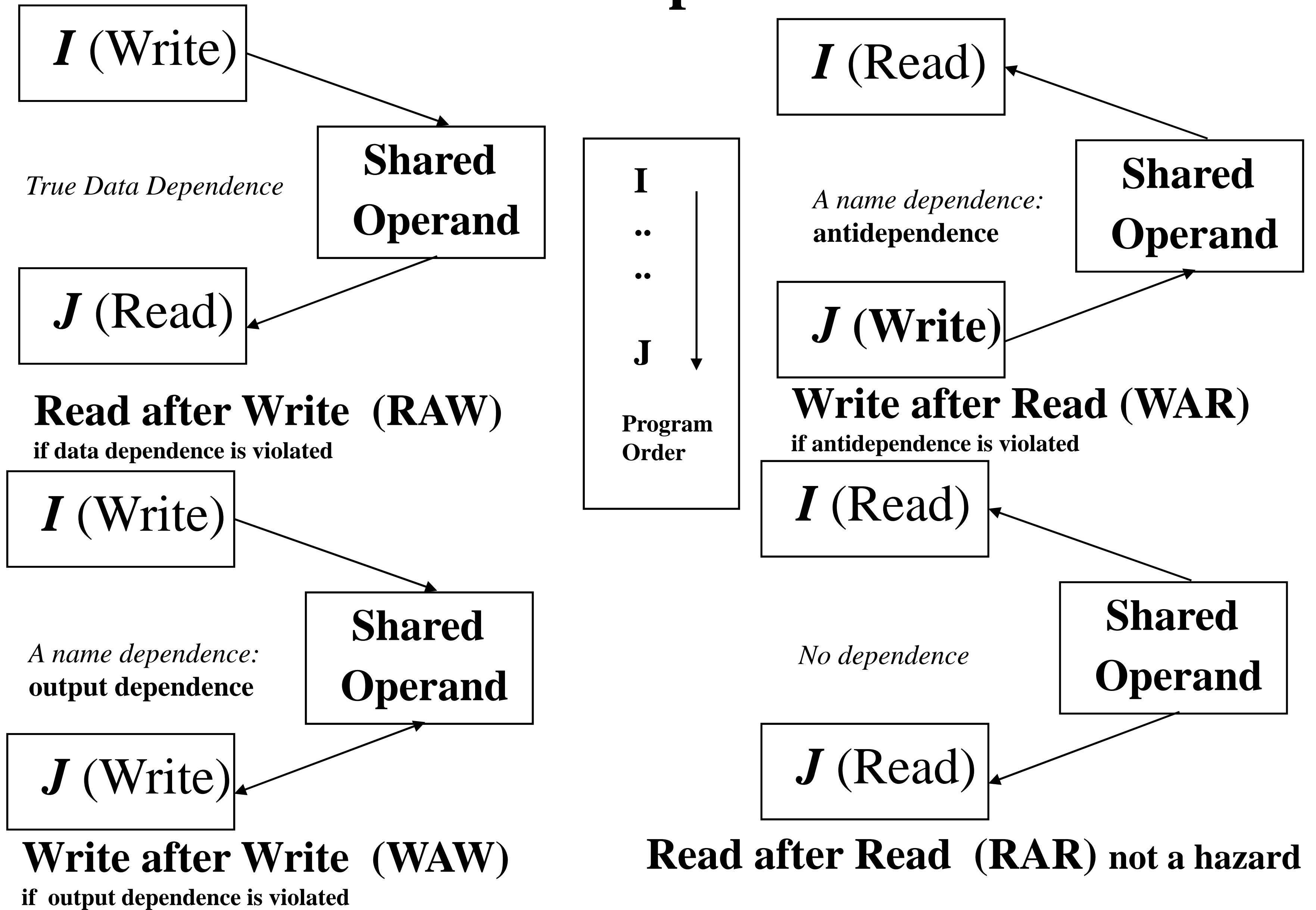
Exam Review

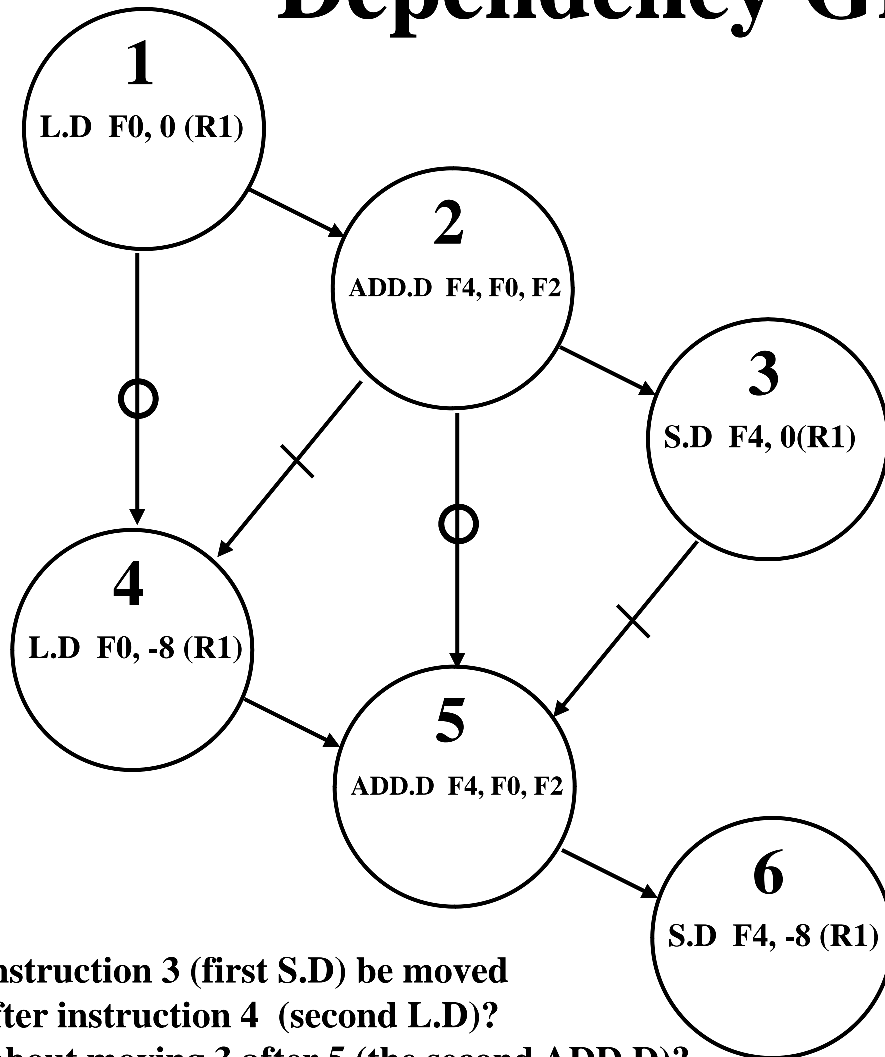
- **Instruction Dependencies**
- **In-order Floating Point/Multicycle Pipelining**
- **Instruction-Level Parallelism (ILP)**
  - **Loop-unrolling**
- **Dynamic Pipeline Scheduling**
  - **The Tomasulo Algorithm**
- **Multiple Instruction Issue (CPI < 1): Superscalar vs. VLIW**
- **Dynamic Hardware-Based Speculation**
- **Loop-Level Parallelism**
  - **Making loop iterations parallel**
  - **Software Pipelining (Symbolic Loop-Unrolling)**
- **Cache & Memory Performance**
- **I/O & System Performance**

Data Hazard/Dependence Classification



Instruction Dependence Example

Dependency Graph



Example Code

1	L.D	F0, 0 (R1)
2	ADD.D	F4, F0, F2
3	S.D	F4, 0(R1)
4	L.D	F0, -8(R1)
5	ADD.D	F4, F0, F2
6	S.D	F4, -8(R1)

Date Dependence:
(1, 2) (2, 3) (4, 5) (5, 6)

Output Dependence:
(1, 4) (2, 5)

Anti-dependence:
(2, 4) (3, 5)

Can instruction 4 (second L.D) be moved just after instruction 1 (first L.D)? If not what dependencies are violated?

Can instruction 3 (first S.D) be moved just after instruction 4 (second L.D)? How about moving 3 after 5 (the second ADD.D)? If not what dependencies are violated?

Control Dependencies

- **Determines the ordering of an instruction with respect to a branch instruction.**
- **Every instruction in a program except those in the very first basic block of the program is control dependent on some set of branches.**
- **An instruction which is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.**
- **An instruction which is not control dependent on the branch cannot be moved so that its execution is controlled by the branch (in the then portion)**
- **It's possible in some cases to violate these constraints and still have correct execution.**
- **Example of control dependence in the then part of an if statement:**

```
if  p1 {
        S1;
};
If p2  {
        S2;
}
```

S1 is control dependent on p1
S2 is control dependent on p2 but not on p1

What happens if S1 is moved here? (arrow pointing to S2)

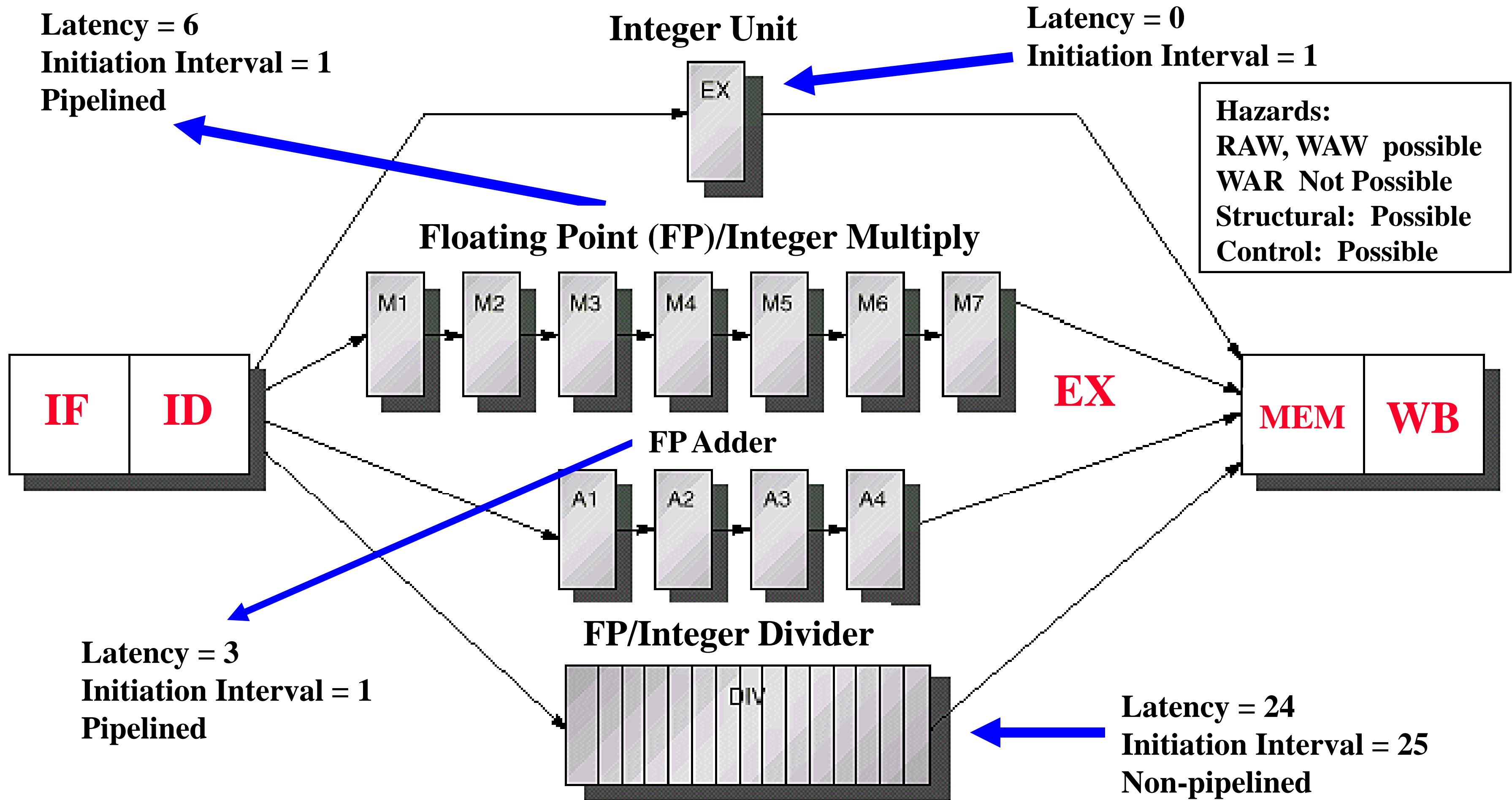
(In Chapter 3.1)

Floating Point/Multicycle Pipelining in MIPS

- **Completion of MIPS EX stage floating point arithmetic operations in one or two cycles is impractical since it requires:**
  - **A much longer CPU clock cycle, and/or**
  - **An enormous amount of logic.**
- **Instead, the floating-point pipeline will allow for a longer latency.**
- **Floating-point operations have the same pipeline stages as the integer instructions with the following differences:**
  - **The EX cycle may be repeated as many times as needed.**
  - **There may be multiple floating-point functional units.**
  - **A stall will occur if the instruction to be issued either causes a structural hazard for the functional unit or cause a data hazard.**
- ***The latency*** **of functional units is defined as the number of intervening cycles between an instruction producing the result and the instruction that uses the result (usually equals stall cycles with forwarding used).**
- ***The initiation*** **or repeat interval is the number of cycles that must elapse between issuing an instruction of a given type.**

(In Appendix A)

Extending The MIPS In-order Integer Pipeline: Multiple Outstanding Floating Point Operations

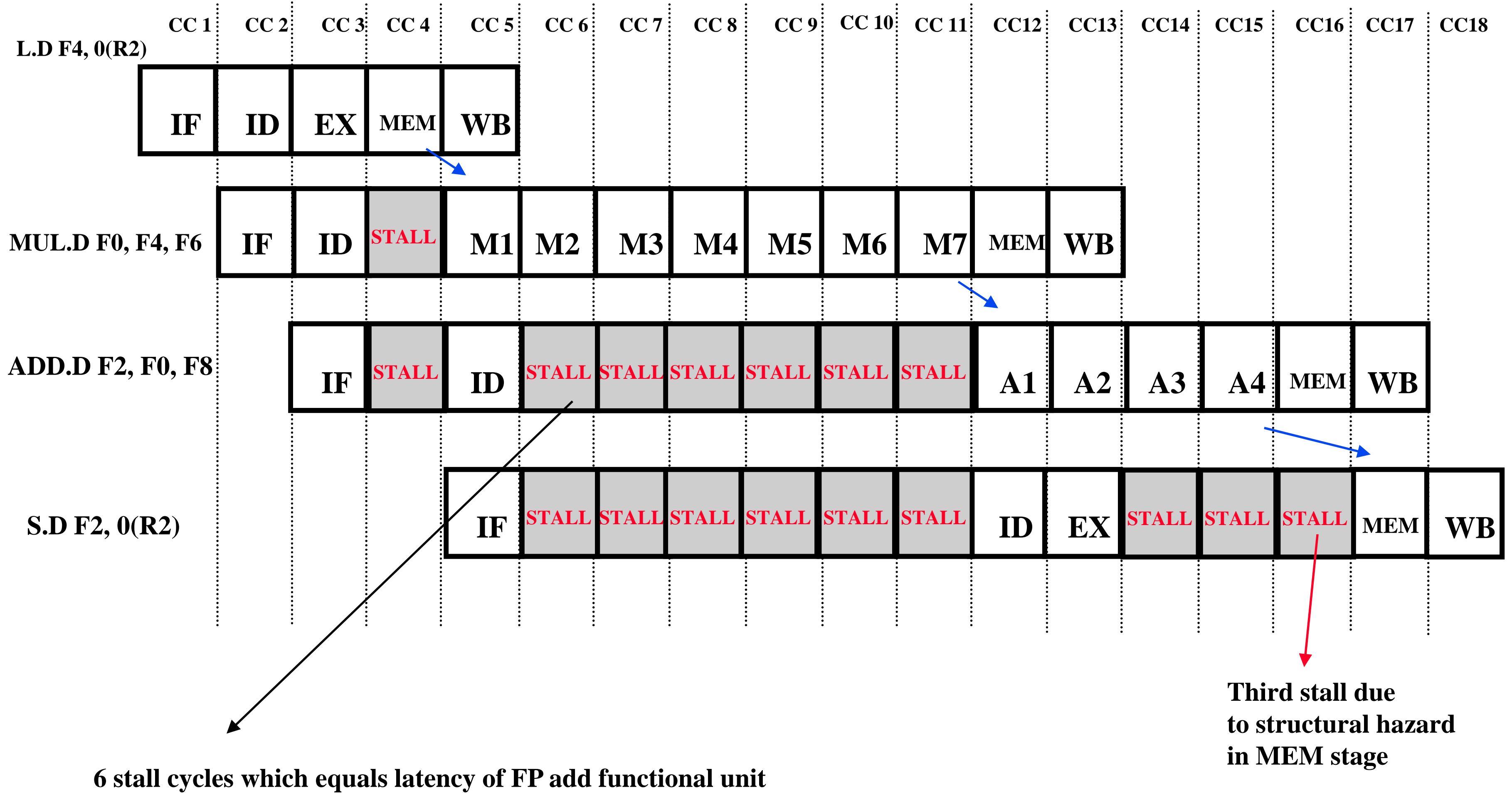


A pipeline that supports multiple outstanding FP operations.

(In Appendix A)

FP Code RAW Hazard Stalls Example

(with full data forwarding in place)



(In Appendix A)

(quiz 2)

Increasing Instruction-Level Parallelism

- **A common way to increase parallelism among instructions is to exploit parallelism among iterations of a loop**
  - **(i.e Loop Level Parallelism, LLP).**
- **This is accomplished by unrolling the loop either statically by the compiler, or dynamically by hardware, which increases the size of the basic block present. This resulting larger basic block provides more instructions that can scheduled or re-ordered to eliminate more stall cycles.**
- **In this loop every iteration can overlap with any other iteration. Overlap within each iteration is minimal.**

```
for (i=1; i<=1000; i=i+1;)
        x[i] = x[i] + y[i];
```

- **In vector machines, utilizing vector instructions is an important alternative to exploit loop-level parallelism,**
- **Vector instructions operate on a number of data items. The above loop would require just four such instructions.**

(In Chapter 4.1)

MIPS Loop Unrolling Example

- **For the loop:**

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

The straightforward MIPS assembly code is given by:

```
Loop:  L.D       F0, 0 (R1)       ;F0=array element
       ADD.D     F4, F0, F2       ;add scalar in F2
       S.D       F4, 0(R1)        ;store result
       DADDUI    R1, R1, # -8     ;decrement pointer 8 bytes
       BNE       R1, R2,Loop      ;branch R1!=R2
```

(Basic block size = 5 instructions)

R1 is initially the address of the element with highest address.
8(R2) is the address of the last element to operate on.

(In Chapter 4.1)

(quiz 3)

MIPS FP Latency For Loop Unrolling Example

- **All FP units assumed to be pipelined.**
- **The following FP operations latencies are used:**

(or Number of Stall Cycles)

Instruction Producing Result	Instruction Using Result	Latency In Clock Cycles
FP ALU Op →	Another FP ALU Op	3
FP ALU Op	Store Double	2
Load Double	FP ALU Op	1
Load Double	Store Double	0

Branch resolved in decode stage, Branch penalty = 1 cycle, Full forwarding is used

(In Chapter 4.1)

Loop Unrolling Example (continued)

- **This loop code is executed on the MIPS pipeline as follows:**

  (Branch resolved in decode stage, Branch penalty = 1 cycle, Full forwarding is used)

No scheduling

			Clock cycle
Loop:	L.D	F0, 0(R1)	1
	stall		2
	ADD.D	F4, F0, F2	3
	stall		4
	stall		5
	S.D	F4, 0 (R1)	6
	DADDUI	R1, R1, # -8	7
	stall		8
	BNE	R1,R2, Loop	9
	stall		10

10 cycles per iteration

Scheduled with single delayed branch slot

Loop:	L.D	F0, 0(R1)
	DADDUI	R1, R1, # -8
	ADD.D	F4, F0, F2
	stall	
	BNE	R1,R2, Loop
	S.D	F4,8(R1)

6 cycles per iteration

10/6 = 1.7 times faster

(In Chapter 4.1)

Loop Unrolling Example (continued)

- The resulting loop code when four copies of the loop body are unrolled without reuse of registers.
- The size of the basic block increased from 5 instructions in the original loop to 14 instructions.

Cycle

No scheduling

```
Loop: 1  L.D      F0, 0(R1)
      2  Stall
      3  ADD.D    F4, F0, F2
      4  Stall
      5  Stall
      6  SD       F4,0 (R1)        ; drop DADDUI & BNE
      7  LD       F6, -8(R1)
      8  Stall
      9  ADDD     F8, F6, F2
      10 Stall
      11 Stall
      12 SD       F8, -8 (R1),     ; drop DADDUI & BNE
      13 LD       F10, -16(R1)
      14 Stall
      15 ADDD     F12, F10, F2
      16 Stall
      17 Stall
      18 SD       F12, -16 (R1)    ; drop DADDUI & BNE
      19 LD       F14, -24 (R1)
      20 Stall
      21 ADDD     F16, F14, F2
      22 Stall
      23 Stall
      24 SD       F16, -24(R1)
      25 DADDUI   R1, R1, # -32
      26 Stall
      27 BNE      R1, R2, Loop
      28 Stall
```

Three branches and three decrements of R1 are eliminated.

Load and store addresses are changed to allow DADDUI instructions to be merged.

The unrolled loop runs in 28 cycles assuming each L.D has 1 stall cycle, each ADD.D has 2 stall cycles, the DADDUI 1 stall, the branch 1 stall cycle, or 28/4 = 7 cycles to produce each of the four elements.

(In Chapter 4.1)

Loop Unrolling Example (continued)

When scheduled for pipeline

```
Loop:   L.D       F0, 0(R1)
        L.D       F6,-8 (R1)
        L.D       F10, -16(R1)
        L.D       F14, -24(R1)
        ADD.D     F4, F0, F2
        ADD.D     F8, F6, F2
        ADD.D     F12, F10, F2
        ADD.D     F16, F14, F2
        S.D       F4, 0(R1)
        S.D       F8, -8(R1)
        DADDUI    R1, R1,# -32
        S.D       F12, 16(R1),F12
        BNE       R1,R2, Loop
        S.D       F16, 8(R1), F16    ;8-32 = -24
```

The execution time of the loop has dropped to 14 cycles, or 14/4 = 3.5 clock cycles per element

compared to 6.8 before scheduling and 6 when scheduled but unrolled.

Unrolling the loop exposed more computations that can be scheduled to minimize stalls by increasing the size of the basic block from 5 instructions in the original loop to 14 instructions in the unrolled loop.

(In Chapter 4.1)

Dynamic Pipeline Scheduling

- **Dynamic instruction scheduling is accomplished by:**
  - **Dividing the Instruction Decode ID stage into two stages:**
    - **Issue: Decode instructions, check for structural hazards.**
    - **Read operands: Wait until data hazard conditions, if any, are resolved, then read operands when available.**

    (All instructions pass through the issue stage in order but can be stalled or pass each other in the read operands stage).
  - **In the instruction fetch stage IF, fetch an additional instruction every cycle into a latch or several instructions into an instruction queue.**
  - **Increase the number of functional units to meet the demands of the additional instructions in their EX stage.**
- **Two dynamic scheduling approaches exist:**
  - **Dynamic scheduling with a Scoreboard used first in CDC6600 (1963)**
  - **The Tomasulo approach pioneered by the IBM 360/91 (1966)**

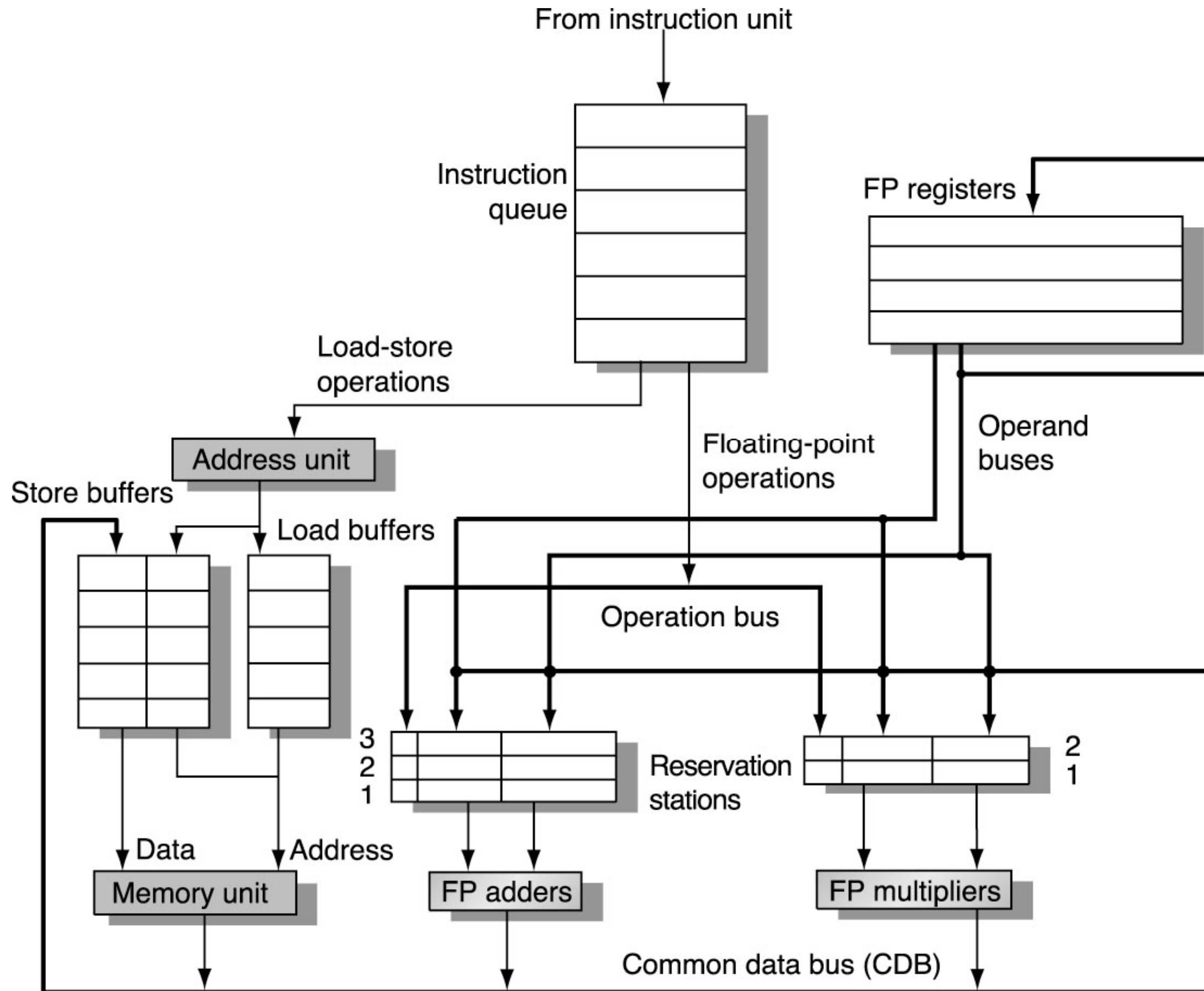
(In Appendix A.8, Chapter 3.2)

Tomasulo Algorithm Vs. Scoreboard

- **Control & buffers *distributed* with Function Units (FU) Vs. centralized in Scoreboard:**
  - **FU buffers are called "*reservation stations*" which have pending instructions and operands and other instruction status info.**
  - **Reservations stations are sometimes referred to as "physical registers" or "renaming registers" as opposed to architecture registers specified by the ISA.**
- **ISA Registers in instructions are replaced by either values (if available) or pointers to reservation stations (RS) that will supply the value later:**
  - **This process is called *register renaming*.**
  - **Avoids WAR, WAW hazards.**
  - **Allows for *hardware-based* loop unrolling.**
  - **More reservation stations than ISA registers are possible , leading to optimizations that compilers can't achieve and prevents the number of ISA registers from becoming a bottleneck.**
- **Instruction results go (forwarded) to FUs from RSs, *not through registers*, over *Common Data Bus (CDB)* that broadcasts results to all FUs.**
- **Loads and Stores are treated as FUs with RSs as well.**
- **Integer instructions can go past branches, allowing FP ops beyond basic block in FP queue.**

(In Chapter 3.2)

Dynamic Scheduling: The Tomasulo Approach



The basic structure of a MIPS floating-point unit using Tomasulo's algorithm

(In Chapter 3.2)

Reservation Station Fields

- **Op** Operation to perform in the unit (e.g., + or –)
- **Vj, Vk** **Value** of Source operands S1 and S2
  - Store buffers have a single **V** field indicating result to be stored.
- **Qj, Qk** Reservation stations producing source registers. (value to be written).
  - No ready flags as in Scoreboard; Qj,Qk=0 => ready.
  - Store buffers only have Qi for RS producing result.
- **A:** Address information for loads or stores. Initially immediate field of instruction then effective address when calculated.
- **Busy:** Indicates reservation station and FU are busy.
- **Register result status:** Qi Indicates which functional unit will write each register, if one exists.
  - Blank (or 0) when no pending instructions exist that will write to that register.

(In Chapter 3.2)

Three Stages of Tomasulo Algorithm

1. **Issue:** Get instruction from pending Instruction Queue.
   - Instruction issued to a free reservation station (no structural hazard).
   - Selected RS is marked busy.
   - Control sends available instruction operands values (from ISA registers) to assigned RS.
   - Operands not available yet are renamed to RSs that will produce the operand (register renaming).
2. **Execution (EX):** Operate on operands.
   - When both operands are ready then start executing on assigned FU.
   - If all operands are not ready, watch Common Data Bus (CDB) for needed result (forwarding done via CDB).
3. **Write result (WB):** Finish execution.
   - Write result on Common Data Bus to all awaiting units
   - Mark reservation station as available.

- **Normal data bus: data + destination ("go to" bus).**
- **Common Data Bus (CDB): data + source ("come from" bus):**
  - 64 bits for data + 4 bits for Functional Unit source address.
  - Write data to waiting RS if source matches expected RS (that produces result).
  - Does the result forwarding via broadcast to waiting RSs.

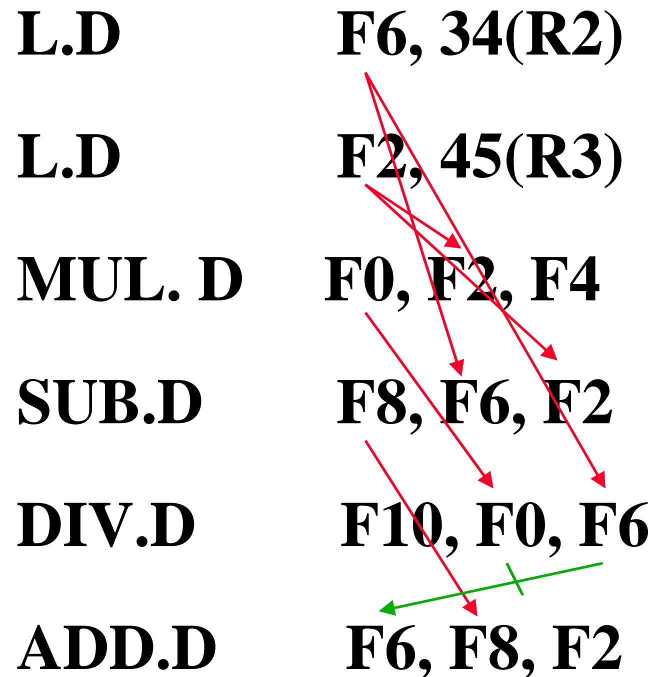
(In Chapter 3.2)

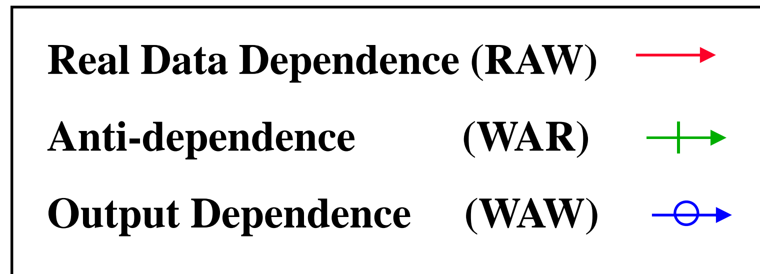
Tomasulo Approach Example

Using the same code used in the scoreboard example to be run on the Tomasulo configuration given earlier:

	# of RSs	EX Latency
Integer	1	0
Floating Point Multiply/divide	2	10/40
Floating Point add	3	2

Pipelined Functional Units





(In Chapter 3.3)

Tomasulo Example: Cycle 57

Instruction status					Execution	Write						
Instruction		j	k	Issue	complete	Result			Busy	Address		
L.D	F6	34+	R2	1	3	4		Load1	No			
L.D	F2	45+	R3	2	4	5		Load2	No			
MUL.D	F0	F2	F4	3	15	16		Load3	No			
SUB.D	F8	F6	F2	4	7	8						
DIV.D	F10	F0	F6	5	56	57						
ADD.D	F6	F8	F2	6	10	11						
Reservation Stations					S1	S2	RS for j	RS for k				
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk				
	0	Add1	No									
	0	Add2	No									
		Add3	No									
	0	Mult1	No									
	0	Mult2	No									
Register result status												
Clock				F0	F2	F4	F6	F8	F10	F12	...	F30
57			FU	M*F4	M(45+R3)		(M–M)+M()	M()–M()	M*F4/M			

Instruction Block done

- **We have:**
  - **In-oder issue,**
  - **Out-of-order execution, completion**

(quiz 4)

Tomasulo Loop Example

```
Loop:  L.D       F0, 0(R1)
       MUL.D     F4,F0,F2
       S.D       F4, 0(R1)
       DADDUI    R1,R1,# -8
       BNE       R1,R2,Loop  ; branch if R1 = R2
```

- **Assume Multiply takes 4 clocks.**
- **Assume first load takes 8 clocks (possibly due to a cache miss), second load takes 4 clocks (cache hit).**
- **Assume R1 = 80 initially.**
- **Assume branch is predicted taken and no branch misprediction.**
- **No branch delay slot is used in this example.**
- **Stores take 4 cycles (ex, mem) and do not write on CDB**
- **We'll go over the execution to complete first two loop iterations.**

(Expanded from loop example in Chapter 3.3)

First two Loop iterations done

Loop Example Cycle 19

Instruction status						Execution	Write					
Instruction		j	k	iteration	Issue	complete	Result		Busy	Address		
L.D	F0	0	R1	1	1	9	10	Load1	No			
MUL.D	F4	F0	F2	1	2	14	15	Load2	No			
S.D	F4	0	R1	1	3	18		Load3	No		Qi	
L.D	F0	0	R1	2	6	10	11	Store1	No			
MUL.D	F4	F0	F2	2	7	15	16	Store2 0	Yes	72	M(72)*R(72)	
S.D	F4	0	R1	2	8	19		Store3	Yes	64	Mult1	
Reservation Stations					S1	S2	RS for j	RS for k				
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:			
	0	Add1	No						L.D		F0, 0(R1)	
	0	Add2	No						MUL.D		F4,F0,F2	
	0	Add3	No						S.D		F4, 0(R1)	
	1	Mult1	Yes	MULTD	M(64)	R(F2)			DADDUI		R1, R1, #-8	
	0	Mult2	No						BNE		R1,R2,loop	
Register result status												
Clock		R1		F0	F2	F4	F6	F8	F10	F12	...	F30
19		56	Qi			Mult1						

Second S.D done (No write on CDB for stores) Second loop iteration done
Issue third iteration BNE

Multiple Instruction Issue: CPI < 1

- To improve a pipeline's CPI to be better [less] than one, and to utilize Instruction Level Parallelism (ILP) better, a number of independent instructions have to be *issued* in the same pipeline cycle.
- Multiple instruction issue processors are of two types:
  - Superscalar: A number of instructions (2-8) is issued in the same cycle, scheduled statically by the compiler or dynamically (Tomasulo).
    - PowerPC, Sun UltraSparc, Alpha, HP 8000 ...
  - VLIW (Very Long Instruction Word):
    A fixed number of instructions (3-6) are formatted as one long instruction word or packet (statically scheduled by the compiler).
      - Example: Explicitly Parallel Instruction Computer (EPIC)
        - Originally a joint HP/Intel effort.
        - ISA: Intel Architecture-64 (IA-64) 64-bit address:
        - First CPU: Itanium, Q1 2001.
- Limitations of the approaches:
  - Available ILP in the program (both).
  - Specific hardware implementation difficulties (superscalar).
  - VLIW optimal compiler design issues.

(Ch 3.6, 3.7, 4.3, 4.5)

Unrolled Loop Example for Scalar (single-issue) Pipeline

```
1  Loop: L.D     F0,0(R1)
2        L.D     F6,-8(R1)
3        L.D     F10,-16(R1)
4        L.D     F14,-24(R1)
5        ADD.D   F4,F0,F2
6        ADD.D   F8,F6,F2
7        ADD.D   F12,F10,F2
8        ADD.D   F16,F14,F2
9        S.D     F4,0(R1)
10       S.D     F8,-8(R1)
11       DADDUI  R1,R1,#-32
12       S.D     F12, 16(R1)
13       BNE     R1,R2,LOOP
14       S.D     F16,8(R1)      ; 8-32 = -24
```

L.D to ADD.D: 1 Cycle
ADD.D to S.D: 2 Cycles

14 clock cycles, or 3.5 per iteration

Loop Unrolling in Superscalar Pipeline: (1 Integer, 1 FP/Cycle)

	Integer instruction	FP instruction	Clock cycle
Loop:	L.D F0,0(R1)		1
	L.D F6,-8(R1)		2
	L.D F10,-16(R1)	ADD.D F4,F0,F2	3
	L.D F14,-24(R1)	ADD.D F8,F6,F2	4
	L.D F18,-32(R1)	ADD.D F12,F10,F2	5
	S.D F4,0(R1)	ADD.D F16,F14,F2	6
	S.D F8,-8(R1)	ADD.D F20,F18,F2	7
	S.D F12,-16(R1)		8
	DADDUI R1,R1,#-40		9
	S.D F16,-24(R1)		10
	BNE R1,R2,LOOP		11
	SD -32(R1),F20		12

- Unrolled 5 times to avoid delays and expose more ILP (unrolled one more time)
- 12 cycles, or 12/5 = 2.4 cycles per iteration (3.5/2.4= 1.5X faster than scalar)
- CPI = 12/ 17 = .7 worse than ideal CPI = .5 because 7 issue slots are wasted

Loop Unrolling in VLIW Pipeline (2 Memory, 2 FP, 1 Integer / Cycle)

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
L.D F0,0(R1)	L.D F6,-8(R1)				1
L.D F10,-16(R1)	L.D F14,-24(R1)				2
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2		3
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2		4
		ADD.D F20,F18,F2	ADD.D F24,F22,F2		5
S.D F4,0(R1)	S.D F8, -8(R1)	ADD.D F28,F26,F2			6
S.D F12, -16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56	7
S.D F20, 24(R1)	S.D F24,16(R1)				8
S.D F28, 8(R1)				BNE R1,R2,LOOP	9

Unrolled 7 times to avoid delays and expose more ILP
7 results in 9 cycles, or 1.3 cycles per iteration
(2.4/1.3 =1.8X faster than 2-issue superscalar, 3.5/1.3 = 2.7X faster than scalar)
Average: about 23/9 = 2.55 IPC (instructions per clock cycle) Ideal IPC =5,
CPI = .39 Ideal CPI = .2 thus about 50% efficiency, 22 issue slots are wasted

Note: Needs more registers in VLIW (15 vs. 6 in Superscalar)

(In chapter 4.3 pages 317-318)

Multiple Instruction Issue with Dynamic Scheduling Example

Assumptions:
Restricted 2-way superscalar:
1 integer, 1 FP Issue Per Cycle

One integer unit
(for ALU, effective address)
One integer unit for branch condition
2 CDBs

Execution cycles:
Integer: 1 cycle
Load: 2 cycles (1 ex + 1 mem)
FP add: 3 cycles

Any instruction following
a branch cannot start execution
until branch condition is evaluated

<u>Branches are single issued</u>,
no delayed branch,
perfect branch prediction

Example Consider the execution of the following simple loop, which adds a scalar in F2 to each element of a vector in memory. Use a MIPS pipeline extended with Tomasulo's algorithm and with multiple issue:

```
Loop:   L.D      F0,0(R1)      ;F0=array element
        ADD.D    F4,F0,F2      ;add scalar in F2
        S.D      F4,0(R1)      ;store result
        DADDIU   R1,R1,#-8     ;decrement pointer
                               ;8 bytes (per DW)
        BNE      R1,R2,LOOP    ;branch R1!=R2
```

Assume that both a floating-point and an integer operation can be issued on every clock cycle, even if they are dependent. Assume one integer functional unit is used for both ALU operations and effective address calculations and a separate pipelined FP functional unit for each operation type. Assume that Issue and Write Results take one cycle each and that there is dynamic branch-prediction hardware and a separate functional unit to evaluate branch conditions. As in most dynamically scheduled processors, the presence of the Write Results stage means that the effective instruction latencies will be one cycle longer than in a simple in-order pipeline. Thus, the number of cycles of latency between a source instruction and an instruction consuming the result is one cycle for integer ALU operations, two cycles for loads, and three cycles for FP add. Create a table showing when each instruction issues, begins execution, and writes its result to the CDB for the first three iterations of the loop. Assume two CDBs and assume that branches single issue (no delayed branches) but that branch prediction is perfect. Also show the resource usage for the integer unit, the floating-point unit, the data cache, and the two CDBs.

Example on page 221

Three Loop Iterations on Restricted 2-way Superscalar Tomasulo

Iteration number	Instructions		Issues at	Executes	Memory access at	Write CDB at	Comment
1	L.D	F0,0(R1)	1	2	3	4	First issue
1	ADD.D	F4,F0,F2	1	5		8	Wait for L.D
1	S.D	F4,0(R1)	2	3	9		Wait for ADD.D
1	DADDIU	R1,R1,#-8	2	4		5	Wait for ALU
1	BNE	R1,R2,Loop	3	6			Wait for DADDIU
2	L.D	F0,0(R1)	4	7	8	9	Wait for BNE complete
2	ADD.D	F4,F0,F2	4	10		13	Wait for L.D
2	S.D	F4,0(R1)	5	8	14		Wait for ADD.D
2	DADDIU	R1,R1,#-8	5	9		10	Wait for ALU
2	BNE	R1,R2,Loop	6	11			Wait for DADDIU
3	L.D	F0,0(R1)	7	12	13	14	Wait for BNE complete
3	ADD.D	F4,F0,F2	7	15		18	Wait for L.D
3	S.D	F4,0(R1)	8	13	19		Wait for ADD.D
3	DAADIU	R1,R1,#-8	8	14		15	Wait for ALU
3	BNE	R1,R2,Loop	9	16			Wait for DADDIU

Figure 3.25 The clock cycle of issue, execution, and writing result for a dual-issue version of our Tomasulo pipeline. The Write Result stage does not apply to either stores or branches, since they do not write any registers. We assume a result is written to the CDB at the end of the clock cycle it is available in. This figure also assumes a wider CDB. For L.D and S.D, the execution is effective address calculation. For branches, the execute cycle shows when the branch condition can be evaluated and the prediction checked; we assume that this can happen as early as the cycle after issue, if the operands are available. Any instructions following a branch cannot start execution until after the branch condition has been evaluated. We assume one memory unit, one integer pipeline, and one FP adder. If two instructions could use the same functional unit at the same point, priority is given to the "older" instruction. Note that the load of the next iteration performs its memory access before the store of the current iteration.

Only one CDB is needed in this case.

Multiple Instruction Issue with Dynamic Scheduling Example

Example Consider the execution of the same loop on a two-issue processor, but, in addition, assume that there are separate integer functional units for effective address calculation and for ALU operations. Create a table as in Figure 3.25 for the first three iterations of the same loop and another table to show the resource usage.

Answer Figure 3.27 shows the improvement in performance: The loop executes in 5 clock cycles less (11 versus 16 execution cycles). The cost of this improvement is both a separate address adder and the logic to issue to it; note that, in contrast to the earlier example, a second CDB is needed. As Figure 3.28 shows this example has a higher instruction execution rate but lower efficiency as measured by the utilization of the functional units.

Three factors limit the performance (as shown in Figure 3.27) of the two-issue dynamically scheduled pipeline:

1. There is an imbalance between the functional unit structure of the pipeline and the example loop. This imbalance means that it is impossible to fully use the FP units. To remedy this, we would need fewer dependent integer operations per loop. The next point is a different way of looking at this limitation.
2. The amount of overhead per loop iteration is very high: two out of five instructions (the `DADDIU` and the `BNE`) are overhead. In the next chapter we look at how this overhead can be reduced.
3. The control hazard, which prevents us from starting the next `L.D` before we know whether the branch was correctly predicted, causes a one-cycle penalty on every loop iteration. The next section introduces a technique that addresses this limitation.

Assumptions:
The same loop in previous example
On restricted 2-way superscalar:
1 integer, 1 FP Issue Per Cycle

Two integer units
one for ALU, one for effective address
One integer unit for branch condition
2 CDBs

Execution cycles:
Integer: 1 cycle
Load: 2 cycles (1 ex + 1 mem)
FP add: 3 cycles

Any instruction following
a branch cannot start execution
until branch condition is evaluated

Branches are single issued,
no delayed branch,
perfect branch prediction

Example on page 223

Same three loop Iterations on Restricted 2-way Superscalar Tomasulo but with Two integer units (one for ALU, one for effective address)

Iteration number	Instructions	Issues at	Executes	Memory access at	Write CDB at	Comment
1	L.D F0,0(R1)	1	2	3	4	First issue
1	ADD.D F4,F0,F2	1	5		8	Wait for L.D
1	S.D F4,0(R1)	2	3	9		Wait for ADD.D
1	DADDIU R1,R1,#-8	2	3		4	Executes earlier
1	BNE R1,R2,Loop	3	5			Wait for DADDIU
2	L.D F0,0(R1)	4	6	7	8	Wait for BNE complete
2	ADD.D F4,F0,F2	4	9		12	Wait for L.D
2	S.D F4,0(R1)	5	7	13		Wait for ADD.D
2	DADDIU R1,R1,#-8	5	6		7	Executes earlier
2	BNE R1,R2,Loop	6	8			Wait for DADDIU
3	L.D F0,0(R1)	7	9	10	11	Wait for BNE complete
3	ADD.D F4,F0,F2	7	12		15	Wait for L.D
3	S.D F4,0(R1)	8	10	16		Wait for ADD.D
3	DADDIU R1,R1,#-8	8	9		10	Executes earlier
3	BNE R1,R2,Loop	9	11			Wait for DADDIU

Figure 3.27 The clock cycle of issue, execution, and writing result for a dual-issue version of our Tomasulo pipeline with separate functional units for integer ALU operations and effective address calculation, which also uses a wider CDB. The extra integer ALU allows the DADDIU to execute earlier, in turn allowing the BNE to execute earlier, and, thereby, starting the next iteration earlier.

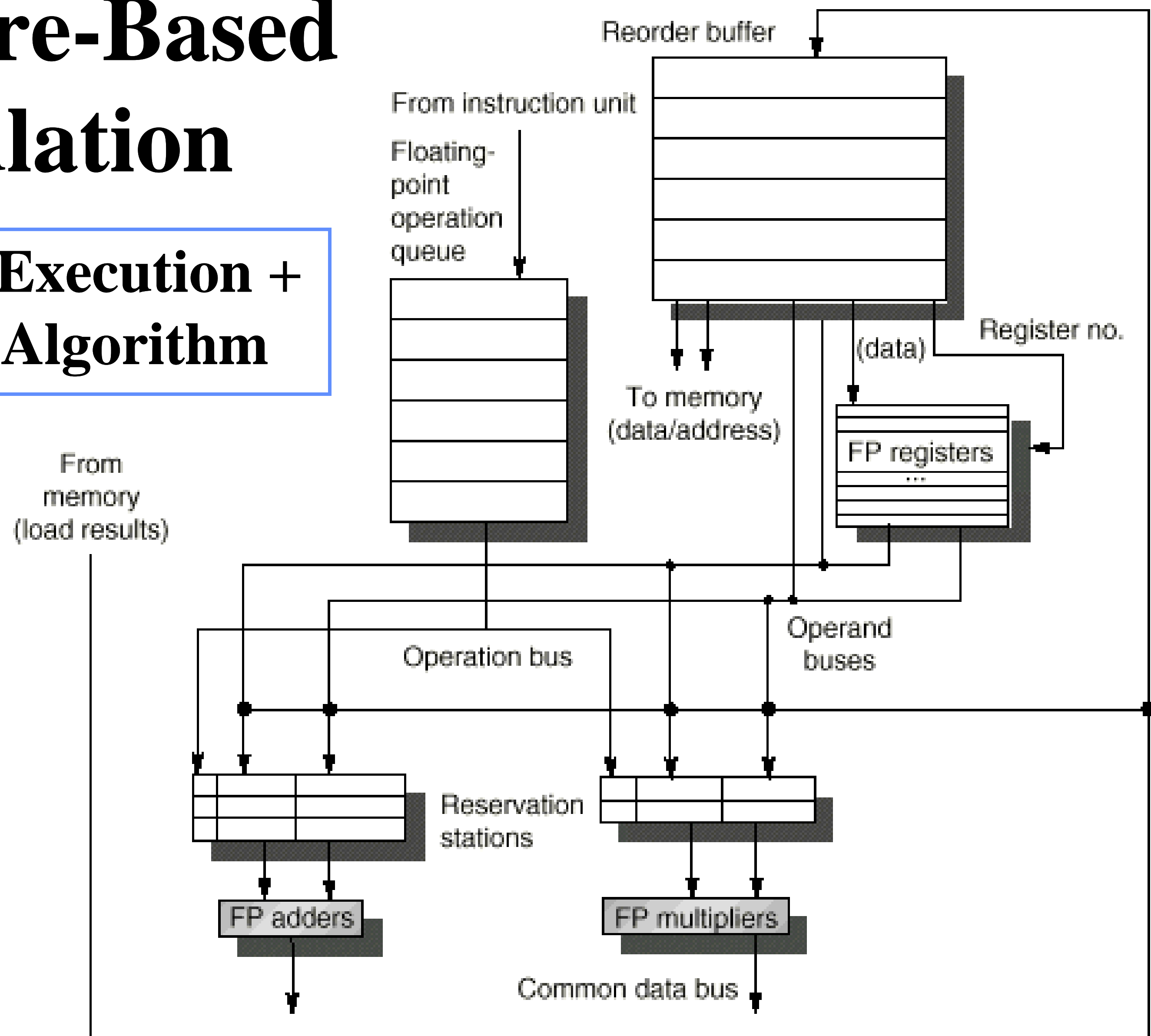
(page 224)

Dynamic Hardware-Based Speculation

- **Combines:**
  - **Dynamic hardware-based branch prediction**
  - **Dynamic Scheduling: issue multiple instructions in order and execute out of order. (Tomasulo)**
- **Continue to dynamically issue, and execute instructions passed a conditional branch in the dynamically predicted branch direction, before control dependencies are resolved.**
  - **This overcomes the ILP limitations of the basic block size.**
  - **Creates <u>dynamically speculated instructions</u> at run-time with no compiler support at all.**
  - **If a branch turns out as mispredicted all such dynamically speculated instructions must be prevented from changing the state of the machine (registers, memory).**
    - **<u>Addition of commit (retire, completion, or re-ordering) stage</u> and forcing instructions to <u>commit in their order in the code</u> (i.e to write results to registers or memory).**
    - **<u>Precise exceptions</u> are possible since instructions *must commit in order*.**

Hardware-Based Speculation

Speculative Execution + Tomasulo's Algorithm



Four Steps of Speculative Tomasulo Algorithm

1. **Issue — Get an instruction from FP Op Queue**

   If a reservation station and a reorder buffer slot are free, issue instruction & send operands & reorder buffer number for destination (this stage is sometimes called "dispatch")

2. **Execution — Operate on operands (EX)**

   When both operands are ready then execute; if not ready, watch CDB for result; when both operands are in reservation station, execute; checks RAW (sometimes called "issue")

3. **Write result — Finish execution (WB)**

   Write on Common Data Bus (CDB) to all awaiting FUs & reorder buffer; mark reservation station available.

4. **Commit — Update registers, memory with reorder buffer result**

   - **When an instruction is at head of reorder buffer & the result is present, update register with result (or store to memory) and remove instruction from reorder buffer.**
   - **A mispredicted branch at the head of the reorder buffer flushes the reorder buffer (sometimes called "graduation")**

   ⇒ **Instructions issue in order, execute (EX), write result (WB) out of order, but must commit in order.**

Multiple Issue with Speculation Example

Example Consider the execution of the following loop, which searches an array, on a two-issue processor, once without speculation and once with speculation:

```
Loop:   LD       R2,0(R1)      ;R2=array element
        DADDIU   R2,R2,#1      ;increment R2
        SD       R2,0(R1)      ;store result
        DADDIU   R1,R1,#4      ;increment pointer
        BNE      R2,R3,LOOP    ;branch if not last element
```

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table as in Figure 3.27 for the first three iterations of this loop for both machines. Assume that up to two instructions of any type can commit per clock.

Answer Figures 3.33 and 3.34 show the performance for a two-issue dynamically scheduled processor, without and with speculation. In this case, where a branch is a key potential performance limitation, speculation helps significantly. The third branch in the speculative processor executes in clock cycle 13, while it executes in clock cycle 19 on the nonspeculative pipeline. Because the completion rate on the nonspeculative pipeline is falling behind the issue rate rapidly, the nonspeculative pipeline will stall when a few more iterations are issued. The performance of the nonspeculative processor could be improved by allowing load instructions to

Branches still single issue

Example on page 235 (quiz 5)

Answer: Without Speculation

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#4	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#4	8	14		15	Wait for BNE
3	BNZ R2,R3,LOOP	9	19			Wait for DADDIU

Figure 3.33 The time of issue, execution, and writing result for a dual-issue version of our pipeline *without* speculation. Note that the L.D following the BNE cannot start execution earlier, because it must wait until the branch outcome is determined. This type of program, with data-dependent branches that cannot be resolved earlier, shows the strength of speculation. Separate functional units for address calculation, ALU operations, and branch condition evaluation allow multiple instructions to execute in the same cycle.

Answer: 2-way Superscalar Tomasulo With Speculation

Iteration number	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LW
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LW
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#4	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LW
3	SD R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#4	8	9		10	14	Executes earlier
3	BNE R2,R3,LOOP	9	13			14	Wait for DADDIU

Figure 3.34 The time of issue, execution, and writing result for a dual-issue version of our pipeline *with* speculation. Note that the L.D following the BNE can start execution early because it is speculative.

Branches Still Single Issue

Loop-Level Parallelism (LLP) Analysis

- **Loop-Level Parallelism (LLP) analysis focuses on whether data accesses in later iterations of a loop are data dependent on data values produced in earlier iterations and possibly making loop iterations independent.**

e.g. in

```
for (i=1; i<=1000; i++)
        x[i] = x[i] + s;
```

the computation in each iteration is independent of the previous iterations and the loop is thus parallel. The use of X[i] twice is within a single iteration.

⇒Thus loop iterations are parallel (or independent from each other).

- **Loop-carried Dependence: A data dependence between different loop iterations (data produced in earlier iteration used in a later one).**
- **LLP analysis is important in software optimizations such as loop unrolling since it usually requires loop iterations to be independent.**
- **LLP analysis is normally done at the source code level or close to it since assembly language and target machine code generation introduces a loop-carried name dependence in the registers used for addressing and incrementing.**
- **Instruction level parallelism (ILP) analysis, on the other hand, is usually done when instructions are generated by the compiler.**

(In Chapter 4.4)

LLP Analysis Example 1

- **In the loop:**

```
for (i=1; i<=100; i=i+1)  {
        A[i+1] = A[i] + C[i];  /*  S1 */
        B[i+1] = B[i] + A[i+1];}  /* S2 */
}
```

(Where A, B, C are distinct non-overlapping arrays)

- **S2 uses the value A[i+1], computed by S1 in the same iteration. This data dependence is within the same iteration <u>(not a loop-carried dependence).</u>**
  - **⇒ does not prevent loop iteration parallelism.**
- **S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] read in iteration i+1 <u>(loop-carried dependence, prevents parallelism).</u> The same applies for S2 for B[i] and B[i+1]**
  - **⇒These two dependencies are loop-carried spanning more than one iteration preventing loop parallelism.**

LLP Analysis Example 2

- **In the loop:**

```
for (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i];        /* S1 */
    B[i+1] − C[i] + D[i];      /* S2 */
}
```

  - **S1 uses the value B[i] computed by S2 in the previous iteration (loop-carried dependence)**
  - **This dependence is not circular:**
    - **S1 depends on S2 but S2 does not depend on S1.**
  - **Can be made parallel by replacing the code with the following:**

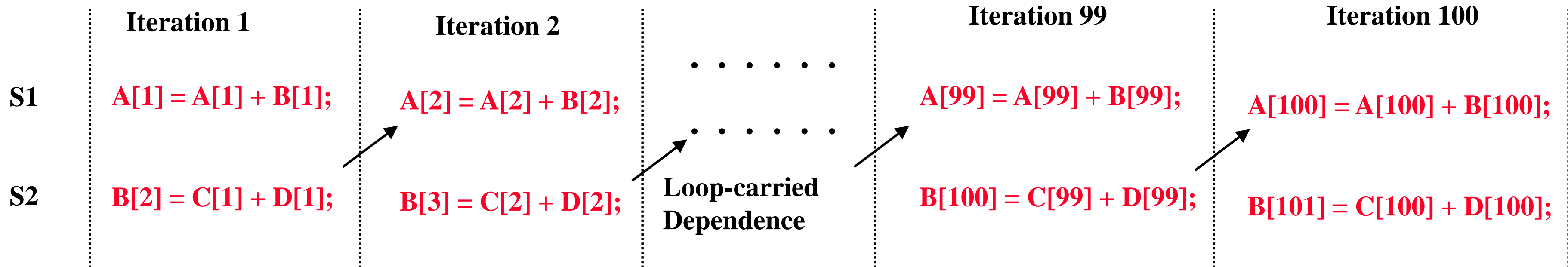
```
A[1] = A[1] + B[1];                 Loop Start-up code
for (i=1; i<=99; i=i+1)  {
    B[i+1] = C[i] + D[i];           Parallel
    A[i+1] = A[i+1] + B[i+1];       loop iterations
}
B[101] = C[100] + D[100];           Loop Completion code
```

(quiz 6)

LLP Analysis Example 2

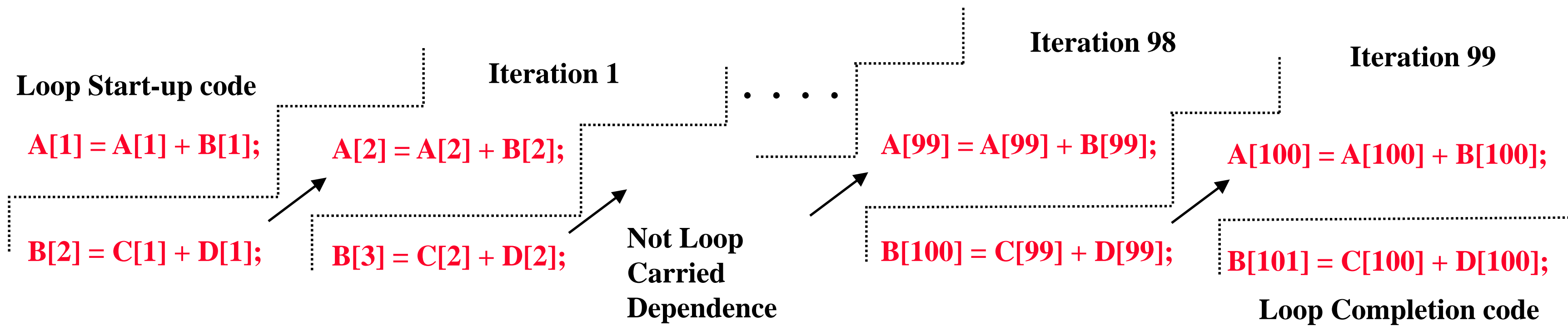
Original Loop:

```
for (i=1; i<=100; i=i+1) {
        A[i] = A[i] + B[i];        /*  S1  */
         B[i+1] = C[i] + D[i];    /*  S2  */
        }
```



Modified Parallel Loop:

```
A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1)  {
      B[i+1] = C[i] + D[i];
      A[i+1] − A[i+1] + B[i+1];
 }
 B[101] = C[100] + D[100];
```



ILP Compiler Support:

Software Pipelining (Symbolic Loop Unrolling)

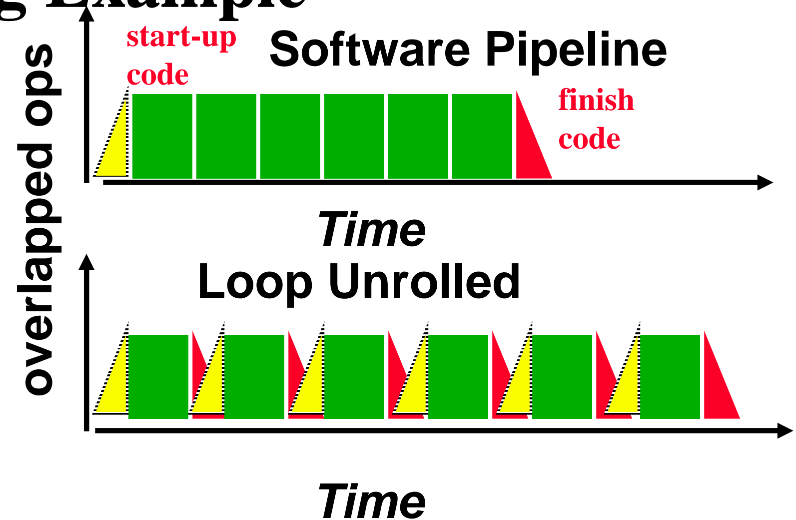
- **A compiler technique where loops are reorganized:**
  - **Each new iteration is made from <u>instructions selected</u> from <u>a number of independent iterations</u> of the original loop.**
- **The instructions are selected to separate dependent instructions within the original loop iteration.**
- **No actual loop-unrolling is performed.**
  - **A software equivalent to the Tomasulo approach?**
- **Requires:**
  - **Additional <u>start-up code</u> to execute code left out from the first original loop iterations.**
  - **Additional <u>finish code</u> to execute instructions left out from the last original loop iterations.**

(In Chapter 4.4)

Software Pipelining Example

Show a software-pipelined version of the code:

```
Loop:   L.D      F0,0(R1)
        ADD.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDUI   R1,R1,#-8
        BNE      R1,R2,LOOP
```



Before: Unrolled 3 times

```
1   L.D      F0,0(R1)
2   ADD.D    F4,F0,F2
3   S.D      F4,0(R1)
4   L.D      F0,-8(R1)
5   ADD.D    F4,F0,F2
6   S.D      F4,-8(R1)
7   L.D      F0,-16(R1)
8   ADD.D    F4,F0,F2
9   S.D      F4,-16(R1)
10  DADDUI   R1,R1,#-24
11  BNE      R1,R2,LOOP
```

After: Software Pipelined

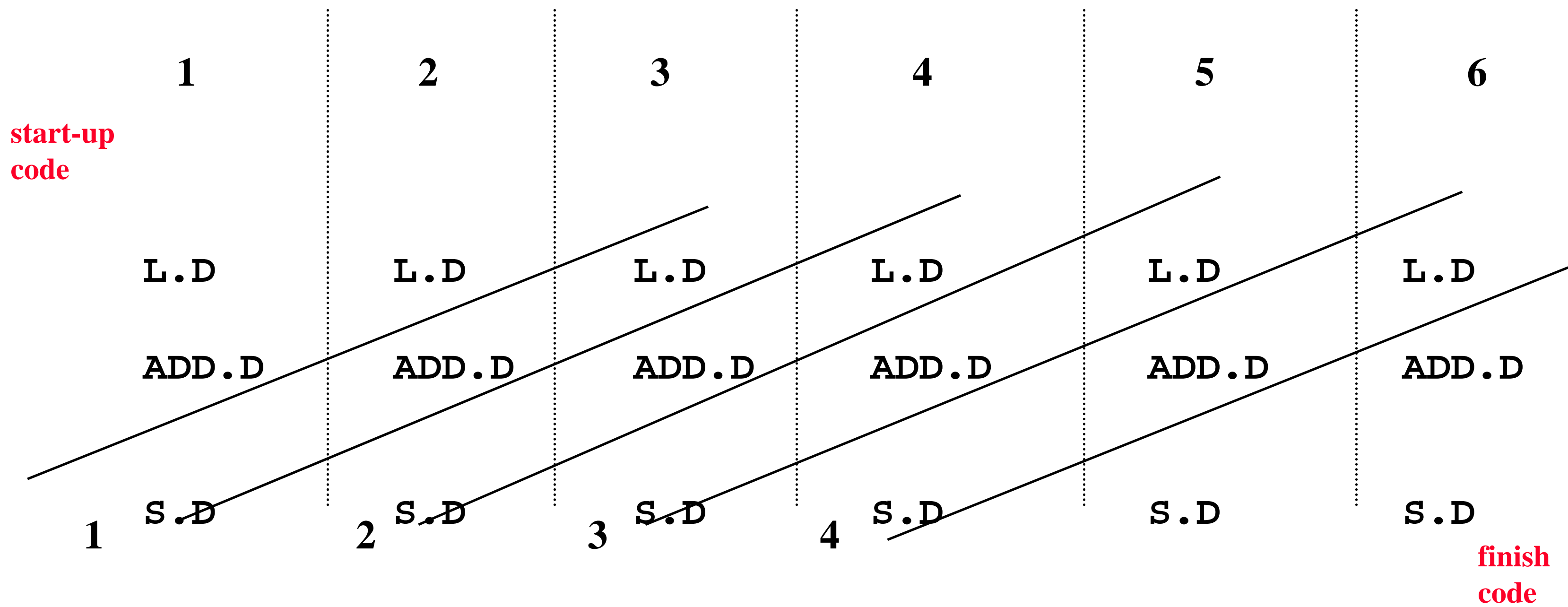
```
    L.D      F0,0(R1)          start-up
    ADD.D    F4,F0,F2          code
    L.D      F0,-8(R1)
1   S.D      F4,0(R1)    ;Stores M[i]
2   ADD.D    F4,F0,F2    ;Adds to M[i-1]
3   L.D      F0,-16(R1);Loads M[i-2]
4   DADDUI   R1,R1,#-8
5   BNE      R1,R2,LOOP
    S.D      F4, 0(R1)         finish
    ADDD     F4,F0,F2          code
    S.D      F4,-8(R1)
```

2 fewer loop iterations

Software Pipelining Example Illustrated

Assuming 6 original iterations for illustration purposes:

```
L.D      F0,0(R1)
ADD.D    F4,F0,F2
S.D      F4,0(R1)
```



4 Software Pipelined loop iterations (2 iterations fewer)

Cache Concepts

- **Cache is the first level of the memory hierarchy once the address leaves the CPU and is searched first for the requested data.**
- **If the data requested by the CPU is present in the cache, it is retrieved from cache and the data access is a cache hit otherwise a cache miss and data must be read from main memory.**
- **On a cache miss a block of data must be brought in from main memory to cache to possibly *replace* an existing cache block.**
- **The allowed block addresses where blocks can be mapped into cache from main memory is determined by *cache placement strategy*.**
- **Locating a block of data in cache is handled by cache block identification mechanism.**
- **On a cache miss the cache block being removed is handled by the *block replacement strategy* in place.**
- **When a write to cache is requested, a number of main memory update strategies exist as part of *the cache write policy*.**

(Review from 550)

Cache Performance:

Average Memory Access Time (AMAT), Memory Stall cycles

- **<u>The Average Memory Access Time (AMAT):</u> The number of cycles required to complete an average memory access request by the CPU.**
- **<u>Memory stall cycles per memory access:</u> The number of stall cycles added to CPU execution cycles for one memory access.**
- **For ideal memory: AMAT = 1 cycle, this results in zero memory stall cycles.**
- **Memory stall cycles per average memory access = (AMAT -1)**
- **Memory stall cycles per average instruction =**

Memory stall cycles per average memory access

x Number of memory accesses per instruction

= (AMAT -1) x (1 + fraction of loads/stores)

Instruction Fetch

Cache Performance

Princeton (Unified L1) Memory Architecture

CPUtime = Instruction count x CPI x Clock cycle time

$CPI_{execution}$ = CPI with ideal memory

CPI = $CPI_{execution}$ + Mem Stall cycles per instruction

CPUtime = Instruction Count x ($CPI_{execution}$ + Mem Stall cycles per instruction) x Clock cycle time

Mem Stall cycles per instruction = Mem accesses per instruction x Miss rate x Miss penalty

CPUtime = IC x ($CPI_{execution}$ + Mem accesses per instruction x Miss rate x Miss penalty) x Clock cycle time

Misses per instruction = Memory accesses per instruction x Miss rate

CPUtime = IC x ($CPI_{execution}$ + Misses per instruction x Miss penalty) x Clock cycle time

(Review from 550)

Memory Access Tree
For Unified Level 1 Cache

AMAT = H1 x 1 + (1 -H1) x (M+ 1) = 1 + M x (1 -H1)

Stall Cycles Per Access = AMAT - 1 = M x (1 -H1)

CPI = $CPI_{execution}$ + Mem accesses per instruction x M x (1 -H1)

M = Miss Penalty
H1 = Level 1 Hit Rate
1- H1 = Level 1 Miss Rate

Cache Performance

Harvard Memory Architecture

For a CPU with separate or split level one (L1) caches for instructions and data (Harvard memory architecture) and no stalls for cache hits:

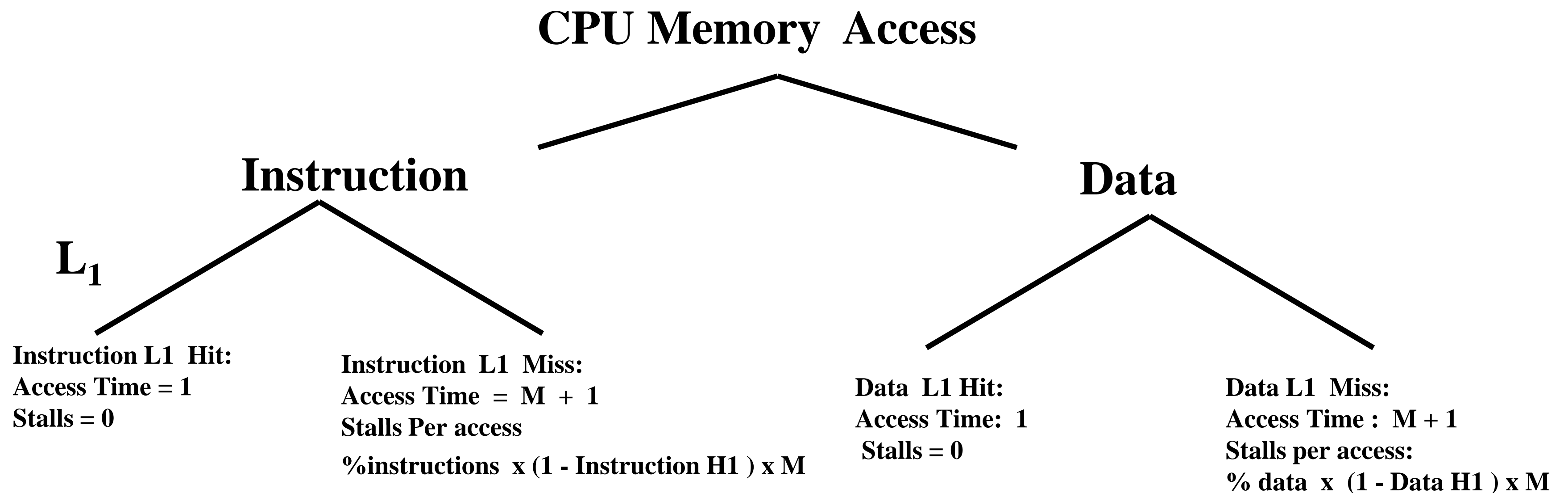
CPUtime = Instruction count x CPI x Clock cycle time

CPI = $CPI_{execution}$ + Mem Stall cycles per instruction

CPUtime = Instruction Count x ($CPI_{execution}$ + Mem Stall cycles per instruction) x Clock cycle time

Mem Stall cycles per instruction =
Instruction Fetch Miss rate x Miss Penalty +
Data Memory Accesses Per Instruction x Data Miss Rate x Miss Penalty

Memory Access Tree
For Separate Level 1 Caches



Stall Cycles Per Access = % Instructions x (1 - Instruction H1) x M + % data x (1 - Data H1) x M

AMAT = 1 + Stall Cycles per access

Cache Write Strategies

1. Write Though: Data is written to both the cache block and to a block of main memory.
   - The lower level always has the most updated data; an important feature for I/O and multiprocessing.
   - Easier to implement than write back.
   - A write buffer is often used to reduce CPU write stall while data is written to memory.
2. Write back: Data is written or updated only to the cache block. The modified or dirty cache block is written to main memory when it's being replaced from cache.
   - Writes occur at the speed of cache
   - A status bit called a dirty or modified bit, is used to indicate whether the block was modified while in cache; if not the block is not written back to main memory.
   - Uses less memory bandwidth than write through.

Cache Write Miss Policy

- **Since data is usually not needed immediately on a write miss two options exist on a cache write miss:**

Write Allocate:

The cache block is loaded on a write miss followed by write hit actions.

No-Write Allocate:

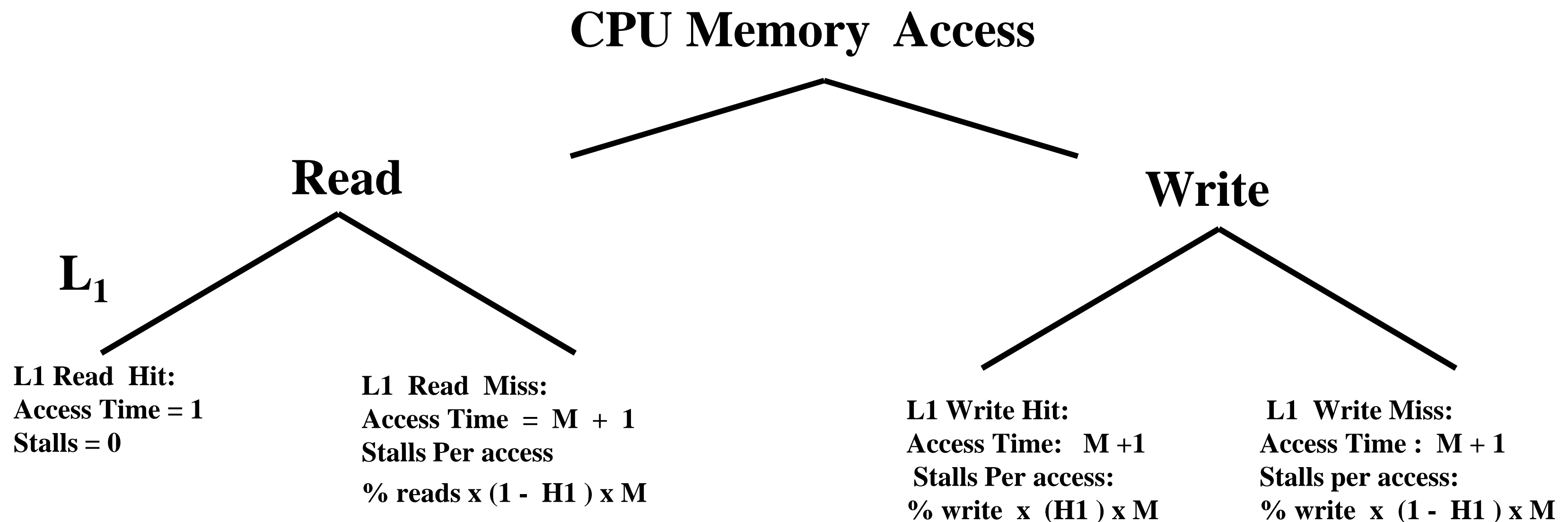
The block is modified in the lower level (lower cache level, or main memory) and not loaded into cache.

While any of the above two write miss policies can be used with either write back or write through:

- **Write back caches always use write allocate to capture subsequent writes to the block in cache.**
- **Write through caches usually use no-write allocate since subsequent writes still have to go to memory.**

Memory Access Tree, Unified L_1

Write Through, No Write Allocate, No Write Buffer



Stall Cycles Per Memory Access = % reads x (1 - H1) x M + % write x M

AMAT = 1 + % reads x (1 - H1) x M + % write x M

M = Miss Penalty
H1 = Level 1 Hit Rate
1- H1 = Level 1 Miss Rate

Reducing Write Stalls For Write Though Cache

- **To reduce write stalls when write though is used, a write buffer is used to eliminate or reduce write stalls:**
  - **<u>Perfect write buffer:</u> All writes are handled by write buffer, no stalling for writes**
  - **In this case:**

    Stall Cycles Per Memory Access = % reads x (1 - H1) x M

    (No stalls for writes)
  - **<u>Realistic Write buffer:</u> A percentage of write stalls are not eliminated when the write buffer is full.**
  - **In this case:**

Stall Cycles/Memory Access = (% reads x (1 - H1) + % write stalls not eliminated) x M

Write Through Cache Performance Example

- **A CPU with $CPI_{execution}$ = 1.1 Mem accesses per instruction = 1.3**
- **Uses a unified L1 Write Through, No Write Allocate, with:**
  - **No write buffer.**
  - **Perfect Write buffer**
  - **A realistic write buffer that eliminates 85% of write stalls**
- **Instruction mix: 50% arith/logic, 15% load, 15% store, 20% control**
- **Assume a cache miss rate of 1.5% and a miss penalty of 50 cycles.**

CPI = $CPI_{execution}$ + mem stalls per instruction

% reads = 1.15/1.3 = 88.5% % writes = .15/1.3 = 11.5%

With No Write Buffer :

Mem Stalls/ instruction = 1.3 x 50 x (88.5% x 1.5% + 11.5%) = 8.33 cycles

CPI = 1.1 + 8.33 = 9.43

With Perfect Write Buffer (all write stalls eliminated):

Mem Stalls/ instruction = 1.3 x 50 x (88.5% x 1.5%) = 0.86 cycles

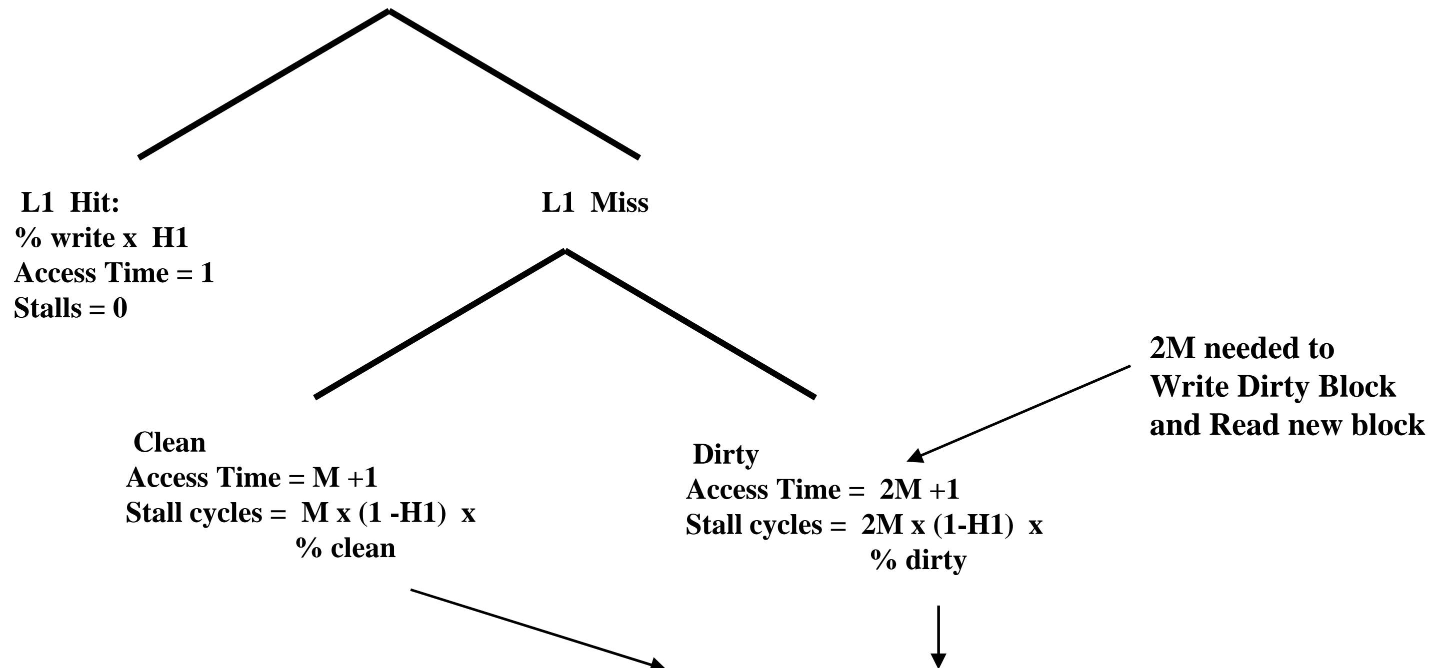
CPI = 1.1 + 0.86 = 1.96

With Realistic Write Buffer (eliminates 85% of write stalls)

Mem Stalls/ instruction = 1.3 x 50 x (88.5% x 1.5% + 15% x 11.5%) = 1.98 cycles

CPI = 1.1 + 1.98 = 3.08

Memory Access Tree Unified L_1
Write Back, With Write Allocate



Stall Cycles Per Memory Access = (1-H1) x (M x % clean + 2M x % dirty)

AMAT = 1 + Stall Cycles Per Memory Access

Write Back Cache Performance Example

- **A CPU with $CPI_{execution}$ = 1.1 uses a unified L1 with with write back, with write allocate, and the probability a cache block is dirty = 10%**
- **Instruction mix: 50% arith/logic, 15% load, 15% store, 20% control**
- **Assume a cache miss rate of 1.5% and a miss penalty of 50 cycles.**

CPI = $CPI_{execution}$ + mem stalls per instruction

Mem Stalls per instruction =

Mem accesses per instruction x Stalls per access

Mem accesses per instruction = 1 + .3 = 1.3

Stalls per access = (1-H1) x (M x % clean + 2M x % dirty)

Stalls per access = 1.5% x (50 x 90% + 100 x 10%) = .825 cycles

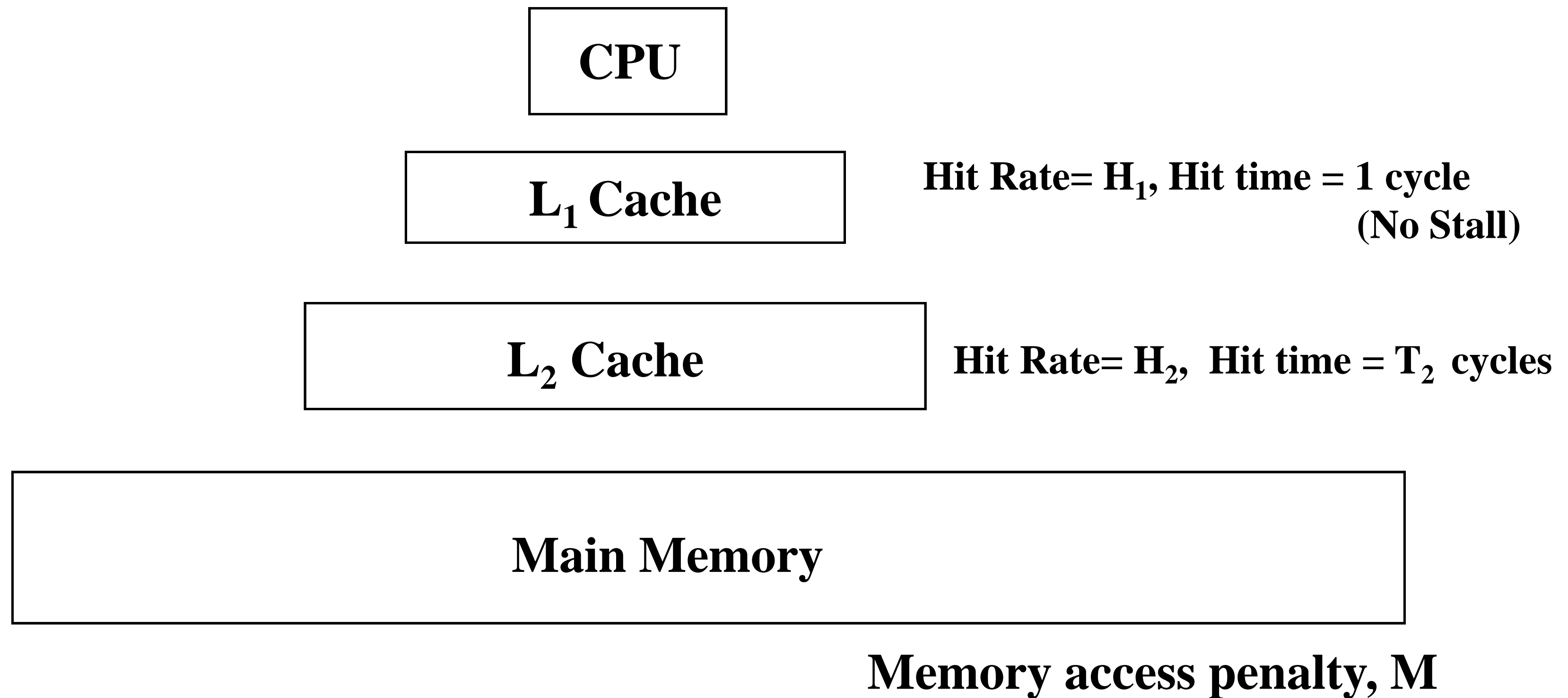
Mem Stalls per instruction = 1.3 x .825 = 1.07 cycles

AMAT = 1 + 1.07 = 2.07 cycles

CPI = 1.1 + 1.07 = 2.17

The ideal CPU with no misses is 2.17/1.1 − 1.97 times faster

2 Levels of Unified Cache: L_1, L_2



Miss Rates For Multi-Level Caches

- **Local Miss Rate:** This rate is the number of misses in a cache level divided by the number of memory accesses to this level. Local Hit Rate = 1 - Local Miss Rate

- **Global Miss Rate:** The number of misses in a cache level divided by the total number of memory accesses generated by the CPU.
- **Since level 1 receives all CPU memory accesses, for level 1:**

  Local Miss Rate = Global Miss Rate = 1 - H1
- **For level 2 since it only receives those accesses missed in 1:**

  Local Miss Rate = Miss rate$_{L2}$= 1- H2

  Global Miss Rate = Miss rate$_{L1}$ x Miss rate$_{L2}$

  = (1- H1) x (1 - H2)

2-Level (Both Unified) Cache Performance (Ignoring Write Policy)

CPUtime = IC x (CPI$_{execution}$ + Mem Stall cycles per instruction) x C

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

- **For a system with 2 levels of cache, assuming no penalty when found in L_1 cache:**

Stall cycles per memory access =

[miss rate L_1] x [Hit rate L_2 x Hit time L_2

+ Miss rate L_3 x Memory access penalty)] =

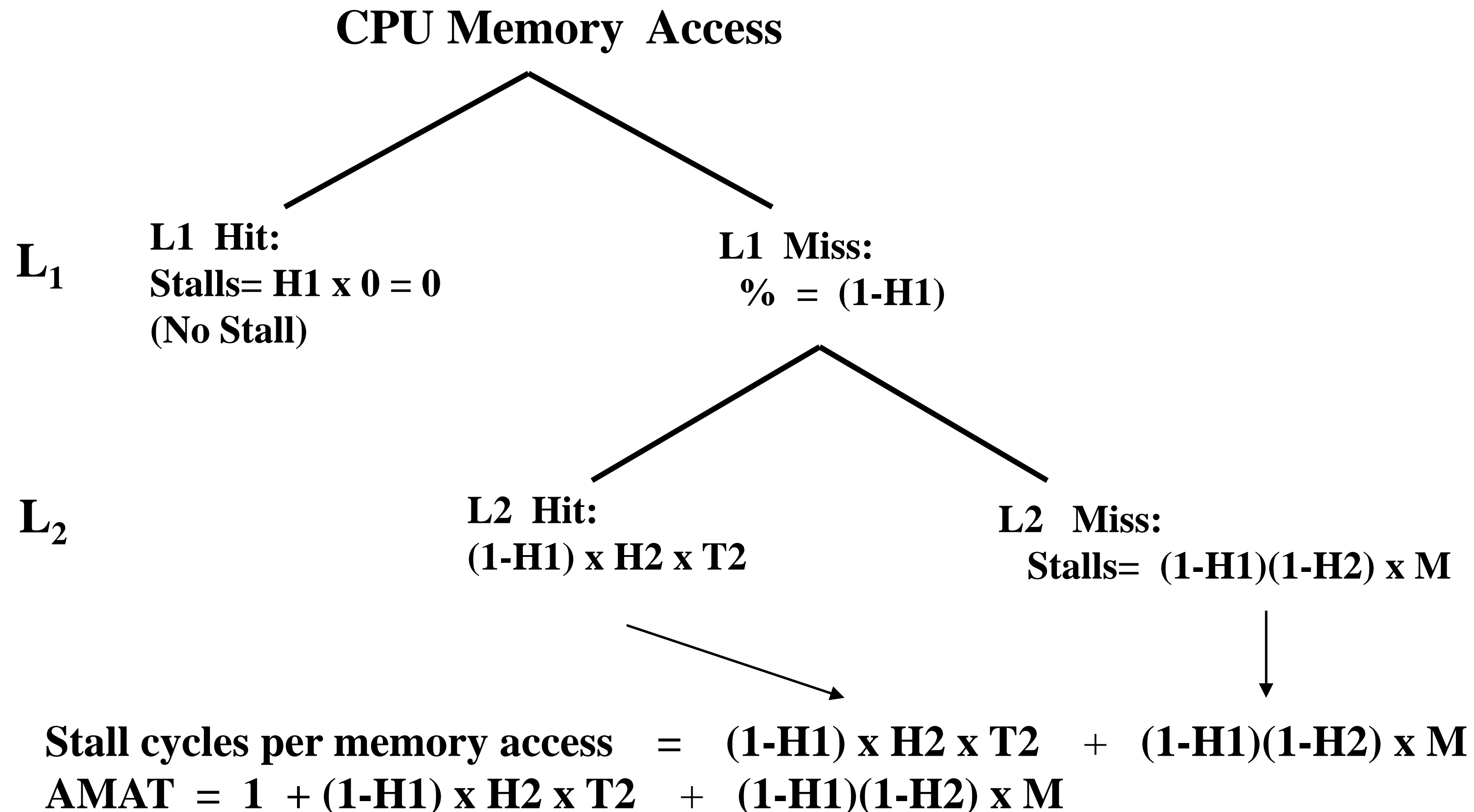
(1-H1) x H2 x T2 + (1-H1)(1-H2) x M

L1 Miss, L2 Hit

L1 Miss, L2 Miss: Must Access Main Memory

2-Level Cache (Both Unified) Performance Memory Access Tree (Ignoring Write Policy)

CPU Stall Cycles Per Memory Access



Stall cycles per memory access = (1-H1) x H2 x T2 + (1-H1)(1-H2) x M

AMAT = 1 + (1-H1) x H2 x T2 + (1-H1)(1-H2) x M

Two-Level Cache Example

- **CPU with $CPI_{execution}$ = 1.1 running at clock rate = 500 MHz**
- **1.3 memory accesses per instruction.**
- **L_1 cache operates at 500 MHz with a miss rate of 5%**
- **L_2 cache operates at 250 MHz with local miss rate 40%, (T_2 = 2 cycles)**
- **Memory access penalty, M = 100 cycles. Find CPI.**

CPI = $CPI_{execution}$ + Mem Stall cycles per instruction

With No Cache, CPI = 1.1 + 1.3 x 100 = 131.1

With single L_1, CPI = 1.1 + 1.3 x .05 x 100 = 7.6

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

Stall cycles per memory access = (1-H1) x H2 x T2 + (1-H1)(1-H2) x M

= .05 x .6 x 2 + .05 x .4 x 100

= .06 + 2 = 2.06

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

= 2.06 x 1.3 = 2.678

CPI = 1.1 + 2.678 = 3.778

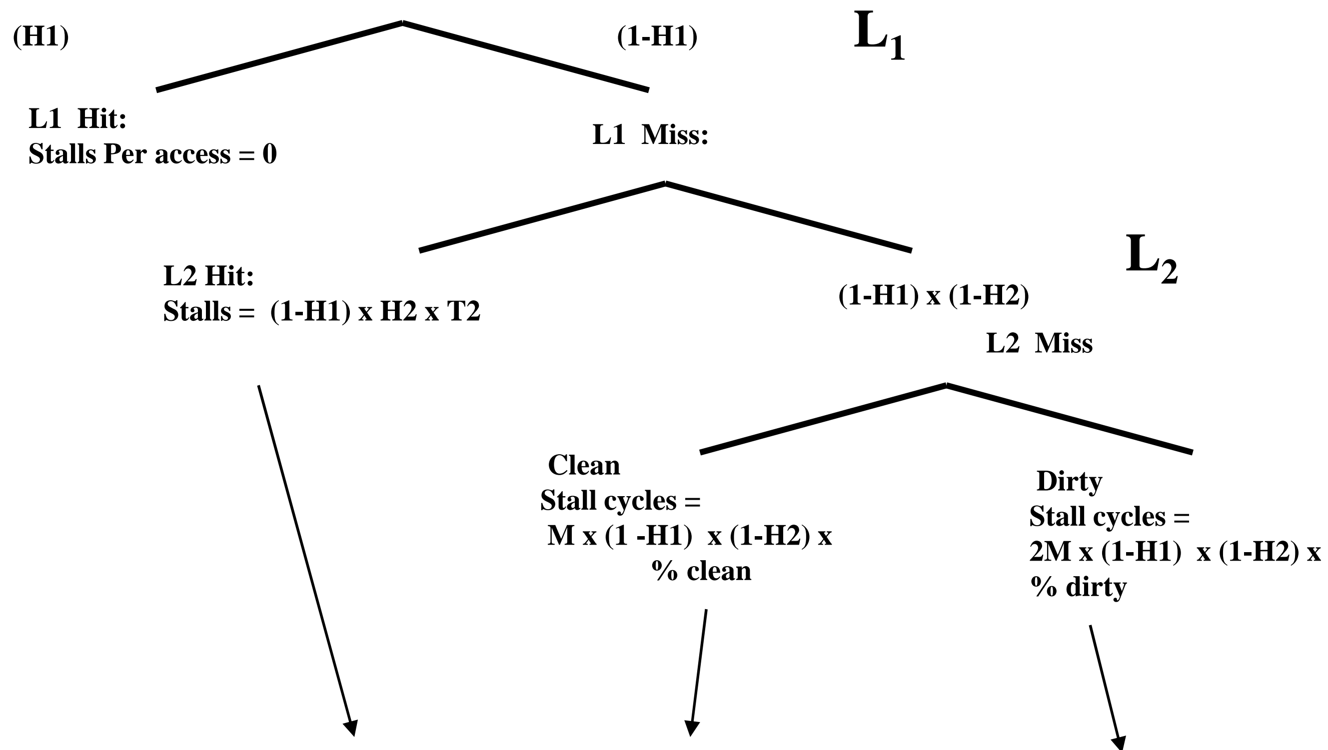
Speedup = 7.6/3.778 = 2

Write Policy For 2-Level Cache

- **Write Policy For Level 1 Cache:**
  - **Usually Write through to Level 2**
  - **Write allocate is used to reduce level 1 miss reads.**
  - **Use write buffer to reduce write stalls**
- **Write Policy For Level 2 Cache:**
  - **Usually write back with write allocate is used.**
    - **To minimize memory bandwidth usage.**
- **The above 2-level cache write policy results in inclusive L2 cache since the content of L1 is also in L2**
    - **Common in the majority of all CPUs with 2-levels of cache**

2-Level (Both Unified) Memory Access Tree

L1: Write Through to L2, Write Allocate, With Perfect Write Buffer
L2: Write Back with Write Allocate



Stall cycles per memory access = (1-H1) x H2 x T2 + M x (1 -H1) x (1-H2) x % clean + 2M x (1-H1) x (1-H2) x % dirty

= (1-H1) x H2 x T2 + (1 -H1) x (1-H2) x (% clean x M + % dirty x 2M)

(quiz 7)

Two-Level Cache Example With Write Policy

- **CPU with $CPI_{execution}$ = 1.1 running at clock rate = 500 MHz**
- **1.3 memory accesses per instruction.**
- **For L_1 :**
  - **Cache operates at 500 MHz with a miss rate of 1-H1 = 5%**
  - **Write though to L_2 with perfect write buffer with write allocate**
- **For L_2:**
  - **Cache operates at 250 MHz with local miss rate 1- H2 = 40%, (T_2 = 2 cycles)**
  - **Write back to main memory with write allocate**
  - **Probability a cache block is dirty = 10%**
- **Memory access penalty, M = 100 cycles. Find CPI.**
- Stall cycles per memory access = (1-H1) x H2 x T2 +

  (1 -H1) x (1-H2) x (% clean x M + % dirty x 2M)

  = .05 x .6 x 2 + .05 x .4 x (.9 x 100 + .1 x200)

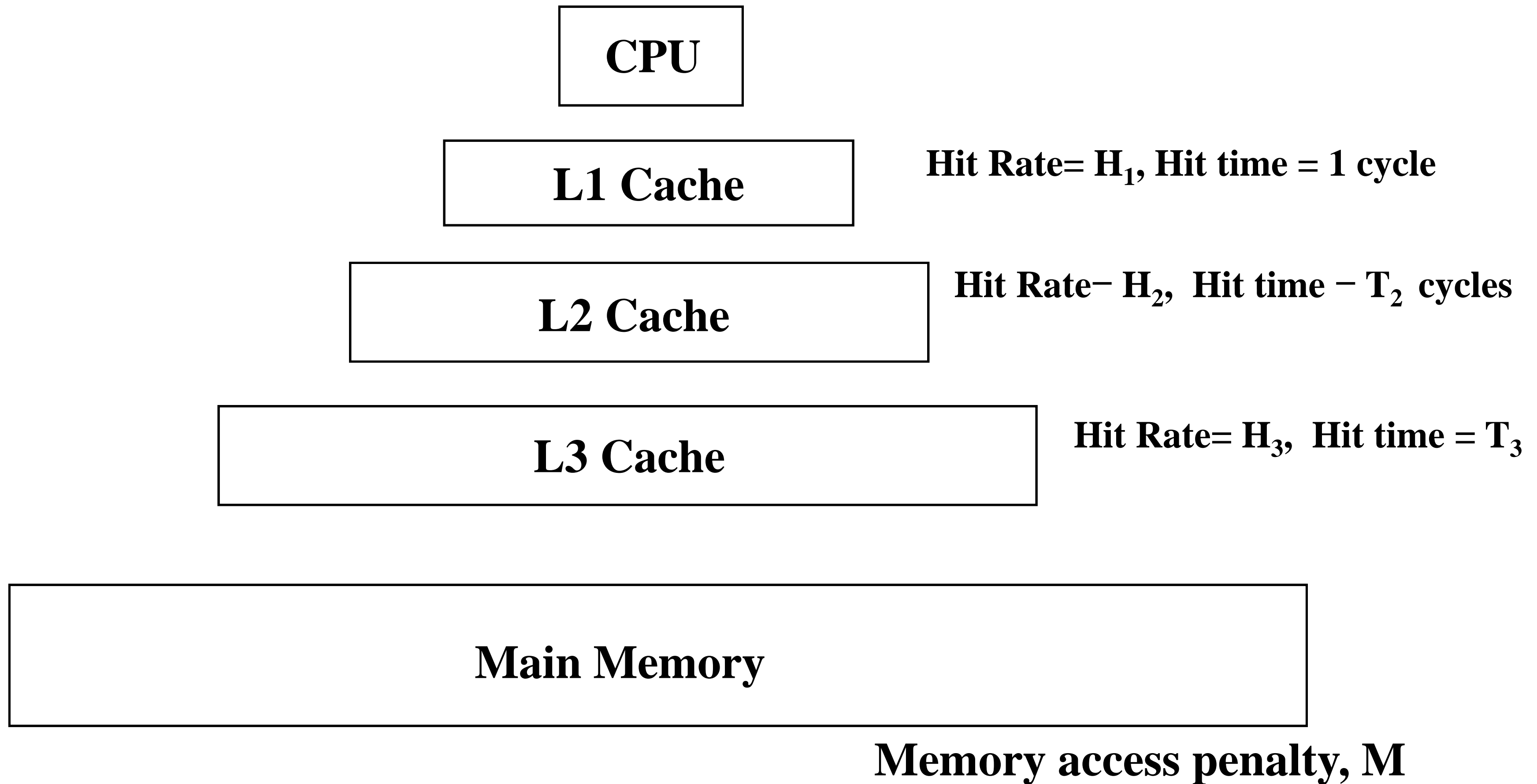
  = .06 + 0.02 x 110 = .06 + 2.2 = 2.26

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

= 2.26 x 1.3 = 2.938

CPI = 1.1 + 2.938 = 4.038 = 4

3 Levels of Unified Cache



3-Level Cache Performance

(Ignoring Write Policy)

CPUtime = IC x ($CPI_{execution}$ + Mem Stall cycles per instruction) x C

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

- **For a system with 3 levels of cache, assuming no penalty when found in L_1 cache:**

Stall cycles per memory access =

[miss rate L_1] x [Hit rate L_2 x Hit time L_2

+ Miss rate L_2 x (Hit rate L3 x Hit time L_3

+ Miss rate L_3 x Memory access penalty)] =

(1-H1) x H2 x T2

+ (1-H1) x (1-H2) x H3 x T3

+ (1-H1)(1-H2) (1-H3)x M

L1 Miss, L2 Hit

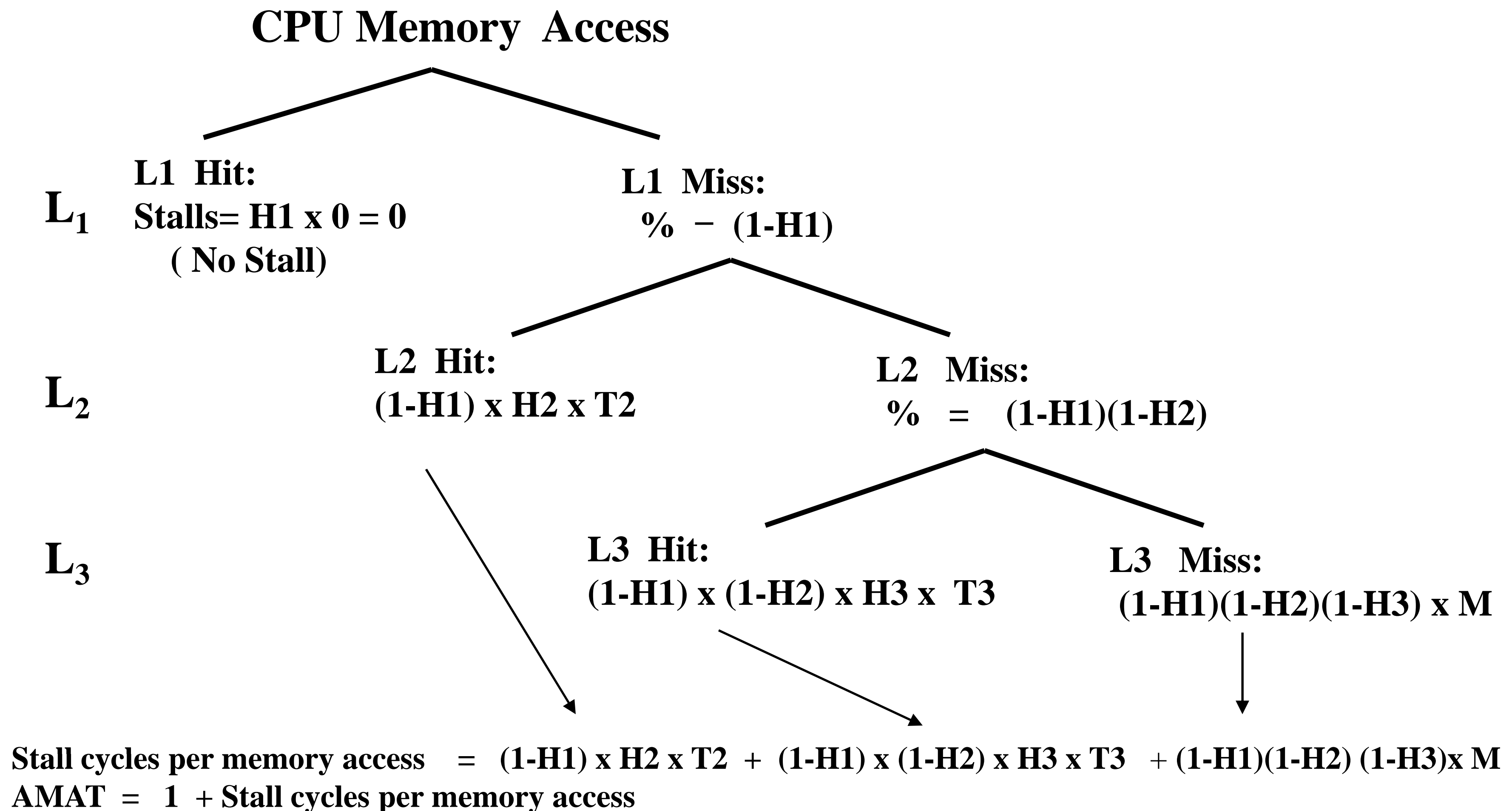
L2 Miss, L3 Hit

L1 Miss, L2 Miss: Must Access Main Memory

3-Level Cache Performance

Memory Access Tree (Ignoring Write Policy)

CPU Stall Cycles Per Memory Access



Stall cycles per memory access = (1-H1) x H2 x T2 + (1-H1) x (1-H2) x H3 x T3 + (1-H1)(1-H2) (1-H3)x M
AMAT = 1 + Stall cycles per memory access

Three-Level Cache Example

- **CPU with $CPI_{execution}$ = 1.1 running at clock rate = 500 MHz**
- **1.3 memory accesses per instruction.**
- **L_1 cache operates at 500 MHz with a miss rate of 5%**
- **L_2 cache operates at 250 MHz with a local miss rate 40%, (T_2 = 2 cycles)**
- **L_3 cache operates at 100 MHz with a local miss rate 50%, (T_3 = 5 cycles)**
- **Memory access penalty, M= 100 cycles. Find CPI.**

With No Cache, CPI = 1.1 + 1.3 x 100 = 131.1

With single L_1, CPI = 1.1 + 1.3 x .05 x 100 = 7.6

With L1, L2 CPI = 1.1 + 1.3 x (.05 x .6 x 2 + .05 x .4 x 100) = 3.778

CPI = $CPI_{execution}$ + Mem Stall cycles per instruction

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

Stall cycles per memory access = (1-H1) x H2 x T2 + (1-H1) x (1-H2) x H3 x T3 + (1-H1)(1-H2) (1-H3)x M

= .05 x .6 x 2 + .05 x .4 x .5 x 5 + .05 x .4 x .5 x 100

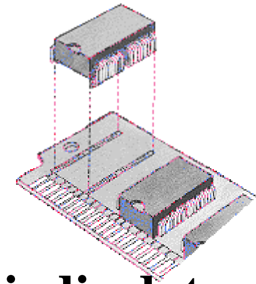
= .097 + .0075 + .00225 = 1.11

CPI = 1.1 + 1.3 x 1.11 = 2.54

Speedup compared to L1 only = 7.6/2.54 = 3

Speedup compared to L1, L2 = 3.778/2.54 = 1.49

Main Memory



- **Main memory generally utilizes Dynamic RAM (DRAM), which use a single transistor to store a bit, but require a periodic data refresh by reading every row.**
- **Static RAM may be used for main memory if the added expense, low density, high power consumption, and complexity is feasible (e.g. Cray Vector Supercomputers).**
- **Main memory performance is affected by:**
  - **Memory latency: Affects cache miss penalty, M. Measured by:**
    - **Access time: The time it takes between a memory access request is issued to main memory and the time the requested information is available to cache/CPU.**
    - **Cycle time: The minimum time between requests to memory (greater than access time in DRAM to allow address lines to be stable)**
  - **Memory bandwidth: The maximum sustained data transfer rate between main memory and cache/CPU.**

(In Chapter 5.8 - 5.10)

Basic Memory Bandwidth Improvement Techniques

- **Wider Main Memory:**

  Memory width is increased to a number of words (usually up to the size of a cache block).

  ⇒ **Memory bandwidth is proportional to memory width.**

  e.g Doubling the width of cache and memory doubles potential memory bandwidth available to the CPU.

- **Interleaved (Multi-Bank) Memory:**

  Memory is organized as a number of independent banks.

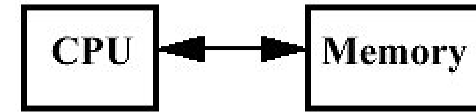
  - **Multiple interleaved memory reads or writes are accomplished by sending memory addresses to several memory banks at once.**
  - **Interleaving factor: Refers to the mapping of memory addressees to memory banks. Goal reduce bank conflicts.**

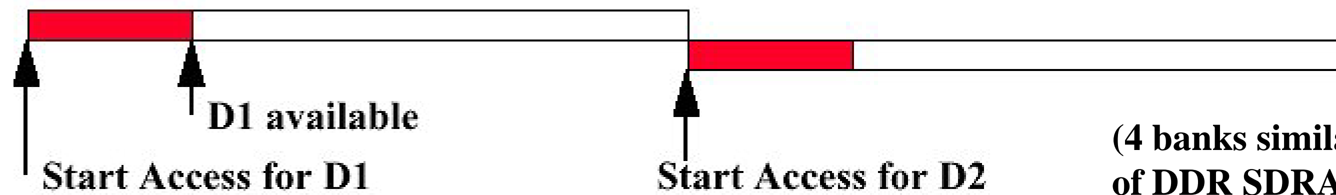
    e.g. using 4 banks (width one word), bank 0 has all words whose address is:

    (word address mod) 4 = 0

Memory Bank Interleaving

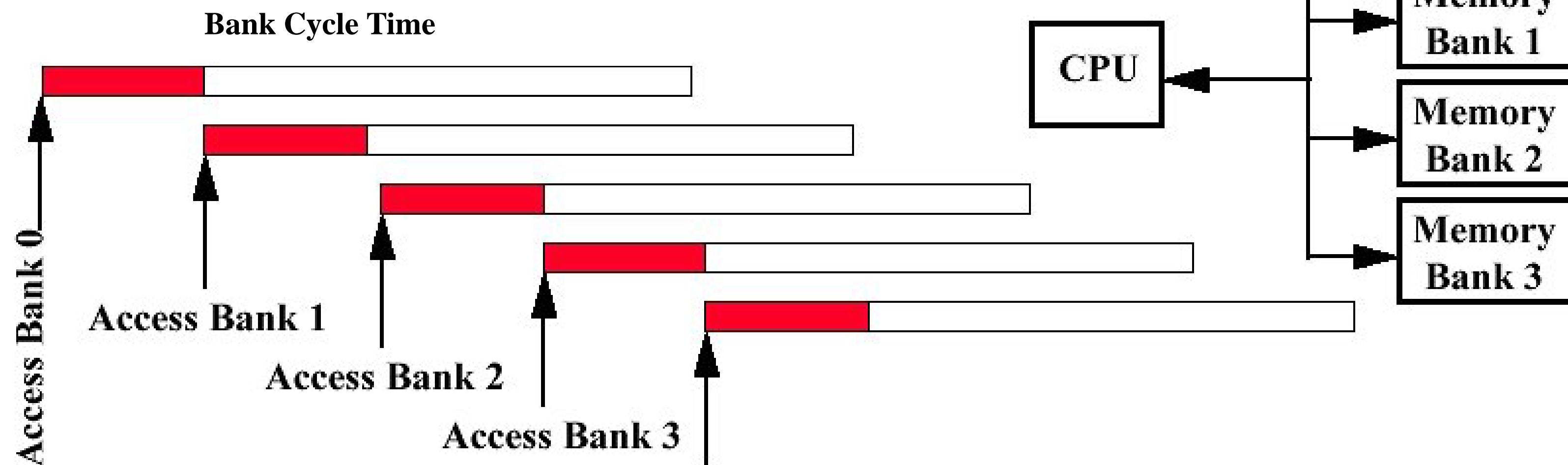
Access Pattern without Interleaving: (One Bank)





(4 banks similar to the organization of DDR SDRAM memory chips

Access Pattern with 4-way Interleaving:



Number of banks $\geq$ Number of cycles to access word in a bank

Memory Width, Interleaving: Performance Example

Given the following system parameters with single unified cache level L_1 (ignoring write policy):

Block size= 1 word Memory bus width= 1 word Miss rate =3% M = Miss penalty = 32 cycles
(4 cycles to send address 24 cycles access time, 4 cycles to send a word)

Memory access/instruction = 1.2 $CPI_{execution}$ (ignoring cache misses) = 2

Miss rate (block size = 2 word = 8 bytes) = 2% Miss rate (block size = 4 words = 16 bytes) = 1%

- **The CPI of the base machine with 1-word blocks = 2 + (1.2 x 0.03 x 32) = 3.15**

Increasing the <u>block size</u> to <u>two words</u> gives the following CPI:

- **32-bit bus and memory, no interleaving, M = 2 x 32 = 64 cycles CPI = 2 + (1.2 x .02 x 64) = 3.54**
- **32-bit bus and memory, interleaved, M = 4 + 24 + 8 = 36 cycles CPI = 2 + (1.2 x .02 x 36) = 2.86**
- **64-bit bus and memory, no interleaving, M = 32 cycles CPI = 2 + (1.2 x 0.02 x 32) = 2.77**

Increasing the <u>block size</u> to <u>four words</u>; resulting CPI:

- **32-bit bus and memory, no interleaving , M = 4 x 32 = 128 cycles CPI = 2 + (1.2 x 0.01 x 128) = 3.54**
- **32-bit bus and memory, interleaved , M = 4 + 24 + 16 = 44 cycles CPI = 2 + (1.2 x 0.01 x 44) = 2.53**
- **64-bit bus and memory, no interleaving, M = 2 x 32 = 64 cycles CPI = 2 + (1.2 x 0.01 x 64) = 2.77**
- **64-bit bus and memory, interleaved, M = 4 + 24 + 8 = 36 cycles CPI = 2 + (1.2 x 0.01 x 36) = 2.43**
- **128-bit bus and memory, no interleaving, M = 32 cycles CPI = 2 + (1.2 x 0.01 x 32) = 2.38**

Program Steady-State Main Memory Bandwidth-Usage Example

- **In the example with three levels of cache (all unified, ignore write policy)**
- **CPU with $CPI_{execution}$ = 1.1 running at clock rate = 500 MHZ**
- **1.3 memory accesses per instruction.**
- **L_1 cache operates at 500 MHz with a miss rate of 5%**
- **L_2 cache operates at 250 MHz with a local miss rate 40%, (T_2 = 2 cycles)**
- **L_3 cache operates at 100 MHz with a local miss rate 50%, (T_3 = 5 cycles)**
- **Memory access penalty, M= 100 cycles.**

- **We found the CPI:**

With No Cache, CPI = 1.1 + 1.3 x 100 = 131.1

With single L_1, CPI = 1.1 + 1.3 x .05 x 100 = 7.6

With L1, L2 CPI = 1.1 + 1.3 x (.05 x .6 x 2 + .05 x .4 x 100) = 3.778

With L1, L2 , L3 CPI = 1.1 + 1.3 x 1.11 = 2.54

Assuming:

instruction size = data size = 4 bytes , all cache blocks are 32 bytes and

For each of the three cases with cache:

What is the total number of memory accesses generated by the CPU per second?

What is the percentage of these memory accesses satisfied by main memory?

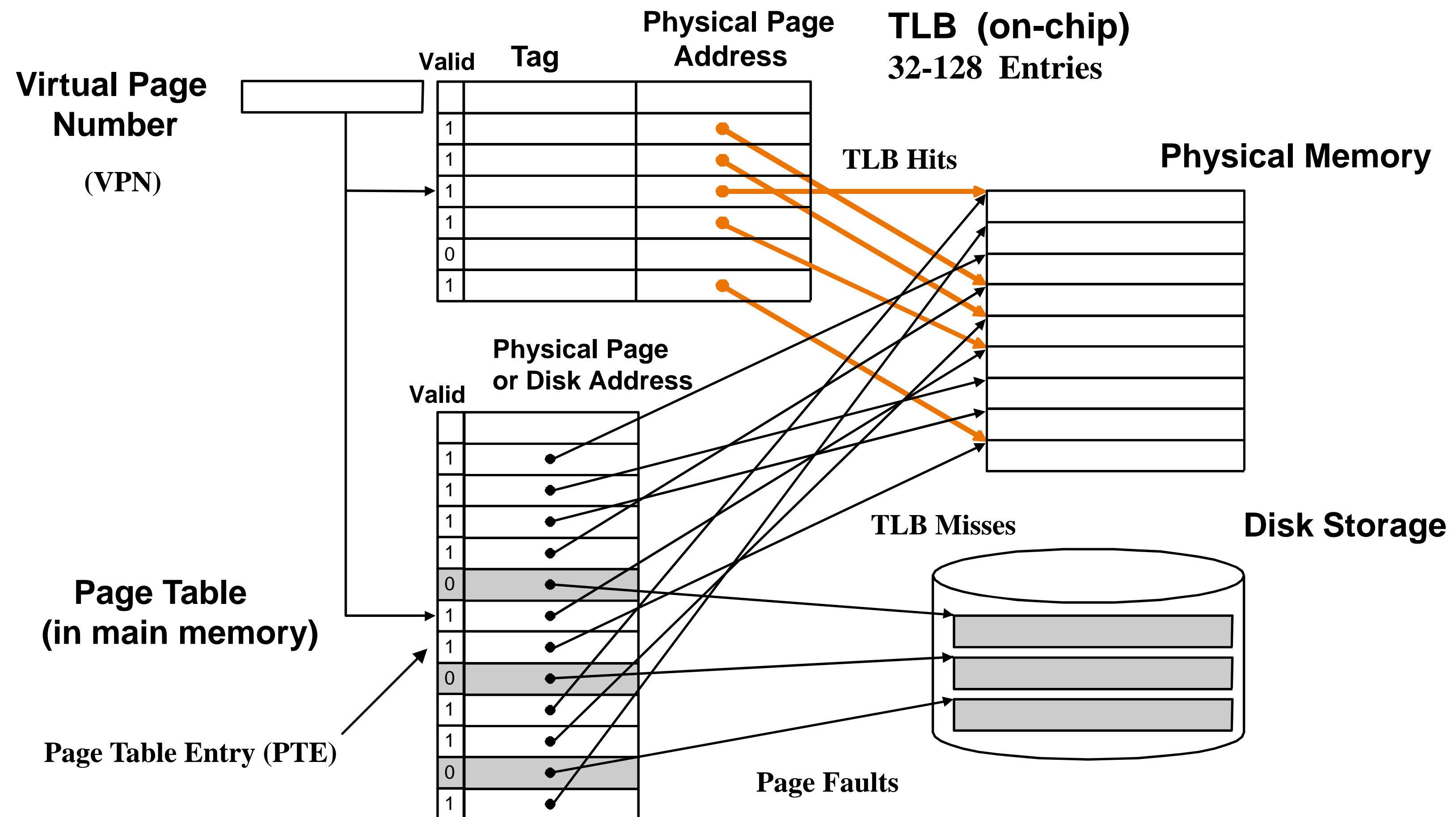
Percentage of main memory bandwidth used by the CPU?

Program Steady-State Main Memory Bandwidth-Usage Example

- **Memory requires 100 CPU cycles = 200 ns to deliver 32 bytes, thus total main memory bandwidth = 32 bytes / (200 ns) = 160 x 10^6 bytes/sec**
- **The total number of memory accesses generated by the CPU per second = (memory access/instruction) x clock rate / CPI = 1.3 x 500 x 10^6 / CPI = 650 x 10^6 / CPI**
  - **With single L1 = 650 x 10^6 / 7.6 = 85 x 10^6 accesses/sec**
  - **With L1, L2 = 650 x 10^6 / 3.778 = 172 x 10^6 accesses/sec**
  - **With L1, L2, L3 = 650 x 10^6 / 2.54 = 255 x 10^6 accesses/sec**
- **The percentage of these memory accesses satisfied by main memory:**
  - **With single L1 = L1 miss rate = 5%**
  - **With L1, L2 = L1 miss rate x L2 miss rate = .05 x .4 = 2%**
  - **with L1, L2, L3 = L1 miss rate x L2 miss rate x L3 miss rate = .05 x .4 x . 5 = 1%**
- **Memory Bandwidth used**
  - **With single L1 = 32 bytes x 85x10^6 accesses/sec x .05 = 136 x10^6 bytes/sec**
    or 136/160 = 85 % of total memory bandwidth
  - **With L1, L2 = 32 bytes x 172 x10^6 accesses/sec x .02 = 110 x10^6 bytes/sec**
    or 110/160 = 69 % of total memory bandwidth
  - **With L1, L2, L3 = 32 bytes x 255 x10^6 accesses/sec x .01 = 82 x10^6 bytes/sec**
    or 82/160 = 51 % of total memory bandwidth

Virtual Memory, Speeding Up Address Translation: Translation Lookaside Buffer (TLB)

- TLB: A small on-chip fully-associative cache used for address translations.
- If a virtual address is found in TLB (a TLB hit), the page table in main memory is not accessed.



CPU Performance with Real TLBs

When a real TLB is used with a TLB miss rate and a TLB miss penalty is used:

CPI = $CPI_{execution}$ + mem stalls per instruction + TLB stalls per instruction

Where:

Mem Stalls per instruction = Mem accesses per instruction x mem stalls per access

Similarly:

TLB Stalls per instruction = Mem accesses per instruction x TLB stalls per access

TLB stalls per access = TLB miss rate x TLB miss penalty

Example:

Given: $CPI_{execution}$ = 1.3 Mem accesses per instruction = 1.4

Mem stalls per access = .5 TLB miss rate = .3% TLB miss penalty = 30 cycles

What is the reluting CPU CPI?

Mem Stalls per instruction = 1.4 x .5 = .7 cycles/instruction

TLB stalls per instruction = 1.4 x (TLB miss rate x TLB miss penalty)

= 1.4 x .003 x 30 = .126 cycles/instruction

CPI = 1. 3 + .7 + .126 = 2.126

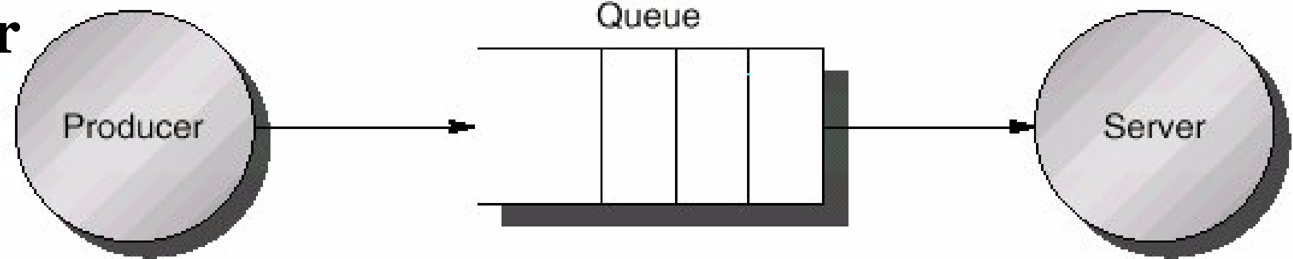
I/O Performance Metrics

- **Diversity:** **The variety of I/O devices that can be connected to the system.**
- **Capacity:** **The maximum number of I/O devices that can be connected to the system.**
- **Producer/server Model of I/O:** **The producer (CPU, human etc.) creates tasks to be performed and places them in a task buffer (queue); the server (I/O device or controller) takes tasks from the queue and performs them.**
- **I/O Throughput:** **The maximum data rate that can be transferred to/from an I/O device or sub-system, or the maximum number of I/O tasks or transactions completed by I/O in a certain period of time**
  - ⇒ **Maximized when task buffer is never empty.**
- **I/O Latency or response time:** **The time an I/O task takes from the time it is placed in the task buffer or queue until the server (I/O system) finishes the task. Includes buffer waiting or queuing time.**
  - ⇒ **Maximized when task buffer is always empty.**

In textbook: Ch. 7.1-7.3, 7.7, 7.8

Producer-Server Model

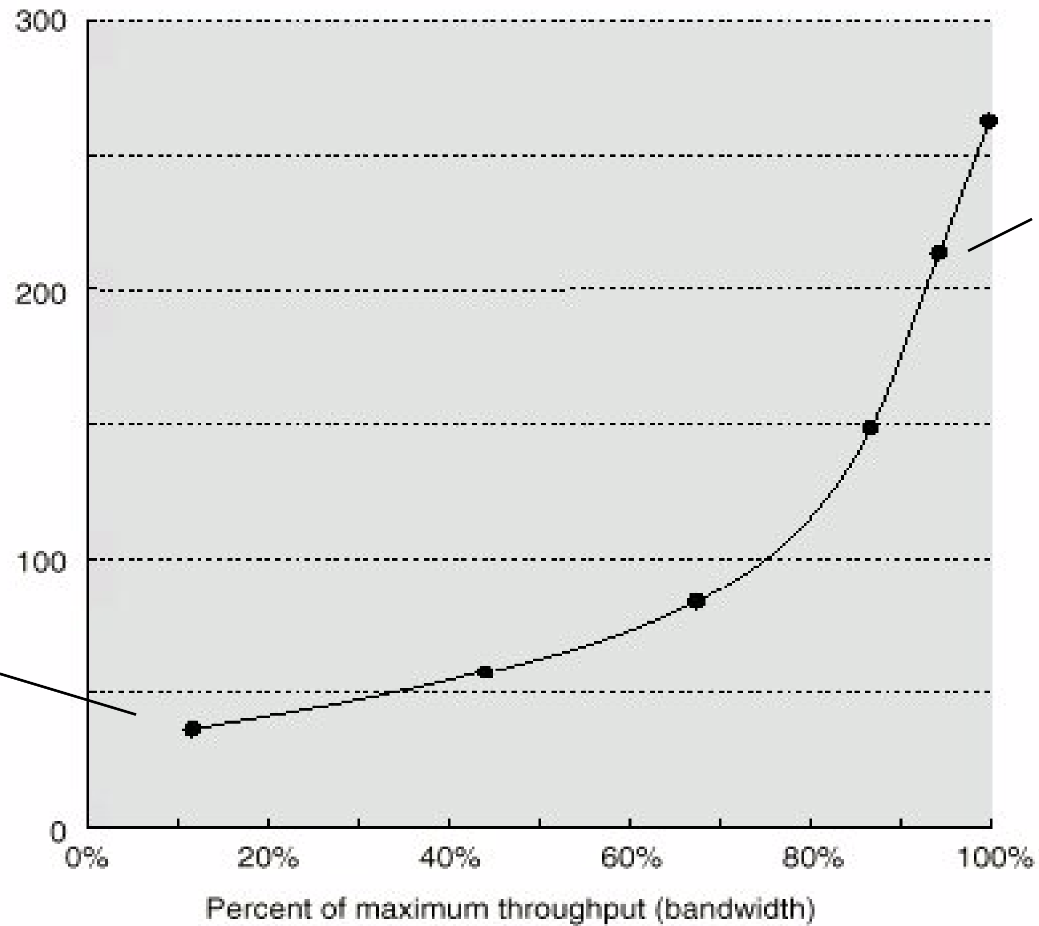
User or CPU



Response Time $=$ $\text{Time}_{\text{System}} = \text{Time}_{\text{Queue}} + \text{Time}_{\text{Server}}$

I/O device + controller

Throughput vs. Response Time

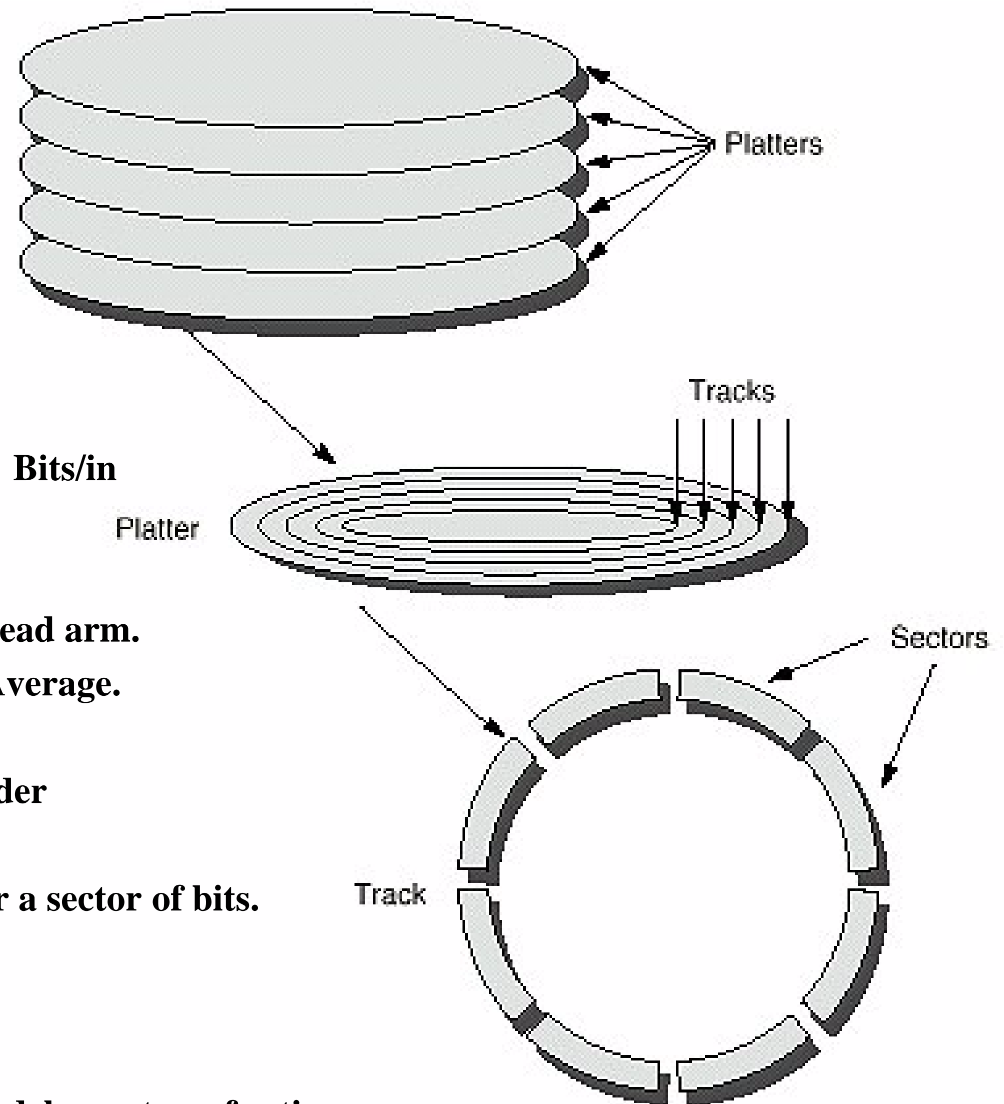


Queue full most of the time. More time in queue

Queue almost empty most of the time
Less time in queue

Magnetic Disks

Characteristics:



- Diameter: 2.5in - 5.25in
- Rotational speed: 3,600RPM-15,000 RPM
- Tracks per surface.
- Sectors per track: Outer tracks contain more sectors.
- Recording or Areal Density: Tracks/in X Bits/in
- Cost Per Megabyte.
- Seek Time: (2-12 ms)
  The time needed to move the read/write head arm.
  Reported values: Minimum, Maximum, Average.
- Rotation Latency or Delay: (2-8 ms)
  The time for the requested sector to be under the read/write head.
- Transfer time: The time needed to transfer a sector of bits.
- Type of controller/interface: SCSI, EIDE
- Disk Controller delay or time.
- Average time to access a sector of data =
  average seek time + average rotational delay + transfer time
  + disk controller overhead

  (ignoring queuing time)

Disk Performance Example

- **Given the following Disk Parameters:**
  - **Average seek time is 5 ms**
  - **Disk spins at 10,000 RPM**
  - **Transfer rate is 40 MB/sec**
- **Controller overhead is 0.1 ms**
- **Assume that the disk is idle, so no queuing delay exist.**
- **What is Average Disk read or write time for a 512-byte Sector?**

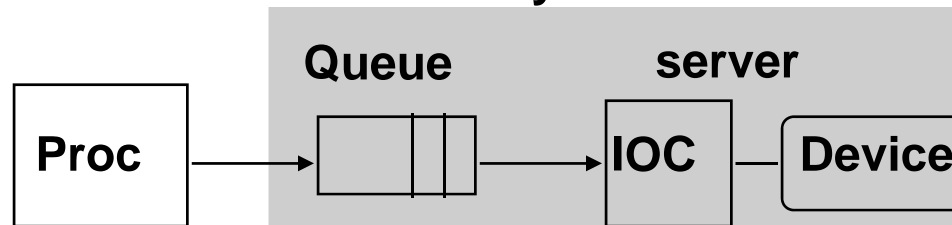
Ave. seek + ave. rot delay + transfer time + controller overhead

5 ms + 0.5/(10000 RPM/60) + 0.5 KB/40 MB/s + 0.1 ms

5 + 3 + 0.13 + 0.1 = 8.23 ms

This time is service time T_{ser} for this task used for queuing delay computation

I/O Performance & Little's Queuing Law



- **Given: An I/O system in equilibrium input rate is equal to output rate) and:**
  - T_{ser} : Average time to service a task = 1/Service rate
  - T_q : Average time per task in the queue
  - T_{sys} : Average time per task in the system, or the response time, the sum of T_{ser} and T_q thus $T_{sys} = T_{ser} + T_q$
  - r : Average number of arriving tasks/sec
  - L_{ser} : Average number of tasks in service.
  - L_q : Average length of queue
  - L_{sys} : Average number of tasks in the system, the sum of L_q and L_{ser}

- **Little's Law states:** $L_{sys} = r \times T_{sys}$ (applied to system)
  $L_q = r \times T_q$ (applied to queue)

- **Server utilization = u = r / Service rate = $r \times T_{ser}$**

u must be between 0 and 1 otherwise there would be more tasks arriving than could be serviced

A Little Queuing Theory: M/G/1 and M/M/1

- **Assumptions:**
  - System in equilibrium
  - Time between two successive arrivals in line are random
  - Server can start on next customer immediately after prior finishes
  - No limit to the queue: works First-In-First-Out
  - Afterward, all customers in line must complete; each avg T_{ser}
- Described "memoryless" or Markovian request arrival (M for C=1 exponentially random), General service distribution (no restrictions), 1 server: *M/G/1 queue*
- When Service times have C = 1, *M/M/1 queue*

$$T_q = T_{ser} \times u \,/\, (1 - u)$$

T_{ser} average time to service a task

r average number of arriving tasks/second

u server utilization (0..1): $u = r \times T_{ser}$

T_q average time/task in queue

T_{sys} Average time per task in the system $T_{sys} = T_q + T_{ser}$

L_q average length of queue: $L_q = r \times T_q$

L_{sys} Average number of tasks in the system $L_{sys} = r \times T_{sys}$

Multiple Server (Disk/Controller) I/O Modeling:

- I/O system with Markovian request arrival rate r
- A single queue serviced by m servers (disks + controllers) each with Markovian Service rate = $1/T_{ser}$

$$T_q = T_{ser} \times u / [m(1 - u)]$$

$$u = r \times T_{ser} / m$$

m	number of servers
T_{ser}	average time to service a task
u	server utilization (0..1): $u = r \times T_{ser} / m$
T_q	average time/task in queue
T_{sys}	Average time per task in the system $T_{sys} = T_q + T_{ser}$
L_q	average length of queue: $L_q = r \times T_q$
L_{sys}	Average number of tasks in the system $L_{sys} = r \times T_{sys}$

I/O Queuing Performance: An M/M/1 Example

- **A processor sends 10 x 8KB disk I/O requests per second, requests & service are exponentially distributed, average disk service time = 20 ms**
- **On average:**
  - **How utilized is the disk, u?**
  - **What is the average time spent in the queue, T_q?**
  - **What is the average response time for a disk request, T_{sys} ?**
  - **What is the number of requests in the queue L_q? In system, L_{sys}?**
- **We have:**

r	**average number of arriving requests/second = 10**
T_{ser}	**average time to service a request = 20 ms (0.02s)**

- **We obtain:**

u	**server utilization:** $u = r \times T_{ser}$ **= 10/s x .02s = 0.2 = 20%**
T_q	**average time/request in queue** $= T_{ser} \times u / (1 - u)$
	= 20 x 0.2/(1-0.2) = 20 x 0.25 = 5 ms (0 .005s)
T_{sys}	**average time/request in system:** $T_{sys} = T_q + T_{ser}$ **= 25 ms**
L_q	**average length of queue:** $L_q = r \times T_q$
	= 10/s x .005s = 0.05 requests in queue
L_{sys}	**average # tasks in system:** $L_{sys} = r \times T_{sys}$ **= 10/s x .025s = 0.25**

Example: Determining the System I/O Bottleneck (ignoring queuing delays)

- **Assume the following system components:**
  - 500 MIPS CPU
  - 16-byte wide memory system with 100 ns cycle time
  - 200 MB/sec I/O bus
  - 20 20 MB/sec SCSI-2 buses, with 1 ms controller overhead
  - 5 disks per SCSI bus: 8 ms seek, 7,200 RPMS, 6MB/sec
- **Other assumptions**
  - All devices used to 100% capacity, always have average values
  - Average I/O size is 16 KB
  - OS uses 10,000 CPU instructions for a disk I/O
  - Ignore disk/controller queuing delays.
- **What is the average IOPS? What is the average I/O bandwidth?**

Example: Determining the I/O Bottleneck
(ignoring queuing delays)

- **The performance of I/O systems is determined by the portion with the lowest I/O bandwidth**
  - **CPU : (500 MIPS)/(10,000 instr. per I/O) = 50,000 IOPS**
  - **Main Memory : (16 bytes)/(100 ns x 16 KB per I/O) = 10,000 IOPS**
  - **I/O bus: (200 MB/sec)/(16 KB per I/O) − 12,500 IOPS**
  - **SCSI-2: (20 buses)/((1 ms + (16 KB)/(20 MB/sec)) per I/O) = 11,120 IOPS**
  - **Disks: (100 disks)/((8 ms + 0.5/(7200 RPMS) + (16 KB)/(6 MB/sec)) per I/0) = 6,700 IOPS**
- **In this case, the disks limit the I/O performance to 6,700 IOPS**
- **The average I/O bandwidth is**
  - **6,700 IOPS x (16 KB/sec) = 107.2 MB/sec**

Example: Determining the I/O Bottleneck Accounting For I/O Queue Time (M/M/m queue)

Here m = 100

- **Assume the following system components:**
  - 500 MIPS CPU
  - 16-byte wide memory system with 100 ns cycle time
  - 200 MB/sec I/O bus
  - 20, 20 MB/sec SCSI-2 buses, with 1 ms controller overhead
  - 5 disks per SCSI bus: 8 ms seek, 7,200 RPMS, 6MB/sec
- **Other assumptions**
  - All devices used to 60% capacity (i.e maximum utilization allowed).
  - Treat the I/O system as an M/M/m queue.
  - Requests are assumed spread evenly on all disks.
  - Average I/O size is 16 KB
  - OS uses 10,000 CPU instructions for a disk I/O
- **What is the average IOPS? What is the average bandwidth?**
- **Average response time per IO operation?**

Example: Determining the I/O Bottleneck Accounting For I/O Queue Time (M/M/m queue)

- **The performance of I/O systems is still determined by the system component with the lowest I/O bandwidth**
  - **CPU : (500 MIPS)/(10,000 instr. per I/O) x .6 = 30,000 IOPS**

    CPU time per I/O = 10,000 / 500,000,000 = .02 ms
  - **Main Memory : (16 bytes)/(100 ns x 16 KB per I/O) x .6 = 6,000 IOPS**

    Memory time per I/O = 1/10,000 = .1ms
  - **I/O bus: (200 MB/sec)/(16 KB per I/O) x .6 = 12,500 IOPS**
  - **SCSI-2: (20 buses)/((1 ms + (16 KB)/(20 MB/sec)) per I/O) = 7,500 IOPS**

    SCSI bus time per I/O = 1ms + 16/20 ms = 1.8ms
  - **Disks: (100 disks)/((8 ms + 0.5/(7200 RPMS) + (16 KB)/(6 MB/sec)) per I/0) x .6 = 6,700 x .6 = 4020 IOPS**

    T_{ser} = (8 ms + 0.5/(7200 RPMS) + (16 KB)/(6 MB/sec) = 8+4.2+2.7 = 14.9ms
- **The disks limit the I/O performance to r = 4020 IOPS**
- **The average I/O bandwidth is 4020 IOPS x (16 KB/sec) = 64.3 MB/sec**
- ***$T_q = T_{ser} \times u /[m(1 - u)] = 14.9ms \times .6 / [100 \times .4] = .22\ ms$***
- ***Response Time = Tser + Tq+ Tcpu + Tmemory + Tscsi =***

  14.9 + .22 + .02 + .1 + 1.8 = 17.04 ms