simple cycles***', cycles in which a state is visited once during the cycle:

  3, 3, 3, 3; 2, 6, 2, 6; 3, 6, 3, 6; 6, 6, 6, 6

  These are written as: (3), (2, 6), (3, 6), (6)

Logical Design of Pipelines



- Always taking initiations when $C_0 = 0$, i.e., minimum latency, is called a '***greedy strategy***' – this will not necessarily result in MAL, but will normally be close

  Greedy Strategies are easy to implement

- There is often more than one '***greedy cycle***', if the starting point is other than the initial state, i.e, (2, 6) starting from 110011 to give an average latency of 4, (3) starting from 110111 to give an average latency of 3

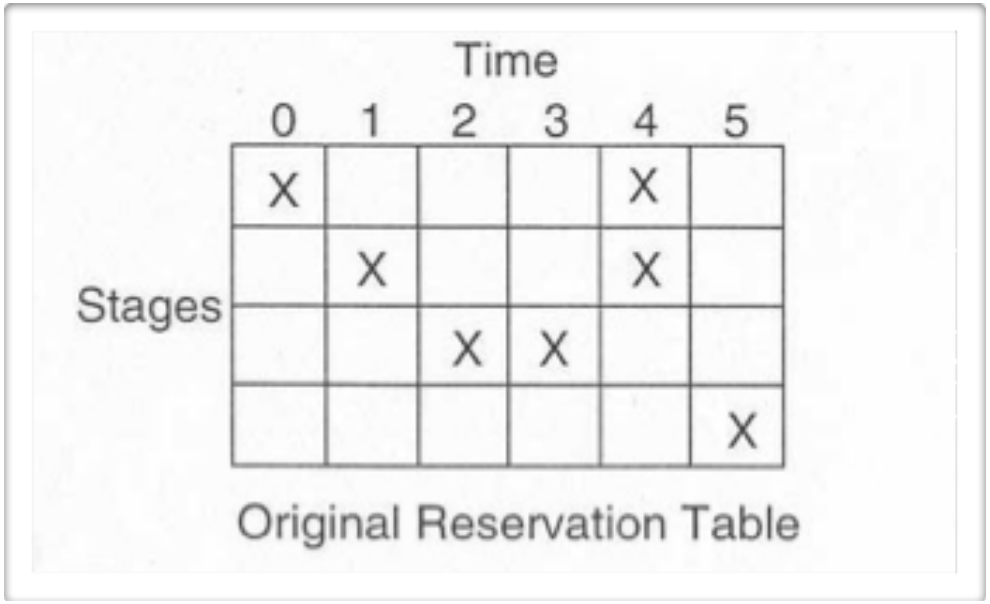
Logical Design of Pipelines

- Average latency for a greedy cycle ≤ Number of '1's in the initial collision vector
  - It has been shown that there is always a simple cycle with AL ≤ AL of a complex cycle, so we only need to consider simple cycles
- Minimum average latency ≥ Maximum number of 'X's in any reservation table row
  - This gives a lower bound on latency (3 in our worked example)
- This is deduced as follows:
  - Let the maximum number of X's in a row be n_{max} = the number of times the most-used stage is used by one initiation
  - Given that there are t time slots in the reservation table
  - The maximum number of initiations = t / n_{max}
  - Therefore, minimum latency = t/(max. number of initiations) = n_{max}
- We now have upper and lower bounds on latency
  - Max X's in row ≤ MAL ≤ Greedy cycle AL ≤ No. of 1's in the initial collision vector

Logical Design of Pipelines

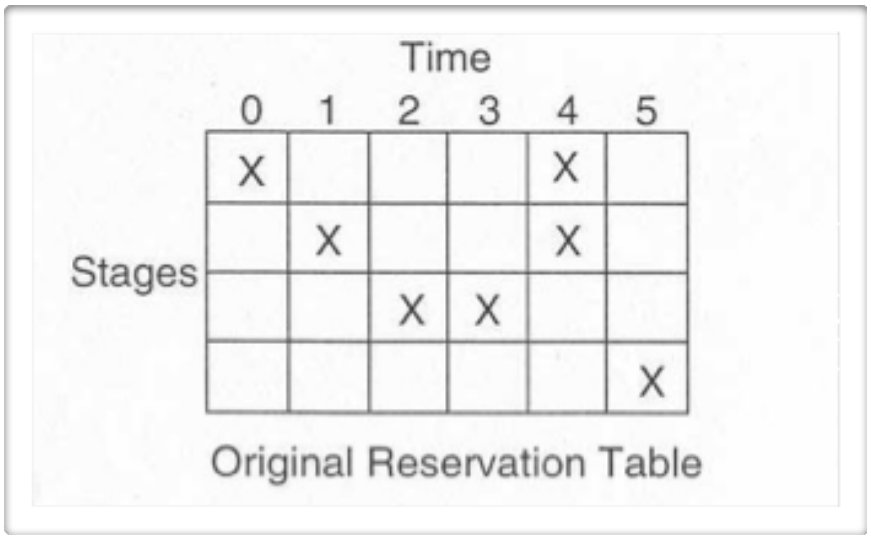
- **Reducing latency**
  - A given pipeline design may not give the required latency
  - A method of reducing latency is to insert delays into the pipeline to expand the reservation table (increase the number of time slots) and thus reduce the chance of collisions
  - In general: Any fixed latency ≥ Lower Bound can be achieved by adding delays
  - An algorithm to identify where to place delays for a latency of *n* cycles is:
    - Starting with the first X in the original reservation table, enter an X in a revised table and mark as forbidden (F) every n cycles (to indicate the positions reserved for initiations every *n* cycles)
    - Repeat for all X's until X falls on a 'F', then delay X for one or more positions until a free position is found
    - Remark all delayed positions with D (delay). Delay all subsequent X's by the same amount

Logical Design of Pipelines



- Consider the above example
- Initial collision vector: 11011
- One simple cycle: (2,5), giving an average latency of 3.5
- Lower bound on latency = 2 (= max. number of X's in any row)

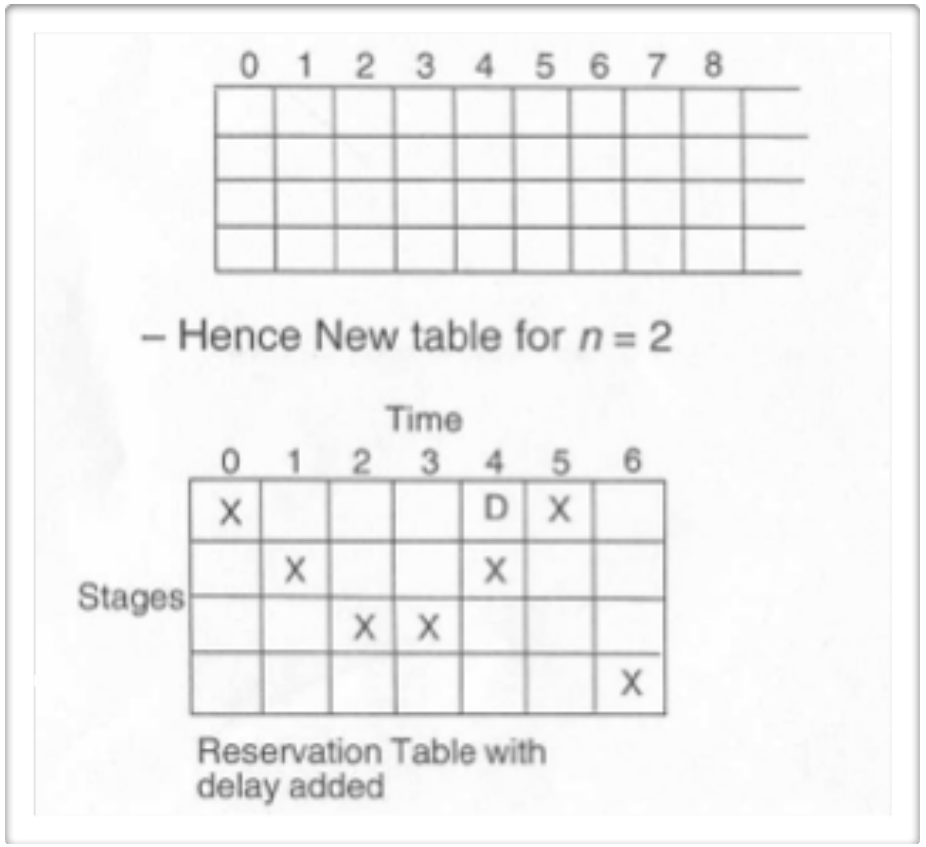
Logical Design of Pipelines



X		F		X			
	X		F	X	F	F	F
		X	X	F	F	F	F
					X		F

Logical Design of Pipelines

- Only one delay required
- Implementation of delay:
  - 1 cycle: Use a simple latch
  - 2 or more cycles: Use multiple latches or dual ported memory (read locations written *n* cycles previously)



Example Sheet 2

- Please attempt to answer the questions from Example Sheet 2 - worked solutions will be discussed in Graham's next lecture