

CS/ECE 5381/7381
Computer Architecture
Spring 2023

Dr. Manikas

Computer Science

Lecture 11: Mar 2, 2023

Assignments

- Project 3 (7381 only)
 - Due TODAY Thur., Mar. 2 (11:59 pm)
- Quiz 5 – due Sat., Mar. 4 (11:59 pm)
 - Covers concepts from Modules 5 and 6

Quiz 5 Details

- The quiz is open book and open notes.
- You are allowed 90 minutes to take this quiz.
- You are allowed 2 attempts to take this quiz - your highest score will be kept.
 - Note that some questions (e.g., fill in the blank) will need to be graded manually
- Quiz answers will be made available 24 hours after the quiz due date.

Instruction-Level Parallelism (ILP) and Its Exploitation

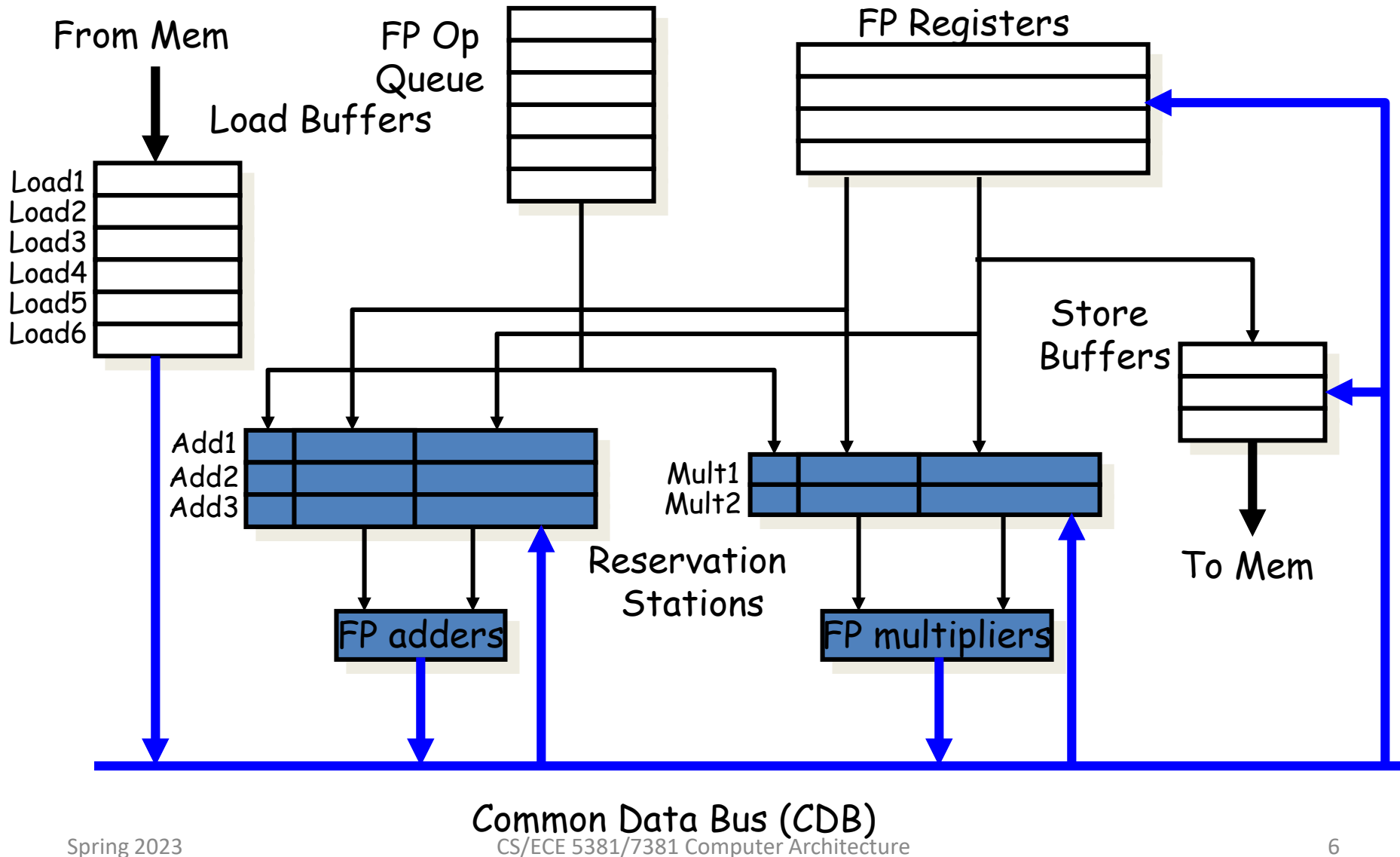
(Chapter 3, Hennessy and Patterson)

Note: some course slides adopted
from publisher-provided material

Outline

- 3.1 ILP Background
- 3.2 Basic Compiler Techniques for ILP
- 3.3 Branch Prediction
- 3.4 Data Hazards and Dynamic Scheduling
- 3.5 Dynamic Scheduling Algorithm
- 3.6 Hardware-Based Speculation

Tomasulo Organization



Reservation Station Components

Op: Operation to perform in the unit (e.g., + or −)

Vj, Vk: **Value** of Source operands

- Store buffers has V field, result to be stored

Qj, Qk: Reservation stations producing source registers (value to be written)

- Note: $Q_j, Q_k = 0 \Rightarrow$ ready
- Store buffers only have Q_i for RS producing result

Busy: Indicates reservation station or FU is busy

Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Three Stages of Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue

If reservation station free (no structural hazard),
control issues instr & sends operands (renames registers).

2. Execute—operate on operands (EX)

When both operands ready then execute;
if not ready, watch Common Data Bus for result

3. Write result—finish execution (WB)

Write on Common Data Bus to all awaiting units;
mark reservation station available

Assume we have the following MIPS code – how is this handled using Tomasulo's algorithm?

	Latency
L.D F6, 34(R2)	2
L.D F2, 45(R3)	2
MUL.D F0, F2, F4	10
SUB.D F8, F6, F2	2
DIV.D F10, F0, F6	40
ADD.D F6, F8, F2	2

Instruction status table:

Instruction		j	k	Issue	Exec comp	Write result
L.D	F6	34	R2	1	3	4
L.D	F2	45	R3	2	4	5
MUL.D	F0	F2	F4	3	15	16
SUB.D	F8	F6	F2	4	7	8
DIV.D	F10	F0	F6	5	56	57
ADD.D	F6	F8	F2	6	10	11

Load buffers:

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

All instructions completed:
In-order issue, out-of-order execution and out-of-order completion

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	F14
57	FU	F2 * F4	M[45+R3]		F8 + F2	F6 - F2	F0 / F6		

Outline

- 3.1 ILP Background
- 3.2 Basic Compiler Techniques for ILP
- 3.3 Branch Prediction
- 3.4 Data Hazards and Dynamic Scheduling
- 3.5 Dynamic Scheduling Algorithm
- 3.6 Hardware-Based Speculation

Hardware-Based Speculation

- Tomasulo's algorithm handles data dependencies
 - Extra hardware for temporary data storage
- What about control dependencies (branching)?
 - Use *hardware-based speculation* to handle dynamic branch prediction

Adding Speculation to Tomasulo

- Must separate execution from allowing instruction to finish or “commit”
- This additional step called **instruction commit**
- When an instruction is no longer speculative, allow it to update the register file or memory

Adding Speculation to Tomasulo

- Requires additional set of buffers to hold results of instructions that have finished execution but have not committed
- This **reorder buffer (ROB)** is also used to pass results among instructions that may be speculated

Reorder Buffer (ROB)

- In Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will find result in the register file
- With speculation, the register file is not updated until the instruction commits
 - (we know definitively that the instruction should execute)

Reorder Buffer (ROB)

- Thus, the ROB supplies operands in interval **between** completion of instruction execution and instruction commit
 - ROB is a source of operands for instructions, just as reservation stations (RS) provide operands in Tomasulo's algorithm
 - ROB extends architecture registers like RS

Speculating with Tomasulo

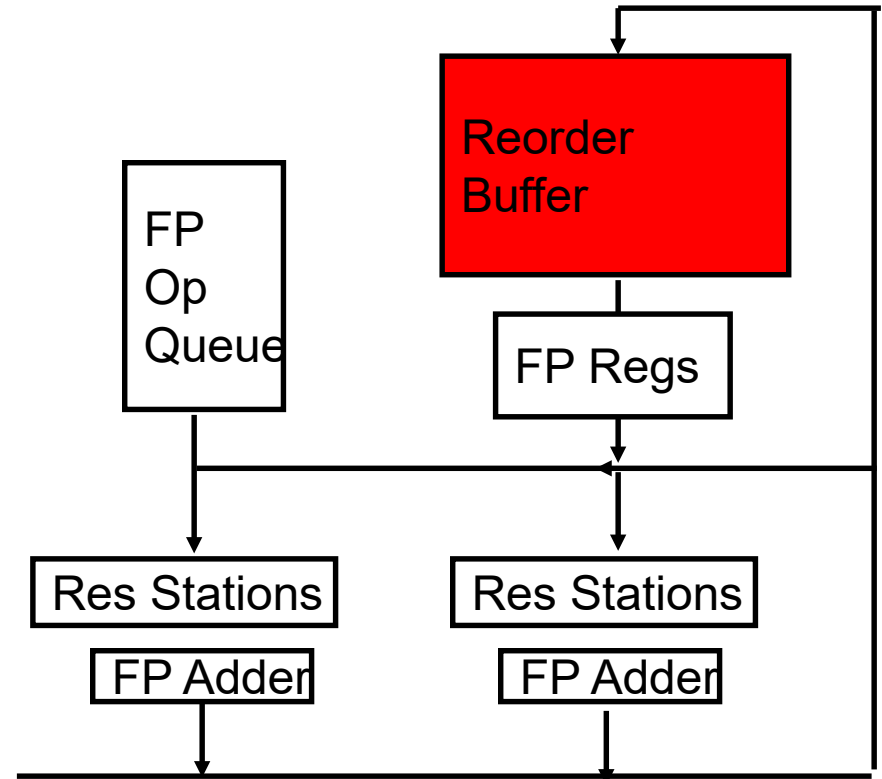
- Key ideas:
 - *separate execution from completion*: allow instructions to execute speculatively but *do not let instructions update registers or memory until they are no longer speculative*
 - therefore, add a final step – after an instruction is no longer speculative – when it is allowed to make register and memory updates, called *instruction commit*

Speculating with Tomasulo

- Key ideas:
 - *allow instructions to **execute** out of order but force them to **commit** in order*
 - add a hardware buffer, called the *reorder buffer (ROB)*, with registers to hold the result of an instruction *between execution completion and commit*

HW support for branching

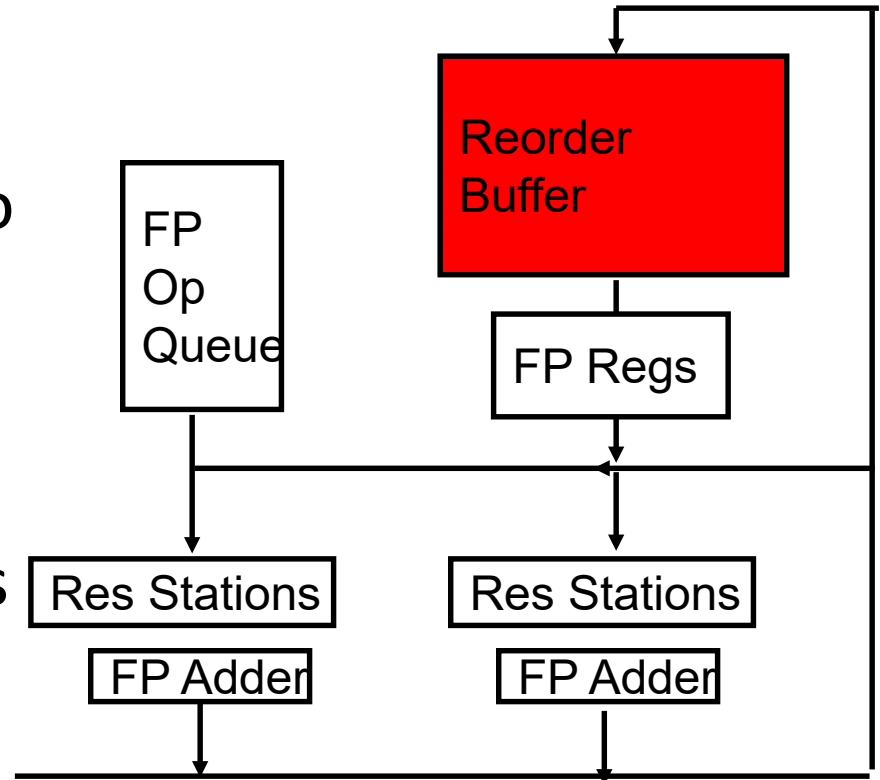
- Need HW buffer for results of uncommitted instructions: *reorder buffer*
 - Use reorder buffer number instead of reservation station when execution completes
 - Supplies operands between execution completion & commit



HW support for branching

Reorder buffer (ROB)

- Instructions commit
- Once an instruction commits, result is put into register
- As a result, easy to undo speculated instructions on mispredicted branches



Speculative Tomasulo Example

LD	F0	10	R2
ADDD	F10	F4	F0
DIVD	F2	F10	F6
BNEZ	F2	Exit	
LD	F4	0	R3
ADDD	F0	F4	F9
SD	F4	0	R3

...

Exit:

Branch Taken

LD	F0	10	R2
ADDD	F10	F4	F0
DIVD	F2	F10	F6
BNEZ	F2	Exit	

LD	F4	0	R3
ADDD	F0	F4	F9
SD	F4	0	R3


...

Exit:

Skip
over

Branch Not Taken

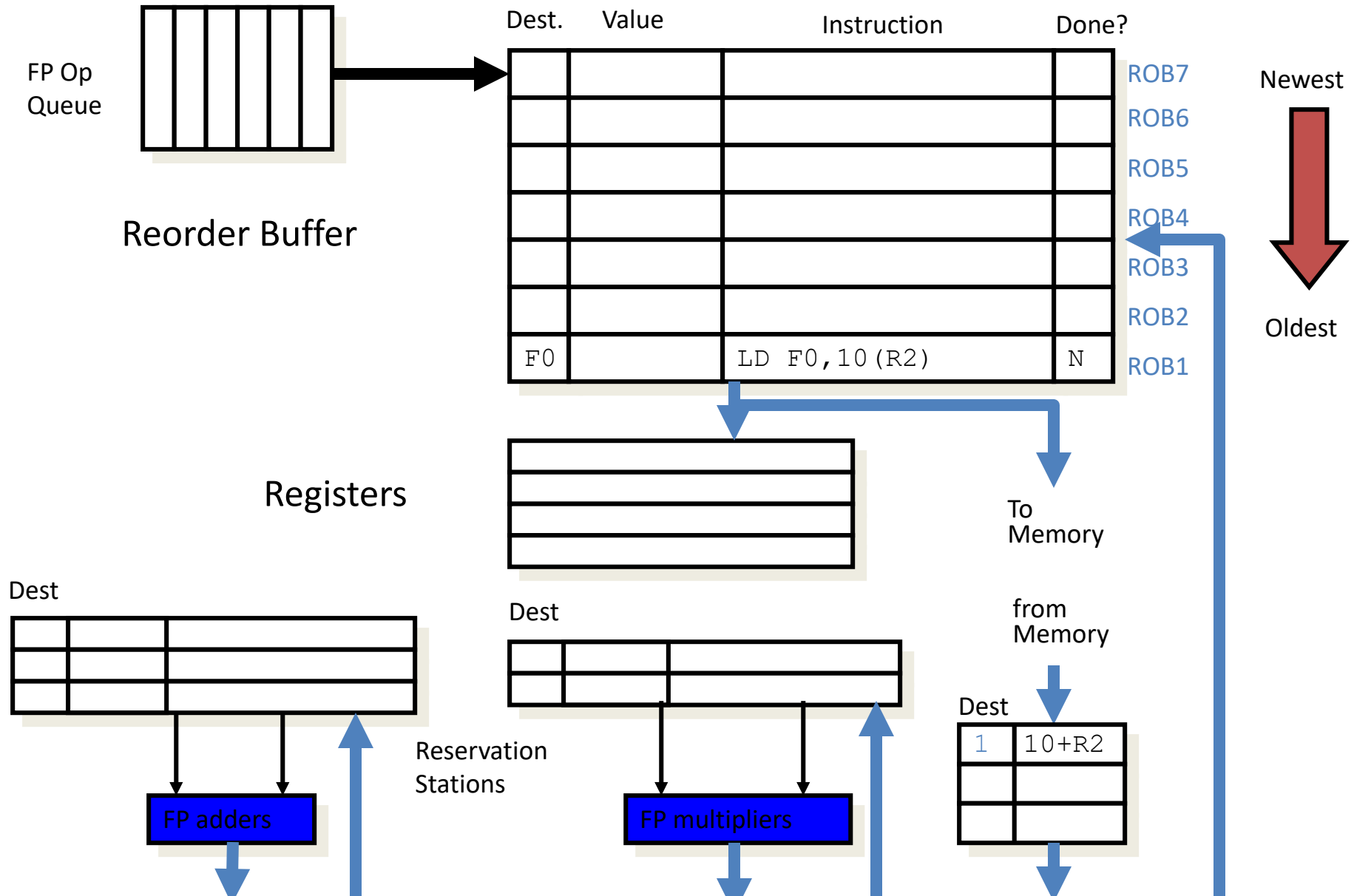
LD	F0	10	R2	
ADDD	F10	F4	F0	
DIVD	F2	F10		F6
BNEZ	F2	Exit		
LD	F4	0	R3	
ADDD	F0	F4	F9	
SD	F4	0	R3	



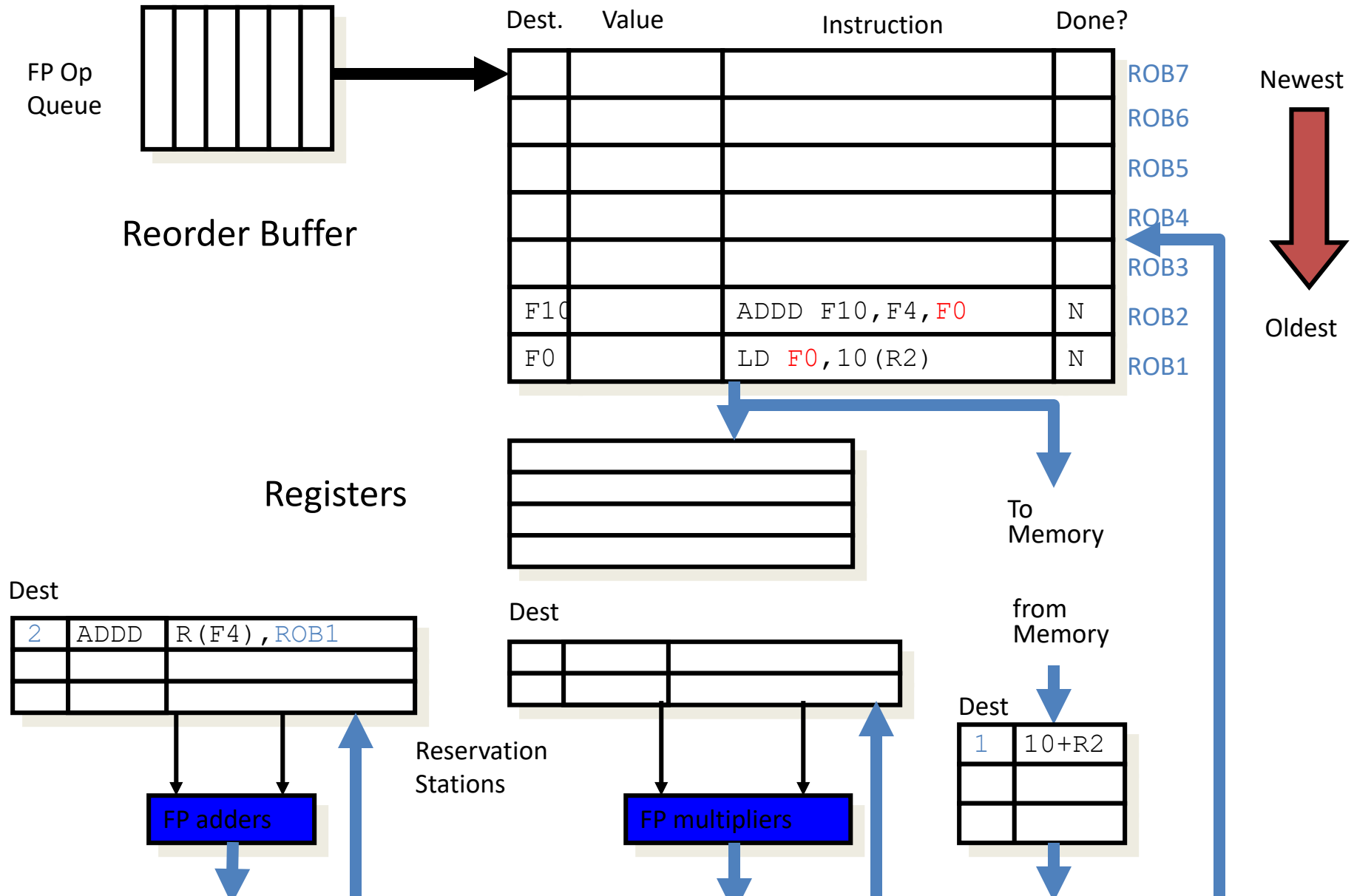
...

Exit:

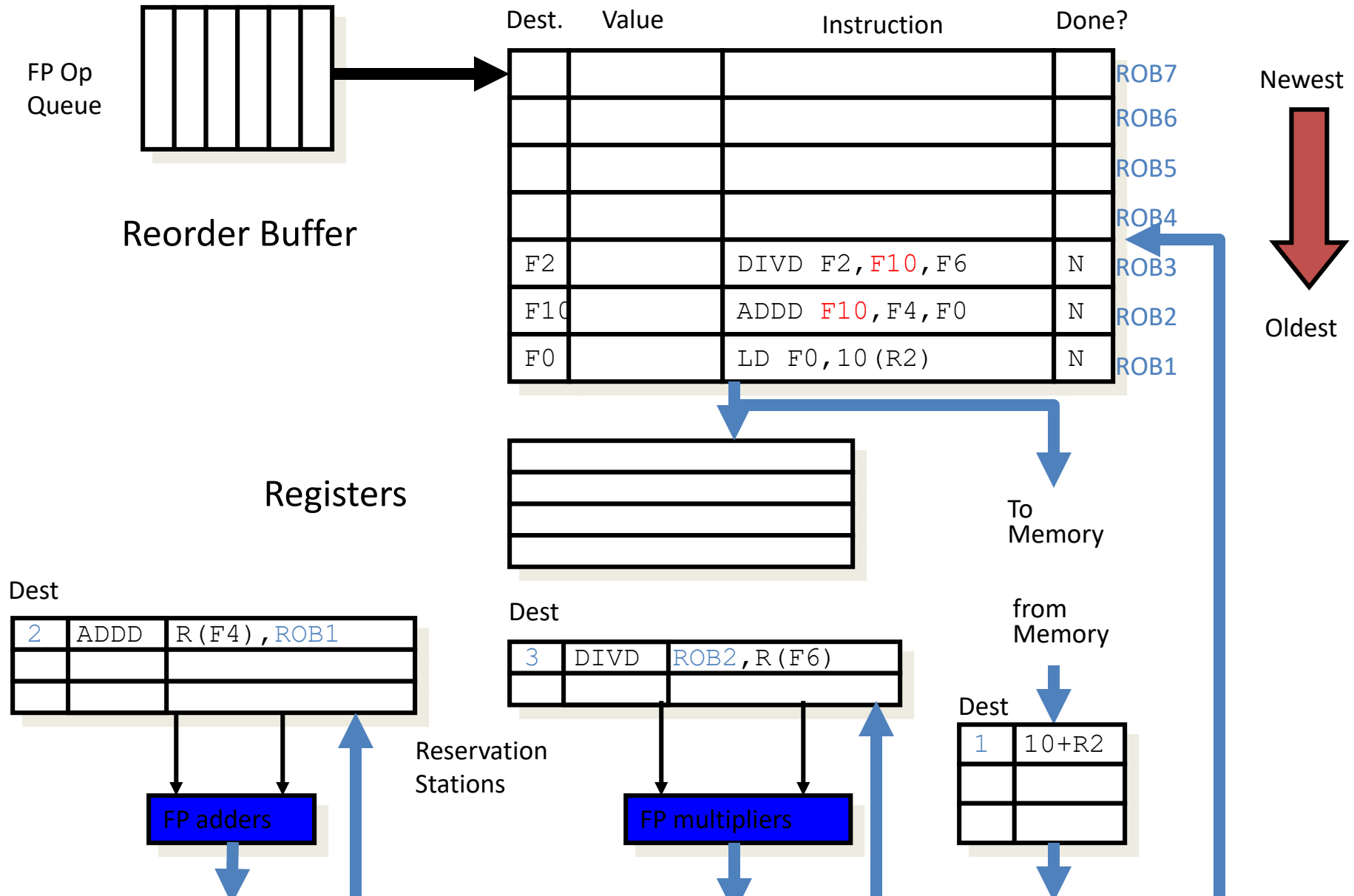
Tomasulo With Reorder buffer:



Tomasulo With Reorder buffer:

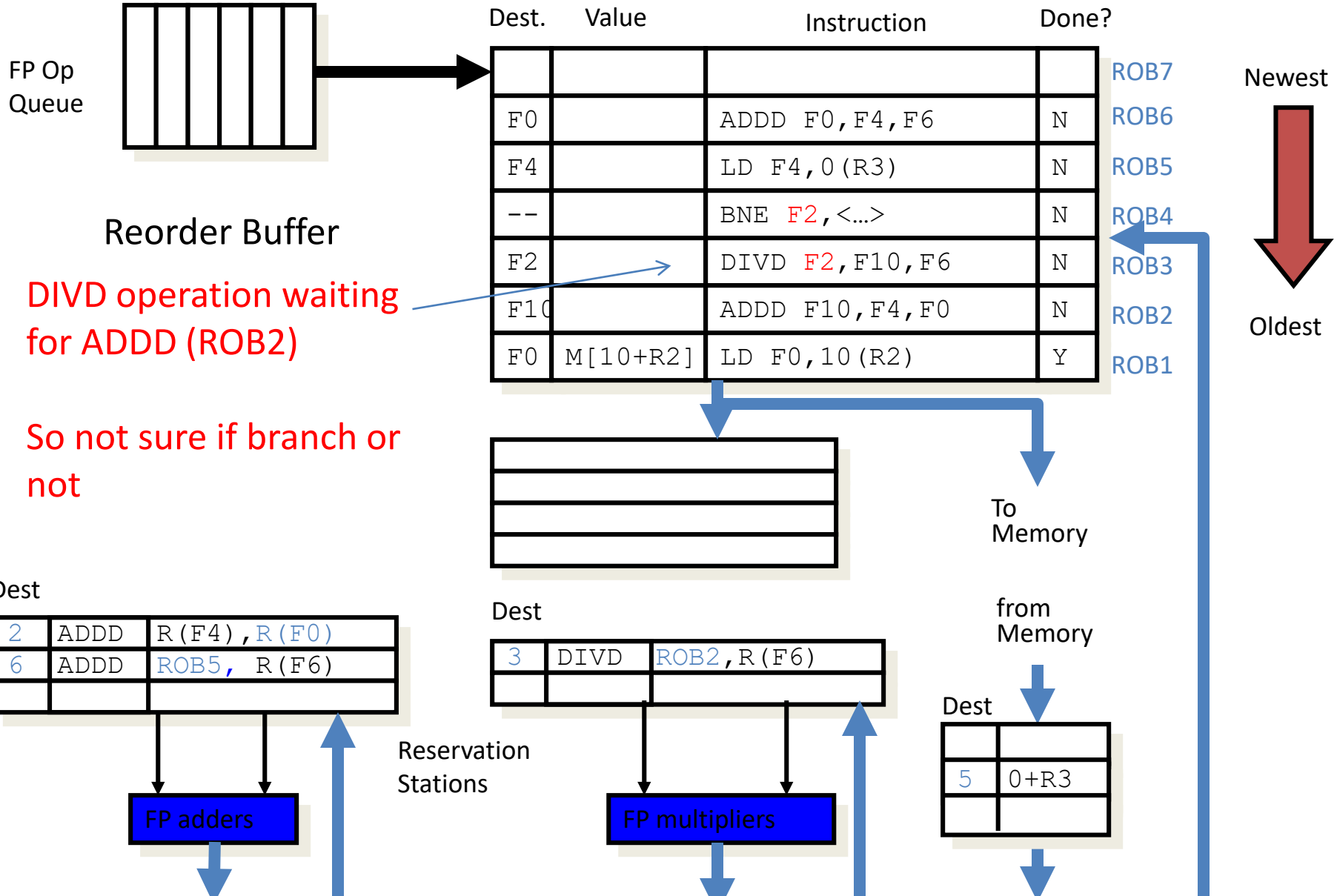


Tomasulo With Reorder buffer:



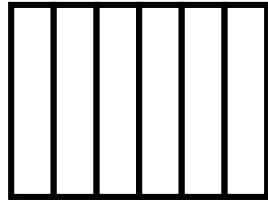
(skip some cycles)

Tomasulo With Reorder buffer:



Tomasulo With Reorder buffer:

FP Op Queue



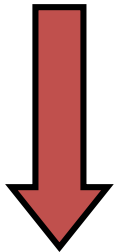
Reorder Buffer

Dest. Value Instruction Done?

--	ROB5	SD F4, 0 (R3)	N
F0		ADDD F0, F4, F6	N
F4		LD F4, 0 (R3)	N
--		BNE F2, <...>	N
F2		DIVD F2, F10, F6	N
F10		ADDD F10, F4, F0	N
F0	M[10+R2]	LD F0, 10 (R2)	Y

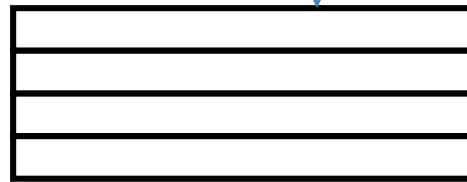
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1

Newest



Oldest

In the meantime, keep processing instructions in order – assume branch NOT taken



To Memory

Dest

2	ADDD	R (F4), R (F0)
6	ADDD	ROB5, R (F6)

FP adders

Reservation Stations

Dest

3	DIVD	ROB2, R (F6)

FP multipliers

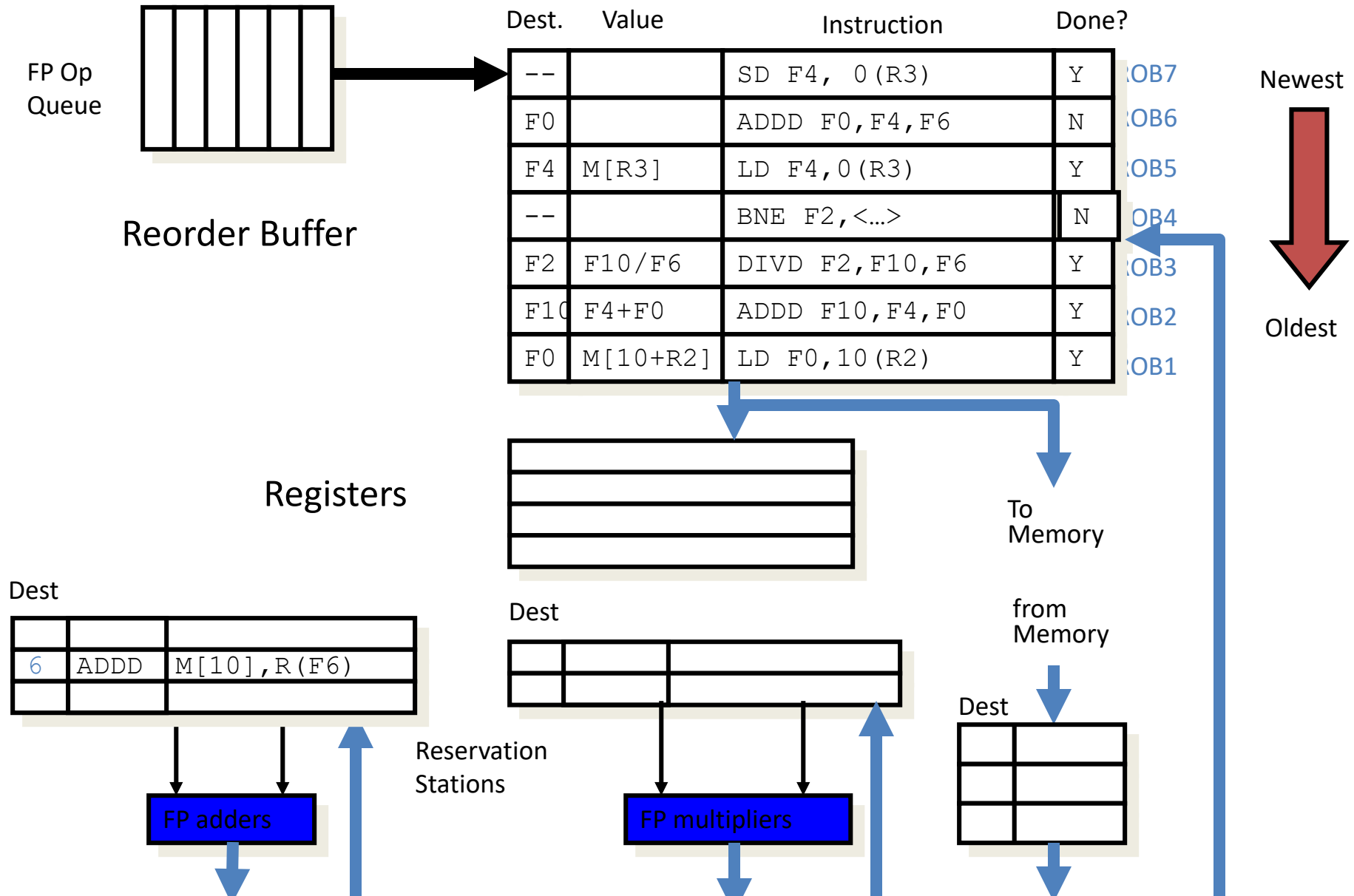
from Memory

Dest

5	0+R3

(skip some more cycles)

Tomasulo With Reorder buffer:



Notes

- We predicted branch NOT taken
- If a branch is **mispredicted**, recovery is done by flushing the ROB of all entries that appear after the mispredicted branch
 - entries before the branch are allowed to continue
 - restart the fetch at the correct branch successor
- When an instruction commits or is flushed from the ROB then the corresponding slots become available for subsequent instructions

Review of Memory Hierarchy

(Appendix B, Hennessy and Patterson)

Note: some course slides adopted from
publisher-provided material

Notes

- This section will be a review of basic memory hierarchy principles covered in CSE 4381 or equivalent course

Outline

- B.1 Introduction
- B.2 Cache Performance
- B.3 Basic Cache Optimizations
- B.4 Virtual Memory

Memory

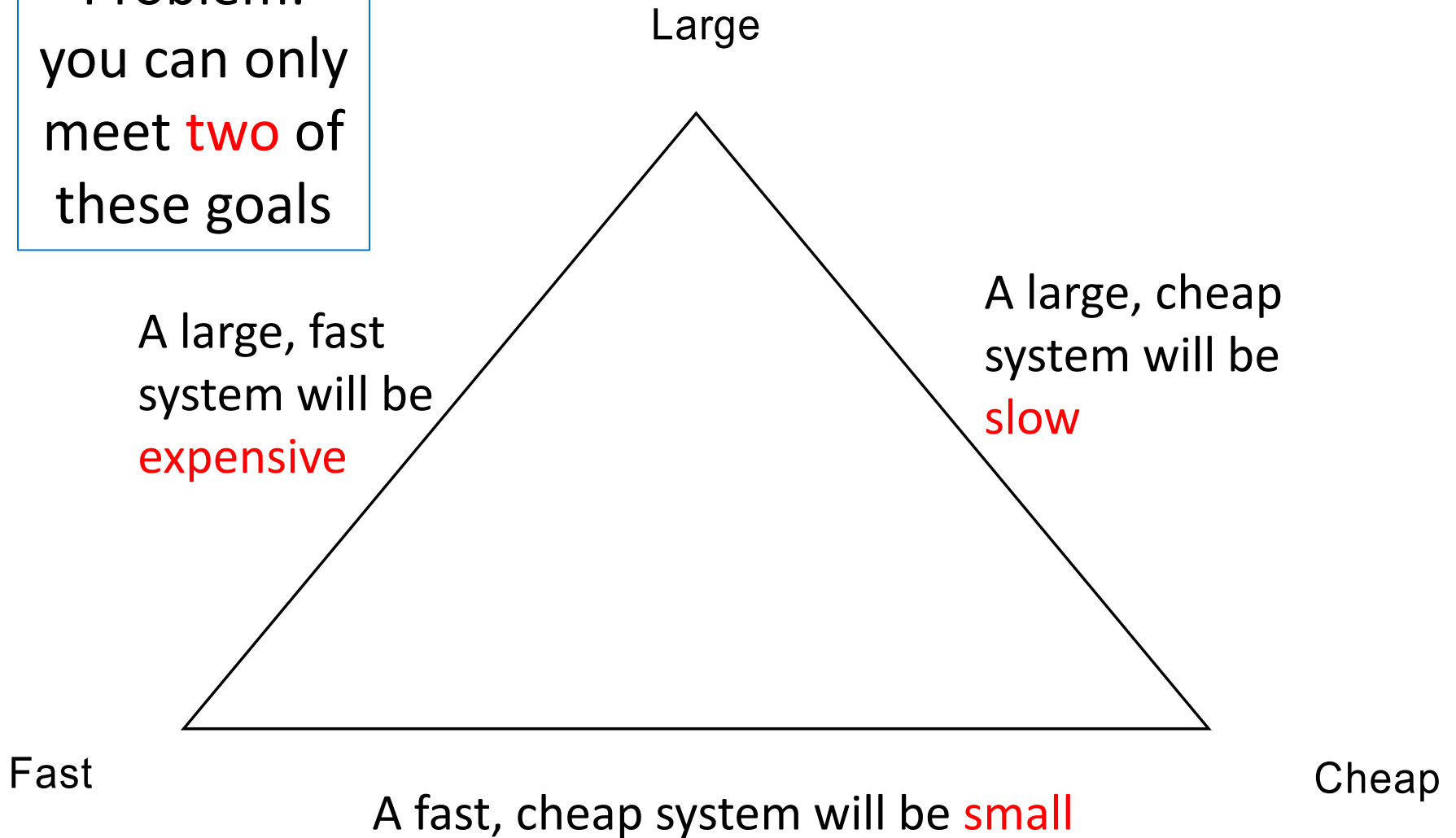
- data and program storage
- RAM
- ROM
- Disk Drives (magnetic or solid-state)



Goals of Memory Design

- **Large** – to store all program instructions and data
- **Fast** – so that CPU can quickly access instructions and data
- **Cheap** – low cost per storage bit

Problem:
you can only
meet **two** of
these goals



Memory Hierarchy

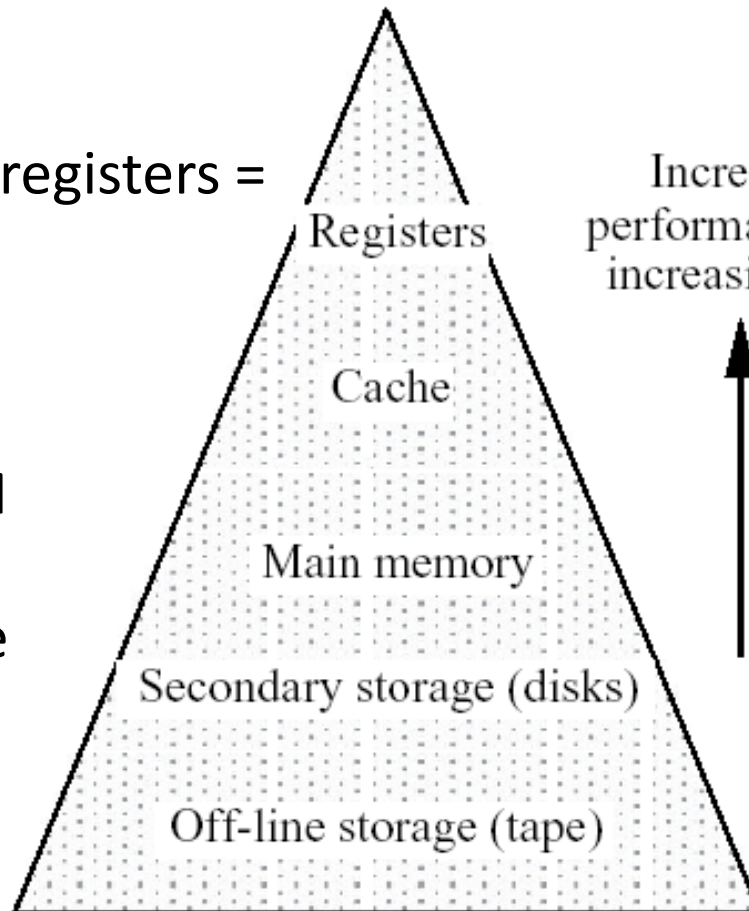
Fast and expensive

32 32-bit (4 B) registers =
128 B

8 GB RAM

1 TB hard drive
(disk or SSD)

15 TB tape reel



Slow and inexpensive

Make this
small to
reduce costs

This can be
large since
cheap

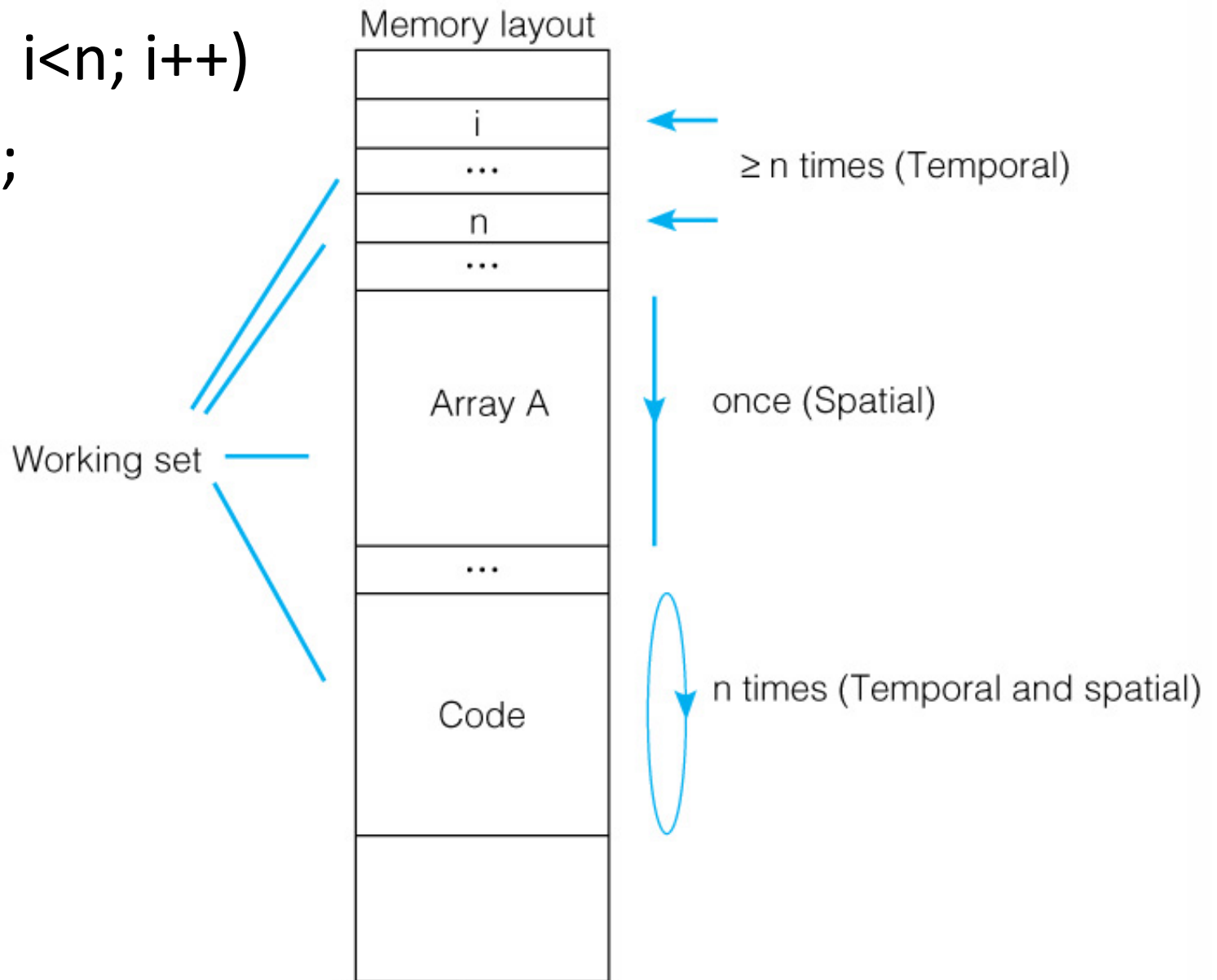
Memory Hierarchy

- Ideal memory
 - Fast access time, large capacity, low cost
- We don't have this in any *one* component
- But we can use a *combination* of components to “appear” as “ideal memory” to CPU
 - This combination is called a *memory hierarchy*
 - Use the “principle of locality”

Principle of Locality

- Programs access a small proportion of their address space at any time
- **Temporal locality**
 - Items accessed recently are likely to be accessed again soon
 - e.g., instructions in a loop, instruction variables
- **Spatial locality**
 - Items near those accessed recently are likely to be accessed soon
 - E.g., sequential instruction access, array data

```
for ((i=0); i<n; i++)  
  A[i] = 0;
```

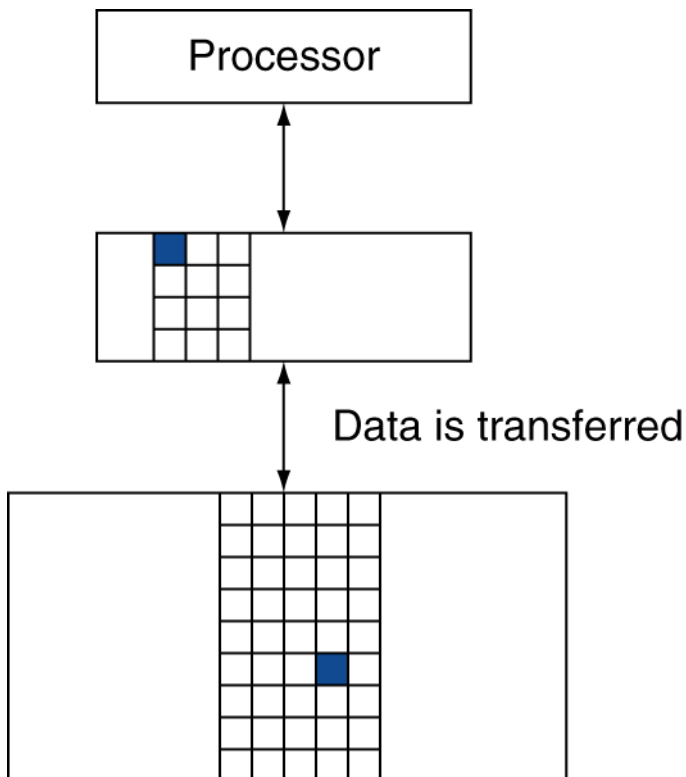


Taking Advantage of Locality

- Memory hierarchy
- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
 - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
 - Cache memory attached to CPU

Memory Hierarchy Levels

- Block (aka line): unit of copying
 - May be multiple words
- If accessed data is present in upper level
 - Hit: access satisfied by upper level
 - Hit ratio: hits/accesses
- If accessed data is absent
 - Miss: block copied from lower level
 - Time taken: miss penalty
 - Miss ratio: misses/accesses
 $= 1 - \text{hit ratio}$
 - Then accessed data supplied from upper level



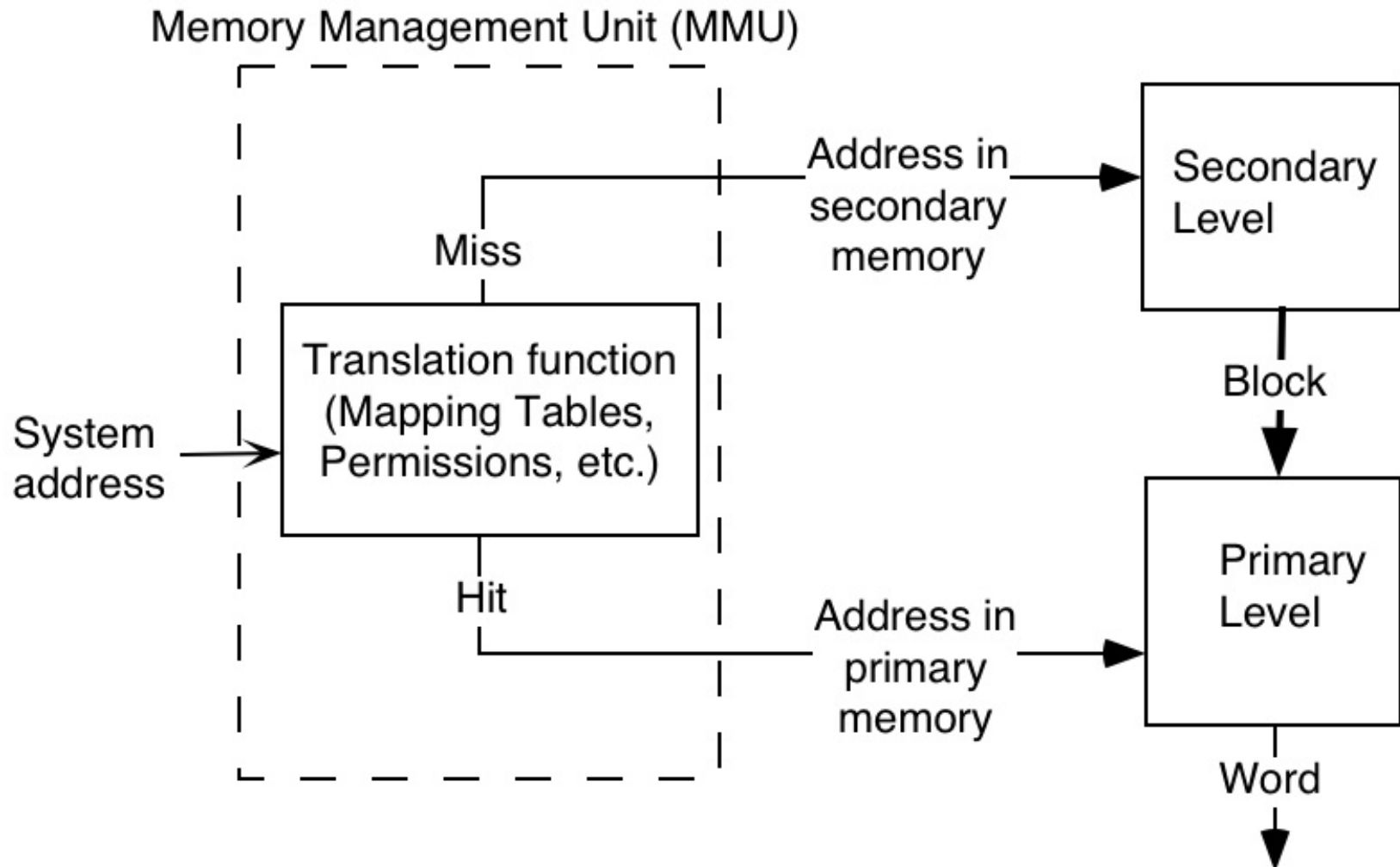
Memory Hierarchy: Terminology

- **Hit**: data appears in some block in the upper (primary) level
 - **Hit Rate**: the fraction of memory access found in the upper level
 - **Hit Time**: Time to access the upper level which consists of:
Primary level access time + Time to determine hit/miss

Memory Hierarchy: Terminology (cont)

- **Miss**: data needs to be retrieved from a block in the lower (secondary) level
 - **Miss Rate** = $1 - (\text{Hit Rate})$
 - **Miss Penalty**: Time to replace a block in the upper level +
Time to deliver the block the processor
- **Hit Time** \ll **Miss Penalty**

Addressing and Accessing a 2-Level Hierarchy



Hits and Misses

- **Miss ratio** = #misses / total # of refs in the same program-execution period
- **Hit ratio** = 1 - miss ratio
 - Let t_p = primary memory access time,
 - t_s = secondary memory access time,
 - h = hit ratio
- Then avg access time for two-level memory hierarchy $t_a = ht_p + (1-h)t_s$
- The qty $t_s > t_p$, so design goal is to minimize miss ratio

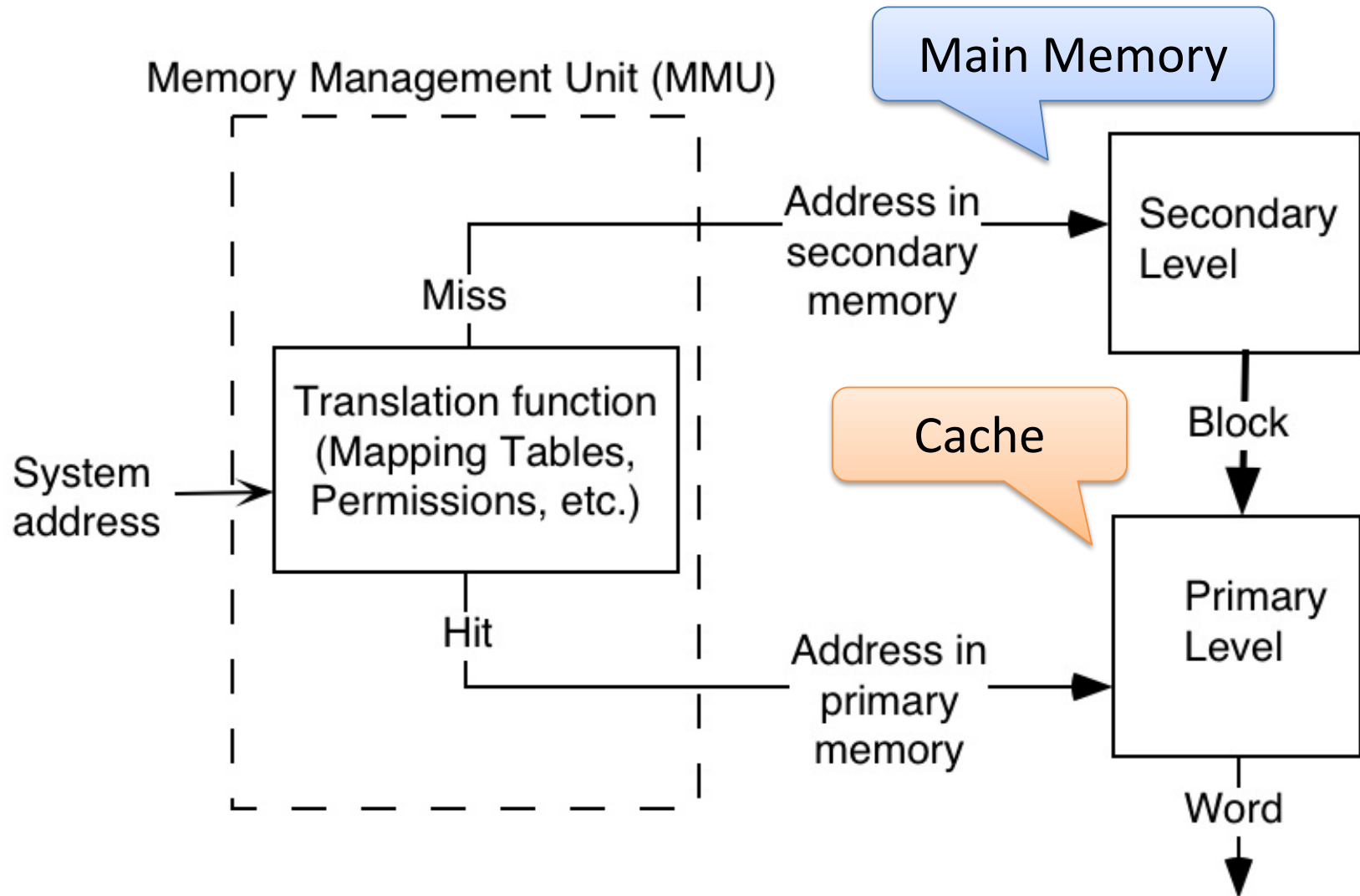
Example B.1-1

- Assume that we have a two-level memory hierarchy with the following memory access times:
 - Primary = 10 ns
 - Secondary = 100 ns (10 times slower)
- What is the **average access time** for this memory hierarchy if the hit rate is **90% (0.9)**?
- What is the **average access time** for this memory hierarchy if the hit rate is **75% (0.75)**?

Cache

- A fast memory (RAM) used to hold active part of program and data
- Speeds up operation of active program
- Often incorporated into CPU
- Operates on *Principle of Locality* (recall from earlier)
 - Part of *2-level memory hierarchy*

Addressing and Accessing a 2-Level Hierarchy



Interpreting Memory Addresses

- Memory size
 - $2^{10} = 1 \text{ K (kilo)}$
 - $2^{20} = 1 \text{ M (mega)}$
 - $2^{30} = 1 \text{ G (giga)}$
 - $2^{40} = 1 \text{ T (tera)}$
 - $2^{50} = 1 \text{ P (peta)}$
- $1 \text{ B (byte)} = 8 \text{ b (bits)}$