

# Advanced Architecture

*Computer Organization and Assembly Languages*

*Yung-Yu Chuang*

*2008/11/10*

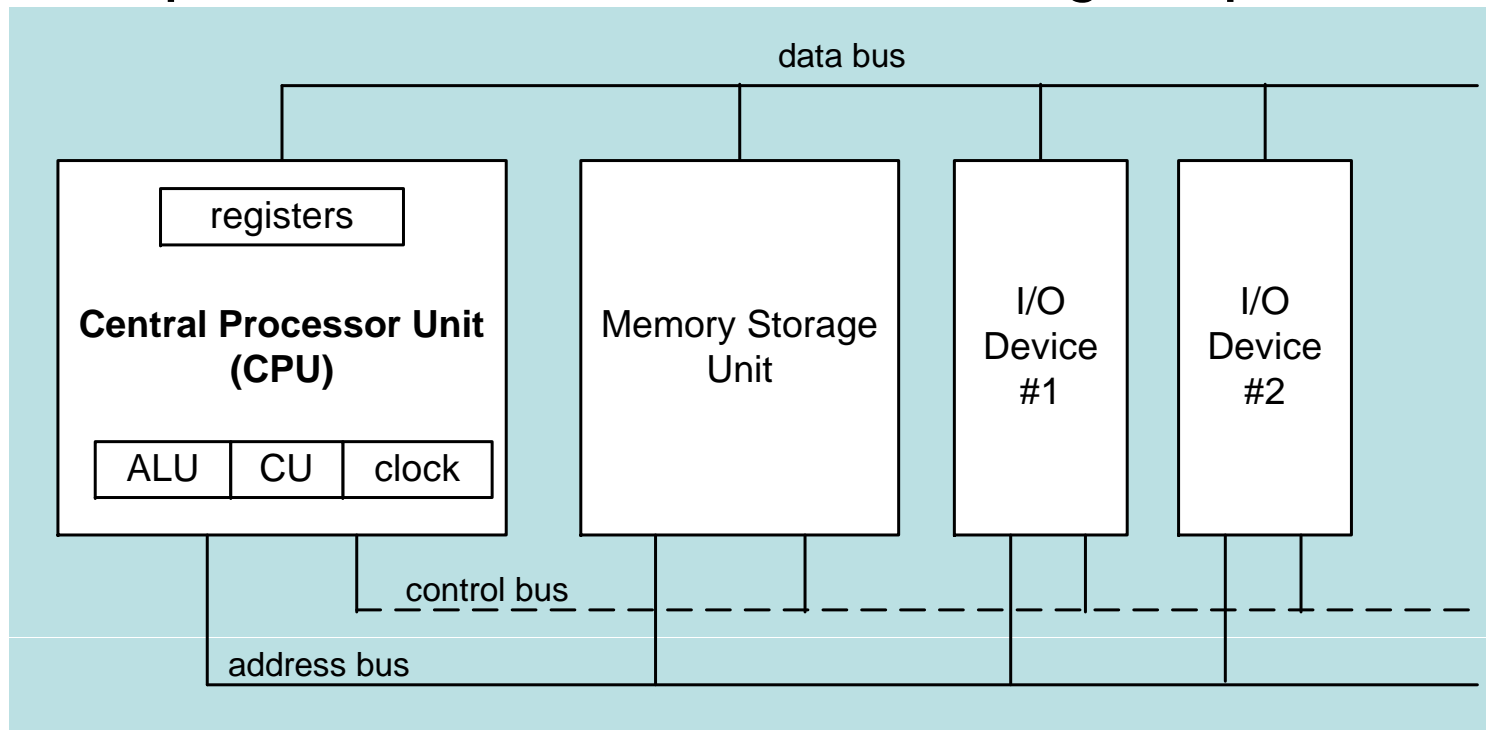
*with slides by S. Dandamudi, Peng-Sheng Chen, Kip Irvine, Robert Sedgwick and Kevin Wayne*

# Basic architecture

# Basic microcomputer design



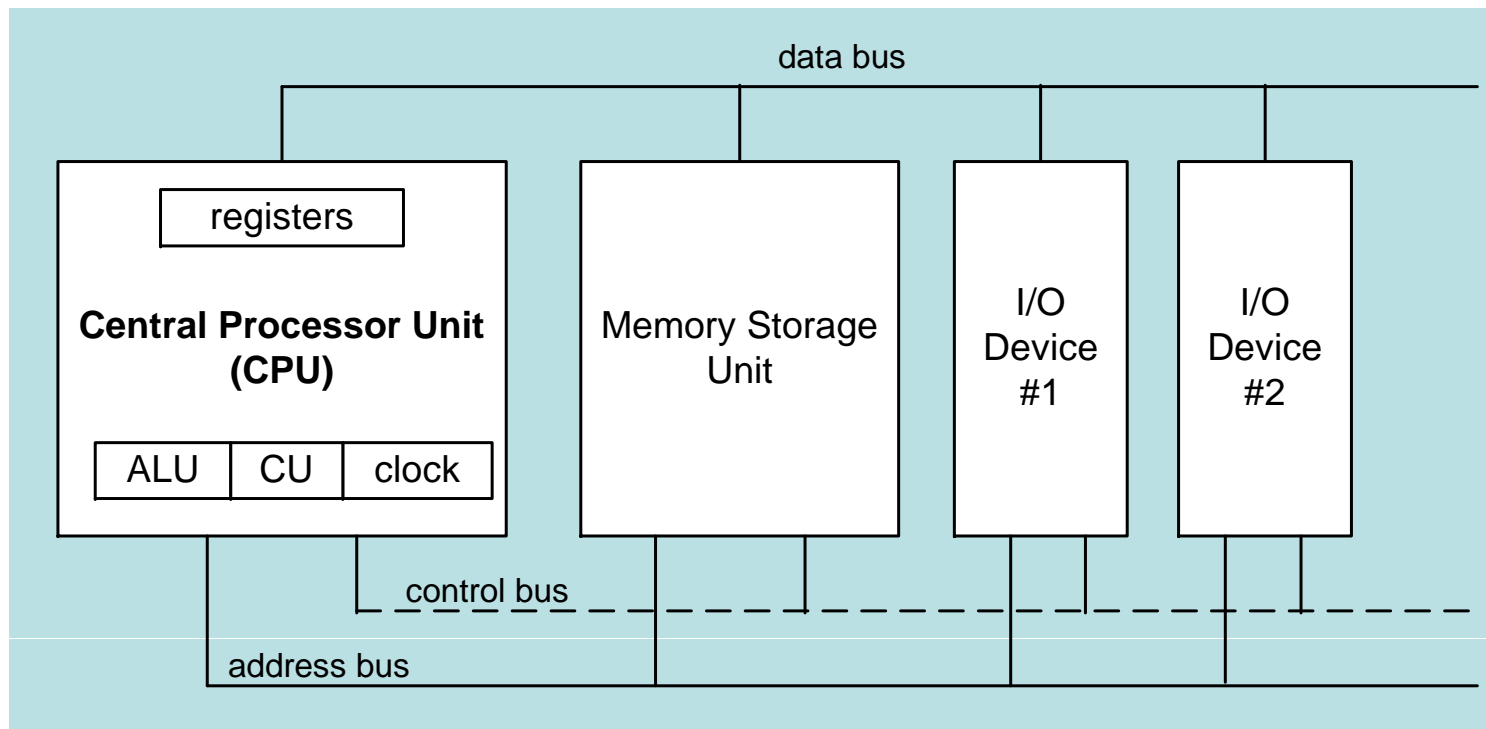
- clock synchronizes CPU operations
- control unit (CU) coordinates sequence of execution steps
- ALU performs arithmetic and logic operations



# Basic microcomputer design



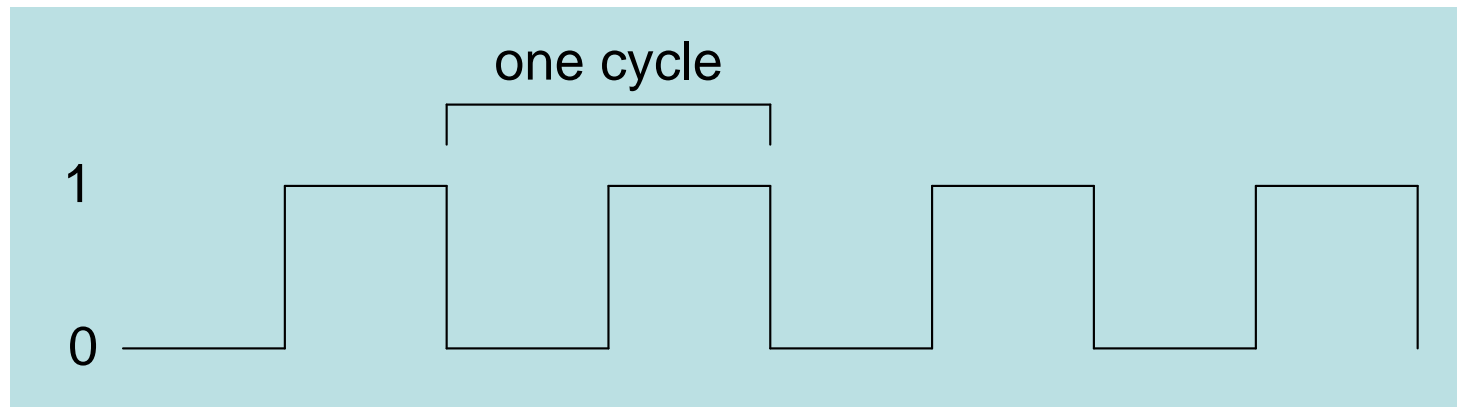
- The memory storage unit holds instructions and data for a running program
- A bus is a group of wires that transfer data from one part to another (data, address, control)



# Clock

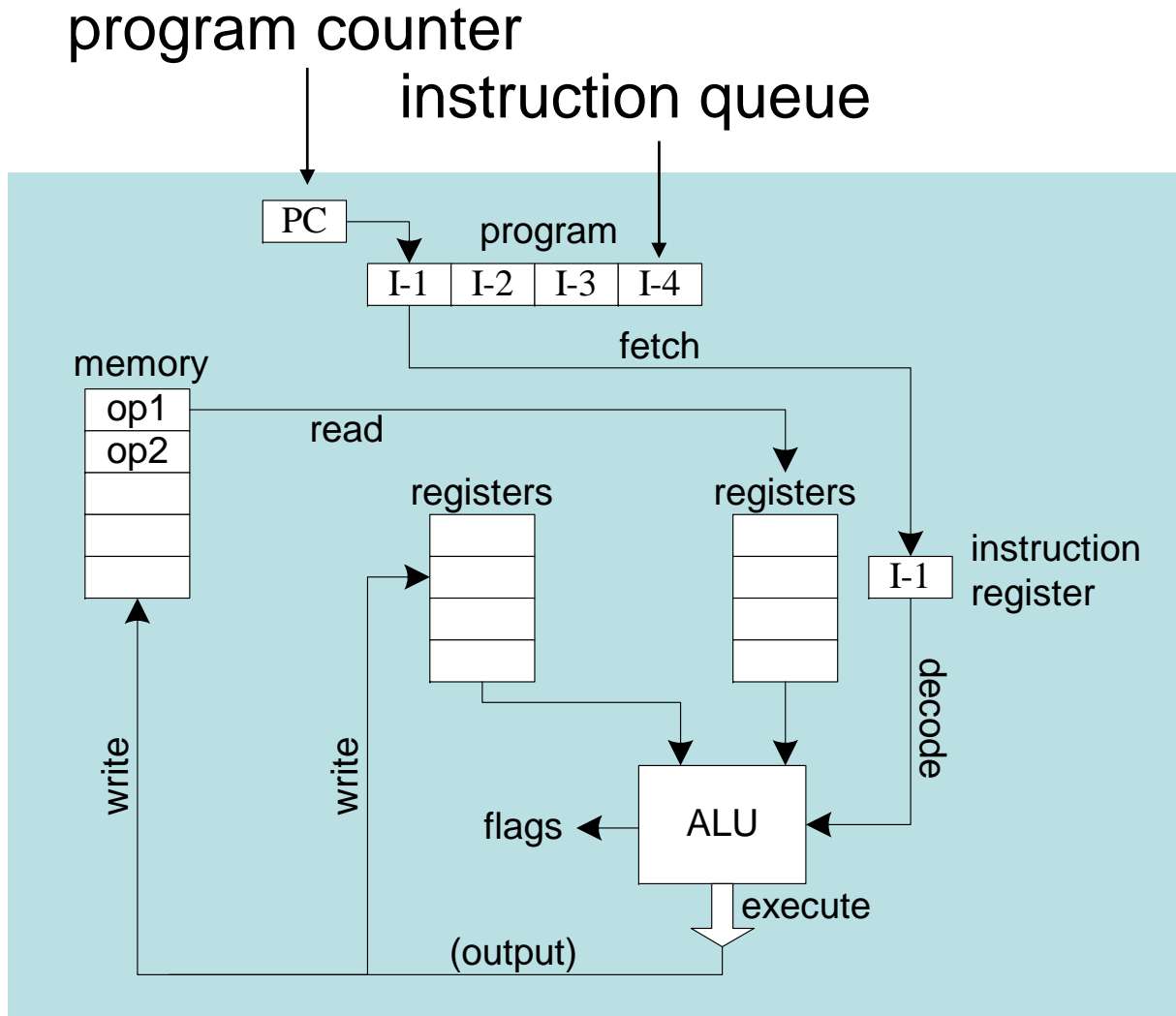
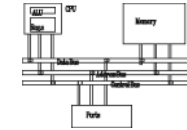


- synchronizes all CPU and BUS operations
- machine (clock) cycle measures time of a single operation
- clock is used to trigger events



- Basic unit of time, 1GHz→clock cycle=1ns
- An instruction could take multiple cycles to complete, e.g. multiply in 8088 takes 50 cycles

# Instruction execution cycle



- Fetch
- Decode
- Fetch operands
- Execute
- Store output

# Pipeline

# Multi-stage pipeline



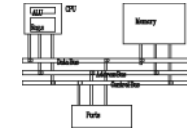
- Pipelining makes it possible for processor to execute instructions in parallel
- Instruction execution divided into discrete stages

Example of a non-pipelined processor. For example, 80386. Many wasted cycles.

		Stages					
		S1	S2	S3	S4	S5	S6
Cycles	1	I-1					
	2		I-1				
	3			I-1			
	4				I-1		
	5					I-1	
	6						I-1
	7	I-2					
	8		I-2				
	9			I-2			
	10				I-2		
	11					I-2	
	12						I-2



# Pipelined execution



- More efficient use of cycles, greater throughput of instructions: (80486 started to use pipelining)

		Stages					
		S1	S2	S3	S4	S5	S6
Cycles	1	I-1					
	2	I-2	I-1				
	3		I-2	I-1			
	4			I-2	I-1		
	5				I-2	I-1	
	6					I-2	I-1
	7						I-2

For  $k$  stages and  $n$  instructions, the number of required cycles is:

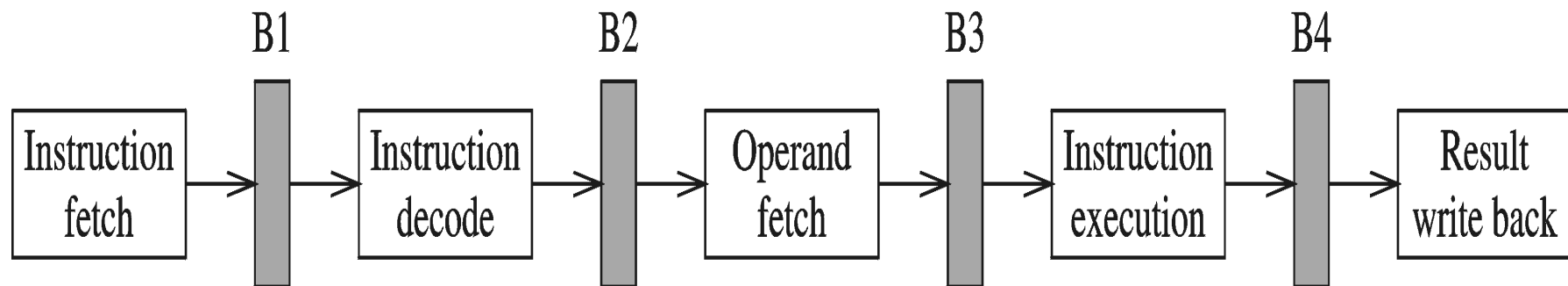
$$k + (n - 1)$$

compared to  $k \cdot n$

# Pipelined execution



- Pipelining requires buffers
  - Each buffer holds a single value
  - Ideal scenario: equal work for each stage
    - Sometimes it is not possible
    - Slowest stage determines the flow rate in the entire pipeline



# Pipelined execution

---

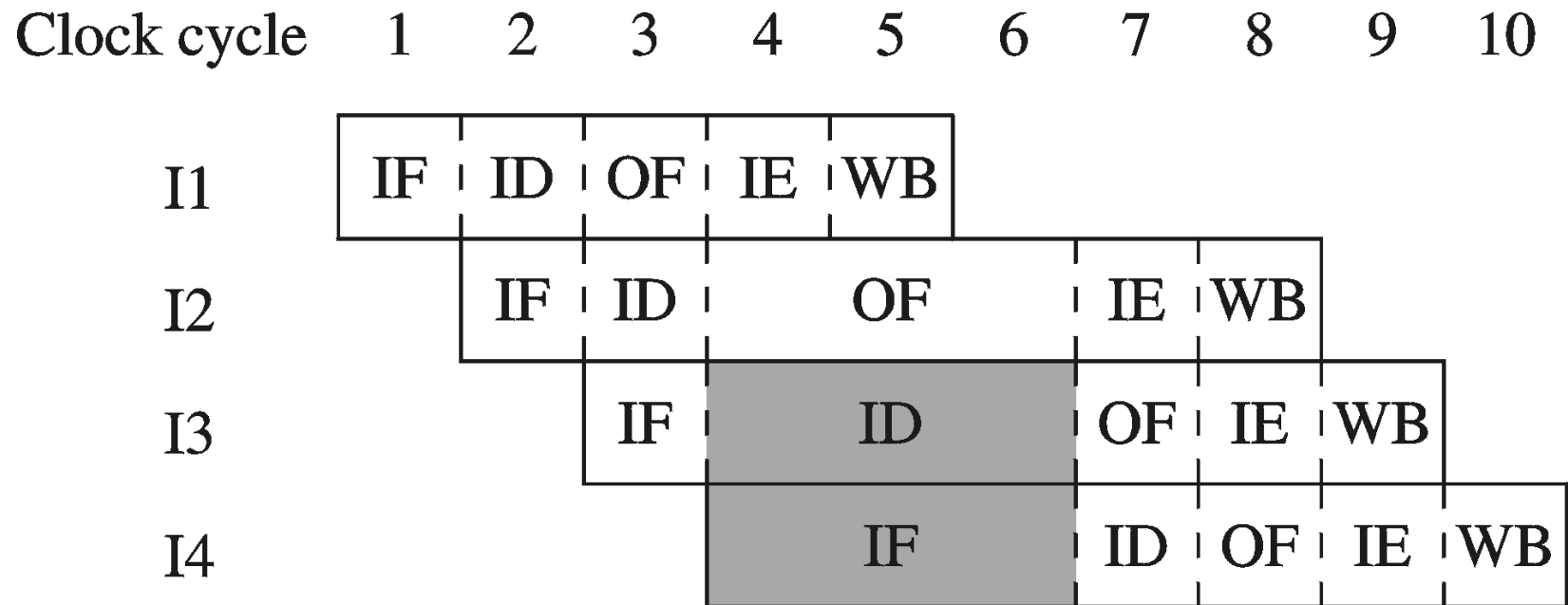


- Some reasons for unequal work stages
  - A complex step cannot be subdivided conveniently
  - An operation takes variable amount of time to execute, e.g. operand fetch time depends on where the operands are located
    - Registers
    - Cache
    - Memory
  - Complexity of operation depends on the type of operation
    - Add: may take one cycle
    - Multiply: may take several cycles

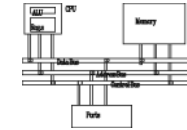
# Pipelined execution



- Operand fetch of I2 takes three cycles
  - Pipeline *stalls* for two cycles
    - Caused by hazards
  - Pipeline stalls reduce overall throughput



# Wasted cycles (pipelined)



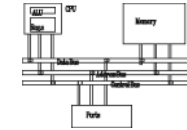
- When one of the stages requires two or more clock cycles, clock cycles are again wasted.

		Stages					
		S1	S2	S3	S4	S5	S6
Cycles	1	I-1					
	2	I-2	I-1				
	3	I-3	I-2	I-1			
	4		I-3	I-2	I-1		
	5			I-3	I-1		
	6				I-2	I-1	
	7				I-2		I-1
	8				I-3	I-2	
	9				I-3		I-2
	10					I-3	
	11						I-3

For  $k$  stages and  $n$  instructions, the number of required cycles is:

$$k + (2n - 1)$$

# Superscalar



A superscalar processor has multiple execution pipelines. In the following, note that Stage S4 has left and right pipelines (u and v).

		Stages						
		S1	S2	S3	S4		S5	S6
Cycles	1	I-1						
	2	I-2	I-1					
	3	I-3	I-2	I-1				
	4	I-4	I-3	I-2	I-1			
	5		I-4	I-3	I-1	I-2		
	6			I-4	I-3	I-2	I-1	
	7				I-3	I-4	I-2	I-1
	8					I-4	I-3	I-2
	9						I-4	I-3
	10							I-4

For  $k$  states and  $n$  instructions, the number of required cycles is:

$$k + n$$

Pentium: 2 pipelines

Pentium Pro: 3

# Pipeline stages

---



- Pentium 3: 10
- Pentium 4: 20~31
- Next-generation micro-architecture: 14
- ARM7: 3

# Hazards

---



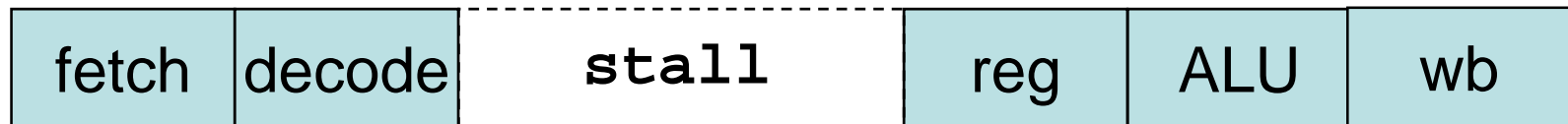
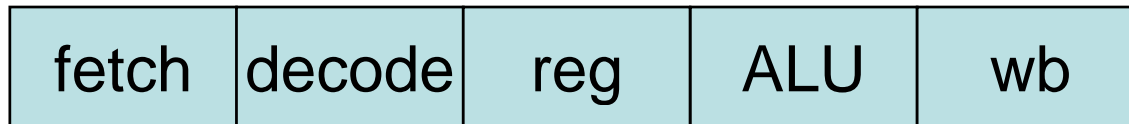
- Three types of hazards
  - Resource hazards
    - Occurs when two or more instructions use the same resource, also called *structural hazards*
  - Data hazards
    - Caused by data dependencies between instructions, e.g. result produced by I1 is read by I2
  - Control hazards
    - Default: sequential execution suits pipelining
    - Altering control flow (e.g., branching) causes problems, introducing control dependencies



# Data hazards



```
add r1, r2, #10      ; write r1
sub r3, r1, #20      ; read r1
```

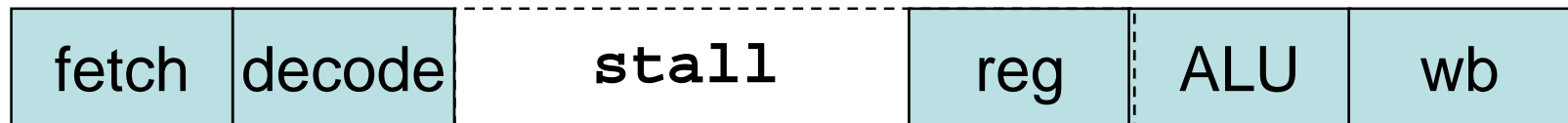


# Data hazards



- Forwarding: provides output result as soon as possible

```
add r1, r2, #10      ; write r1
sub r3, r1, #20      ; read r1
```

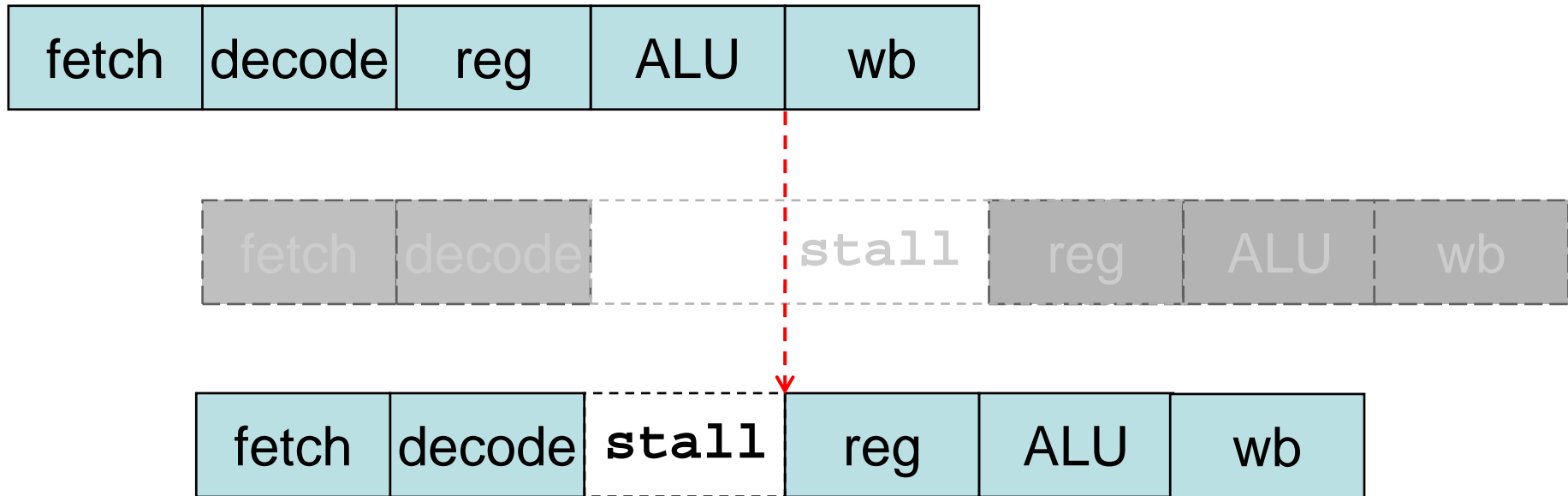


# Data hazards



- Forwarding: provides output result as soon as possible

```
add r1, r2, #10      ; write r1
sub r3, r1, #20      ; read r1
```



# Control hazards

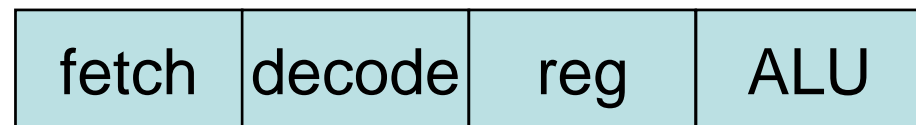
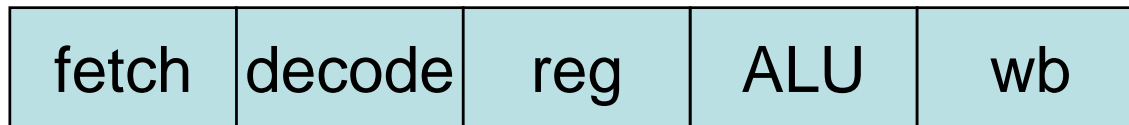


`bz r1, target`

`add r2, r4, 0`

`...`

`target: add r2, r3, 0`



# Control hazards

---



- Branches alter control flow
  - Require special attention in pipelining
  - Need to throw away some instructions in the pipeline
    - Depends on when we know the branch is taken
    - Pipeline wastes three clock cycles
      - Called *branch penalty*
  - Reducing branch penalty
    - Determine branch decision early

# Control hazards



- Delayed branch execution
  - Effectively reduces the branch penalty
  - We always fetch the instruction following the branch
    - Why throw it away?
    - Place a useful instruction to execute
    - This is called *delay slot*

```
add    R2,R3,R4
branch target
sub    R5,R6,R7
. . .
```

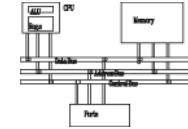
```
branch target
add    R2,R3,R4
sub    R5,R6,R7
. . .
```

Delay slot



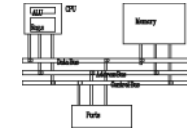
# Branch prediction

---



- Three prediction strategies
  - Fixed
    - Prediction is fixed
      - Example: **branch-never-taken**
        - » Not proper for loop structures
  - Static
    - Strategy depends on the branch type
      - Conditional branch: always not taken
      - Loop: always taken
  - Dynamic
    - Takes run-time history to make more accurate predictions

# Branch prediction



- Static prediction
  - Improves prediction accuracy over Fixed

Instruction type	Instruction Distribution (%)	Prediction: Branch taken?	Correct prediction (%)
Unconditional branch	$70 \times 0.4 = 28$	Yes	28
Conditional branch	$70 \times 0.6 = 42$	No	$42 \times 0.6 = 25.2$
Loop	10	Yes	$10 \times 0.9 = 9$
Call/return	20	Yes	20

Overall prediction accuracy = **82.2%**



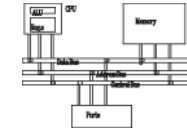
# Branch prediction

---



- Dynamic branch prediction
  - Uses runtime history
    - Takes the past  $n$  branch executions of the branch type and makes the prediction
  - Simple strategy
    - Prediction of the next branch is the **majority** of the previous  $n$  branch executions
    - Example:  $n = 3$ 
      - If two or more of the last three branches were taken, the prediction is “branch taken”
    - Depending on the type of mix, we get more than 90% prediction accuracy

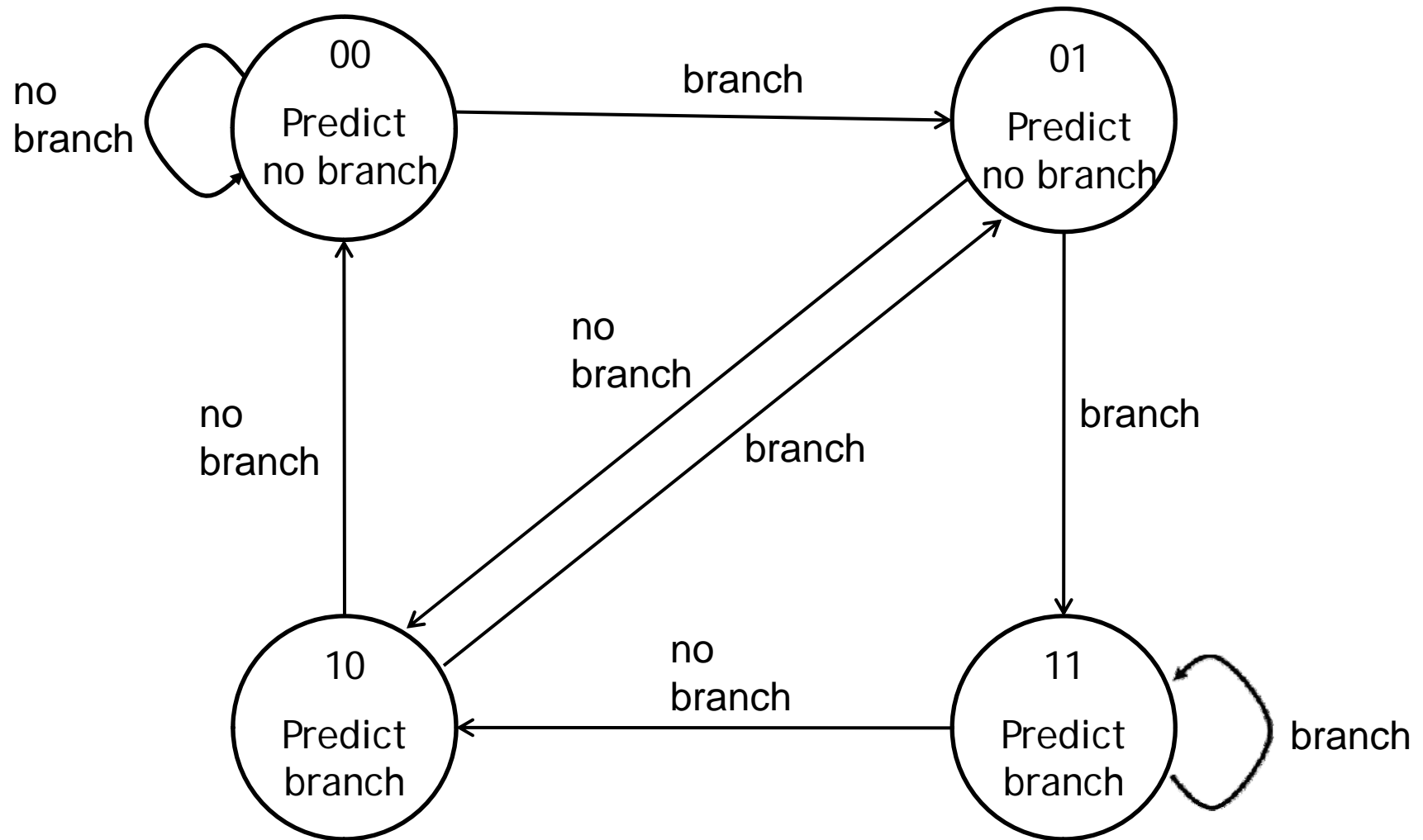
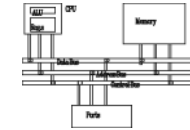
# Branch prediction



- Impact of past  $n$  branches on prediction accuracy

n	Type of mix		
	Compiler	Business	Scientific
0	64.1	64.4	70.4
1	91.9	95.2	86.6
2	93.3	96.5	90.8
3	93.7	96.6	91.0
4	94.5	96.8	91.8
5	94.7	97.0	92.0

# Branch prediction



# Multitasking

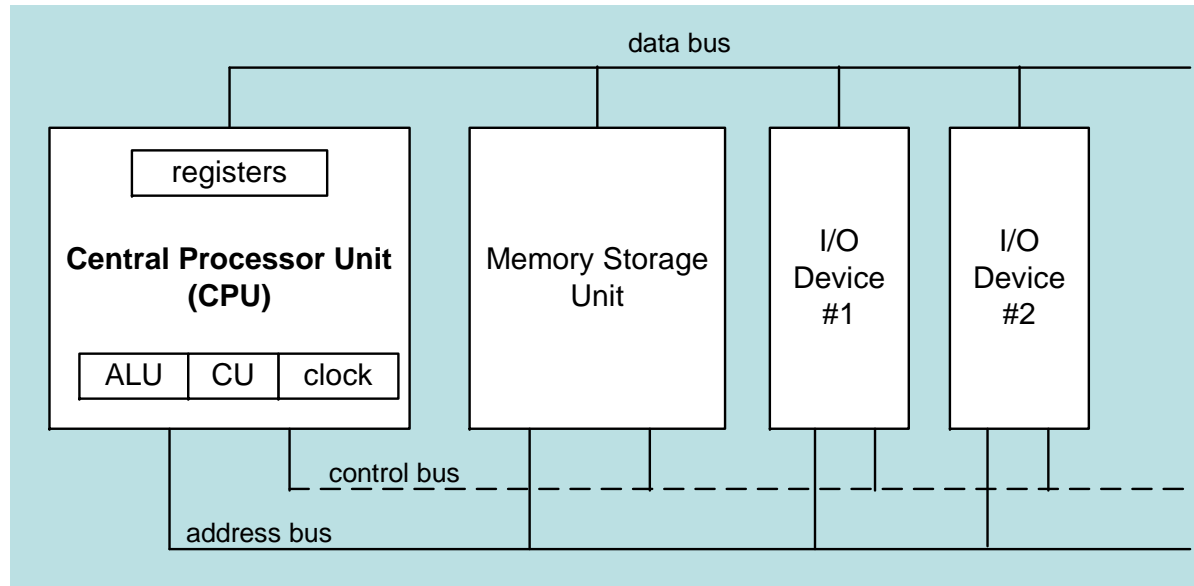
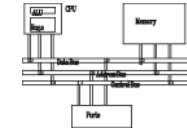
---



- OS can run multiple programs at the same time.
- Multiple threads of execution within the same program.
- Scheduler utility assigns a given amount of CPU time to each running program.
- Rapid switching of tasks
  - gives illusion that all programs are running at once
  - the processor must support task switching
  - scheduling policy, round-robin, priority

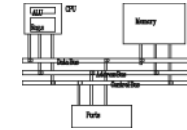
**Cache**

# SRAM vs DRAM

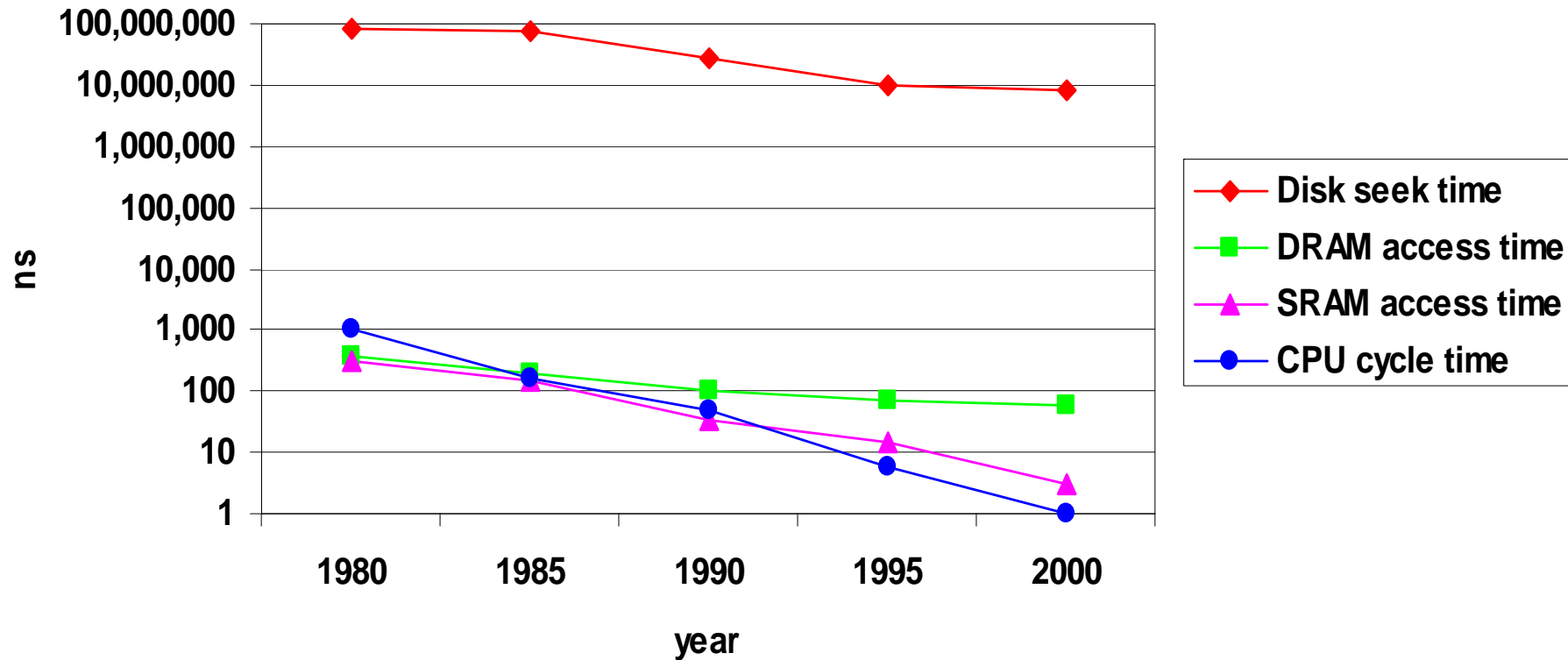


	Tran. per bit	Access time	Needs refresh?	Cost	Applications
SRAM	4 or 6	1X	No	100X	cache memories
DRAM	1	10X	Yes	1X	Main memories, frame buffers

# The CPU-Memory gap



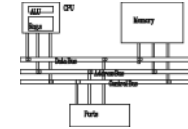
The gap widens between DRAM, disk, and CPU speeds.



	register	cache	memory	disk
Access time (cycles)	1	1-10	50-100	20,000,000

# Memory hierarchies

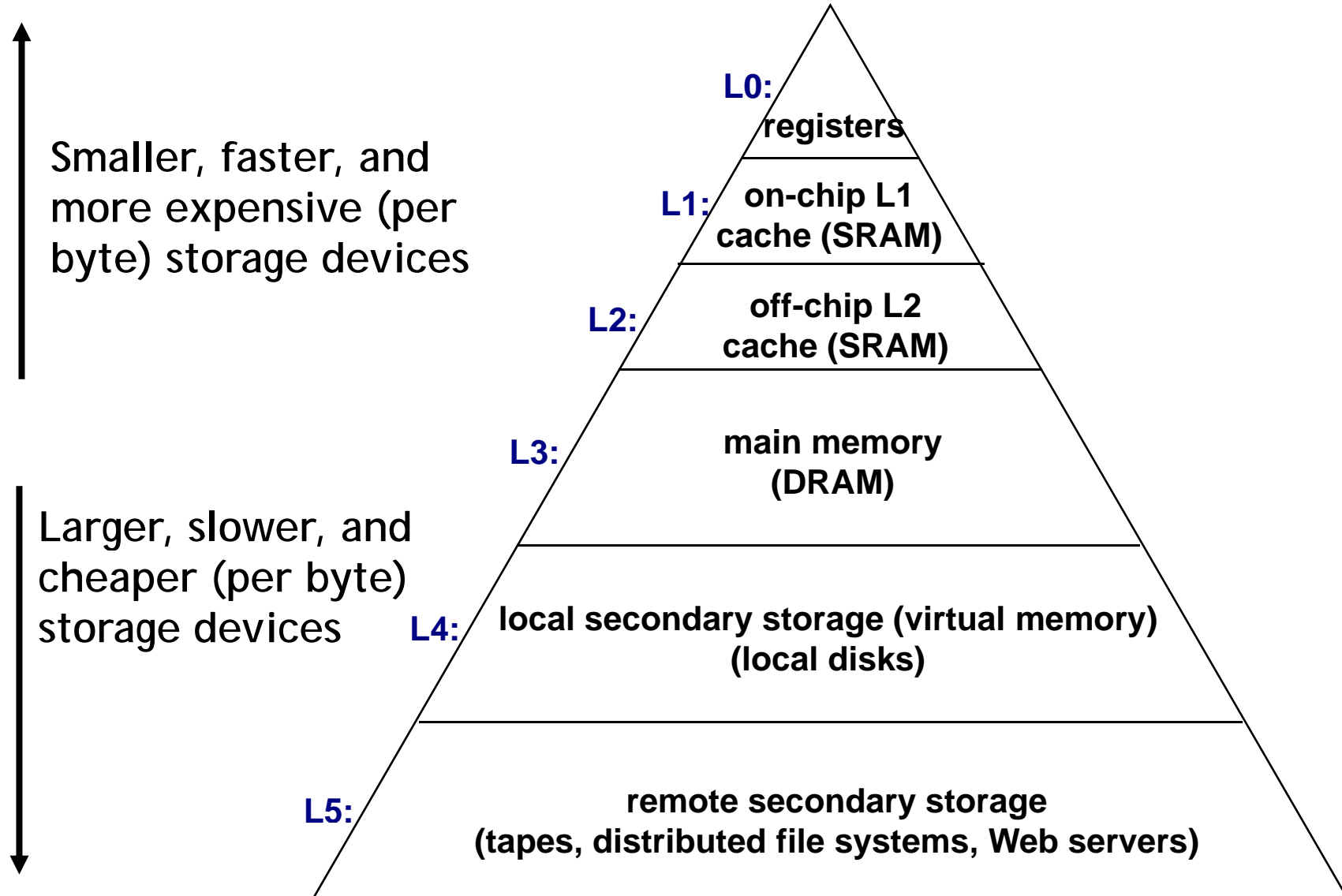
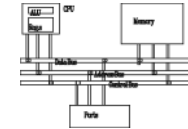
---



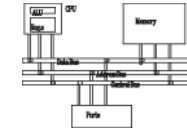
- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good **locality**.
- They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**.



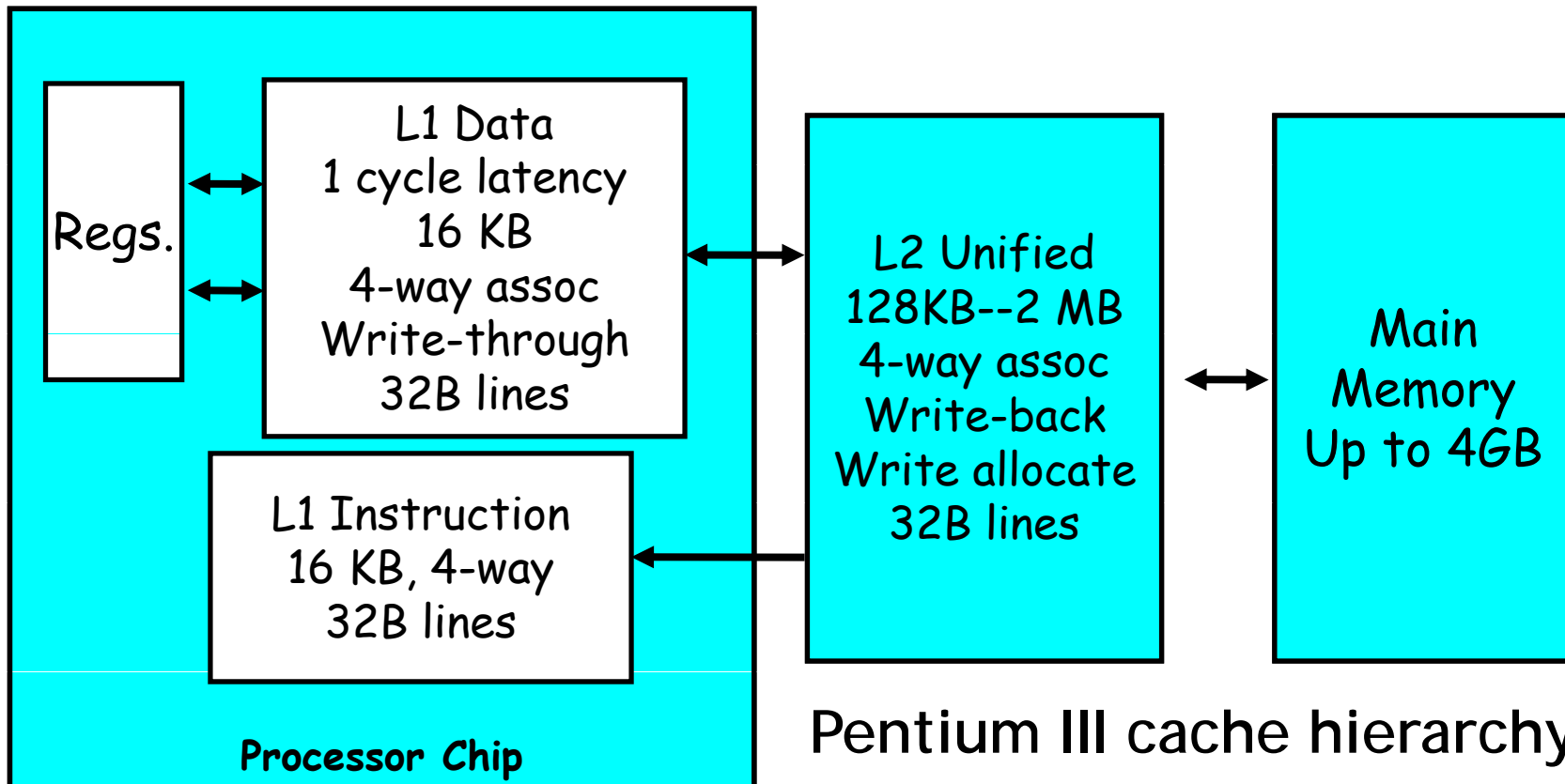
# Memory system in practice



# Reading from memory



- Multiple machine cycles are required when reading from memory, because it responds much more slowly than the CPU (e.g. 33 MHz). The wasted clock cycles are called wait states.



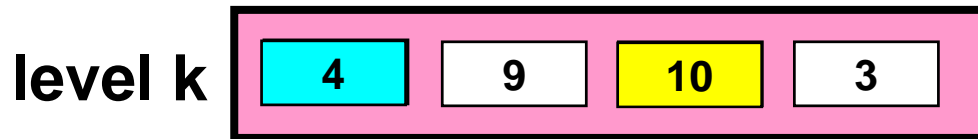
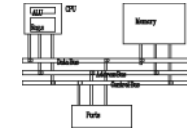
# Cache memory

---

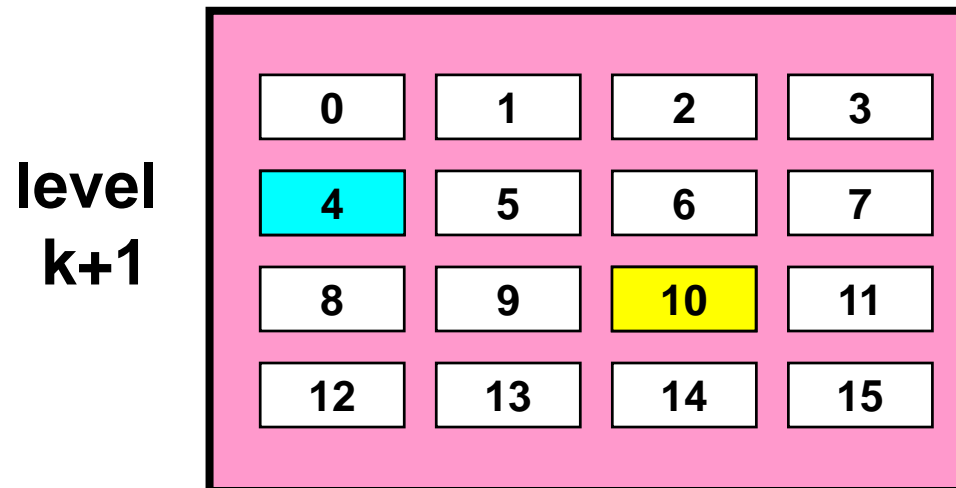


- High-speed expensive static RAM both inside and outside the CPU.
  - Level-1 cache: inside the CPU
  - Level-2 cache: outside the CPU
- Cache hit: when data to be read is already in cache memory
- Cache miss: when data to be read is not in cache memory. When? compulsory, capacity and conflict.
- Cache design: cache size, n-way, block size, replacement policy

# Caching in a memory hierarchy



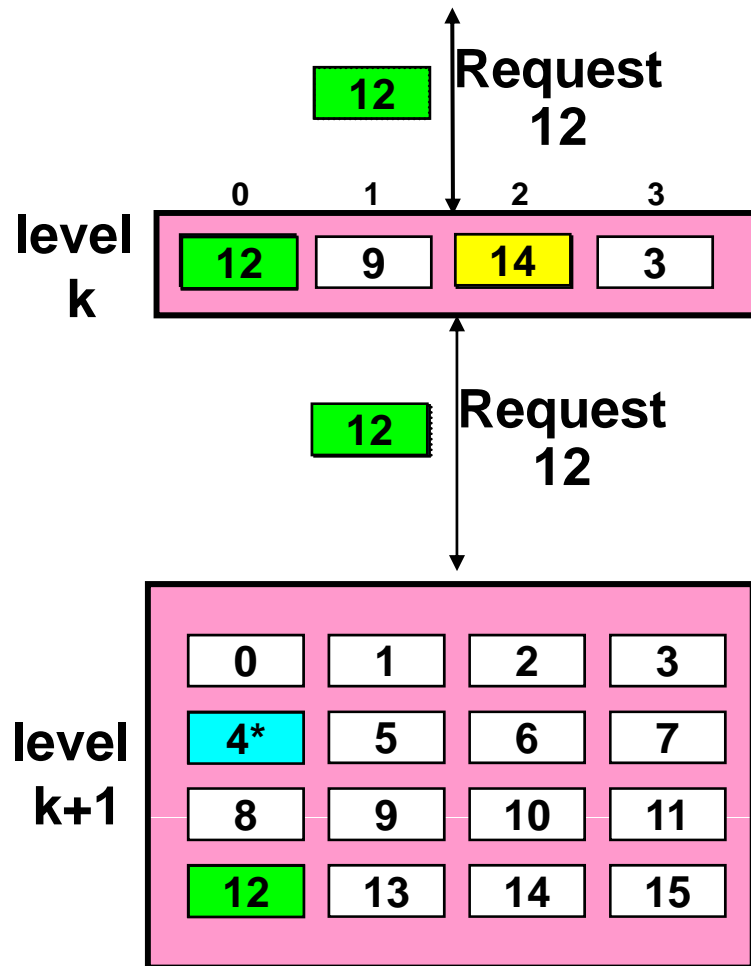
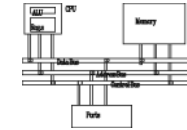
Data is copied between levels in block-sized transfer units



Smaller, faster, more Expensive device at level k caches a subset of the blocks from level k+1

Larger, slower, cheaper Storage device at level k+1 is partitioned into blocks.

# General caching concepts



- Program needs object  $d$ , which is stored in some block  $b$ .
- **Cache hit**
  - Program finds  $b$  in the cache at level  $k$ . E.g., block 14.
- **Cache miss**
  - $b$  is not at level  $k$ , so level  $k$  cache must fetch it from level  $k+1$ . E.g., block 12.
  - If level  $k$  cache is full, then some current block must be replaced (evicted). Which one is the “victim”?
    - **Placement policy:** where can the new block go? E.g.,  $b \bmod 4$
    - **Replacement policy:** which block should be evicted? E.g., LRU

# Locality

---



- Principle of Locality: programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
  - **Temporal locality:** recently referenced items are likely to be referenced in the near future.
  - **Spatial locality:** items with nearby addresses tend to be referenced close together in time.
- In general, *programs with good locality run faster than programs with poor locality*
- Locality is the reason why cache and virtual memory are designed in architecture and operating system. Another example is web browser caches recently visited webpages.

# Locality example

---



```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- Data
  - Reference array elements in succession (stride-1 reference pattern): **Spatial locality**
  - Reference sum each iteration: **Temporal locality**
- Instructions
  - Reference instructions in sequence: **Spatial locality**
  - Cycle through loop repeatedly: **Temporal locality**





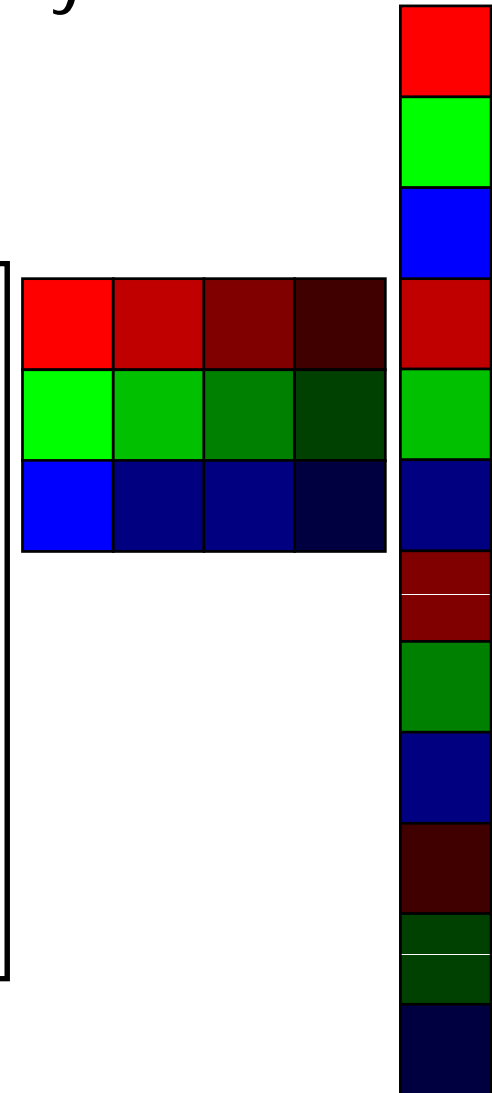
# Locality example



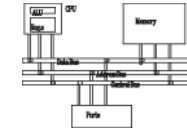
- Does this function have good locality?

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
} stride-N reference pattern
```

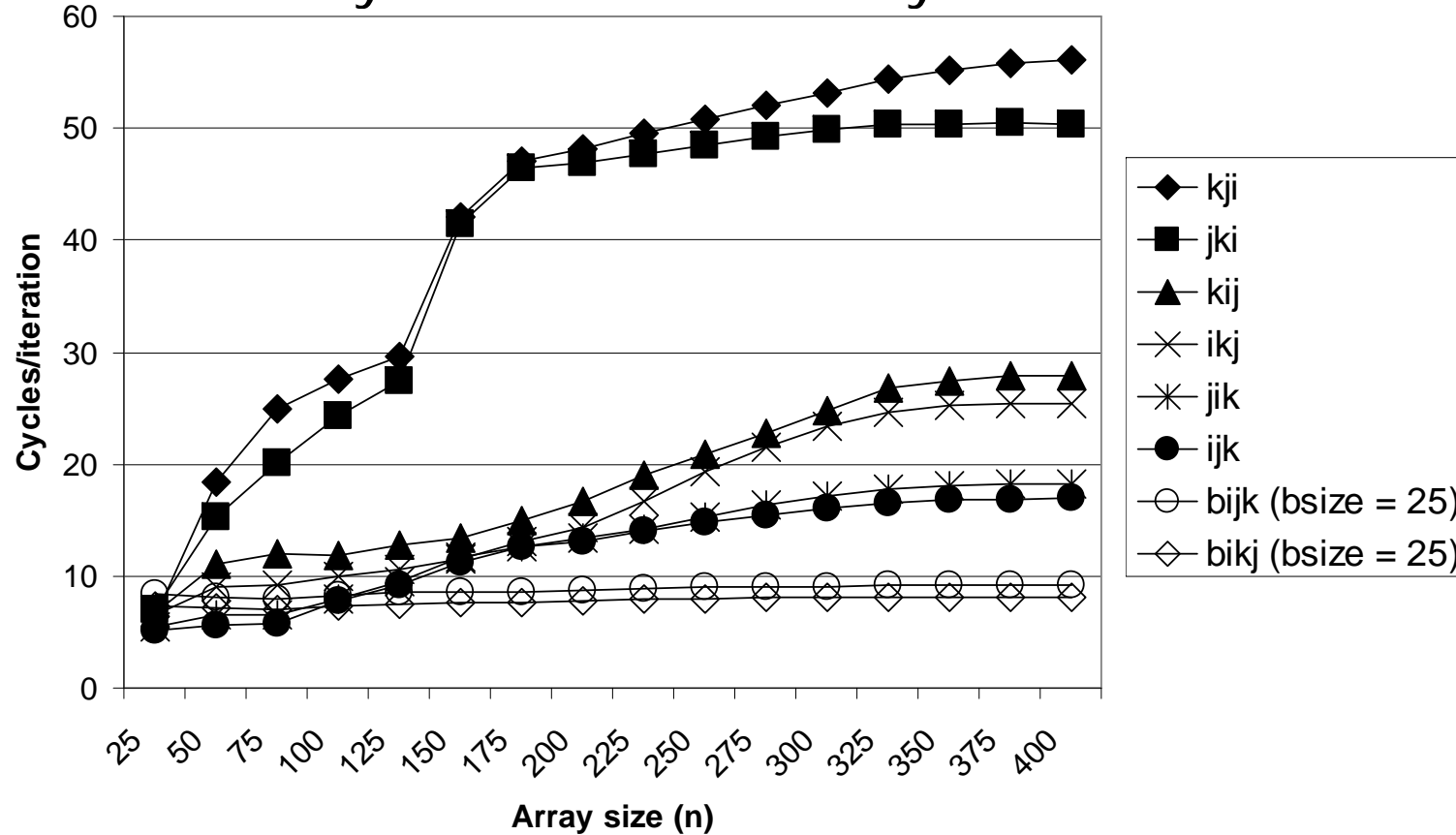


# Blocked matrix multiply performance



- Blocking (bijk and bikj) improves performance by a factor of two over unblocked versions (ijk and jik)

- relatively insensitive to array size.



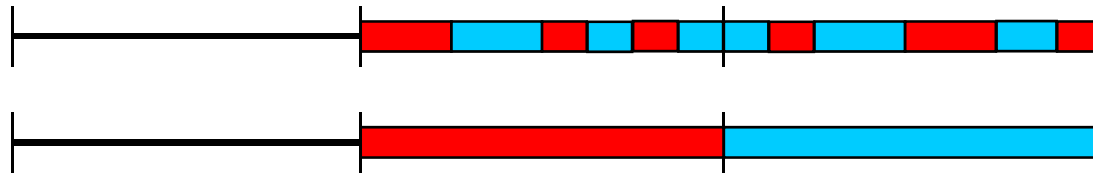
# Cache-conscious programming



- make sure that memory is cache-aligned

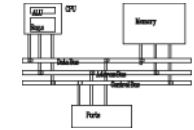


- Split data into hot and cold (list example)

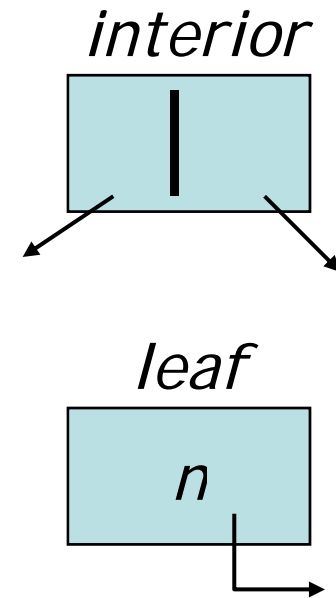


- Use union and bitfields to reduce size and increase locality

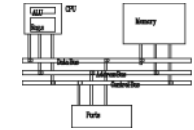
# K-d tree



```
struct KdAccelNode {  
    u_int flags;  
  
    float split; // Interior  
    u_int aboveChild; // Interior  
  
    u_int nPrims; // Leaf  
    MailboxPrim *Primitives; // Leaf  
}
```

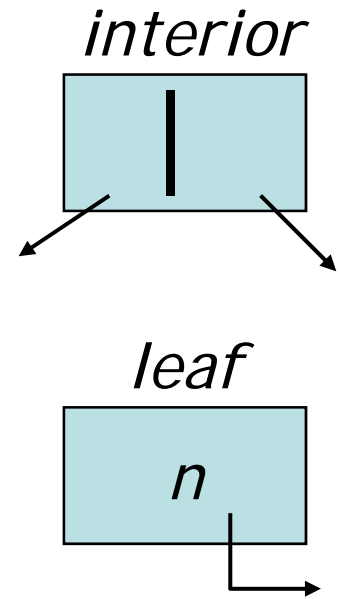


# K-d tree

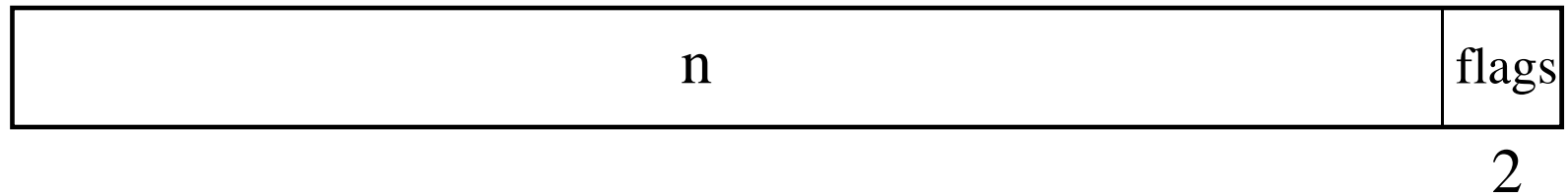
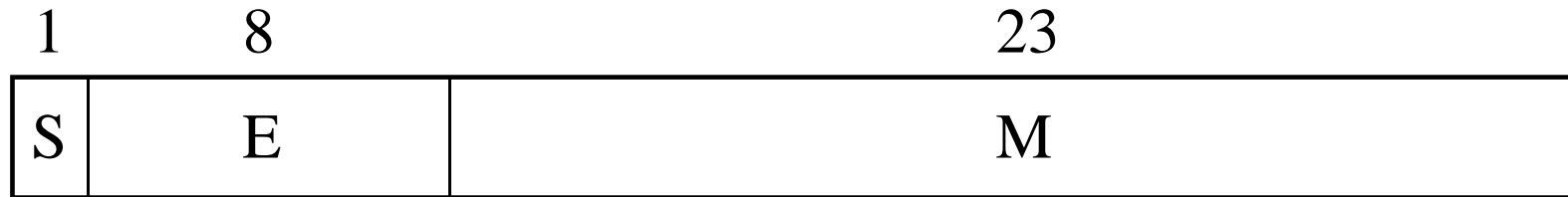
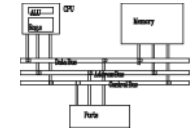


8-byte (reduced from 20-byte, 20% gain)

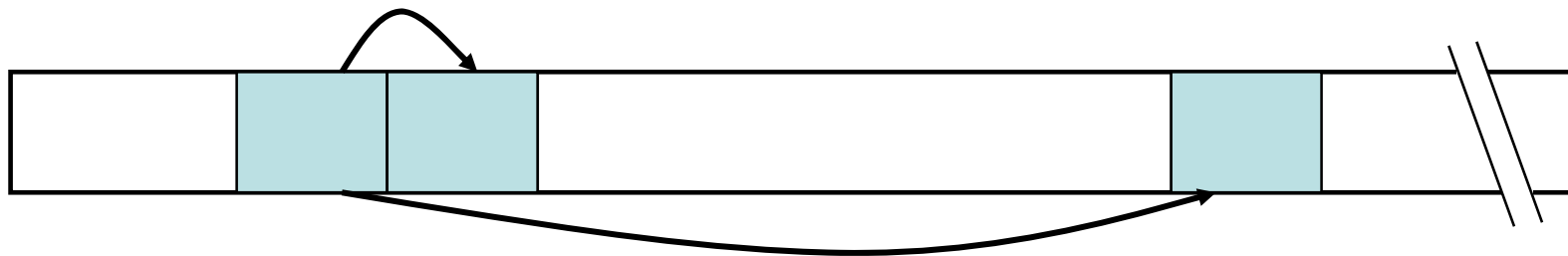
```
struct KdAccelNode {  
    ...  
    union {  
        u_int flags; // Both  
        float split; // Interior  
        u_int nPrims; // Leaf  
    };  
    union {  
        u_int aboveChild; // Interior  
        MailboxPrim **primitives; // Leaf  
    };  
}
```



# Tree representation



Flag: 0,1,2 (interior x, y, z) 3 (leaf)



# **RISC v.s. CISC**

# Trade-offs of instruction sets

---



high-level language  $\xrightarrow[\text{semantic gap}]{\text{compiler}}$  machine code  
*C, C++*  
*Lisp, Prolog, Haskell...*

- Before 1980, the trend is to increase instruction complexity (one-to-one mapping if possible) to bridge the gap. Reduce fetch from memory. Selling point: number of instructions, addressing modes. (CISC)
- 1980, RISC. Simplify and regularize instructions to introduce advanced architecture for better performance, pipeline, cache, superscalar.



# RISC



- 1980, Patterson and Ditzel (Berkeley), RISC
- Features
  - Fixed-length instructions
  - Load-store architecture
  - Register file
- Organization
  - Hard-wired logic
  - Single-cycle instruction
  - Pipeline
- Pros: small die size, short development time, high performance
- Cons: low code density, not x86 compatible

# RISC Design Principles

---



- Simple operations
  - Simple instructions that can execute in one cycle
- Register-to-register operations
  - Only load and store operations access memory
  - Rest of the operations on a register-to-register basis
- Simple addressing modes
  - A few addressing modes (1 or 2)
- Large number of registers
  - Needed to support register-to-register operations
  - Minimize the procedure call and return overhead

# RISC Design Principles

---



- Fixed-length instructions
  - Facilitates efficient instruction execution
- Simple instruction format
  - Fixed boundaries for various fields
    - opcode, source operands,...

# CISC and RISC

---



- CISC – complex instruction set
  - large instruction set
  - high-level operations (simpler for compiler?)
  - requires microcode interpreter (could take a long time)
  - examples: Intel 80x86 family
- RISC – reduced instruction set
  - small instruction set
  - simple, atomic instructions
  - directly executed by hardware very quickly
  - easier to incorporate advanced architecture design
  - examples: ARM (Advanced RISC Machines) and DEC Alpha (now Compaq), PowerPC, MIPS

# CISC and RISC



	CISC (Intel 486)	RISC (MIPS R4000)
#instructions	235	94
Addr. modes	11	1
Inst. Size (bytes)	1-12	4
GP registers	8	32

# Why RISC?

---



- Simple instructions are preferred
  - Complex instructions are mostly ignored by compilers
    - Due to semantic gap
- Simple data structures
  - Complex data structures are used relatively infrequently
  - Better to support a few simple data types efficiently
    - Synthesize complex ones
- Simple addressing modes
  - Complex addressing modes lead to variable length instructions
    - Lead to inefficient instruction decoding and scheduling

# Why RISC? (cont'd)

---



- Large register set
  - Efficient support for procedure calls and returns
    - Patterson and Sequin's study
      - Procedure call/return: 12–15% of HLL statements
        - » Constitute 31–33% of machine language instructions
        - » Generate nearly half (45%) of memory references
  - Small activation record
    - Tanenbaum's study
      - Only 1.25% of the calls have more than 6 arguments
      - More than 93% have less than 6 local scalar variables
      - Large register set can avoid memory references

# ISA design issues



# Instruction set design

---



- Issues when determining ISA
  - Instruction types
  - Number of addresses
  - Addressing modes

# Instruction types

---



- Arithmetic and logic
- Data movement
- I/O (memory-mapped, isolated I/O)
- Flow control
  - Branches (unconditional, conditional)
    - set-then-jump (**cmp AX, BX; je target**)
    - Test-and-jump (**beq r1, r2, target**)
  - Procedure calls (register-based, stack-based)
    - Pentium: **ret**; MIPS: **jr**
    - Register: faster but limited number of parameters
    - Stack: slower but more general

# Operand types

---

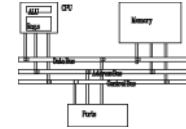


- Instructions support basic data types
  - Characters
  - Integers
  - Floating-point
- Instruction overload
  - Same instruction for different data types
  - Example: Pentium

```
mov    AL, address    ;loads an 8-bit value
mov    AX, address    ;loads a 16-bit value
mov    EAX, address   ;loads a 32-bit value
```

# Operand types

---



- Separate instructions

- Instructions specify the operand size

- Example: MIPS

<code>lb</code>	<code>Rdest, address</code>	<code>;loads a byte</code>
<code>lh</code>	<code>Rdest, address</code>	<code>;loads a halfword</code> <code>;(16 bits)</code>
<code>lw</code>	<code>Rdest, address</code>	<code>;loads a word</code> <code>;(32 bits)</code>
<code>ld</code>	<code>Rdest, address</code>	<code>;loads a doubleword</code> <code>;(64 bits)</code>

**Number of addresses**

# Number of addresses

---



- Four categories
  - 3-address machines
    - two for the source operands and one for the result
  - 2-address machines
    - One address doubles as source and result
  - 1-address machine
    - Accumulator machines
    - Accumulator is used for one source and result
  - 0-address machines
    - Stack machines
    - Operands are taken from the stack
    - Result goes onto the stack

# Number of addresses



---

Number of addresses	instruction	operation
3	OP A, B, C	$A \leftarrow B \text{ OP } C$
2	OP A, B	$A \leftarrow A \text{ OP } B$
1	OP A	$AC \leftarrow AC \text{ OP } A$
0	OP	$T \leftarrow (T-1) \text{ OP } T$

---

A, B, C: memory or register locations

AC: accumulator

T: top of stack

T-1: second element of stack

# 3-address



Example: RISC machines, TOY

$$Y = \frac{A - B}{C + (D \times E)}$$

**SUB Y, A, B ; Y = A - B**

**MUL T, D, E ; T = D × E**

**ADD T, T, C ; T = T + C**

**DIV Y, Y, T ; Y = Y / T**

opcode	A	B	C
--------	---	---	---



# 2-address



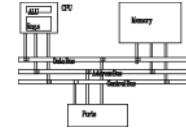
Example: IA32

```
MOV Y, A      ; Y = A
SUB Y, B      ; Y = Y - B
MOV T, D      ; T = D
MUL T, E      ; T = T × E
ADD T, C      ; T = T + C
DIV Y, T      ; Y = Y / T
```

$$Y = \frac{A - B}{C + (D \times E)}$$

opcode	A	B
--------	---	---

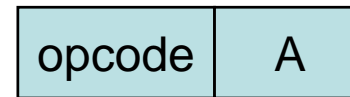
# 1-address



Example: IA32's MUL (EAX)

$$Y = \frac{A - B}{C + (D \times E)}$$

LD	D	; AC = D
MUL	E	; AC = AC × E
ADD	C	; AC = AC + C
ST	Y	; Y = AC
LD	A	; AC = A
SUB	B	; AC = AC - B
DIV	Y	; AC = AC / Y
ST	Y	; Y = AC



# 0-address



Example: IA32's FPU, HP3000

$$Y = \frac{A - B}{C + (D \times E)}$$

PUSH A	; A	opcode
PUSH B	; A, B	
SUB	; A-B	
PUSH C	; A-B, C	
PUSH D	; A-B, C, D	
PUSH E	; A-B, C, D, E	
MUL	; A-B, C, D×E	
ADD	; A-B, C+(D×E)	
DIV	; (A-B) / (C+(D×E))	
POP Y		

# Number of addresses

---



- A basic design decision; could be mixed
- Fewer addresses per instruction results in
  - a less complex processor
  - shorter instructions
  - longer and more complex programs
  - longer execution time
- The decision has impacts on register usage policy as well
  - 3-address usually means more general-purpose registers
  - 1-address usually means less

# Addressing modes

# Addressing modes

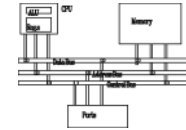
---



- How to specify location of operands? Trade-off for address range, address flexibility, number of memory references, calculation of addresses
- Operands can be in three places
  - Registers
    - Register addressing mode
  - Part of instruction
    - Constant
    - Immediate addressing mode
    - All processors support these two addressing modes
  - Memory
    - Difference between RISC and CISC
    - CISC supports a large variety of addressing modes
    - RISC follows load/store architecture

# Addressing modes

---



- Common addressing modes
  - Implied
  - Immediate (**lda R1, 1**)
  - Direct (**st R1, A**)
  - Indirect
  - Register (**add R1, R2, R3**)
  - Register indirect (**sti R1, R2**)
  - Displacement
  - Stack

# Implied addressing

---



instruction

opcode

- No address field; operand is implied by the instruction  
**CLC ; clear carry**
- A fixed and unvarying address

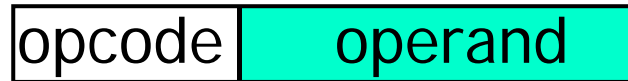


# Immediate addressing

---

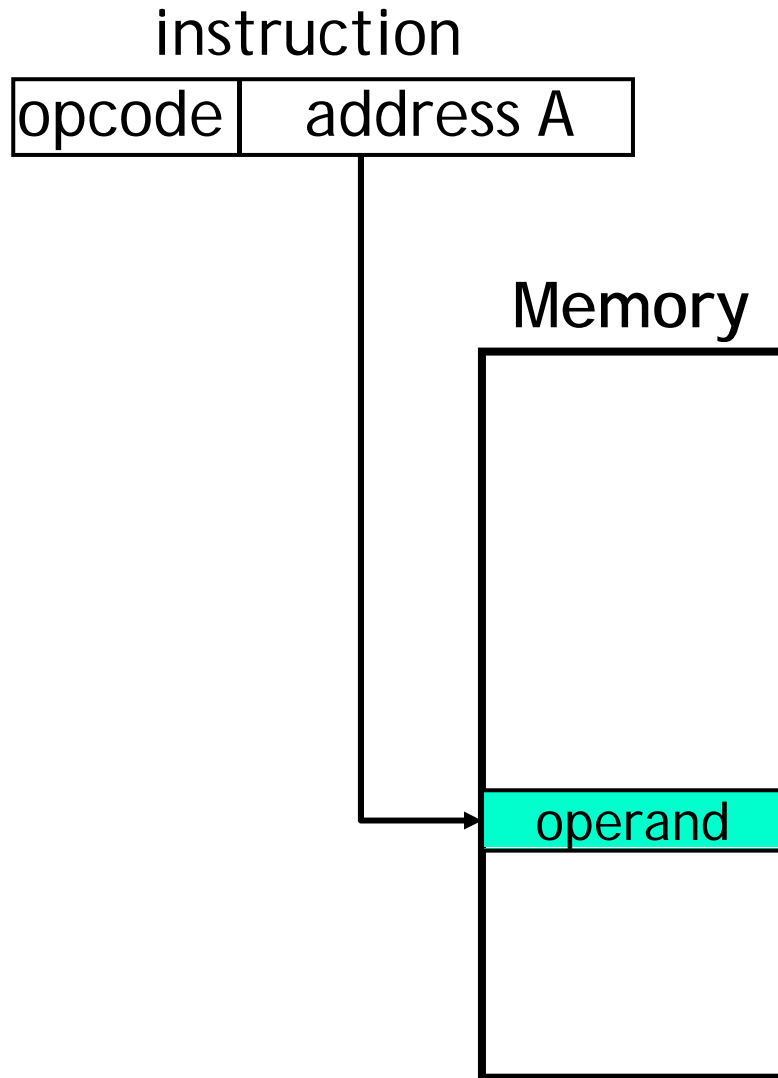


instruction



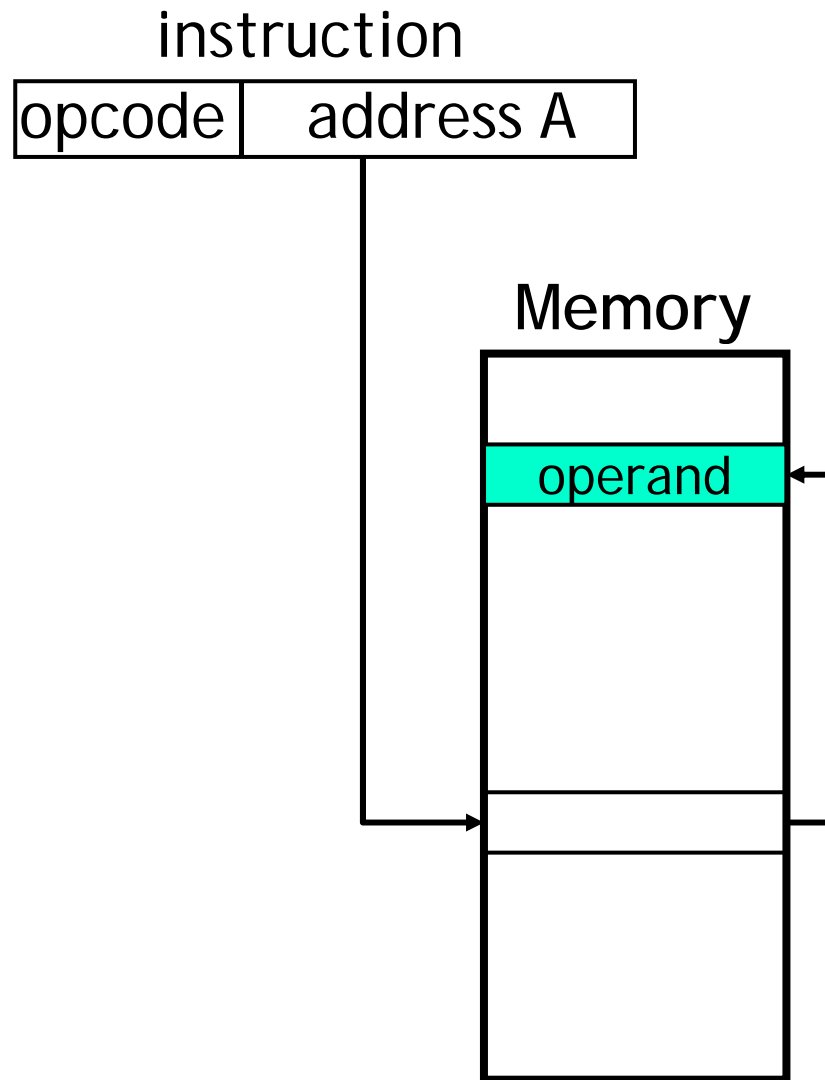
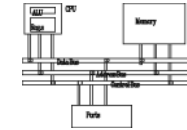
- Address field contains the operand value  
**ADD 5; AC=AC+5**
- Pros: no extra memory reference; faster
- Cons: limited range

# Direct addressing



- Address field contains the effective address of the operand  
**ADD A; AC=AC+[A]**
- single memory reference
- Pros: no additional address calculation
- Cons: limited address space

# Indirect addressing

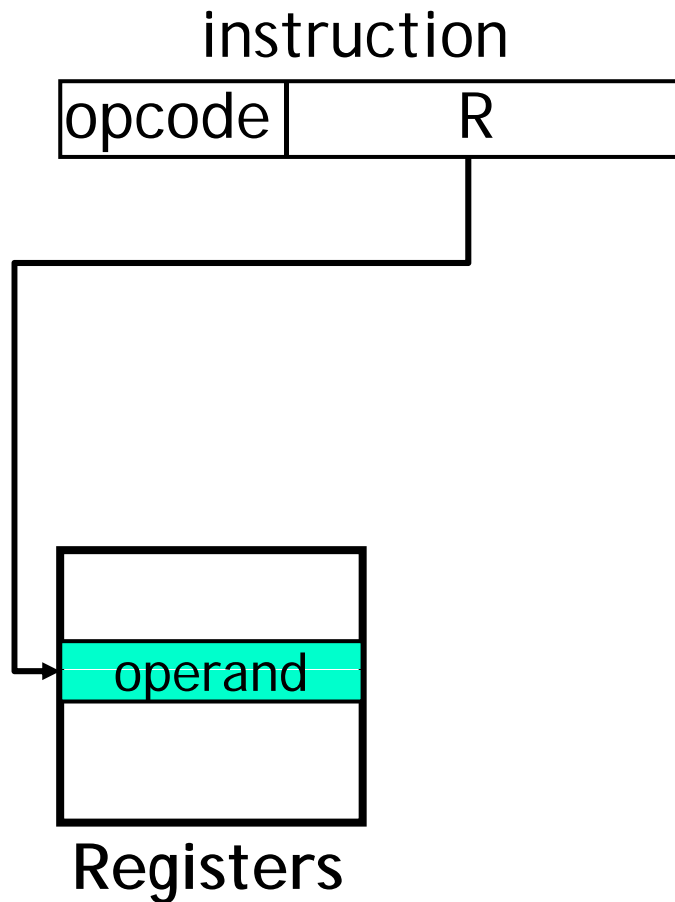


- Address field contains the address of a pointer to the operand

`ADD [A]; AC=AC+[[A]]`

- multiple memory references
- Pros: large address space
- Cons: slower

# Register addressing

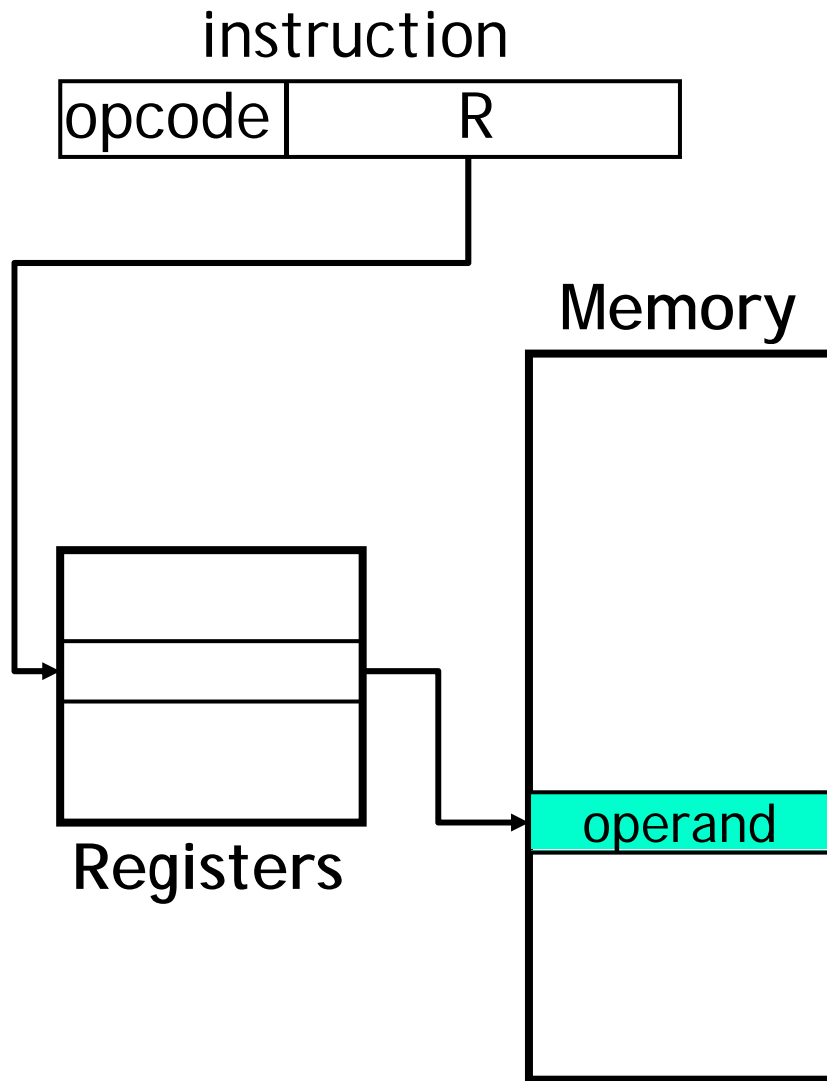


- Address field contains the address of a register

**ADD R; AC=AC+R**

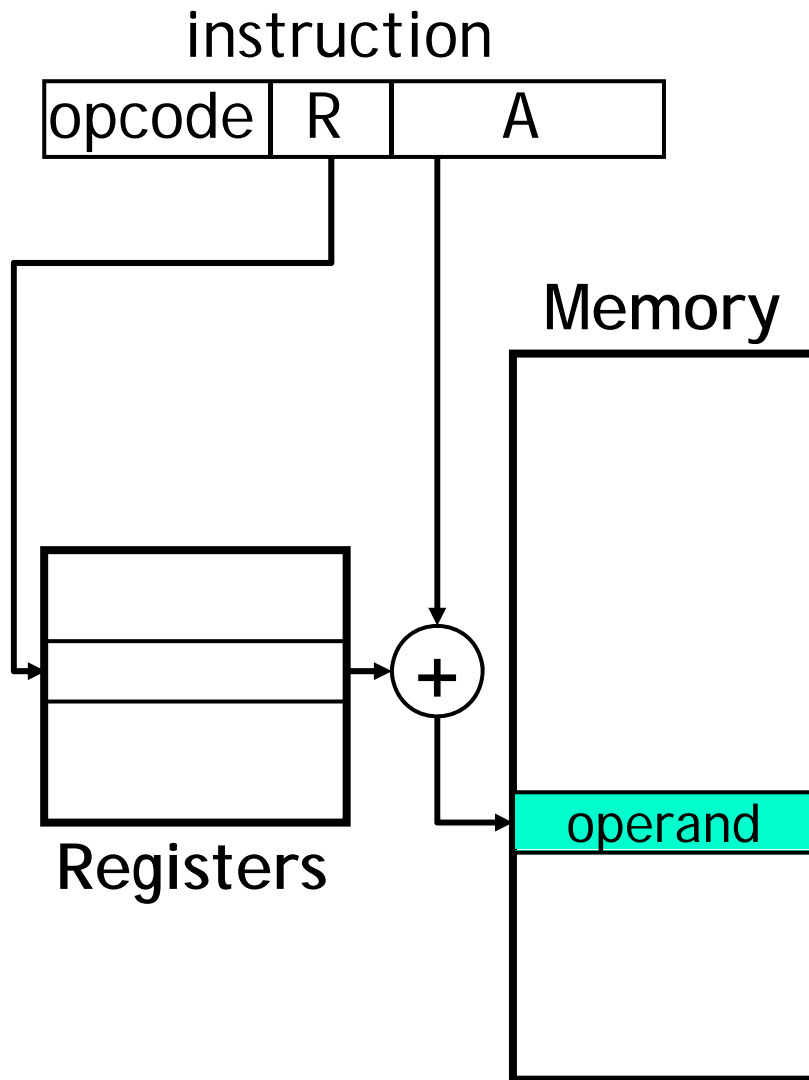
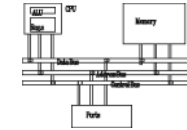
- Pros: only need a small address field; shorter instruction and faster fetch; no memory reference
- Cons: limited address space

# Register indirect addressing



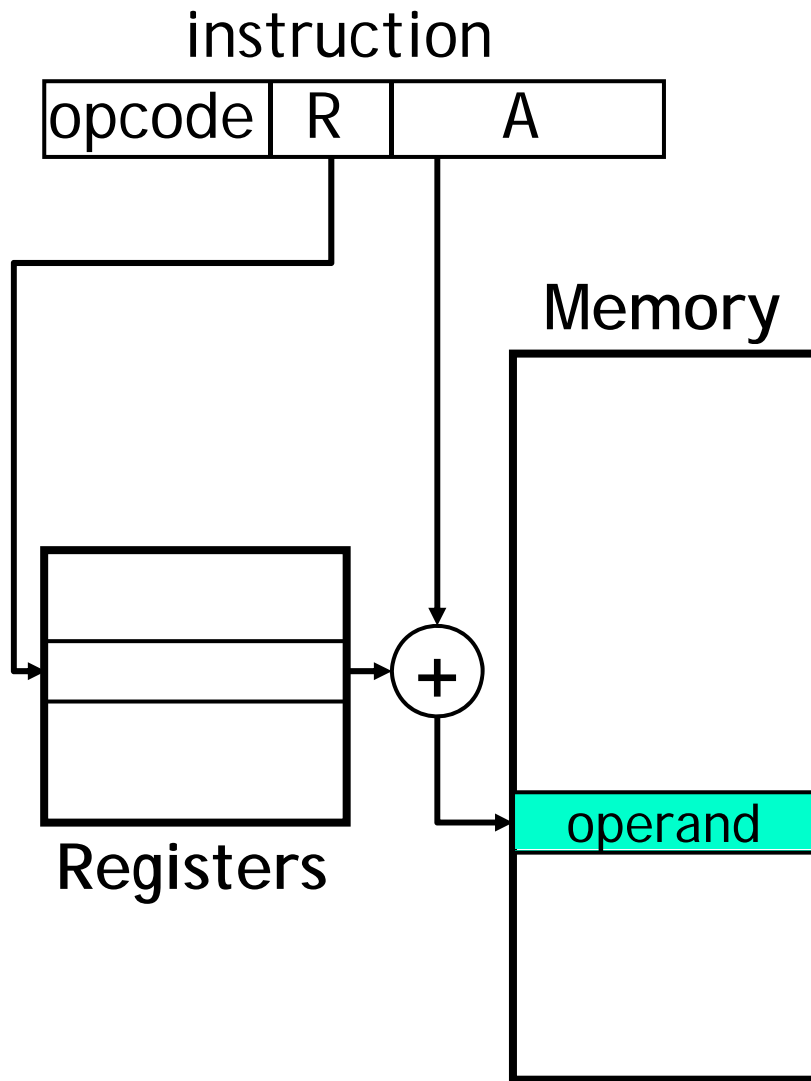
- Address field contains the address of the register containing a pointer to the operand  
**ADD [R]; AC=AC+[R]**
- Pros: large address space
- Cons: extra memory reference

# Displacement addressing



- Address field could contain a register address and an address  
`MOV EAX, [A+ESI*4]`
- $EA = A + [R \times S]$  or vice versa
- Several variants
  - Base-offset: `[EBP+8]`
  - Base-index: `[EBX+ESI]`
  - Scaled: `[T+ESI*4]`
- Pros: flexible
- Cons: complex

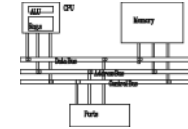
# Displacement addressing



```
MOV EAX, [A+ESI*4]
```

- Often, register, called indexing register, is used for displacement.
- Usually, a mechanism is provided to efficiently increase the indexing register.

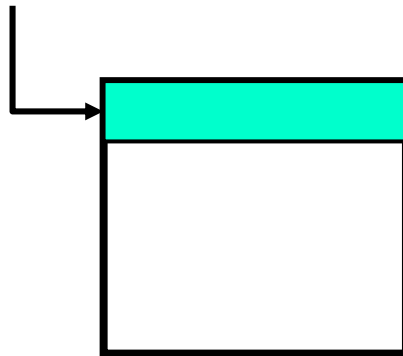
# Stack addressing



instruction

opcode

implicit



Stack

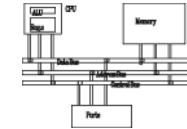
- Operand is on top of the stack

**ADD [R]; AC=AC+[R]**

- Pros: large address space
- Pros: short and fast fetch
- Cons: limited by FILO order

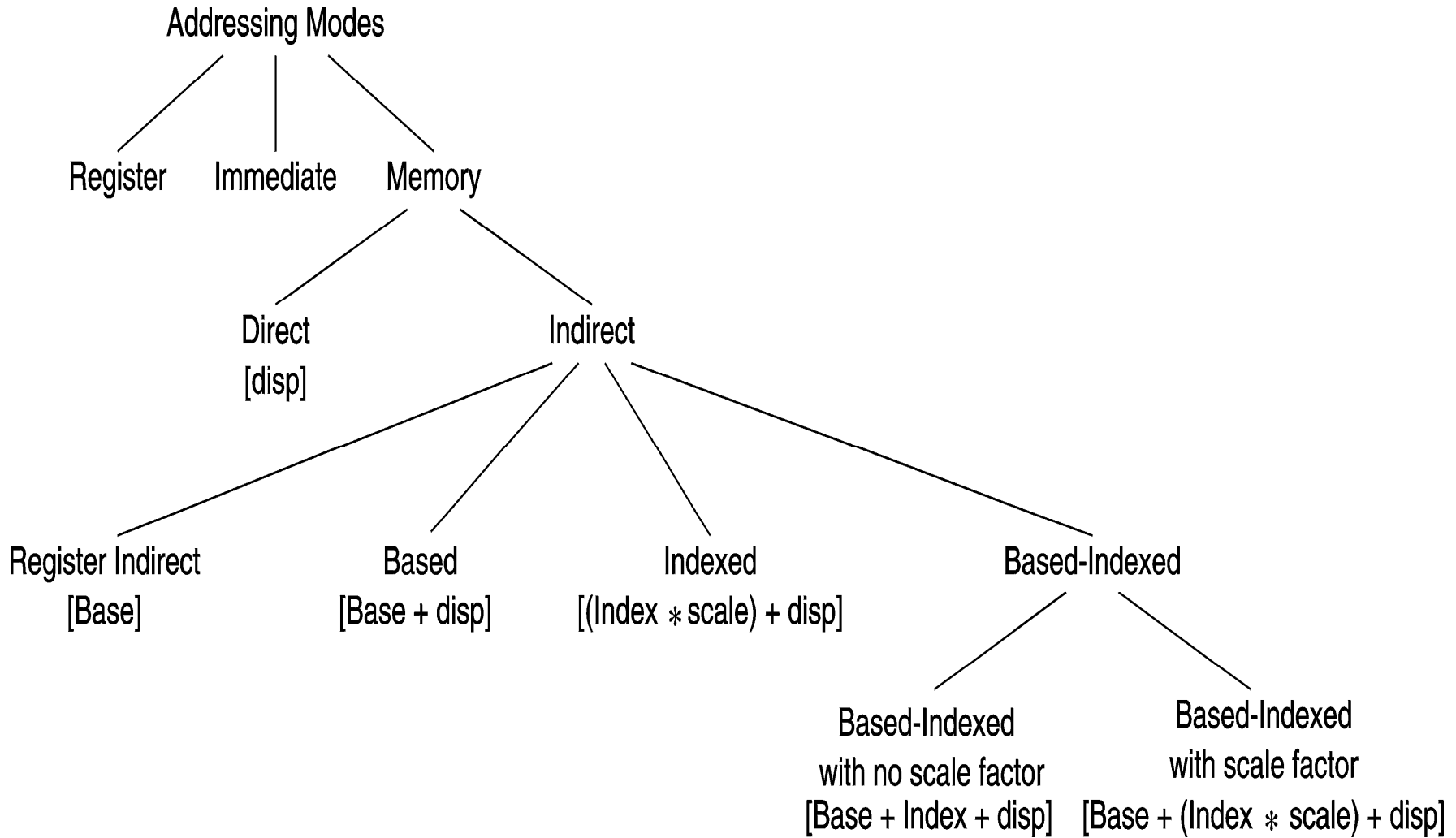


# Addressing modes

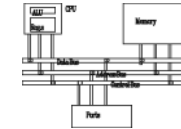


Mode	Meaning	Pros	Cons
Implied		Fast fetch	Limited instructions
Immediate	Operand=A	No memory ref	Limited operand
Direct	EA=A	Simple	Limited address space
Indirect	EA=[A]	Large address space	Multiple memory ref
Register	EA=R	No memory ref	Limited address space
Register indirect	EA=[R]	Large address space	Extra memory ref
Displacement	EA=A+[R]	Flexibility	Complexity
stack	EA=stack top	No memory ref	Limited applicability

# IA32 addressing modes

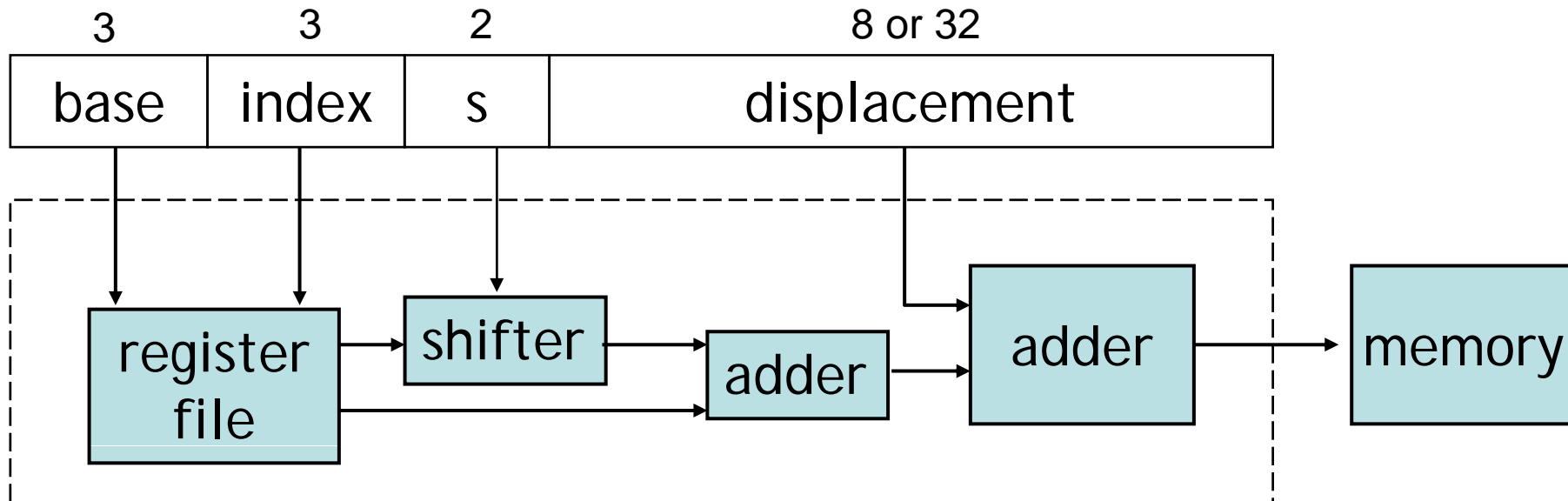


# Effective address calculation (IA32)



$$\begin{matrix}
 \text{SEGMENT} & + & \text{BASE} & + & \langle \text{INDEX} * \text{SCALE} \rangle & + & \text{DISPLACEMENT} \\
 \left[ \begin{matrix} \text{CS} \\ \text{SS} \\ \text{DS} \\ \text{ES} \\ \text{FS} \\ \text{GS} \end{matrix} \right] & + & \left[ \begin{matrix} \text{EAX} \\ \text{ECX} \\ \text{EDX} \\ \text{EBX} \\ \text{ESP} \\ \text{EBP} \\ \text{ESI} \\ \text{EDI} \end{matrix} \right] & + & \left[ \begin{matrix} \text{EAX} \\ \text{ECX} \\ \text{EDX} \\ \text{EBX} \\ \text{EBP} \\ \text{ESI} \\ \text{EDI} \end{matrix} \right] & * & \left[ \begin{matrix} 1 \\ 2 \\ 4 \\ 8 \end{matrix} \right] & + & \left[ \begin{matrix} \text{NO DISPLACEMENT} \\ \text{8-BIT DISPLACEMENT} \\ \text{32-BIT DISPLACEMENT} \end{matrix} \right]
 \end{matrix}$$

A dummy format for one operand

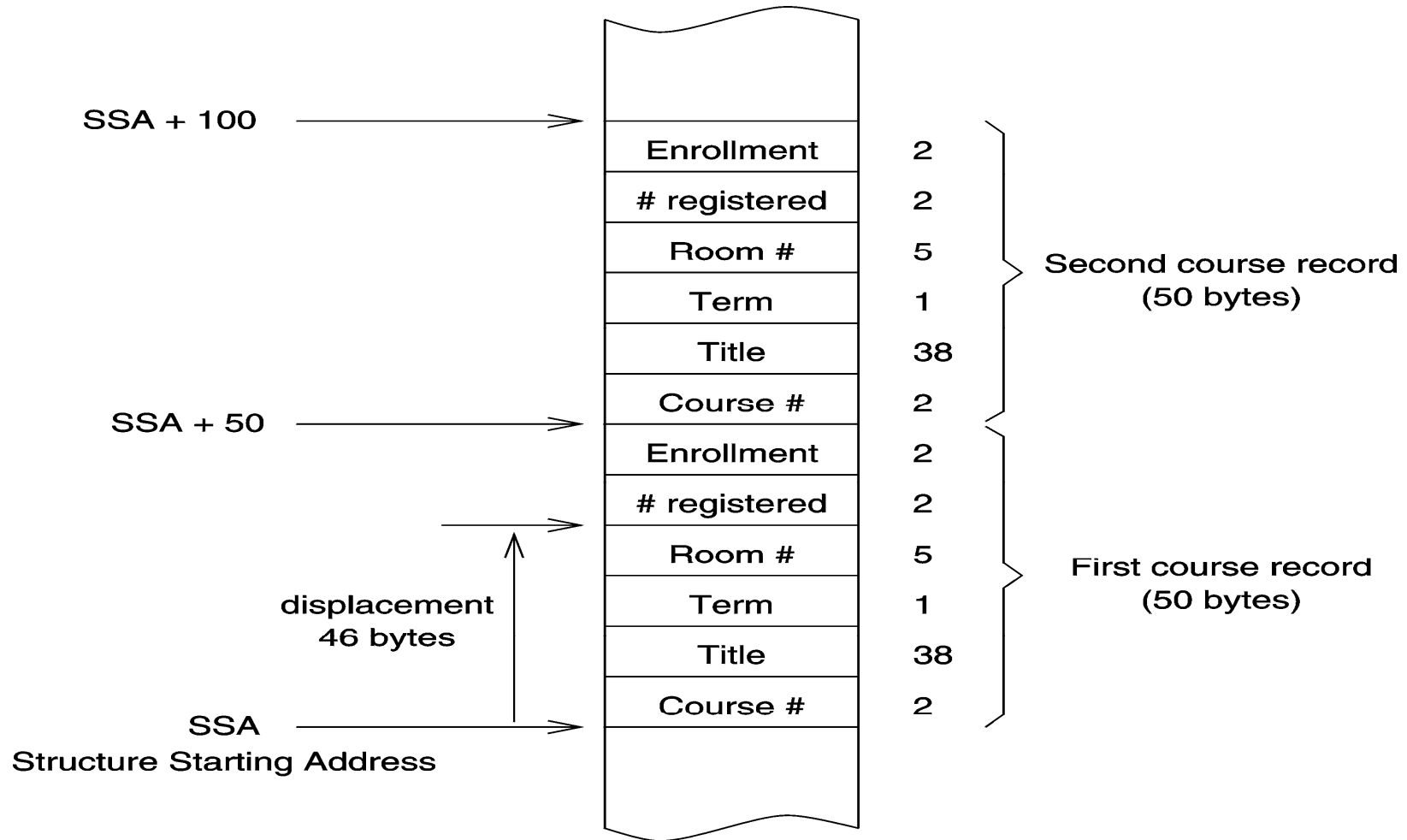
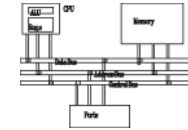


# Based Addressing



- Effective address is computed as  
base + signed displacement
  - Displacement:
    - 16-bit addresses: 8- or 16-bit number
    - 32-bit addresses: 8- or 32-bit number
- Useful to access fields of a structure or record
  - Base register → points to the base address of the structure
  - Displacement → relative offset within the structure
- Useful to access arrays whose element size is not 2, 4, or 8 bytes
  - Displacement → points to the beginning of the array
  - Base register → relative offset of an element within the array

# Based Addressing

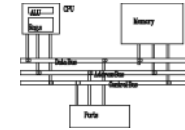


# Indexed Addressing



- Effective address is computed as  
(index \* scale factor) + signed displacement
  - 16-bit addresses:
    - displacement: 8- or 16-bit number
    - scale factor: none (i.e., 1)
  - 32-bit addresses:
    - displacement: 8- or 32-bit number
    - scale factor: 2, 4, or 8
- Useful to access elements of an array (particularly if the element size is 2, 4, or 8 bytes)
  - Displacement → points to the beginning of the array
  - Index register → selects an element of the array (array index)
  - Scaling factor → size of the array element

# Indexed Addressing



## Examples

```
add AX, [DI+20]
```

- We have seen similar usage to access parameters off the stack

```
add AX,marks_table[ESI*4]
```

- Assembler replaces `marks_table` by a constant (i.e., supplies the displacement)
- Each element of `marks_table` takes 4 bytes (the scale factor value)
- ESI needs to hold the element subscript value

```
add AX,table1[SI]
```

- SI needs to hold the element offset in *bytes*
- When we use the scale factor we avoid such byte counting

# Based-Indexed Addressing

---



## Based-indexed addressing with no scale factor

- Effective address is computed as  
base + index + signed displacement
- Useful in accessing two-dimensional arrays
  - Displacement → points to the beginning of the array
  - Base and index registers point to a row and an element within that row
- Useful in accessing arrays of records
  - Displacement → represents the offset of a field in a record
  - Base and index registers hold a pointer to the base of the array and the offset of an element relative to the base of the array



# Based-Indexed Addressing



- Useful in accessing arrays passed on to a procedure
  - Base register → points to the beginning of the array
  - Index register → represents the offset of an element relative to the base of the array

## Example

Assuming BX points to **table1**

```
mov AX, [BX+SI]
```

```
cmp AX, [BX+SI+2]
```

compares two successive elements of **table1**

# Based-Indexed Addressing

---



## Based-indexed addressing with scale factor

- Effective address is computed as  
$$\text{base} + (\text{index} * \text{scale factor}) + \text{signed displacement}$$
- Useful in accessing two-dimensional arrays when the element size is 2, 4, or 8 bytes
  - Displacement ==> points to the beginning of the array
  - Base register ==> holds offset to a row (relative to start of array)
  - Index register ==> selects an element of the row
  - Scaling factor ==> size of the array element