Instruction-level parallelism: Tomasulo - Reorder Buffer

Dr. Tao Xie

These slides are adapted from notes by Dr. David Patterson (UCB)

Why can Tomasulo overlap iterations of loops?

- Register renaming
  - Multiple iterations use different physical destinations for registers
- Reservation stations
  - Permit instruction issue to advance past integer control flow operations
  - Also buffer old values of registers - totally avoiding the WAR stall that we saw in the scoreboard.
- Other perspective: Tomasulo building data flow dependency graph on the fly.

What about Precise Interrupts?

- Interrupts would be imprecise in Tomasulo.
- Tomasulo had:

  In-order issue, out-of-order execution, and out-of-order completion

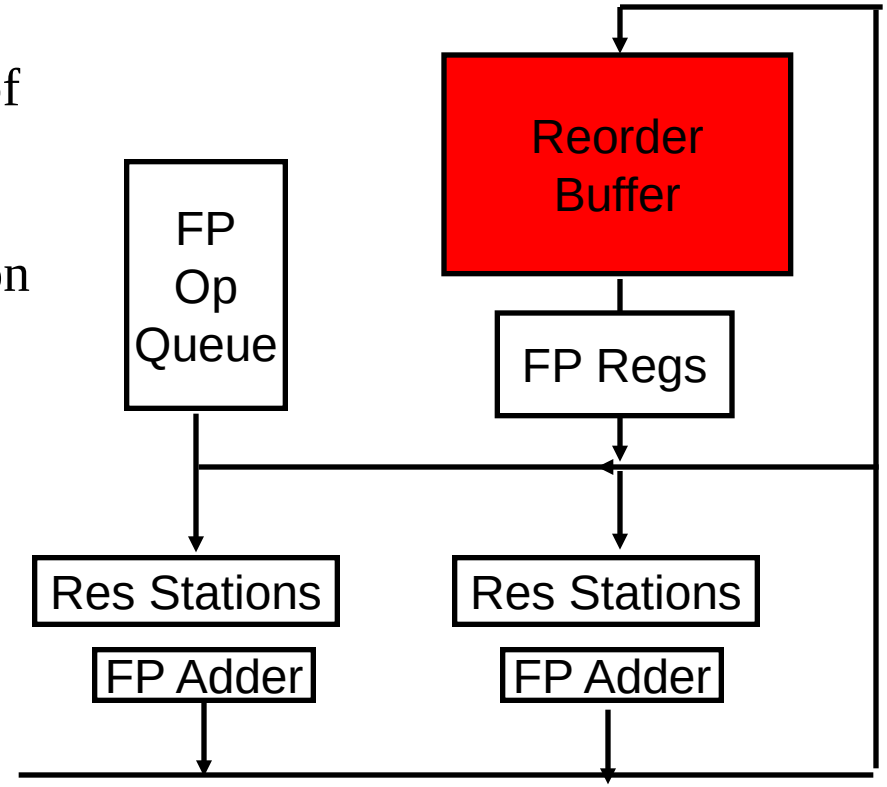
- Need to "fix" the out-of-order completion aspect so that we can find precise breakpoint in instruction stream.

Speculating with Tomasulo

- Modern processors such as PowerPC 603/604, MIPS R10000, Intel Pentium II/III/4, Alpha 21264 *extend Tomasulo's approach to support speculation*
- 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*
  - *allow instructions to execute and complete 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 completion and commit*

HW support for precise interrupts

- 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 complete & commit
  - (Reorder buffer can be operand source => more registers like RS)
  - Instructions commit
  - Once instruction commits, result is put into register
  - As a result, easy to undo speculated instructions on mispredicted branches or exceptions



ROB Data Structure

ROB entry fields

- **Instruction type**: branch, store, register operation (i.e., ALU or load)
- **State**: indicates if instruction has completed and value is ready
- **Destination**: where result is to be written – register number for register operation (i.e. ALU or load), memory address for store
  - branch has no destination result
- **Value**: holds the value of instruction result till time to commit

Additional reservation station field

- **Destination**: Corresponding ROB entry number

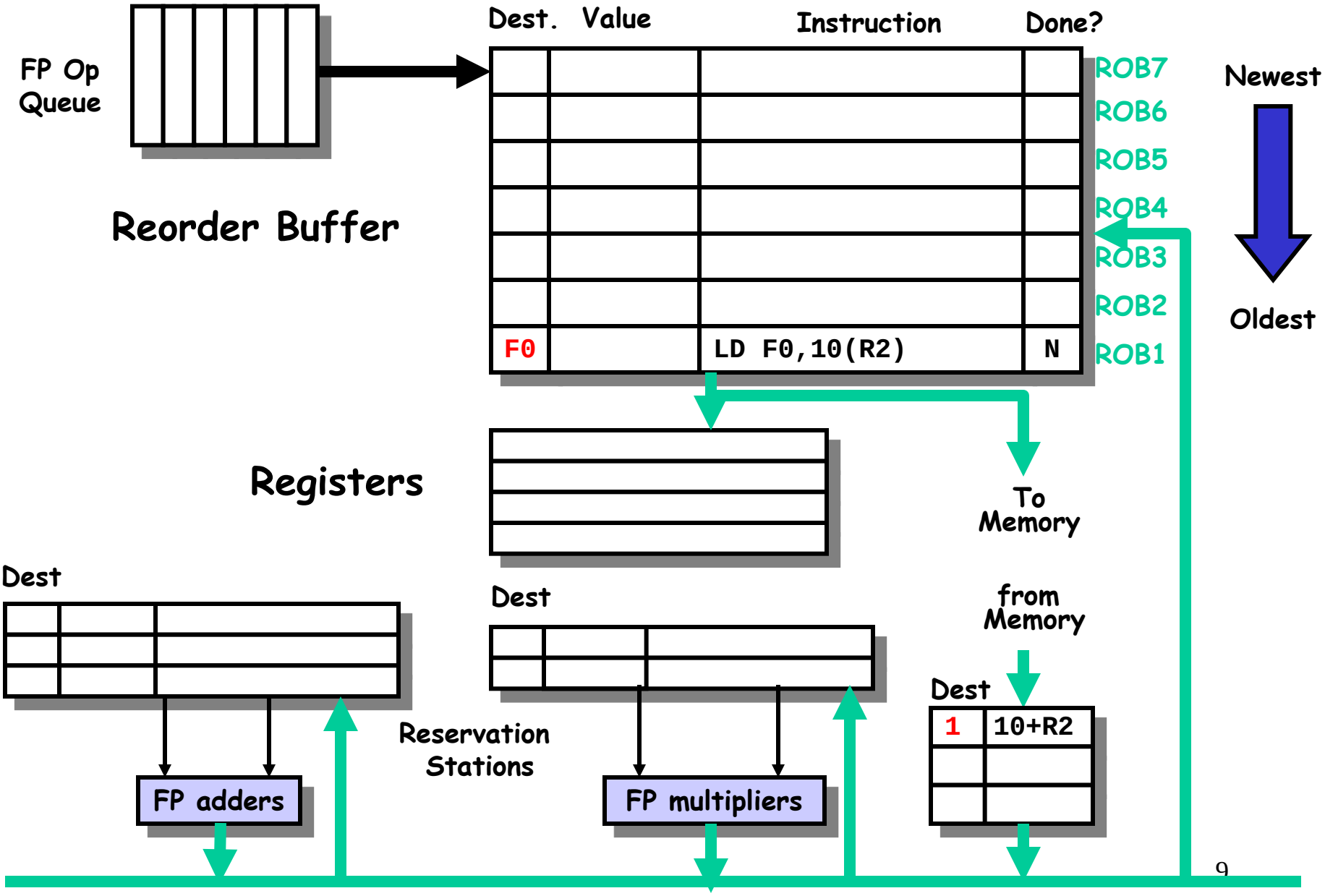
Four Steps of Speculative Tomasulo Algorithm

- Issue—get instruction from FP Op Queue
  - If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called "dispatch")
- Execution—operate on operands (EX)
  - When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called "issue")
- Write result—finish execution (WB)
  - Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.
- Commit—update register with reorder result
  - When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called "graduation")

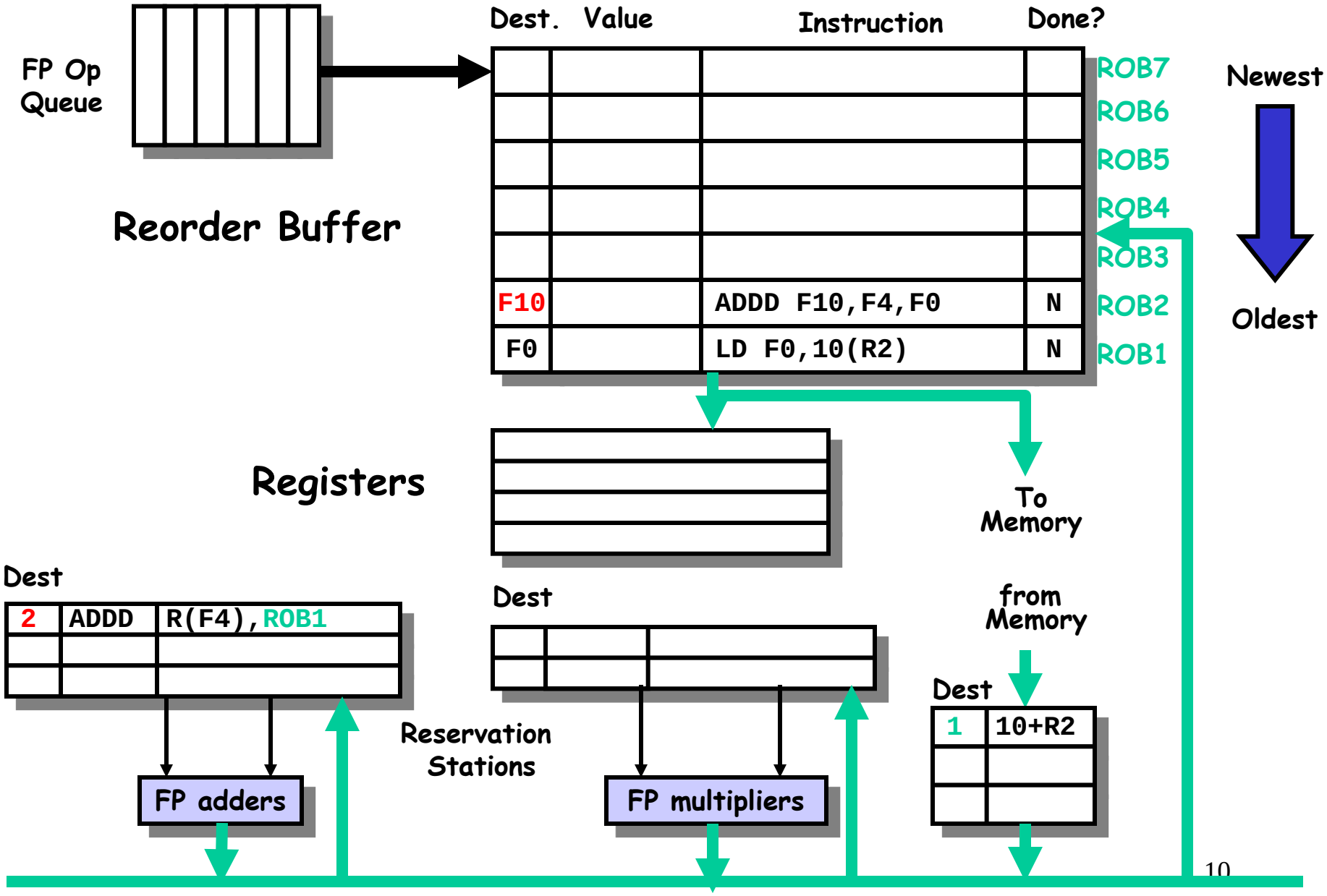
Speculative Tomasulo Example

```
      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:
```

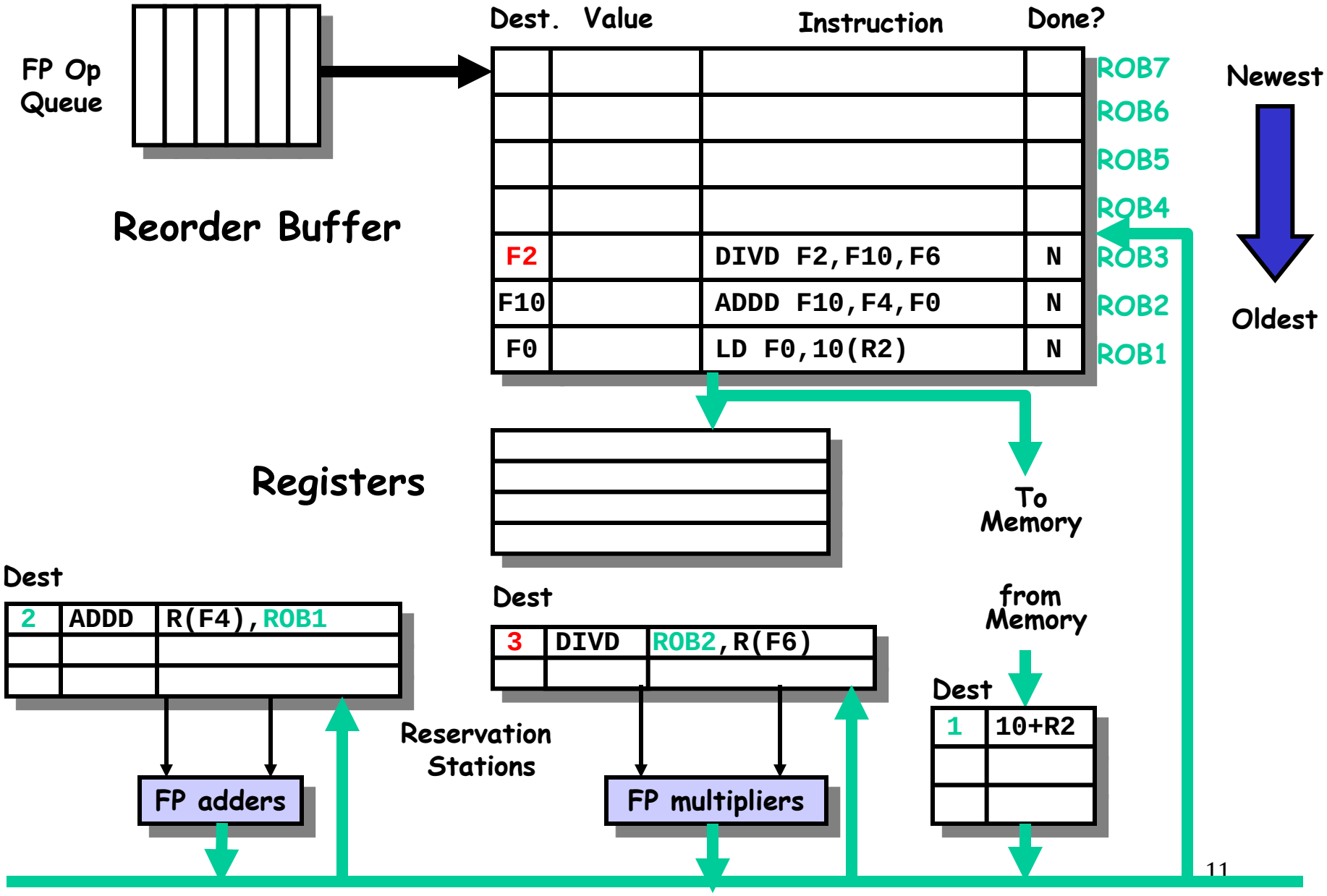
Tomasulo with Reorder buffer



Tomasulo with Reorder buffer

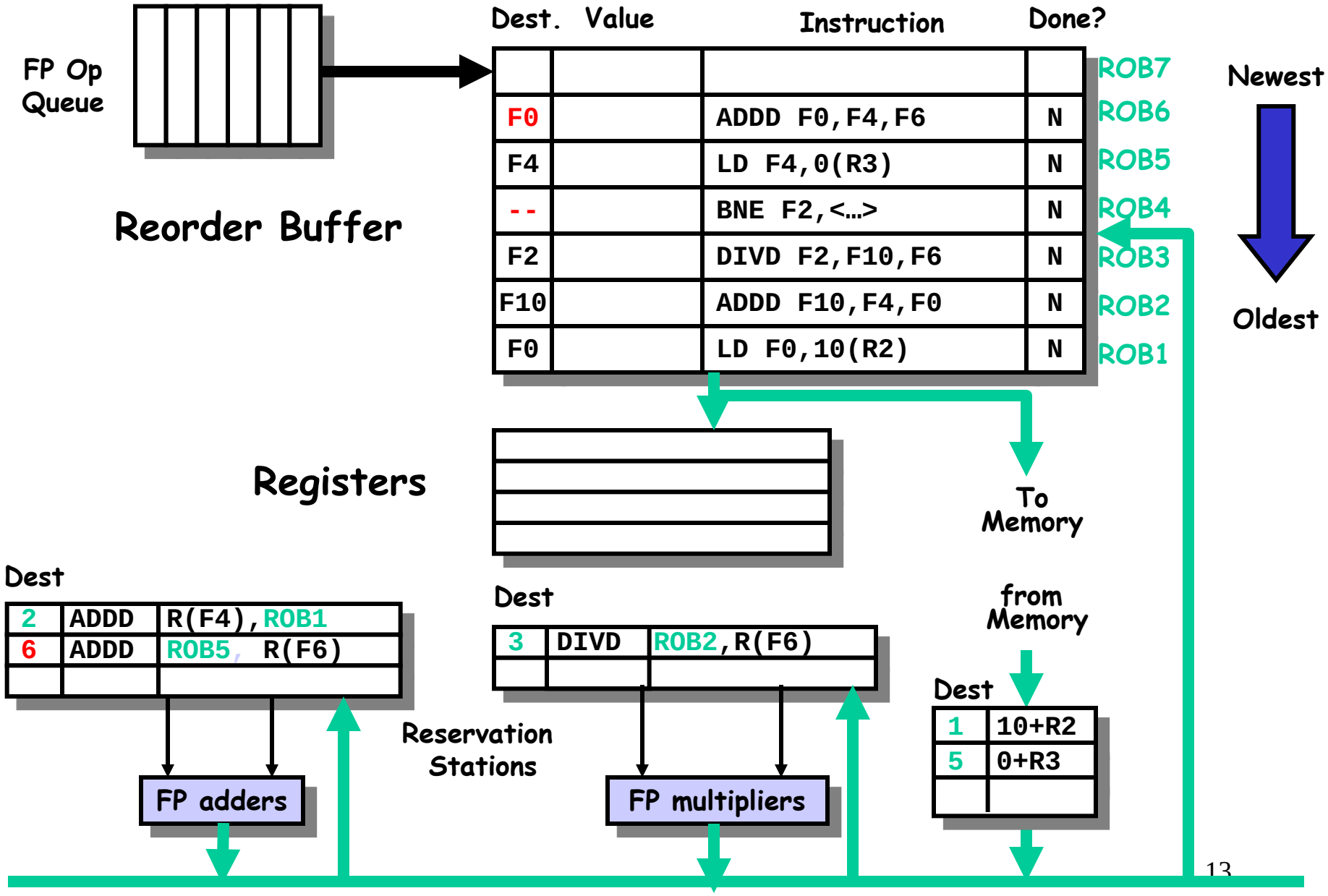


Tomasulo with Reorder buffer

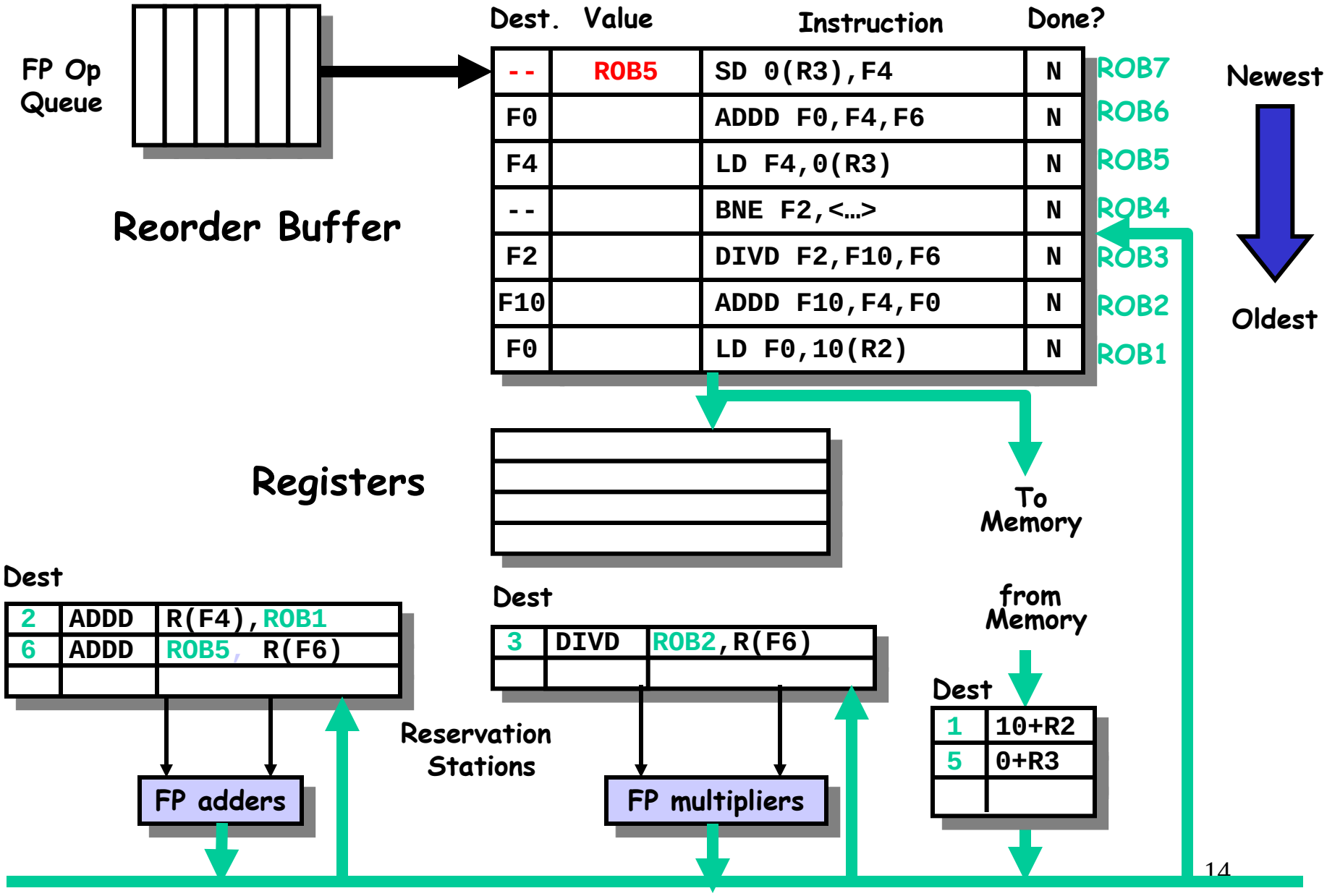


- Skip some cycles

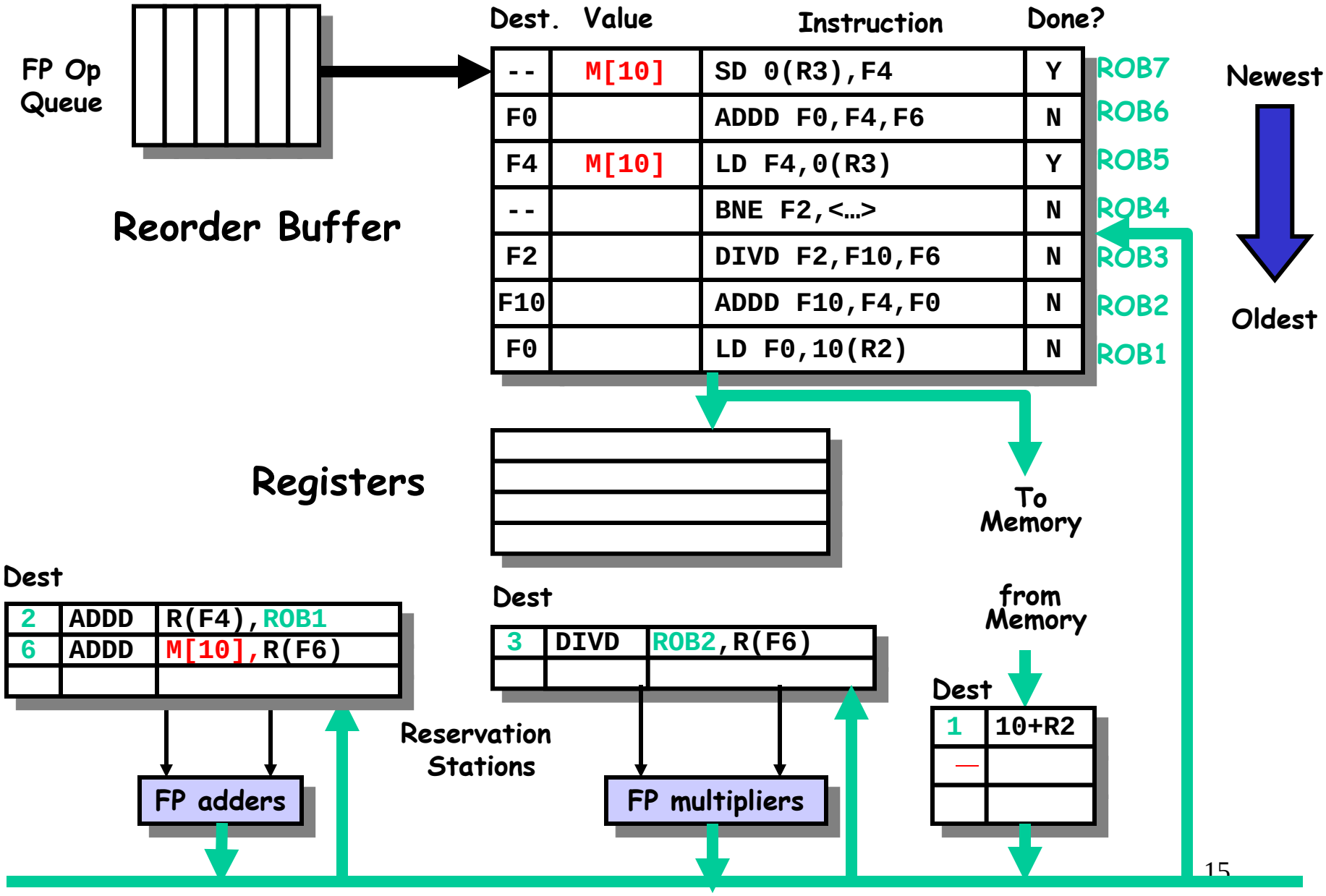
Tomasulo with Reorder buffer



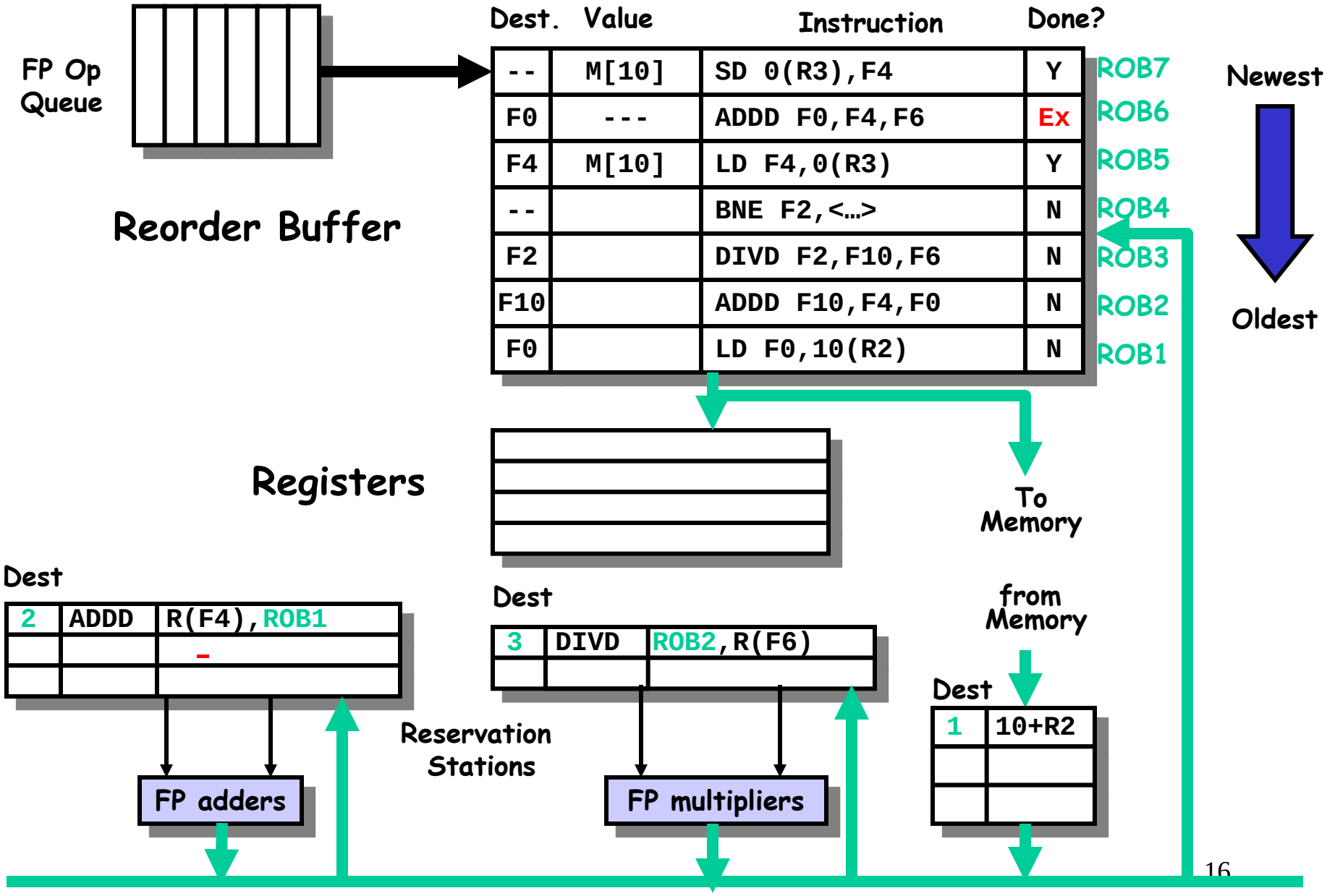
Tomasulo with Reorder buffer



Tomasulo with Reorder buffer



Tomasulo with Reorder buffer



Notes

- 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

Tomasulo Algorithm vs. Scoreboard

- Control & buffers distributed with Function Units (FU) vs. centralized in scoreboard;
  - FU buffers called "reservation stations"; have pending operands
- Registers in instructions replaced by values or pointers to reservation stations(RS); called register renaming ;
  - avoids WAR, WAW hazards
  - More reservation stations than registers, so can do optimizations compilers can't
- Results to FU from RS, not through registers, over Common Data Bus that broadcasts results to all FUs
- Load and Stores treated as FUs with RSs as well
- Integer instructions can go past branches, allowing FP ops beyond basic block in FP queue

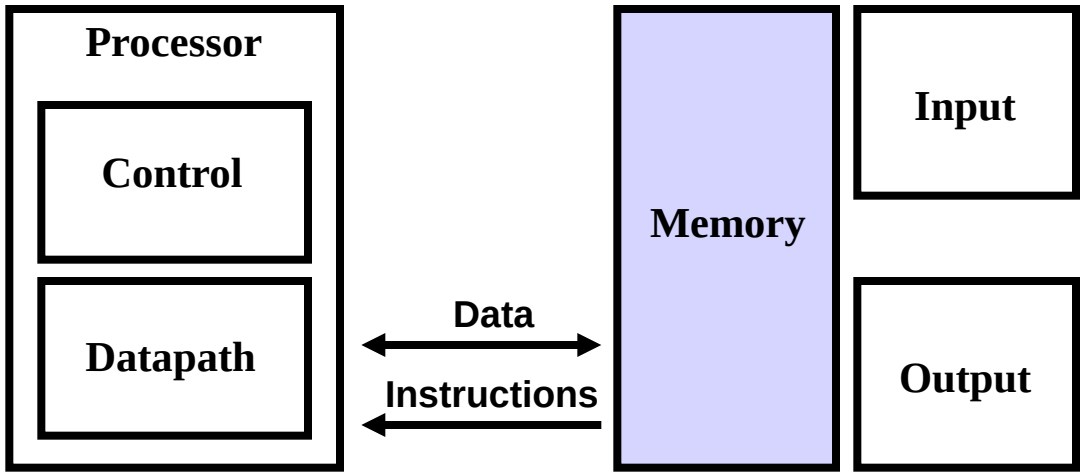
Summary

- Reservations stations: *implicit register renaming* to larger set of registers + buffering source operands
  - Prevents registers as bottleneck
  - Avoids WAR, WAW hazards of Scoreboard
  - Allows loop unrolling in HW
- Not limited to basic blocks
  (integer units gets ahead, beyond branches)
- Today, helps cache misses as well
  - Don’t stall for L1 Data cache miss (insufficient ILP for L2 miss?)
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation
- 360/91 descendants are Pentium III; PowerPC 604; MIPS R10000; HP-PA 8000; Alpha 21264

Memory Hierarchy: Introduction

Memory Systems - the Big Picture

- Memory provides processor with
  - Instructions
  - Data
- Problem: memory is too slow and too small



"Five Classics Components" Picture

Technology Trends

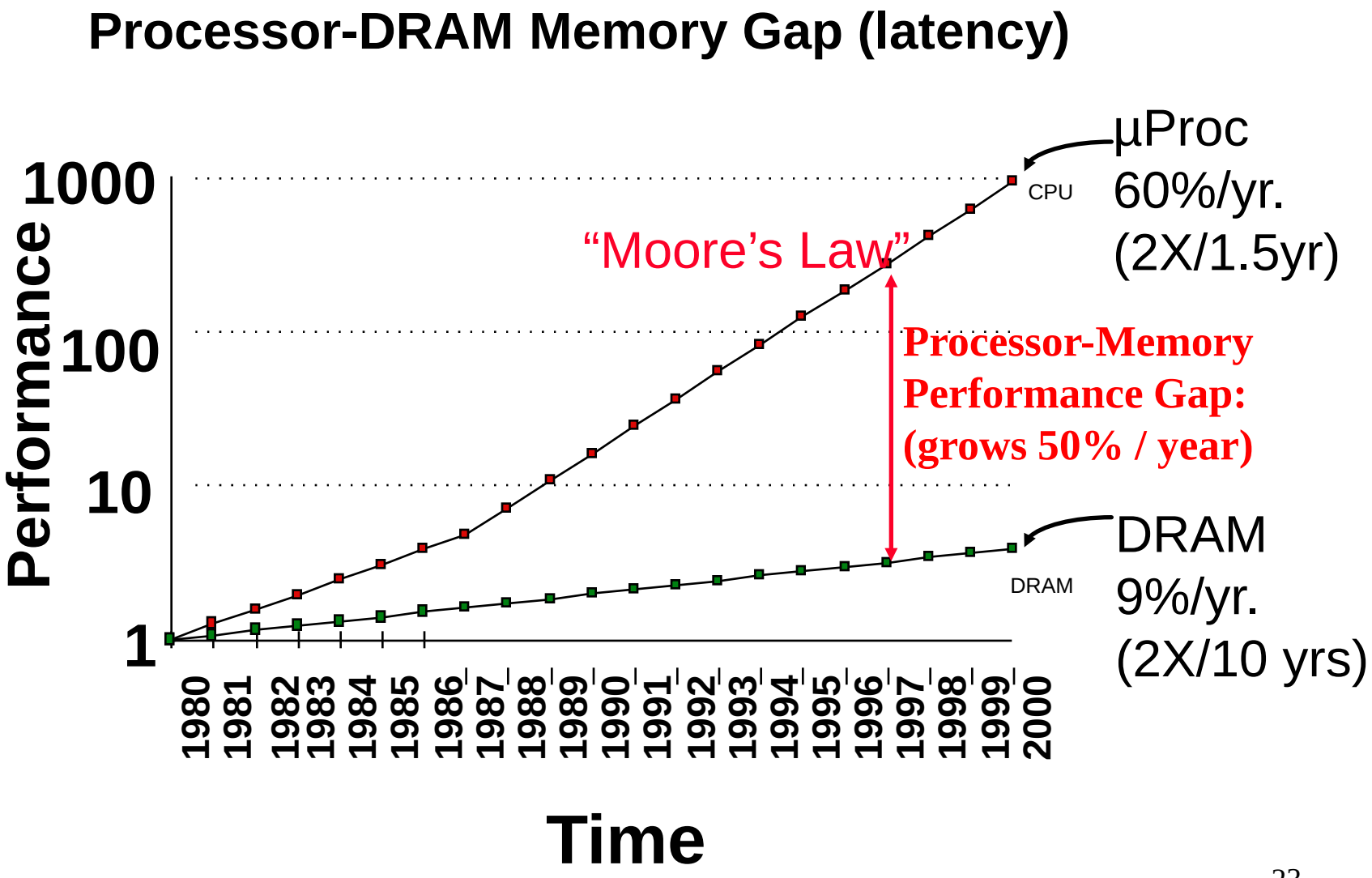
	Capacity	Speed (latency)
Logic:	2x in 3 years	2x in 3 years
DRAM:	4x in 3 years	2x in 10 years
Disk:	4x in 3 years	2x in 10 years

DRAM

Year	Size	Cycle Time
1980	64 Kb	250 ns
1983	256 Kb	220 ns
1986	1 Mb	190 ns
1989	4 Mb	165 ns
1992	16 Mb	145 ns
1995	64 Mb	120 ns

1000:1! **2:1!**

Why Cares About Memory Hierarchy?



Today's Microprocessor

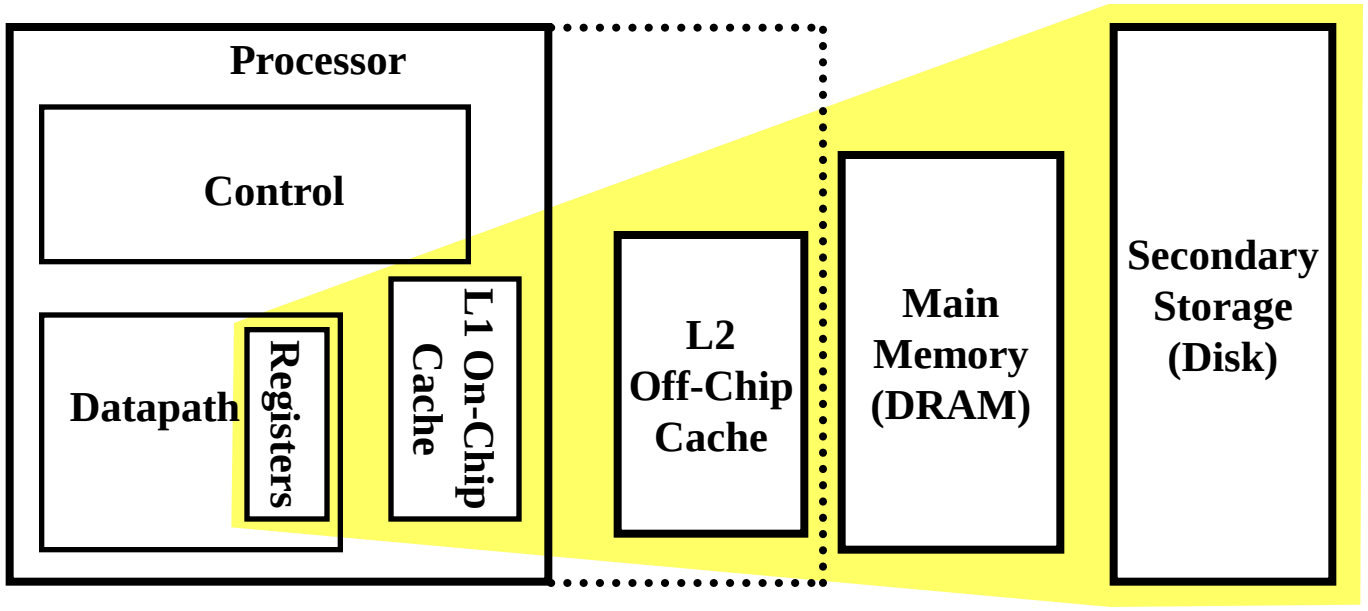
- Rely on caches to bridge gap
- Microprocessor-DRAM performance gap
  - time of a full cache miss in instructions executed

1st Alpha (7000):	340 ns/5.0 ns = 68 clks x 2 or	136 instructions
2nd Alpha (8400):	266 ns/3.3 ns = 80 clks x 4 or	320 instructions
3rd Alpha (t.b.d.):	180 ns/1.7 ns =108 clks x 6 or	648 instructions

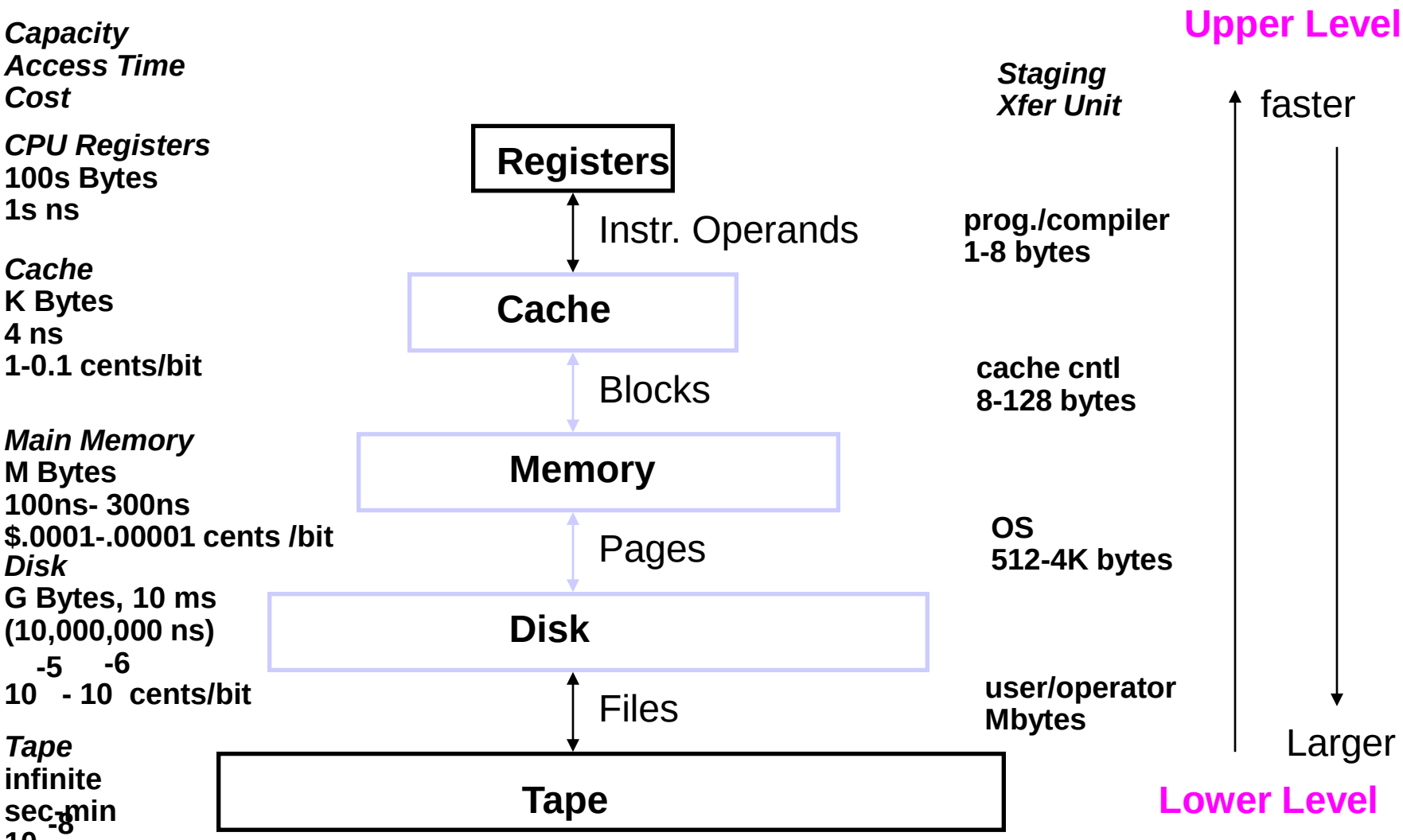
  - 1/2X latency x 3X clock rate x 3X Instr/clock $\Rightarrow$ 5X

Memory Hierarchy - the Big Picture

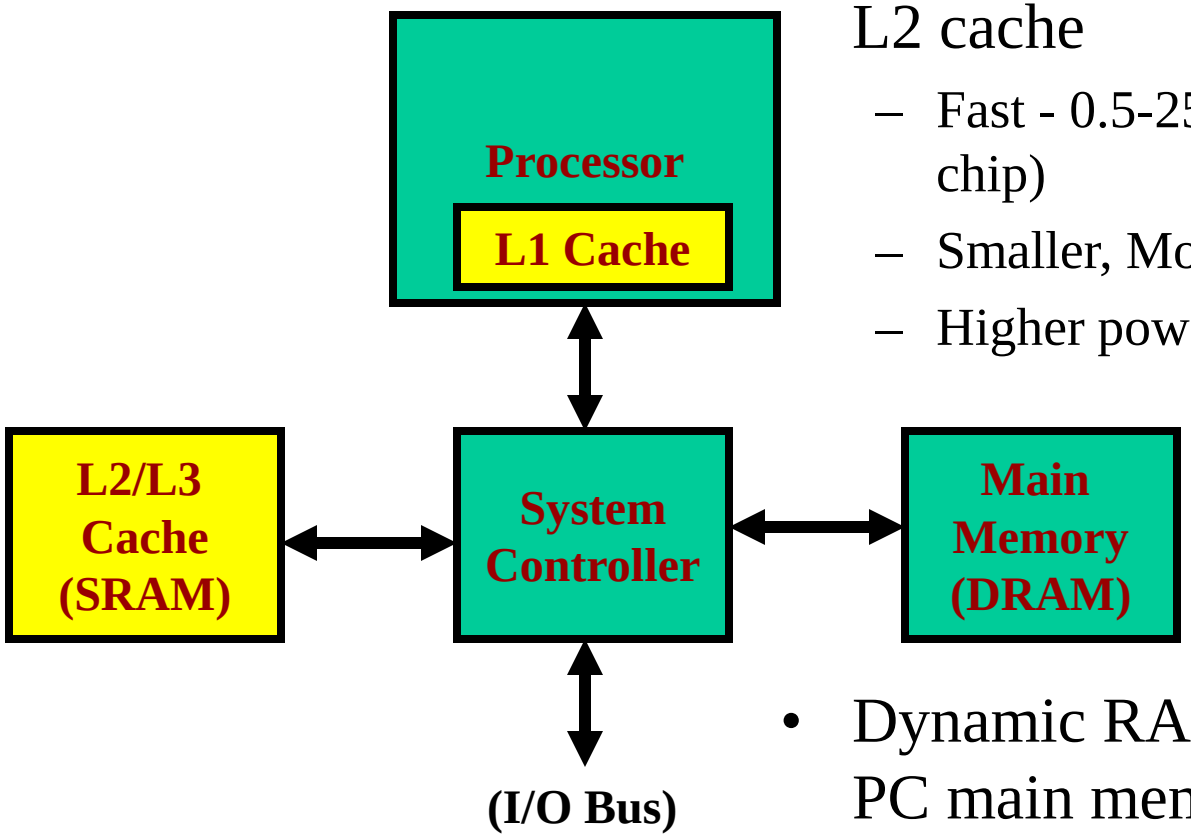
- Problem: memory is too slow and too small
- Solution: memory hierarchy



Levels of Memory Hierarchy



Memory Configuration in Current PCs

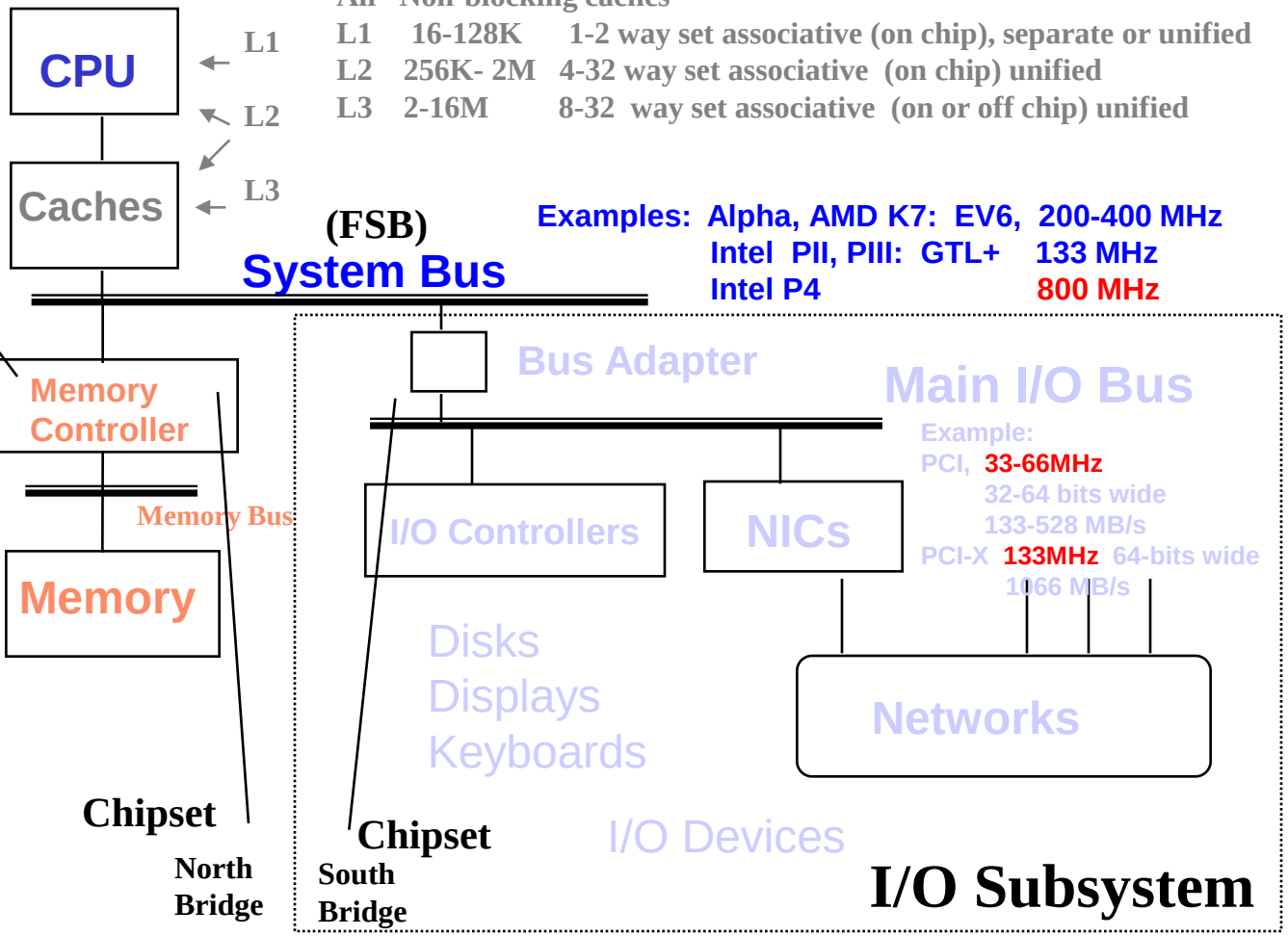


- Static RAM (SRAM) - used for L1, L2 cache
  - Fast - 0.5-25ns access time (less for on-chip)
  - Smaller, More Expensive
  - Higher power consumption
- Dynamic RAM (DRAM) - used for PC main memory
  - Slower - 80-250ns access time*
  - Larger, Cheaper
  - Lower power consumption

System Components

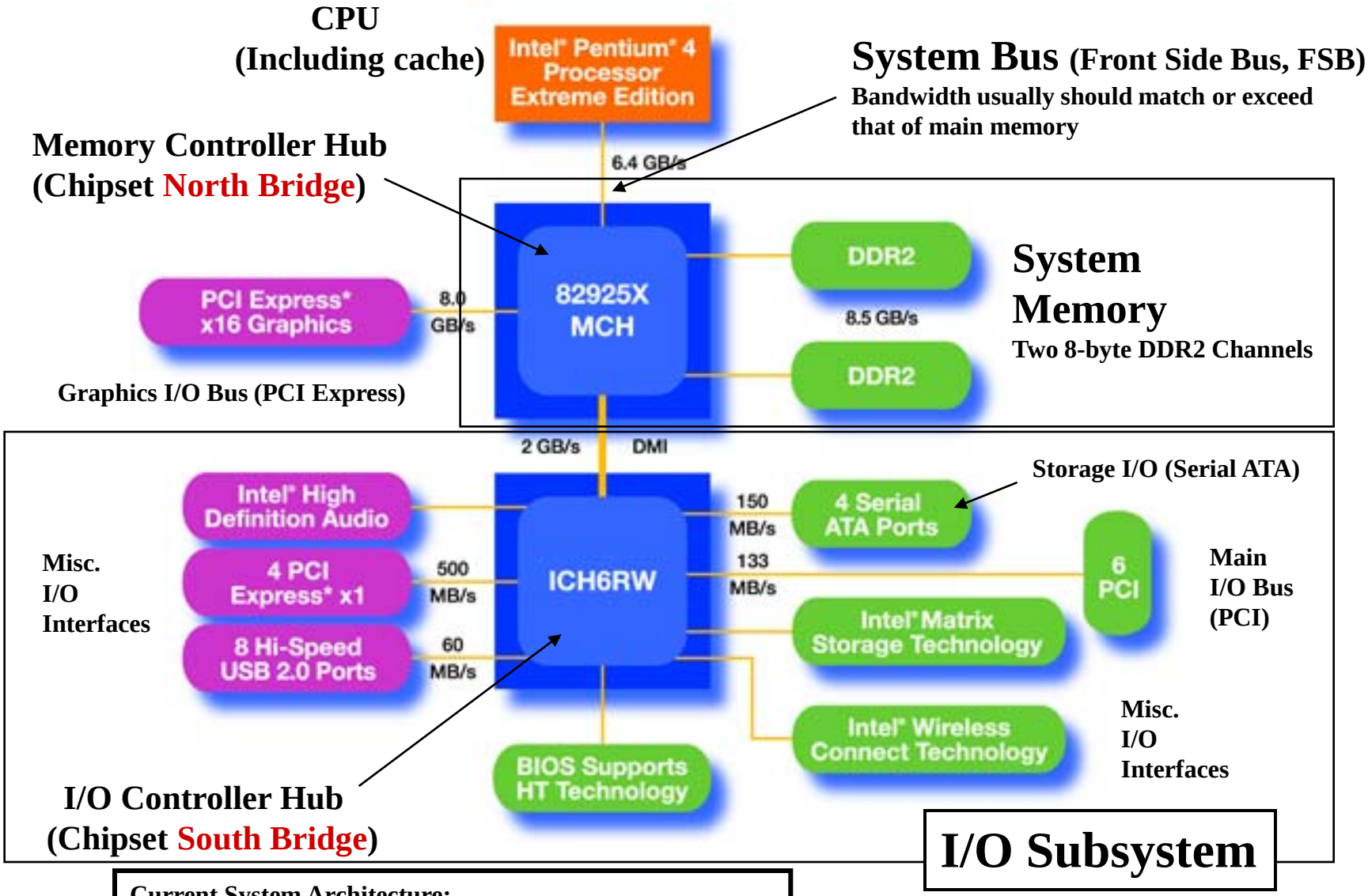
CPU Core
1 GHz - 3.6 GHz
4-way Superscaler
RISC or RISC-core (x86):
- Deep Instruction Pipelines
- Dynamic scheduling
- Multiple FP, integer FUs
- Dynamic branch prediction
- Hardware speculation

All Non-blocking caches
L1 16-128K 1-2 way set associative (on chip), separate or unified
L2 256K- 2M 4-32 way set associative (on chip) unified
L3 2-16M 8-32 way set associative (on or off chip) unified



Important issue: Which component creates a system performance bottleneck?

Intel Pentium 4 System Architecture
(Using The Intel 925 Chipset, 2004)



Current System Architecture:
Isolated I/O: Separate memory and I/O buses.

Source: http://www.anandtech.com/showdoc.aspx?i=2088&p=4

The Principle of Locality

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon
    (e.g., straightline code, array access)
- Last 15 years, HW relied on locality for speed

Why Hierarchy Works

- The principle of locality
  - Programs access a relatively small portion of the address space at any instant of time.

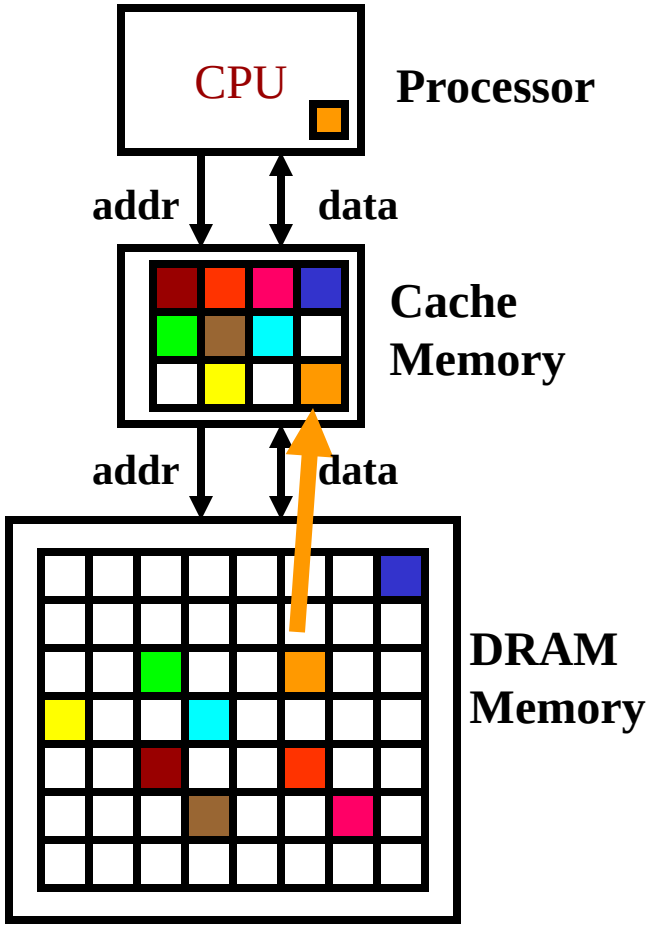
  - Temporal locality: recently accessed data is likely to be used again
  - Spatial locality: data near recently accessed data is likely to be used soon
- Result: the illusion of large, fast memory

Cache Operation

- Insert between CPU, Main Mem.
- Implement with fast Static RAM
- Holds some of a program's
  - data
  - instructions
- Operation:

Hit: Data in Cache (no penalty)

Miss: Data not in Cache (miss penalty)



Cache Performance Measures

- *Hit rate*: fraction found in the cache
  - So high that we usually talk about *Miss rate = 1 - Hit Rate*
- *Hit time*: time to access the cache
- *Miss penalty*: time to replace a block from lower level, including time to replace in CPU
  - *access time*: time to acccess lower level
  - *transfer time*: time to transfer block
- Average memory-access time (AMAT)

  = Hit time + Miss rate x Miss penalty (ns or clocks)

Fundamental Questions

- **Q1: Where can a block be placed in the upper level?**
  (Block placement)
- **Q2: How is a block found if it is in the upper level?**
  (Block identification)
- **Q3: Which block should be replaced on a miss?**
  (Block replacement)
- **Q4: What happens on a write?**
  (Write strategy)