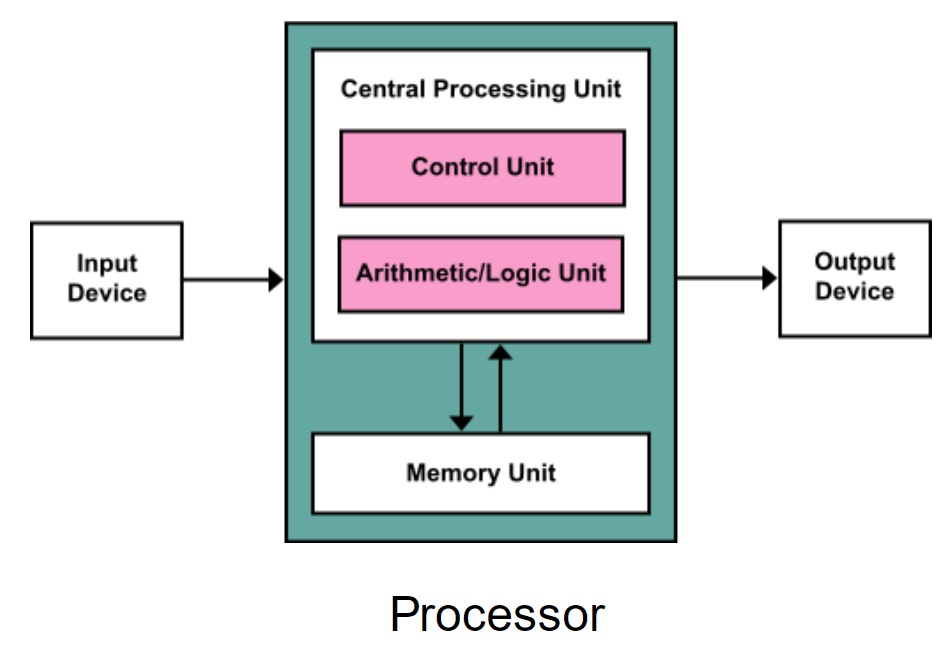


Moore’s Law – IC resources double every 18-24 months  
-number of transistors per square inch on an Integrated Circuit (IC) doubles every 18-24 months

Compilers translate programs written in a high level language (C++, Java) into assembly language

Assembler translate from assembly language to machine language, i.e. add A,B into 1001010100101110

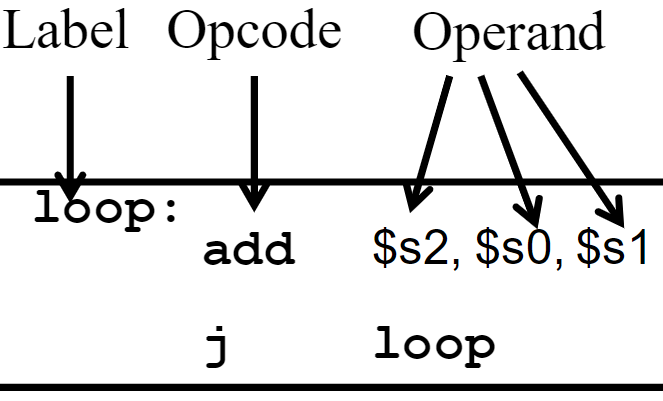
Von Neumann Model

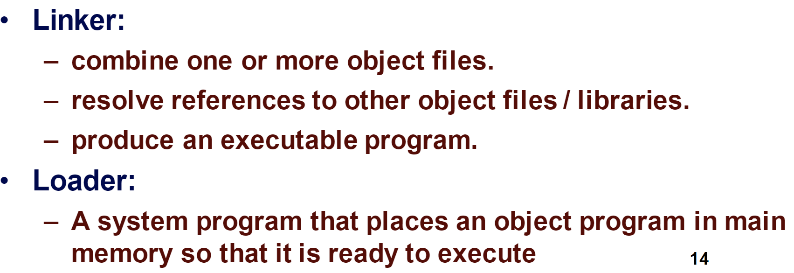
ALU performs data manipulation operations  
Control directs operation of other components  
Memory store instructions and data  
I/O interface to environment

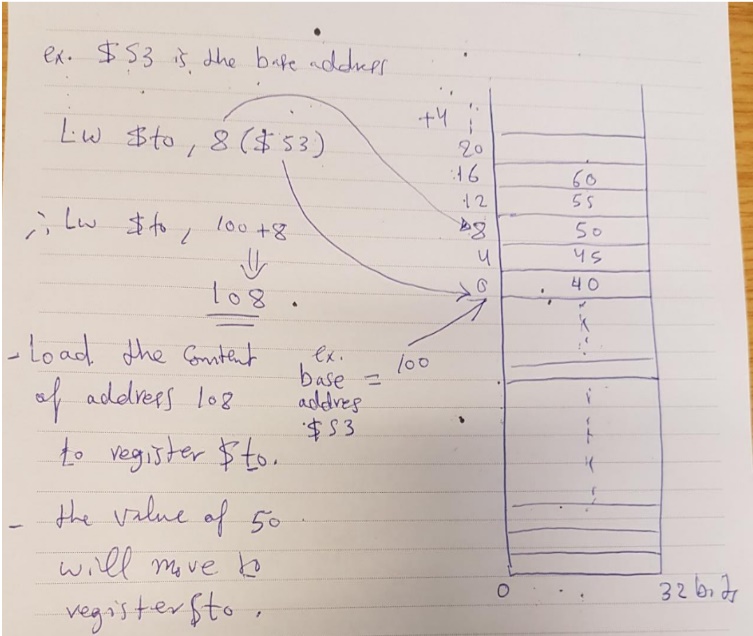
Clock cycles:  
CPU execution time= CPU clock cycles for a program \* clock cycle time  
 CPU clock cycles for a program / clock rate   
CPU clock cycles for a program= instructions for a program \* average clock cycle per instruction (CPI)

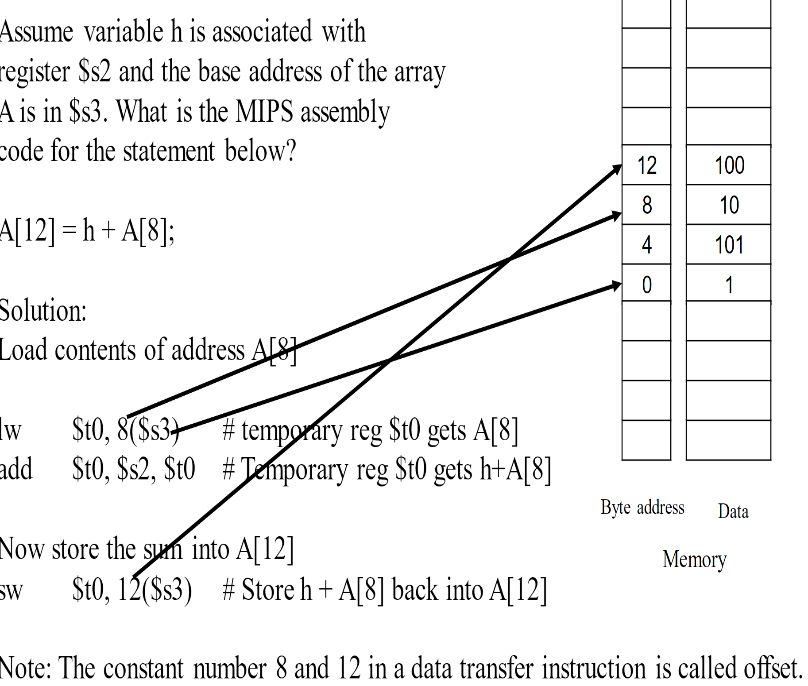
CPU execution time= Instruction count \* CPI \* Clock Cycle time   
CPU execution time= (Instruction count \* CPI)/Clock Rate

MIPS – Microprocessor without Interlocked Pipeline Stages is a reduced instruction set computer (RISC) instruction set architecture (ISA)  
Assembly – code written faster, better understanding of how things work & embedded systems/network processors still need assembly language. Must be rewritten for other computer architectures, not portable, longer codes to write.

Opcode- instruction (arithmetic, branch)  
Operand- perform the command on? (registers, memory, constant)

memory-reference: lw- load word   
sw- store word  
arthimetic-logical: add, sub, and, or, slt (set on less than)  
branching: beq(branch if equal), j(jump)

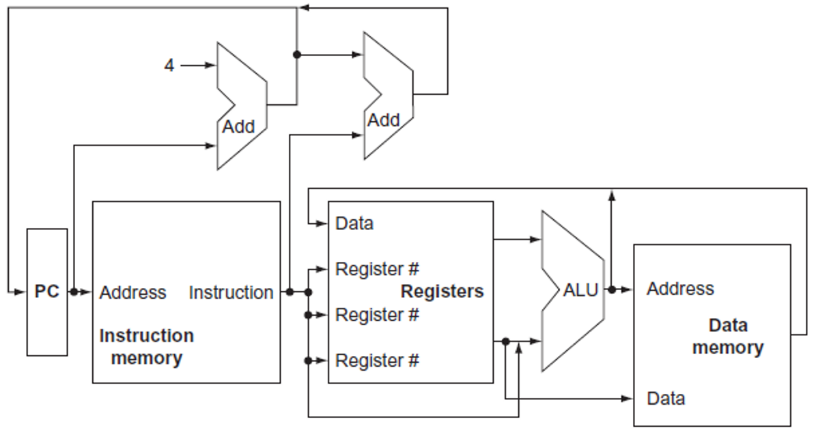
Register size= 32 bits (word), $s0-> $s7, $t0->$t9 (temp registers), $0 always== 0.  
F=(g+i)-(i+j)  
ADD $t0,$s1,$s2 | ADD $t1,$s3,$s4 | SUB $s0, $t0, $t1

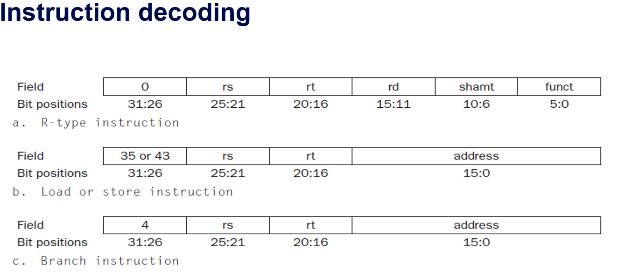
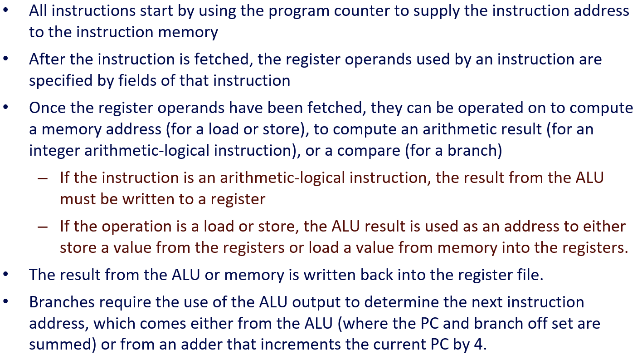


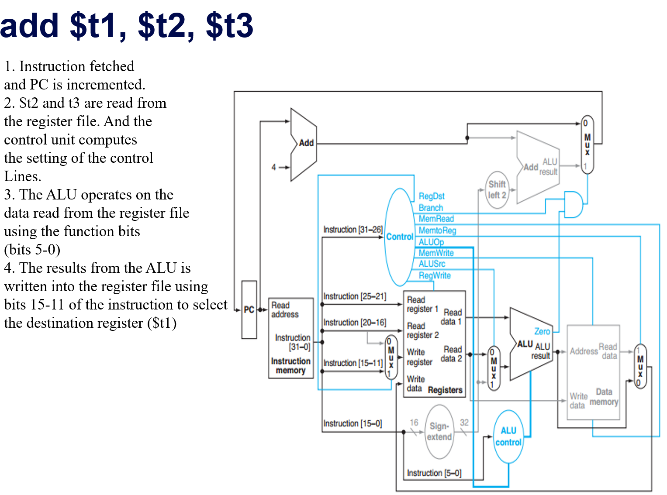
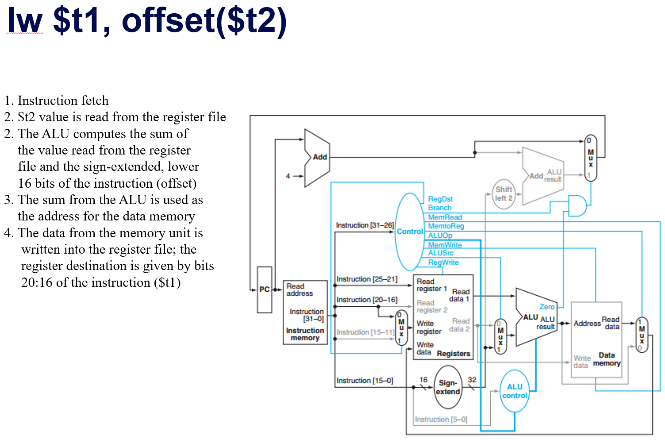
Conditional branches:  
beq register1, register2, L1 (Go to L1 if register1=register2)  
bne register1, register2, L1 (Go to L1 If register1=/=register2)  
Unconditional branches:  
j Exit  
j $ra  
slt $t0, $s3, $s4 (set $t0 to 1 if $s3 is less than $s4, otherwise set $t0 to 0

If (i==j) f=g+h; else f=g-h  
 bne $s3, $s4, step1 #go to Else if i =/= j  
 add $s0, $s1, $s2 #f=g+h (skipped if above true)  
 j Exit #jump to Exit  
step1: sub $s0, $s1, $s2 #f=g-h  
Exit:

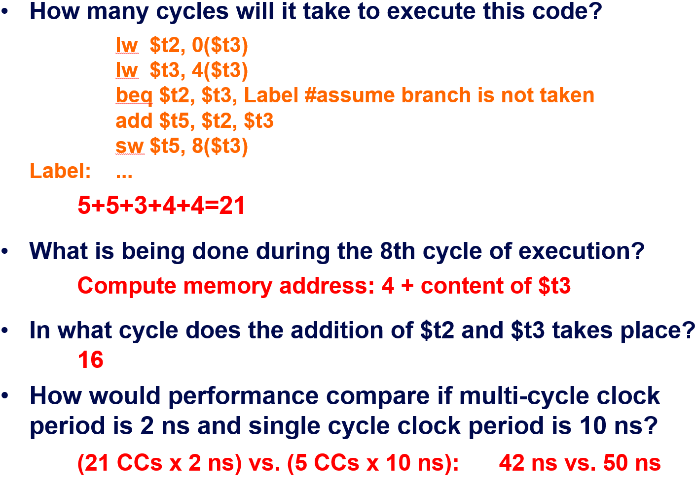
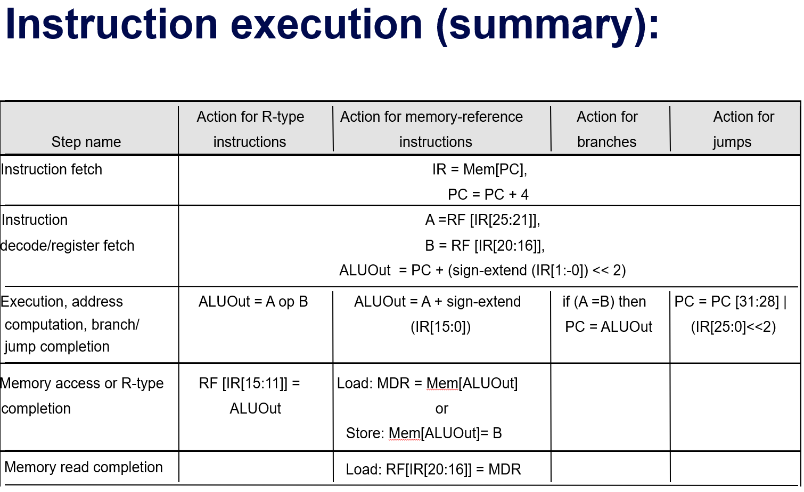
While(save[i]==k): i=i+1;  
i=$s3, k=$s5, save=$s6  
Loop: add $t1, $s3, $s6 #$t1 = address of save[i]  
 lw $t0, 0($t1)  
 bne $t0, $s5, Exit #go to Exit if save[i] =/= k  
 addi $s3, $s3, 1 #i=i+1  
 j Loop #go to Loop  
Exit:  
OR  
Loop: lw $t0, $s3($s6) #get content of address of save[i]  
 bne $t0, $s5, Exit #go to Exit if save[i] =/=k  
 addi $s3, $s3, 1 #i=i+1  
 j Loop #jump to Loop  
Exit:

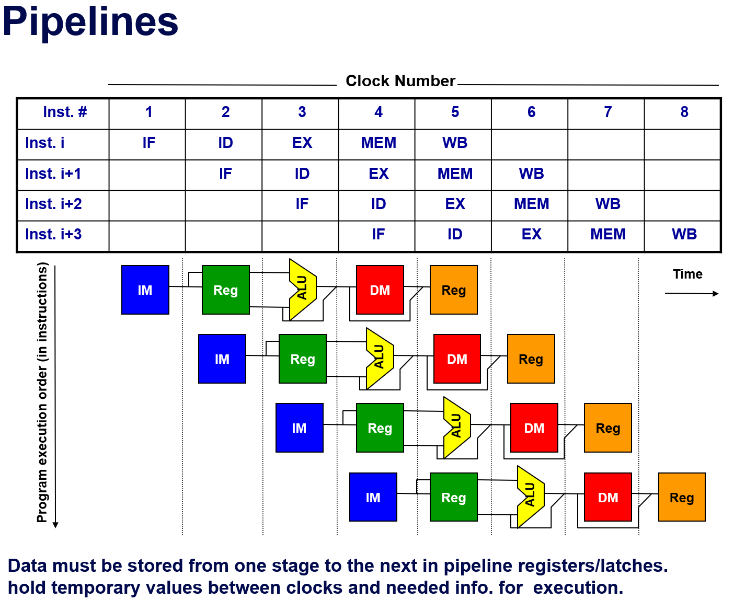
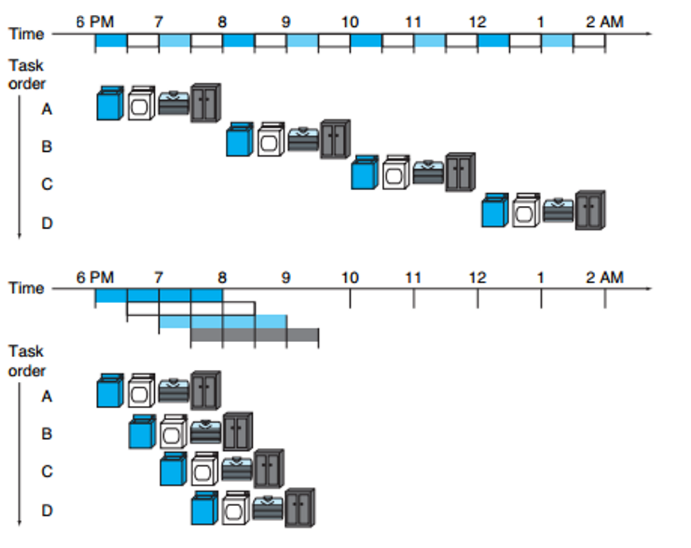
PC: Program counter – Register to hold the address of current instruction being executed. Jal (unconditional jump instruction) saves PC+4 in $ra  
  
Steps:   
Send program counter (PC) to the memory that contains the code and fetch instruction from memory  
Read one or two registers, using fields of the instruction to select the registers to read (load is 1 register, most others are two)  
Memory reference instructions use ALU for address calculations, arithmetic-logical instructions for operation execution, branches for comparison (JUMP doesn’t use ALU)  
THEN:  
Memory-reference instructions will need to access the memory to read data for a load or write data for store  
Arithmetic-logical or load instruction must write data from ALU or memory back into register  
Branch instruction may need to change instruction address based on comparison; otherwise the PC should be incremented by 4 to get the address of next instruction

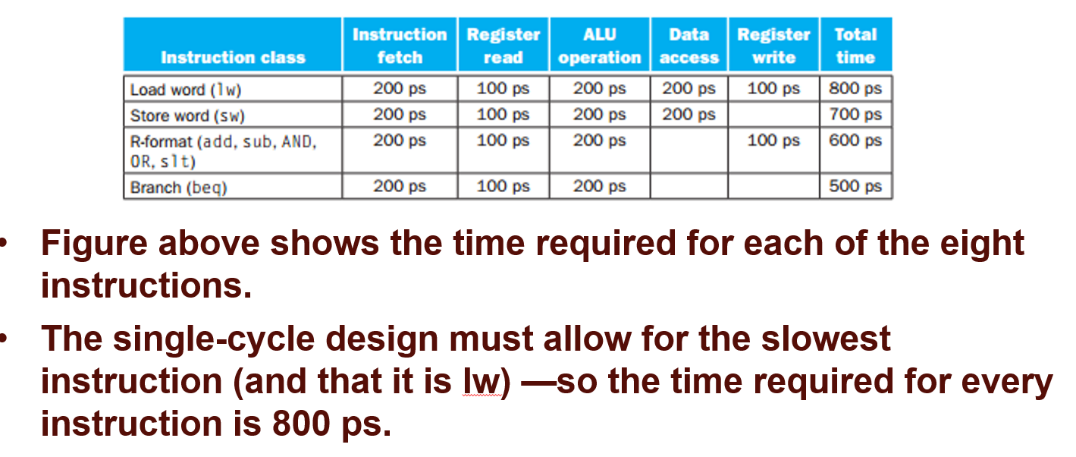


Op field/Opcode, reg1, reg2, ----  
31:26 – Opcode  
25:21, 20:16 – RS, RT (Register1,2)  
25:21 RS (base register load and store instructions)  
15:0 ADDRESS (16 bit offset for branch equal, load, store)  
20:16 RT (load destination register), 15:11 RD (r type instruction)

Single Cycle Implementation: CPI = 1  
Clock Cycle = propagation delay of the longest datapath operation among all instruction types  
  
R-Type: mem+RF+ALU+RF =6ns  
LW: mem+RF+ALU+mem+RF =8ns  
SW: mem+RF+ALU+mem =7ns  
BEQ: mem+RF+ALU;max{Add,mem+add} =5ns  
Clock Cycle = 8ns = longest datapath operation (LW)

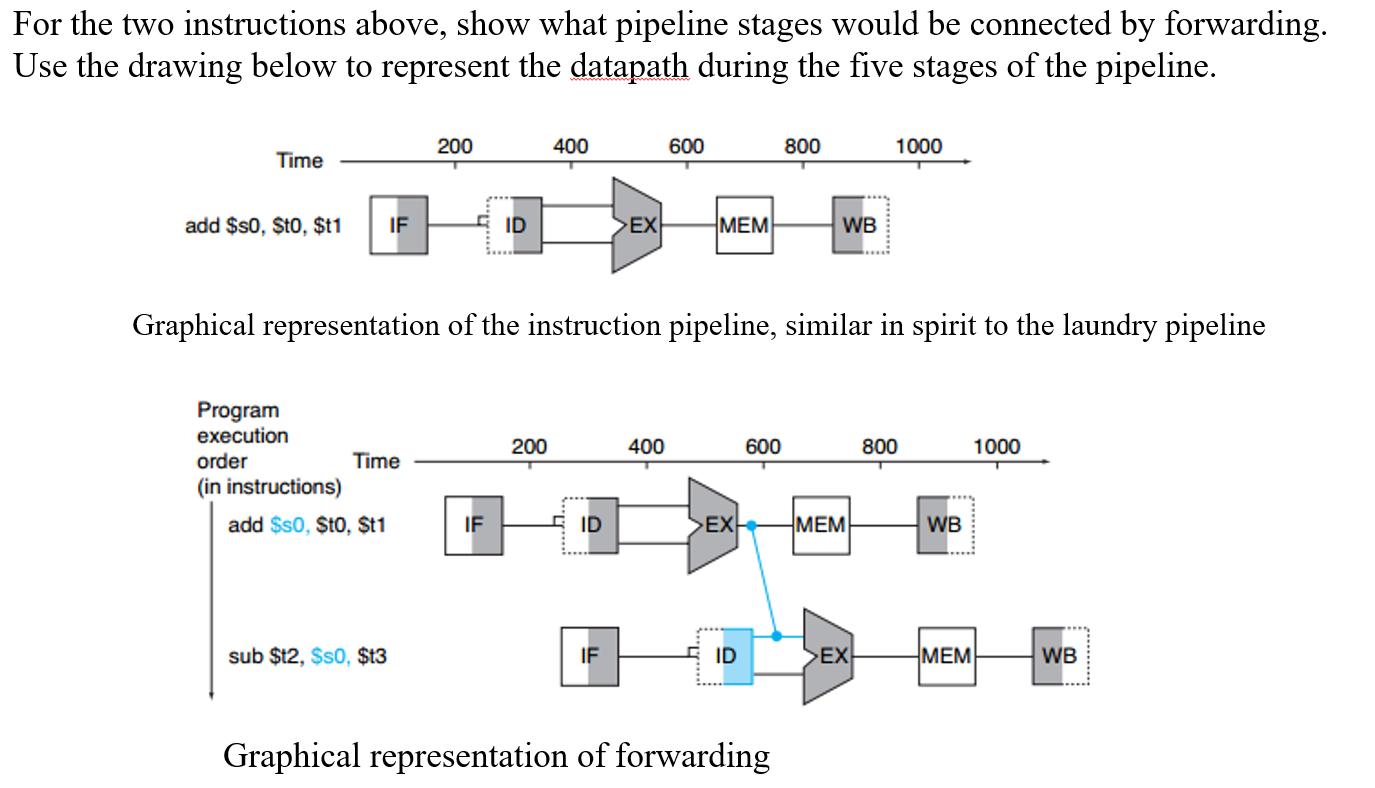
Multi Cycle Approach  
Break instruction into steps, execute each step in one cycle  
Balance amount of work to be done; Restrict each cycle to use only one major functional unit; at the end of a cycle:  
 Store values for use in later cycles  
 Introduce additional internal registers  
5 steps:  
IF -> ID -> EX -> MEM -> WB  
Instruction Fetch – Fetch instruction at address ($PC), store instruction in register, increment $PC  
Instruction Decode – Decode instruction type and read register, store register contents in registers, compute new PC address  
Execute – Compute memory address (LW/SW) or Perform R-type operation (R-type instruction) or Update PC (branch, jump), Store memory address or register operation result  
Memory Access – MemRead, RegWrite, MemWrite – R-type instruction completion   
Write Back – Write memory content read into register file   
R-type: 4 cycles IF – ID – EX – WB   
LW: 5 cycles IF – ID – EX – MEM - WB  
SW: 4 cycles IF – ID – EX – MEM   
Branch/Jump: 3 cycles IF – ID – ALU

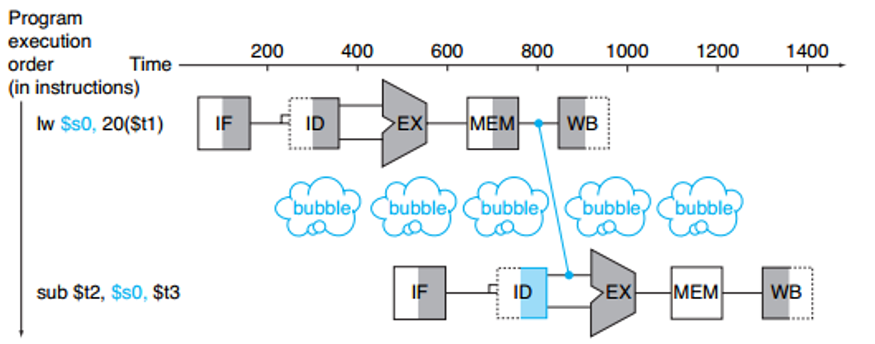
Pipelining:

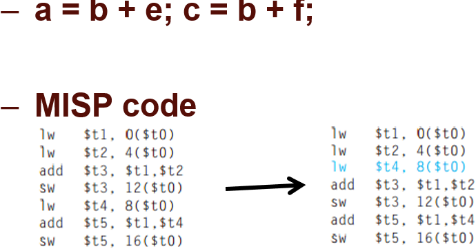


Single-Cycle: Worst case clock cycle of 800ps (whole cycle)  
Multi-Cycle: Worst case clock cycle of 200ps (stage)

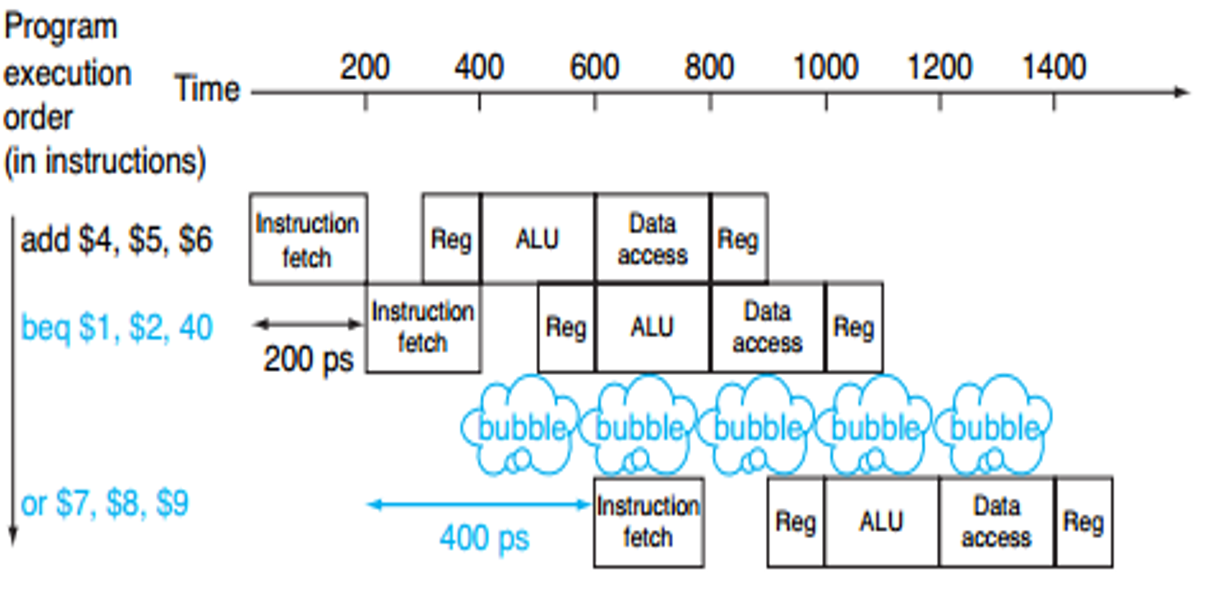
Structural Hazards: hardware cannot support the combination of instructions in the same clock cycle  
- If an instruction tries to access data from memory while another tries fetching and instruction from the same memory

Data Hazard(Pipeline Hazard): when a planned instruction cannot execute in the clock cycle because data that is needed to execute is not available yet  
- If a subtract instruction uses the sum from an add instruction just before, it would need to wait 3 clock cycles to execute as the sum from the add is written in the 5th clock cycle.  
Hazard resolved by forwarding/bypassing:

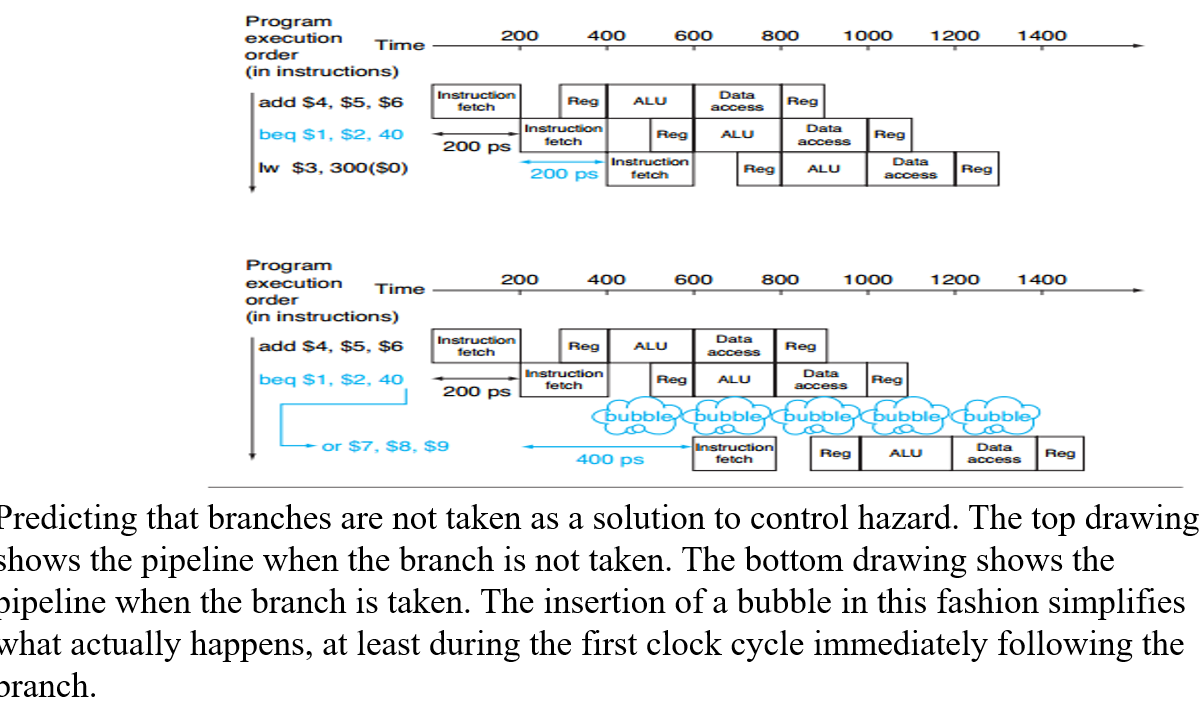
Load-use data hazard: Specific form of data hazard in which data being loaded by load instruction has not yet become available for another instruction.   
Pipeline Stall (or bubble): Waiting one clock cycle with no stages happening:

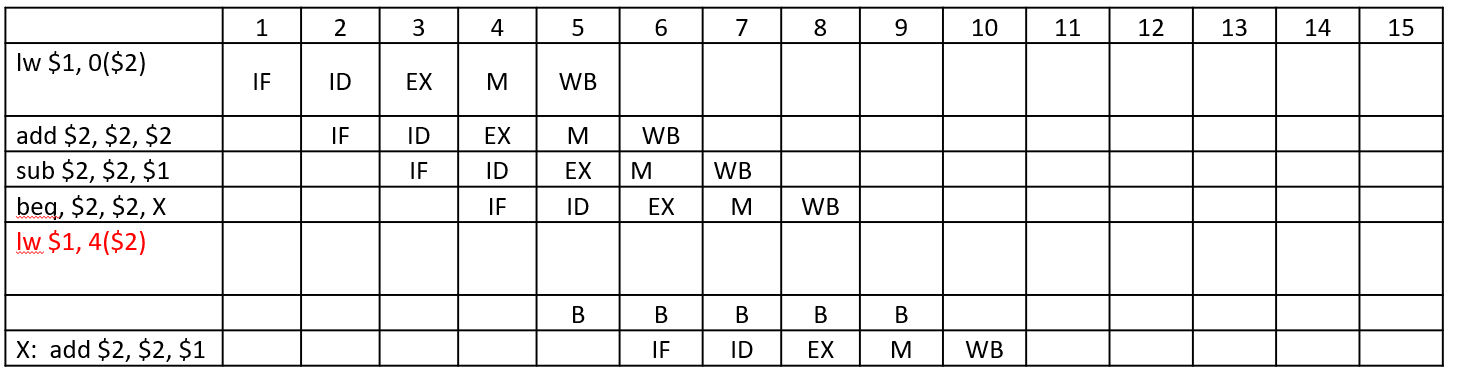
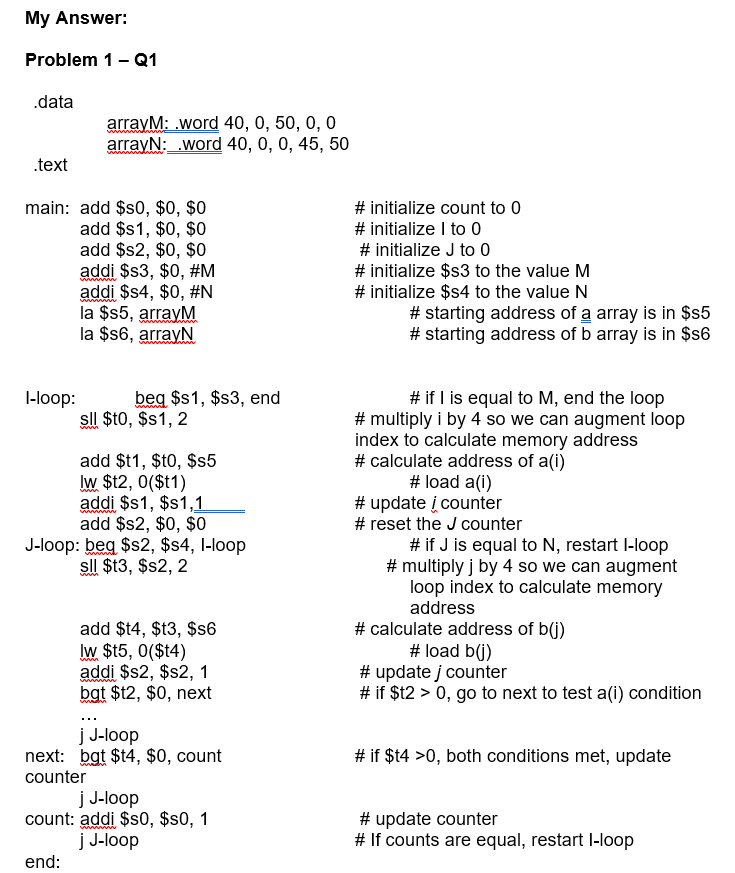


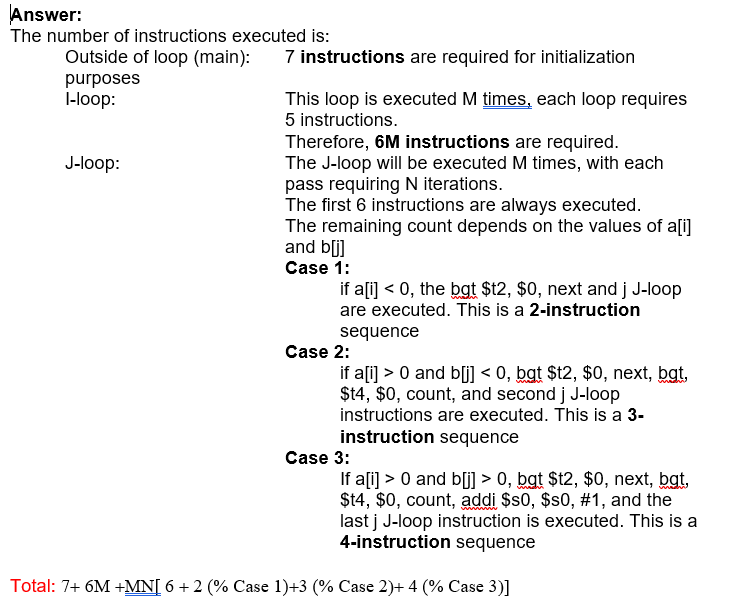
Another method of resolving data hazards: reordering code to avoid load-use pipeline stalls:

Control Hazard (branch hazard): Proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; flow of instruction addresses in not what expected  


Branch prediction (resolving a branch hazard): assume a given outcome for the branch and proceed from that assumption rather than waiting to ascertain actual outcome:

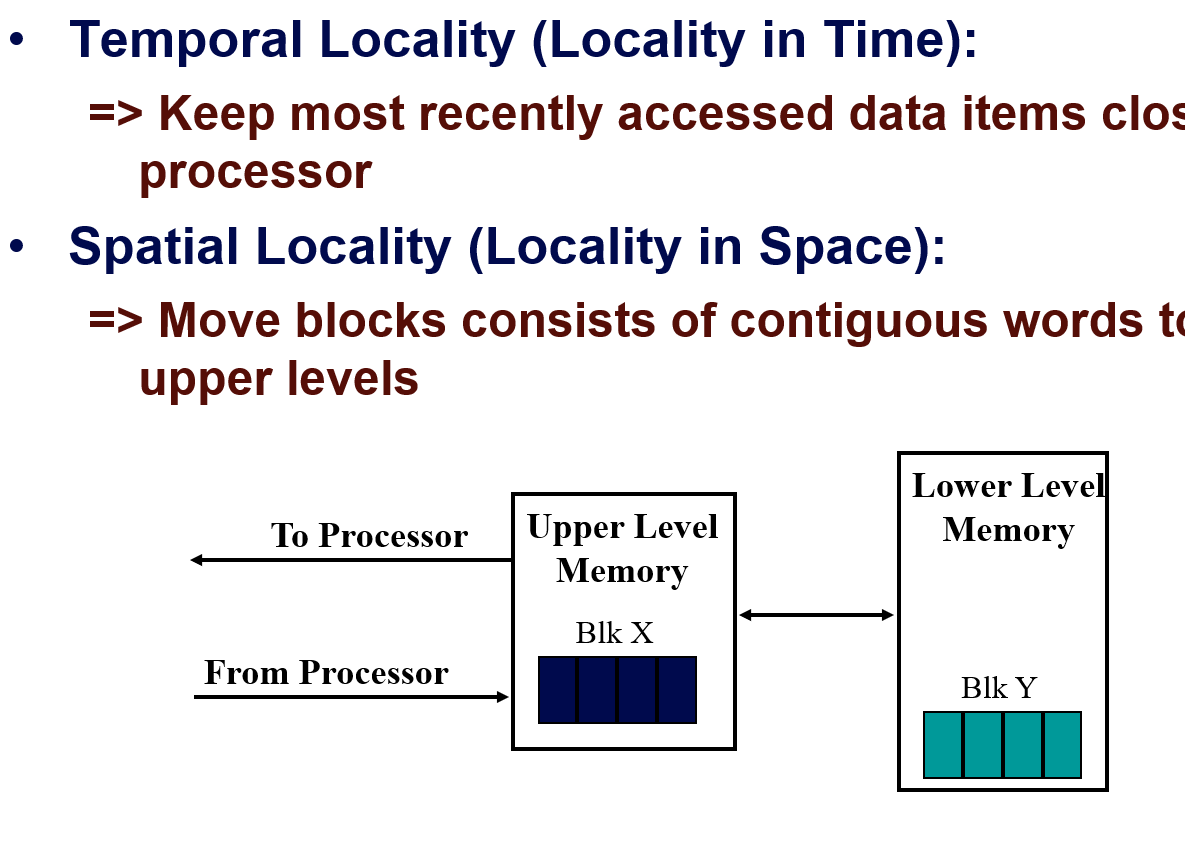




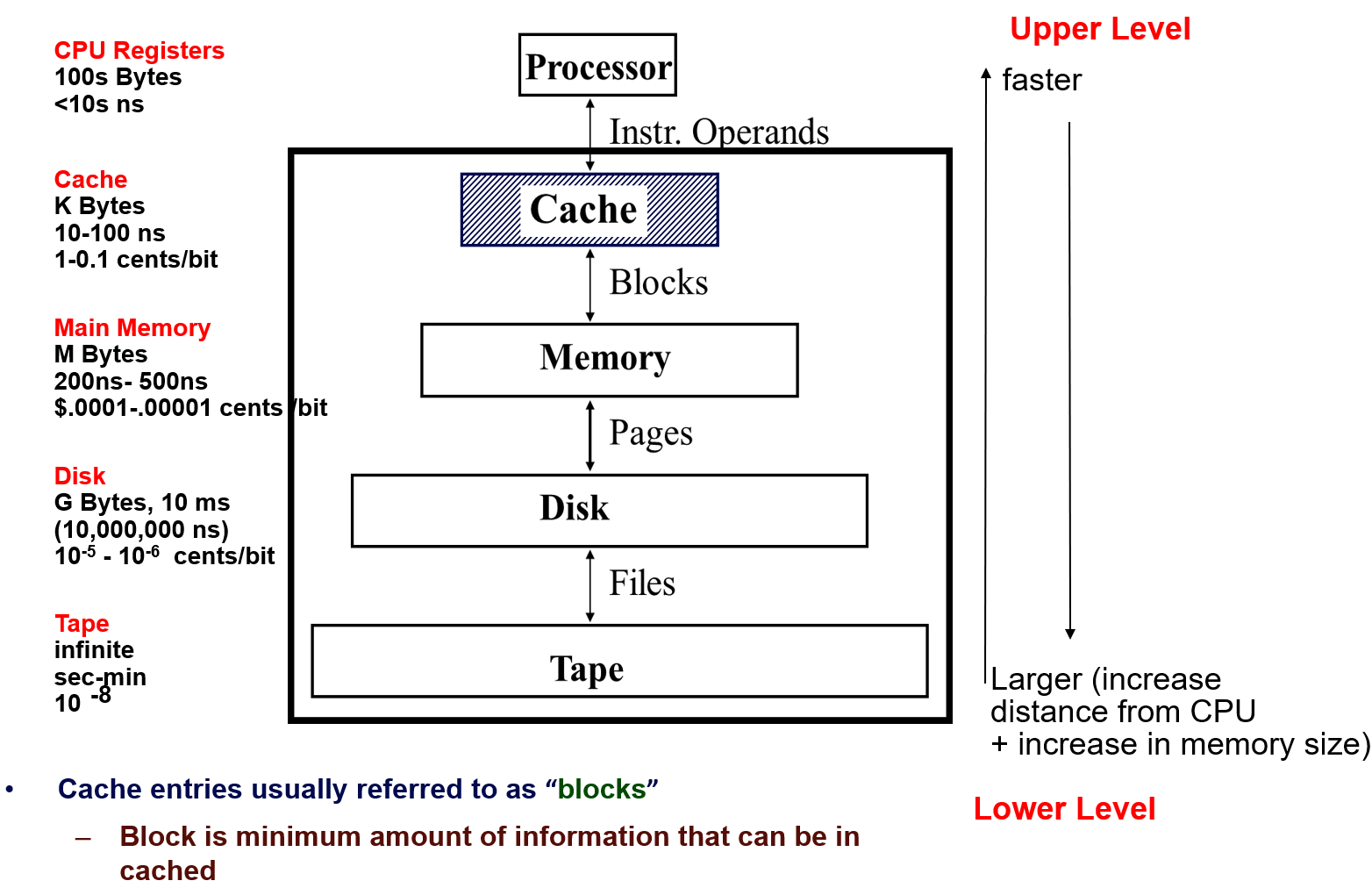


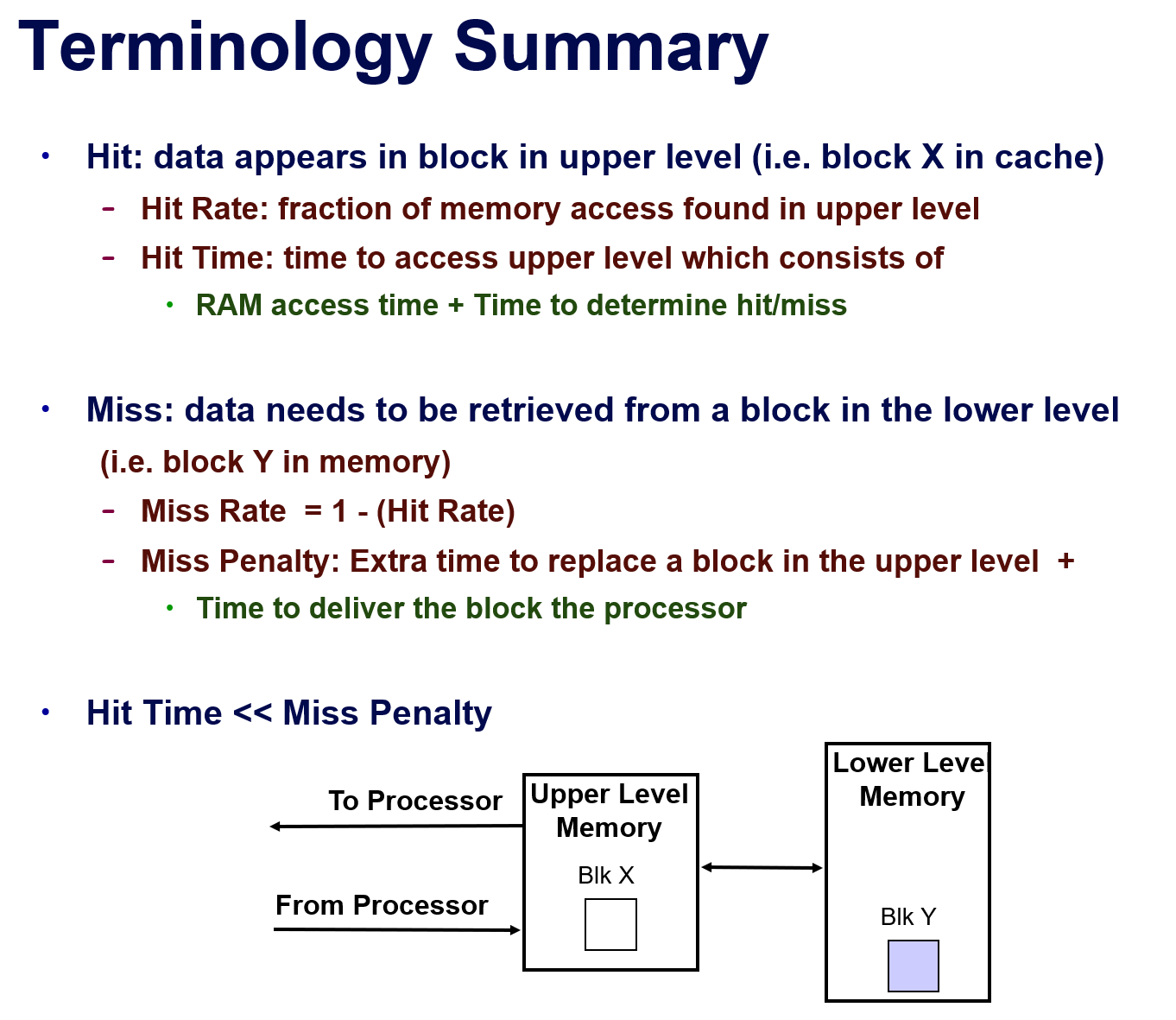
Memory Hierarchy:  
SRAM: Static Random Access Memory: Caches  
DRAM: Dynamic Random Access Memory: Main Memory  
ROM, Flash  
Disks, Tape  
  
Fast= small & expensive  
Large = slow & inexpensive

DRAM access on order of 100ns  
If clock rate for 5 stages was 1GHz, memory and fetch stages stall for 100 cycles if using only DRAM  
  
Principle of locality: most programs don’t access all code or data uniformly  
in a loops, small subset of instructions accessed over and over again, block of memory addressed accessed sequentially  
Temporal Locality: if an item is referenced, it will tend to be referenced again soon  
Spatial locality: if an item is referenced, items whose addresses are close by will be referenced soon  
Memory hierarchies: a structure that uses multiple levels of memories; as the distance from the processor increases, the size of memories and access time both increases



Registers <-> Memory: by compiler  
Cache <-> Memory: by hardware  
Memory <-> disks: by hardware and operating systems (virtual memory) by programmer (files)





Average Memory Access Time AMAT = (Hit Time) + (1-H) \* (Miss Penalty)  
AMAT is a common metric used to analyse memory system performance  
Hit Time: basic time of every access  
Hit Rate(H): fraction of access that hit  
Miss Penalty: extra time to fetch a block from lower level, including time to replace in CPU  
  
Cache entries referred to as “blocks”  
Block: minimum amount of information that can be in a cache  
Direct mapped cache: Block can only go to 1 place in cache, Usually:(Block Address) MOD (# of blocks in cache)  
Memory Block 12 can only go into Cache Block 4 (12 MODULUS 8)

Caches have address tag on each block frame that provide block address  
Each entry usually has a valid bit (tells us if cache data is useful)  
Block offset field selects data from block (i.e. address of desired data within block)  
index field selects a specific set  
Tag field is compared against it for a hit

