Computer Organization (CS 1410) Lecture 24 Name :_____

1) Is the operating system hardware or software?

2) What are the functions/goals/role of the operating system?

3) On a single core CPU system, how can several programs be running simultaneously?

4) If the operating system kernel and several user programs are loaded into memory and running concurrently, what prevents one user program (say program A) from accessing (reading or writing) another program's data in memory?

5) How is a computer system protected from a user program that goes into an infinite loop?

6) Modern CPU's have *dual-modes of operation* where there are two (or more) modes of operation: user mode and system(/supervisor/monitor) mode. A mode-bit(s) within the CPU's *processor-status-word* (PSW) register is used to indicate whether the CPU is executing in user or system mode. The set of all machine-language instructions are divided into:

- ***privileged instructions* that can only be executed in system mode, and**
- **non-privileged instructions that can be executed in any mode of operation.**

Every time an instruction is executed by the CPU, the hardware checks to see if the instruction is privileged and whether the mode is user. Whenever this case is detected, an exception (internal interrupt) is generated that turns CPU control back over to the operating system.

Can you think of some privileged instructions related to the answer to questions (4) and (5) above?

7) In a multi-user system user A's program needs to access (R/W) data files on the harddisk. What prevents user A's program from accessing user B's (or the OS's) files?

8) Assume special I/O instructions are used to fill I/O-controller registers. Why can't a user program use these instructions to communicate with the I/O device directly and "by-pass" the operating system's protection checking?

9) Assume that memory-mapped I/O is used. Since Load and Store instructions are used to communicate with the I/O-controller registers, why can't a user program communicate with the I/O device directly and "by-pass" the operating system's protection checking?

10) What are some common I/O devices on the computer system? (arrange them by their speed)

11) Suppose we had a block transfer from an I/O device to memory. The block consists of 1024 words and one word can be transferred at a time. For each of the following, indicate the number of interrupts needed to transfer a block:

a) programmed-I/O

b) interrupt-driven I/O

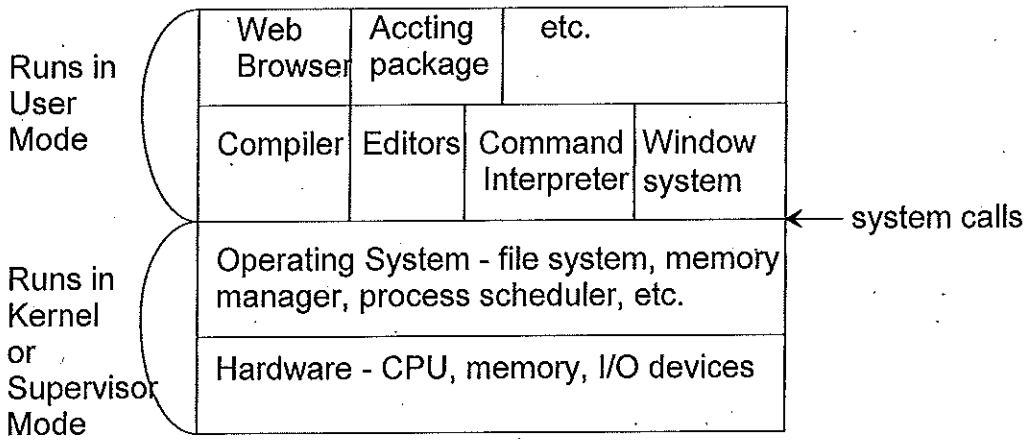
c) DMA (direct-memory access)

12) What is the main difference between programmed I/O and interrupt-driven I/O?

13) What is the main difference between interrupt-driven I/O and DMA?

What is an operating system (OS)?

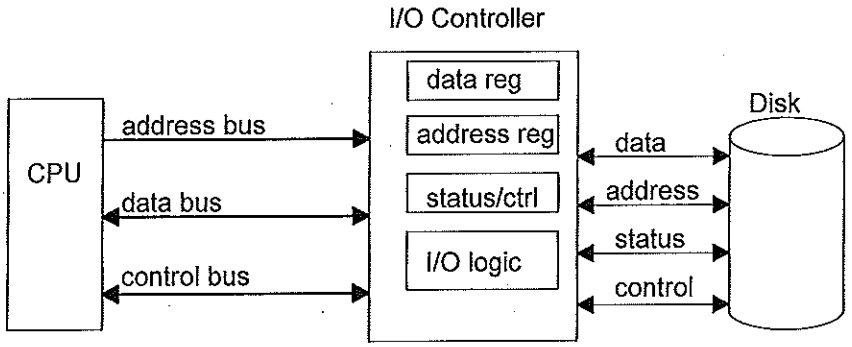
- A program that operates as the interface between the user and the hardware



Goals of OS

1. Make computer convenient to use by providing a virtual/extended machine that is easier to use and program than the underlying hardware,

   e.g., writing/reading to file on disk



OS provides high-level system calls so programmer does not need to know the details of the disk

2. Manage computer resources/hardware efficiently
   Resources - processor(s), memory, timers, disks, network

Resources are competed for by all running programs

Examples:
- which programs are loaded in limited memory
- restricts access to memory used by other programs and the operating system
- which program can run on the CPU

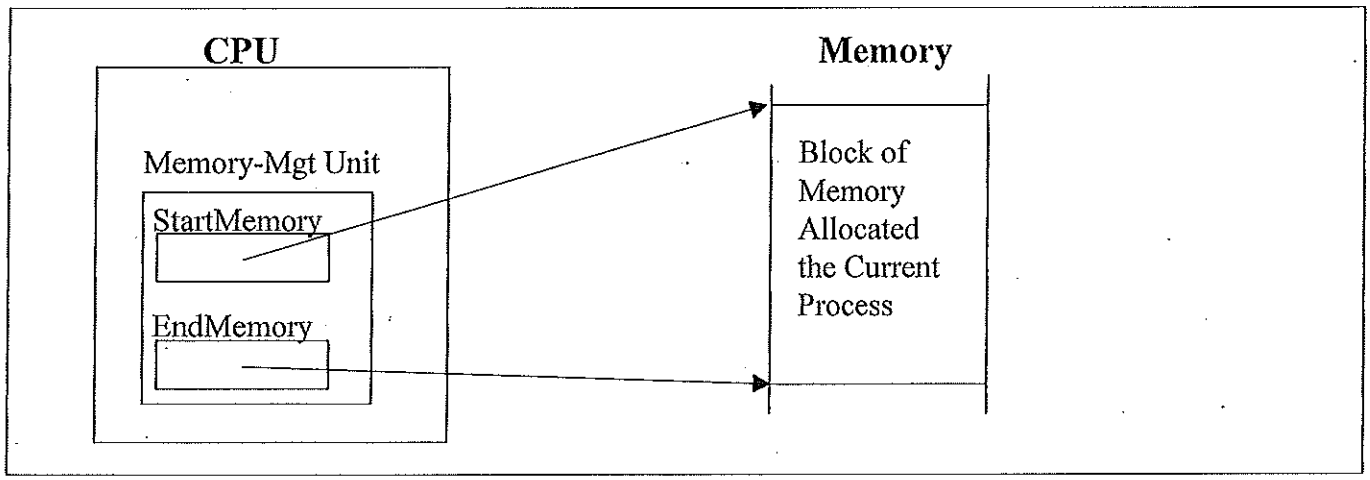
We can view the OS as resource manager that is responsible for resource allocation, tracking resources, accounting, and mediating conflicting requests

Hardware support for Operating Systems

Need protection from user programs that:
1. access memory of other programs or the OS
2. go into an infinite loop
3. access files of other programs

Protection Techniques

1) **Restrict a user program to its allocated address space in memory.** In a simple computer, a user program might be allocated a single continguous address space in memory. The two special purpose CPU registers: StartMemory and EndMemory can bracket the user program's address space. All memory addresses that the user program performs can be checked by hardware in the CPU to make sure that they fall between the values in these registers. If the user program tries to access memory outside the range of addresses indicated by these registers, an interrupt/exception is raised to return control back to the operating system. On more complex computers, a memory-management unit (MMU) provides a more sophicated address mapping scheme (paging, segmentation, paged segments, none). Modifications to the memory-management registers are privileged.



2) **Dual-Mode Operation** - the CPU has two (or more) modes of operation: user mode and system(/supervisor/monitor/privileged) mode with some privileged (machine/assembly language) instructions only executable in system mode. A mode-bit within the CPU's *processor-status-word* (PSW) register is used to indicate whether the CPU is executing in user or system mode. The set of all machine-language instructions are divided into:
   a) privileged instructions that can only be executed in system mode, and
   b) non-privileged instructions that can be executed in any mode of operation.

   Every time an instruction is executed by the CPU, the control-unit hardware checks to see if the instruction is privileged and whether the mode is user. Whenever this case is detected, an exception (internal interrupt) is generated that turns CPU control back over to the operating system.

3) **CPU Timer** - the operating system sets a count-down timer before turning control over to a user program. If the timer expires, it generates an interrupt a user pgm before the user pgm is started. Remember that only one program (in a single CPU system) can be executing at a time so when the OS turns control over to a user program it has "lost control."

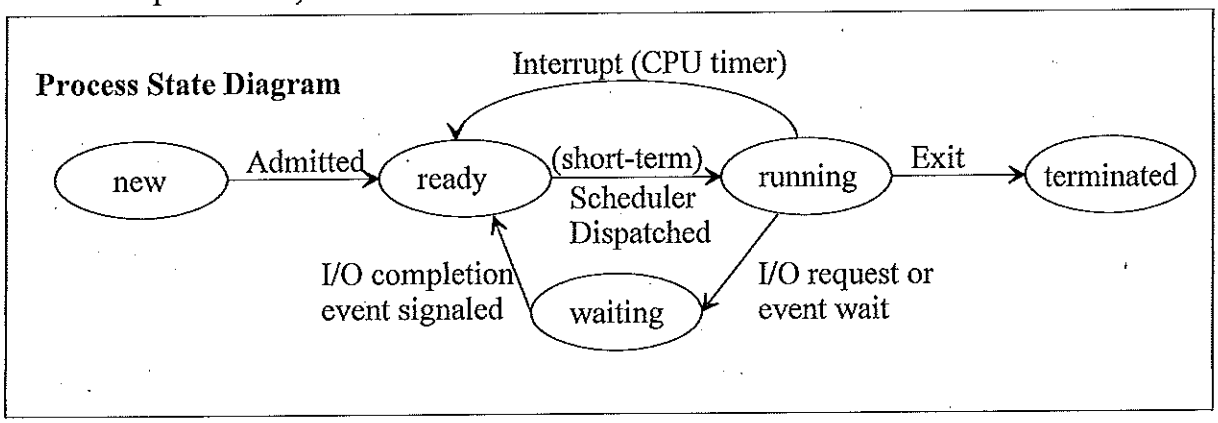
   Modifications to the CPU timer are privileged

4) Protection to restrict a process from access files of other programs varies depending on whether the computer is using memory-mapped I/O or instruction-based I/O. If memory-mapped I/O is being used, the memory address associated with the external device I/O registers are outside of the process accessable memory address space. Thus, our solution (3) above is enough to force a process to request I/O through operating system calls.

   If instruction-based I/O instructions array being used. I/O has a separate address space from memory, but we can make these I/O instructions privileged so they can only be executed in system mode. Thus, a process could not execute them directly.

OS manages processes (running programs):

A *process* is the term for a running program. A process's state consists of the CPU register values, its run-time stack in memory, and it's other memory content. Many processes may be executing concurrently, but only one can be executing on a CPU at a time. When the CPU switches to another process, a *context switch* occurs which involves saving the complete state of the previously executing process before loading the state of the next process to execute on the CPU. Depending on the hardware, this can take up to 100 microseconds (i.e., very slow in computer terms).



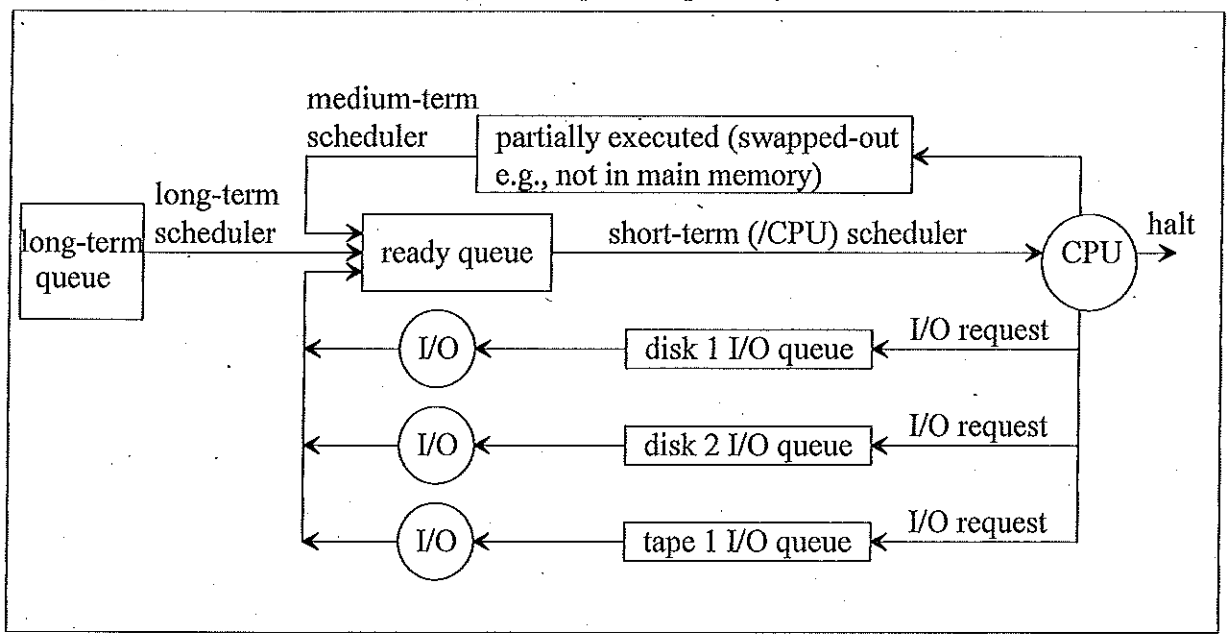
Queues are used to hold *process control blocks (PCB)* that represent processes internally to the OS.

Process Control Block

Next PCB in queue pointer
Process State
Program Counter
Registers
Memory Mgt. Info
CPU Scheduling Info.
Accounting Info.
I/O Status Info

OS maintains queues and does scheduling:

The PCB for a process moves around from queue to queue depending on its state.



I/O queues - since I/O is so slow, several programs might have outstanding requests to use an I/O device so a queue for each I/O device is necessary.

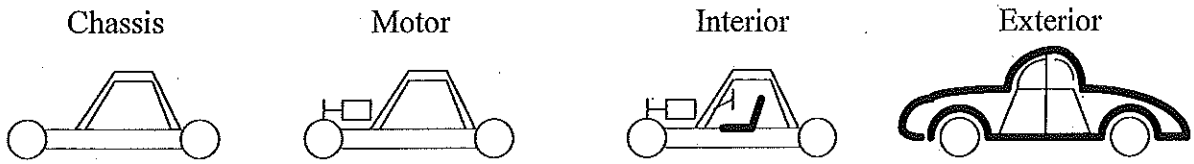
Ready (Short-term) queue - programs that are in memory and ready to execute. All they need is the CPU to run.

Medium-term queue - programs that are partially executed, but have been swapped out of memory to disk

Long-term queue - user has requested the a program be executed, but it has not yet been loaded into memory

Computer Org. (CS 1410) Lecture 25 Name: _____

1. Assume that an automobile assembly process takes 4 hours.



a) If the stages take the following amounts of time, then what is the time between completions of automobiles?
Chassis 1 hour
Motor 1 hour
Interior 1 hour
Exterior 1 hour

b) If the stages take the following amounts of time, then what is the time between completions of automobiles?
Chassis 45 minutes
Motor 1 hour
Interior 1 hour and 15 minutes
Exterior 1 hour

2. We could follow the instruction/machine cycle into stages for instruction pipelined.

- **Fetch Instruction - read instruction pointed at by the program counter (PC) from memory into Instr. Reg. (IR)**
- **Decode Instruction - figure out what kind of instruction was read**
- **Fetch Operands - get operand values from the memory or registers**
- **Execute Instruction - do some operation with the operands to get some result**
- **Write Result - put the result into a register or in a memory location**

Two approaches for designing a computer is CISC (Complex Instr. Set Computer - pre-1980) and RISC (Reduced Instruction Set Computer post 1985, MIPS was one of the first commerical RISC processor). A CISC philosophy was to make assembly language (AL) as much like a high-level language (HLL) as possible to reduce the "*semantic gap*" between AL and HLL. The rational for CISC at the time was to:

- **reduce compiler complexity and aid assembly language programming. Compilers were not too good during the 50's to 70's, (e.g., they made poor use of general purpose registers so code was inefficient) so some programs were written in assembly language.**
- **reduce the program size. More powerful/complex instructions reduced the number of instructions necessary in a program. Memory during the 50's to 70's was limited and expensive.**
- **improve code efficiency by allowing complex sequence of instructions to be implemented in microcode. For example, the Digital Equipment Corporation (DEC) VAX computer had an assembly-language instruction "MATCHC *substrLength, substr, strLength, str*" that looks for a substring within a string.**

The architectural characteristics of CISC machines include:

- **complex, high-level like AL instructions**
- **variable format machine-language instructions that execute using a variable number of clock cycles**
- **many addressing modes (e.g., the DEC VAX had 22 addressing modes)**

a) Why are complex instructions of CISC (Complex Instr. Set Computer) machines difficult to pipeline?

b) Why are RISC machines usually Load & Store machines (i.e., only Load and Store instructions access memory)?

Unconditional Branch/"jump": j someLabel

opcode	large offset from PC or absolute address

Arithmetic: add R1, R2, R3

opcode	dest reg	operand 1 reg	operand 2 reg	unused

Arithmetic with immediate: addi R1, R2, 8

opcode	operand 1 reg	operand 2 reg	immediate value

Conditional Branch: beq R1, R2, end_if

opcode	operand 1 reg	operand 2 reg	PC-relative offset to label

Load/Store: lw R1, 16(R2)

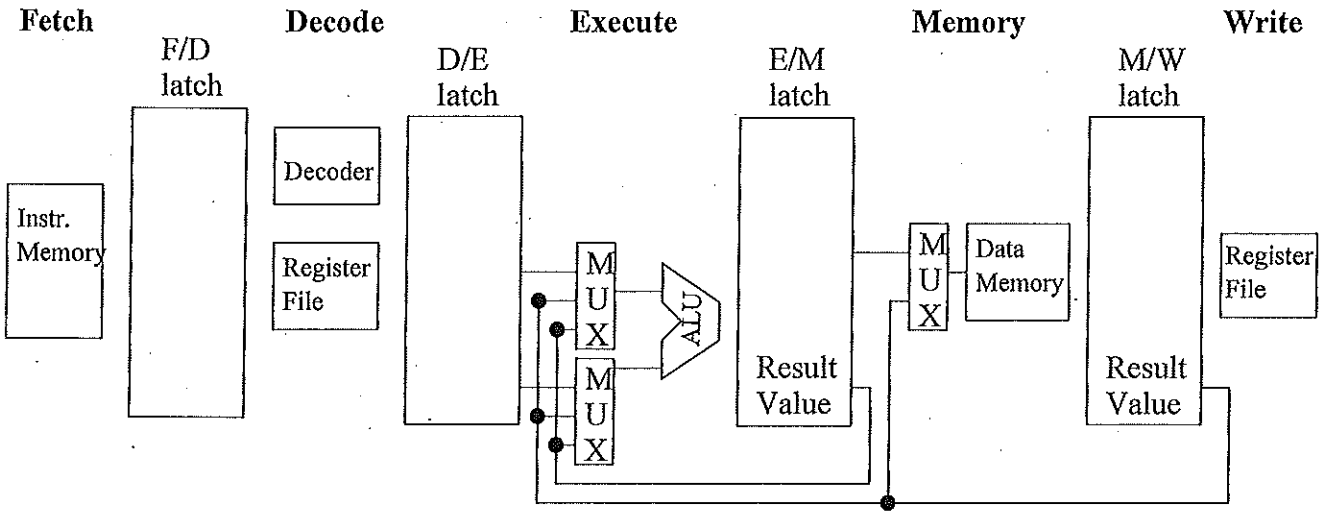
opcode	operand reg	base reg	offset from base reg

3. The whole question refers to a pipelined, RISC machine with five stages:

- F, fetch - fetch the instruction from memory
- D, decode - determine the type of instruction and read any necessary register values
- E, execute - perform ALU operation or memory address calculation for LOAD or STORE instructions
- M, memory - access memory on LOAD or STORE instruction
- W, write - write register values

a) Complete the following timing diagram assuming NO by-pass signal paths.

Without by-pass signal paths	**Time →**																			
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
ADD R1, R3, R4	F	D	E	M	W															
ADD R2, R4, R5																				
ADD R3, R2, R1																				
LOAD R2, 12(R3)																				
STORE R2, 16(R2)																				



b) Complete the following timing diagram assuming by-pass signal paths as shown above.

With by-pass signal paths	**Time →**																			
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
ADD R1, R3, R4	F	D	E	M	W															
ADD R2, R4, R5																				
ADD R3, R2, R1																				
LOAD R2, 12(R3)																				
STORE R2, 16(R2)																				

4. *Control Hazards* - branching causes problems since the pipeline can be filled with the wrong instructions.

```
IF:     BEQ  R3, R8, ELSE
        ADD  R4, R5, R6          /* ADD should not be executed if the branch is taken */
        SUB  R8, R5, R6
        .
        .
        B  END_IF
ELSE:   MUL  R3, R3, R2          /* MUL should not be executed if the previous B executes*/
        .
        .
END_IF:
```

a) During which stage is the target address (addr. of "ELSE" label) calculated for the BEQ instruction?

b) During which stage of BEQ instruction is the comparison between registers (R3 and R8) performed (i.e., when is the *outcome* (*taken* or *not taken*) of the branch known)?

If we always (statically) continue to fetch sequentially until the outcome of a conditional branch is known:
c) How many cycle branch penalty for a taken outcome?

d) How many cycle branch penalty for a not-taken outcome?

Branch Prediction - predict whether the branch will be taken and fetch accordingly

Static Techniques:
a) Predict never taken - continue to fetch sequentially. If the branch is not taken, then there is no wasted fetches.

b) Predict always taken - fetch from branch target as soon as possible
(From analyzing program behavior, > 50% of branches are taken.)

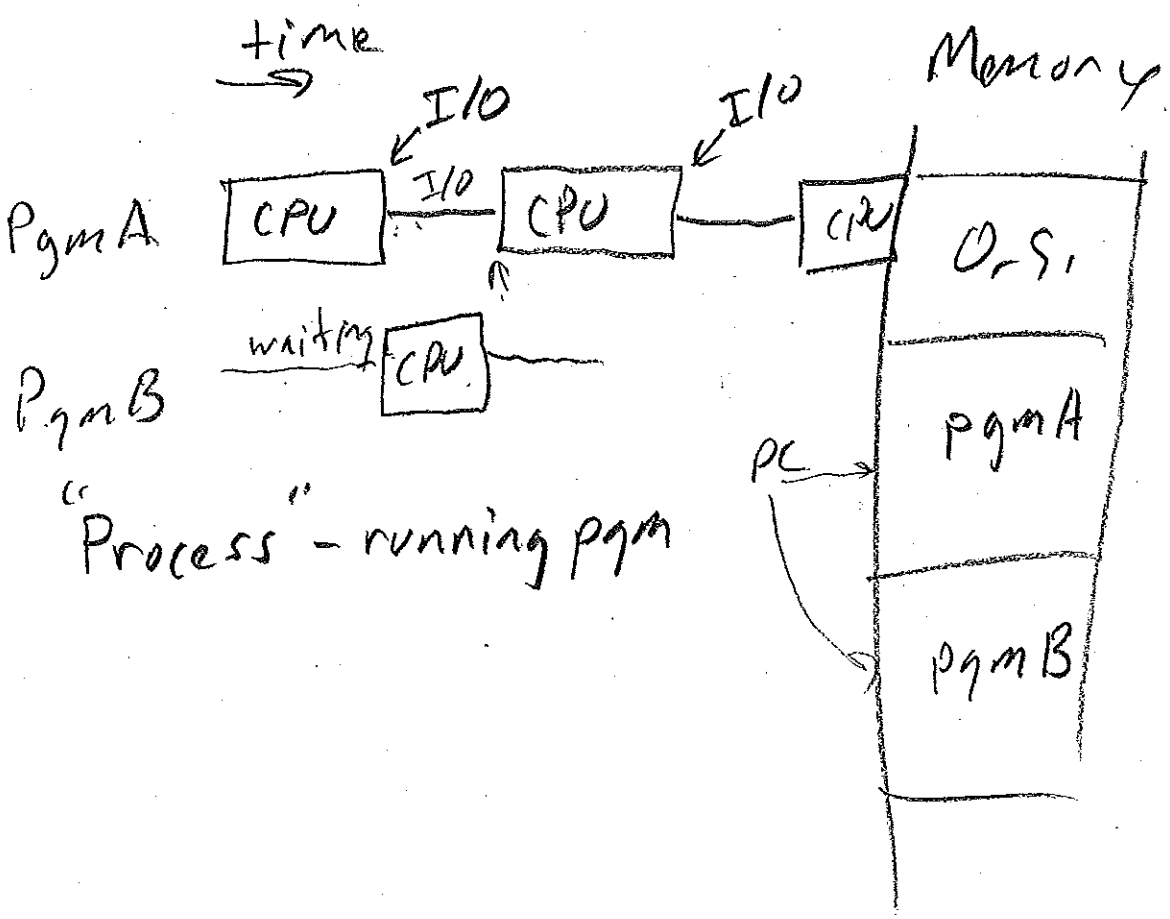
c) Predict by opcode - compiler helps by having different opcodes based on likely outcome of the branch
Consider the HLL constructs:

HLL	**AL**
While (x > 0) do	BR_LE_PREDICT_NOT_TAKEN R3, #0, END_WHILE
{loop body}	
end while	END_WHILE:

Studies have found about a 75% successful prediction rate using this technique.

5. Suppose that you are writing a compiler for a machine that has opcodes to statically predict whether or not branches will be taken (BEQ, BEQ_PREDICT_TAKEN, BEQ_PREDICT_NOT_TAKEN, etc.). For each of the following HLL statements, predict whether or not the compiler should predict taken or not. (Briefly justify your answer)

a)	b)	c)	d)
integer x	integer x	integer i	char ch
if (x > 0) then	if (x = 0) then	for i := 1 to 500 do	if (ch >= 'a' and ch <= 'z') then
end if	end if	end for	end if

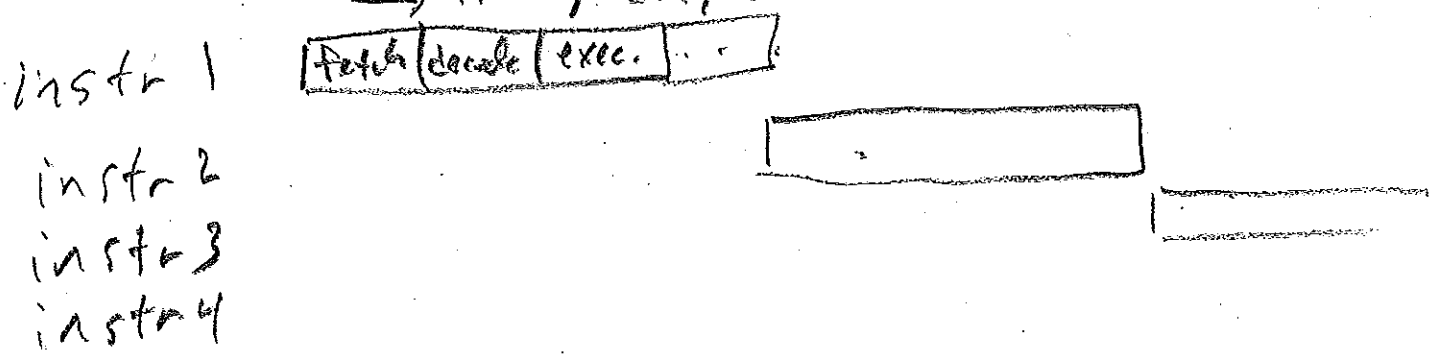


"Process" - running pgm

Pipelining - '80's

Pre-'80's serial execution of pgm - mental model

→ timing diagram



Assembly line: Auto. 4-hrs

Pre-assembly line

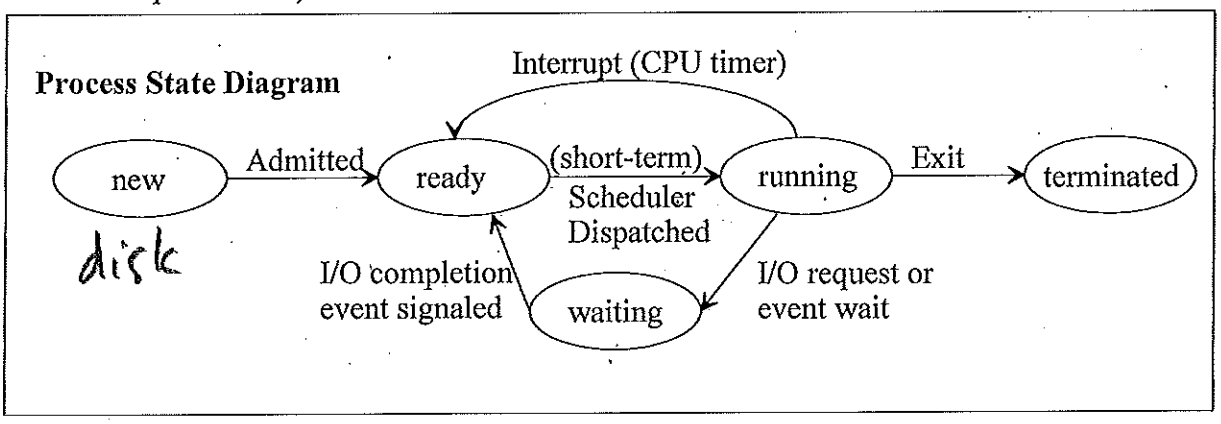
4 1-hr stages

4) Protection to restrict a process from access files of other programs varies depending on whether the computer is using memory-mapped I/O or instruction-based I/O. If memory-mapped I/O is being used, the memory address associated with the external device I/O registers are outside of the process accessable memory address space. Thus, our solution (3) above is enough to force a process to request I/O through operating system calls.

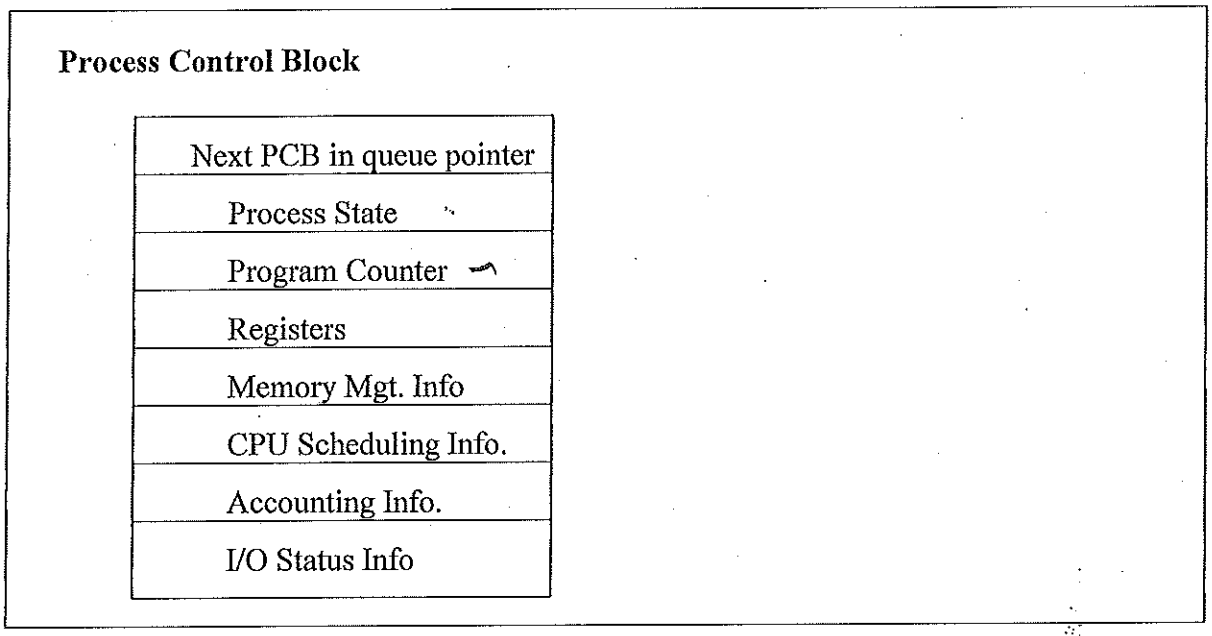
   If instruction-based I/O instructions array being used. I/O has a separate address space from memory, but we can make these I/O instructions privileged so they can only be executed in system mode. Thus, a process could not execute them directly.

OS manages processes (running programs):

A *process* is the term for a running program. A process's state consists of the CPU register values, its run-time stack in memory, and it's other memory content. Many processes may be executing concurrently, but only one can be executing on a CPU at a time. When the CPU switches to another process, a *context switch* occurs which involves saving the complete state of the previously executing process before loading the state of the next process to execute on the CPU. Depending on the hardware, this can take up to 100 microseconds (i.e., very slow in computer terms).



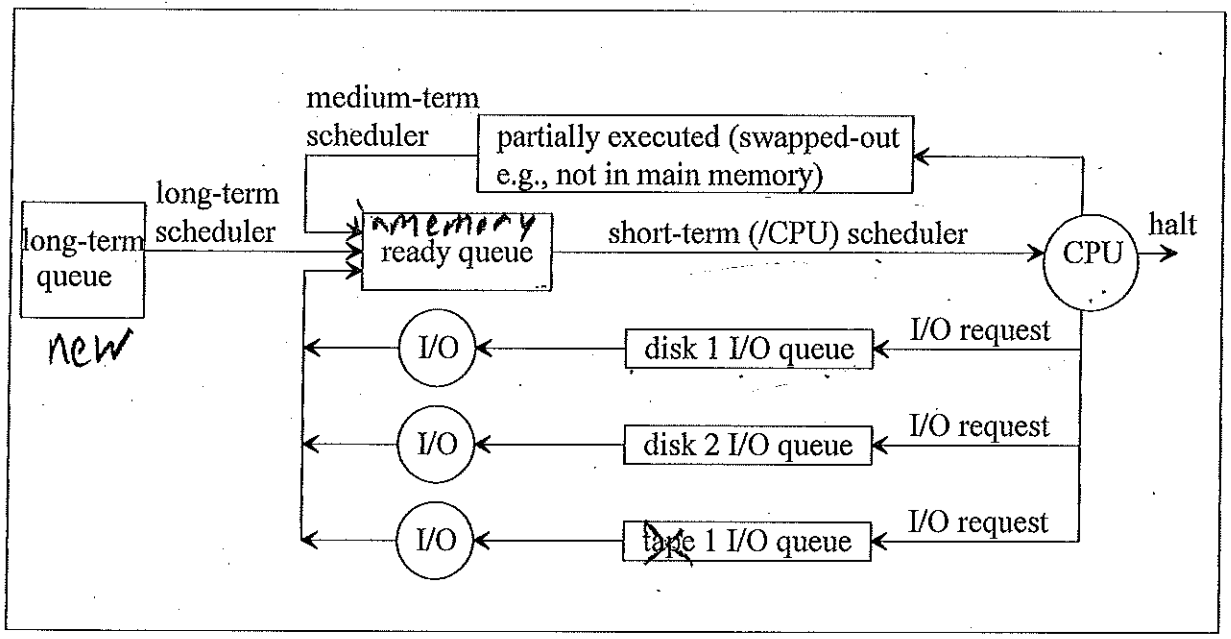
Queues are used to hold *process control blocks (PCB)* that represent processes internally to the OS.



1.+25-(2)

OS maintains queues and does scheduling:

The PCB for a process moves around from queue to queue depending on its state.



I/O queues - since I/O is so slow, several programs might have outstanding requests to use an I/O device so a queue for each I/O device is necessary.

Ready (Short-term) queue - programs that are in memory and ready to execute. All they need is the CPU to run.

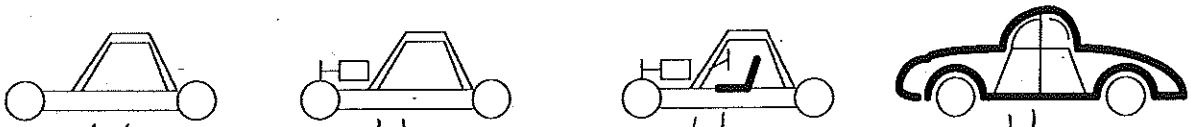
Medium-term queue - programs that are partially executed, but have been swapped out of memory to disk

Long-term queue - user has requested the a program be executed, but it has not yet been loaded into memory

Computer Org. (CS 1410) Lecture 25 Name: _____

1. Assume that an automobile assembly process takes 4 hours.

Chassis Motor Interior Exterior



1 hr 1 hr 1 hr 1 hr

1 car/hr completion rate

a) If the stages take the following amounts of time, then what is the time between completions of automobiles?
Chassis 1 hour
Motor 1 hour
Interior 1 hour
Exterior 1 hour

b) If the stages take the following amounts of time, then what is the time between completions of automobiles?
Chassis 45 minutes
Motor 1 hour
Interior 1 hour and 15 minutes
Exterior 1 hour

2. We could follow the instruction/machine cycle into stages for instruction pipelined.

- **Fetch Instruction - read instruction pointed at by the program counter (PC) from memory into Instr. Reg. (IR)**
- **Decode Instruction - figure out what kind of instruction was read**
- **Fetch Operands - get operand values from the memory or registers**
- **Execute Instruction - do some operation with the operands to get some result**
- **Write Result - put the result into a register or in a memory location**

Two approaches for designing a computer is CISC (Complex Instr. Set Computer - pre-1980) and RISC (Reduced Instruction Set Computer post 1985, MIPS was one of the first commerical RISC processor). A CISC philosophy was to make assembly language (AL) as much like a high-level language (HLL) as possible to reduce the "*semantic gap*" between AL and HLL. The rational for CISC at the time was to:

- **reduce compiler complexity and aid assembly language programming. Compilers were not too good during the 50's to 70's, (e.g., they made poor use of general purpose registers so code was inefficient) so some programs were written in assembly language.**
- **reduce the program size. More powerful/complex instructions reduced the number of instructions necessary in a program. Memory during the 50's to 70's was limited and expensive.**
- **improve code efficiency by allowing complex sequence of instructions to be implemented in microcode. For example, the Digital Equipment Corporation (DEC) VAX computer had an assembly-language instruction "MATCHC *substrLength, substr, strLength, str*" that looks for a substring within a string.**

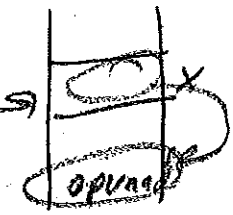
The architectural characteristics of CISC machines include:

- **complex, high-level like AL instructions**
- **variable format machine-language instructions that execute using a variable number of clock cycles**
- **many addressing modes (e.g., the DEC VAX had 22 addressing modes)**

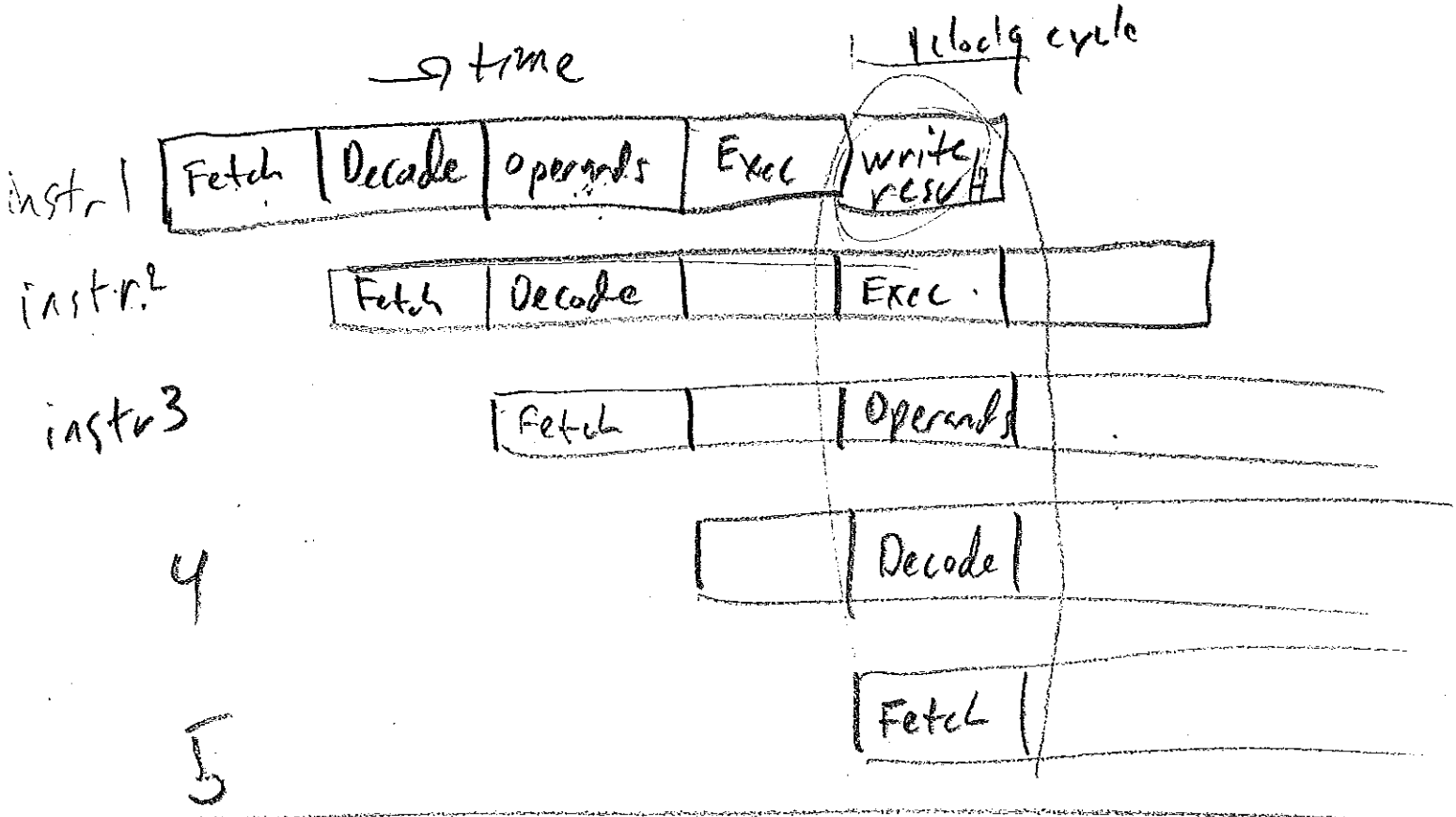
a) Why are complex instructions of CISC (Complex Instr. Set Computer) machines difficult to pipeline?

MARIE
add,
addi — indirect

addix

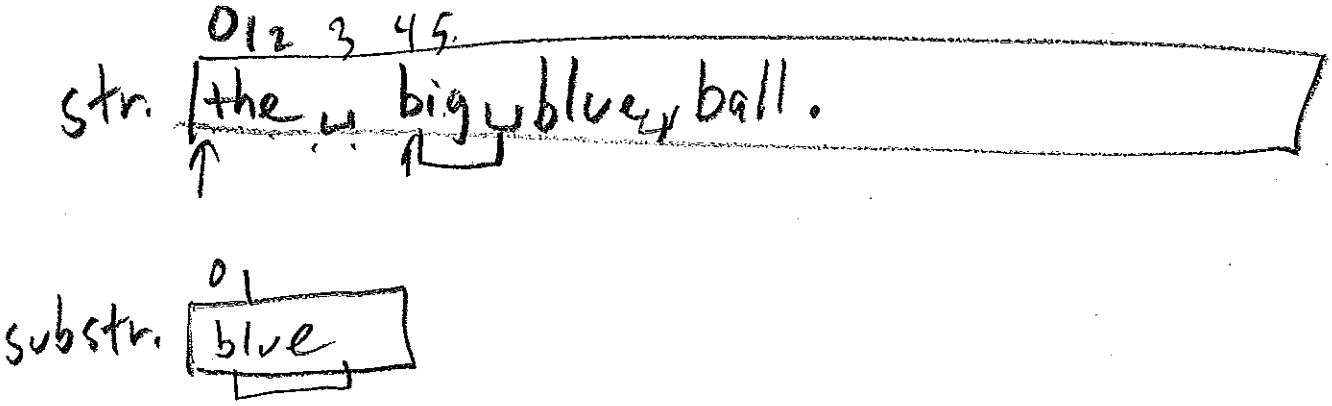


Lect 25-(4)



Complex instr. –

Simple instr. (MIPS) add, sub, load, store, . . .



HLL pgm:

```
for startIndex = 0 to (length-1) do
    if str[startIndex] == substr[0] then
        found = True
        for substrIndex = 1 to (substrLength-1) do
            if substr[substrIndex] ≠
                str[startIndex + substrIndex]
                break out of loop
                found = False
            end if
        end for
        if found then
            return startIndex
        end if
end for
return -1
```

b) Why are RISC machines usually Load & Store machines (i.e., only Load and Store instructions access memory)?

- simple instr. - exec. short always
- fixed length instr. format MIPS/RISC ML all 32-bits

Unconditional Branch/"jump": j someLabel

opcode	large offset from PC or absolute address

Arithmetic: add R1, R2, R3

opcode 8	dest reg 5	operand 1 reg 5	operand 2 reg 5	unused

Arithmetic with immediate: addi R1, R2, 8

opcode 8	operand 1 reg 5	operand 2 reg 5	immediate value 8 24

Conditional Branch: beq R1, R2, end_if

opcode	operand 1 reg	operand 2 reg	PC-relative offset to label

Load/Store: lw R1, 16(R2)

opcode	operand reg	base reg 2	offset from base reg

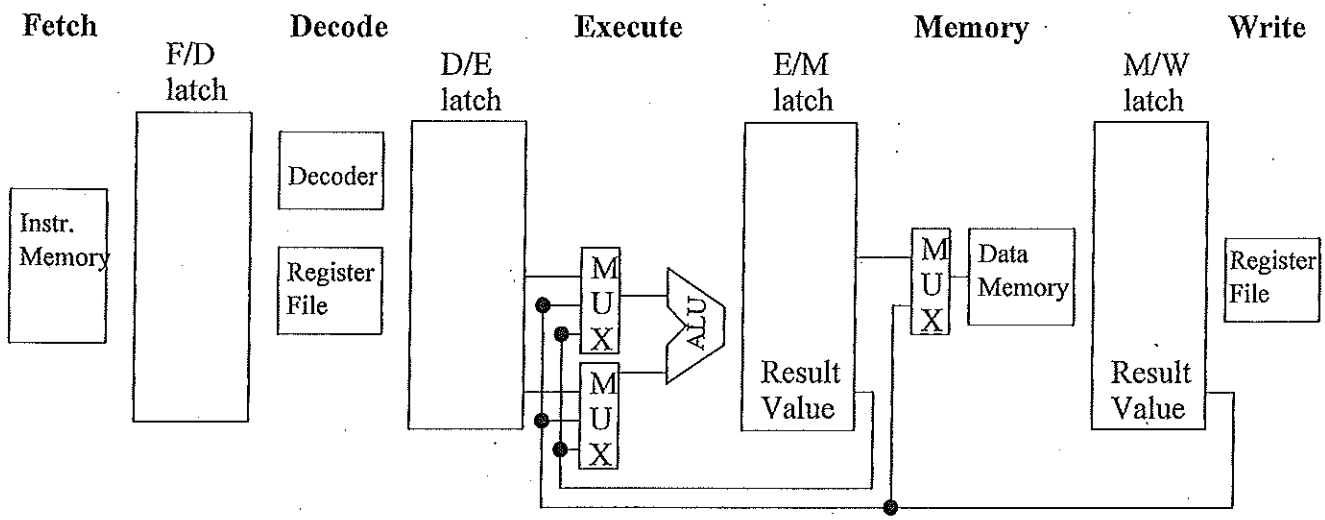
8-bits

3. The whole question refers to a pipelined, RISC machine with five stages:

- F, fetch - fetch the instruction from memory
- D, decode - determine the type of instruction and read any necessary register values
- E, execute - perform ALU operation or memory address calculation for LOAD or STORE instructions
- M, memory - access memory on LOAD or STORE instruction
- W, write - write register values

a) Complete the following timing diagram assuming NO by-pass signal paths.

Without by-pass signal paths	**Time →**																			
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
ADD R1, R3, R4	F	D	E	M	W															
ADD R2, R4, R5																				
ADD R3, R2, R1																				
LOAD R2, 12(R3)																				
STORE R2, 16(R2)																				



b) Complete the following timing diagram assuming by-pass signal paths as shown above.

With by-pass signal paths	**Time →**																			
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
ADD R1, R3, R4	F	D	E	M	W															
ADD R2, R4, R5																				
ADD R3, R2, R1																				
LOAD R2, 12(R3)																				
STORE R2, 16(R2)																				

Computer Org. (CS 1410) Lecture 26 Name: _____

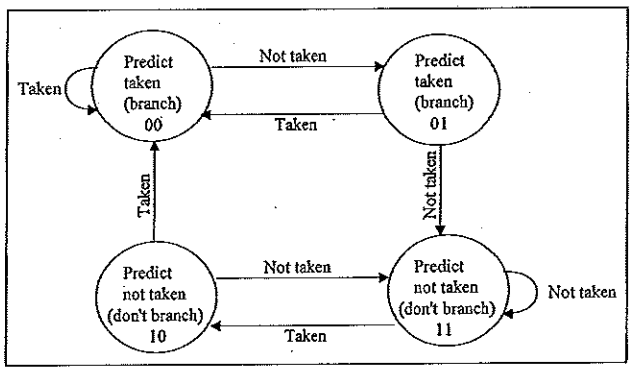
Dynamic Techniques: try to improve prediction by recording program's history of conditional branch
Problem: How do we avoid always fetching the instruction after the branch?

```
Instr. Addr.                                          time ---->
 400    WHILE:    BEQ  R3, R8, END_WHILE      F     D     E
 404              ADD  R4, R5, R6                   F
                  :
                  :
 436              J  WHILE
 440    END_WHILE:
```

Need target of branch, but its not calculated yet! Plus, how do we know that we have just fetched a branch since it has not been decoded yet?

Solution: Branch-prediction buffer (BPB)/Branch-History Table (BHT)- small, fully-associative cache to store information about most recently executed branch instructions. In a BPB, the Branch instruction address acts as the tag since that's what you know about an instruction at stage F. During the F stage, the Branch-prediction buffer is checked to see if the instruction being fetched is a branch (e.g., if the PC matches an address in the BPB) instruction.

Valid Bit	Branch Instruction Address (tag field)	Target Address of Branch	Prediction Bits	



If the instruction is a branch instruction and it is in the Branch-prediction buffer, then the target address and prediction can be supplied by the BPB **by the end of F for the branch instruction.**

1. If the branch instruction is in the Branch-prediction buffer, will the target address supplied correspond to the correct instruction to be execute next?

2. What if the instruction is a branch instruction and it is not in the Branch-prediction buffer?

3. Should the Branch-prediction buffer contain entries for unconditional J as well as conditional branch instructions?

The table below shows the advantage of using a Branch-prediction buffer to improve accuracy of the branch prediction. It shows the impact of past ***n*** branches on prediction accuracy. Typically, two prediction bits are use so that two wrong predictions in a row are need to change the prediction -- see above state diagram.

		Type of mix	
n	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

Notice:

- the big jump in using the knowledge of just 1 past branch to predict the branch
- notice the big jump in going from using 1 to 2 past branches to predict the branch for scientific applications.

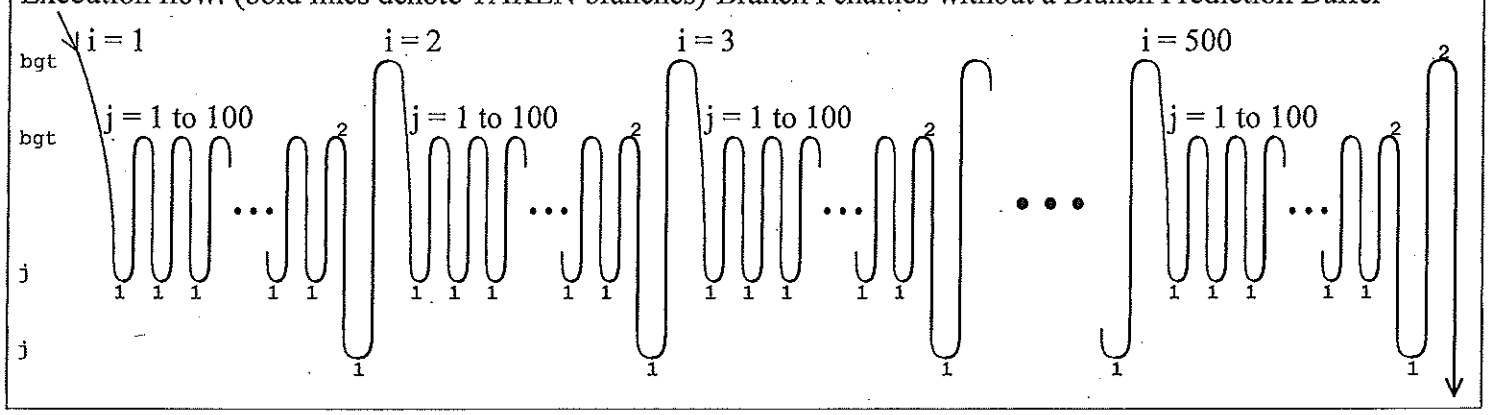
4. **What types of data do scientific applications spend most of their time processing?**
5. **What would be true about the code for processing this type of data?**

Consider the nested loops:

```
for (i = 1; i <= 500; i++) {

  for (j = 1; j <= 100; j++) {

      <do something>

  } // end for j

} // end for i
```

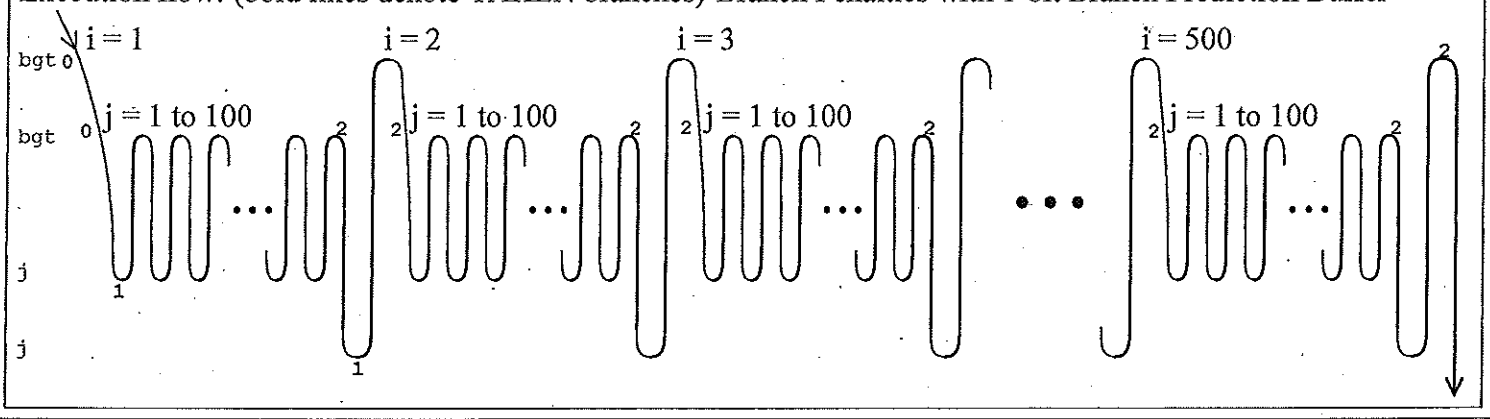
```
for_init_1:        li r3, 1
for_compare_1:     bgt r3, 500, end_for_1

for_init_2:        li r4, 1
for_compare_2:     bgt r4, 100, end_for_2

                   addi r4, r4, 1
                   j for_compare_2
end_for_2:
                   addi r3, r3, 1
                   j for_compare_1
end_for_1:
```

Execution flow: (bold lines denote TAKEN branches) Branch Penalties without a Branch Prediction Buffer



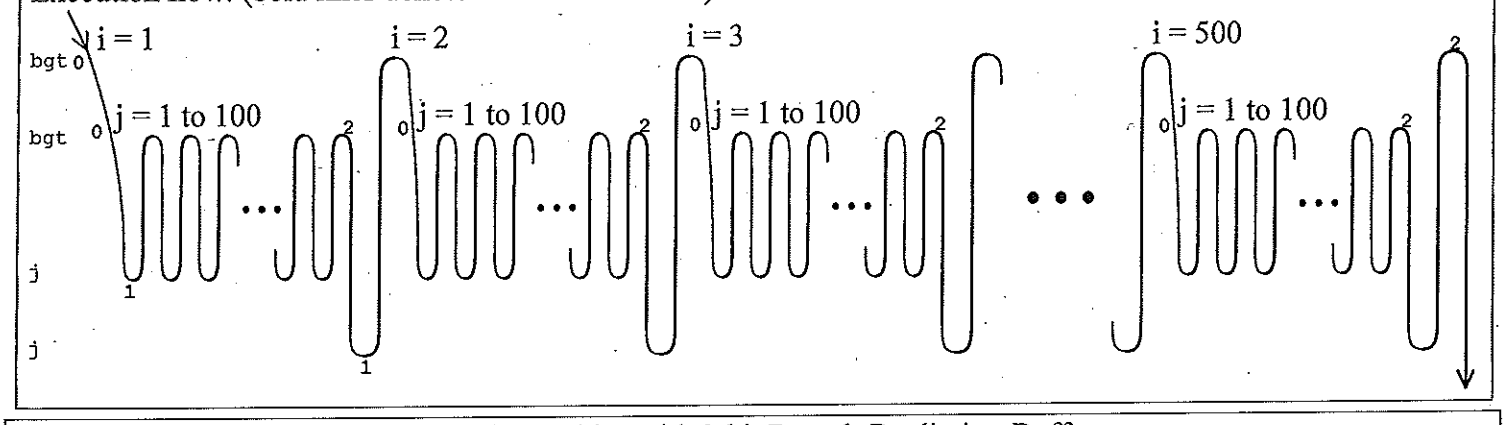
Branch Penalties without a Branch-Prediction-Buffer					
Branch Instruction	for 1 conditional (bgt r3, 500, ...)	for 2 conditional (bgt r4, 100, ...)	end for 2 uncond. (j for_compare_2)	end for 1 uncond. (j for_compare_1)	Total
Penalties	2 x 1 = 2	2 x 500 = 1000	1 x 100 x 500 = 50000	1 x 500 = 500	51,502

Execution flow: (bold lines denote TAKEN branches) Branch Penalties with 1-bit Branch Prediction Buffer



Branch Penalties with 1-bit Branch-Prediction-Buffer					
Branch Instruction	for 1 conditional (bgt r3, 500, ...)	for 2 conditional (bgt r4, 100, ...)	end for 2 uncond. (j for_compare_2)	end for 1 uncond. (j for_compare_1)	Total
Penalties	2 x 1 = 2	2 + (2+2) x 499 = 1998	1 x 1 = 1	1 x 1 = 1	2,002

Execution flow: (bold lines denote TAKEN branches) Branch Penalties with 2-bit Branch Prediction Buffer



Branch Penalties with 2-bit Branch-Prediction-Buffer					
Branch Instruction	for 1 conditional (`bgt r3, 500, ...`)	for 2 conditional (`bgt r4, 100, ...`)	end for 2 uncond. (`j for_compare_2`)	end for 1 uncond. (`j for_compare_1`)	Total
Penalties	2 x 1 = 2	2 x 500 = 1000	1 x 1 = 1	1 x 1 = 1	1,004

6. Consider the following bubble sort algorithm that sorts an array numbers[1..n]:

```
BubbleSort (int n, int numbers[])
    int bottom, test, temp;
    boolean exchanged = true;
    bottom = n - 2;
    while (exchanged) do
        exchanged = false;
        for test = 0 to bottom do
            if number[test] > number[test + 1] then
                temp = number[test];
                number[test] = number[test + 1];
                number[test + 1] = temp;
                exchanged = true;
            end if
        end for
        bottom = bottom - 1;
    end while
end BubbleSort
```

Part (a) answer	Part (b) answer

a) Where in the code would unconditional branches be used and where would conditional branches be used?

b) If the compiler could predict by opcode for the conditional branches (i.e., select whether to use machine language statements like: "BRANCH_LE_PREDICT_NOT_TAKEN" or "BRANCH_LE_PREDICT_TAKEN"), then which conditional branches would be "PREDICT_NOT_TAKEN" and which would be "PREDICT_TAKEN"?

c) Assumptions:

- n = 100 and the numbers are initially in descending order before the bubble sort algorithm is called
- the five-stage RISC pipeline
- target addresses of all branches is known at the end of the D stage (so uncond. branch penalty of 1)
- the outcome of conditional branches is known at the end of the E stage (so cond. branch penalty of 2)
- ignore any data hazards

Under the above assumptions, answer the following questions:

i) If fixed predict-never-taken is used by the hardware, then what will be the total branch penalty (# cycles wasted) for the algorithm? (Here assume NO branch-prediction-buffer)

ii) If a branch-prediction-buffer with one history bit per entry is used, then what will be the total branch penalty (# cycles wasted) for the algorithm? (Assume predict-not taken is used if there is no match in the branch-prediction-buffer) Explain your answer.

iii) If a branch-prediction-buffer with two history bit per entry is used, then what will be the total branch penalty (# cycles wasted) for the algorithm? (BPB - wrong twice before prediction changed) Explain your answer.

Computer Org. (CS 1410) Lecture 26 Name: _____

Dynamic Techniques: try to improve prediction by recording program's history of conditional branch

Problem: How do we avoid always fetching the instruction after the branch?

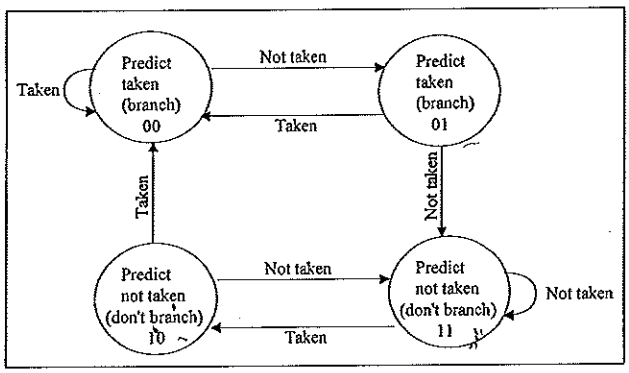
while R3 ≠ R8 do

Instr. Addr.			time →		
400	WHILE:	BEQ R3, R8, END_WHILE	F	D	E
404		ADD R4, R5, R6		F	D
		⋮			F
436		J WHILE	F	D	E
440	END_WHILE:				

Need target of branch, but its not calculated yet! Plus, how do we know that we have just fetched a branch since it has not been decoded yet?

Solution: Branch-prediction buffer (BPB)/Branch History Table (BHT)- small, fully-associative cache to store information about most recently executed branch instructions. In a BPB, the Branch instruction address acts as the tag since that's what you know about an instruction at stage F. During the F stage, the Branch-prediction buffer is checked to see if the instruction being fetched is a branch (e.g., if the PC matches an address in the BPB) instruction.

Valid Bit	Branch Instruction Address (tag field)	Target Address of Branch	Prediction Bits	
1	400	440	0	0
1	436	400	0	0



If the instruction is a branch instruction and it is in the Branch-prediction buffer, then the target address and prediction can be supplied by the BPB **by the end of F for the branch instruction.**

1. If the branch instruction is in the Branch-prediction buffer, will the target address supplied correspond to the correct instruction to be execute next?

2. What if the instruction is a branch instruction and it is not in the Branch-prediction buffer?

3. Should the Branch-prediction buffer contain entries for unconditional J as well as conditional branch instructions?

The table below shows the advantage of using a Branch-prediction buffer to improve accuracy of the branch prediction. It shows the impact of past *n* branches on prediction accuracy. Typically, two prediction bits are use so that two wrong predictions in a row are need to change the prediction -- see above state diagram.

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

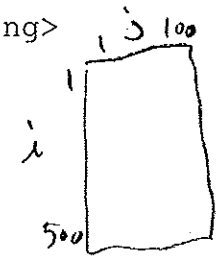
Notice:

- the big jump in using the knowledge of just 1 past branch to predict the branch
- notice the big jump in going from using 1 to 2 past branches to predict the branch for scientific applications.

4. **What types of data do scientific applications spend most of their time processing?**
5. **What would be true about the code for processing this type of data?**

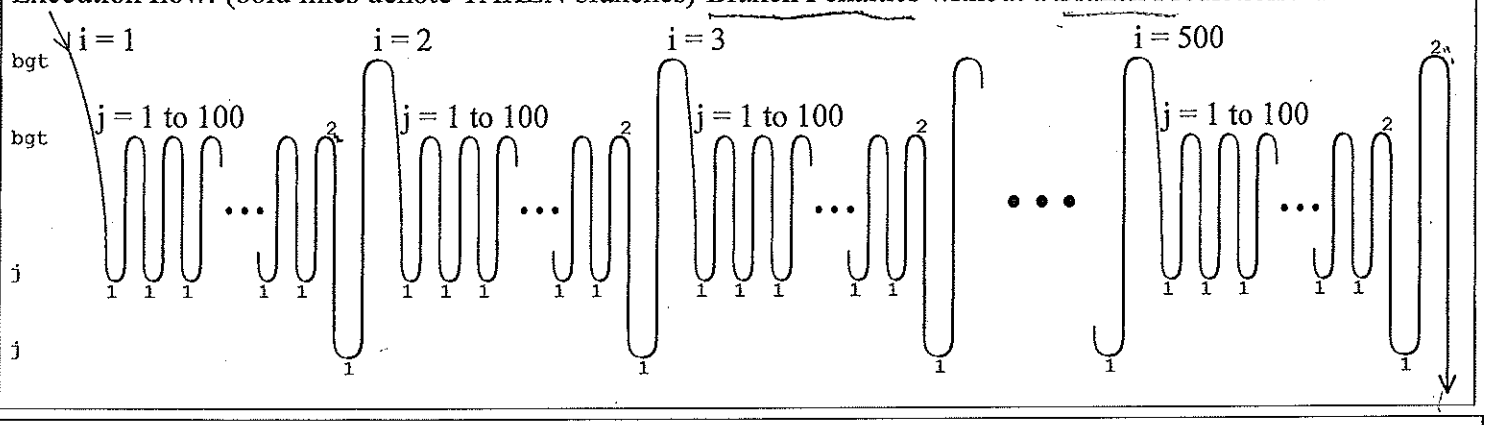
Consider the nested loops:

```
for (i = 1; i <= 500; i++) {

  for (j = 1; j <= 100; j++) {

      <do something>

  } // end for j

} // end for i
```



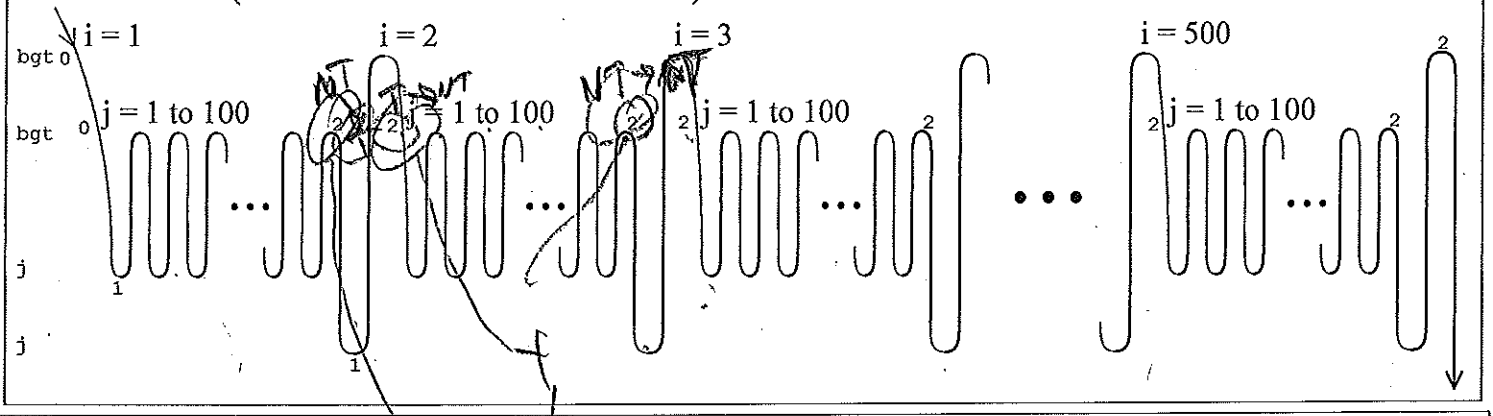

```
for_init_1:        li r3, 1
for_compare_1:     bgt r3, 500, end_for_1

for_init_2:        li r4, 1
for_compare_2:     bgt r4, 100, end_for_2

                   addi r4, r4, 1
                   j for_compare_2

end_for_2:

                   addi r3, r3, 1
                   j for_compare_1

end_for_1:
```

Execution flow: (bold lines denote TAKEN branches) Branch Penalties without a Branch Prediction Buffer



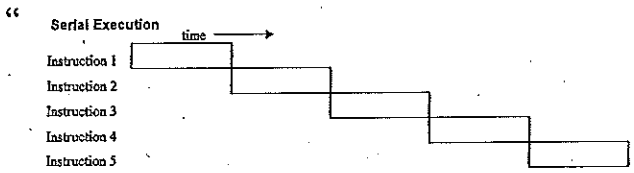
Branch Penalties without a Branch-Prediction-Buffer					
Branch Instruction	for 1 conditional (`bgt r3, 500, ...`)	for 2 conditional (`bgt r4, 100, ...`)	end for 2 uncond. (`j for_compare_2`)	end for 1 uncond. (`j for_compare_1`)	Total
Penalties	2 x 1 = 2	2 x 500 = 1000	1 x 100 x 500 = 50000	1 x 500 = 500	51,502

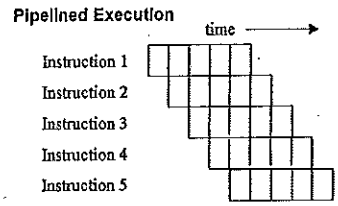
Execution flow: (bold lines denote TAKEN branches) Branch Penalties with 1-bit Branch Prediction Buffer



Branch Penalties with 1-bit Branch-Prediction-Buffer					
Branch Instruction	for 1 conditional (`bgt r3, 500, ...`)	for 2 conditional (`bgt r4, 100, ...`)	end for 2 uncond. (`j for_compare_2`)	end for 1 uncond. (`j for_compare_1`)	Total
Penalties	2 x 1 = 2	2 + (2+2) x 499 = 1998	1 x 1 = 1	1 x 1 = 1	2,002

Comp. Org. Lecture 27 Name:_____

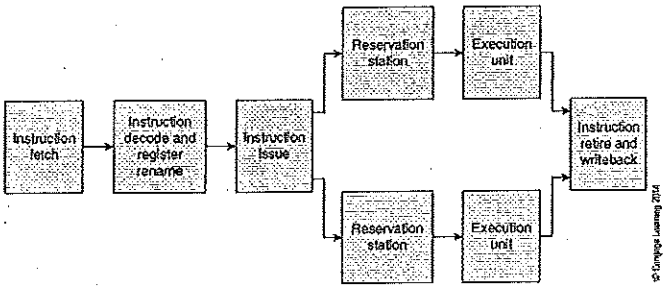


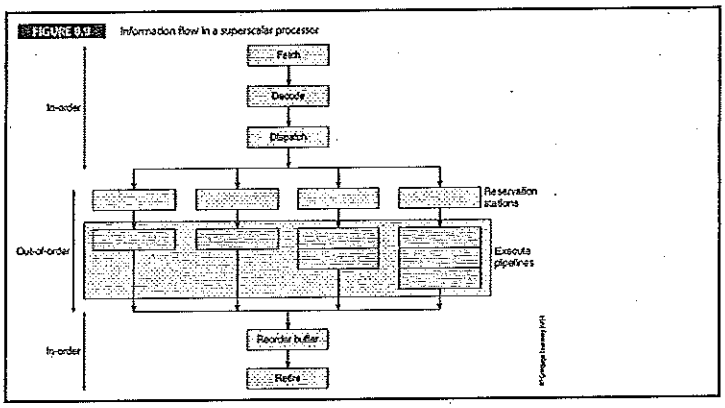


Beyond RISC" - goal of multiple instructions completed per clock cycle

superscalar (e.g., modern Intel x86, AMD processors) - multiple instructions in the same stage of execution in duplicate pipeline hardware

FIGURE 8.5 The generic superscalar processor





- Instruction Fetch - obtain "next" instruction(s) from memory (I cache)
- Instruction Decode - decode instr(s) and rename user-visible registers to avoid data hazards (WAW: write-
- after-write & WAR: write-after-read) introduced by out-of-order execution. Consider instruction sequence of:

  Instruction 1: MUL R3, R3, R5
  Instruction 2: ADDI R4, R3, 1
  Instruction 3: ADDI R3, R5, 8
  Instruction 4: SUB R7, R3. R4

1. If these instructions were issued (selected to be executed) out-of-order and completed out-of-order, then:
a) why would writing R3 in instruction 3 before reading R3's value in instruction 2 cause a problem? (WAR)

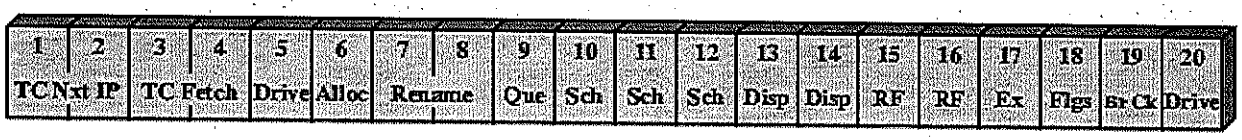
b) why would writing R3 in instruction 3 before writing R3 in instruction 1 cause a problem? (WAW)

c) If we had more registers (say R33 - R64) and utilized them dynamically as the program executes (called "register renaming"), which registers could we rename to eliminate the WAR and WAW dependencies?

- Instruction *issue* - sent instruction to reservations unit associated with an appropriate execution unit (integer ALU, fl. pt. ALU, LOAD/STORE memory unit, etc.) to await execution
- Reservation station - *dispatch* instruction to execution unit when unit becomes free and all of the instruction's operand values are known, i.e., all RAW data dependences have cleared
- Instruction retire - writes results of potentially out-of-order instructions back to registers to ensure correct in-order completion. Also, communicates with the reservation stages when instruction completion frees resources (e.g., "virtual" registers used in register renaming) .

Intel x86 Processor (e.g., Pentium 4) Operation:

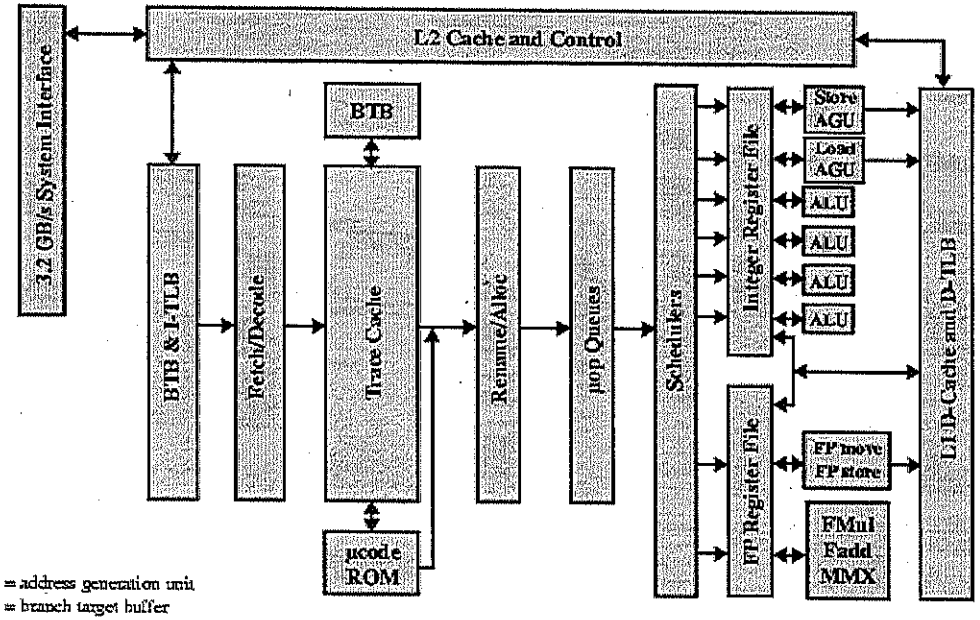
- Fetch x86 (CISC) instructions from memory in order of static program
- Translate each x86 instruction into one or more fixed length RISC instructions (micro-operations)
- Execute micro-ops on superscalar pipeline
  - micro-ops may be executed out of order
  - up to 4 micro-ops dispatched per clock cycle
- Commit results of micro-ops to register set in original x86 program flow order
- Outer CISC shell with inner RISC core
- Inner RISC core pipeline at least 20 stages (Some micro-ops require multiple execution stages)



TC Next IP = trace cache next instruction pointer
TC Fetch = trace cache fetch
Alloc = allocate

Rename = register renaming
Que = micro-op queuing
Sch = micro-op scheduling
Disp = Dispatch

RF = register file
Ex = execute
Flgs = flags
Br Ck = branch check



AGU = address generation unit
BTB = branch target buffer
D-TLB = data translation lookaside buffer
I-TLB = instruction translation lookaside buffer

2. Suppose we have a 16MB (2^{24} bytes) memory that is byte addressable, and a 128KB (2^{17} bytes) cache with 64 (2^6) bytes per block.

a) How many total lines are in the cache?

b) If the cache is direct-mapped, how many cache lines could a specific memory block be mapped to?

c) If the cache is direct-mapped, what would be the format (tag bits, cache line bits, block offset bits) of the address? (Clearly indicate the number of bits in each)

d) If the cache is fully-associative, how many cache lines could a specific memory block be mapped to?

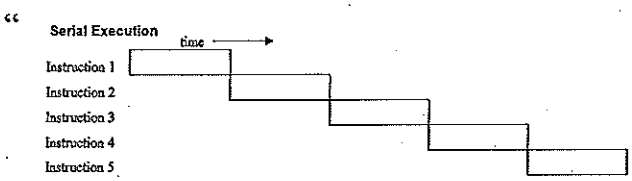
e) If the cache is fully-associative, what would be the format of the address?

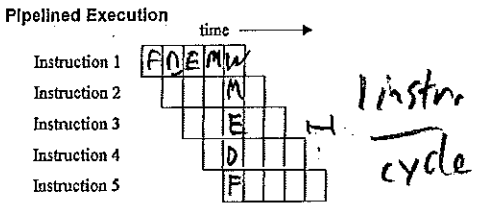
f) If the cache is 4-way set associative, how many cache lines could a specific memory block be mapped to?

g) If the cache is 4-way set associative, how many sets would there be?

h) If the cache is 4-way set associative, what would be the format of the address?

Comp. Org. Lecture 27 Name:_____





Beyond RISC" - goal of multiple instructions completed per clock cycle

superscalar (e.g., modern Intel x86, AMD processors) - multiple instructions in the same stage of execution in duplicate pipeline hardware

FIGURE 8.5 The generic superscalar processor

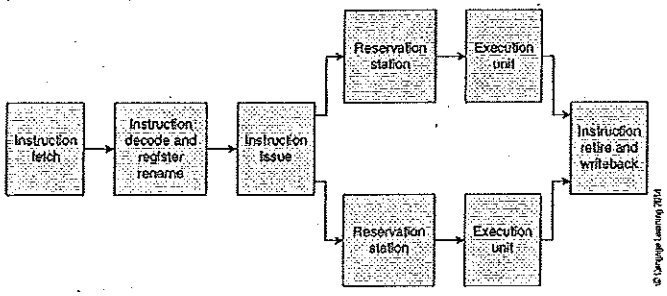
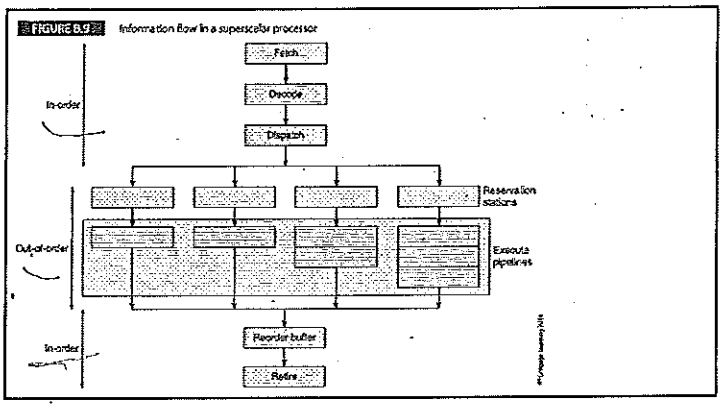


FIGURE 8.9 Information flow in a superscalar processor



- Instruction Fetch - obtain "next" instruction(s) from memory (I cache)
- Instruction Decode - decode instr(s) and rename user-visible registers to avoid data hazards (WAW: write-
- after-write & WAR: write-after-read) introduced by out-of-order execution. Consider instruction sequence of:

Instruction 1: MUL R3, R3, R5 RAW
Instruction 2: ADDI R4, R3, 1 WAR
Instruction 3: ADDI R3, R5, 8 WAW
Instruction 4: SUB R7, R3, R4 (R32-63)

1. If these instructions were issued (selected to be executed) out-of-order and completed out-of-order, then:

a) why would writing R3 in instruction 3 before reading R3's value in instruction 2 cause a problem? (WAR)

b) why would writing R3 in instruction 3 before writing R3 in instruction 1 cause a problem? (WAW)

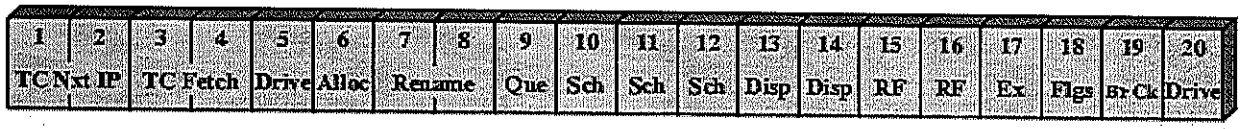
c) If we had more registers (say R33 - R64) and utilized them dynamically as the program executes (called "register renaming"), which registers could we rename to eliminate the WAR and WAW dependencies?

- Instruction *issue* - sent instruction to reservations unit associated with an appropriate execution unit (integer ALU, fl. pt. ALU, LOAD/STORE memory unit, etc.) to await execution
- Reservation station - *dispatch* instruction to execution unit when unit becomes free and all of the instruction's operand values are known, i.e., all RAW data dependences have cleared
- Instruction retire - writes results of potentially out-of-order instructions back to registers to ensure correct in-order completion. Also, communicates with the reservation stages when instruction completion frees resources (e.g., "virtual" registers used in register renaming) .

Lect 27 - 1

Intel x86 Processor (e.g., Pentium 4) Operation:

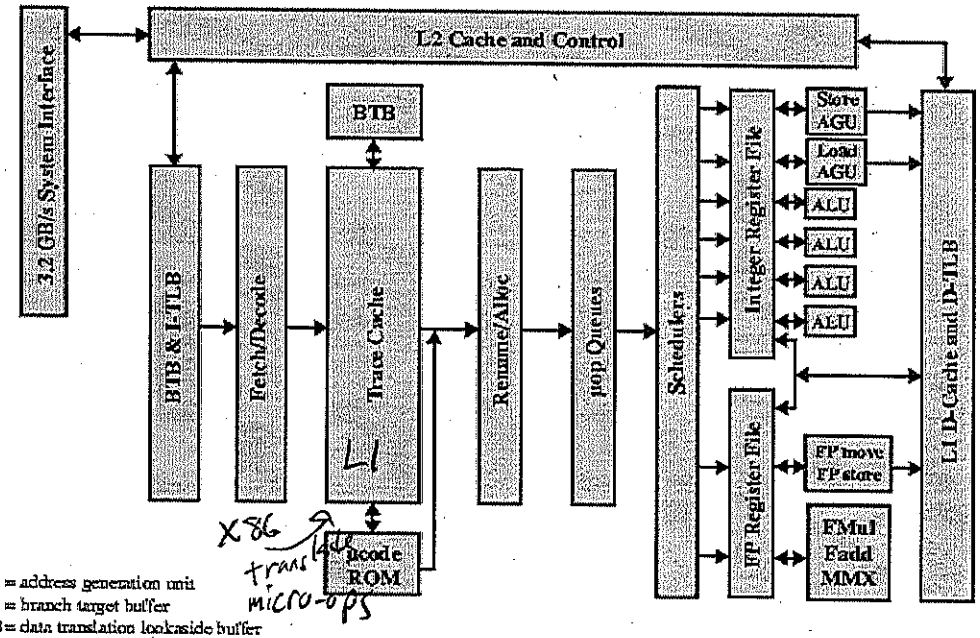
- Fetch x86 (CISC) instructions from memory in order of static program
- Translate each x86 instruction into one or more fixed length RISC instructions (micro-operations)
- Execute micro-ops on superscalar pipeline
  - micro-ops may be executed out of order
  - up to 4 micro-ops dispatched per clock cycle
- Commit results of micro-ops to register set in original x86 program flow order
- Outer CISC shell with inner RISC core
- Inner RISC core pipeline at least 20 stages (Some micro-ops require multiple execution stages)



TC Next IP = trace cache next instruction pointer
TC Fetch = trace cache fetch
Alloc = allocate

Rename = register renaming
Que = micro-op queuing
Sch = micro-op scheduling
Disp = Dispatch

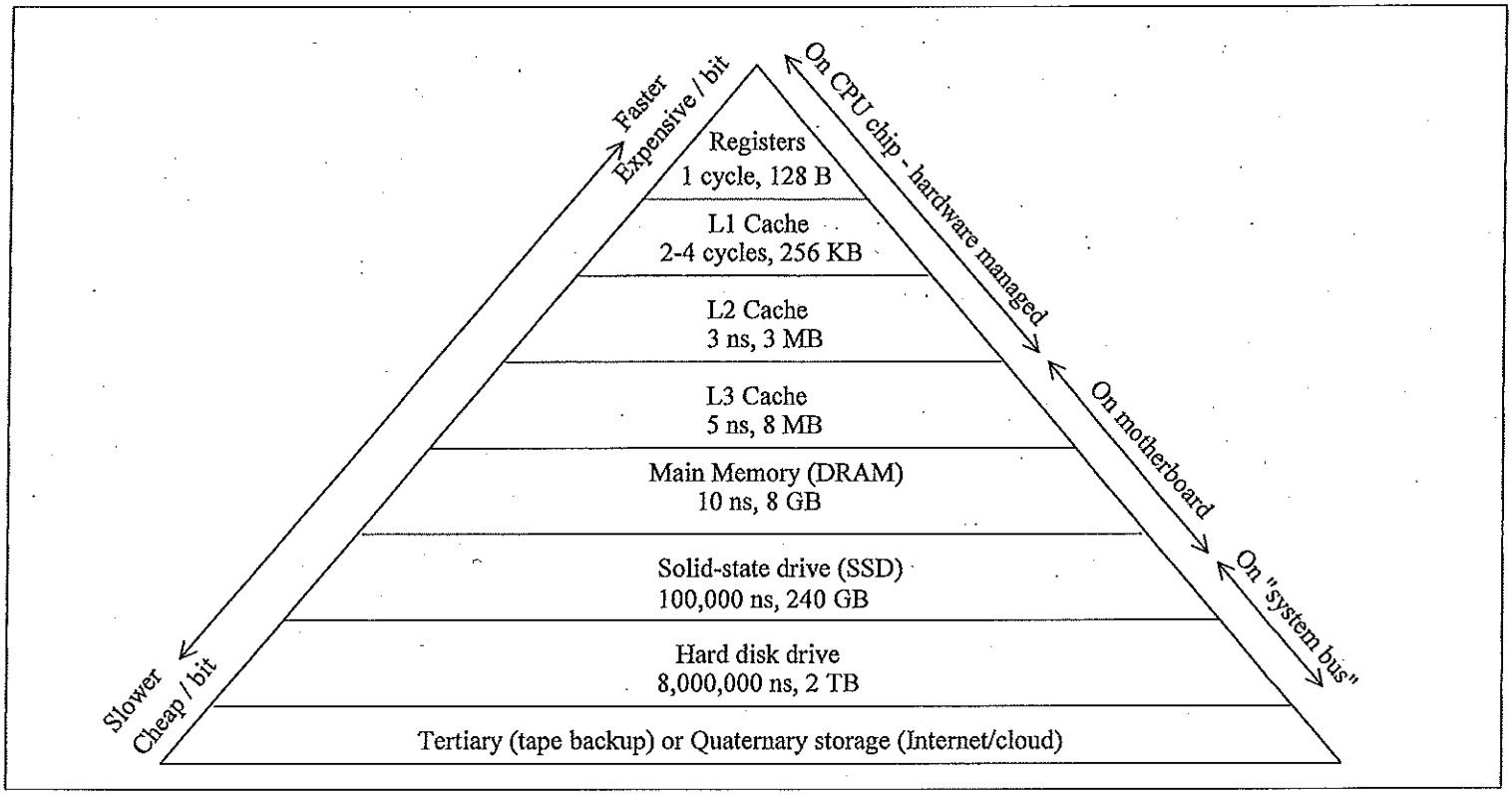
RF = register file
Ex = execute
Flgs = flags
Br Ck = branch check



AGU = address generation unit
BTB = branch target buffer
D-TLB = data translation lookaside buffer
I-TLB = instruction translation lookaside buffer

Memory Hierarchy

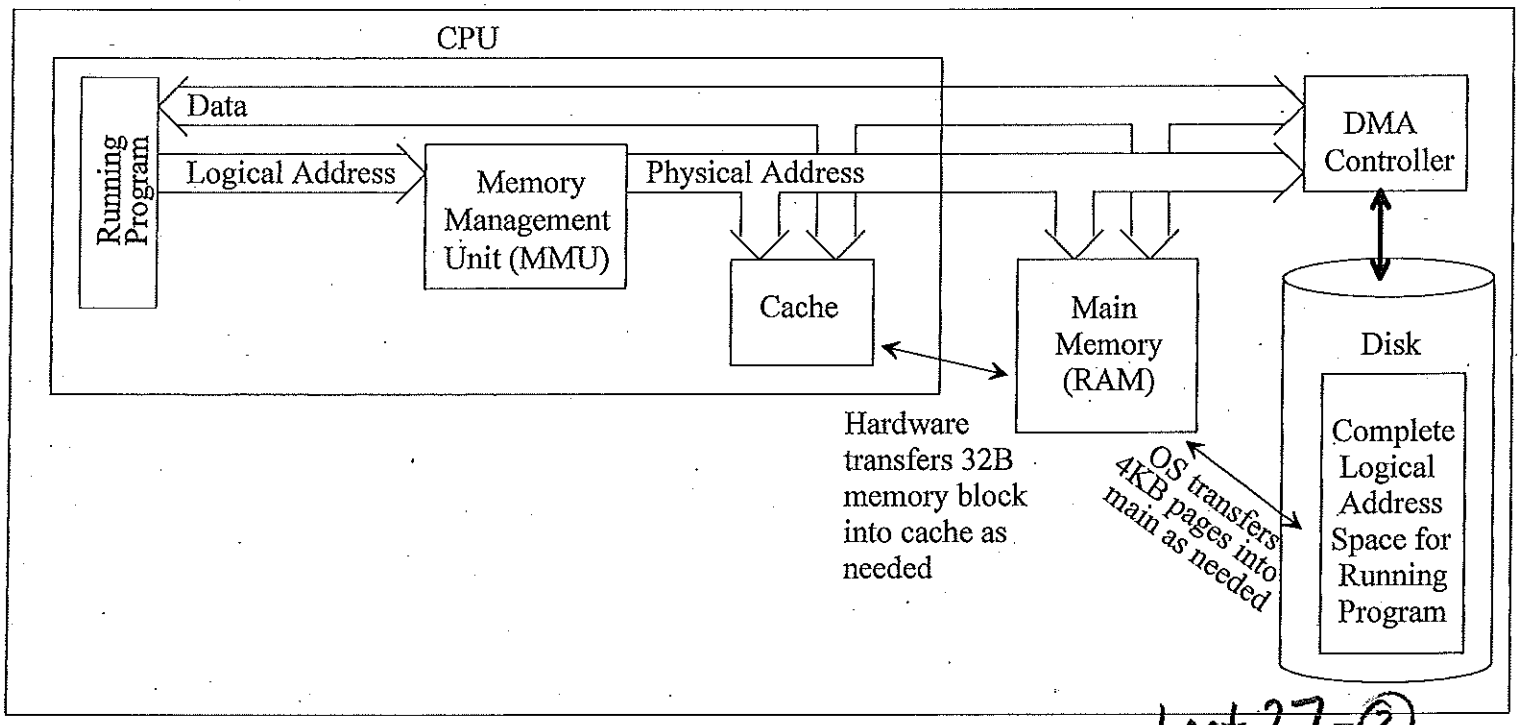
Goal: "Fast", "unlimited" storage at a reasonable cost per bit.



Recall the von Neumann bottleneck - single, relatively slow path between the CPU and main memory.

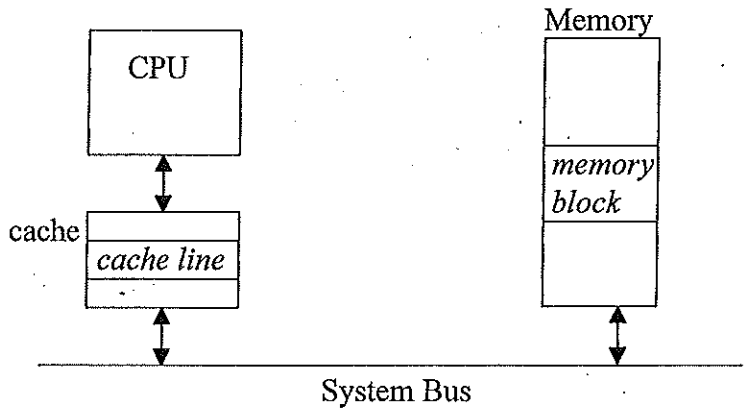
Fast: When you need something from "memory" check "faster" cache(s) first for a copy

"Unlimited" storage: *Virtual memory* - executing program's *logical address* space (ML pgm, run-time stack, heap, global memory) completing out on disk with main memory (DRAM) acting like "cache" for hard disk.



Lect 27-3

Main Idea of a Cache - keep a copy of frequently used information as "close" (w.r.t access time) to the processor as possible.



Steps when the CPU generates a memory request:
1) check the (faster) cache first
2) If the addressed memory value is in the cache (called a ***hit***), then no need to access memory
3) If the addressed memory value is NOT in the cache (called a ***miss***), then transfer the block of memory containing the reference to cache. (The CPU is *stalled* and idle while this occurs)
4) The cache supplies the memory value from the cache.

Effective (Average) Memory Access Time
Suppose that the hit time (i.e., access time of cache, t_c) is 2 ns, the cache miss penalty (i.e., load cache line from memory might involve multiple reads) is 150 ns, and the hit ratio is 99% (so miss ratio is 1%).

Effective Access Time ≈ (hit time) + (miss penalty * miss ratio)

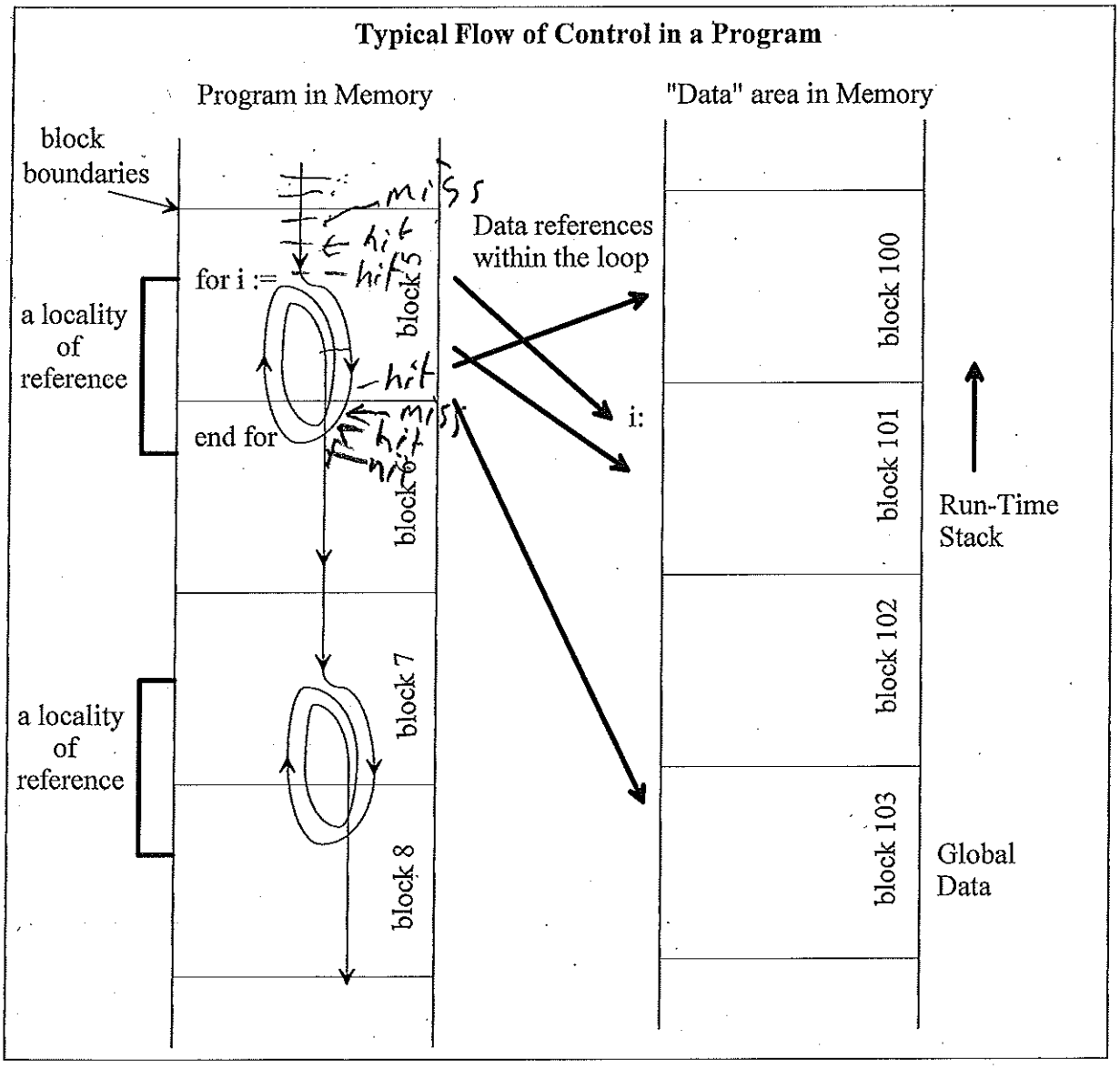
Effective Access Time = 2 + 150 * (1 - 0.99) = 2 + 1.5 = 3.5 ns

(One way to reduce the miss penalty is to not have the cache wait for the whole block to be read from memory before supplying the accessed memory word.)

Fortunately, programs exhibit ***locality of reference*** that helps achieve high hit-ratios:

1) ***spatial locality*** - if a (logical) memory address is referenced, nearby memory addresses will tend to be referenced soon.

2) ***temporal locality*** - if a memory address is referenced, it will tend to be referenced again soon.



Lect 27-5

Three Types of Cache

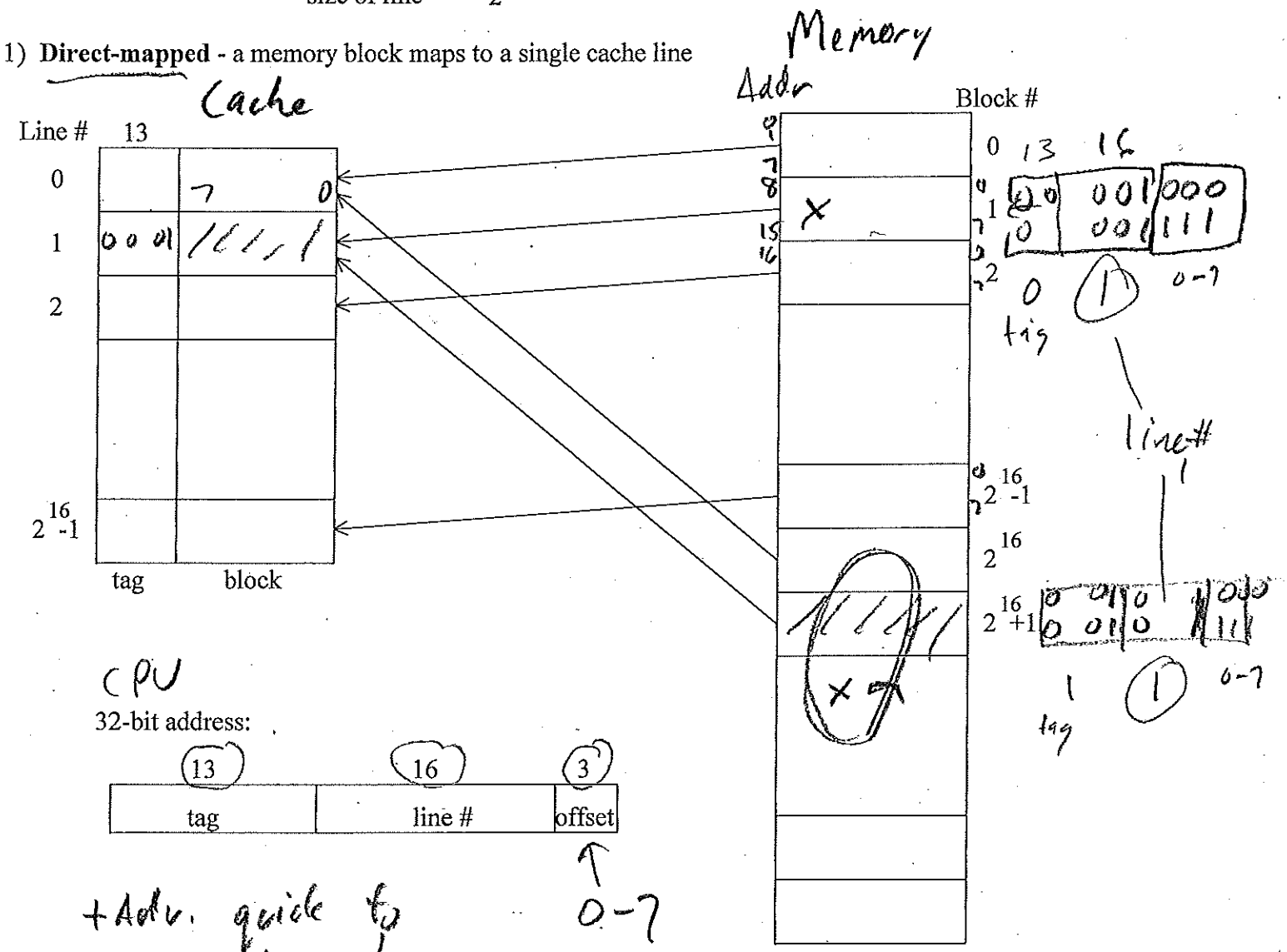
Cache - Small fast memory between the CPU and RAM/Main memory.

Example:

- 32-bit address
- 512 KB (2^{19}) cache
- 8 byte per block/line
- byte-addressable memory

$$\text{Number of Cache Line} = \frac{\text{size of cache}}{\text{size of line}} = \frac{2^{19}}{2^3} = 2^{16}$$

1) **Direct-mapped** - a memory block maps to a single cache line



CPU

32-bit address:

13	16	3
tag	line #	offset

0-7

+Adv. quick to search cache

- cache conflict

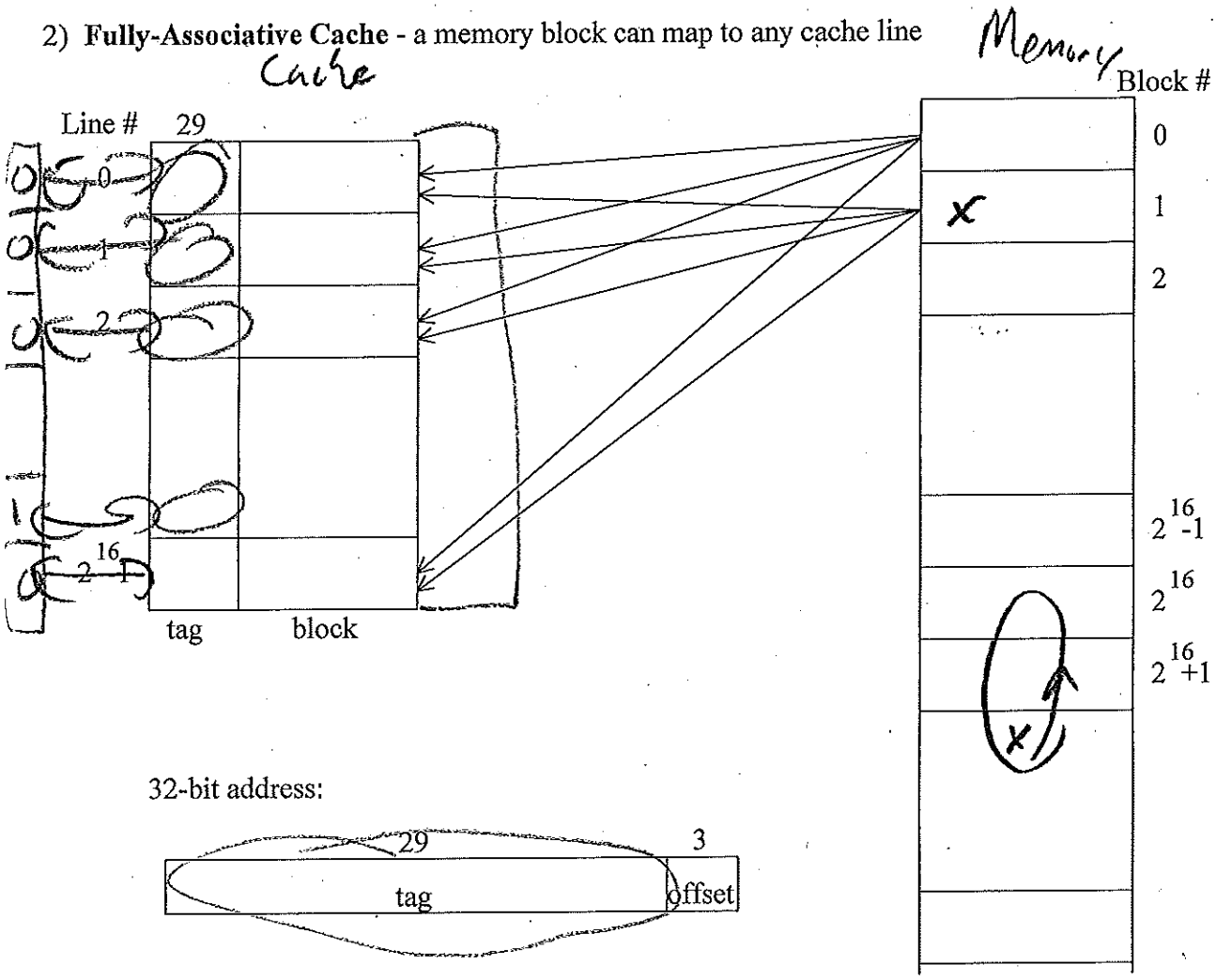
Lect 27-6

Same Cache Example:

- 32-bit address, byte-addressable memory
- 512 KB (2^{19})
- 8 byte per block/line

$$\text{Number of Cache Line} = \frac{\text{size of cache}}{\text{size of line}} = \frac{2^{19}}{2^3} = 2^{16}$$

2) **Fully-Associative Cache** - a memory block can map to any cache line



32-bit address:

Advantage: Flexibility on what's in the cache

Disadvantage: Complex circuit to compare all tags of the cache with the tag in the target address

Therefore, they are expensive and slower so use only for small caches (say 8-64 lines) BPB

Replacement algorithms - on a miss of a full cache, we must select a block in the cache to replace

- LRU - replace the cache block that has not been used for the longest time (need additional bits)
- Random - select a block randomly (only slightly worse that LRU)
- FIFO - select the block that has been in the cache for the longest time (slightly worse that LRU)

Lect 27 - 7

Same Cache Example:

- 32-bit address, byte-addressable memory
- 512 KB (2^{19})
- 8 byte per block/line

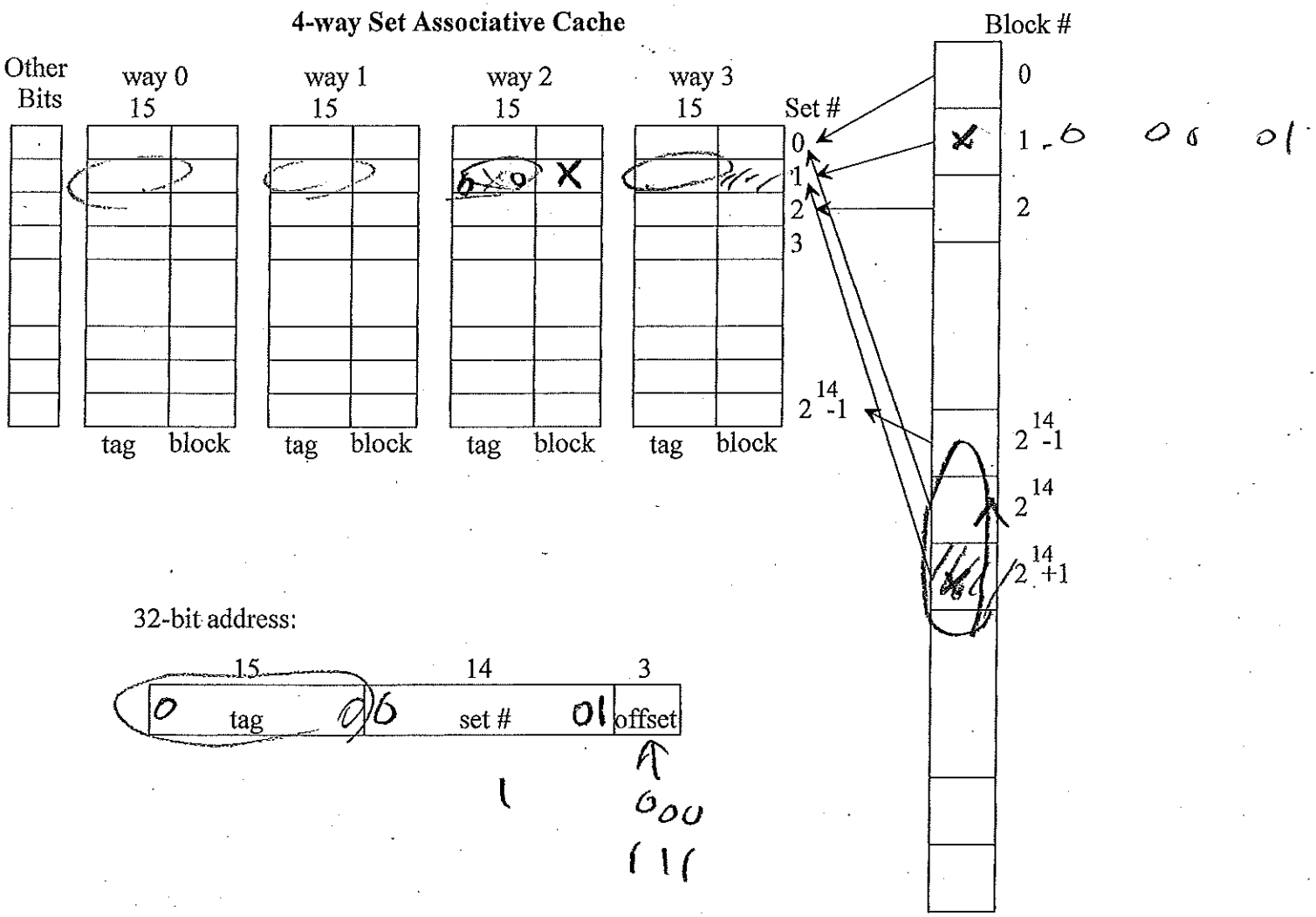
$$\text{Number of Cache Line} = \frac{\text{size of cache}}{\text{size of line}} = \frac{2^{19}}{2^3} = 2^{16}$$

3) **Set-Associative Cache** - a memory block can map to a small (2, 4, or 8) set of cache lines
Common Possibilities:

- 2-way set associative - each memory block can map to either of two lines in the cache
- 4-way set associative - each memory block can map to either of four lines in the cache

$$\text{Number of Sets} = \frac{\text{number of cache lines}}{\text{size of each set}} = \frac{2^{16}}{4} = \frac{2^{16}}{2^2} = 2^{14}$$

4-way Set Associative Cache

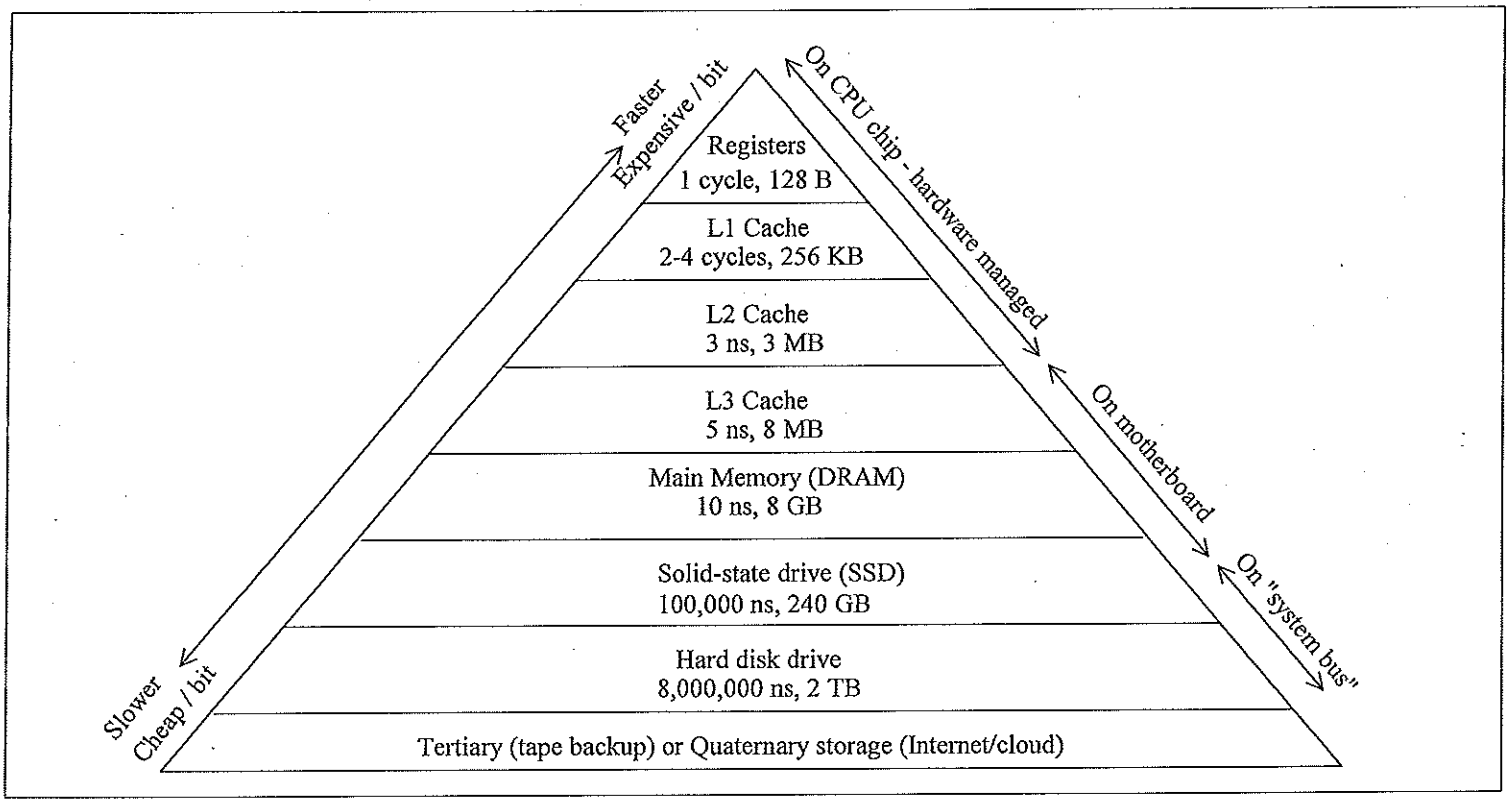


32-bit address:

Lect 27 - 8

Memory Hierarchy

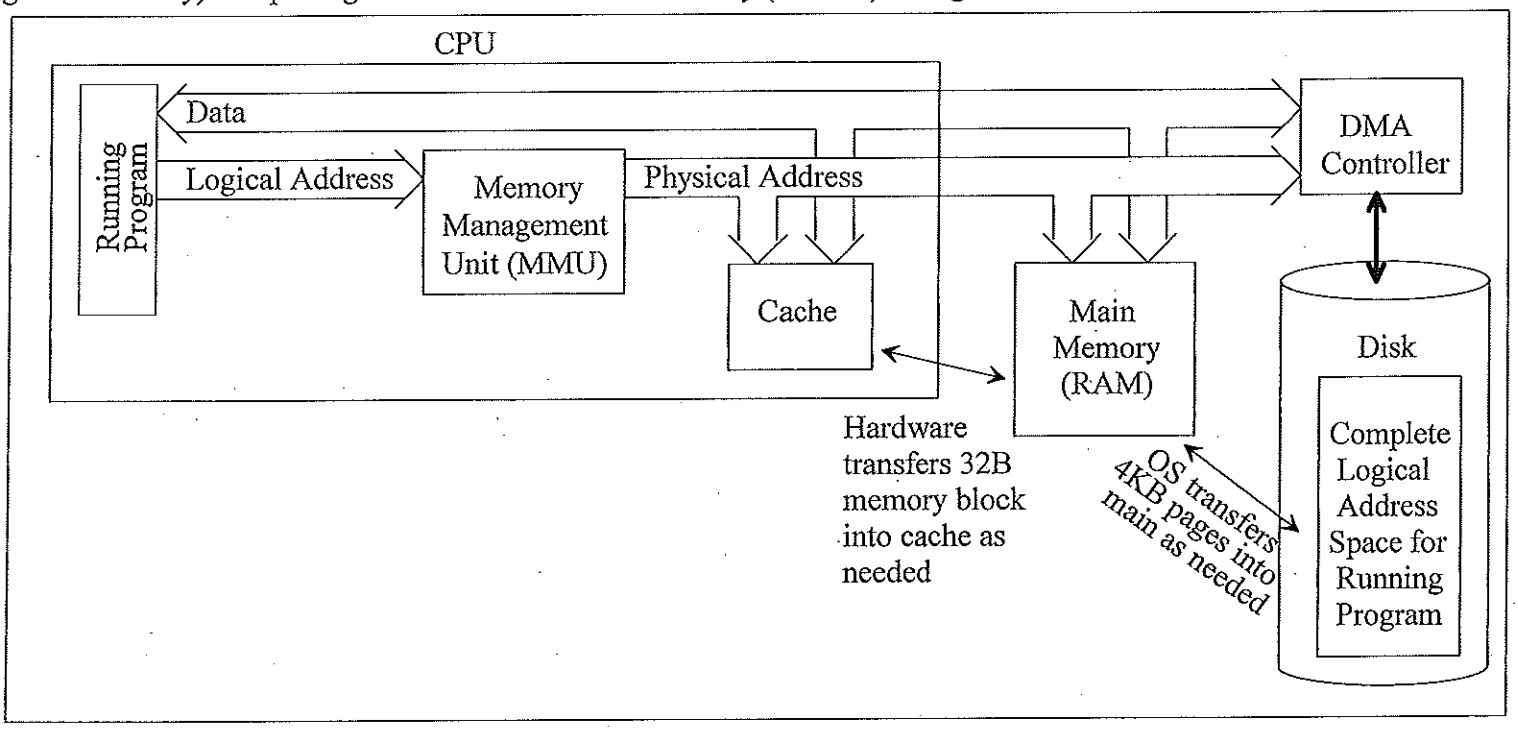
Goal: "Fast", "unlimited" storage at a reasonable cost per bit.



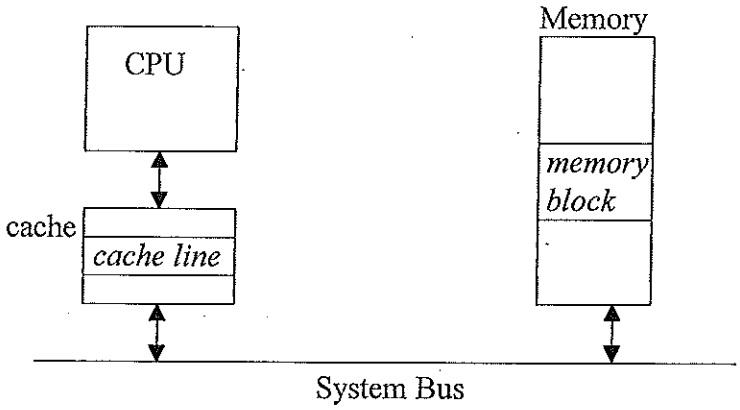
Recall the von Neumann bottleneck - single, relatively slow path between the CPU and main memory.

Fast: When you need something from "memory" check "faster" cache(s) first for a copy

"Unlimited" storage: *Virtual memory* - executing program's *logical address* space (ML pgm, run-time stack, heap, global memory) completing out on disk with main memory (DRAM) acting like "cache" for hard disk.



Main Idea of a Cache - keep a copy of frequently used information as "close" (w.r.t access time) to the processor as possible.



Steps when the CPU generates a memory request:
1) check the (faster) cache first
2) If the addressed memory value is in the cache (called a ***hit***), then no need to access memory
3) If the addressed memory value is NOT in the cache (called a ***miss***), then transfer the block of memory containing the reference to cache. (The CPU is *stalled* and idle while this occurs)
4) The cache supplies the memory value from the cache.

Effective (Average) Memory Access Time
Suppose that the hit time (i.e., access time of cache, t_c) is 2 ns, the cache miss penalty (i.e., load cache line from memory might involve multiple reads) is 150 ns, and the hit ratio is 99% (so miss ratio is 1%).

Effective Access Time $\approx$ (hit time) + (miss penalty * miss ratio)

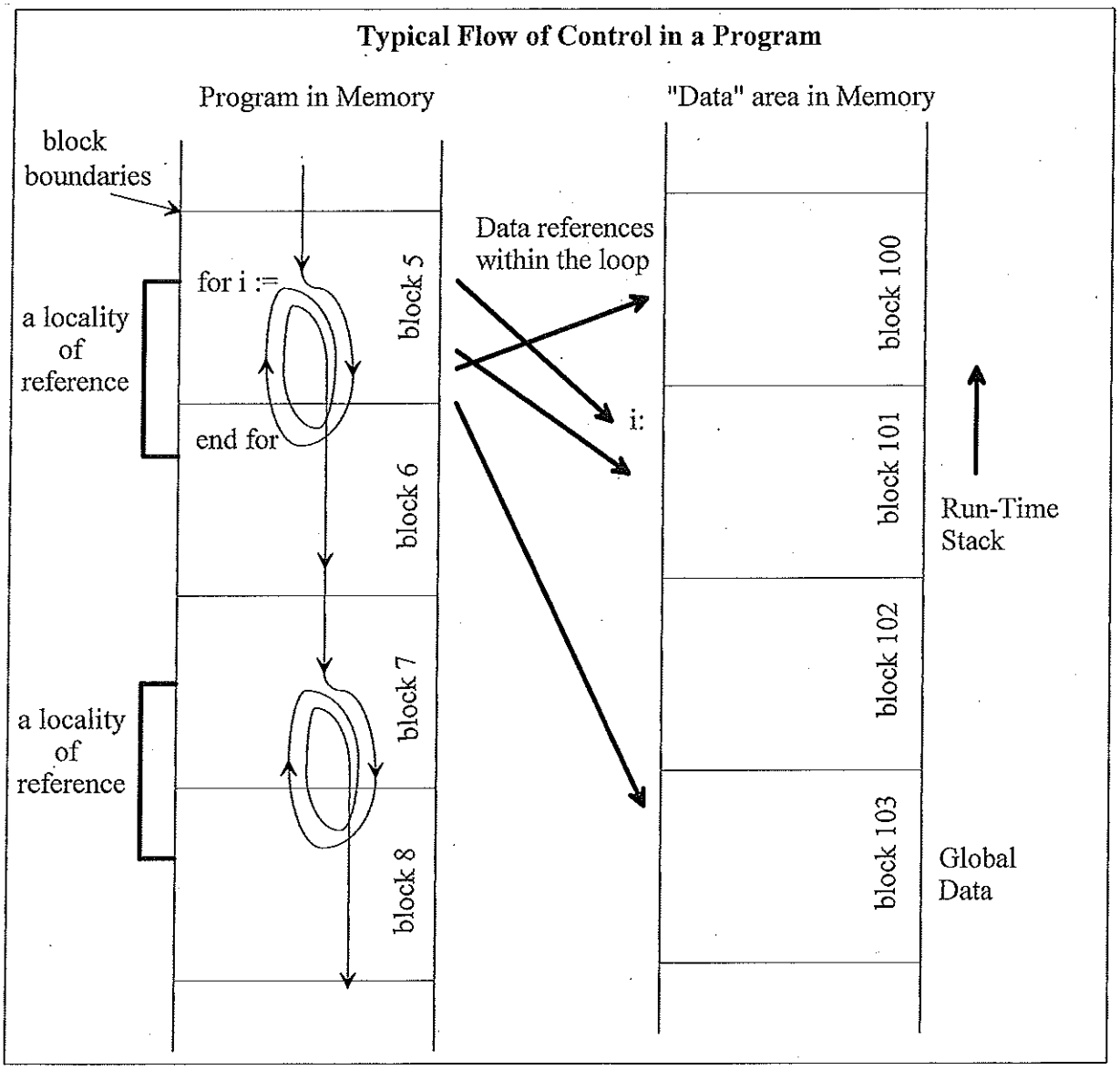
Effective Access Time = 2 + 150 * (1 - 0.99) = 2 + 1.5 = 3.5 ns

(One way to reduce the miss penalty is to not have the cache wait for the whole block to be read from memory before supplying the accessed memory word.)

Fortunately, programs exhibit ***locality of reference*** that helps achieve high hit-ratios:

1) ***spatial locality*** - if a (logical) memory address is referenced, nearby memory addresses will tend to be referenced soon.

2) ***temporal locality*** - if a memory address is referenced, it will tend to be referenced again soon.



Three Types of Cache

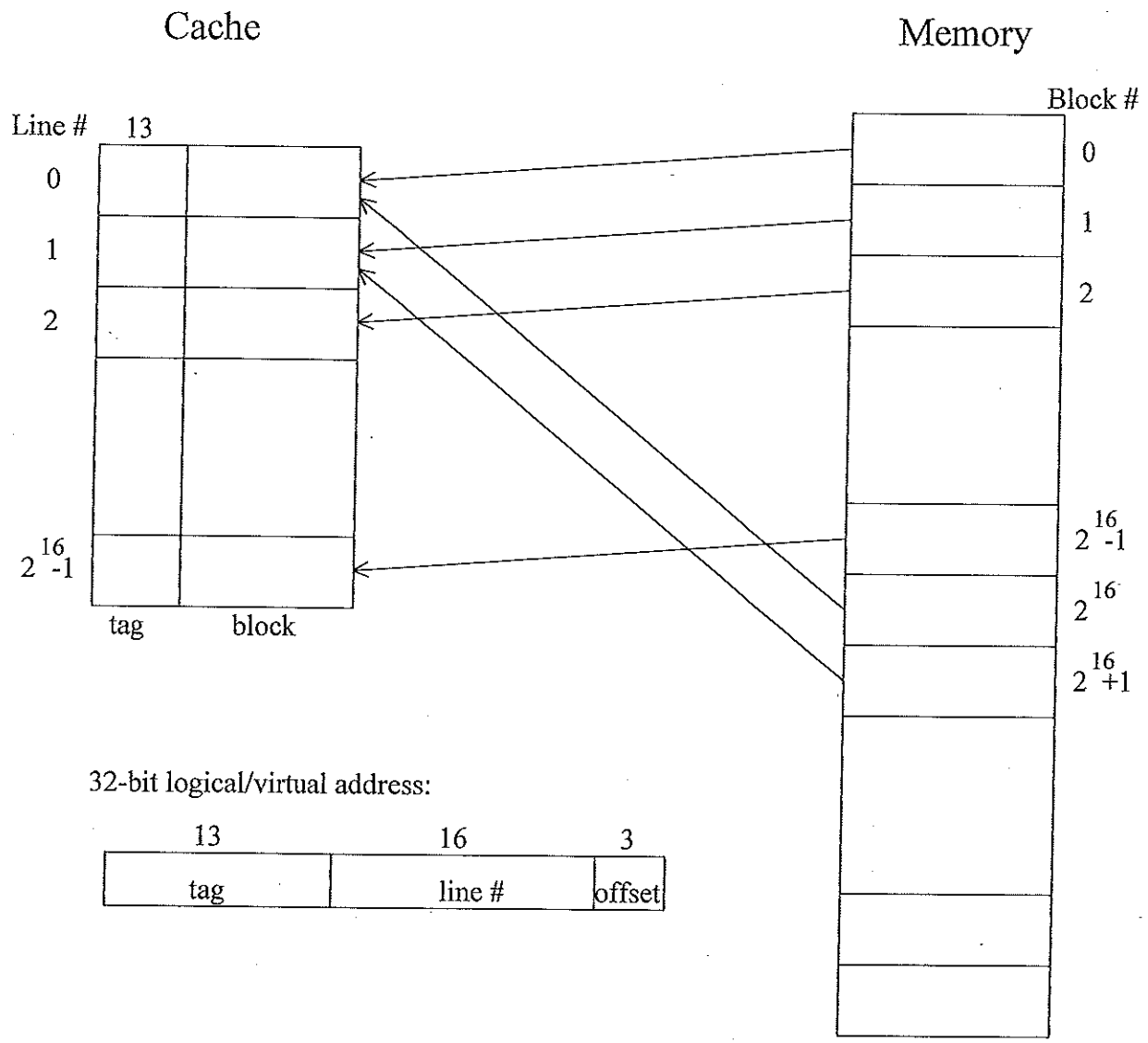
Cache - Small fast memory between the CPU and RAM/Main memory.

Example:

- 32-bit logical/virtual address
- 512 KB (2^{19}) cache size (assume only one level of cache)
- 8 byte per block/line
- byte-addressable memory

$$\text{Number of Cache Line} = \frac{\text{size of cache}}{\text{size of line}} = \frac{2^{19}}{2^3} = 2^{16}$$

1) **Direct-mapped** - a memory block maps to a single cache line



32-bit logical/virtual address:

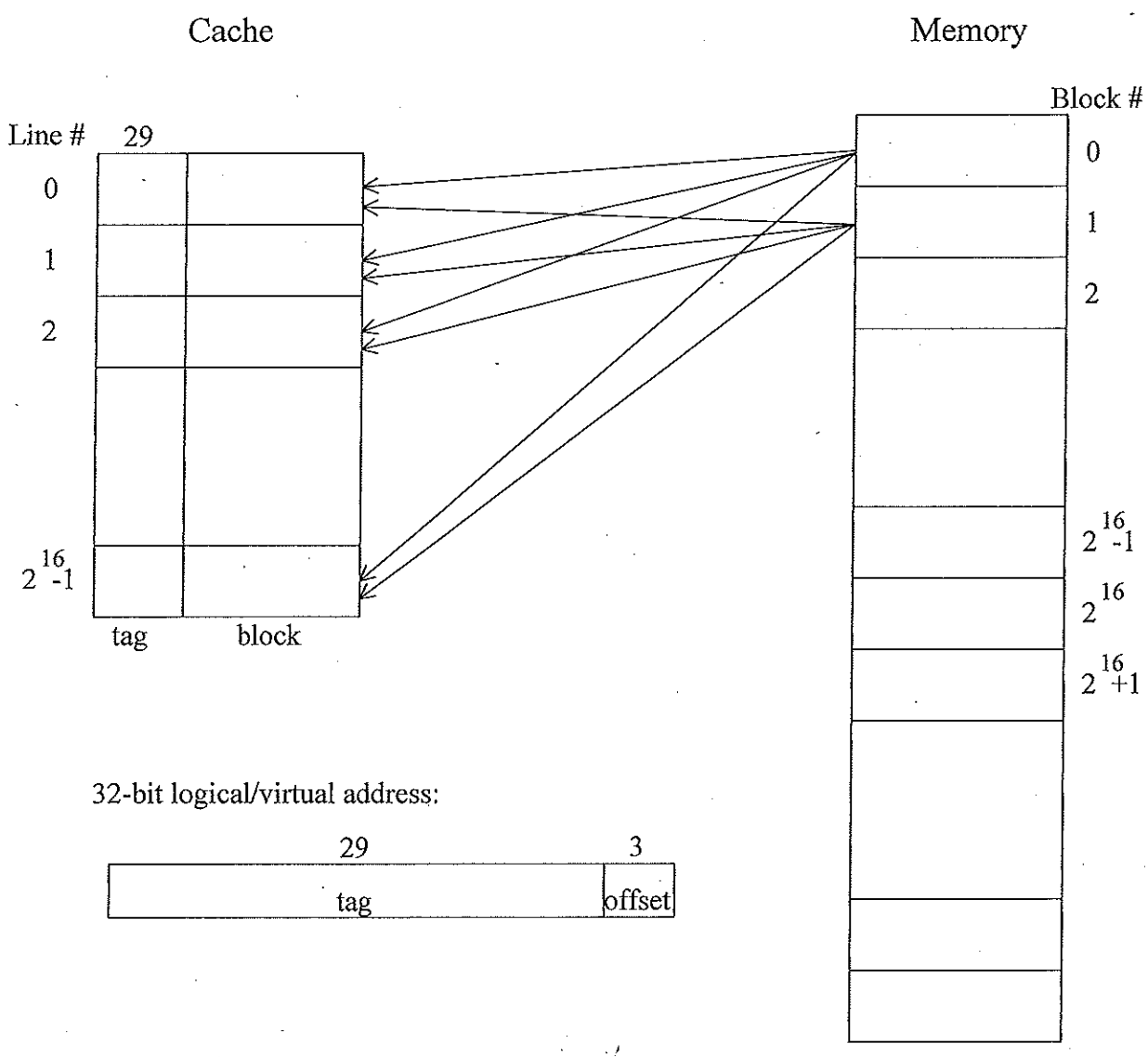
13	16	3
tag	line #	offset

Same Cache Example:

- 32-bit logical/virtual address, byte-addressable memory
- 512 KB (2^{19}) cache size (assume only one level of cache)
- 8 byte per block/line

$$\text{Number of Cache Line} = \frac{\text{size of cache}}{\text{size of line}} = \frac{2^{19}}{2^3} = 2^{16}$$

2) **Fully-Associative Cache** - a memory block can map to any cache line



32-bit logical/virtual address:

Advantage: Flexibility on what's in the cache

Disadvantage: Complex circuit to compare all tags of the cache with the tag in the target address

Therefore, they are expensive and slower so use only for small caches (say 8-64 lines)

Replacement algorithms - on a miss of a full cache, we must select a block in the cache to replace

- LRU - replace the cache block that has not been used for the longest time (need additional usage bits for each line)
- Random - select a block randomly (only slightly worse that LRU and easier to implement)

Same Cache Example:

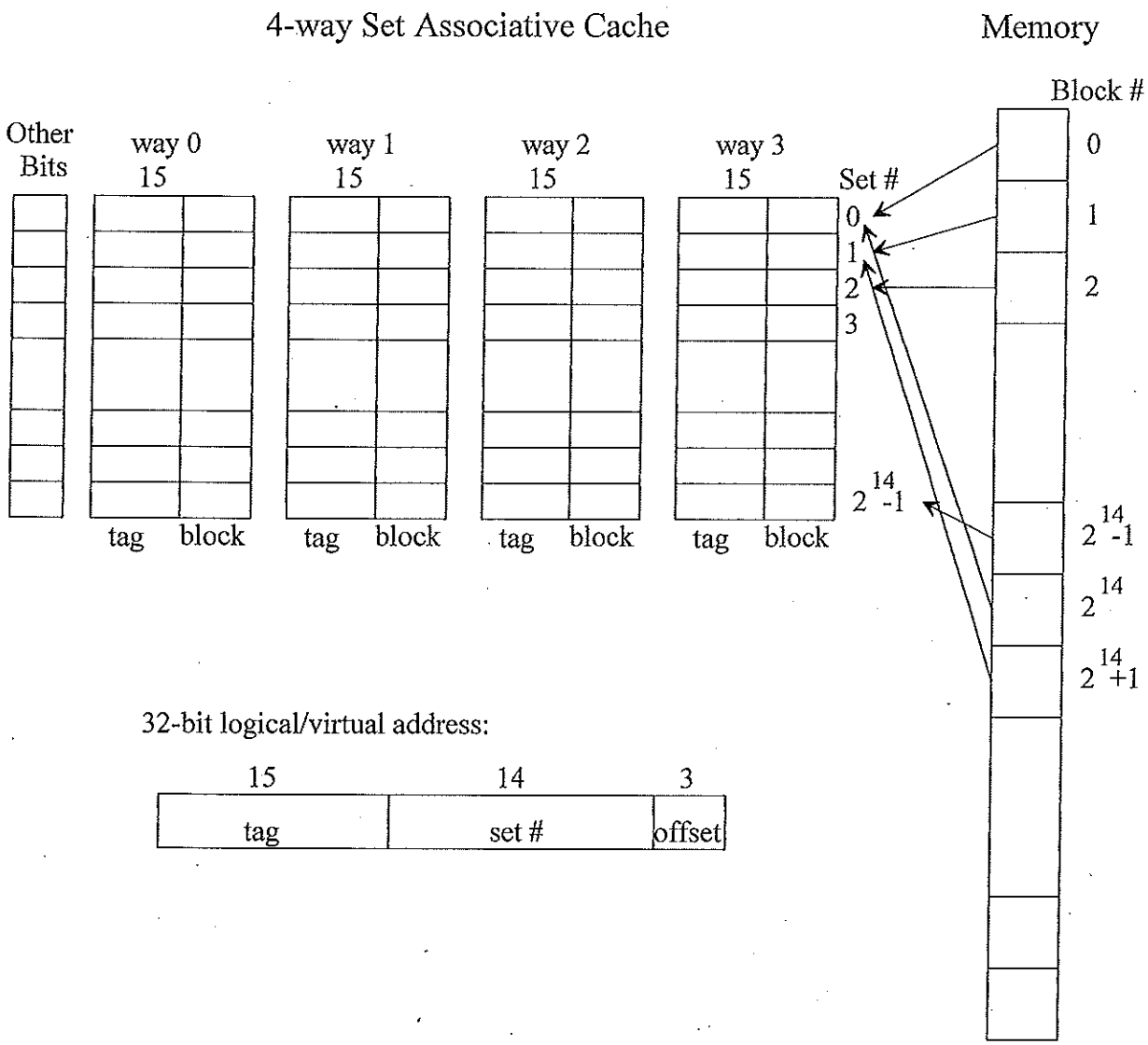
- 32-bit logical/virtual address, byte-addressable memory
- 512 KB (2^{19}) cache size (assume only one level of cache)
- 8 byte per block/line

$$\text{Number of Cache Line} = \frac{\text{size of cache}}{\text{size of line}} = \frac{2^{19}}{2^3} = 2^{16}$$

3) **Set-Associative Cache** - a memory block can map to a small (2, 4, or 8) set of cache lines
Common Possibilities:

- 2-way set associative - each memory block can map to either of two lines in the cache
- 4-way set associative - each memory block can map to either of four lines in the cache

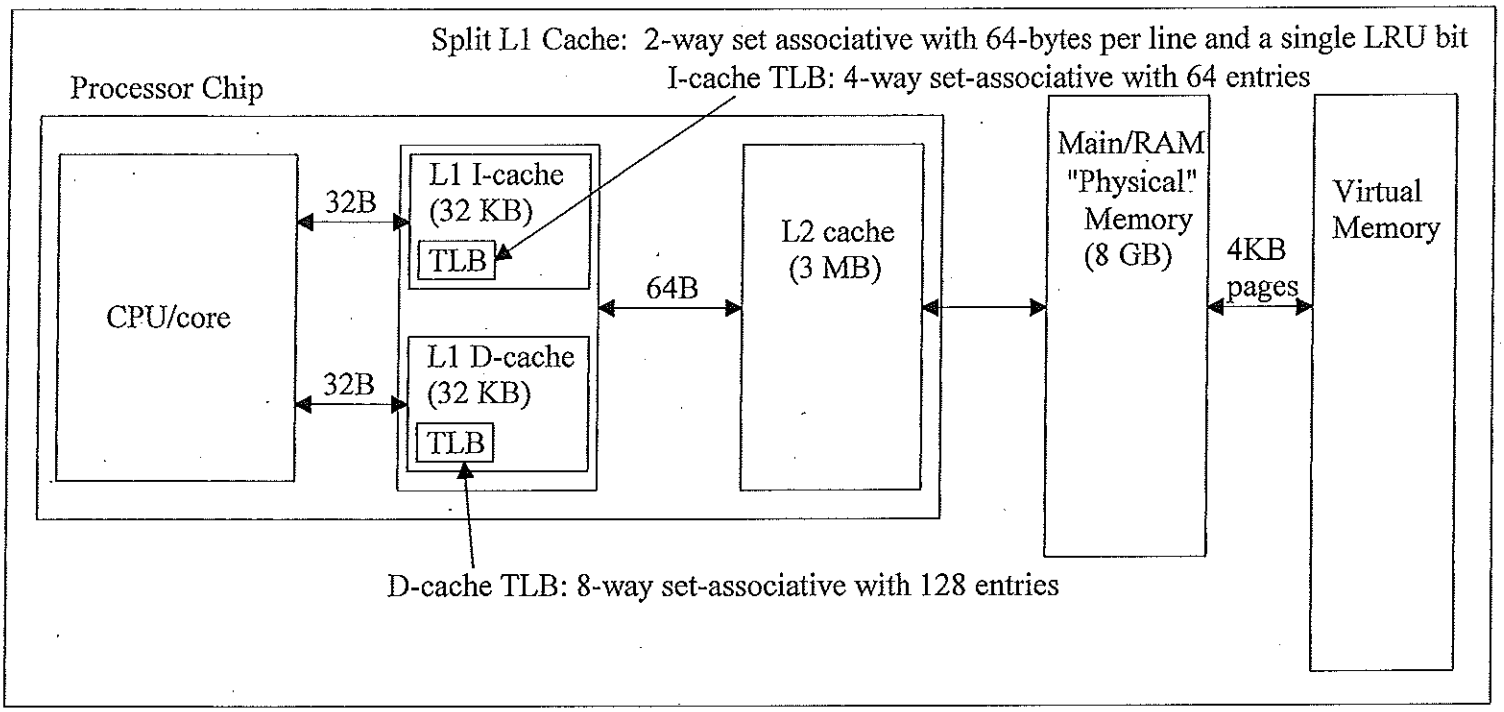
$$\text{Number of Sets} = \frac{\text{number of cache lines}}{\text{size of each set}} = \frac{2^{16}}{4} = \frac{2^{16}}{2^2} = 2^{14}$$



32-bit logical/virtual address:

15	14	3
tag	set #	offset

Typical system view of the memory hierarchy

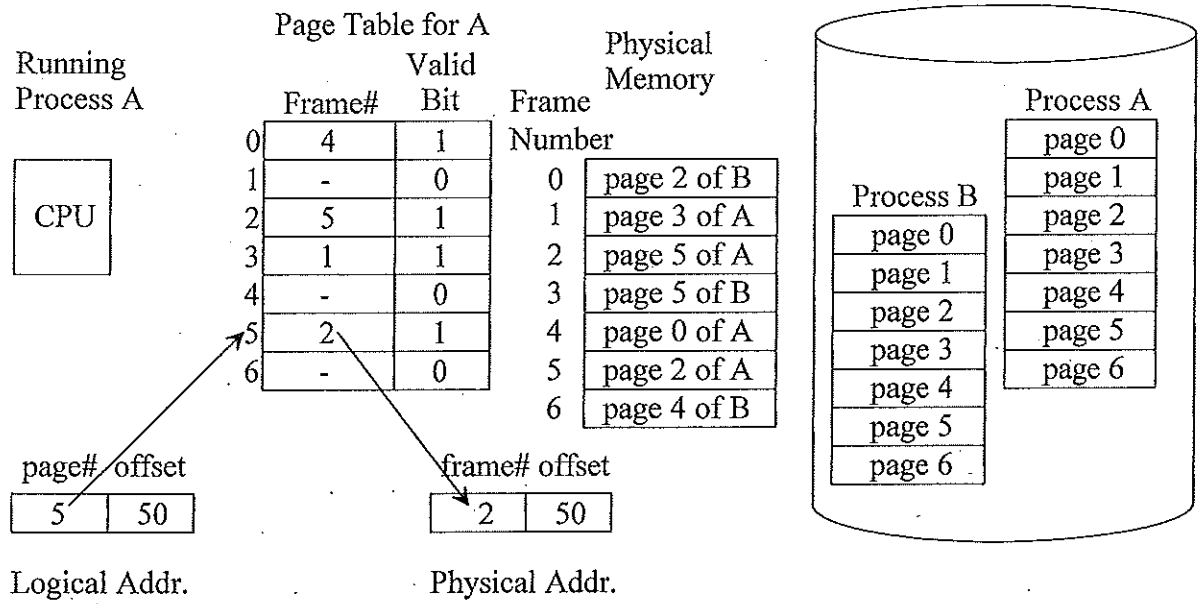


Virtual Memory - programmer views memory as large address space without concerns about the amount of physical memory or memory management. (What do the terms **32-bit** (or **64-bit) operating system** mean?)

Benefits:
1) programs can be bigger that physical memory size since only a portion of them may actually be in physical memory
2) higher degree of multiprogramming is possible since only portions of programs are in memory

An Operating System **goal** with hardware support is to make virtual memory efficient and transparent to the user.

Memory-Management Unit (MMU) for paging



Note: The "Valid" bit is sometimes called the *Resident R-bit*.

Demand paging is a common way for OSs to implement virtual memory. Demand paging ("lazy pager") only brings a page into physical memory when it is needed. A "Valid bit" is used in a page table entry to indicate if the page is in memory or only on disk.

A ***page fault*** occurs when the CPU generates a logical address for a page that is not in physical memory. The MMU will cause a ***page-fault trap*** (interrupt) to the OS.

Steps for OS's page-fault trap handler:

1) Check page table to see if the page is exists in logical address space. If it is invalid, terminate the process; otherwise continue.

2) Find a free frame in physical memory (take one from the free-frame list or replace a page currently in memory).

3) Schedule a disk read operation to bring the page into the free page frame. (We might first need to schedule a previous disk write operation to update the virtual memory copy of a "dirty" page that we are replacing.)

4) Since the disk operations are soooooo sloooooooow, the OS would context switch to another ready process selected from the ready queue.

5) After the disk (a DMA device) reads the page into memory, it invokes an I/O completion interrupt. The OS will then update the PCB and page table for the process to indicate that the page in now in memory and the process is ready to run.

6) When the process is selected by the short-term scheduler to run, it repeats the instruction that caused the page fault. The memory reference that caused the page fault will now succeed.

Performance of Demand Paging

To achieve acceptable performance degradation (5-10%) of our virtual memory, we must have a very low page fault rate (probability that a page fault will occur on a memory reference).

When does a CPU perform a memory reference?

1) Fetch instructions into CPU to be executed
2) Fetch operands used in an instruction (load and store instructions on RISC machines)

Example:

Let p be the page fault rate, and ma be the memory-access time.

Assume that p = 0.02, ma = 50 ns and the time to perform a page fault is 12,200,000 ns (12.2 ms).

$$\begin{pmatrix}\text{effective memory}\\ \text{access time}\end{pmatrix} = \begin{pmatrix}\text{prob. of}\\ \text{no page fault}\end{pmatrix} * \begin{pmatrix}\text{main memory}\\ \text{access time}\end{pmatrix} + \begin{pmatrix}\text{prob. of}\\ \text{page fault}\end{pmatrix} * \begin{pmatrix}\text{page fault}\\ \text{time}\end{pmatrix}$$

$$= (1 - p) * 50\text{ns} + p * 12{,}200{,}000$$
$$= 0.98 * 50\text{ns} + 0.02 * 12{,}200{,}000$$
$$= 244{,}049\text{ns}$$

The program would appear to run very slowly!!!

If we only want say 10% slow down of our memory, then the page fault rate must be much better!

$55 = (1 - p) * 50\text{ns} + p * 12{,}200{,}000\text{ns}$

$55 = 50 - 50p + 12{,}200{,}000p$

$p = 0.0000004$ or 1 page fault in 2,439,990 references

Fortunately, programs exhibit ***locality of reference*** that helps achieve low page-fault rates. Page size is typically 4 KB.

Storage of the Page Table Issues

1) Where is it located?

If it is in memory, then each memory reference in the program, results in two memory accesses; one for the page table entry, and another to perform the desired memory access.

Solution: TLB (Translation-lookaside Buffer) - small, fully-associative cache to hold PT entries
Ideally, when the CPU generates a memory reference, the PT entry is found in the TLB, the page is in memory, and the block with the page is in the cache, so NO memory accesses are needed.
However, each CPU memory reference involves two cache lookups **and** these cache lookups must be done sequentially, i.e., first check TLB to get physical frame # used to build the physical address, then use the physical address to check the tag of the L1 cache.

Alternatively, the L1 cache can contain virtual addresses (called a *virtual cache*). This allows the TLB and cache access to be done in parallel. If the cache hits, the result of the TLB is not used. If the cache misses, then the address translation is under way and used by the L2 cache.

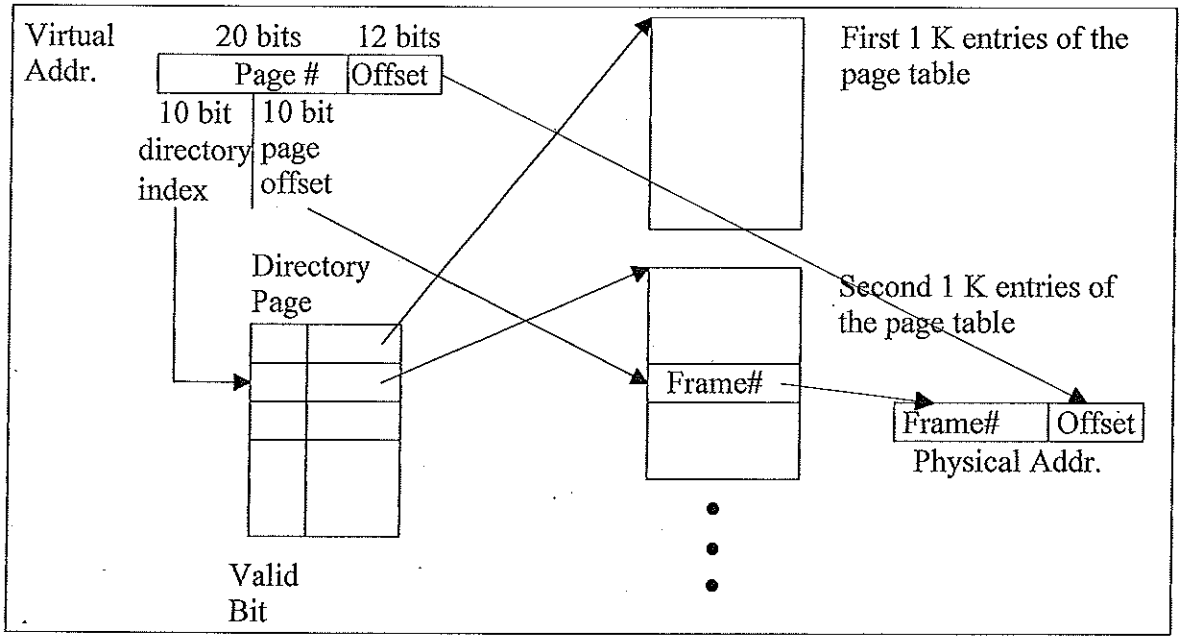
2) Ways to handle large page tables:

Page table for each process can be large
e.g., 32-bit address, 4 KB (2^{12} bytes) pages, byte-addressable memory, 4 byte PT entry

1 M (2^{20}) of page table entries, or 4MB for the whole page table with 4 byte page table entries

A solution:

a) two-level page table - the first level (the "directory") acts as an index into the page table which is scattered across several pages. Consider a 32-bit example with 4KB pages and 4 byte page table entries.

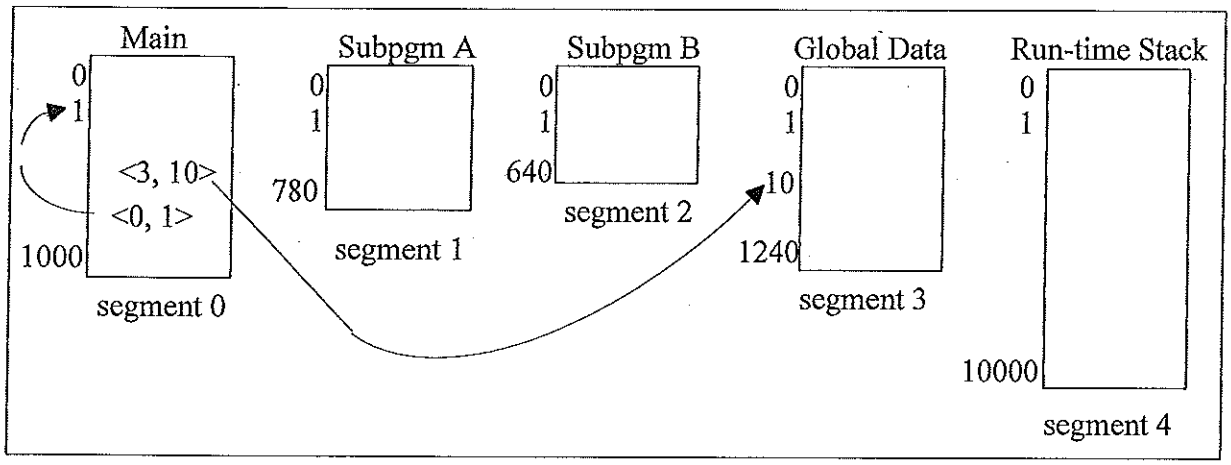


Problem with paging:

1) Protection unit is a page, i.e., each Page Table Entry can contain protection information, but the virtual address space is divided into pages along arbitrary 4KB boundaries.

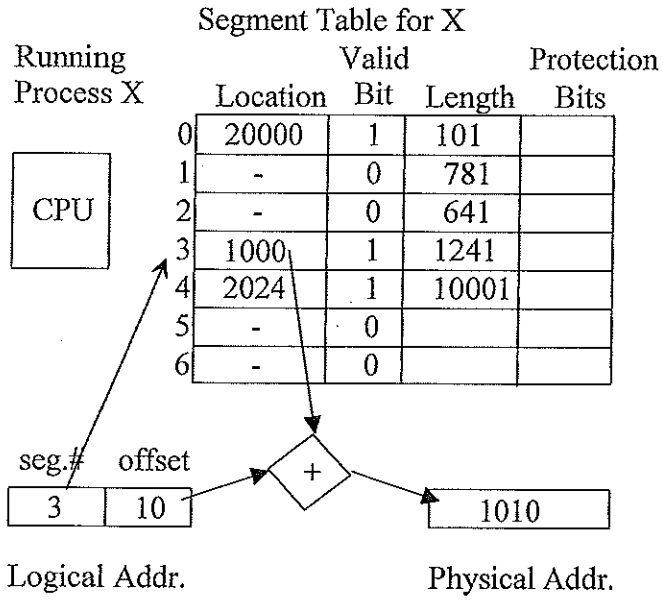
Segmentation - divides virtual address space in terms of meaningful program modules which allows each to be associated with different protection. For example, a segment containing a matrix multiplication subprogram could be shared by several programs.

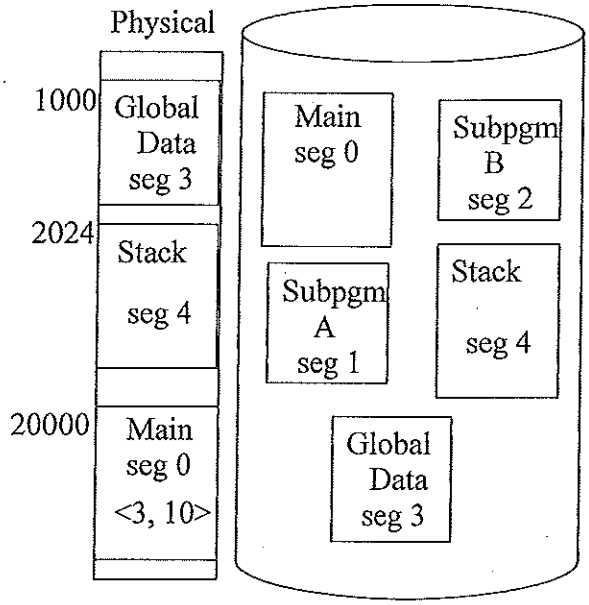
Programmer views memory as multiple address spaces, i.e., segments. Memory references consist of two parts: < segment #, offset within segment >.



As in paging, the operating system with hardware support can move segments into and out of memory as needed by the program.

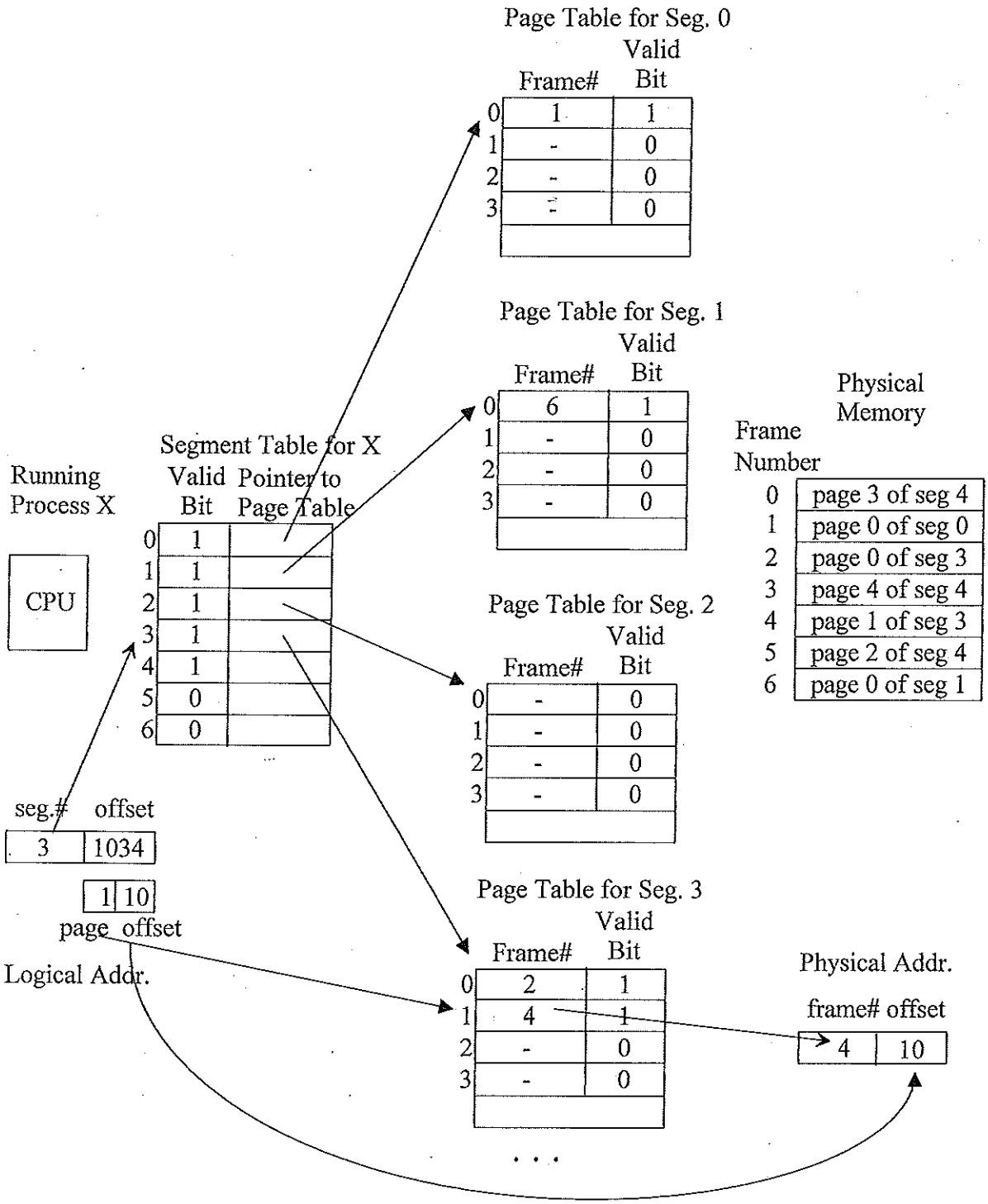
Each process (running program) has its own segment table similar to a page table for performing address translations.



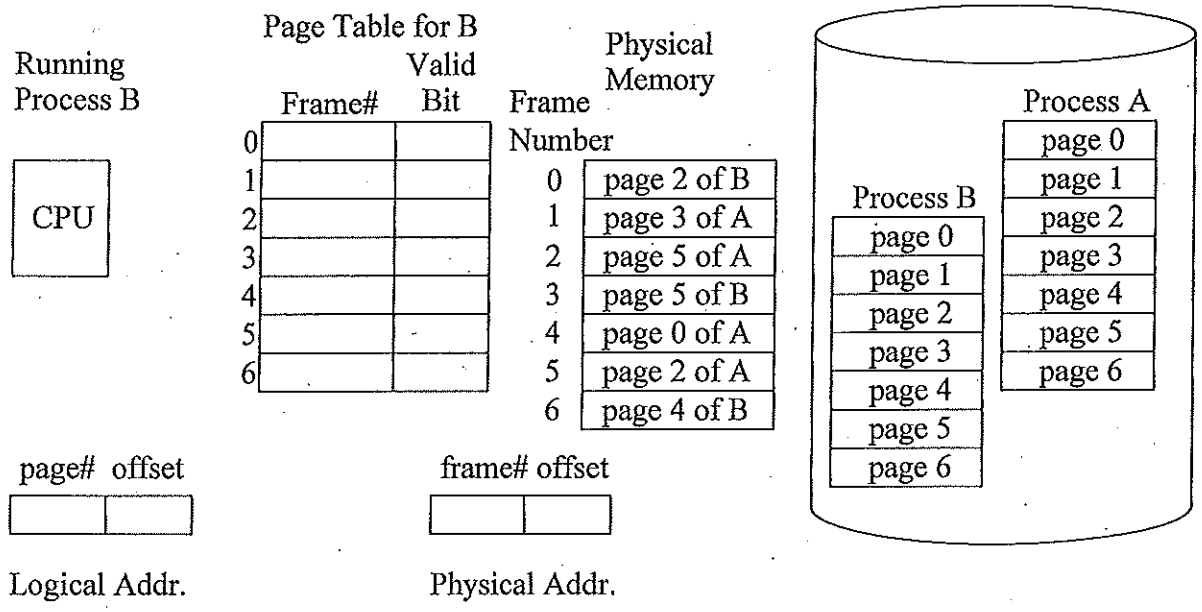


Problems with Segmentation:
1) hard to manage memory efficiently due to external fragmentation
2) segments can be large in size so not many can be loaded into memory at one time

Solution: Combination of paging with segmentation by paging each segment.


Page Table for Seg. 0
Frame# Valid Bit
0 1 1
1 - 0
2 - 0
3 - 0
Page Table for Seg. 1
Frame# Valid Bit
0 6 1
1 - 0
2 - 0
3 - 0
Page Table for Seg. 2
Frame# Valid Bit
0 - 0
1 - 0
2 - 0
3 - 0
Page Table for Seg. 3
Frame# Valid Bit
0 2 1
1 4 1
2 - 0
3 - 0
. . .
Running Process X
CPU
Segment Table for X
Valid Bit
Pointer to Page Table
0 1
1 1
2 1
3 1
4 1
5 0
6 0
seg.# offset
3 1034
1 10
page offset
Logical Addr.
Physical Memory
Frame Number
0 page 3 of seg 4
1 page 0 of seg 0
2 page 0 of seg 3
3 page 4 of seg 4
4 page 1 of seg 3
5 page 2 of seg 4
6 page 0 of seg 1
Physical Addr.
frame# offset
4 10

1. Consider the demand paging system with 1024-byte pages.



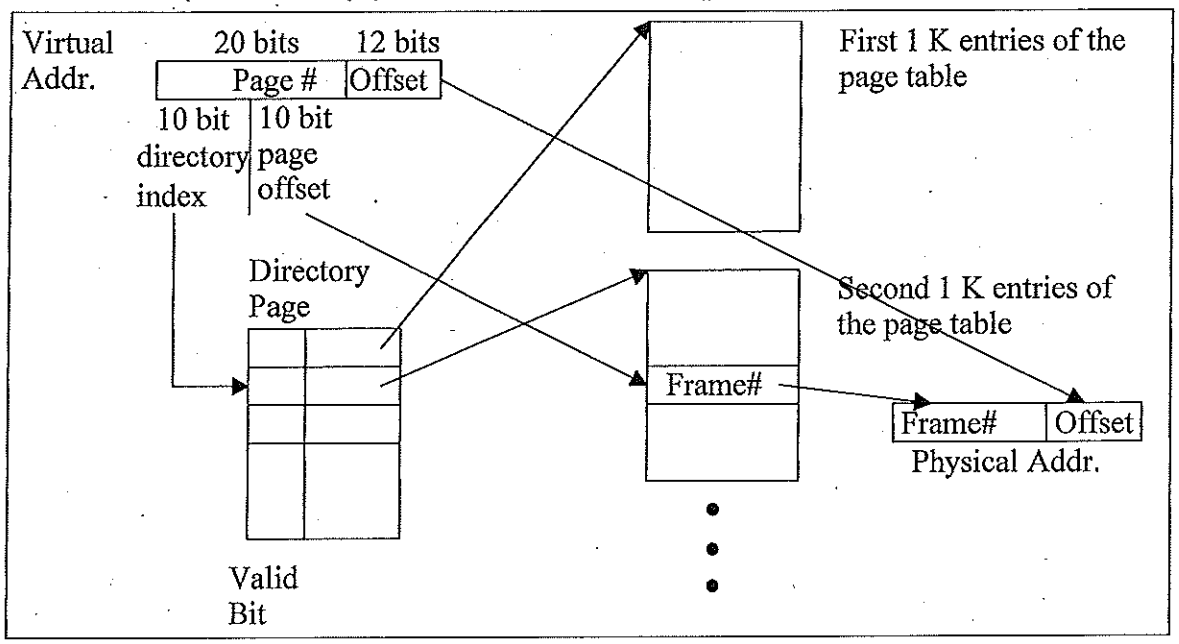
a) Complete the above page table for <u>Process B</u>.

b) If process B is currently running and the CPU generates a logical/virtual address of 2060_{10}, then what would be the corresponding physical address?

2. For a 32-bit address machine with 4 KB (2^{12} bytes) pages and 4 byte page-table entries, how big would the page table be?

3. How does a TLB (translation-lookaside buffer) speed the process of address translation?

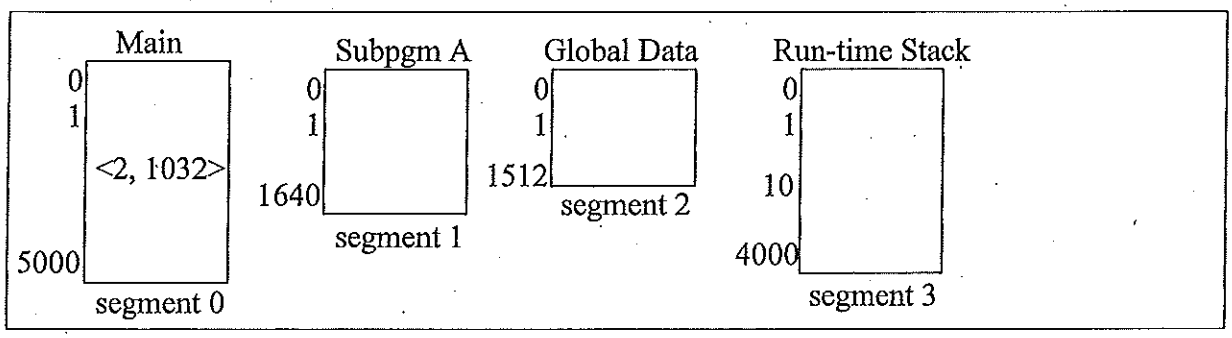
4. 32-bit computers typically had 4KB pages, 4 byte page table entries, and used two-level page tables where the first level (the "directory") acts as an index into the page table which is scattered across several pages.

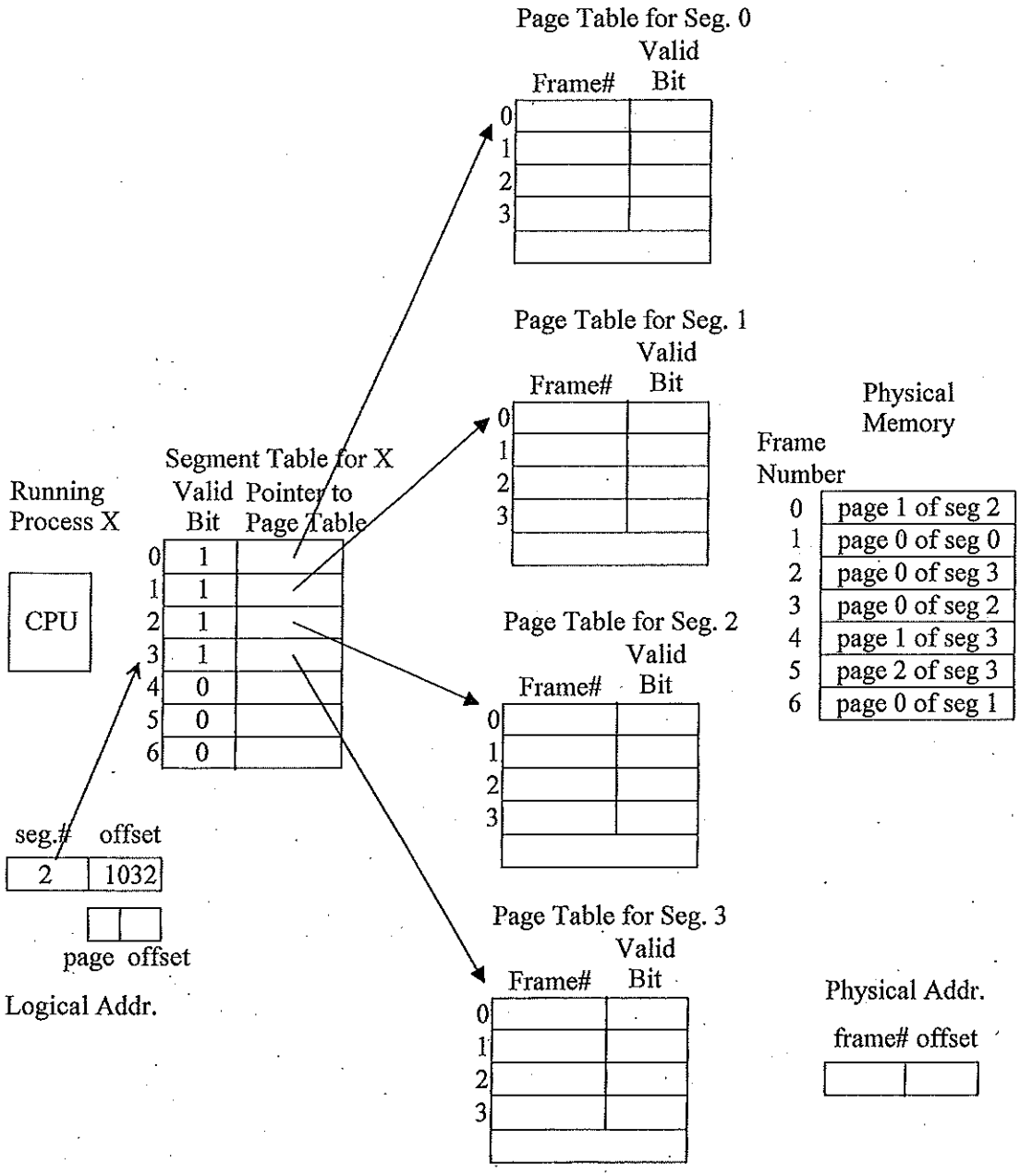


A 64-bit computer might not support a full 64-bit address space. How could a 3-level page table support 42-bit address space?

5. If only segmentation was used, what problems would you predict with moving whole segments into and out of memory?

6. Assuming a page size of 1024 bytes, complete the Page Tables for the pages in memory, and determine the physical address for the logical address <2, 1032>.

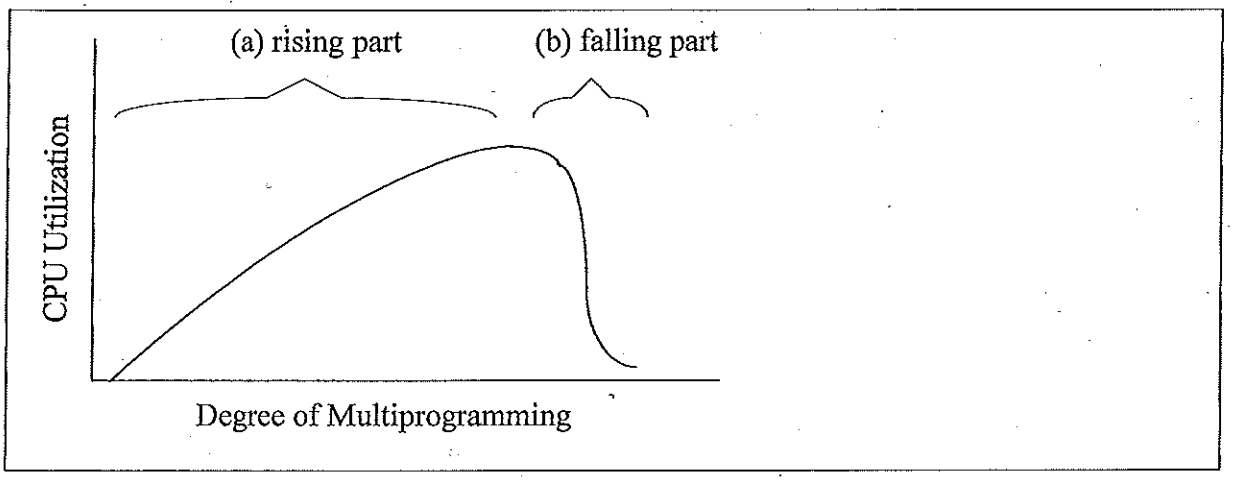




Design issues for Paging Systems

Conflicting Goals:

- Want as many (partial) processes in memory (high degree of multiprogramming) as possible so we have better CPU & I/O utilization ⇒ allocate as few page frames as possible to each process
- Want as low of page-fault rate as possible ⇒ allocate enough page frames to hold all of a process' current working set (which is dynamic as a process changes locality)



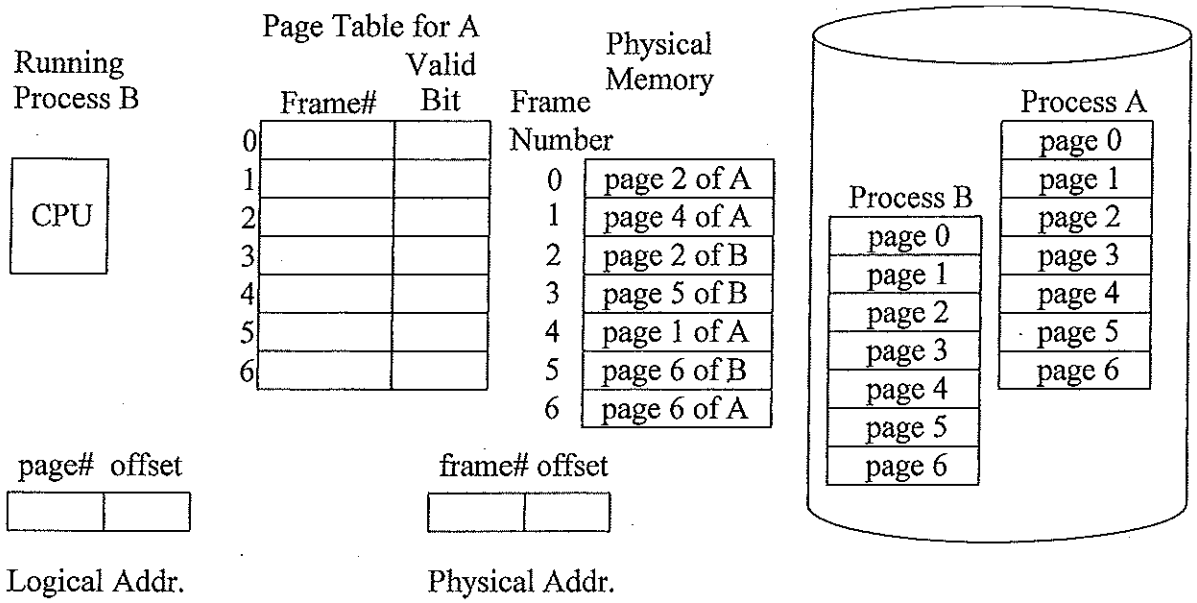
7. Explain the shape of each section indicated on the above curve:

a) (rising part of the curve)

b) (falling part of the curve)

8. There are many similarities between the cache-memory level and memory-disk level of the memory hierarchy, but there are also important differences. For example, a cache miss stalls the running program temporarily, but a page fault causes the running program to turnover the CPU to another program. Why are these cases treated differently by the computer system?

1. Consider the demand paging system with **4096-byte pages.**



a) Complete the above page table for **Process A.**

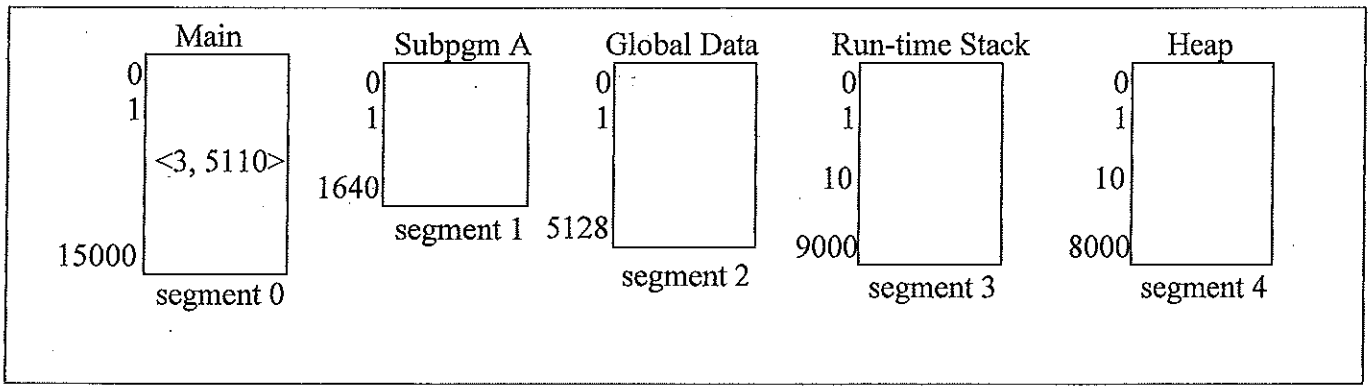
b) If process A is currently running and the CPU generates a logical/virtual address of 8206_{10}, then what would be the corresponding physical address?

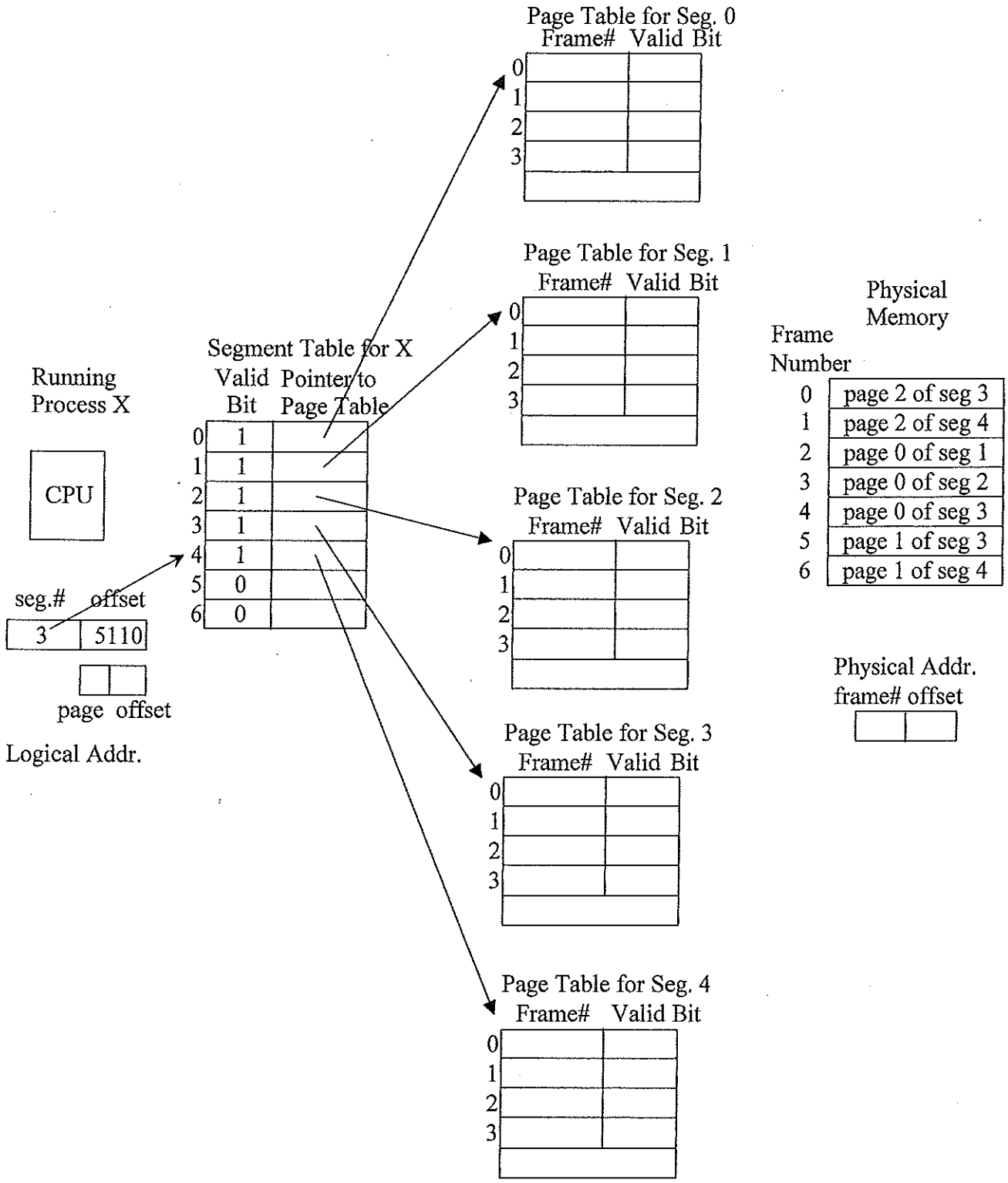
2. Explain how a TLB (translation-lookaside buffer) speeds the process of address translation?

3. What advantages does combining paging and segmentation (i.e., paging of each segment) have over:
a) only paging

b) only segmentation

4. Assuming a **page size of 4096 bytes**, complete the Page Tables for the pages in memory, and determine the physical address for the logical address <3, 5110>.




Page Table for Seg. 0
Frame# Valid Bit
0
1
2
3
Page Table for Seg. 1
Frame# Valid Bit
0
1
2
3
Page Table for Seg. 2
Frame# Valid Bit
0
1
2
3
Page Table for Seg. 3
Frame# Valid Bit
0
1
2
3
Page Table for Seg. 4
Frame# Valid Bit
0
1
2
3
Segment Table for X
Valid Bit
Pointer to Page Table
0 1
1 1
2 1
3 1
4 1
5 0
6 0
Running Process X
CPU
seg.# offset
3 5110
page offset
Logical Addr.
Physical Memory
Frame Number
0 page 2 of seg 3
1 page 2 of seg 4
2 page 0 of seg 1
3 page 0 of seg 2
4 page 0 of seg 3
5 page 1 of seg 3
6 page 1 of seg 4
Physical Addr.
frame# offset

5. (This question deals with the following toy virtual memory system on the next page which is tiny...)
You have a byte-addressable memory with 8 bytes per memory block. The memory management unit has a two-entry TLB (fully-associate cache with a Page # as the tag) and a slower (vague I know) page-table for a process P. The cache is 2-way set-associative and has a total of 4 cache lines (tag bits shown in binary). Assume page size of 16 bytes, so two memory blocks per frame. In the diagram, memory is divided into blocks, where each block's content is represented abstractly by a letter.

Given the system state as depicted above, answer the following questions:

a) How many bits are in a virtual address for process P?
b) How many bits are in a physical address?
c) Show the address format for a logical/virtual address including field names and number of bits.
d) Using your format in part (c), convert the virtual address 50_{10} to binary and put it in the appropriate fields. Now, explain how these fields are used to translate to the corresponding physical address.
e) Show the address format for a physical address including field names and number of bits that are used to check the cache.
f) Given that virtual address 12_{10} translates to physical address 60_{10}. Using your format in part (e), convert the physical address 60_{10} to binary and put it in the appropriate fields. Now, explain how these fields are used to locate physical address 60 in the cache.
g) Given that virtual address 100_{10} is located on virtual page 6, offset 4. Indicate exactly how this address would be translated to its corresponding physical address and how the data would be accessed. Include in your explanation how the TLB, the page table, cache, and memory are used.

TLB

Page	Frame
3	2
6	5

Cache

	Tag	Data	Tag	Data
Set 0	010_2	G	101_2	M
Set 1	100_2	d	011_2	B

Page Table for P

	Frame	Valid Bit
0	3	1
1	0	1
2	-	0
3	2	1
4	1	1
5	-	0
6	5	1
7	-	0
8	7	1
9	-	0
10	6	1
11	-	0
12	-	0
13	-	0
14	4	1
15	-	0

Main Memory

Frame		Block
0	C	0
	D	1
1	I	2
	J	3
2	G	4
	H	5
3	A	6
	B	7
4	c	8
	d	9
5	M	10
	N	11
6	U	12
	V	13
7	Q	14
	R	15

Virtual Memory for Process P

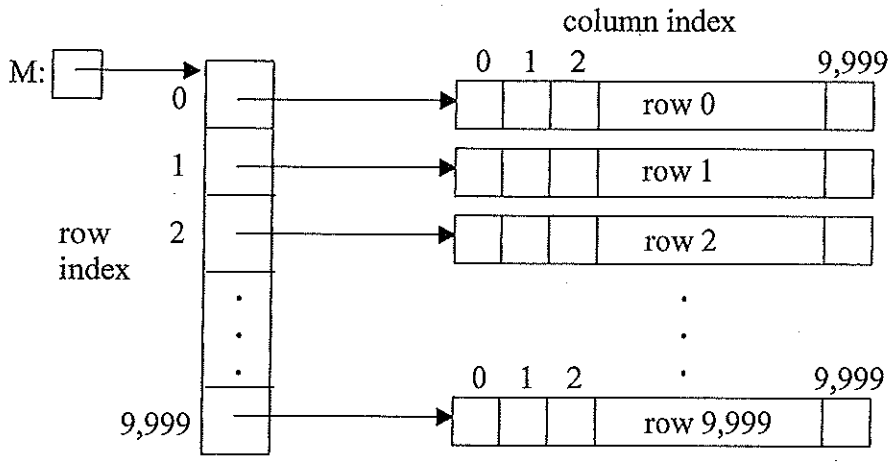
Page		Block
0	A	0
	B	1
1	C	2
	D	3
2	E	4
	F	5
3	G	6
	H	7
4	I	8
	J	9
5	K	10
	L	11
6	M	12
	N	13
7	O	14
	P	15
8	Q	16
	R	17
9	S	18
	T	19
10	U	20
	V	21
11	W	22
	X	23
12	Y	24
	Z	25
13	a	26
	b	27
14	c	28
	d	29
15	e	30
	f	31

6. Consider the following two sections of C code that both sum the elements of a 10,000 x 10,000 two-dimensional array M which contains floating points.

Code A	Code B

sum = 0.0;
for (r = 0; r < 10000; r++)
  for (c = 0; c < 10000; c++)
  sum = sum + M[r][c];
	sum = 0.0;
	for (c = 0; c < 10000; c++)
	  for (r = 0; r < 10000; r++)
	  sum = sum + M[r][c];

Explain why Code A takes 1.27 seconds while Code B takes 2.89 seconds. Hint: C uses row-major ordering to store two-dimensional arrays i.e.,



7. Translate the following high-level language code segment to MIPS assembly language. Use the register mappings as indicated in the comments.

```
CNT_LOW = 0        # $t0 is CNT_LOW
CNT_MED = 0        # $t1 is CNT_MED
CNT_OTHER = 0      # $t2 is CNT_OTHER

INPUT X            # use syscall read_int  (li $v0, 5 before syscall which returning the value in $v0)
WHILE  X > 0  DO
   IF X < 10 THEN
      CNT_LOW = CNT_LOW + 1
   ELSE IF X >= 20 AND X < 50 THEN
      CNT_MED = CNT_MED + 1
   ELSE
      CNT_OTHER =CNT_OTHER + 1
   END IF
   INPUT  X
END WHILE
```

8. Consider the following **recursive** subprogram binarySearch that **finds the index of a target value in an array.** It should return -1 if the target is not found in the array.

```
function binarySearch (address of array, integer startIndex, integer endIndex, integer target) return an array index
    local integer variable:  midIndex
    if startIndex > endIndex then
        return -1
    else
        midIndex = (startIndex + endIndex) / 2    (use srl by one position to divide by 2)
        if target == array[midIndex] then
            return midIndex
        else if target < array[midIndex] then
            return binarySearch(array, startIndex, midIndex-1, target)
        else
            return binarySearch(array, midIndex+1, endIndex, target)
        end if
    end if
end function binarySearch
```

a) Using the MIPS register conventions ($a0-$a3, $t0-$t9, $s0-$s7, $v0-$v1, $sp, $ra), what registers would be used to pass each of the following parameters into findMin:

array	startIndex	endIndex	target

b) Using the MIPS register conventions, which of these parameters ("array", "startIndex", "endIndex", "target") should be moved into $s-registers?

c) (2 points) Using the MIPS register conventions, what register should be used for the local variable "midIndex"?

d) (12 points) For the registers indicated above, write the assembly language code for the complete function binarySearch.

The final for Computer Organization is from 10:00 - 11:50 AM on Thursday December 15 in ITT 328. The test will be closed book and notes, except for **three** 8.5" x 11" sheet of paper (front and back) with notes, the green MARIE Assembly Language handout (available on eLearning: Unit 2 folder Lecture 10), and the your MIPS Assembly Language Guide (available on eLearning: Unit 3 folder Lecture 16).

About 75% of the Final will focus on the material since the last test, and about 25% will focus on the material from tests 1 and 2. (At most 15% combined from Hardware Support for the Operating System sections 8.1-8.2, I/O sections 7.1-7.4, General Idea of Pipelining Processors (section 5.5) General Idea of Cache (section 6.1-6.4) and General Idea of Virtual Memory (section 6.5))

MIPS Assembly Language
MIPS Processor Architecture: registers, register conventions, addressing modes, memory layout
Basic MIPS Instruction Set: loads/stores, arithmetic instructions, logical instructions, shift/rotate instructions, branch/jump instructions
SPIM Assembler Directives: .data, .text, .word, .globl, .asciiz
MIPS Instruction Set: three ML instruction formats
Subprograms: MIPS Register conventions
MIPS Logical, Shift/Rotate Instructions
SPIM I/O and other System Calls:
SPIM Assembler Directives: .asciiz, .ascii, .align, .space
Arrays: element addressing 1-d, 2-d, 3-d, and higher
Walking pointer through an array

In addition to knowledge about the above concepts, the following assembly-language programming skills are to be tested too:
1) translate high-level language control statements (while, for, if, etc.) into MIPS assembly language (be able to handle complex Boolean expressions involving ANDs, ORs, etc.)
2) translate high-level language code containing array accesses into MIPS assembly language
3) use MIPS register conventions to decide which arguments/parameters and local variables should be stored in caller-saved ($a and $t-registers) or callee-saved ($s-registers)
4) translate high-level language subprograms into MIPS assembly language (passing parameters into the subprogram using the $a registers, building the call frame on the run-time stack if necessary, save $s and $ra registers if necessary, passing the value returned by a function in the $v0 register, restoring $s and $ra registers if necessary, jr back to the caller)

Hardware Support for the Operating System sections 8.1-8.2
You should understand the general concept of how the operating system with hardware support provide protection from user programs that:
1. go into infinite loops
2. try to access memory of other programs or the OS
3. try to access files of other programs

This involves understanding the concepts of
1. CPU timer
2. dual-mode operation of the CPU, and idea of privileged instructions and non-privileged instructions
3. ways to restrict a user program to its allocated address space
4. ways to restrict a user program from issuing I/O commands to I/O devices

I/O sections 7.1-7.4
General I/O characteristics
I/ O Controller role and function
I/O address mapping: Isolated-I/O vs. memory-mapped I/O
I/O Data Transfer: programmed I/O, interrupt-driven I/O, and direct-memory access (DMA)
General interrupt mechanism
Usage of interrupts by the hardware/operating system to restrict a user program's activities

Memory Hierarchy:
General Idea of Cache (section 6.1-6.4): 3 types of cache (direct-mapped, fully-associative, set-associative) and address format of each; split vs. unified L1 cache
General Idea of Virtual Memory (section 6.5): paging, page table, TLB, logical-to-physical address translation, multi-level page table; segmentation; combined paging of each segment

Misc. material:
Process control blocks (PCB) and OS queues for I/O and process scheduling
General Idea of Pipelining Processors (section 5.5)
General Idea of Superscalar processors (section 9.4.1 and Pentium IV p. 249)