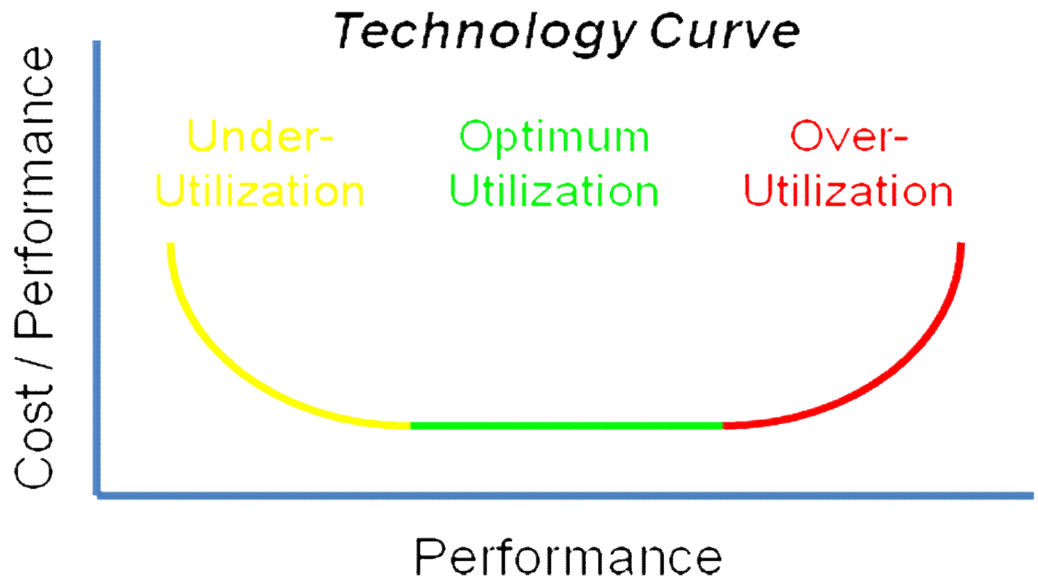
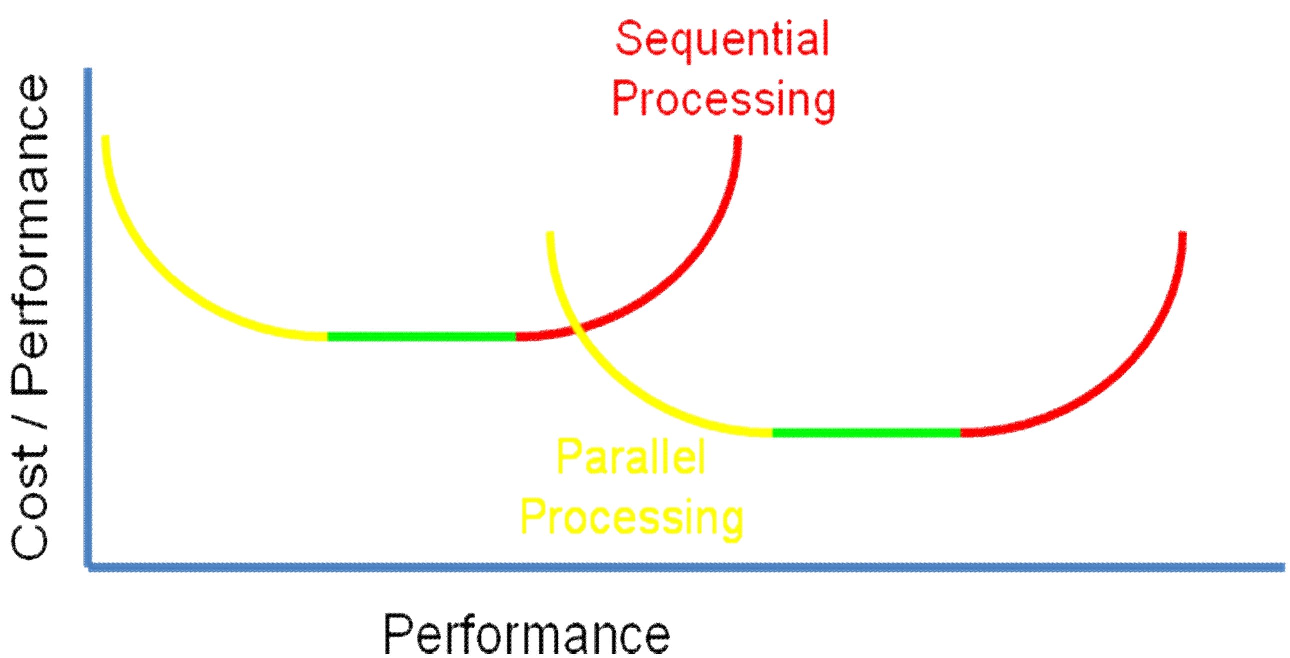
Business Economics of Parallel Processing

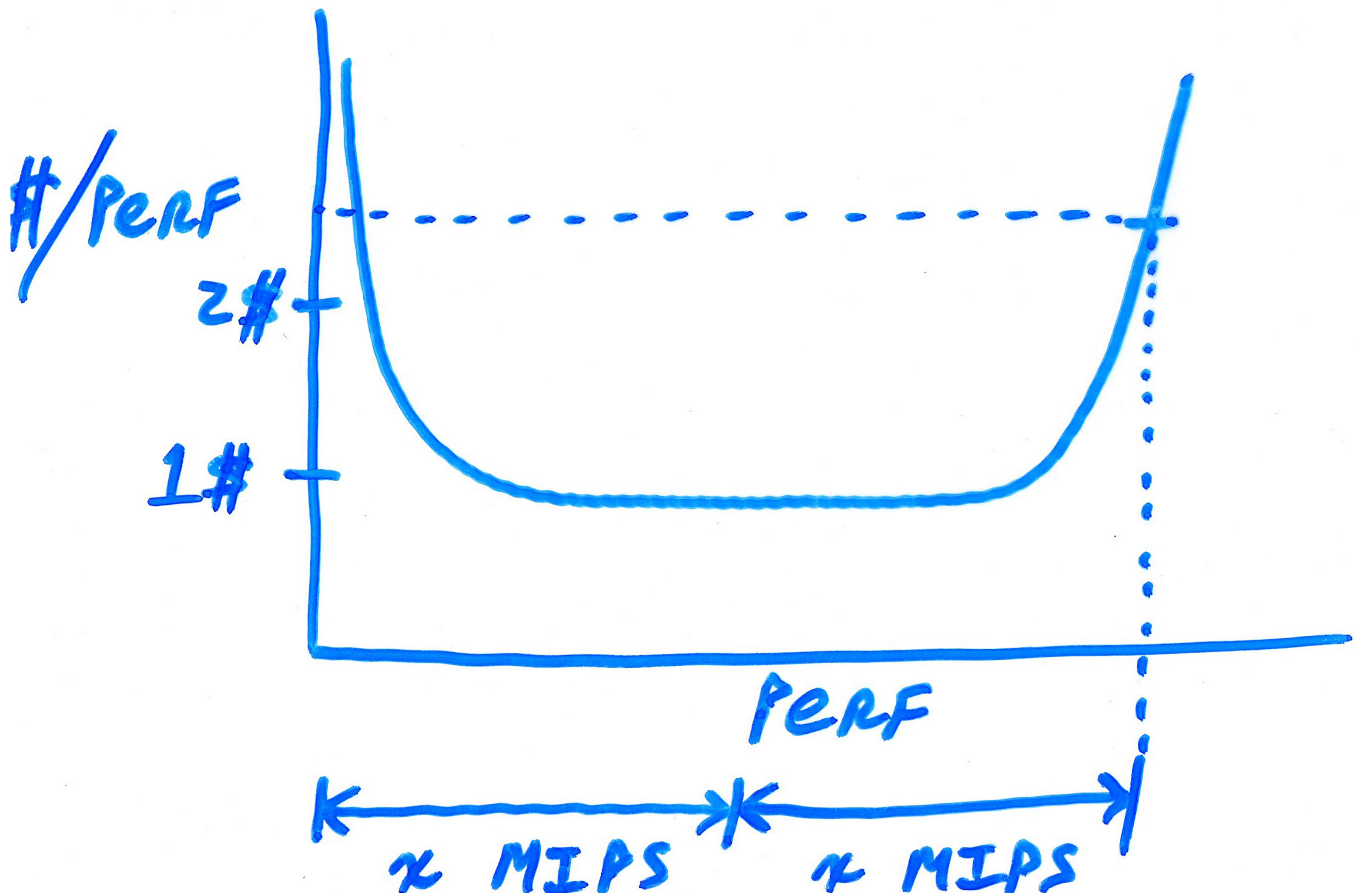
- **Cost per Performance vs. Performance (Time) Curve for Technology**
  - **- Exhibits 3 Distinct Regions over the lifetime of the Technology**
    - **1) Under-Utilization: Initially high C/P, but trending lower to Optimum**
      - **Technology is just introduced consisting of brand new features**
      - **New Capabilities require a "Learning Curve" by Designers**
        - **Learning incurs time and therefore, increases Cost**
        - **New features not fully utilized and therefore, decreases P**
      - **Designers should attempt to move down this curve ASAP**
    - **2) Optimum Utilization: Lowest C/P of lifecycle, trending is flat with P**
      - **"Sweet-Spot" Operating Range ("Envelope") of the Technology**
      - **Designers cost-effectively utilize full capability of the Technology**
      - **C/P flat because P is within operating range of Technology**
        - **Evident as different "Models" of same basic design using T**
    - **3) Over-Utilization: Initially Optimum C/P, but trending higher with P**
      - **Technology is being pushed too hard, so C/P rises quickly**
      - **Designers forced to spend time (and cost) optimizing design**
      - **If higher P desired, need to "Hop" to a newer Tech Curve quickly**



- **The time has come to Hop to the New Technology Curve of Parallel Proc**
  - **- Transition may (temporarily) incur not just higher C/P, but even lower P**

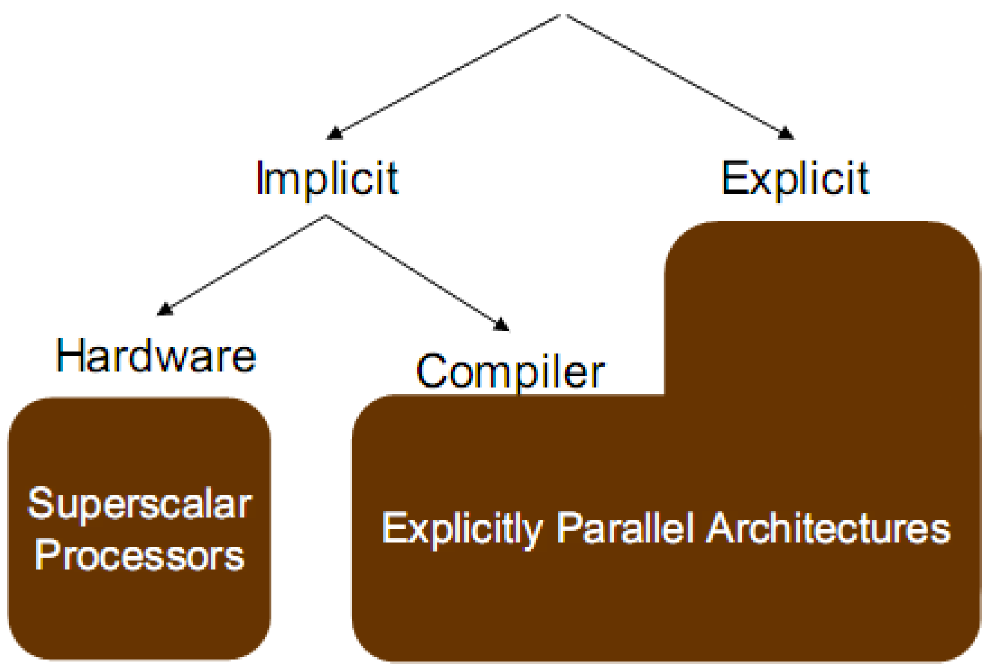


ECONOMICS OF PARALLEL PROCESSING



Implicit Hardware Parallel Processing

- **Implicit Parallel Processing**
  - **- Transparent to System-Level and Application-Level SW Programmers**
  - **- Handled completely, and automatically, by built-in on-chip HW**
  - **- No Explicit External steps or specifications needed by SW Programmer**
    Programmer doesn't even know "Parallelism" is being implemented
  - **- Goal is to Provide Speedup while preserving Sequential Behavior**
    From a SW viewpoint, Results will be Identical to a Sequential Model



- **We will study Implicit HW PP first, then Apply Ideas Explicitly in SW PP**

- **Parallel Processing vs. Sequential Processing**

  Sequential Algorithms are advantageous in that they have:

  - **Simpler Software Representations**

    Programs are written in a simple language and execute linearly

    "Easy" to Design and Debug because of "One Thing at a Time" Model

  - **Simpler Hardware Realizations**

    Smaller number of components (1 CPU, 1 Control, 1 Memory)

    Simple Interconnection Scheme and Timing Synchronization

  - **Simpler Temporal Behavior for Human Understanding**

    People get confused when too many things happen at same time

  Parallel Algorithms are advantageous in that they offer:

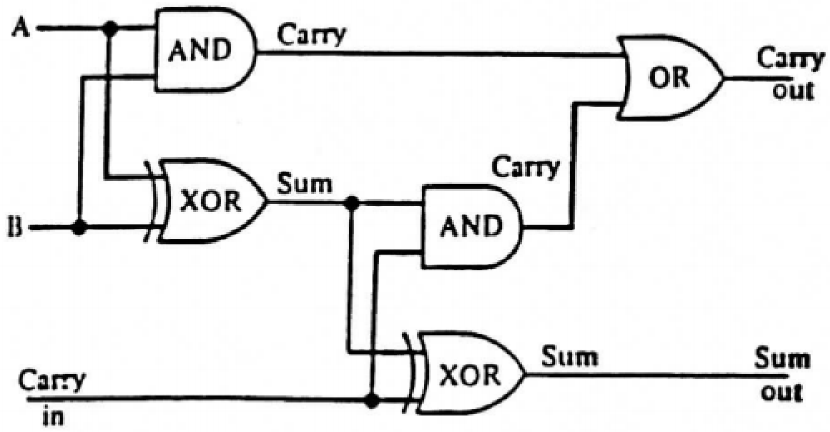
  - **Speed**
  - **Cost Effectiveness**

    Given a technology curve, it is cheaper to use 2 CPU's in Parallel vs. trying to push (over-utilize) single CPU's performance to 2x faster

- **Hardware and Software Designers Go Parallel**

  Not because it is easier, but because they want to go faster, cost-effectively

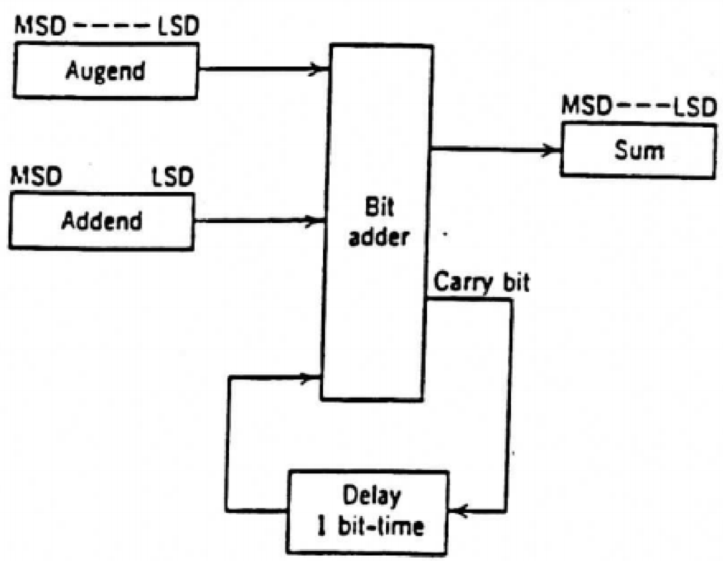
- **Parallel Processing can be Viewed and Implemented at Various Levels**
  - **- Logic Level: HW is inherently “Parallel” since all gates work at same time**

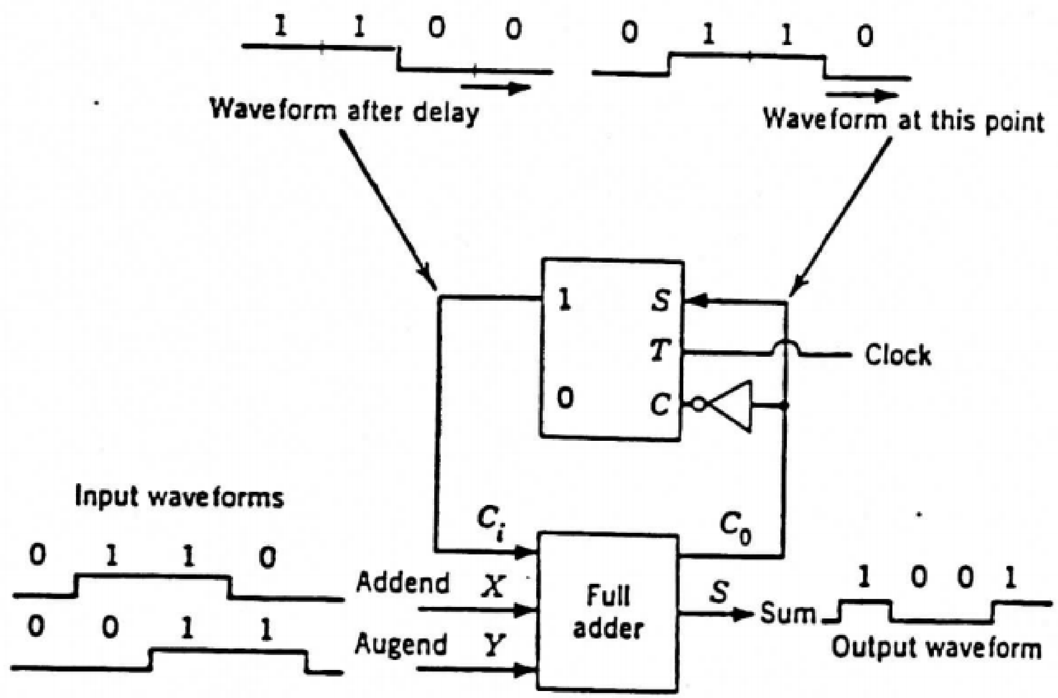


Truth Table

Inputs			Outputs	
A	B	Carry in	Sum out	Carry out
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

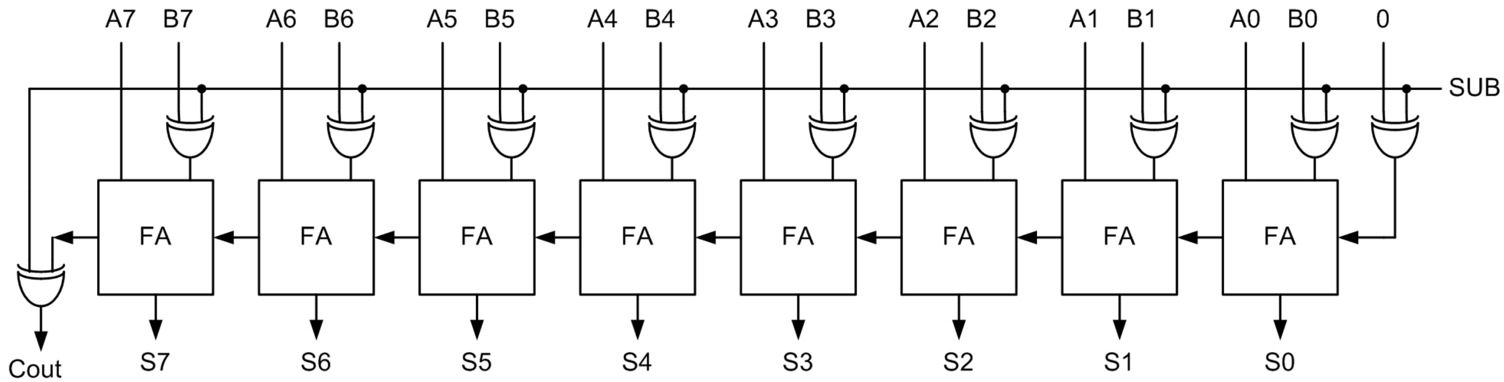
Serial (bit-wise Sequential) Adder is Simple, Cheap, but Slow





By Using a Full Adder per bit, we can attempt a Parallel Architecture

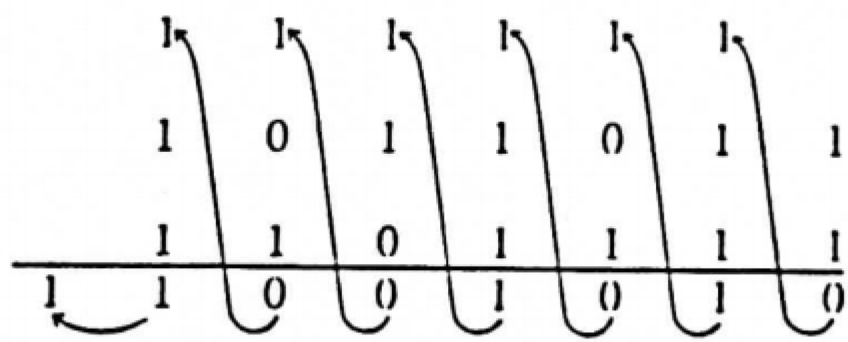
"Simultaneous" processing of several bits (e.g. 8) via parallel H/W



8-bit Ripple Carry Adder Architecture

Ripple Carry Adder actually illustrates a sub-optimal parallelism scheme

- Although it uses 8x more HW (FAs) than the Serial, It is not 8x as fast
- Limited by the Ripple effect of the Carry Data between FA units
  - Each FA (column) needs to wait for the carry from FA to its right
  - Data Dependency effectively creates a Sequential bit-by-bit adder



- **Instruction Level (Microscopic and Macroscopic)**
  - **Microscopic view:**

    Intra-Instruction Concurrency

    A single instruction is divided into different phases or stages

    Possible implementation: Pipelining

    Real-Life Example: Laundry (Stage 1: Washer; Stage 2: Dryer)

    Two successive loads of laundry can be 'In Flight' at same time

    Overlap phases of consecutive instructions to Increase Throughput

    1) IF: Instruction Fetch **2) ID: Instruction Decode**

    3) EX: Execute Instruction **4) WB: Write Back Result**

	Cycles							
Instruction #	1	2	3	4	5	6	7	8
Instruction i	IF	ID	EX	WB				
Instruction i+1		IF	ID	EX	WB			
Instruction i+2			IF	ID	EX	WB		
Instruction i+3				IF	ID	EX	WB	
Instruction i+4					IF	ID	EX	WB

- Macroscopic view:
  - Inter-Instruction Concurrency
  - Simultaneous execution of several instructions
  - The program is sequential, but many steps can be done in parallel
    - Assuming Data Independence between instructions run in parallel
  - Possible Implementation: SuperScalar Architecture
    - Multiple Function Units: Put both an Adder and Multiplier in HW
      - Now, one pair of numbers can be added, and at the same time, another pair of numbers can be multiplied
- The two above techniques (Micro- and Macro-) can also be combined

	Cycles						
Instruction type	1	2	3	4	5	6	7
Integer	IF	ID	EX	WB			
Floating point	IF	ID	EX	WB			
Integer		IF	ID	EX	WB		
Floating point		IF	ID	EX	WB		
Integer			IF	ID	EX	WB	
Floating point			IF	ID	EX	WB	
Integer				IF	ID	EX	WB
Floating point				IF	ID	EX	WB

- **Program Level**

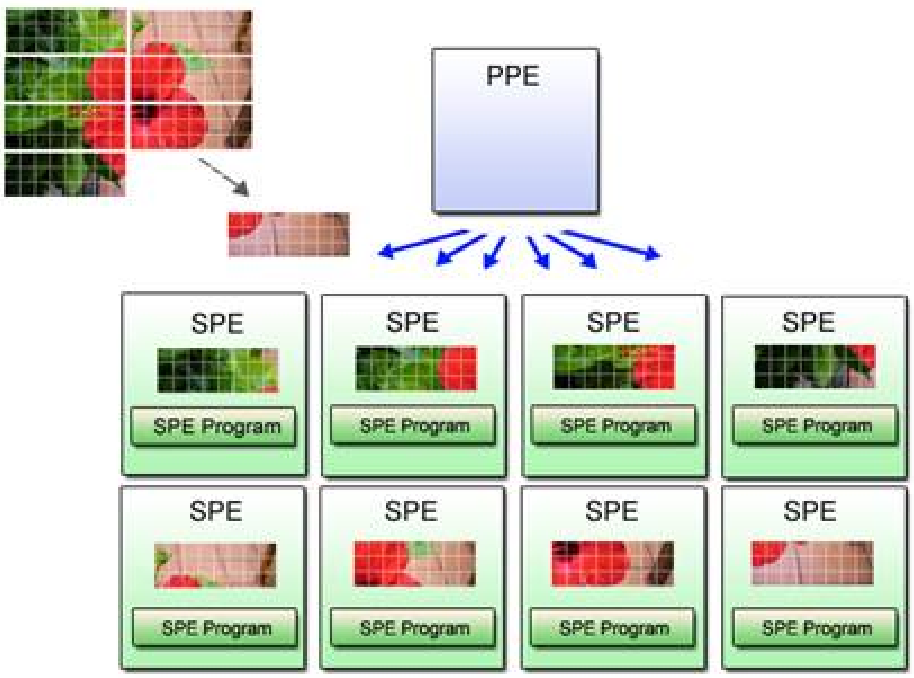
  Instruction Group Concurrency

  Simultaneous execution of several processes (instruction groups)

  e.g.) Several subroutines can be run at same time

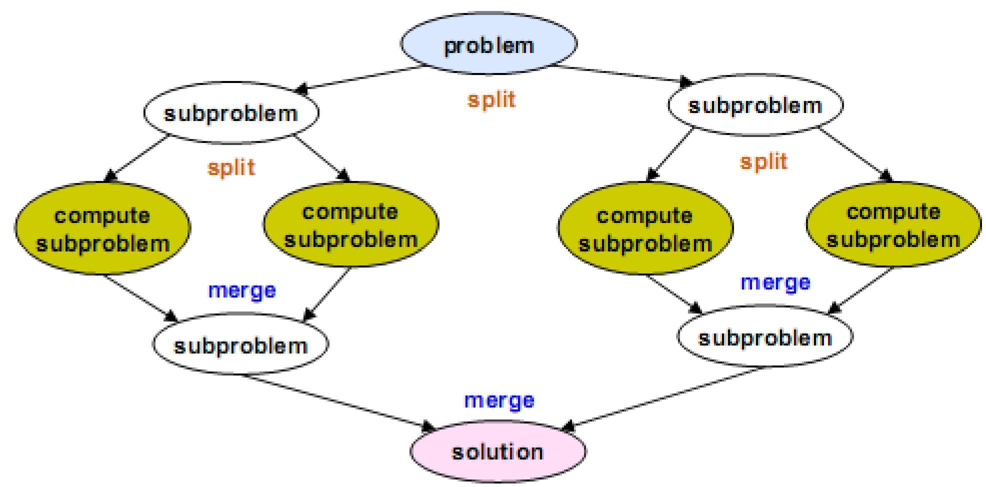
  Possible implementation: Multi-Processors, Multi-Computers

We will study two main types of Partitioning in SW Parallel Processing



Data Partitioning

Each PE does all needed tasks on a small piece of the Data Set

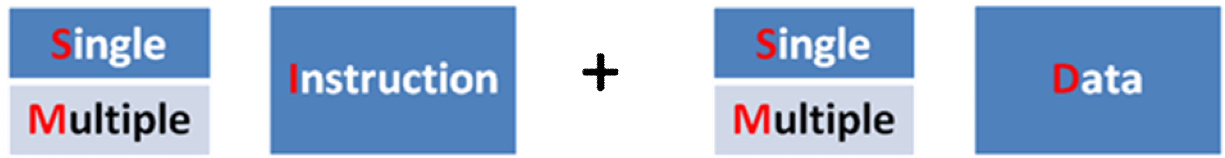


Task Partitioning

Each PE does just one specific task across all elements of entire Data Set

- **Classification of HW Parallel Processors**

  Flynn's 4 classes based on number of instructions and data handled

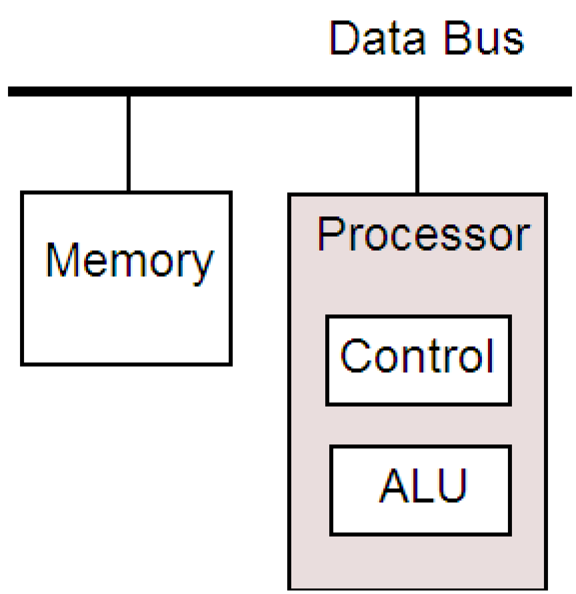


(1) SISD: Single Instruction, Single Data

One instruction applied to one piece of data Sequentially

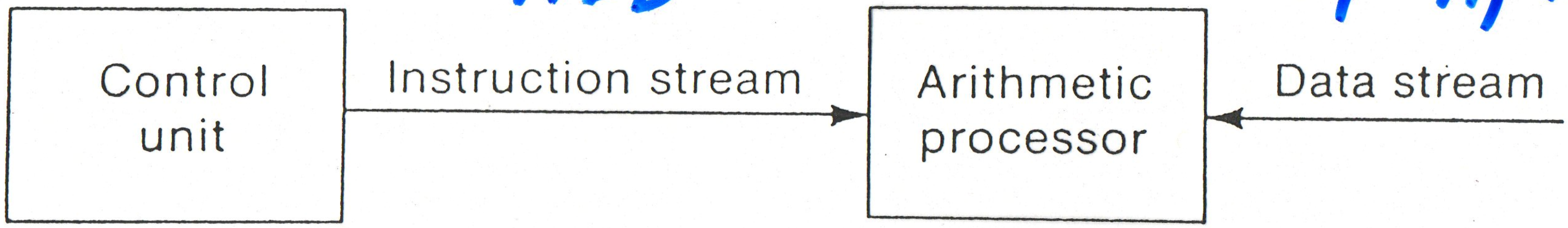
Single Processor, Control Unit, and Memory

Classic Von Neumann Architecture



Internal (microscopic) parallel processing could occur with pipelining

ADD $\qquad C_1 = A_1 + B_1$



a) Model of an SISD computer

```
      DO 20 I=1,100
20    C(I) = A(I) + B(I)
```

(2) SIMD: Single Instruction, Multiple Data

The same instruction is performed on many pieces of different data

Several Processors share a common Control Unit and Memory

All processors receive same instruction but operate on different data

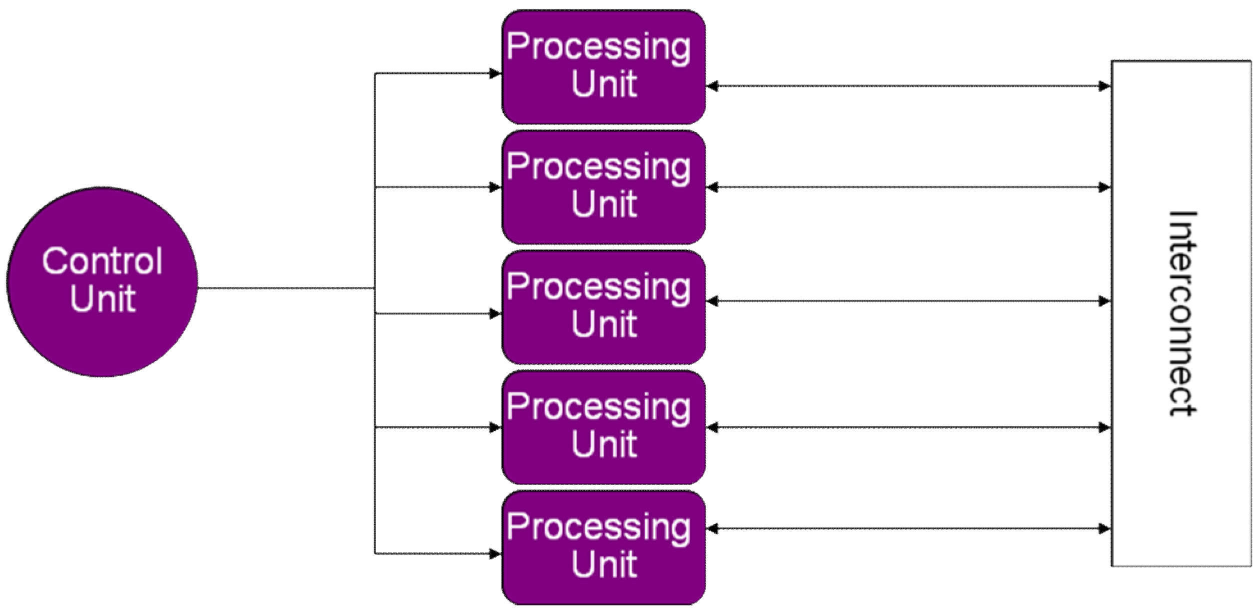
Processors are synchronized

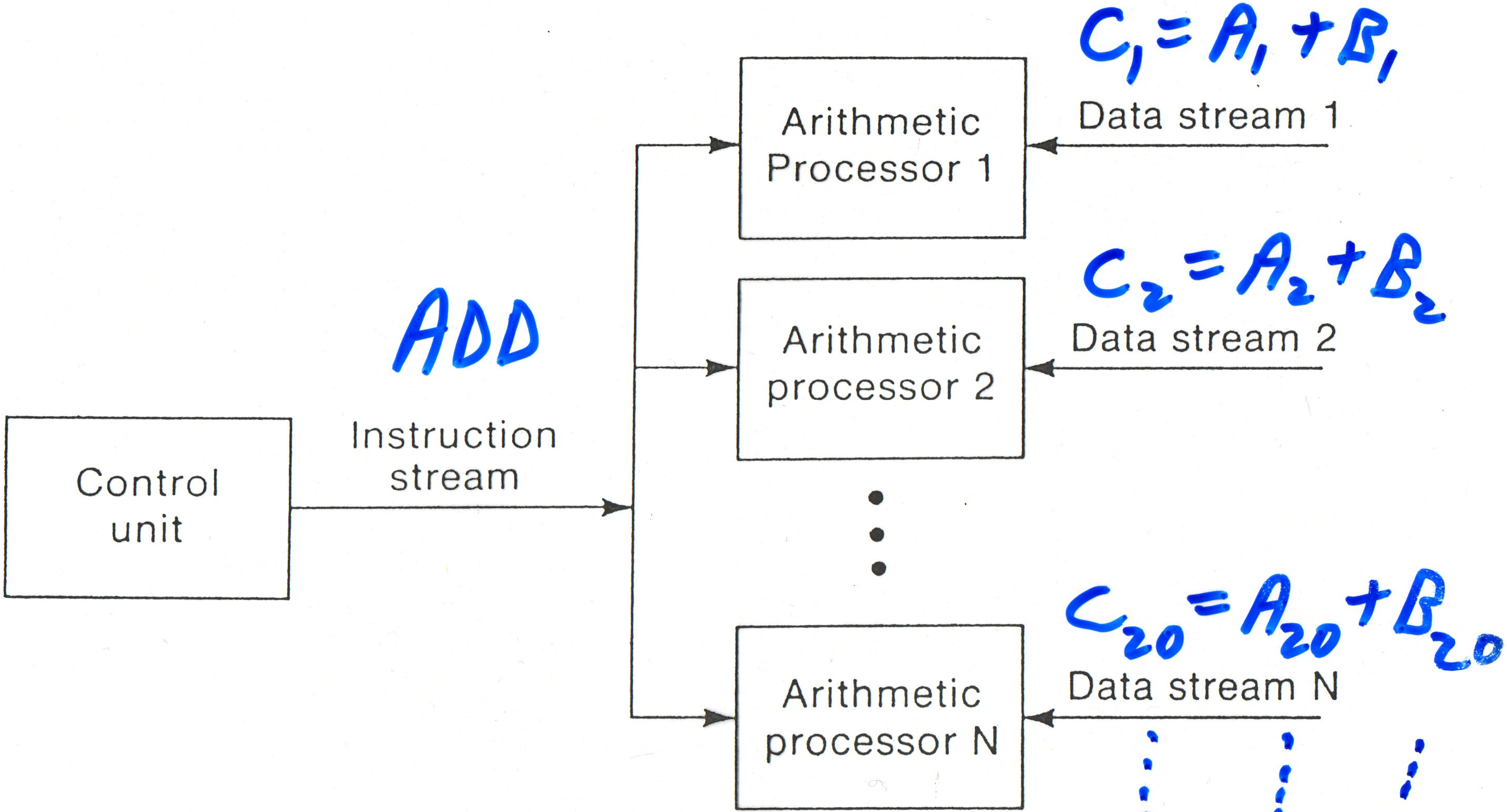
e.g.) Vector computer with 100 processors can do loop in one cycle:

```
      Do 20 I = 1, 100
 20   C(I) = A(I) + B(I)
```

Generally, programmer assists in the identification of parallelism:

e.g.) Above loop is coded as: C(1:100) = A(1:100) + B(1:100)

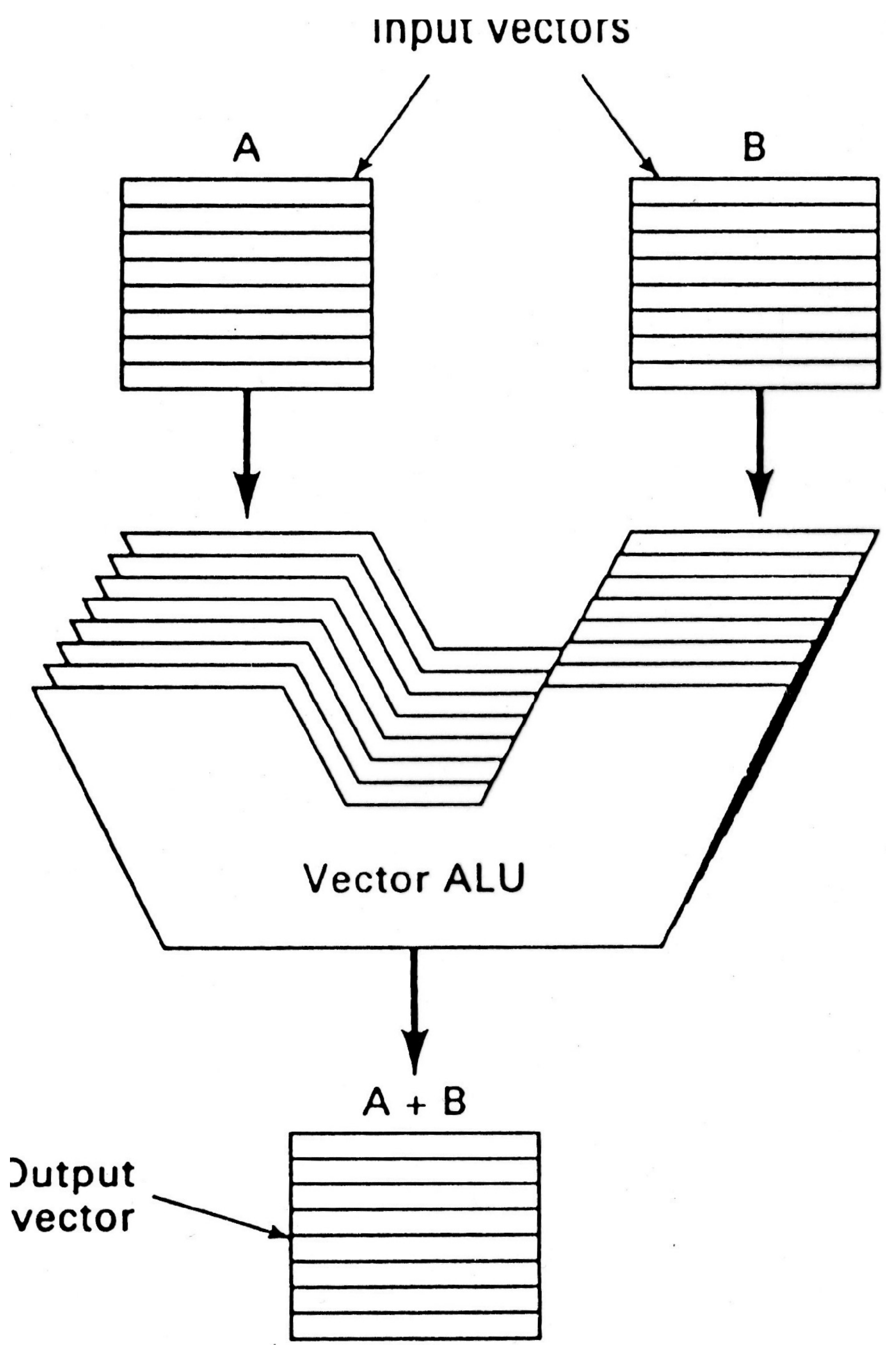




b) Model of an SIMD computer

$C_{100} = A_{100} + B_{100}$

$$C(1:100) = A(1:100) + B(1:100)$$



(a)

$$\begin{array}{ccccc} A & & B & & A + B \\ \begin{bmatrix} 4 \\ 6 \\ 3 \\ 2 \\ 7 \\ 8 \\ 3 \\ 8 \end{bmatrix} & + & \begin{bmatrix} 5 \\ 3 \\ 1 \\ 3 \\ 4 \\ 1 \\ 8 \\ 8 \end{bmatrix} & = & \begin{bmatrix} 9 \\ 9 \\ 4 \\ 5 \\ 11 \\ 9 \\ 11 \\ 16 \end{bmatrix} \end{array}$$

(b)

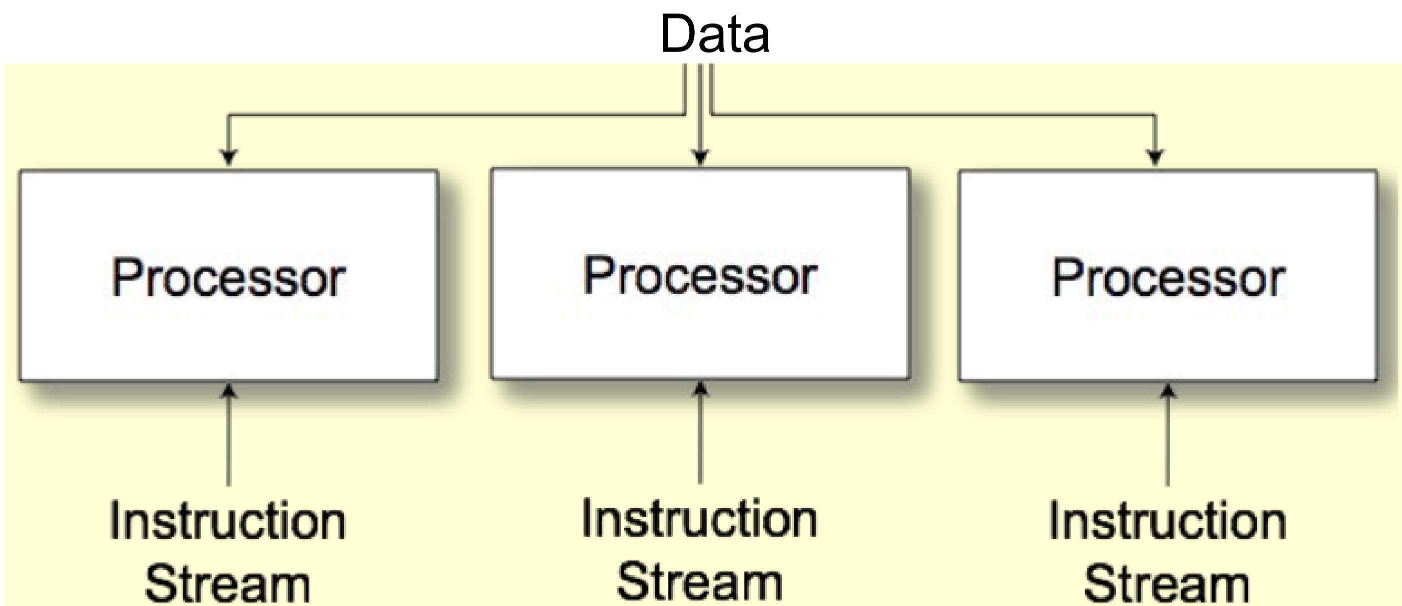
Fig. 2-6. (a) A vector ALU. (b) An example of vector addition.

(3) MISD: Multiple Instruction, Single Data

Traditionally, this has been considered a Non-sensible configuration

Why do different operations on the same data at the same time?

What application would need to + - * / on the same data in parallel?



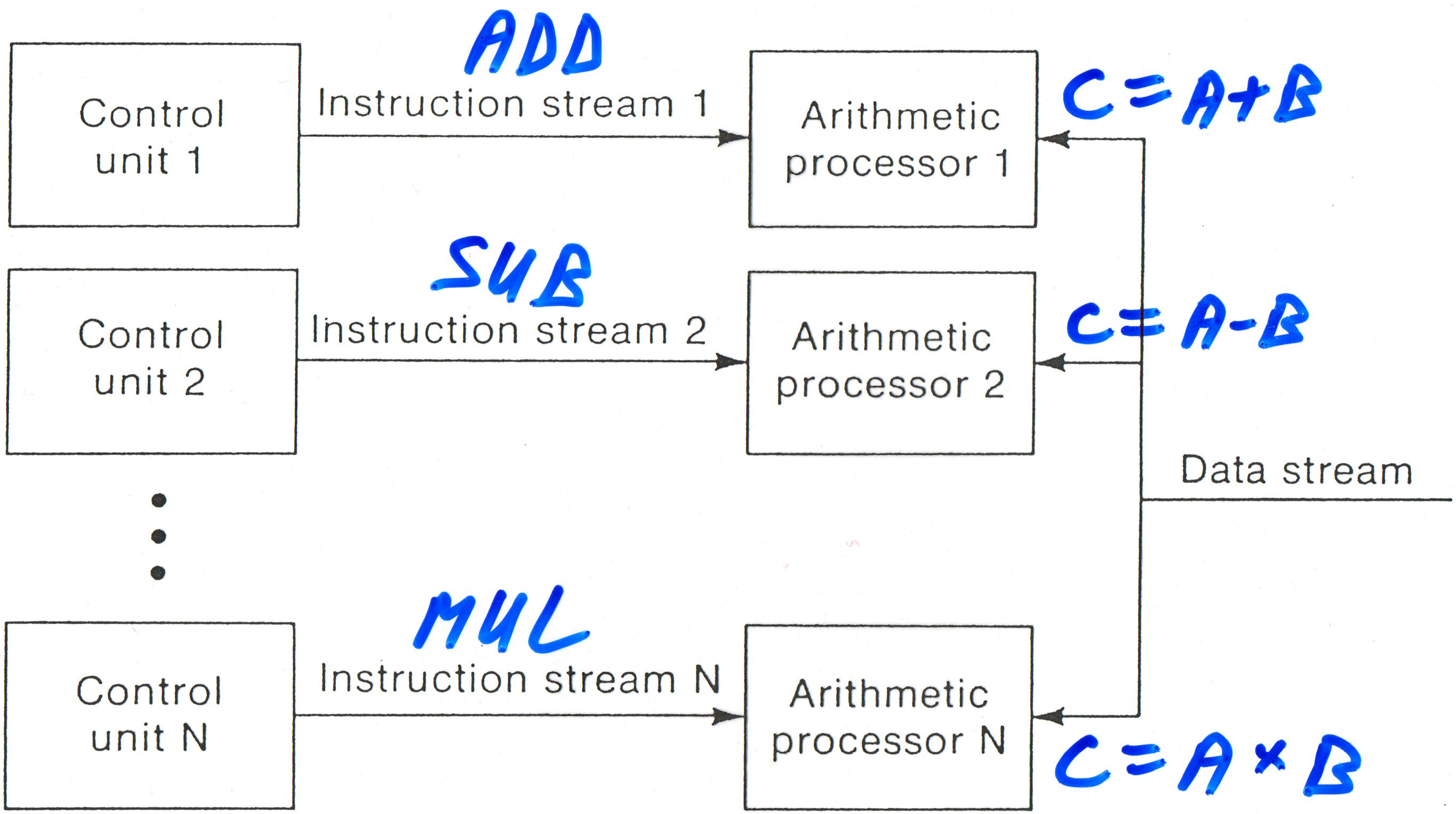
But perhaps it *can* be used in the context of Speculative Computation

Especially in a world when massive parallelism is (freely) available

Ex: Video Game awaiting one of four possible user inputs ⇦ ⇧ ⇩ ⇨

Could pre-compute all four alternatives on quad-core processor

Would be faster than waiting for input before computing result



c) Model of an MISD computer

IS THIS USEFUL ?

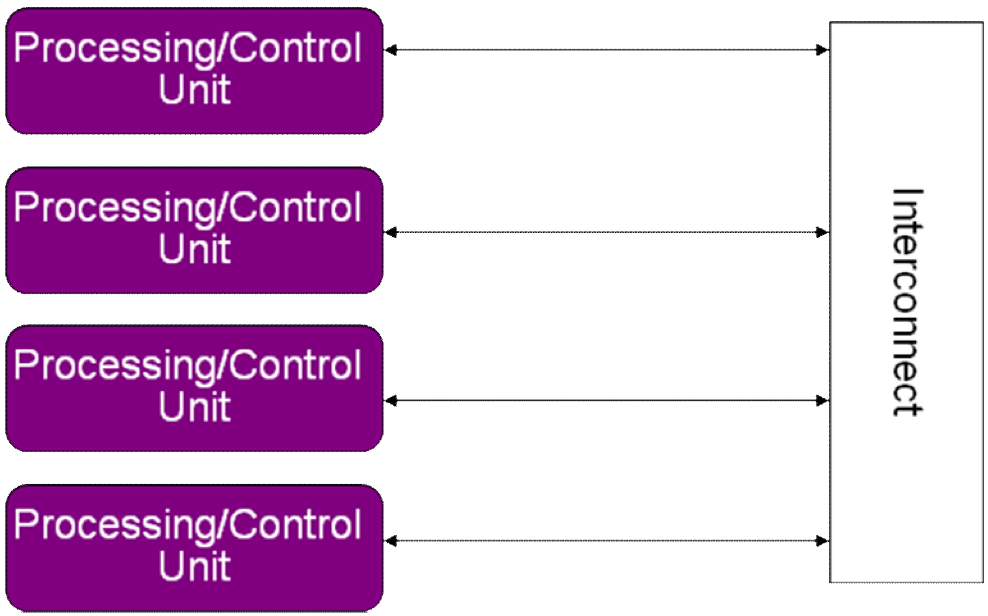
(4) MIMD: Multiple Instruction, Multiple Data

Many different instructions executed on many different sets of data

Several Processors with separate Control Units

Each processor:

- Runs its own instruction sequence
- Works on a different part of the problem
- Communicates data to other processors if necessary

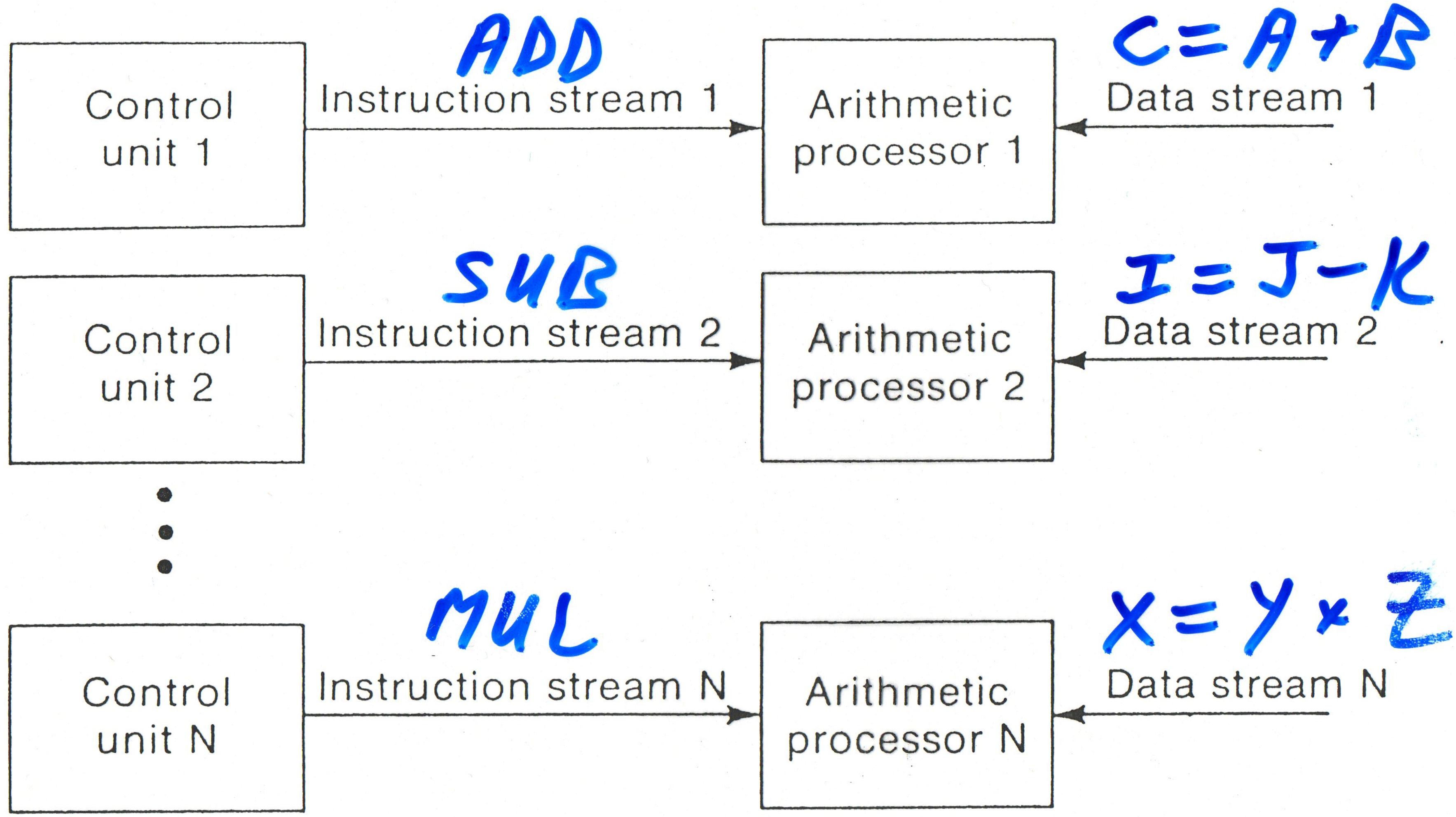


MIMD is the most relevant generic Arch. in the context of HW PP

Multi-, Many-Core and Parallel Programming in general is MIMD

Two Classic Variants of MIMD: Shared Memory & Distributed Memory

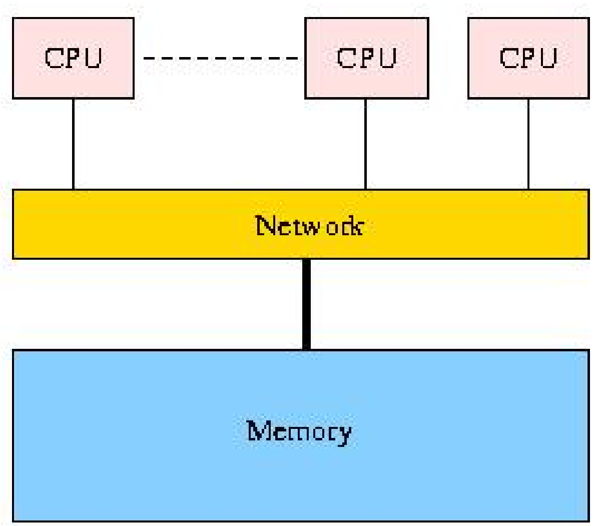
Leads to Two Classic SW Models: Shared Mem & Message Passing



d) Model of an MIMD computer

CAN PROCESS SEVERAL DIFFERENT INSTRUCTIONS ON DIFFERENT SETS OF DATA CONCURRENTLY.

- Shared Memory MIMD
    - All processors have direct access to all of the memory
    - Advantages:
        - Sharing memory for data, OS System Code, etc. reduces costs
        - Conceptually Easier for SW Processes to Communicate
    - Disadvantages:
        - Memory Contention can be severe when running shared code
            - Want collective Mem bandwidth to increase linearly with P's
            - However, this hard to achieve in practice
        - Interconnection HW between P's and M's scales badly as *O(n**2)*
            - Network becomes complex, costly and impractical as *n* grows



Shared Memory System

- Distributed Memory MIMD

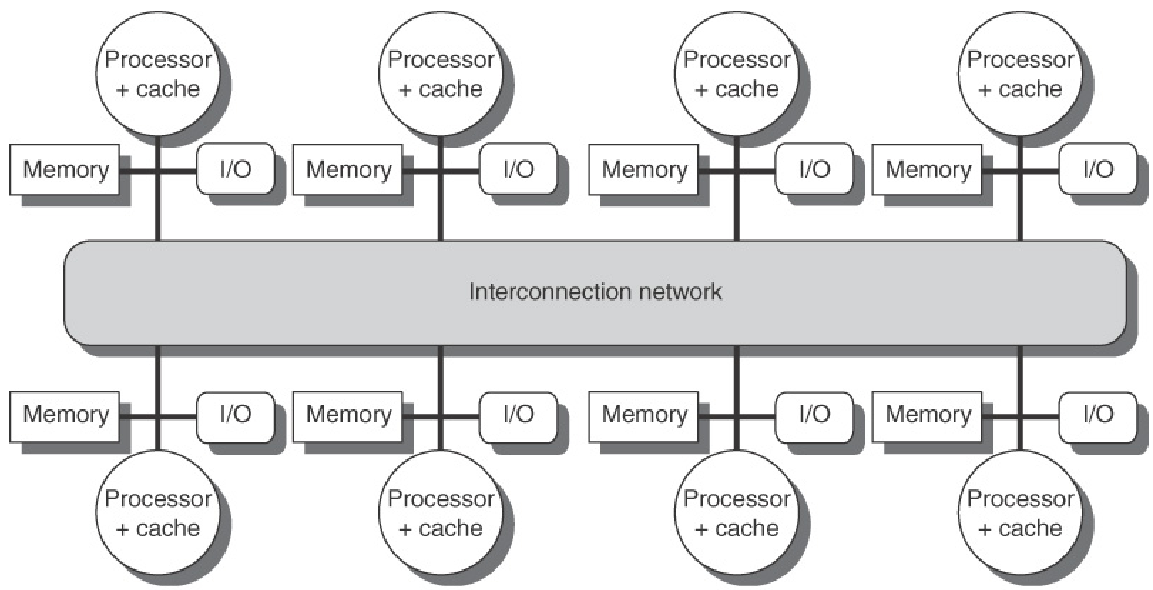
  Each processor has its own individual memory

  Advantages:

  - Memory Contention is reduced
  - HW Scaling is improved since each extra P adds its own M
  - Interconnection Network traffic reduced to message bursts

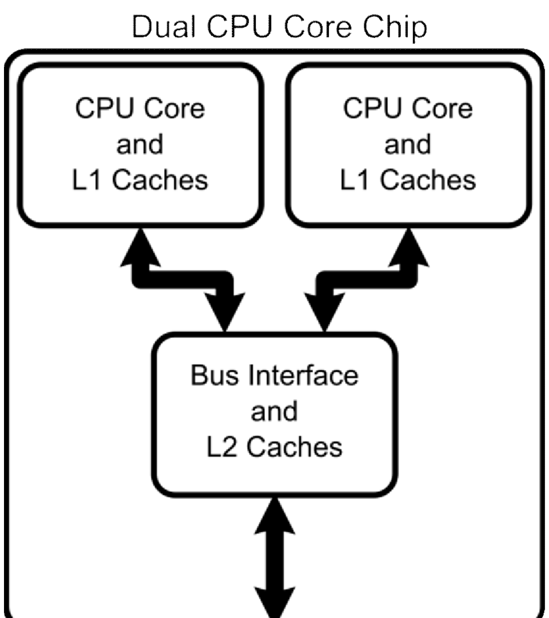
  Disadvantages:

  - For data to be shared, it must be passed from P-to-P as a Message
  - Processor Overhead to route messages can be substantial
  - SW program must arrange Message Passing, Synchronization, etc.

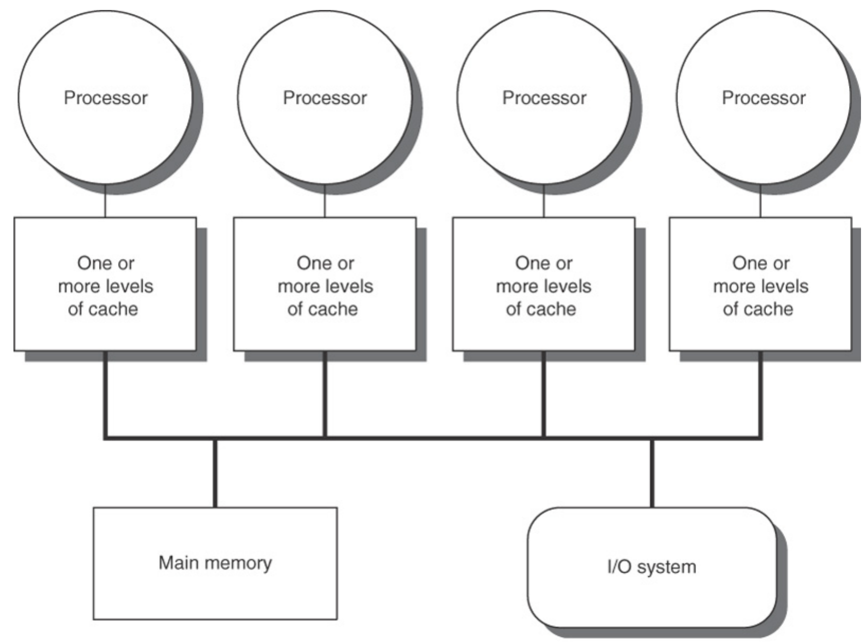


Distributed Memory System

- Combination Shared & Distributed Memory MIMD
  - In classic binary classification of Shared vs. Distributed MIMDs:
    - Shared Mem MIMD HW would use Shared Mem SW API
      - e.g.) OpenMP: Open Multi-Processing
    - Distributed Mem MIMD HW would use Message Passing SW API
      - e.g.) MPI: Message Passing Interface
  - However, all MIMD Architectures have a Combination of Mem Types
    - Definition of “Memory” should really include L1, L2… Cache Mem
      - Cache is typically physically close (and private) to each P
  - So Delineation (and Choice of best SW API) is not always clear



Micro-(Shared or Distributed Mem?)



Macro-(Shared or Distributed Mem?)

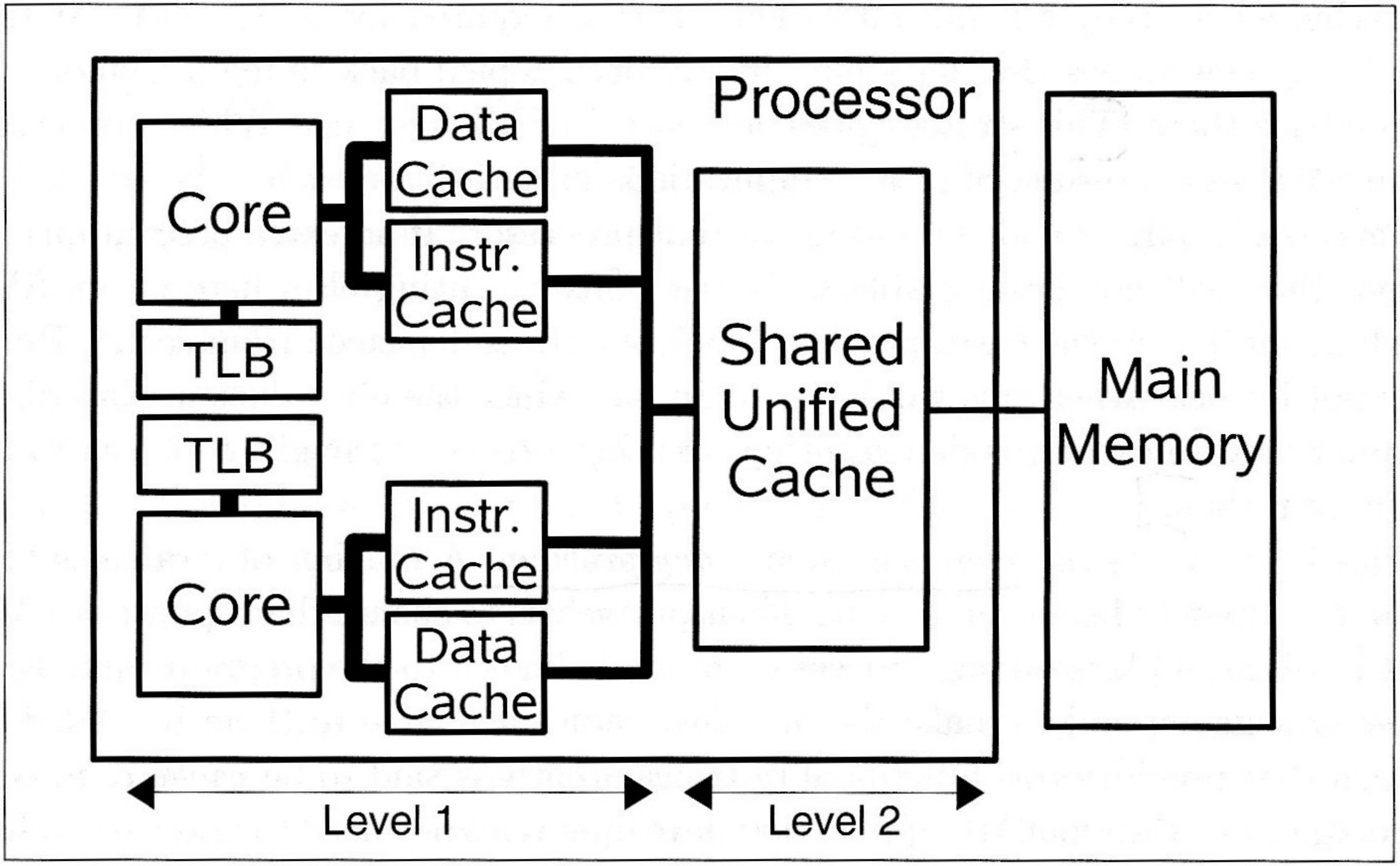


Figure 1.1: **Block diagram of a generic, cache-based dual core processor** – In this imaginary processor, there are two levels of cache. Those closest to the core are called "level 1." The higher the level, the farther away from the CPU (measured in access time) the cache is. The level-1 cache is private to the core, but the cache at the second level is shared. Both cores can use it to store and retrieve instructions, as well as data.

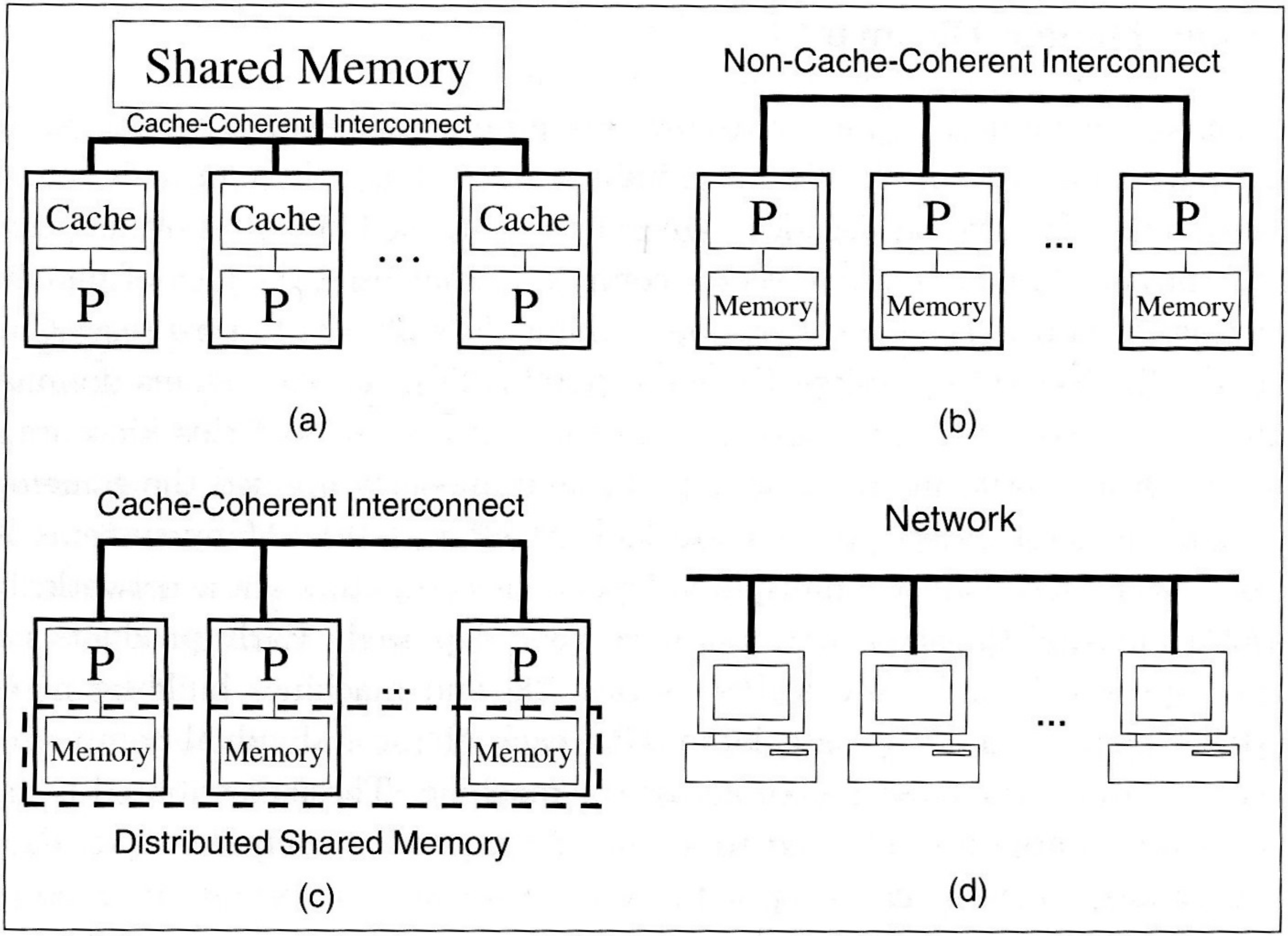


Figure 1.2: **Distributed- and shared-memory computers** – The machine (a) has physically shared memory, whereas the others have distributed memory. Howeve the memory in (c) is accessible to all processors.

- We will discuss two Fast Processor Techniques:
  - Pipelining: Microscopic Concurrency
    - Decompose the Instruction into a sequence of subprocesses
  - Vector / Multiple Function Unit Processors: Macroscopic Concurrency
    - Have several functional units, each performing a different instruction

- Pipelining:
  - Key implementation technique used to make fast CPUs.
  - Enables multiple instructions to be overlapped in execution.
  - Exploits parallelism among the instructions in a sequential instruction stream
  - Key Advantage: Transparent to Programmer
    - Preserves Simplicity of Sequential Model to Software World (OS + App)

- Analogy: Assembly line
  - Work to be done in an instruction is broken into smaller pieces
  - Each piece takes a fraction of the time needed for the entire instruction
  - A pipeline is partitioned into stages or segments
  - Each stage in the pipeline completes a part of the instruction
  - Pipe stages are hooked together, so all stages must operate in lock-step
  - Time required per step of the pipeline is determined by the slowest pipe stage

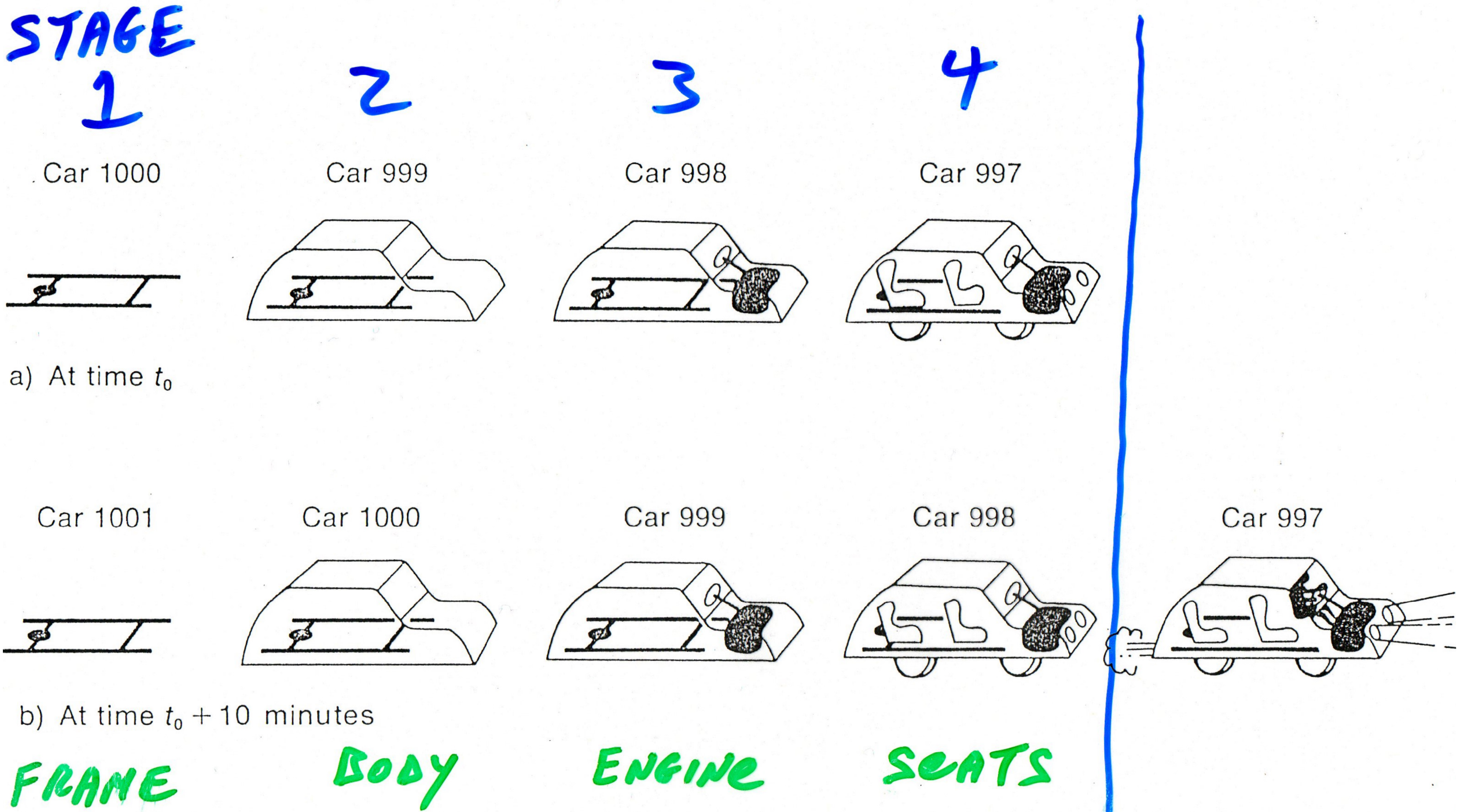
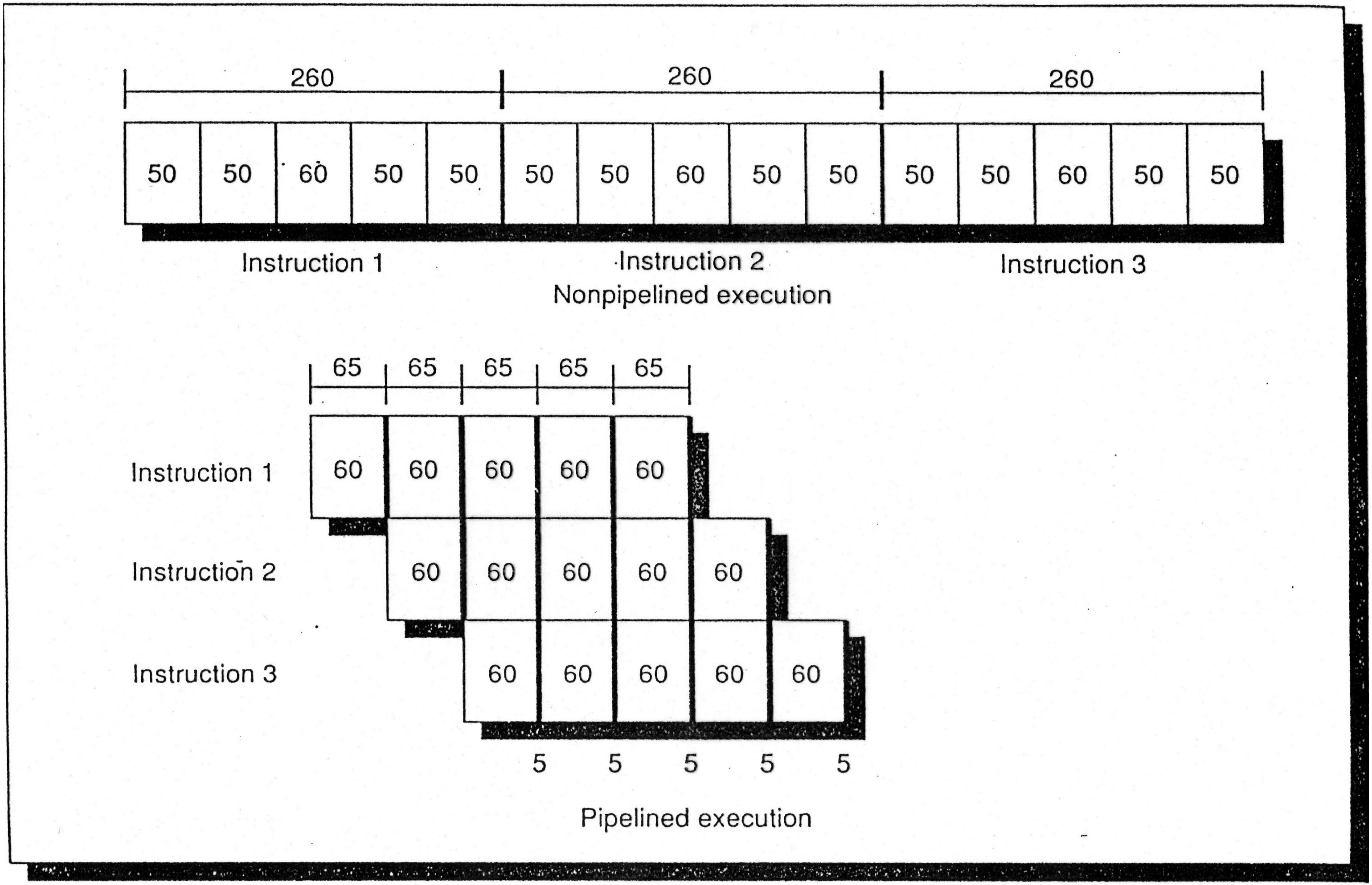
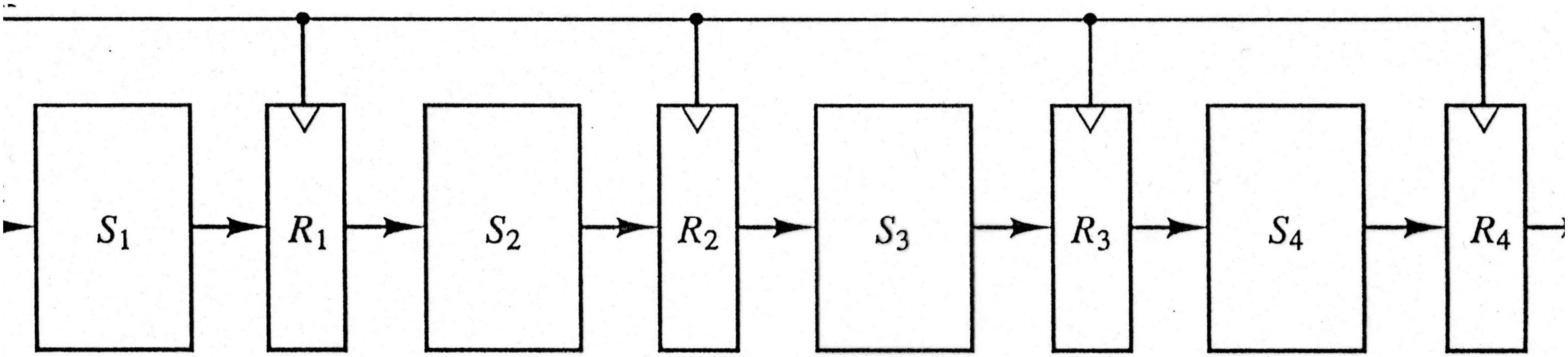


Figure 9-1 Automobile frame assembly



260
260
260
50
50
60
50
50
50
50
60
50
50
50
50
60
50
50
Instruction 1
Instruction 2
Instruction 3
Nonpipelined execution
65
65
65
65
65
Instruction 1
60
60
60
60
60
Instruction 2
60
60
60
60
60
Instruction 3
60
60
60
60
60
5
5
5
5
5
Pipelined execution

- **Pipeline designer's goal:**
  - **Balance the length of each of the pipeline's stages**
  - **Reduce stalls (caused by hazards)**
- **Throughput:**
  - **Determined by how often an instruction exits the pipeline**
  - **Number of instructions output per clock cycle**
  - **For an instruction stream consisting of N instructions:**
    - **Each instruction is divided into K equal segments (stages)**
    - **A non-pipelined machine would need: NK time steps**
    - **A pipelined machine would need:**
      - **K time steps for the first instruction (assuming pipeline was empty)**
      - **One time step for each of the remaining (N-1) instructions**
      - **Total = K + (N - 1) time steps**
    - **Speedup = NK / (K + N - 1)**
  - **Speedup from pipelining (asymptotically) equals the number of pipe stages**
    - **For N >> (K - 1), denominator approaches N**
    - **So, Speedup = NK / N = K**
  - **If stages are perfectly balanced, and no stalls occur (ideal conditions),**
    - **Throughput(pipelined) = Throughput(non-pipelined) x Num_Pipe_Stages**



- **Interesting Note:**
  - **Although pipelining increases the overall system throughput,**
    - **The total time needed by each instruction remains the same.**
    - **e.g.) For a five stage pipeline, each instruction still takes five clock cycles**
      - **On each clock cycle:**
        - **Hardware is executing some part of five different instructions**
        - **An instruction is completed and exits the pipe, and another enters**
  - **In fact, Total time needed for each instruction may actually increase !**
    - **Increase is due to the Overhead needed to control the pipeline**
    - **Latches are required b/w pipe stages, adding setup and propagation time**
      - **Latches separate the stages from each other**
  - **The increase in instruction throughput means that a program runs faster,**
    - **even though no single instruction runs faster.**

- Pipelining can be implemented in various places in the hardware:
  - Combination of techniques in various HW components is typically used
    - Memory: Interleaved memory banks
      - Partition Memory cycle time into access + wait
      - Volatile Memory (e.g. RAM) needs to Refresh on a regular basis
        - Refresh causes a no-access period, the wait period
      - If no interleaving, then no access can occur during wait periods
      - Use (at least) a pair of Memory Banks and alternate between them
        - Overlap M2 access phase with M1 wait phase
    - Arithmetic Logic Unit: Math operations are phased
      - Many alternatives are possible
        - e.g.) Floating Point Addition can be partitioned into the following steps:
          1) Compare Exponents
          2) Align Mantissas
          3) Add / Subtract Mantissas
          4) Normalize Result

- Control Unit: Instruction fetch, decode, execute

Although different partitions and granularities are possible, the Basic Steps of Instruction Execution are:

1) Fetch Instruction (FI)

Read Program Counter and fetch instruction from memory

This is primarily a Memory Read Stage

2) Decode Instruction (DA)

Decode the bits to determine the operation and operands needed

At end of this phase, the instruction and location of its ops is known

3) Fetch Operand(s) (FO)

Read the operand(s) needed for the operation from memory

This is primarily a Memory Read Stage

4) Execute (EX)

Execute the operation and Write Back the result to memory

This stage is sometimes called WB, or is broken into two EX + WB

- **Hazards:**

  Pipelining changes the relative timing between any two instructions

    Some overlap in their execution; some phases should not be overlapped

  Overlap introduces Hazards due to interaction between instructions that:

    Prevent next instruction from executing during its designated clock cycle

    Reduce the pipeline's performance from the ideal speedup possible

  - Three Types of Hazards:

  1) Data hazards:

    Instruction cannot be performed until operands are available.

    This can arise when an instruction depends on the results of a previous instruction in a way that is exposed by their overlapping in the pipeline

  2) Control hazards:

    The Next instruction to be executed is determined by the previous

    Arises from the pipelining of branches and other "decision-point" instruct.

      Can change the program counter from its normal "+1" increment

  3) Resource hazards:

    Instruction cannot be performed until resources are available

    Arises when not enough of the right kind of hardware is available

  Hazards can make it necessary to stall the pipeline

- **Stalls:**

  Major differences b/w stalls in a pipelined machine vs. a non-pipelined one

  - Non-Pipelined:

  - **Only one instruction can be executing at any one time**
    - **An Instruction cannot be stopped mid-stream**
    - **Once started, an instruction will complete**
  - **Just freeze program counter and halt after current instruction completes**

  - Pipelined:

  - **Multiple instructions are being executed (in different phases) in parallel**
    - **Several Instructions are "In-Flight" at the same time**
    - **Each is in a different phase of execution**
  - **Stall will allow some instructions to proceed to their next phase(s),**
    - **Other instructions are delayed (frozen at their current phase)**

  Typically, when an instruction is stalled in a Pipelined System:

  - **Instructions earlier than it can continue**
  - **All instructions later in the pipeline than it (behind it) are also stalled**
  - **No new instructions are fetched or enter the pipeline during the stall**

- **Data Hazards:**

  Occur when the order of access to operands is changed by the pipeline

  (vs. the normal order encountered by sequentially executing instructions)

  In non-pipelined Seq. HW, only one instruct at a time is touching operands

  Two instructions can create a hazard by writing and reading the same variable

  - Example: Assembly code reading from and writing to Registers

  Output = INS Op1, Op2

  R1 = MUL R2, R3

  R4 = ADD R1, R5

  R8 = SUB R6, R7

  The ADD instruction has a source, R1, that is the destination of the MUL

  It is possible that MUL does not write R1 until after ADD reads R1

  ADD starts Operand Fetch (FO) before MUL completes Execute (EX) step

  Pipeline overlap has exposed a Data "Race" Condition on R1

  Unless precautions are taken, ADD will use an old value of R1

  Non-deterministic behavior could also result:

  e.g.) If an interrupt occurs b/w MUL and ADD, then ADD <u>will</u> get the new R1

- **The most common solution to this problem is a hardware pipeline interlock**

  Interlock detects a hazard and stalls the pipeline until the hazard is cleared

  Pipeline is stalled beginning with the instruction that wants to use the data until the earlier sourcing instruction completes and produces it.

  In above example: ADD and following instructions are stalled until MUL writes

  This delay cycle (pipeline stall) creates a "bubble" in the timing diagram

- **Example of Data Hazard Causing Pipeline Stall**

S1: X = X + 1

S2: Z = X + Y **[S2 has a data hazard on X]**

S3: A = B + C

S4: J = K + L

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
S1	FI	DA	FO	EX						
S2		FI	DA	--	FO	EX				
S3			FI	--	DA	FO	EX			
S4				--	FI	DA	FO	EX		

- **Example of Two Data Hazards Causing Two Pipeline Stalls**

S1: X = X + 1

S2: Z = X + Y [S2 has a data hazard on X from S1]

S3: A = Z + B [S3 has a data hazard on Z from S2]

S4: J = K + L

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
S1	FI	DA	FO	EX						
S2		FI	DA	--	FO	EX				
S3			FI	--	DA	--	FO	EX		
S4				--	FI	--	DA	FO	EX	

- **HW-based Pipelining is done "automatically" and Transparently to the SW**
  Extra HW is needed for Detection and Prevention (via stalls) of Data Hazards
  Extra HW also needed between Pipeline Stages

- **Very Long Pipelines (with a large number of stages) can be designed**
  Results in Large K, so Throughput is increased (multiplied by) Large K
  But, Probability of a Data Hazard Increases with Number of Instrucs "In Flight"

- **Minimizing Impact of Data Hazards**
  - **Pipeline stalls represent lost computing cycles (essentially a No-Op)**
  - **Compiler can assist by trying to schedule the pipeline to avoid these stalls**
  - **Code sequence is rearranged to eliminate (or at least reduce) the hazard**
    - **Example:**

R1 = MUL R2, R3		**R1 = MUL R2, R3**
R4 = ADD R1, R5	**Rearranged to:**	**R8 = SUB R6, R7**
R8 = SUB R6, R7		**R4 = ADD R1, R5**

  - **A Smart Compiler can rearrange Instructions to Reduce Stalls from Data Hzds**
    - **But not all data dependencies can be eliminated**
      - **Typically, programmer writes to X because updated X will be read soon**
    - **And a "Smart" Complier would also be a Slow and Expensive Compiler**
      - **Compiler is essentially a sequence of embedded "Case" Statements**
        - **IF <source code> THEN <generate ASM code>**
      - **Optimizing Compiler would need to be Machine Savvy and HW Specific**
        - **Compiler writer would need to invest more time in Learning HW**
      - **Resulting Compiler may not be cost-effective and competitive**
      - **Case in Point: Itanimum-64 EPIC Very Long Instruction Word Arch.**

- **Control Hazards: Caused primarily by Conditional Branch Instructions**
  - **Can Change the Normal Contiguous, Sequential Instruction Stream Flow**
    - **Normally, Program Counter (PC) is just Incremented by 1**
      - **Location of Next Instruction is just after Current Instruction's location**
  - **Conditional Branch causes a delay in knowing which instruction is next**
    - **Test Condition needs to complete (EX) before Result of PC is Known**
  - **Easiest (most conservative and safest) solution is a Pipeline Flush**
    - **Penalty of Control Hazard > Penalty of Data Hazard**

- **Example 1 of Control Hazard Using Pipeline Flush Solution (Assume X = 0)**

S1: X = X + 1
S2: IF X > 10 THEN GOTO S4 [Data Hazard on S1]
S3: A = B + C [Control hazard on S2]
S4: J = K + L [Control hazard on S2]

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
S1	FI	DA	FO	EX							
S2		FI	DA	--	FO	EX					
S3			FI	--	--	--	FI	DA	FO	EX	
S4				--	--	--		FI	DA	FO	EX

- **Example 2 of Combined Data and Control Hazards (Assume X = 100)**

  Pipeline is Stalled on Data Hazard; Pipeline is Flushed on Control Hazard

S1:	X = X + 1	
S2:	IF X > 10 THEN GOTO S4	[Data Hazard on S1]
S3:	A = B + C	[Control hazard on S2]
S4:	J = K + L	[Control hazard on S2]

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
S1	FI	DA	FO	EX							
S2		FI	DA	--	FO	EX					
S3			FI	--	--	--					
S4				--	--	--	FI	DA	FO	EX	

Note: In Example 2, S3 is not executed and Flush clears the pipe for entry of S4

Conservative Flush remedy saves time in the end

In Example 1 (where X = 0), Flush of S3 requires S3 to be FI again

Flush purged the FI stage of the instruction that really would have run

Both Cases incur an expensive Pipeline Startup (reload) Penalty

- **Minimizing Impact of Control Hazards**

  Jumps can be classified into three categories:

  1) Unconditional

  2) Conditional

  3) Loop

  Amount of Stalling due to Branches can be Reduced in various ways:

  - Just assume the jump will not be taken and continue filling pipeline

  Using this technique in Example 1 above would save the FI of S3

  Requires (a delayed) pipeline flush if jump really is taken

  Flush would have been "cheaper" if performed ahead of time

  because the pipeline would have been stalled anyway

  Also, need to undo any pre-executed effects of (wrong) instruction

  Overhead can be Reduced through Buffering before "Commit"

  - "Guessing" Statically (During Compile Time) which path will be taken

  Attach an extra bit onto each branch instruct. that is set by compiler

  Bit serves as a "hint" for the HW as to most likely branch direction

  The Bit (the Prediction) is not modified during program execution

Stereotypical Behaviors can be identified for certain branch types

Jumps at end of Loops are a special case of conditional Jumps

Loop conditional jumps are almost always taken (Set Hint Bit = 1)

- Example:

Loop: X(I) = Y(I)

I = I + 1

If (I < 10) GoTo *Loop*

Jumps to System Error Routines will almost never be taken (Bit=0)

IF A = 0 THEN GoTo *Sys_Error*("Div_by_0")

C = D / A

- Prefetching on both paths of a branch

Requires 2 pipelines in the HW for parallel execution along both paths

Complicates control structure of pipeline

Used only on highest speed, highest cost machines

Half the answers computed will eventually be discarded

Requires Buffering before "Commit" to Output Destinations

VLIW Architectures often fetch both paths speculatively

- Compiler (Re)-Scheduling of Instructions (Delayed Branching)
  - Splits conditional jump into test (IF) and action (THEN) part
  - Inserts useful instructions instead of no-op stalls b/w IF and THEN
    - These instructions would be done regardless of branch outcome
  - Location following a branch instruction is called a Branch Delay Slot
    - Instructions in the delay slots are always fetched and executed
    - Change in ordering should be SW Transparent
      - Needs to be Independent of Data and of the Branch is taken
  - Example:

X = Y + Z	IF B < C
IF B < C	X = Y + Z
THEN A = B + C	THEN A = B + C

- Performance of Different Control Branch Handling Schemes

  Assume a 5 stage pipeline with maximum speedup of 5X if no Stalls

Scheduling Scheme	*Pipelined Speedup over Non-Pipelined*
Stall Pipeline	3.5
Predict Taken/Not Taken	4.4
Compiler rescheduling	4.6

- **Limitations of Pipelining**

  Pipeline speedup potential is limited by the number of pipeline stages (K)

    Throughput(pipelined) = Throughput(non-pipelined) x Num_Pipe_Stages

  Number of stages is limited by the total number of separate "functions" into which the instruction can be decomposed into (typically 4-8).

  In Extreme Limiting Case (K>>), Latch Overhead > Work Done per Stage

  Pipeline speedup potential is also limited by Instruction Stride

    Stride is an unbroken consecutive string of instructions through pipeline

    Once stride is broken (e.g. by a branch flush), pipeline needs to reload

    Pipeline startup penalty is higher for smaller N and larger K

    If only a small number of instrucs are processed consecutively, N is small

      So, cannot assume N >> (K - 1) and that Speedup = NK / (K + N -1) = K

      For example, for N = 5 instructions, and a K = 8 stage pipeline:

        Speedup = (5) (8) / 12 = 3.3   which is significantly < 8

    Probability for long consecutive strings highest in matrix math computation

  In Worst Case Scenario of High Latch Overhead (K>>) and Poor Stride, Pipelining can result in a net overall Performance *D*egradation

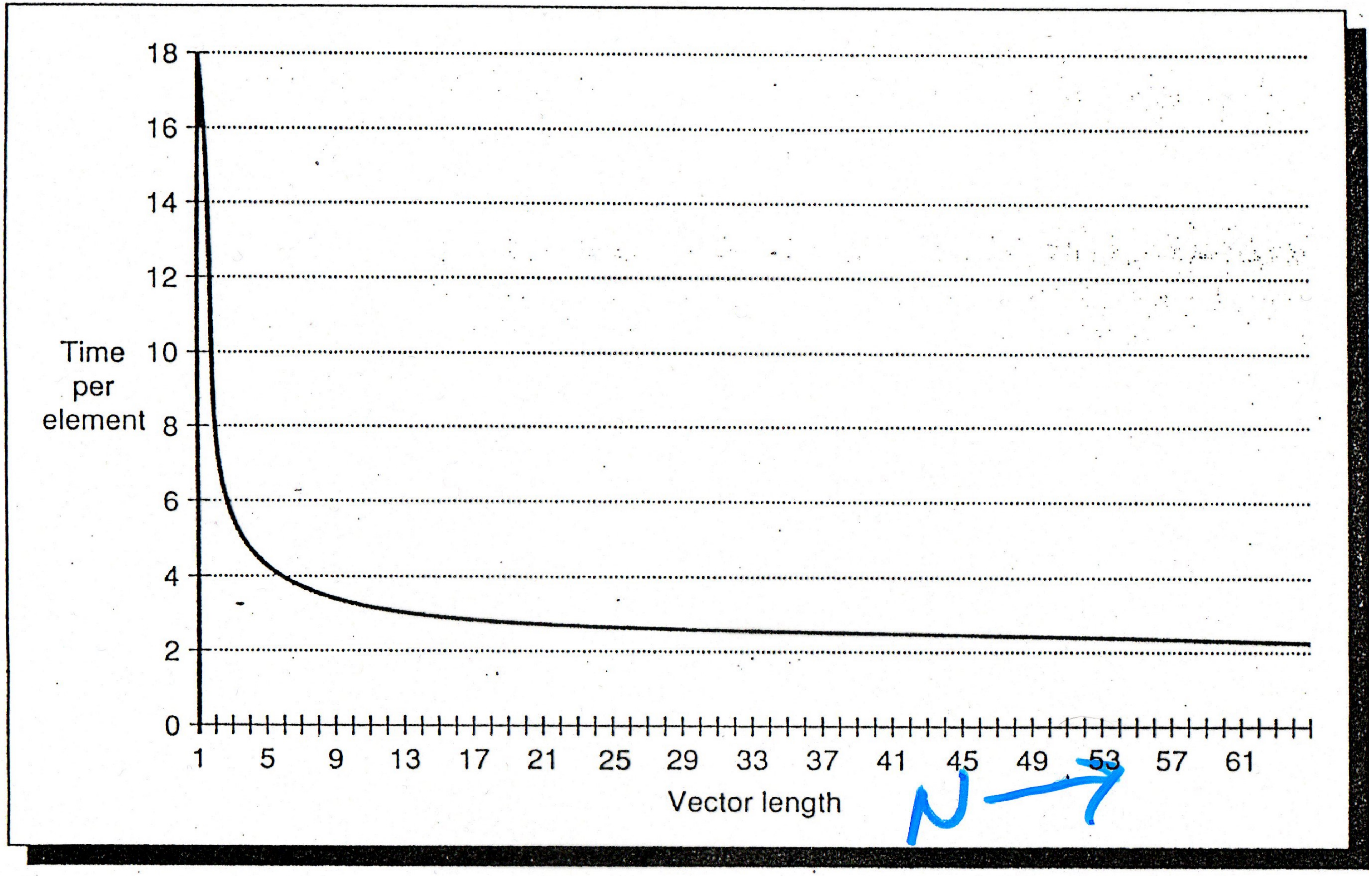


FIGURE 7.9 The impact of just the vector start-up cost on a loop consisting of a vector assignment. For short vectors, the impact of the 16-cycle start-up cost is enormous, decreasing performance by up to nine times. The strip-mining overhead has not

- **Macroscopic Instruction Parallelism**
  - **Utilize Parallel HW to Process several (whole) instructions in Parallel**
    - **Not just overlapped in phases as in Pipelining**
  - **Can actually change the order of instructs relative to how they appear in prog.**
    - **Successive Instructions in Code actually execute in Parallel at same time**
      - **In Pipelining, later instruction is still “later” in order of HW execution**
        - **Pipelining preserves ordering, even if just by one stage later**
    - **In Aggressive Form, Later instruction might even go before an Earlier one**
  - **More complicated than Pipelining (Microscopic Instruction Parallelism)**
    - **Requires the use of Multiple Function Units in HW**
      - **At any one time, many instructions may be in their execute stage**
        - **Instructions are not just Phased as in pipelining, but run in Parallel**
    - **Requires Resource Hazard Analysis**
      - **Need to Check if Proper Function Unit is available for Parallel execution**
    - **Requires more comprehensive Data Dependency analysis**
      - **In Aggressive Instruction Re-ordering, more Data Hazards are possible**
        - **More Variations (Types) of Data Hazards can arise**
      - **Data Hazard analysis in pipelining is simpler (Instructs only Overlapped)**
  - **Can be combined with Pipelining (Each Function Unit itself is Pipelined)**

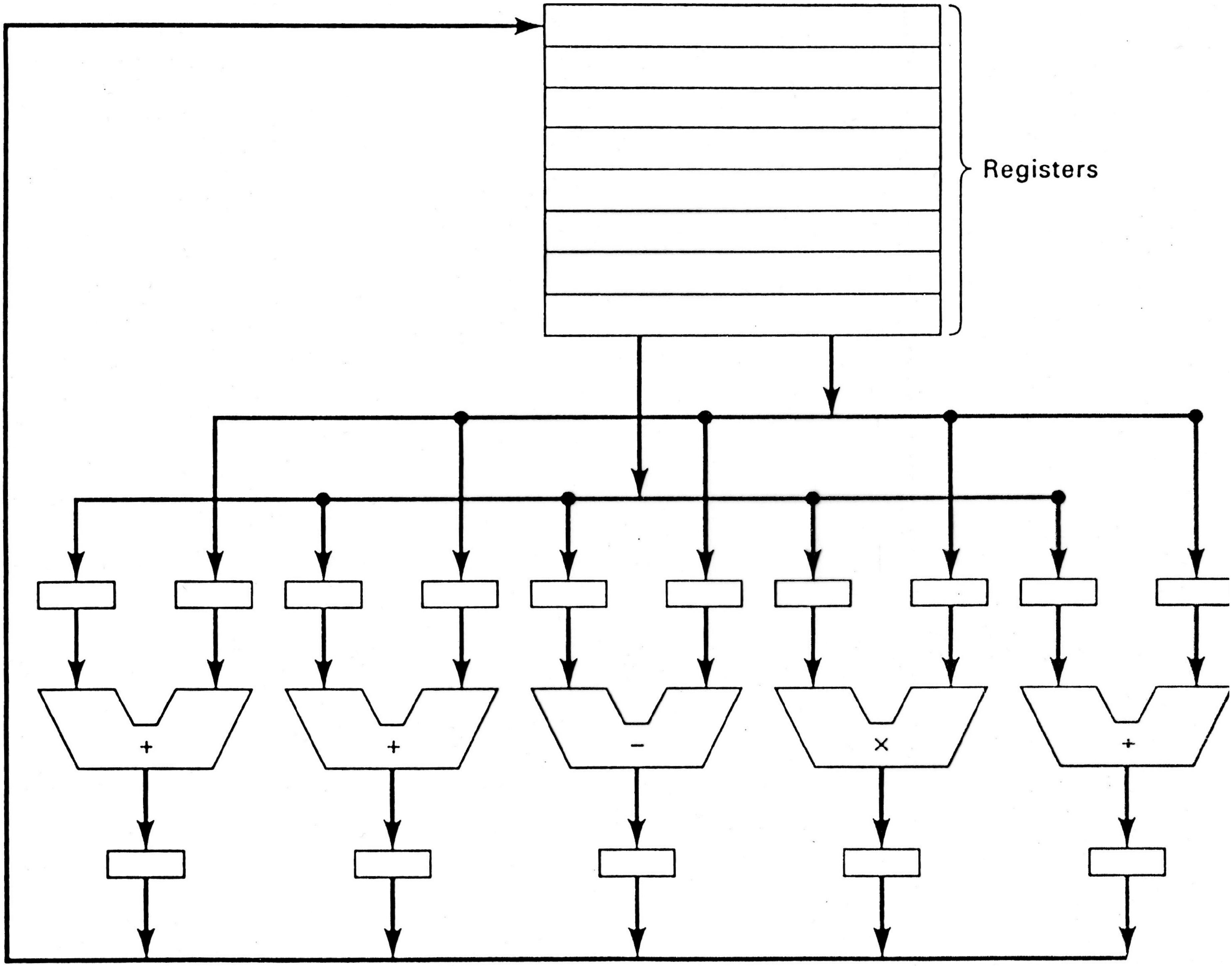


Fig. 2-4. A CPU with six functional units that can run in parallel.

- **Concurrent Execution of Sequential Algorithms:**
  - Procedural SW Programming Languages inject an "apparent" Sequentiality
    - One instruction must "apparently" be completed before next is started
  - This "apparent" Seq. SW view leaves room for some hidden concurrency
  - Goal: Obtain faster execution while retaining simplicity of sequential rep.
  - Approach: Remove any unnecessary sequentiality from the SW program

  - A Sequential Algorithm has:
    - \- Inherent Sequentiality
      - An ordering of operations which are an implicit part of the algorithm
      - These must be preserved as a fundamental part of the SW program
      - Changing the order of these instructions will alter what was intended
    - \- Artificial Sequentiality
      - Injected by the semantics of the SW specification of an algorithm
      - Most languages do not enable programmer to specify concurrency
      - Temporary variables contribute to sequential step appearance

  - By Eliminating Artificial Sequentialities, Execution can be Accelerated
    - Continues to preserve the required dependencies for correct behavior
    - Maintains "apparent" sequentiality while transparently using parallelism
    - Identifying inherent sequentiality requires more detailed hazard analysis

- Separating Inherent Sequentiality from Artificial Sequentiality

Example using a Data Dependency Flow Graph

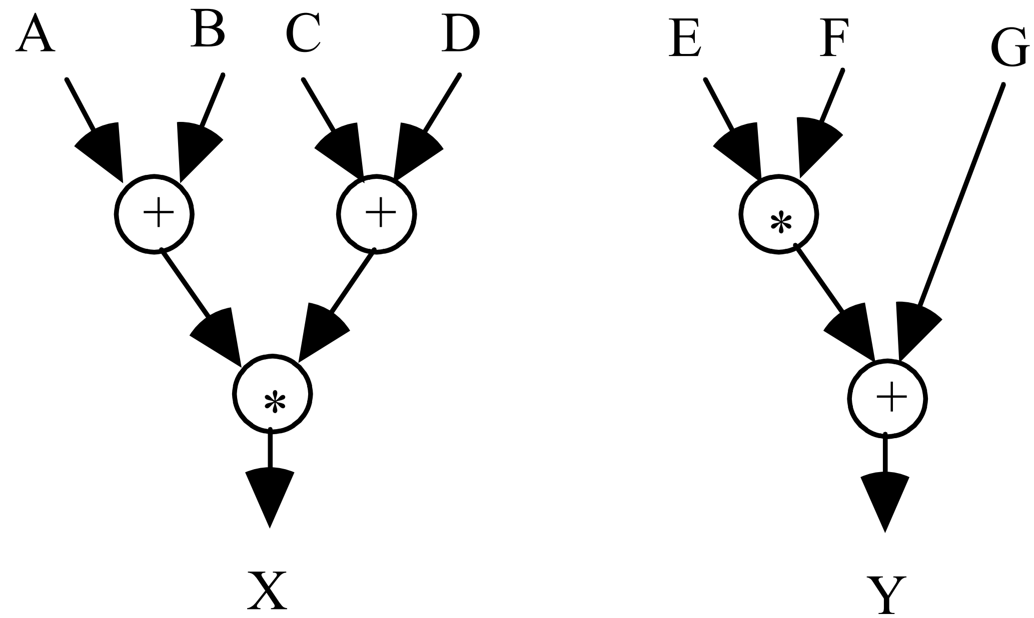
$T1 = A + B$

$T2 = C + D$

$X = T1 * T2$

$T1 = E * F$

$Y = T1 + G$



Five Time Steps

Two Time Steps

Assembly Language Representation appears to require 5 time steps

However, Data Flow Analysis reveals 2 Independent Computational Flows

- **(A, B, C, D) to compute X, and (E, F, G) to compute Y**

The Two Computational Flows can occur in Parallel if HW resources avail.

- **Also, operations within each flow can occur in parallel if HW is available**

If Inherent Sequentiality is preserved, parallel result is same as seq. result

- **Multi-Function Units**
  - **Augment HW w/ multiple Function Units to enable parallel proc. of instructs**
    - **Need to Check if proper type of Function Unit is available at certain time**
    - **Requires resource hazard analysis**
  - **Keep parallel processing transparent to programmer**
    - **Remove artificial sequentiality whenever possible, but not inherent seq.**
    - **Requires data hazard analysis to differentiate artificial vs. inherent seq.**
  - **If Resource is available and no data hazards exist, then Control Unit can:**
    - **Issue and begin executing a later instruction at same time, or**
    - **In aggressive case even before an earlier one is started (out of order)**
  - **Control unit does "lookahead" to identify instructions to process in parallel**
    - **Look-Ahead Control unit needs to perform:**
      - **- Detection: Determine which instructions can be executed in parallel**
        - **This analysis is based only on SW; it is HW Machine Independent**
      - **- Scheduling: Assigns concurrently executable instructions to FUs**
        - **This analysis is HW Dependent**
        - **Must Know Specific Number and Type of FUs on Target Machine**
    - **Look-Ahead must occur quickly; otherwise, no gain in speedup occurs**
      - **Time to Detect and Schedule Should be << Time to Execute on FUs**

Typically, Look-Ahead Control Unit HW is Complicated and Expensive

- Amount of HW and its complexity for CU can exceed the HW for FUs
- This is a technical justification for “extreme” RISC, VLIW and EPIC
  - Off-load Look-Ahead Control and Parallelism to SW Compilers
  - But effective compilers could not be written, so this approach and
    - EPIC VLIW HW have not (yet) been successful business models

- Degree: The number of instructions scanned ahead of the current instruction
  - Multiple degrees of "lookahead" are possible
  - We assume a (simple) single instruction lookahead issuing scheme:
    - Control unit issues consecutive instructions until a hazard is detected.
    - At that point, all issuing stops until the blocked statement can execute.
  - Higher degrees enable more potential speedup but are more complicated
    - e.g.) Using a double instruction (2^{nd} degree of) lookahead,
      - If scanned instruction has a hazard, “skip” over it and continue
        - scanning until the second instruction with a hazard is detected.
  - Using a Lookahead Degree > 1 can create Out-of-Order Execution
    - Consider two consecutive instructions: i followed by j
    - In single lookahead (our examples), j will never go before i
    - In higher degree of lookahead, j could be issued and complete before i

- **An Instruction can be issued if:**
  - **1) No data dependency is detected on any instruction currently executing.**
  - ***AND***
  - **2) The appropriate type of resource (function unit) is available.**

- Example:

High-Level Language Source	**Compiler Generated Reg Transfer Code**
A = (B + C) * (D + E)	S1: R1 = B + C
F = G + H + I + J	S2: R2 = D + E
H = K * L	S3: A = R1 * R2
	S4: R3 = G + H
	S5: R4 = I + J
	S6: F = R3 + R4
	S7: H = K * L

- CASE 1: One adder and one multiplier unit available.

Time	*1*	*2*	*3*	*4*	*5*
Adder	R1 = B+C	R2 = D+E	R3 = G+H	R4 = I+J	F = R3+R4
Multiplier			A = R1*R2		H = K*L
Hazard:	S2:adder	S3:R2	S5:adder	S6:R4,adder	

- Example: (same code as previous Case 1, but increase number of FUnits)

High-Level Language Source	Compiler Generated Reg Transfer Code
A = (B + C) * (D + E)	S1: R1 = B + C
F = G + H + I + J	S2: R2 = D + E
H = K * L	S3: A = R1 * R2
	S4: R3 = G + H
	S5: R4 = I + J
	S6: F = R3 + R4
	S7: H = K * L

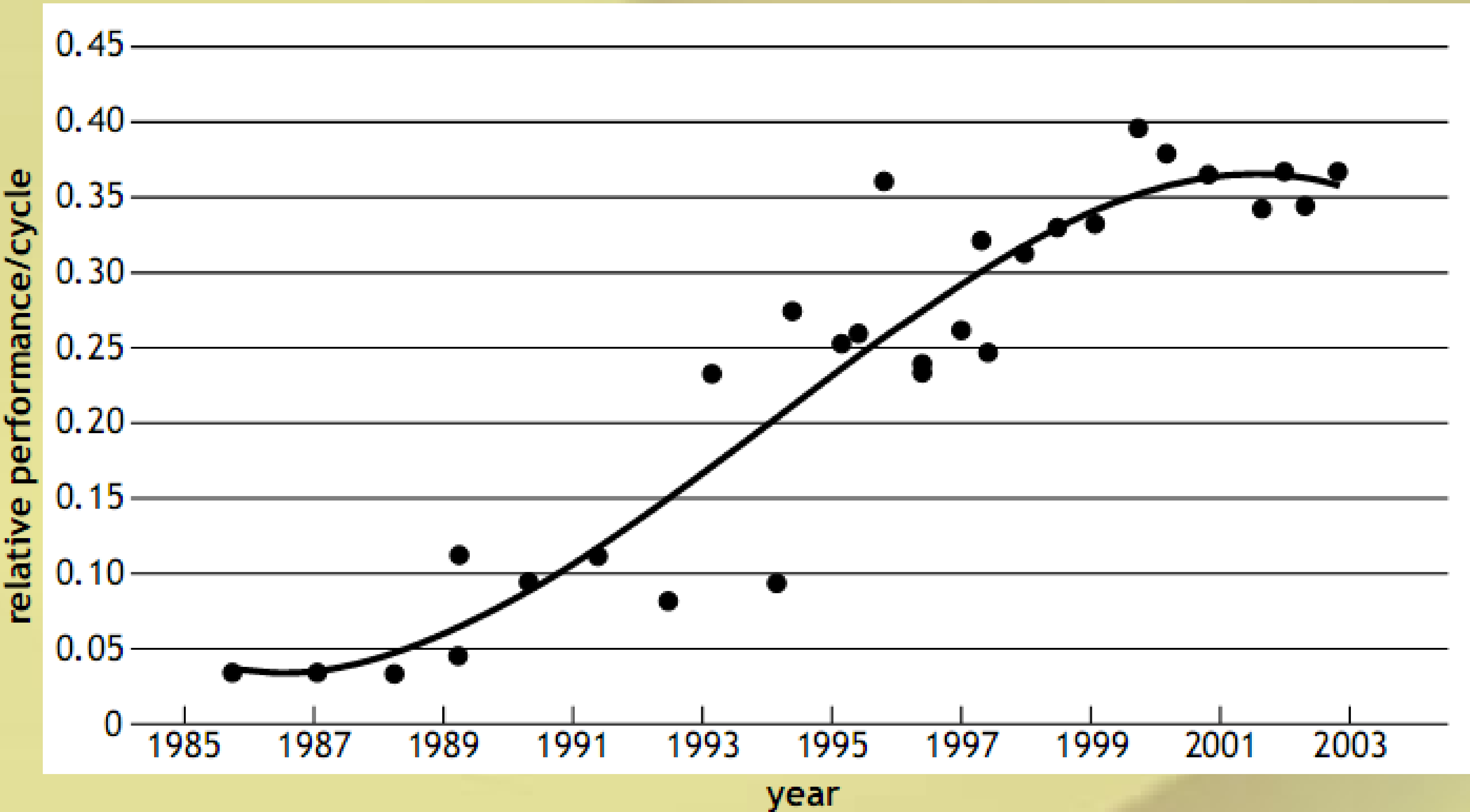
- CASE 2: Two adders and one multiplier unit available

Time	*1*	*2*	*3*	*4*	*5*
Adder 1	R1 = B+C	R3 = G+H	F = R3+R4		
Adder 2	R2 = D+E	R4 = I+J			
Multiplier		A = R1*R2	H = K*L		
Hazard:	S3: R1, R2	S6: R3, R4			

Speedup Results: Case 1: Finished in 5 time steps vs. 7 time steps
Case 2: Finished in 3 time steps vs. 7 time steps

- **Multi-Function Unit approach has yielded “reasonable” speedups in the past**
  - **“Reasonably” cost-effective from a HW point of view**
    - **e.g.) Cost of second adder is (almost) a “copy and paste” procedure**
  - **Simple Degree of Lookahead, and simple CU can yield good speed gains**
- **However: There are practical limits to "Transparent" ILP Parallel Processing**
  - **Decreasing marginal rates of return occur as more FUnits are added**
  - **Inherent sequentiality of source code algorithm is the ultimate bottleneck**
  - **High degree of lookahead needed to utilize large number of functional units**
    - **Need to continue issuing later instructions even if earlier one is blocked**
      - **Don’t want to hold up Instruct(j) because Instruc(i)’s FUnit is busy**
      - **Instruc(j)’s Type of Function Unit may be available**
    - **One method that allows this is Virtual Functional Units**
      - **Prevents blockage of instruction scanning due to a busy FU**
      - **Each FUnit is augmented with a queue of Virtual Functional Units**
      - **Instruc(i) will be dispatched assuming:**
        - **1) A physical Function Unit <u>or</u> a Virtual Function Unit is available**
        - **2) There are no data dependency hazards**
      - **VFUs will not necessarily allow Instruc(i) to be completed earlier**
        - **Execution of Instruc(i) will still eventually require a physical FU**

Intel Performance from ILP



- **The figure above shows that incremental performance increases from more aggressive ILP are arguably not worth the additional transistors, die size, and power utilization costs required in the Hardware.**

- **Data Hazard Classifications are Named by the ordering of Reads & Writes**

Consider two Instructions: Statement S1 followed by S2 (S2 occurs After S1)

Analysis of "overlaps" between the Reads and Writes of S1 and S2 is needed

Writes on the Left Hand Side (LHS) of Equal Sign against Reads on RHS

Three types of Data Hazards are possible: RAW, WAR, WAW

S2 is said to "Depend" on S1 if any one of the three data hazards exist

"Depend" implies that S1 and S2 must be executed in sequential order

Their order cannot be changed (S2 cannot be executed before S1)

Nor can S2 be executed in parallel with S1 (cannot use MFU ILP)

1) RAW (Read After Write):

True, Data Flow Dependency

RAW is the most common type of hazard that occurs

Example: S2 reading a source (R1) that S1 writes to

S1: R1 = R2 + R3

S2: R4 = R1 + R6

Example : S3 depends on S2, and S2 depends on S1, so no ILP possible

S1: R1 = 99

S2: R2 = R1

S3: R3 = R2

2) WAR (Write After Read):

Anti-Dependency: Mirror Image of RAW True Data Flow dependence

S2 writes to a target that also serves as a read source of S1

If S2 writes a target before it is read by S1, S1 gets (wrong) newer value

Example of WAR: S2 writes a target (R1) that S1 reads from

S1: R4 = R1 + R6
S2: R1 = R2 + R3

WARs can be Avoided by Buffering Source Operands (Register Renaming)

In above example, store value of R1 in a Buffer (R9) prior to S1 and S2

When S1 executes, it reads the Buffered value of R1 inside of R9

Therefore, results will still be correct even if S2 completes before S1

Example of Using Register Renaming to Solve WAR of above:

S0: R9 = R1
S1: R4 = R9 + R6
S2: R1 = R2 + R3

Another Example of Using Register Renaming to Avoid WAR:

		B = 3
B = 3	**Renamed to:**	**Z = B**
A = B + 1		**A = Z + 1**
B = 7		**B = 7**

3) WAW (Write After Write):

Output Dependency

S1 and S2 write to the same target

With aggressive ILP, S1 and S2 may update target at same time

A situation similar to a "Race" Condition can occur

Final Value of the Target is a function of who (S1 or S2) "wins" the Race

If the order of the instructions is changed,

Then the writes could end up being performed in the wrong order,

causing a change in the final output value of a (any) variable,

because the value written by S1 rather than S2 is left in target

Example:

S1: A = 3 + X
Sn: B = A / 4
S2: A = 5 * Y

Common target between S1 and S2 is the variable A

Changing the order of S1 and S2 will change final value of B

Buffering (Variable/Register Renaming) can be used to alleviate WAW

Example:

S1: A2 = 3 + X
Sn: B = A2 / 4
S2: A = 5 * Y

- **If only Single Instruction (Degree 1) Lookahead is used,**
  Then only the first type of Data Hazard (RAW) needs to be detected
- **WAR & WAW type of hazards only a potential problem for Lookahead > 1**
  i.e. An Instruct. is allowed to proceed even when a previous one is stalled
  Significantly complicates the amount (cost) of hazard analysis required
    Speedup (performance) improvement is typically small
- **Majority of ILP Speedup is achieved with simple RAW Single Level Lookahead**
  The examples used in this class were single level lookahead
  Relatively simple to maintain dependence while avoiding RAW hazard
  Preserves In-Order Execution of Instructions
- **Consider a boat with seats, each seat representing a FU of a MFUnit machine**
  Each boat represents one timeslot of parallel execution of all the MFUs
- **In-Order Instruct. Processor Dispatching Algorithm: Stall on 1st RAW conflict**
  1. **Fetch Instruction from front of line**
  2. **If that particular type of seat is unavailable, Resource Hazard, GoTo 4**
  3. **Check for RAW Hazard against any other Instructions already seated**
     If RAW detected, GoTo 4; Else Instruct. takes seat on boat, GoTo 1
  4. **Dispatch boat (all instructions on that boat execute in parallel on MFUs)**
  5. **Pull Next fresh boat (timeslot) up to dock and GoTo 1**

- If a more aggressive ILP Lookahead paradigm is used, then:
    - Operand Buffering (Register Renaming) for WARs and WAWs is needed
    - Overhead at Receiving dock is needed in addition to loading dock dispatch
- Out-of-Order Processor Dispatching Scheme (Loading Dock side):
    - Instructions Line up in a queue to board boat
    - Instruction Waits until its input operands are available (no RAW Hazard)
        - Once input operands are available, instruction is allowed to board
            - even though earlier, older Ins may still be waiting in line in front of it
        - That is, dispatcher can "skip" over blocked Instructions in an effort to
            - get other (later) instructions "on board" to fully pack the boat
    - Boat is dispatched when:
        - It is full (all FU seats are taken), or (in a more real-life scenario, when
        - Dispatcher reaches his/her lookahead "limit" over skipped passengers)
- Out-of-Order (OoO) Processor Retirement Scheme (Receiving Dock side):
    - The Instructions / Passengers / Results arrive OoO and are queued
    - A particular Instruction's result is written back (graduated or retired)
        - only after all earlier, older instructions arrive and
        - have had their results written back.
    - That is, the arriving passengers / results need to be put back in order.

- **Dispatch Algorithm must work much faster than a Function Unit execute time**
- **Most effective with help of Compiler (higher level reordering at source code)**
- **Example VLIW (Very Long Instruction Word) Architectures**

  An alternative architecture for exploiting Instruction-Level Parallelism

  VLIW Architectures are characterized by:

  - **A Processor that contains a large number of Function Units**
  - **A Long Instruction Word with a group of different fields, one for each FU**

    e.g.) A group of Four Fields for Four Function Units
  - **All (say, 4) sub-Instructions packed together in a single VLIW Instruction are pre-grouped together for Independent, Parallel Execution**
  - **Entire group of instructions is dispatched to the FUs in parallel**

  Exploiting the Full Capability of a VLIW CPU is the Compiler's Responsibility

  Compiler Must:

  Be intelligent enough to decide how to build the very long words

  Assemble many primitive operations into a single "instruction word"

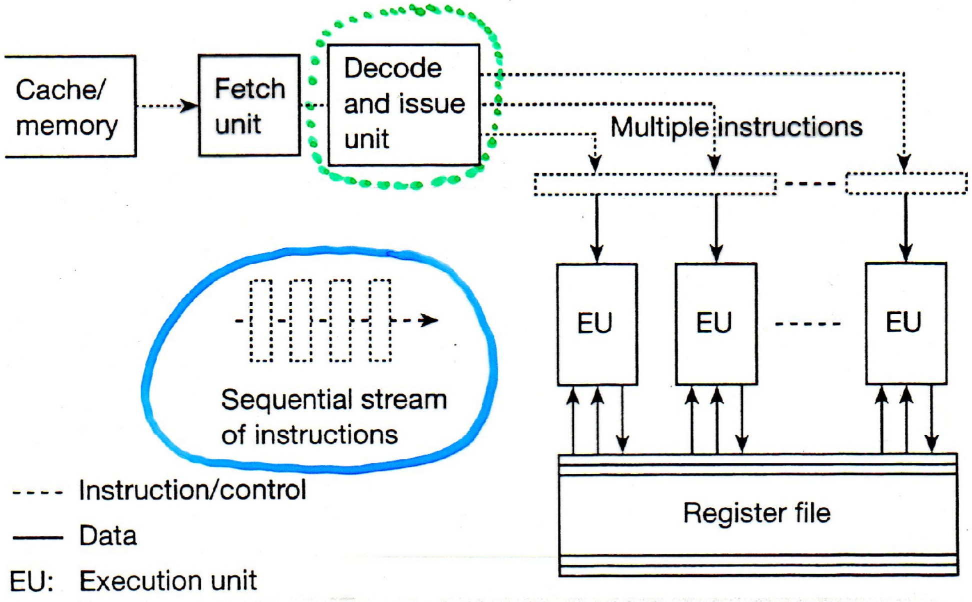
  Group together independent instructions executable in parallel

  Guarantee no dependencies b/w instructions that issue at same time

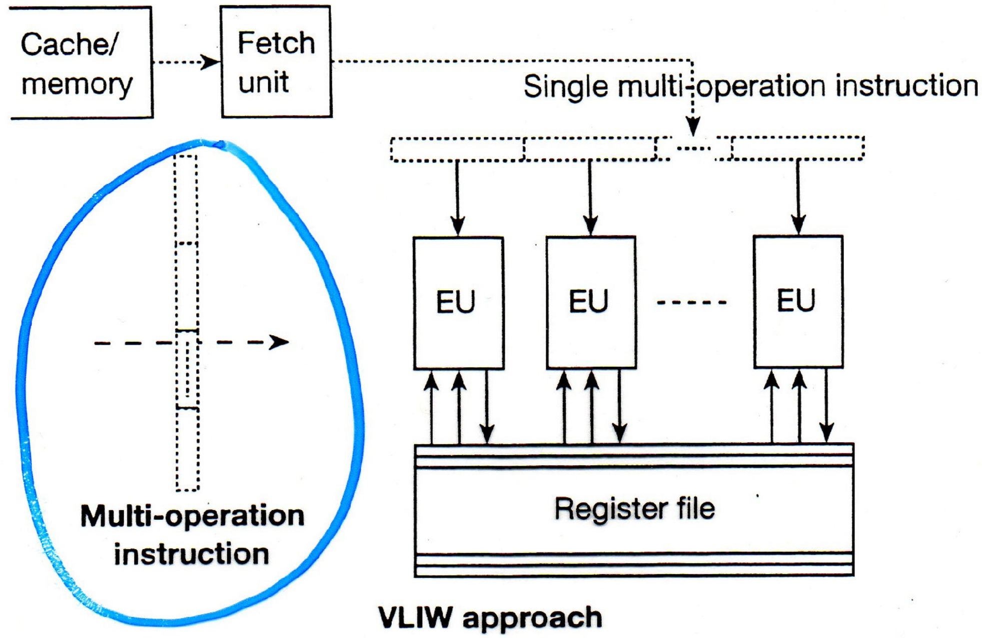
  Keep as many of the FUs busy by filling all the available operation slots

  Ensure that there are no resource hazards in the specific HW

- **VLIW Static Scheduling vs. The SuperScalar Dynamic Scheduling**
  - **SuperScalar processors perform dynamic scheduling/reordering in HW**
    - **Since the ILP is handled by the H/W, it is much more complex**
  - **Thus, Modern CPUs have developed very complicated hardware units for:**
    - **1) Rearranging Instructions at run time for effective OoO Execution**
    - **2) Performing Branch Prediction**
  - **The VLIW Architecture overcomes the two above complications by:**
    - **1) Having compiler pack several RISC instructions into one long word**
      - **Processor can then take unpack operations without further analysis**
        - **Processor simply gives each operation to an appropriate FU**
      - **These instructions are already certified to be executable in parallel**
      - **Processor H/W does not need to have the ability to detect and schedule the parallel operations in Real Time.**
    - **2) Eliminates Branch Prediction by executing all branch outcomes**
      - **After true outcome of branch is known, invalid results are discarded**
  - **In VLIW, the Hazard Analysis for ILP is handled completely by compiler**
    - **No dynamic scheduling nor reordering of Instructs is performed in H/W**
    - **The VLIW control logic has less responsibility, and is therefore simpler**
    - **Hardware can be smaller, cheaper, and require less power to operate**



Superscalar approach



VLIW approach

- **Example of a VLIW Architecture: The IA-64 (Itanium)**

  IA-64 had many advanced ideas in Computer Architecture Technology,

  But Compilers were not able to utilize its architecture effectively,

  So it was unsuccessful in the business market (nick named the "Itanic")

- **Data (and Control) Hazard Reduction is Hard to do in either HW or SW !**

- **Big-Picture Goal of this Discussion: Contrast *Program* Order vs. *Data* Order**

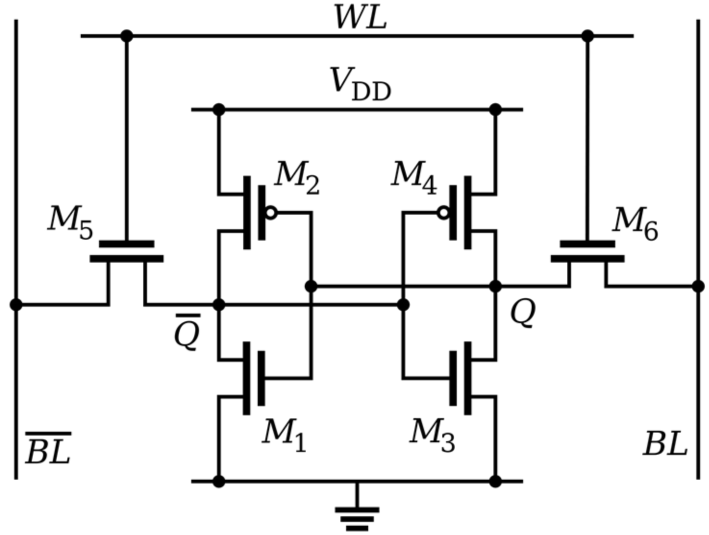
  The order in which program source code appears is only part of the story

  The order in which data flows is the other, sometimes more important part

- **Two Types of RAM Memory Terminology**

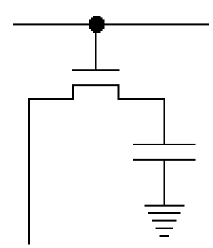
1) Static: More complex, more expensive, faster (7ns) design used for cache

Active Flip Flop gates require several transistors



2) Dynamic: Simpler, cheaper, slower (70ns) used for Main Memory "RAM"

Passive Capacitor Design is simpler and denser, therefore cheaper



- We discuss issues with #2: Dynamic Main Memory (Stick on MOBO) "RAM"

- Dynamic Storage: Uses continuous recirculation of some physical quantity
- Volatile Storage: Loses the stored information with time or power-off
  - e.g.) Capacitive memory cell requires power to refresh its charge
- Destructive Read-Out: Process of reading information effectively destroys it
  - Data then needs to be restored or effectively written back
    - e.g.) Capacitive memory cell gets discharged when it is read
- Because of RAM’s Dynamic, Volatile and Destructive Read characteristics,
  - a Wait period is needed by the Memory Controller to Refresh each cell
- Three Time Periods:
  1) Access Time: The interval b/w arrival of the signal activating memory and the completion of write-in or read-out
  2) Waiting Time: The "resting period" or the time the storage device needs to settle sufficiently before the next reference to it can be performed
     RAM memory is effectively off-duty and busy refreshing itself
  3) Cycle Time = Access Time + Waiting Time
     The speed with which consecutive accesses can occur
     The minimum time between successive R/Ws to that Memory Bank

- **Good memory management reduces the perceived impact of RAM Wait times**
- **Memory Interleaving: Divide memory into a number of separate banks**

  Goal: Avoid CPU idle during Wait (restore/refresh) period of mem cycle time

  Various degrees (“ways”) of interleaving architectures are possible

    e.g.) If memory is four-way interleaved, four separate memory banks exist

      Each contains one-quarter of the total words in memory

      Consecutive elements are interleaved between the four banks

  It is common to divide the words in the banks using a modulo function

    e.g.) If 4-way interleaving is used, words 1, 5, 9... will be in bank one

  Assume the CPU can overlap requests between banks (typically so)

    Thus, while an access (fetch) or wait (restore) is in progress for one bank, another different bank may be accessed

    That is, Memory Banks can work in Parallel

  Mainly useful for words in a straight-line program segment (i.e. instructions)

    Operands (data) normally results in out-of-sequence requests to memory

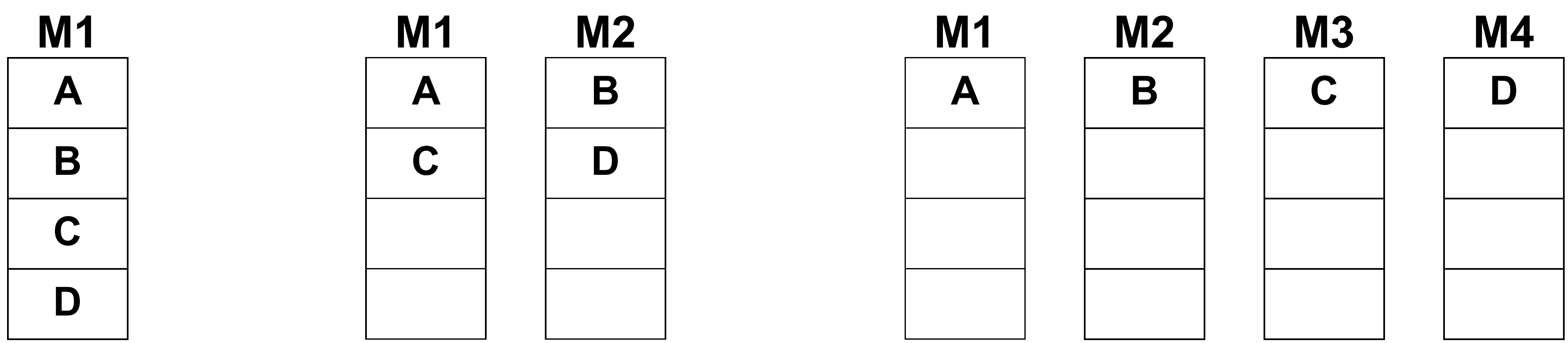
  In this way, it is possible to access consecutive words of memory rapidly

    Can effectively hide the Dynamic RAM Memory Wait Times from the CPU

    Average effective speed of memory can be increased effectively

- Consider a memory consisting of k modules
  - If all k modules are kept busy continuously,
    - Then effective cycle time can be decreased by up to a factor of k
    - Actual speedup varies depending upon the SW access pattern to memory
  - In our examples, we assume Access Time = Wait Time
    - Cycle Time = Access Time + Wait Time
  - Implementation:
    - A module is selected by the low-order (log2 k) bits of the MM address
    - The high-order bits determine the location within that module

- Example: Various Degrees of Interleaving for consecutive elements A, B, C, D



Non-Interleaved 2-Way Interleaved 4-Way Interleaved

- **Example: Assume that Memory has a 0.6 Access Time and 1.2 Cycle Time.**

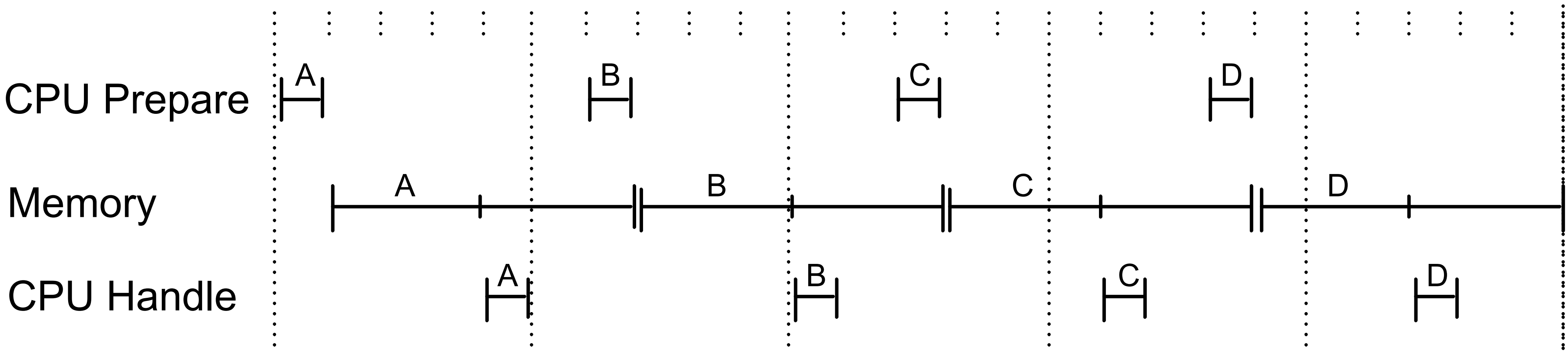
  The CPU takes 0.2 to prepare a memory request and 0.2 to process the result

  CPU can either prepare a memory request or process a result in parallel with the memory. The CPU can issue a request for an operand (e.g. B) before actually receiving and processing a previously requested operand (e.g. A).

  The total time needed by the overall system to perform an operation spans the time between the CPU's preparation of the very first memory request to the time when the CPU completes processing of the last memory request.

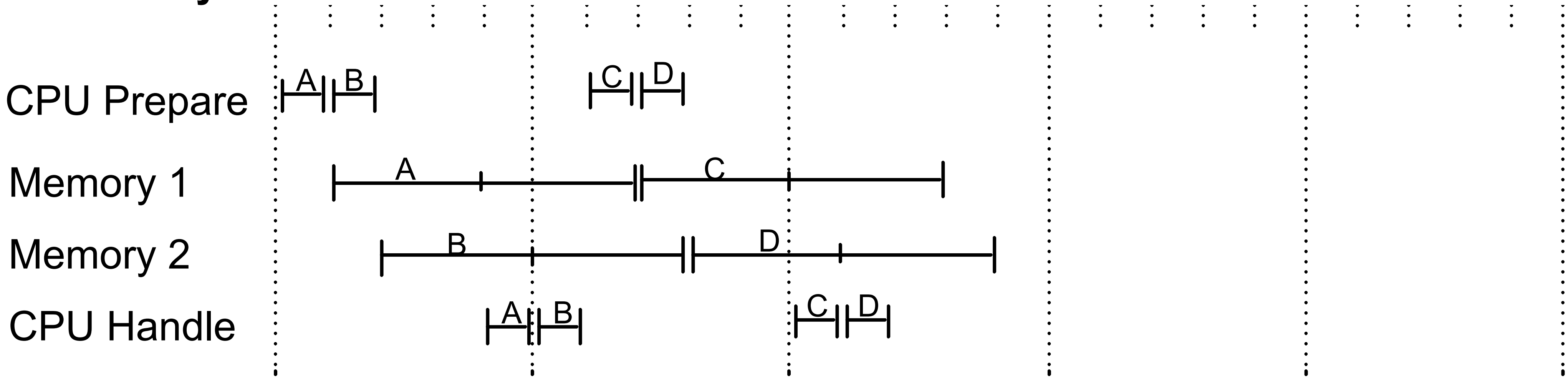
  Four operands, A thru D are to be retrieved from memory and processed.

➢ **Non-Interleaved:**



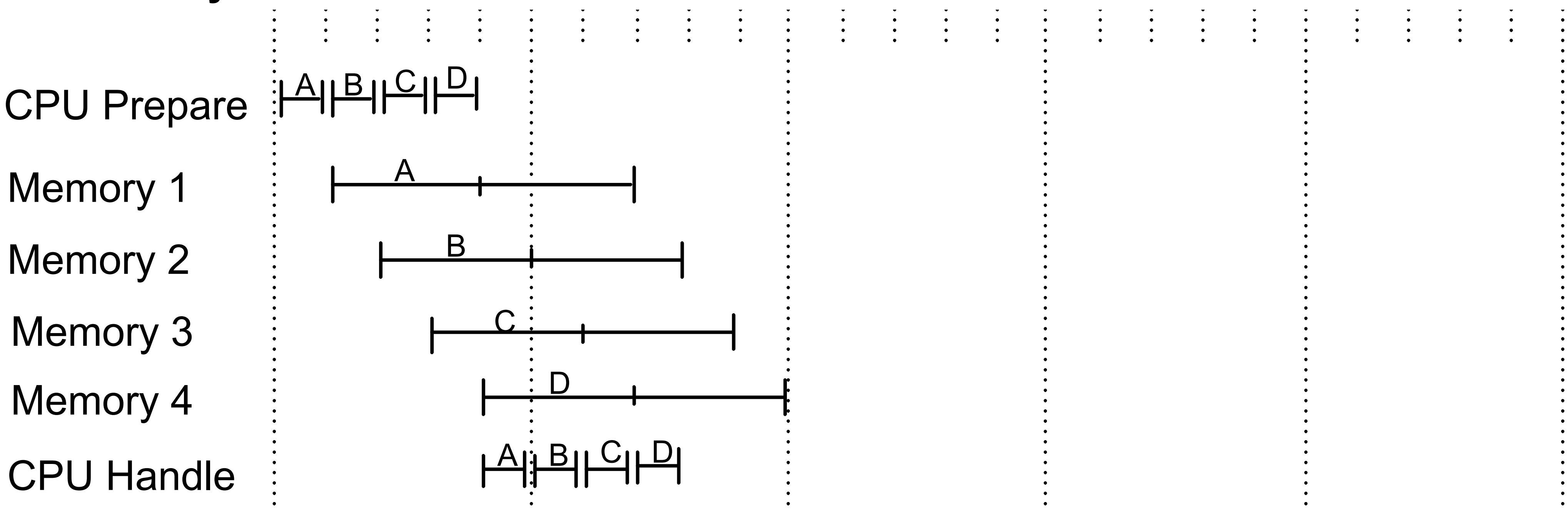
Total Time = 4.6 CPU Utilization = 8 / 23 = 34%

- **Two-Way Interleaved:**



Total Time = 2.4 **CPU Utilization = 8 / 12 = 66%** **Speedup = 4.6 / 2.4 = 1.9**

- **Four-Way Interleaved:**



Total Time = 1.6 **CPU Utilization = 8 / 8 = 100%** **Speedup = 4.6 / 1.6 = 2.8**

- **Interleaved memory makes contiguous block transfers very efficient**
  So, transferring blocks from the MM to the cache can be done quite fast

- **Interleaved Memory for Data Elements (Instead of Instruction Elements)**
  Interleaved memory works best with consecutive addresses (e.g. instructions)
  But, optimization can also be performed for data accesses (e.g. matrices)
    Best arrangement varies with dimensions of matrix; we look at an example.
  Typical Matrix operations involve access to rows, columns, and diagonals

➢ **Arrangement 1: "Straight" Storage Scheme across 4 Memory Modules**

M1	M2	M3	M4
A(00)	A(01)	A(02)	A(03)
A(10)	A(11)	A(12)	A(13)
A(20)	A(21)	A(22)	A(23)
A(30)	A(31)	A(32)	A(33)

Allows access to Consecutive elements of Rows & Diagonals without conflict
Conflict on Columns
  Example: A(y2) are all be read from M3 while M1, M2, and M4 are all idle

- **Arrangement 2: "Skewed" Storage Scheme - Barrel shift each row by one**

  Row 0: No Shift; Row 1: Shift Right One; Row 2: Shift Right Two ....

M1	M2	M3	M4
A(00)	A(01)	A(02)	A(03)
A(13)	A(10)	A(11)	A(12)
A(22)	A(23)	A(20)	A(21)
A(31)	A(32)	A(33)	A(30)

Allows access to Rows and Columns without conflict.

(Minor) Conflict on Diagonals; Degree of Interleaving is 2 (instead of 4)

- **Arrangement 3: "Two's Skewing" Storage - Barrel shift each row by two**

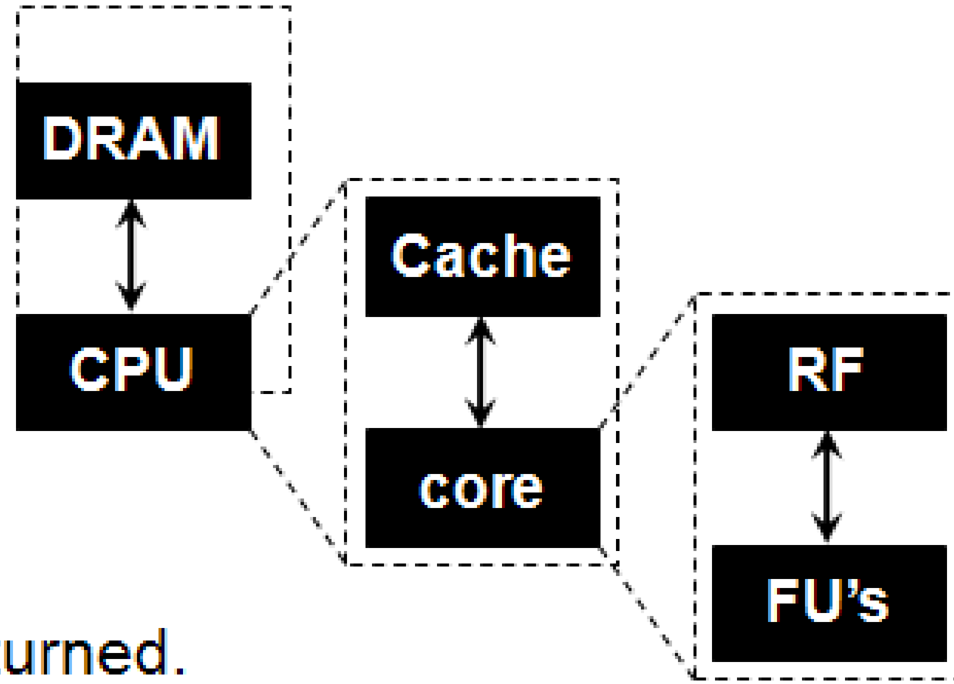
  Insert an extra module; Skip one module per row (wasteful of memory cells)

M1	M2	M3	M4	M5
A(00)	A(01)	A(02)	A(03)	--
A(13)	--	A(10)	A(11)	A(12)
A(21)	A(22)	A(23)	--	A(20)
--	A(30)	A(31)	A(32)	A(33)

Allows access to Rows, Columns, and Diagonals without conflict (4-way)

- **SW Programmer needs to be aware of HW Memory Access Patterns**
  - **DRAM, Cache, Register Files all have different bandwidths and latencies**
  - **Each level of the Memory Hierarchy can represent a 10X performance delta**

- Abstractly describe any system (or subsystem) as a combination of black-boxed storage, computational units, and the bandwidth between them.



- These can be hierarchically composed.

- A volume of data must be transferred from the storage component, processed, and another volume of data must be returned.

- Consider the basic parameters governing performance of the channel: **Bandwidth, Latency**
  - Bandwidth can be measured in: GB/s, Gflop/s, MIPS, etc…
  - Latency can be measured in: seconds, cycles, etc…

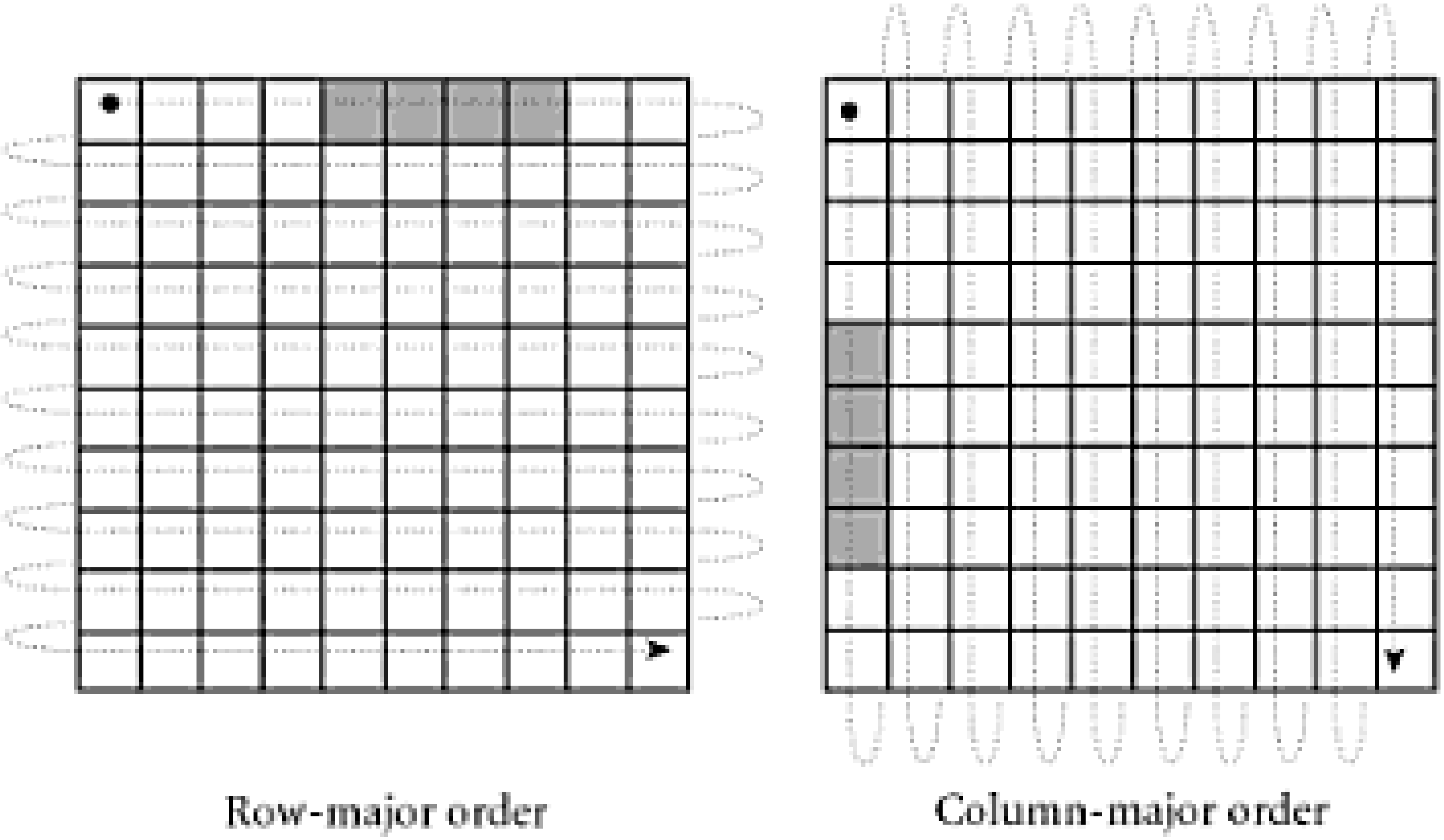


Figure 7.7 Row- and column-major memory layout for two-dimensional arrays. In row-major order, the elements of a row are contiguous in memory; in column-major order, the elements of a column are contiguous. The second cache line of each array is shaded, on the assumption that each element is an eight-byte floating-point number, that cache lines are 32 bytes long (a common size), and that the array begins at a cache line boundary. If the array is indexed from A[0,0] to A[9,9], then in the row-major case elements A[0,4] through A[0,7] share a cache line; in the column-major case elements A[4,0] through A[7,0] share a cache line.

- **Row-Major vs. Column-Major Memory Storage Patterns for Matrices**

  Single Dimension Arrays are stored in contiguous locations in memory

  So A[1] is followed by A[2], then A[3], etc.

  Multi-Dimensional Arrays (Matrices) can be either Row- or Column-Major

  Row-Major:

  Consecutive Mem Locations differ by 1 in the last subscript of Matrix

  So M[2, 4] is followed by M[2, 5]

  Column-Major:

  Consecutive Mem Locations differ by 1 in the first subscript of Matrix

  So M[2, 4] is followed by M[3, 4]

- **Difference can be Important when using nested loops to access all elements of a multi-dimensional Matrix – something that is very commonly done.**
  - ➢ **Speed of loop is memory bound and dependent on cache hit rate**

- **Demo: Which Loop Nesting is Better, or does it even matter ?**

```
for (i=0; i<1000; i++){
  for (j=0; j<1000; j++){
    M[i][j] = 0 ;
  }
}
```

```
for (j=0; j<1000; j++){
  for (i=0; i<1000; i++){
    M[i][j] = 0 ;
  }
}
```

- **Yes ….. it matters …… sometimes …… and as far as which one is better …… it depends**
  - **It depends on the Size of the Matrix and the Programming Language Used.**
- **For Small Matrices, all of the elements will fit completely in cache throughout the loops**
  - **So orientation of cache lines will not matter**
  - **Both loop nestings will perform the same**
  - **SW Programmer should be aware of the particular machine's HW cache size**
- **For Large Matrices, cache lines accessed early in the traversal will be evicted to make room for lines accessed later in the traversal.**

- **If matrix elements are accessed in order of consecutive addresses, then**
  - **Each miss will bring into the cache a line consisting not only of the desired element (the one that missed), but the next several elements as well.**
  - **Spatial locality of reference will effectively be leveraged to predict next needed elements, and therefore, will boost the cache hit rate and performance of SW.**
- **If matrix elements are accessed *across* cache lines instead (the wrong way), then**
  - **Spatial locality of reference will be lost because no pre-fetching occurs.**
  - **Practically every access will be a cache miss dramatically reducing performance**

- **It Depends on the Language used too:**

Best for C / **Row-Major**

```
for i
   for j
      M[i][j]
```

Best for Fortran / **Column-Major**

```
for j
   for i
      M[i][j]
```

- **Introduction to Scheduling: Process States**

A Process moves through various states between being submitted and completed:

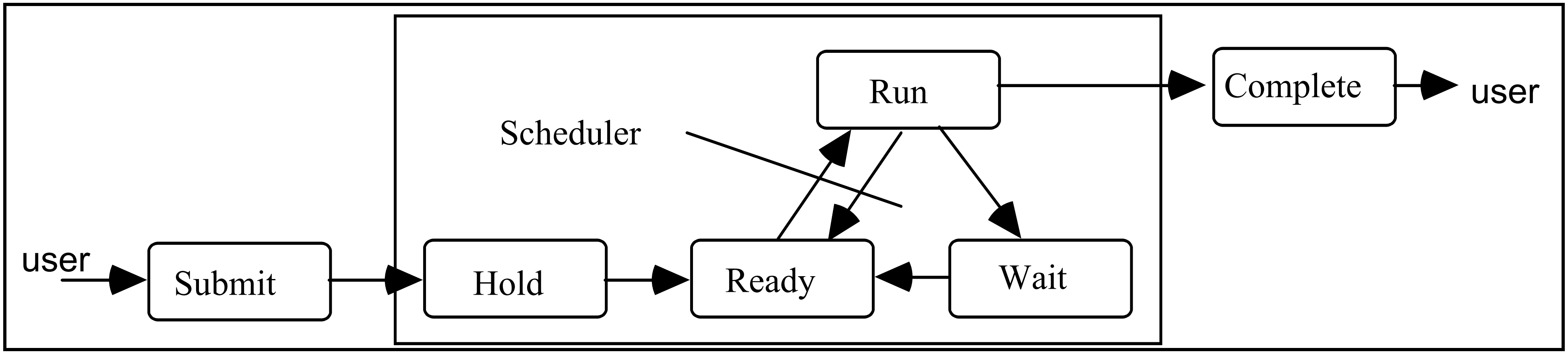
1) Hold State: A user's job has been Submitted and Spooled onto disk.
Process has simply been "read" into the system's high-speed storage area.
There can be many processes (from many users) queued in the Hold State.

2) Ready: Process is "ready to run", but must wait for its turn on the processor.
The resources (e.g. memory) it needs are available & can be allocated to it.
Process must wait for the Scheduler to give it time on the CPU.
There can be many processes in the Ready Queue waiting for the CPU.

3) Run: Process is currently being executed.
Process has been allocated the CPU and is actively running.
On a uniprocessor, only one process at a time can be in the Run state.
Process continues to possess the CPU and run until it either:
- Is interrupted by the Scheduler (because its time slice is up), or
- It performs I/O, thereby (voluntarily) moving itself to the Wait State.

4) Wait: Process is waiting for some event to happen (e.g. I/O operation).
Process gives up CPU while waiting for a slower device to complete its request.
Also called the Blocked State (i.e., process blocks itself by doing I/O).
After I/O is complete, process returns to Ready Queue.
There can be several processes waiting for I/O to complete for them.

5) Complete: Process has finished and all of its resources are now deallocated.



The life cycle of a process can be represented by transitions between these states.

- **The Ready-Run-Wait Triad is the most significant portion of a Job's Life Cycle**

  A Process will typically cycle many times through the Ready-Run-Wait States.

- **Note the following two Opportunities for Global System Optimization**

  1) Avoiding (unnecessary) Transitions to the Wait State with Small Incremental I/O

  - Input/Output Buffering by the Operating System

  When a Process does I/O while Running, it is moved to Wait State due to slow I/O

  Substantial Overhead can be incurred with numerous small-sized I/O's

  Operating System's Goal is to Maximize Performance of Individual Jobs & System

  Minimizing one Job's swap to Wait States will also maximize overall System

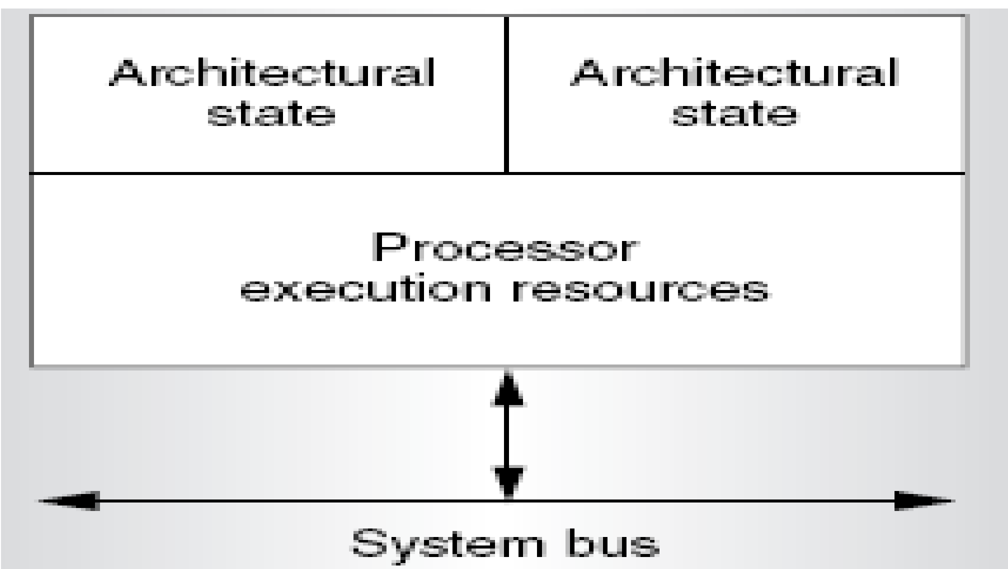
  OS buffers numerous small-sized incremental I/O's into fewer large-sized I/O's

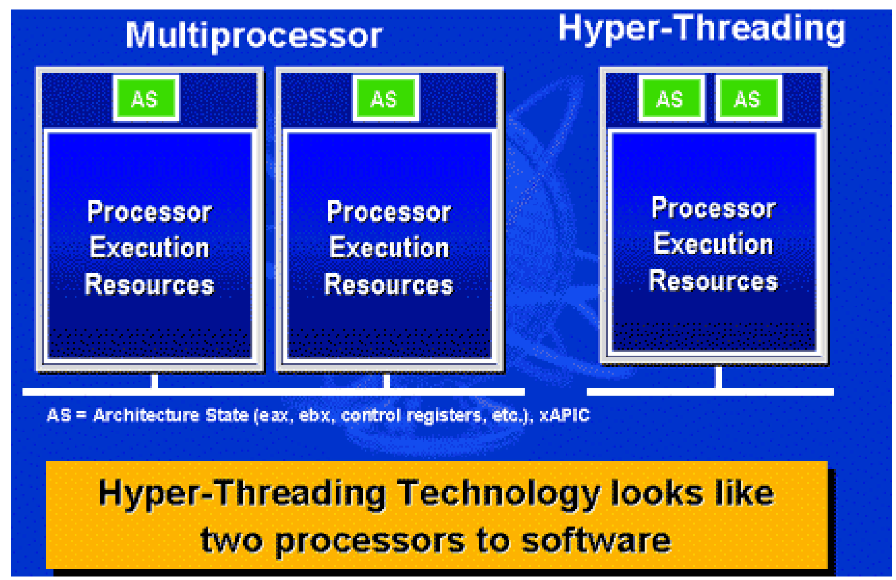
  Under certain circumstances, OS can buffer I/O with no apparent effect on outcome

  e.g.) If Output not read by user until entire program completes

2) Minimizing the Run-Time Penalty and Overhead of State Transitions

- <u>HyperThreading</u>: Intel Technique to Accelerate Context Switching via extra HW
  - Significant SW (run-time) overhead is spent Swapping Processes In & Out of Run
    - Complete Architectural State of Process needs to be stored upon swap out,
      - then retrieved and loaded on a context switch swap in.
    - Architectural State includes the Program Counter and Register Values
  - HyperThreading Duplicates the Registers that store a thread's Architectural State
  - One Physical Processor appears to be Two Logical Processors to the OS SW
    - OS SW can effectively keep 2 threads (2 AStates) "alive" at the same time
  - However, execution resources are not duplicated as in a true multi-processor
    - HT enables switching between threads to be accomplished at fast(er) HW speed
    - So only one Thread actually runs at any instant in time (not true parallelism)





- **HyperThreading Yields a Cost- Effective Performance Boost**
  - **About 5% extra HW Register Support has been shown to give about a 15% speedup**
- **HyperThreading is, for the most part, Transparent to OS SW and Application-level SW**
  - **However, to benefit from HyperThreading, Application SW should be multi-threaded**
    - **To obtain peak performance, the number of active threads should be equal to the number of logical processors in the system.**
  - **OS can also maximize perf gains by knowing difference b/w logical vs. physical Procs**
    - **For example, assume a dual-core chip, where each core (P1, P2) is HyperThreaded**
      - **So OS now sees four logical processors (L1 & L2 from P1; L3 & L4 from P2)**
      - **If only two threads (T1 and T2) need to be run,**
        - **Mapping T1 to L1, and T2 to L2 will yield poor results because both are on P1**
          - **T1 and T2 would now need to share the single set of resources on P1**
          - **P2, the other real physical core with another set of resources, is unused**
        - **Better OS mapping would run T1 on L1 (or L2), and T2 on L3 (or L4) to better balance the workload among the real physical processors, P1 and P2.**
- **HT was first introduced on the Intel Xeon processor in early 2002 for the server market Then later in the same year on high-end (3Ghz+) Intel Pentium 4 for consumer market.**
- **Intel estimates that HT will be used in about 75 percent of its chips**

➢ **Key point: Although OS sees 2 logical procs, HT performance is not 2X better; only 15%**