POLITECNICO MILANO 1863
NECST laboratory

# Exercise Session 9

Recap, Scoreboard+Register Renaming, Tomasulo Speculation, Cache Coherency (Extra: Simple Scheduling)

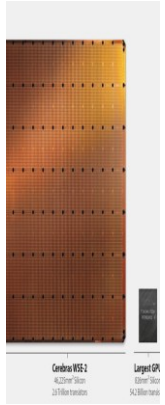
Advanced Computer Architectures

19th May 2025

Davide Conficconi <davide.conficconi@polimi.it>

DI ELETTRONICA
ONE E BIOINGEGNERIA

# Recall: Material (EVERYTHING OPTIONAL)

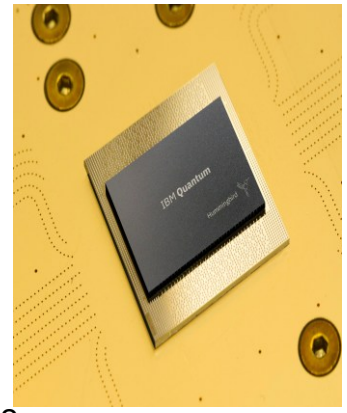


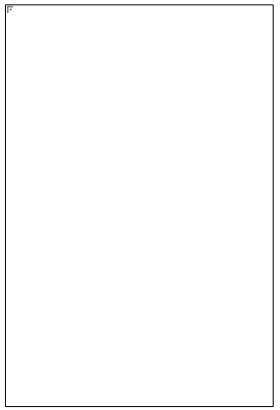
https://webeep.polimi.it/course/view.php?id=14754

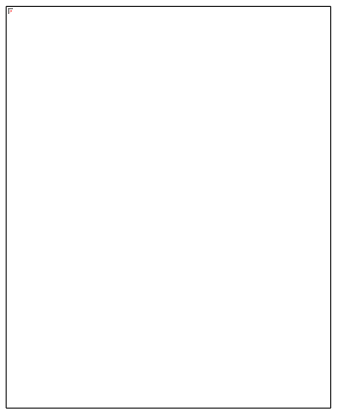
https://tinyurl.com/aca-grid25

Textbook: Hennessy and Patterson, Computer Architecture: A Quantitative Approach

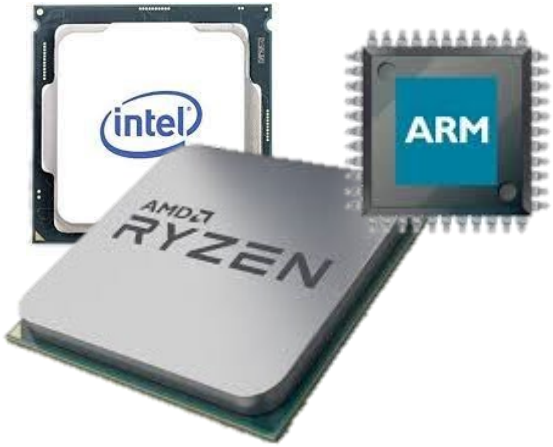
Other Interesting Reference

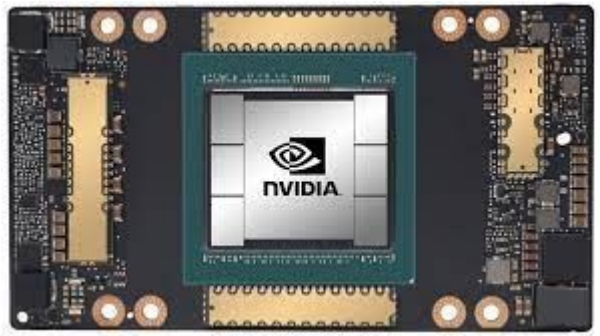


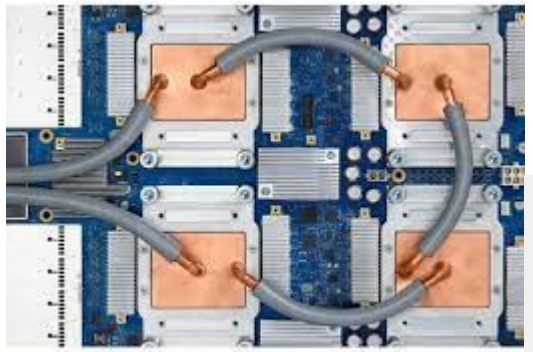



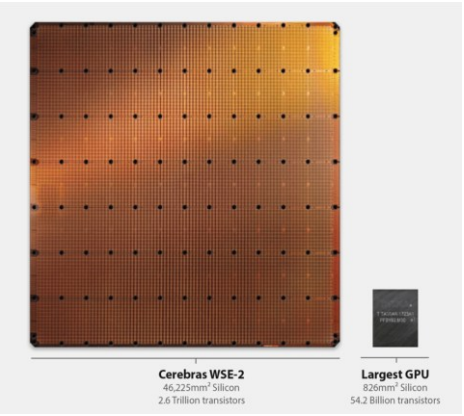



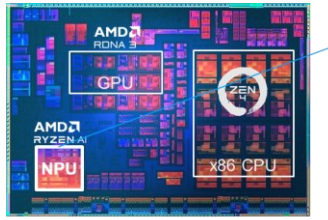
# Recall: The End Goal is:
# Pick the Best Architecture and Understand Performance Engineering
(it is a matter of trade-offs)



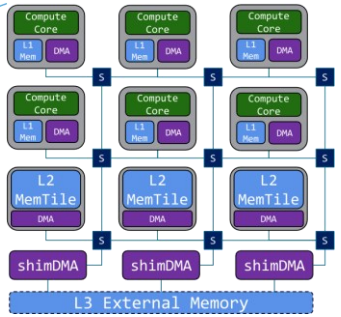



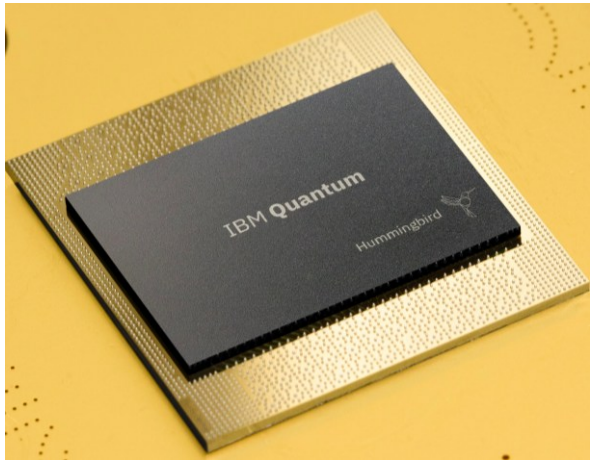


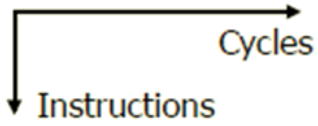

Phoenix floorplan

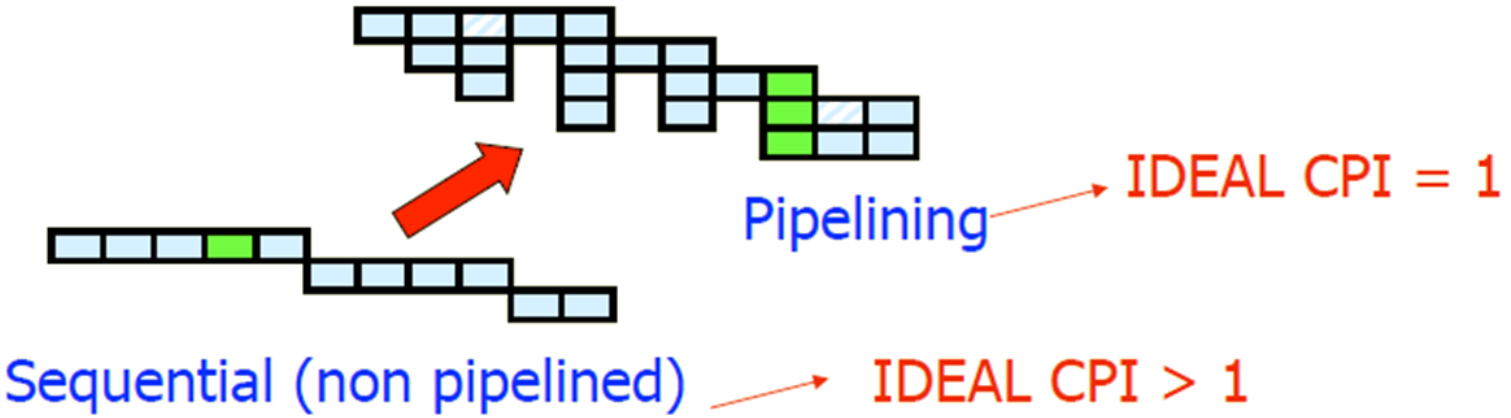






# Recall: The ILP Architecture Journey
## Steps towards exploiting more ILP



# Recall: Pipeline performance

Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls

**Ideal pipeline CPI**: measure of the maximum performance attainable by the implementation

**Structural hazards**: HW cannot support this combination of instructions

**Data hazards**: Instruction depends on result of prior instruction still in the pipeline

**Control hazards**: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches, jumps, exceptions)

# Recall: Three Classes of Hazards

Structural Hazards: Attempt to use the same resource from different instructions simultaneously

*Example:* Single memory for instructions and data

Data Hazards: Attempt to use a result before it is ready

*Example:* Instruction depending on a result of a previous instruction still in the pipeline

Control Hazards: Attempt to make a decision on the next instruction to execute before the condition is evaluated

*Example:* Conditional branch execution

# Recall: Three Classes of Hazards

Structural Hazards: Attempt to use the same resource from different instructions simultaneously

*Example:* Single memory for instructions and data

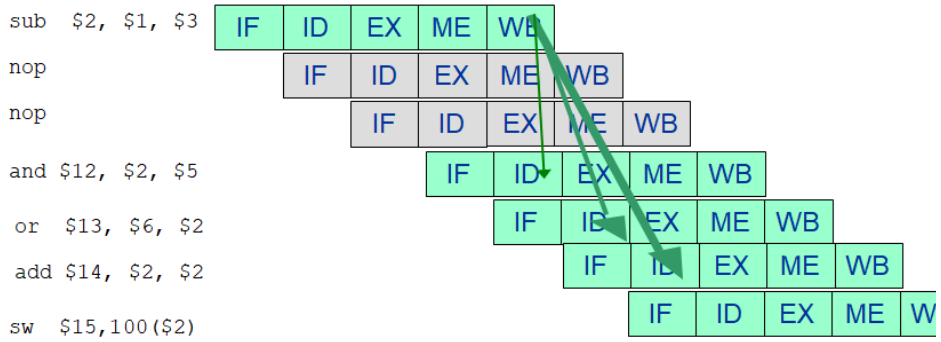
Data Hazards: Attempt to use a result before it is ready

*Example:* Instruction depending on a result of a previous instruction still in the pipeline

Control Hazards: Attempt to make a decision on the next instruction to execute before the condition is evaluated

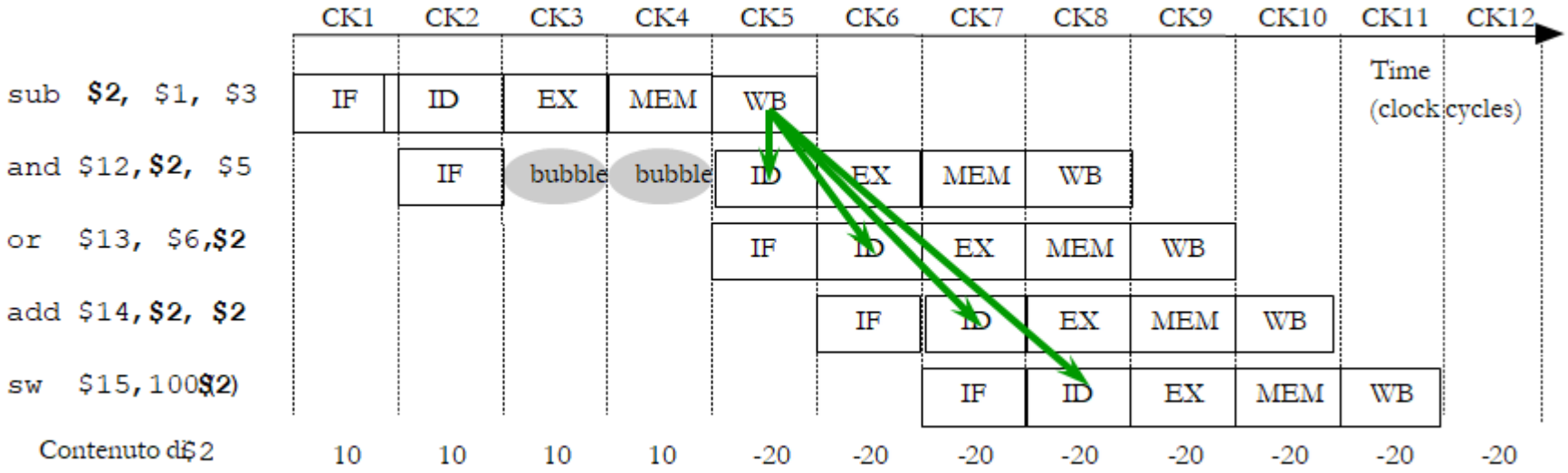
*Example:* Conditional branch execution

# Recall: Data Hazards possible solutions

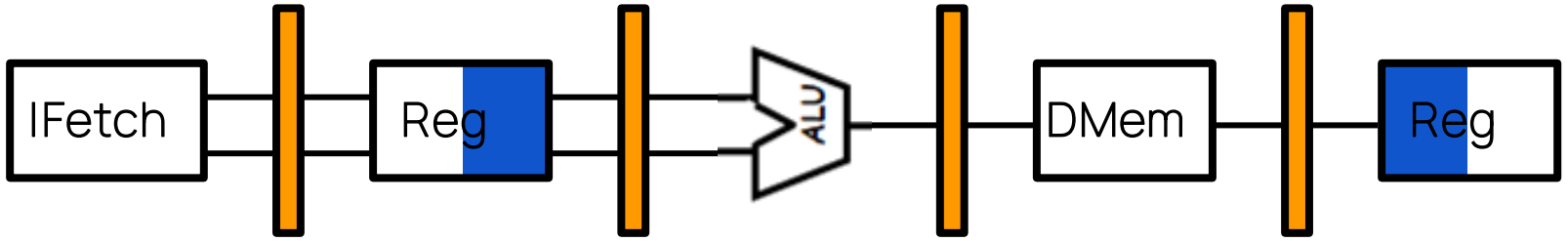


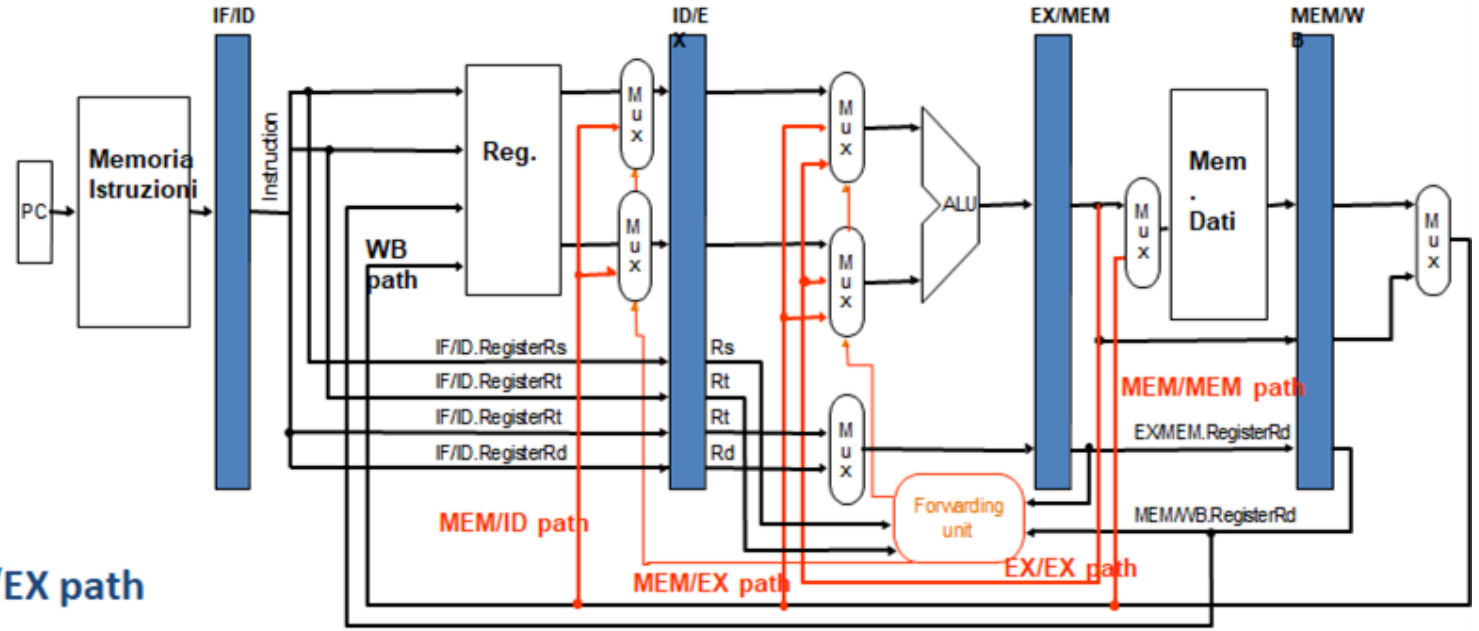

```
sub  $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15,100($2)
add $4, $10, $11
and $7, $8, $9
lw $16, 100($18)
lw $17, 200($19)
```

```
sub $2, $1, $3
add $4, $10, $11
and $7, $8, $9
lw $16, 100($18)
lw $17, 200($19)
and  $12, $2, $5
or $13, $6, $2
add  $14, $2, $2
sw $15,100($2)
```



# Recall: Pipelining and Forwarding




- EX/EX path
- MEM/EX path
- MEM/ID path
- MEM/MEM path

# Recall: Three Classes of Hazards

Structural Hazards: Attempt to use the same resource from different instructions simultaneously

*Example:* Single memory for instructions and data

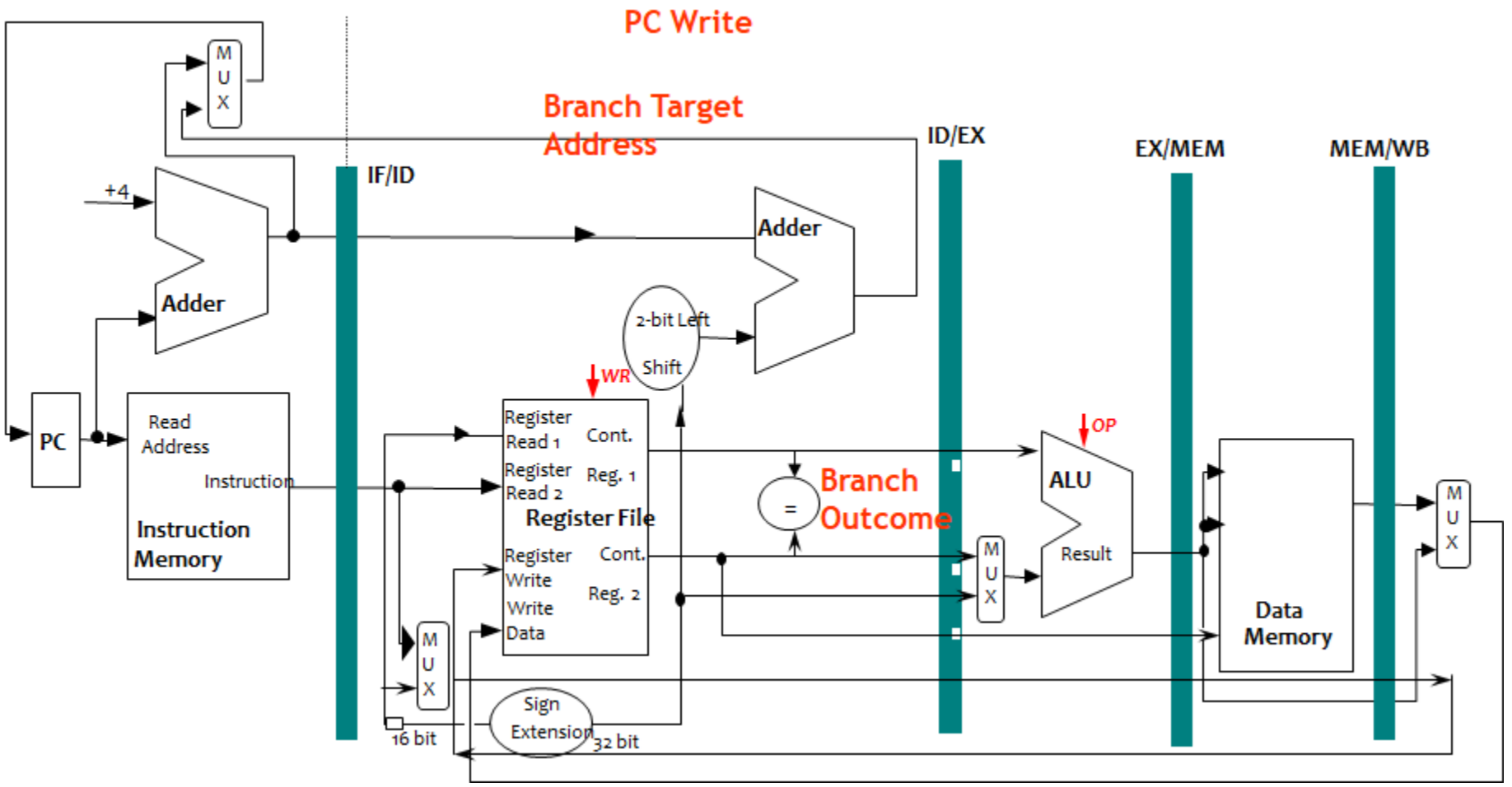
Data Hazards: Attempt to use a result before it is ready

*Example:* Instruction depending on a result of a previous instruction still in the pipeline

Control Hazards: Attempt to make a decision on the next instruction to execute before the condition is evaluated

*Example:* Conditional branch execution

# Recall: Early-evaluation PC



# Recall: Static Branch Prediction Techniques

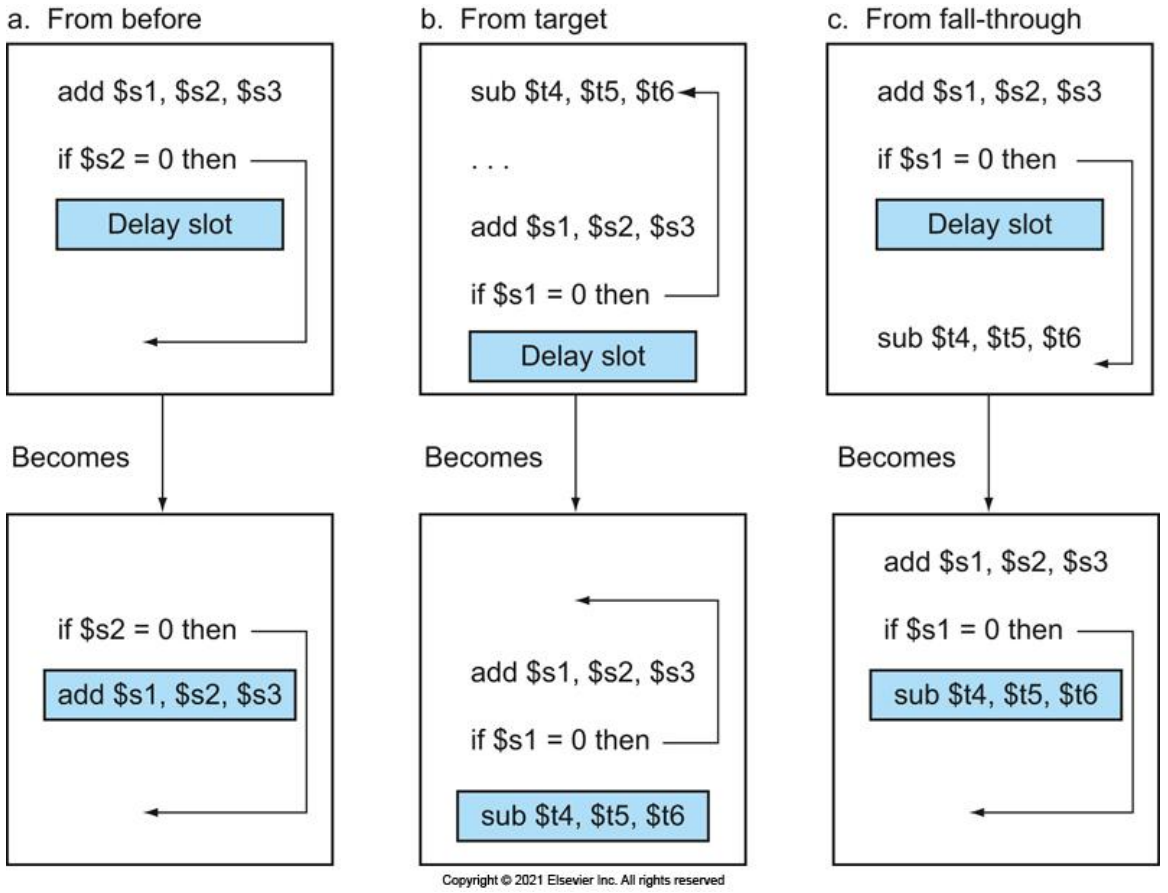
Branch Always Not Taken (Predicted-Not-Taken)

Branch Always Taken (Predicted-Taken)

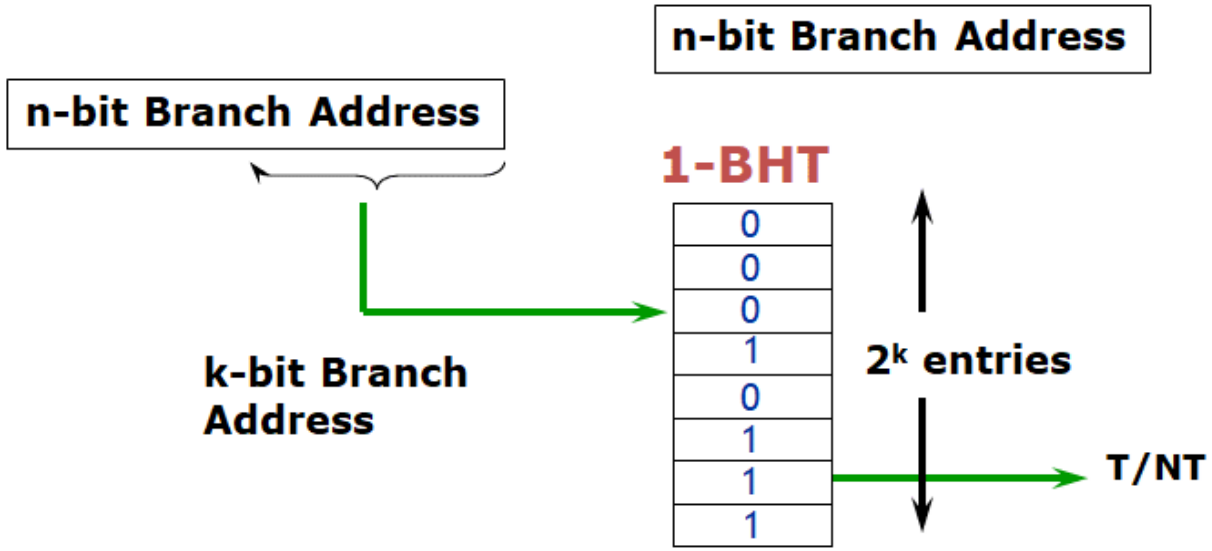
Backward Taken Forward Not Taken (BTFNT)

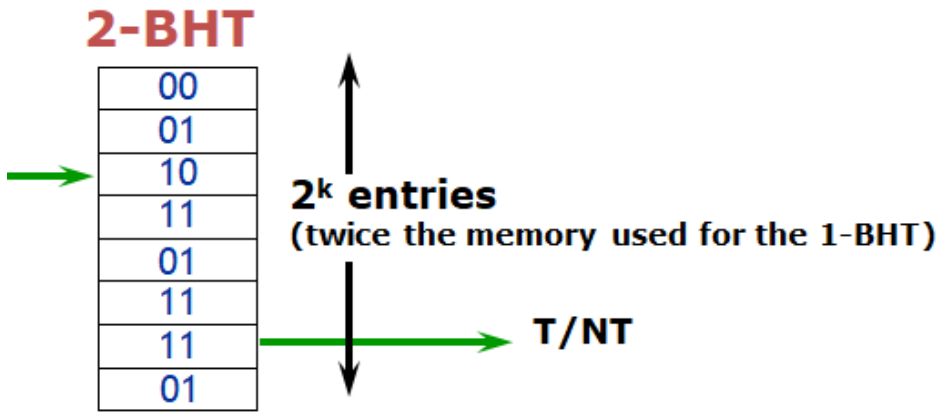
Profile-Driven Prediction

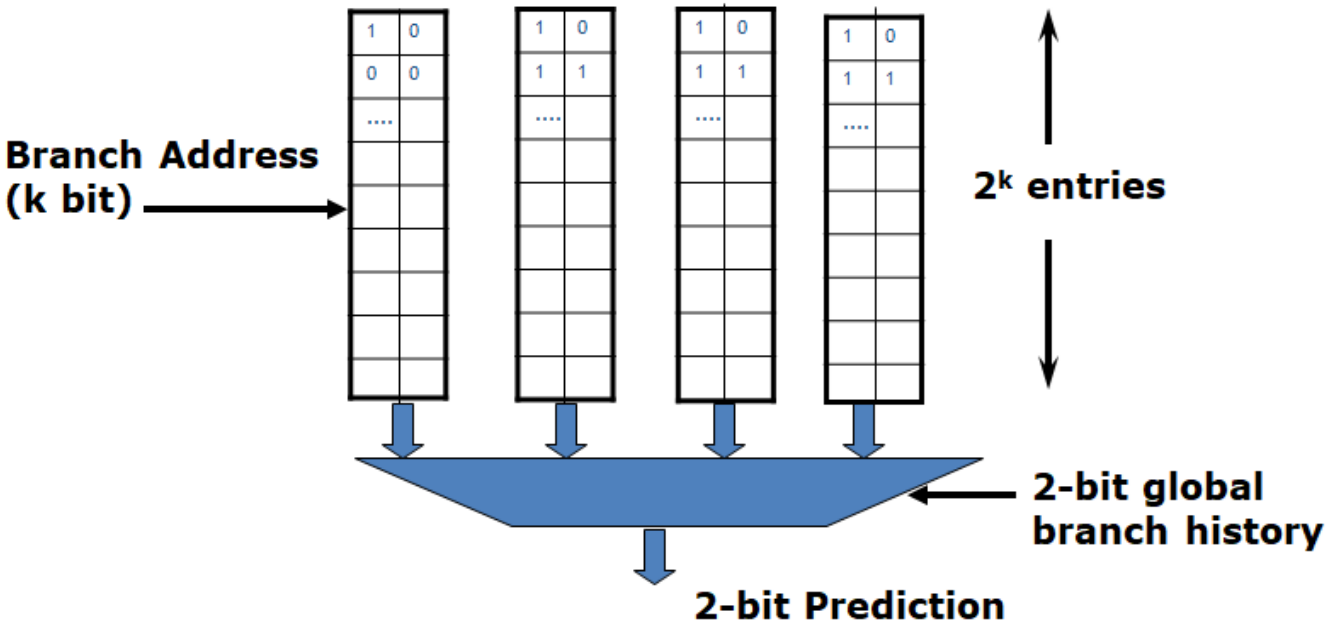
Delayed Branch

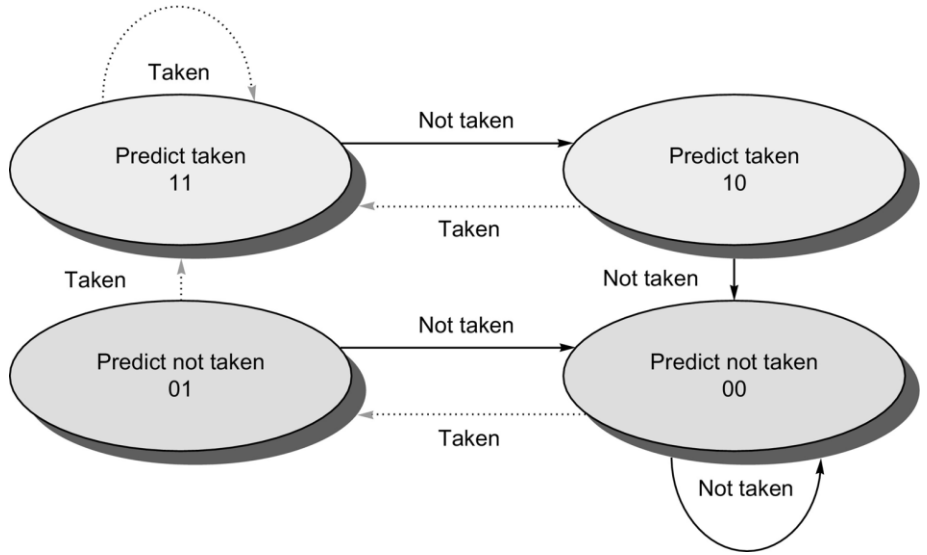


# Recall: Dynamic Branch Prediction

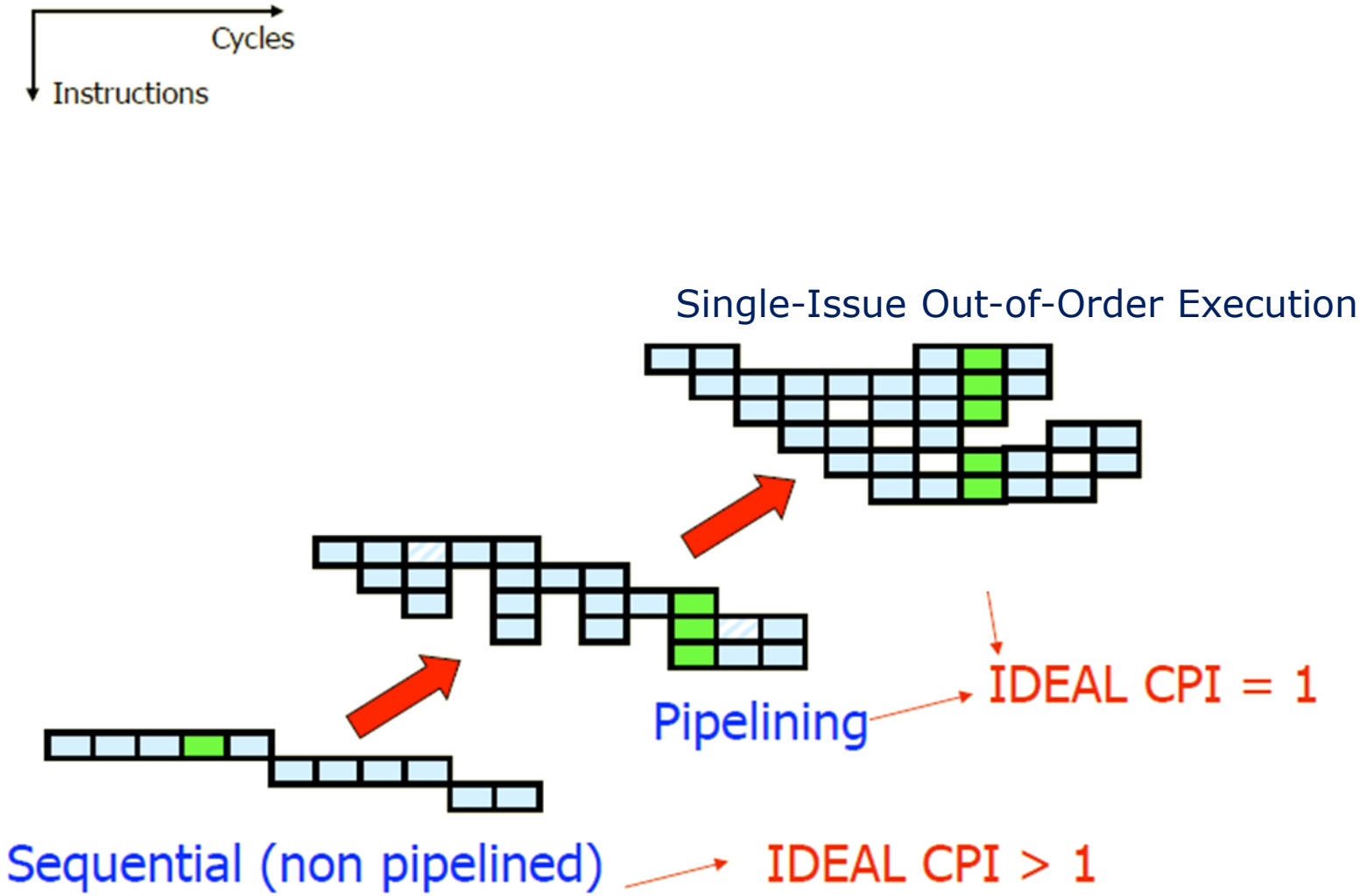


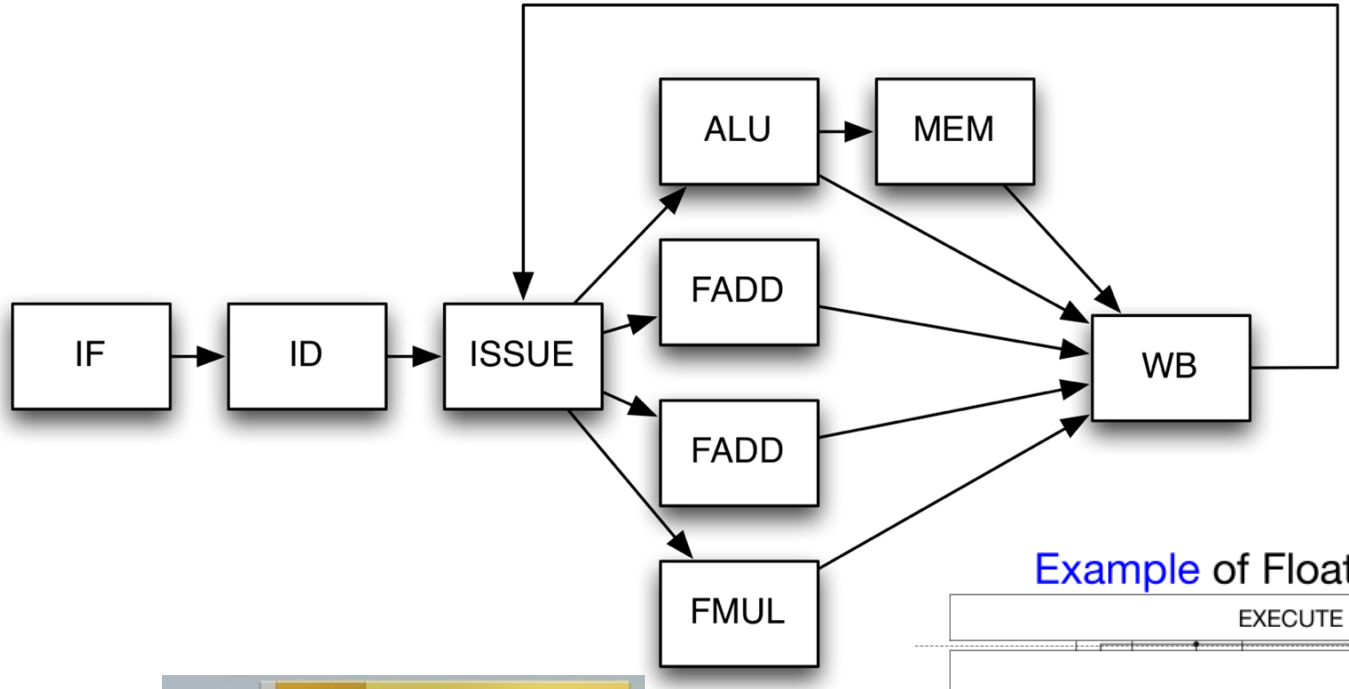



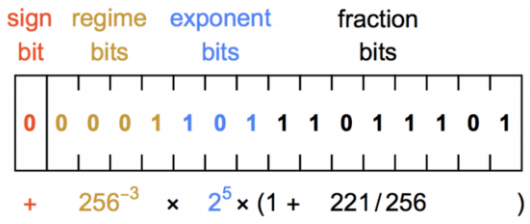



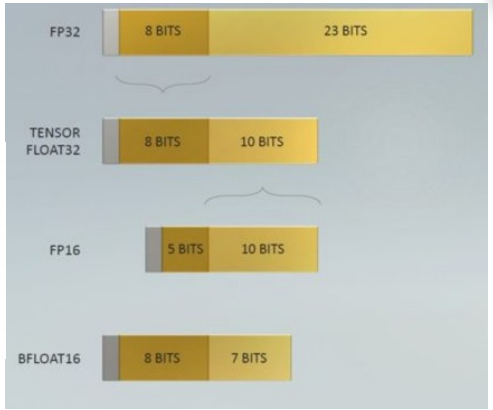
# Recall: The ILP Architecture Journey
## Steps towards exploiting more ILP

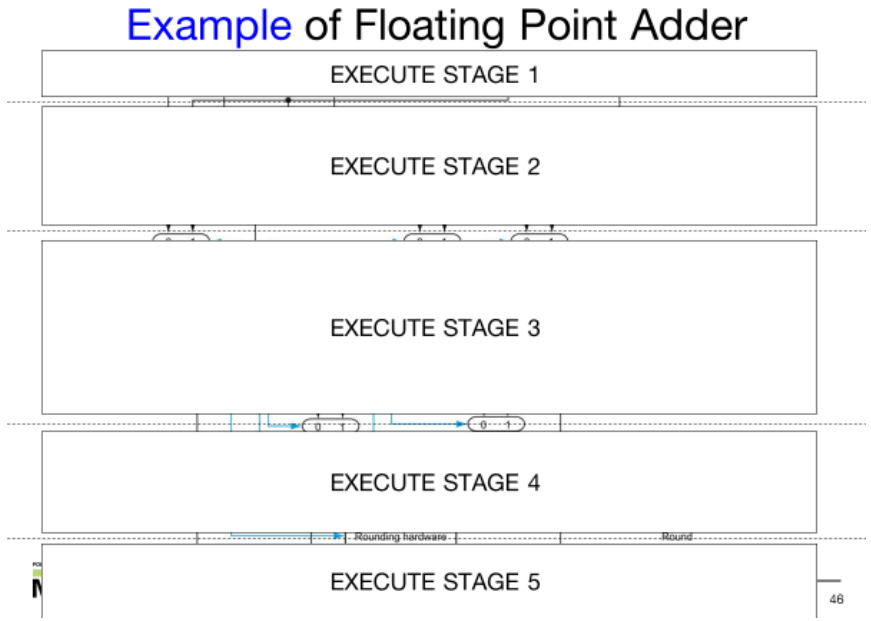


# Recall: Floating-Point → Complex/Multicycle Pipeline



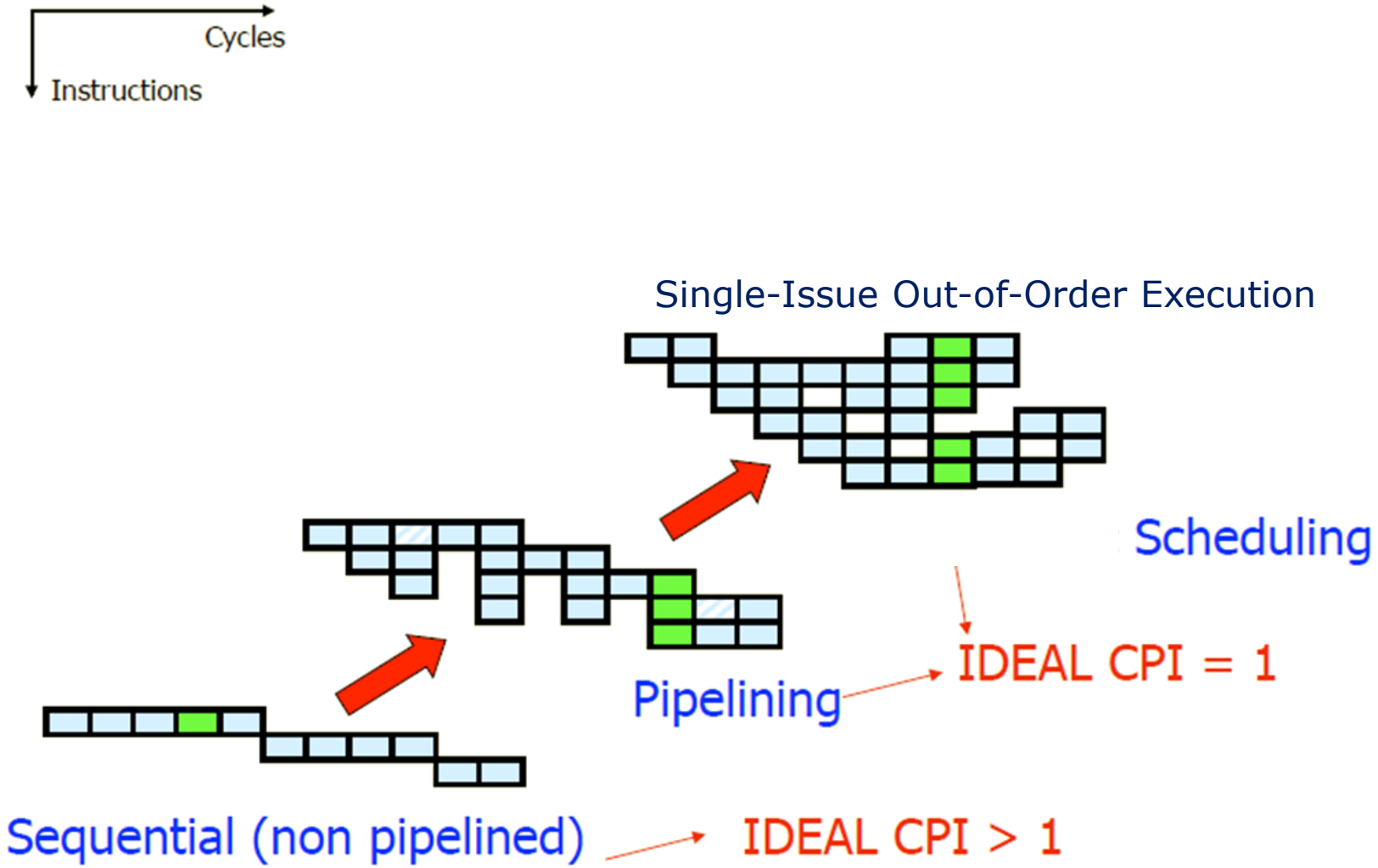




https://www.sigarch.org/posit-a-potential-replacement-for-ieee-754/

# Recall: The ILP Architecture Journey
## Steps towards exploiting more ILP



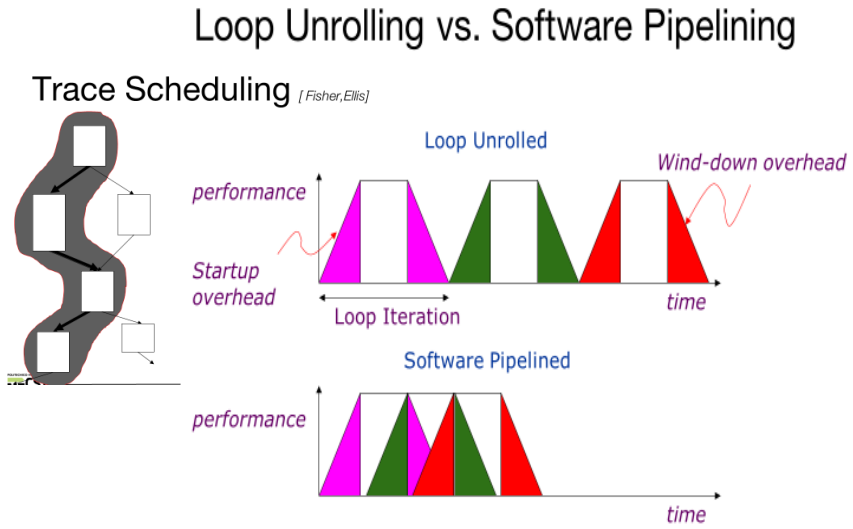
# Recall: Scheduling: Static and Dynamic

- Static Scheduling: Rely on software for identifying potential parallelism

- Dynamic Scheduling: Depend on the hardware to locate parallelism

# Recall: Scheduling: Static and Dynamic

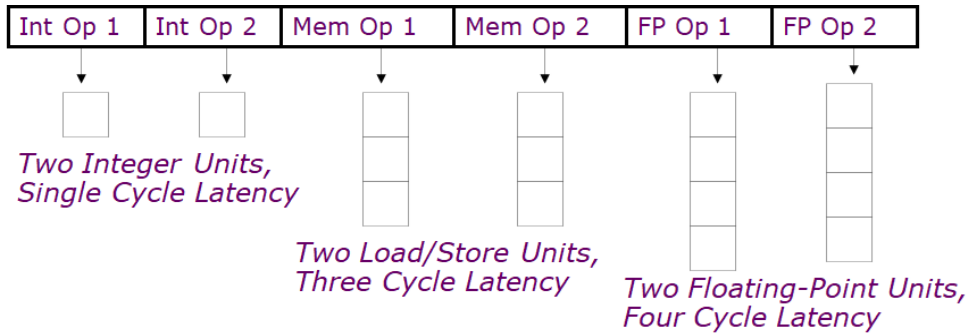
- Static Scheduling: Rely on software for identifying potential parallelism

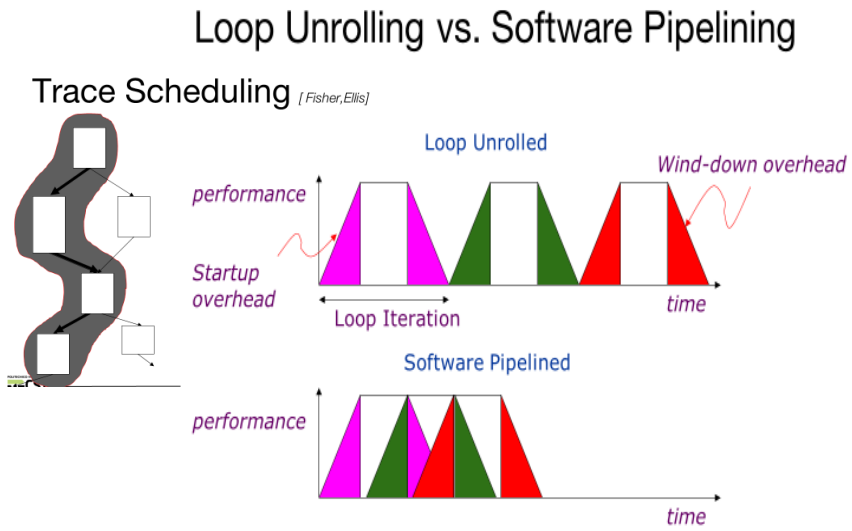



[Joseph A. Fisher. Very Long Instruction Word Architectures and the ELI-512. In ISCA, 1983]

# Recall: Scheduling: Static and Dynamic

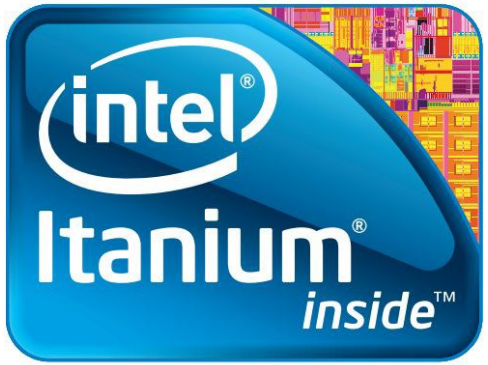
- Static Scheduling: Rely on software for identifying potential parallelism




VLIW Disadvantages:

- Statically finding parallelism
- Code size
- No hazard detection hardware
- Binary code compatibility




## VLIW Issues and an "EPIC Failure"

- Compiler couldn't handle complex dependencies in integer code (pointers)
- Code size explosion
- Unpredictable branches
- Variable memory latency (unpredictable cache misses)
  - Out of Order techniques dealt with cache latencies
- Out of Order subsumed VLIW benefits
- "*The Itanium approach...was supposed to be so terrific –until it turned out that the wished-for compilers were basically impossible to write.*"
  - Donald Knuth, Stanford
- Pundits noted delays and under performance of Itanium product ridiculed by the chip industry

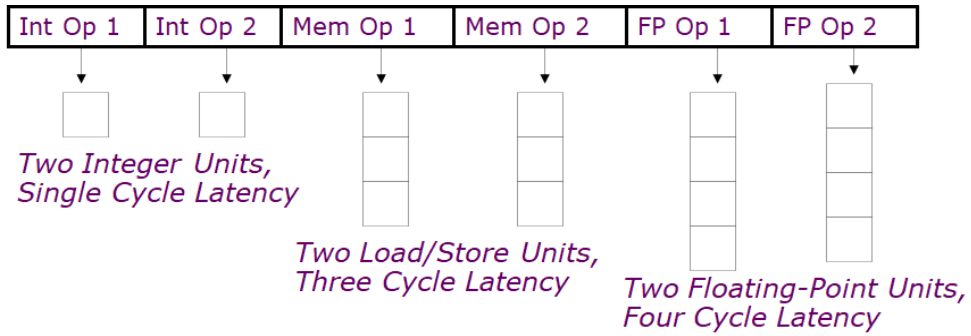
Itanimum ⇒ "Itanic" (like infamous ship *Titanic*)

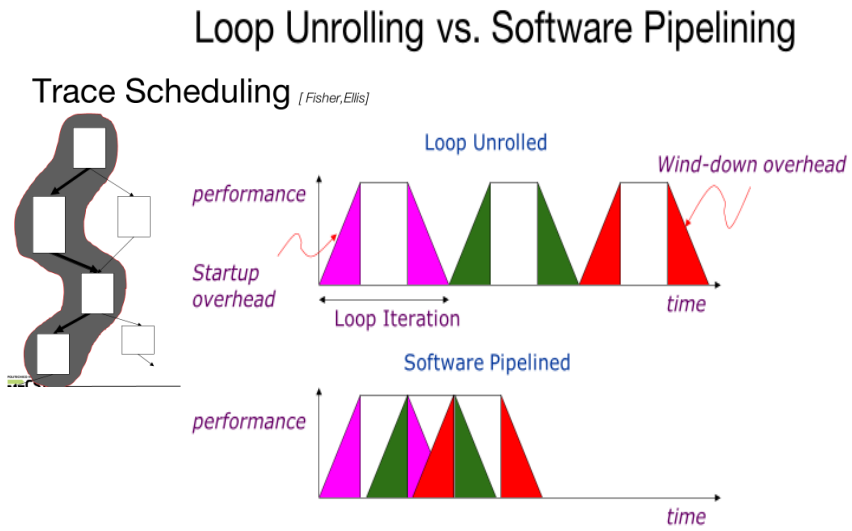
18

Image from H&P Turing Lecture: A New Golden Age for Computer Architecture: Domain-Specific Hardware/Software Co-Design, Enhanced Security, Open Instruction Sets, and Agile Chip Development

# Recall: Scheduling: Static and Dynamic

- Static Scheduling: Rely on software for identifying potential parallelism

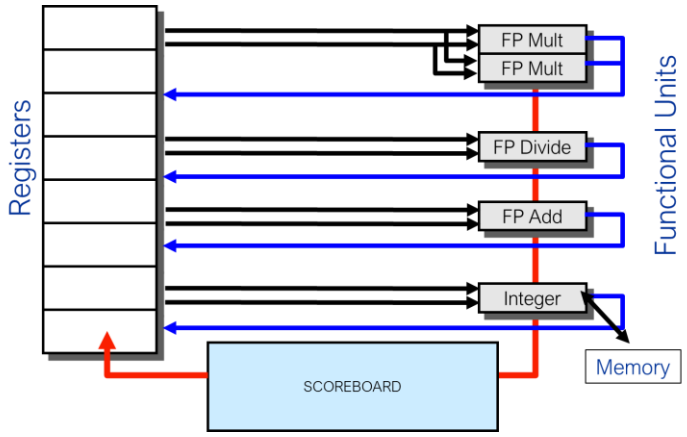


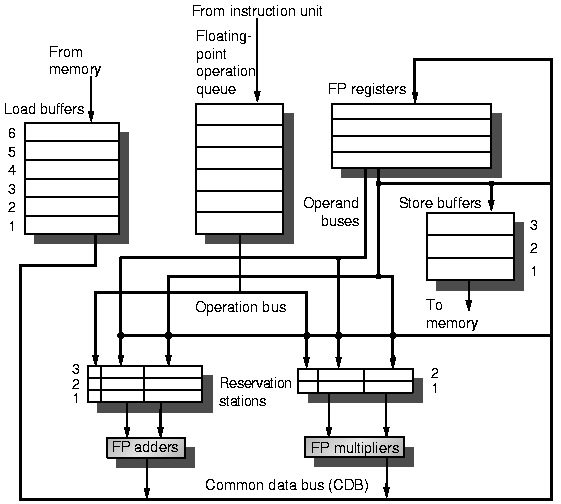




VLIW Disadvantages:

- Statically finding parallelism
- Code size
- No hazard detection hardware
- Binary code compatibility

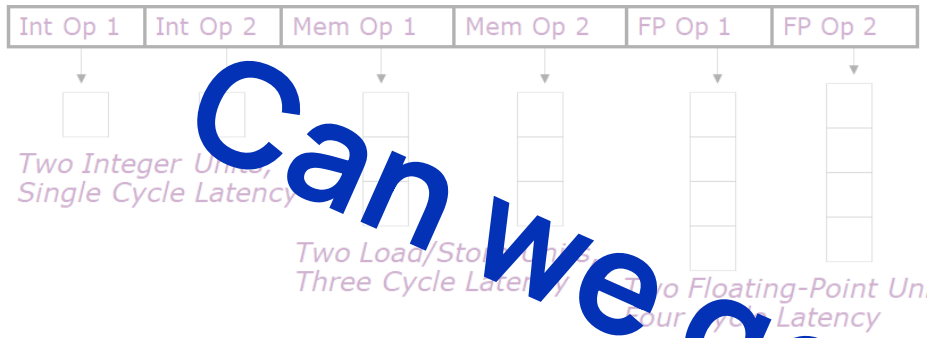
- Dynamic Scheduling: Depend on the hardware to locate parallelism



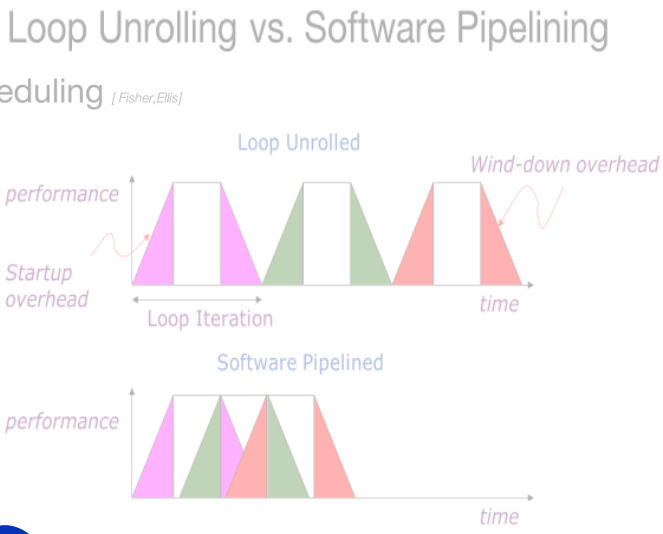



# Recall: Scheduling: Static and Dynamic

- Static Scheduling: Rely on software for identifying potential parallelism



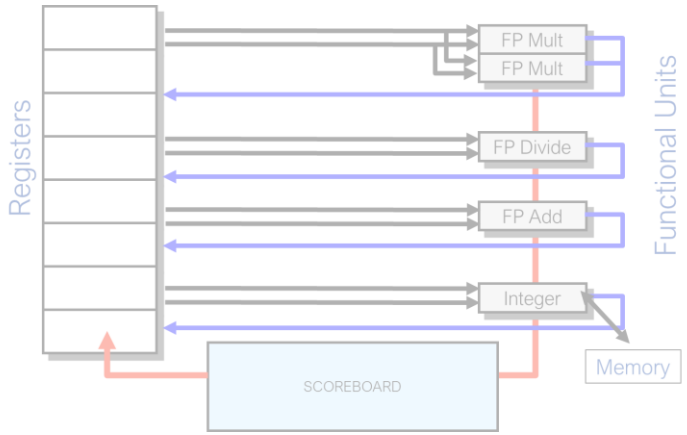

Can we go beyond this?

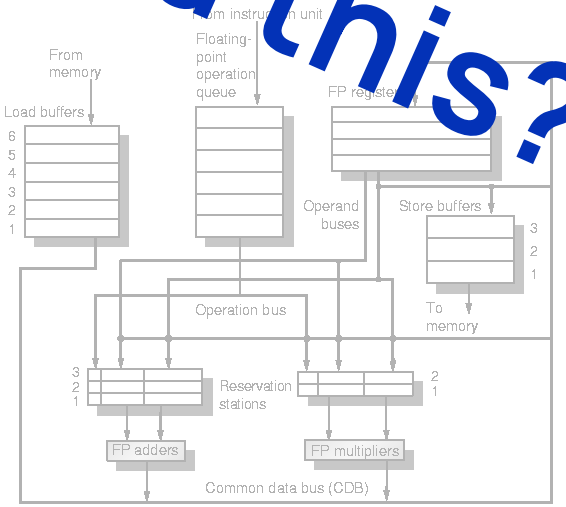



VLIW Disadvantages:

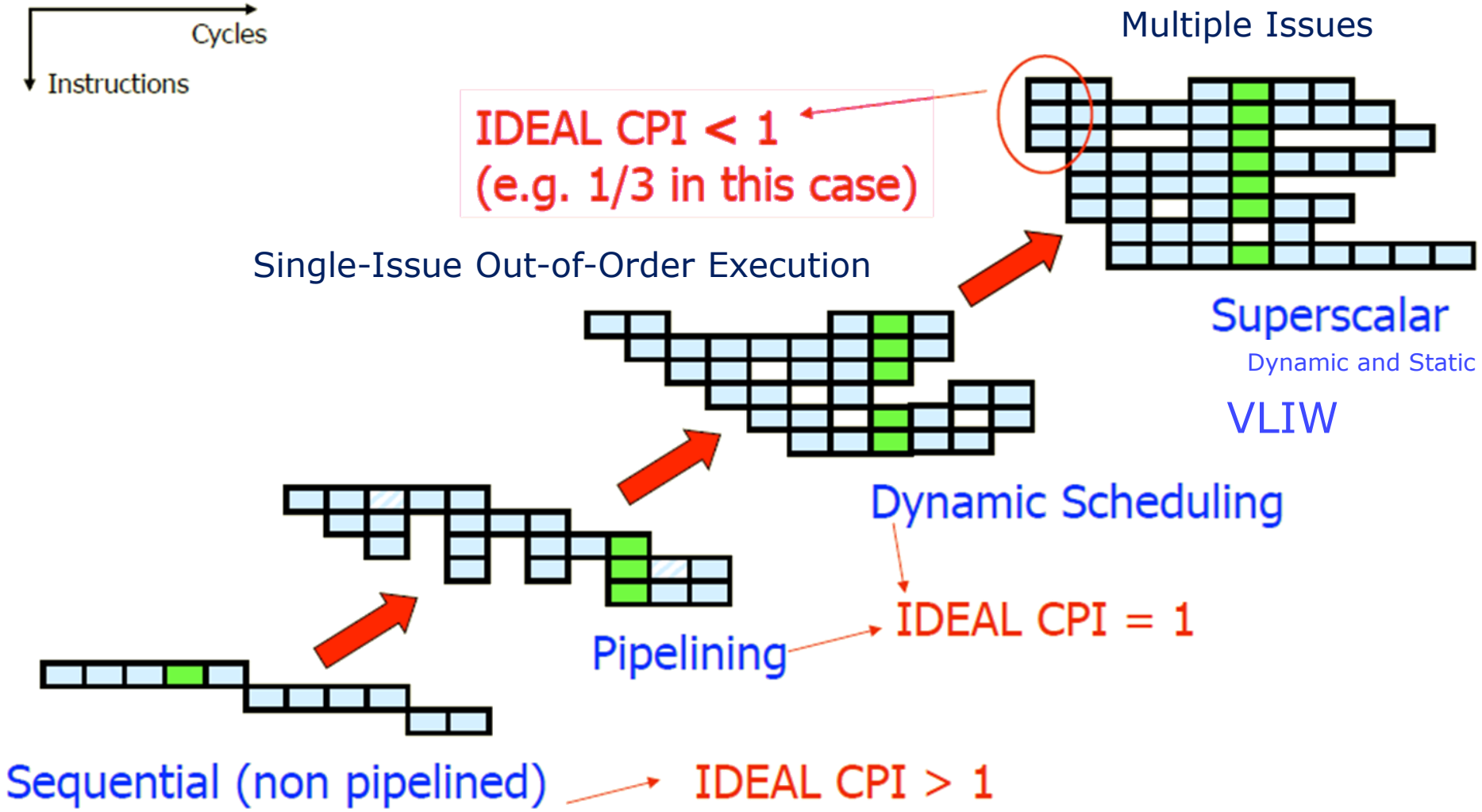
- Statically finding parallelism
- Code size
- No hazard detection hardware
- Binary code compatibility

- Dynamic Scheduling: Depend on the hardware to locate parallelism

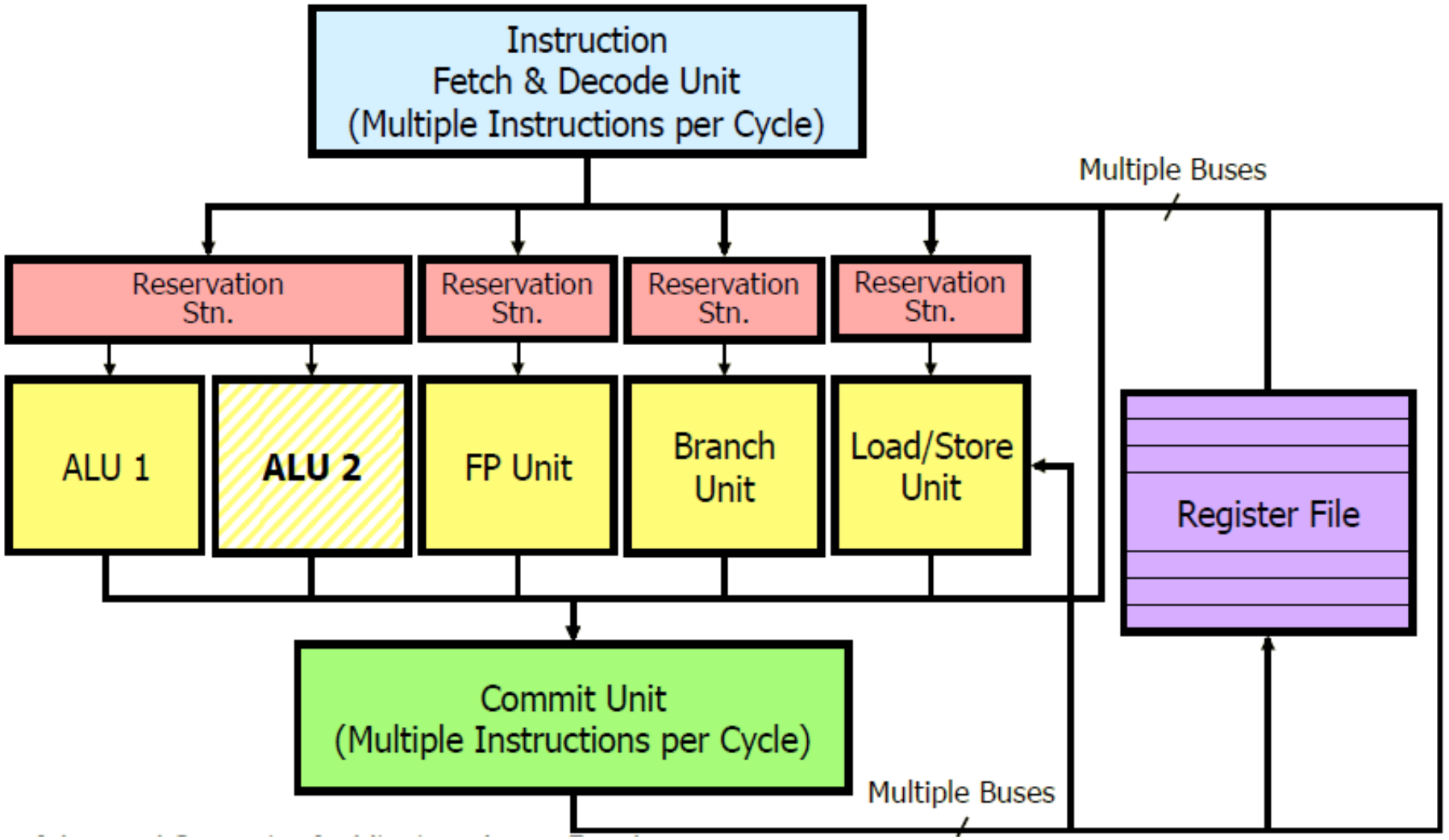


# Recall: The ILP Architecture Journey
## Superscalar Architectures

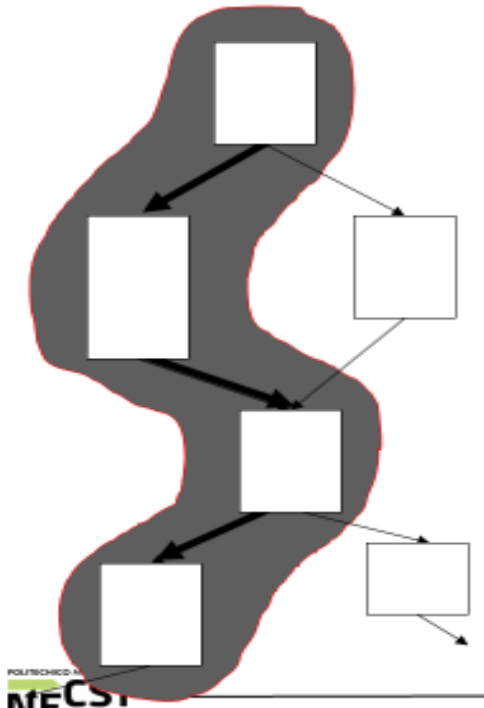


# Recall: Superscalar Processors



# Exe Scoreboard

## Trace Scheduling *[ Fisher,Ellis]*



Pick string of basic blocks, a trace, that represents most frequent branch path

Use profiling feedback or compiler heuristics to find common branch paths

Schedule whole "trace" at once

Add fixup code to cope with branches jumping out of trace

NE CST laboratory

POLITECNICO MILANO 1863

10

[Parallel operation in the control data 6600]

# Recall: the Scoreboard pipeline

| ISSUE | READ OPERAND | EXE COMPLETE | WB |
|---|---|---|---|
| Decode instruction; | Read operands; | Operate on operands; | Finish exec; |
| Structural FUs check;<br>WAW checks | RAW check;<br>TBC WAR not violated | Notify Scoreboard on completion; | WAR &Struct check (FUs will hold results); |

# Recall: the Scoreboard pipeline with Register Renaming

| ISSUE | READ OPERAND | EXE COMPLETE | WB |
|---|---|---|---|
| Decode instruction; allocate new physical register for result | Read operands; | Operate on operands; | Finish exec; |
| Structural FUs check; ~~WAW checks~~ free physical registers check | RAW check; TBC WAR not violated | Notify Scoreboard on completion; | ~~WAR &~~ Struct check (FUs will hold results); |

No checks for WAR or WAW hazards!

# Exe 3 Scoreboard: the Code

```
I1:  LD F6 32+ R2
I2:  ADDD F2 F6 F4
I3:  MULTD F0 F4 F2
I4:  SUBD F12 F2 F6
I5:  ADDD F0 F12 F2
```

# Exe 3.1 Scoreboard: Conflicts

```
I1:  LD F6 32+ R2
I2:  ADDD F2 F6 F4
I3:  MULTD F0 F4 F2
I4:  SUBD F12 F2 F6
I5:  ADDD F0 F12 F2
```

# Exe 3.3 Scoreboard: CC 0

| | Instruction | ISSUE | READ OPERAND | EXE COMPLETE | WB | Hazards | Unit |
|---|---|---|---|---|---|---|---|
| I1 | LD F6 32+ R2 | | | | | | |
| I2 | ADDD F2 F6 F4 | | | | | | |
| I3 | MULTD F0 F4 F2 | | | | | | |
| I4 | SUBD F12 F2 F6 | | | | | | |
| I5 | ADDD F0 F12 F2 | | | | | | |

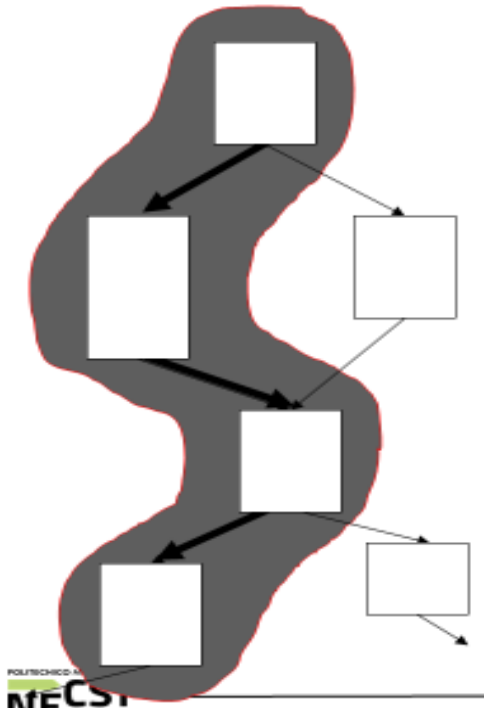
| F0 | F2 | F4 | F6 | F8 | F10 | F12 | ... | F30 |
|---|---|---|---|---|---|---|---|---|
| P0 | P2 | P4 | P6 | P8 | P10 | P12 | ... | P30 |

Initialized Rename Table – registers from P32 in the free list

4 FPALU 3 cc latency, single write port for the pool
1 MEM 2 cc latency

# Exe Scoreboard

## Trace Scheduling *[ Fisher,Ellis]*



Pick string of basic blocks, a trace, that represents most frequent branch path

Use profiling feedback or compiler heuristics to find common branch paths

Schedule whole "trace" at once

Add fixup code to cope with branches jumping out of trace

POLITECNICO MILANO 1863

10

[Parallel operation in the control data 6600]

# Recall: Exe Scoreboard

**3 FPU, latency 2 cc**
**4 LDU, latency 4 cc**

| Instruction | ISSUE | READ OPERAND | EXE COMPLETE | WB | Unit |
|---|---|---|---|---|---|
| I1: `lw $f1, 0($r0)` | 1 | 2 | 8 | 9 | LDU1 |
| I2: `faddi $f1, $f1, C1` | 10 | 11 | 13 | 14 | FPU1 |
| I3: `faddi $f2, $f1, C2` | 11 | 15 | 17 | 18 | FPU2 |
| I4: `sw $f2, 0($r0)` | 12 | 19 | 23 | 24 | LDU2 |
| I5: `lw $f2, 4($r0)` | 19 | 20 | 29 | 30 | LDU3 |
| I6: `fadd $f2, $f2, $f2` | 31 | 32 | 34 | 35 | FPU3 |
| I7: `sw $f2, 4($r0)` | 32 | 36 | 40 | 41 | LDU4 |

**I1 cache miss**
→ **Penalty of 2 cc**
**I5 cache miss**
→ **Penalty 5 cc**

**RAW f1 I1-I2**
**RAW f1 I2-I3**
**RAW f2 I3-I4**
**RAW f2 I5-I6**

**RAW f2 I6-I7**
**WAW f2 I5-I6**
**WAW f2 I5-I3**
**WAW f1 I2-I1**

**WAR f2 I4-I5**
**WAR f2 I4-I6**

# Exe Scoreboard

**3 FPU, latency 2 cc**
**4 LDU, latency 4 cc**

| Instruction | ISSUE | READ OPERAND | EXE COMPLETE | WB | Unit |
|---|---|---|---|---|---|
| I1:lw $f1, 0($r0) | 1 | 2 | 8 | 9 | LDU1 |
| I2:faddi $f1, $f1, C1 | 10 | 11 | 13 | 14 | FPU1 |
| I3:faddi $f2, $f1, C2 | 11 | 15 | 17 | 18 | FPU2 |
| I4:sw $f2, 0($r0) | 12 | 19 | 23 | 24 | LDU2 |
| I5:lw $f2, 4($r0) | 19 | 20 | 29 | 30 | LDU3 |
| I6:fadd $f2, $f2, $f2 | 31 | 32 | 34 | 35 | FPU3 |
| I7:sw $f2, 4($r0) | 32 | 36 | 40 | 41 | LDU4 |

# What if Explicit Register Renaming?

I1 cache miss
→ Penalty of 2 cc
I5 cache miss
→ Penalty 5 cc

RAW f1 I1-I2
RAW f1 I2-I3
RAW f2 I3-I4
RAW f2 I5-I6

RAW f2 I6-I7
WAW f2 I5-I6
WAW f2 I5-I3
WAW f1 I2-I1

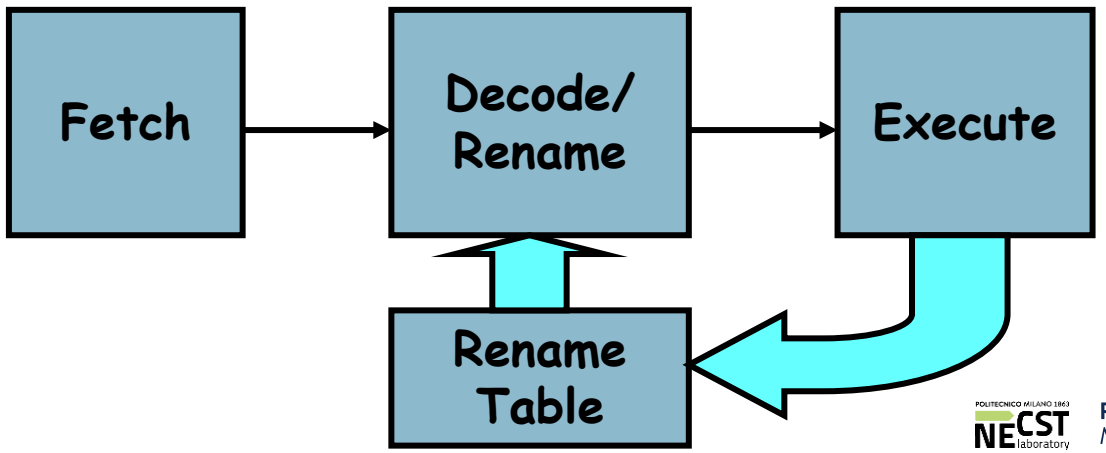
WAR f2 I4-I5
WAR f2 I4-I6

# Recall: Explicit Register Renaming

```
ld x1, (x3)          ld P1, (Px)
addi x3, x1, #4      addi P2, P1, #4
sub x6, x7, x9       sub P3, Py, Pz
add x3, x3, x6       add P4, P2, P3
ld x6, (x1)          ld P5, (P1)
add x6, x6, x3       add P6, P5, P4
sd x6, (x1)          sd P6, (P1)
ld x6, (x11)         ld P7, (Pw)
```

**Physical RF**

| | |
|---|---|
| P0 | 32/64 bitS |
| P1 | |
| P2 | |
| P3 | |
| ... | |
| P62 | |
| P63 | |



# Exe Scoreboard+ Explicit RR

**3 FPU, latency 2 cc**
**4 LDU, latency 4 cc**

| Instruction | ISSUE | READ OPERAND | EXE COMPLETE | WB | Unit |
|---|---|---|---|---|---|
| I1: `lw $f1, 0($r0)` | 1 | 2 | 8 | 9 | LDU1 |
| I2: `faddi $f1, $f1, C1` | 2 | 10 | 12 | 13 | FPU1 |
| I3: `faddi $f2, $f1, C2` | 3 | 14 | 16 | 17 | FPU2 |
| I4: `sw $f2, 0($r0)` | 4 | 18 | 22 | 23 | LDU2 |
| I5: `lw $f2, 4($r0)` | 5 | 6 | 15 | 16 | LDU3 |
| I6: `fadd $f2, $f2, $f2` | 6 | 17 | 19 | 20 | FPU3 |
| I7: `sw $f2, 4($r0)` | 7 | 21 | 25 | 26 | LDU4 |

**I1 cache miss**
→ **Penalty of 2 cc**
**I5 cache miss**
→ **Penalty 5 cc**

**RAW f1 I1-I2**
**RAW f1 I2-I3**
**RAW f2 I3-I4**
**RAW f2 I5-I6**

**RAW f2 I6-I7**
**WAW f2 I5-I6**
**WAW f2 I5-I3**
**WAW f1 I2-I1**

**WAR f2 I4-I5**
**WAR f2 I4-I6**

# Recall: HW-based Speculation

HW-based Speculation combines 3 ideas:

- **Branch Prediction** to choose which instruction to execute

- **Dynamic Scheduling** supporting out-of-order execution but in-order commit to prevent any irrevocable actions (such as register update or taking exception) until an instruction commits

- **Speculation** to execute instructions before control dependencies are resolved

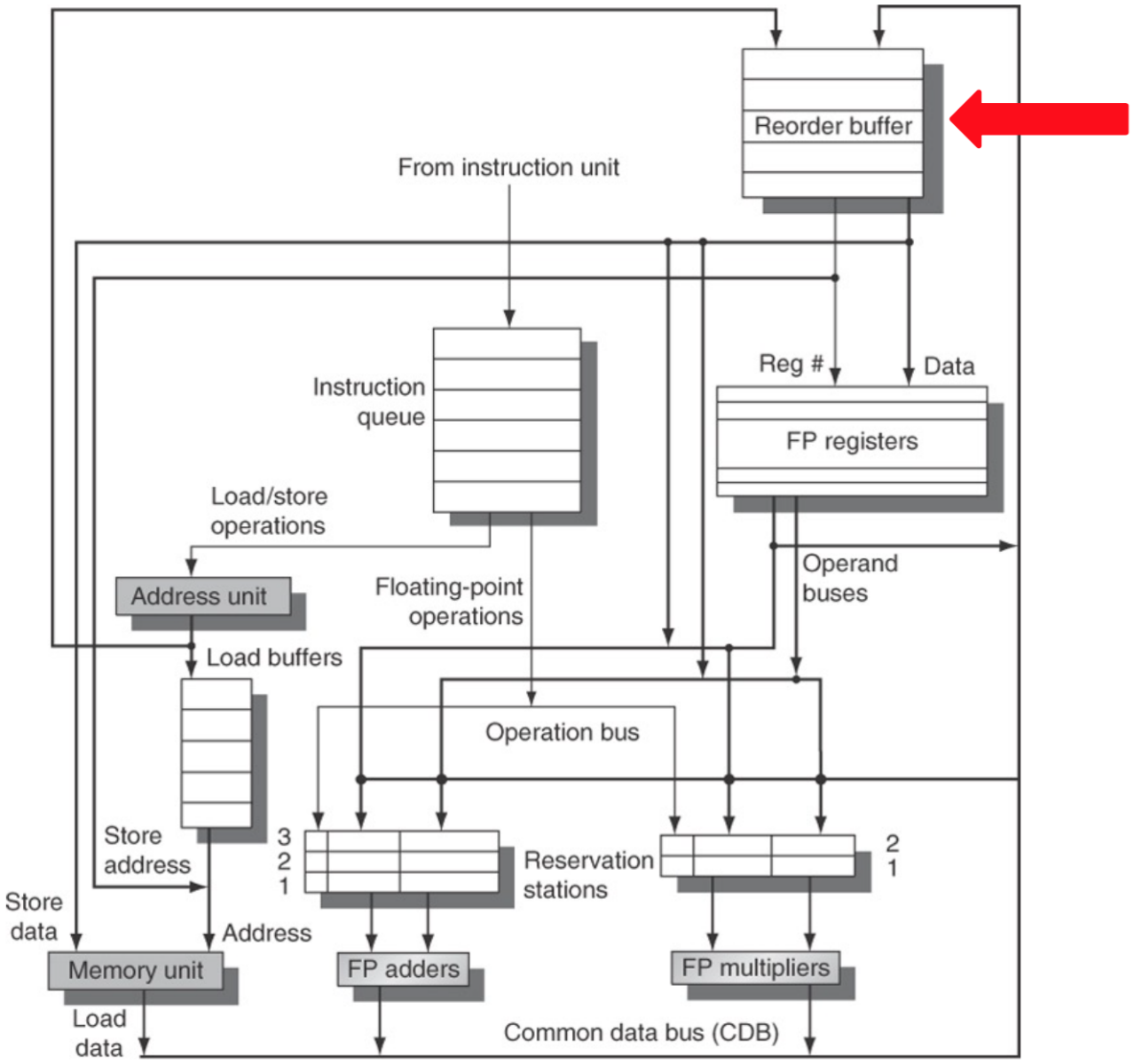
The key idea behind speculation is:

to issue and execute instructions dependent on a branch **before** the branch outcome is known;

to allow instructions to **execute out-of-order** but to force them to **commit in-order;**

to prevent any irrevocable action (such as updating state or taking an exception) until an instruction commits;

# Recall: Tomasulo with Reorder Buffer



# Recall: Separating Completion from Commit

Re-order buffer holds register results from completion until commit

- Entries allocated in program order during decode
- Buffers completed values and exception state until in-order commit point

- Completed values can be used by dependents before committed (bypassing)
- Each entry holds program counter, instruction type, destination register specifier and value if any, and exception status (info often compressed to save hardware)

Memory reordering needs special data structures

- Speculative store address and data buffers
- Speculative load address and data buffers

# Recall: Relationship between precise interrupts and speculation

Speculation: guess and check

Important for branch prediction:

Need to "take our best shot" at predicting branch direction

If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly:

This is exactly the same as precise exceptions!

Technique for both precise interrupts/exceptions and speculation: *in-order completion or commit*

## Precise Interrupts

*It must appear as if an interrupt is taken between two instructions* (say $I_i$ and $I_{i+1}$)

- the effect of all instructions up to and including $I_i$ is totally complete
- no effect of any instruction after $I_i$ has taken place

The interrupt handler either aborts the program or restarts it at $I_{i+1}$.

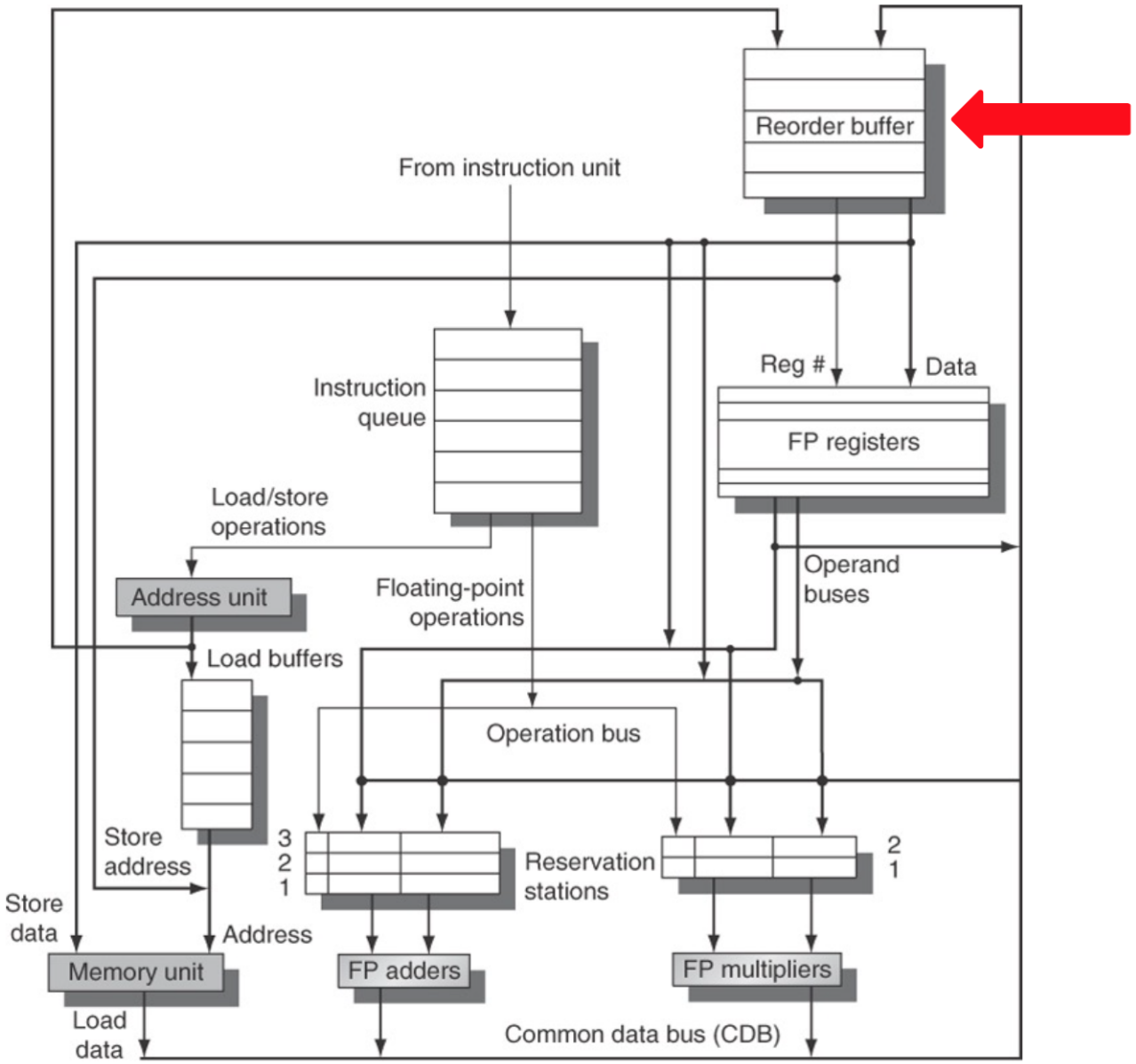
## Handling Exceptions

Exceptions are handled by not recognizing the exception until instruction that caused it is ready to commit in ROB

If a speculated instruction raises an exception, the exception is recorded in the ROB

This is why reorder buffers in all new processors

# Exe Tomasulo with Reorder Buffer



# Recall: the Tomasulo pipeline

| ISSUE | EXECUTION | WRITE |
|---|---|---|
| Get Instruction from Queue and Rename Registers | Execute and Watch CDB; | Write on CDB; |
| Structural RSs check; WAW and WAR solved by Renaming (!!!in-order-issue!!!); | Check for Struct on FUs; RAW delaying; Struct check on CDB; | (FUs will hold results unless CDB free) RSs/FUs marked free |

# Recall: the Tomasulo pipeline with ROB

| ISSUE | EXECUTION | WRITE | COMMIT |
|---|---|---|---|
| Get Instruction from Queue and Rename Registers<br>Add Instruction to ROB | Execute and Watch CDB; | Write on CDB;<br>Write on ROB | Update register with result (or store to memory);<br>remove Instr from ROB |
| Structural RSs check;<br>Structural ROB check;<br>WAW and WAR solved by Renaming (!!!in-order-issue!!!); | Check for Struct on FUs;<br>RAW delaying;<br>Struct check on CDB; | (FUs will hold results unless CDB free)<br>RSs/FUs marked free | In-order commit;<br>Mispredicted branch flushes ROB |

# Exe 4 Tomasulo with ROB: the Code

```
LOOP:I1: MULTD F2, F6, F8
     I2: ADDD F0, F6, F4
     I3: MULTD F10, F0, F2
     I4: ADDD F0, F12, F14
     I5: SUBI R0, R0, 4
     I6: BNEZ R0, LOOP
```

# Exe 4 Tomasulo with ROB: the Conflicts

```
LOOP:I1: MULTD F2, F6, F8
     I2: ADDD F0, F6, F4
     I3: MULTD F10, F0, F2
     I4: ADDD F0, F12, F14
     I5: SUBI R0, R0, 4
     I6: BNEZ R0, LOOP
```

# Exe.4 Tomasulo with ROB

- 3 RESERVATION STATIONS (RS1, RS2, RS3) + 3 MULT unit (MULT1,MULT2, MULT3) with latency 4
- 3 RESERVATION STATIONS (RS4, RS5, RS6) + 3 ADDD/SUBD (ADDD1, ADDD2, ADDD3) with latency 2
- 7-slot ROB
- Enough RS and FUs for integer operations, and separated CDB

To be clear, will show when the SUBI and BNEZ are issued

In the case of hazard on CDB, the oldest instruction has priority

**Let's assume that we discover the BNEZ misprediction 5 clock cycles after the issue**

# Exe Tomasulo with ROB CC0

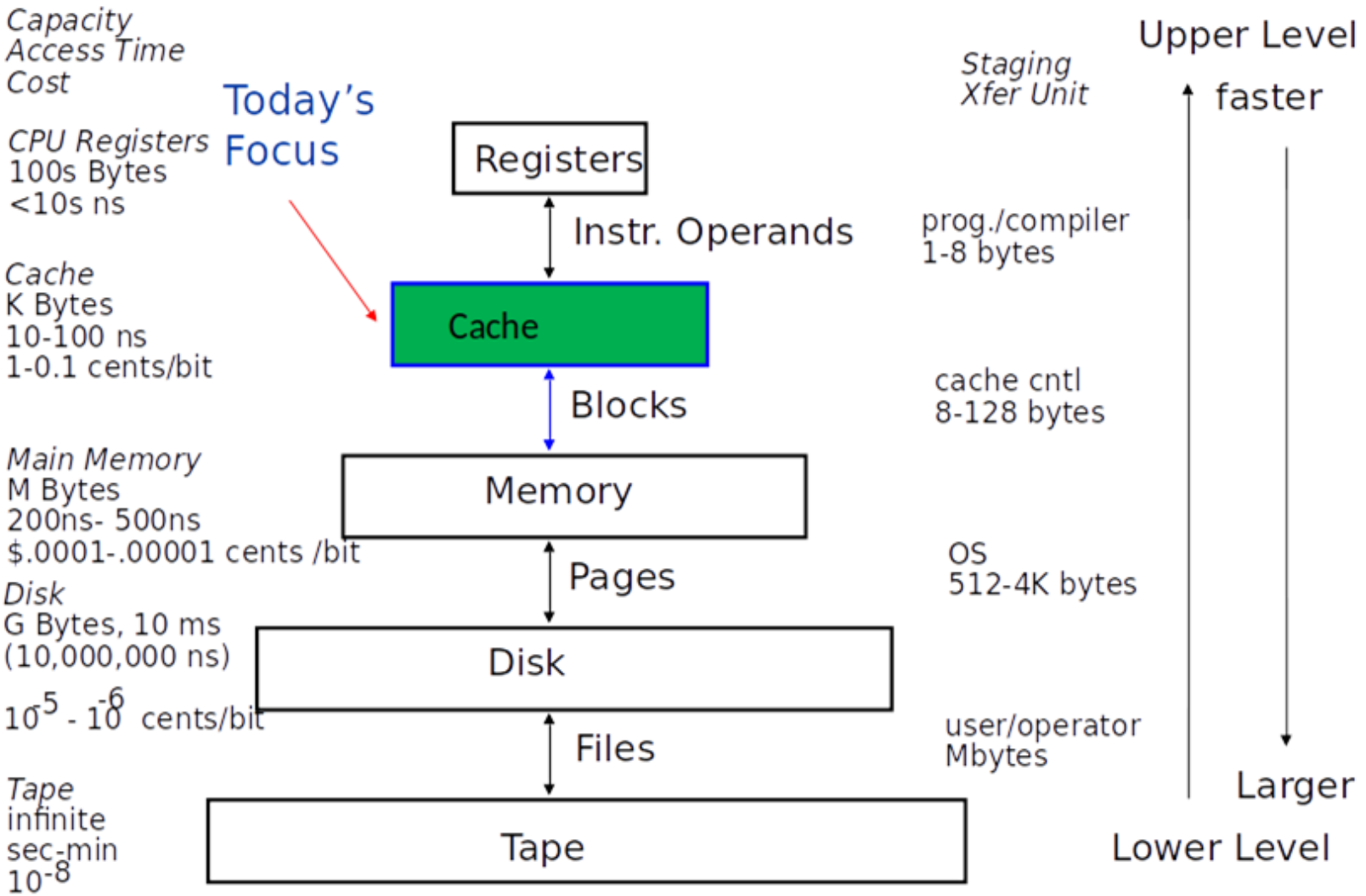
- 3 RESERVATION STATIONS (RS1, RS2, RS3) + 3 MULT unit (MULT1,MULT2, MULT3) with latency 4
- 3 RESERVATION STATIONS (RS4,RS5, RS6) + 3 ADDD/SUBD (ADDD1, ADDD2, ADDD3) with latency 2
- 7-slot ROB

| Instruction | ISSUE | START EXE | WB | Commit | ROB idx | Hazards Type | RSi | Unit |
|---|---|---|---|---|---|---|---|---|
| It.1: MULTD F2, F6, F8 | | | | | | | | |
| It.1: ADDD F0, F6, F4 | | | | | | | | |
| It.1: MULTD F10, F0, F2 | | | | | | | | |
| It.1: ADDD F0, F12, F14 | | | | | | | | |
| It.2: MULTD F2, F6, F8 | | | | | | | | |
| It.2: ADDD F0, F6, F4 | | | | | | | | |
| It.2: MULTD F10, F0, F2 | | | | | | | | |
| It.2: ADDD F0, F12, F14 | | | | | | | | |

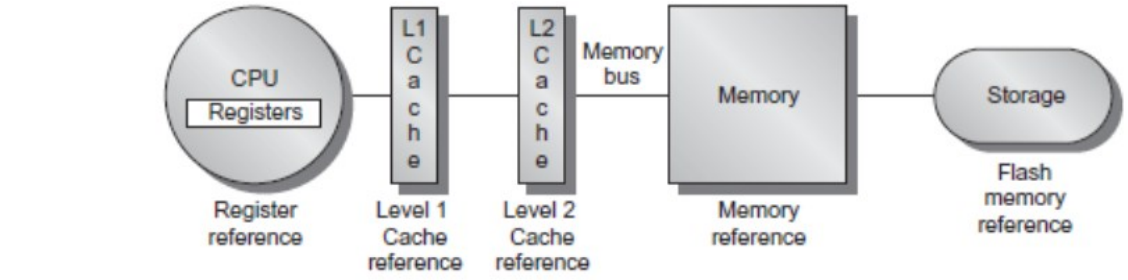
# Recall: Locality Principles

- 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, induction variables
- **Spatial** locality
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

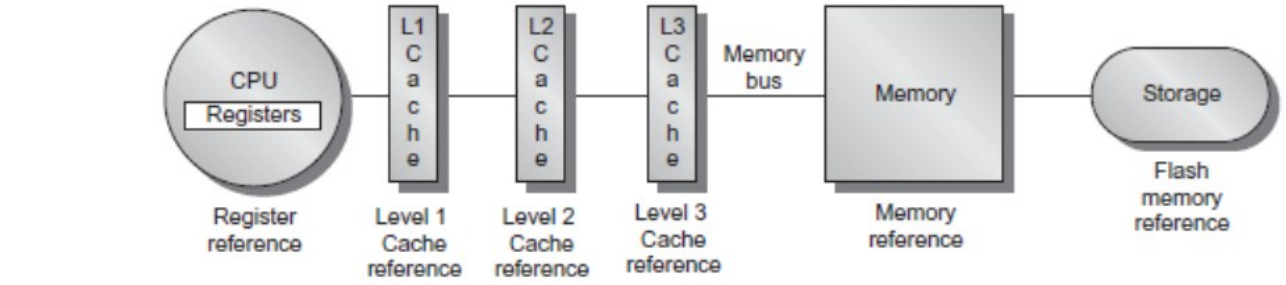
# Recall: Memory hierarchy



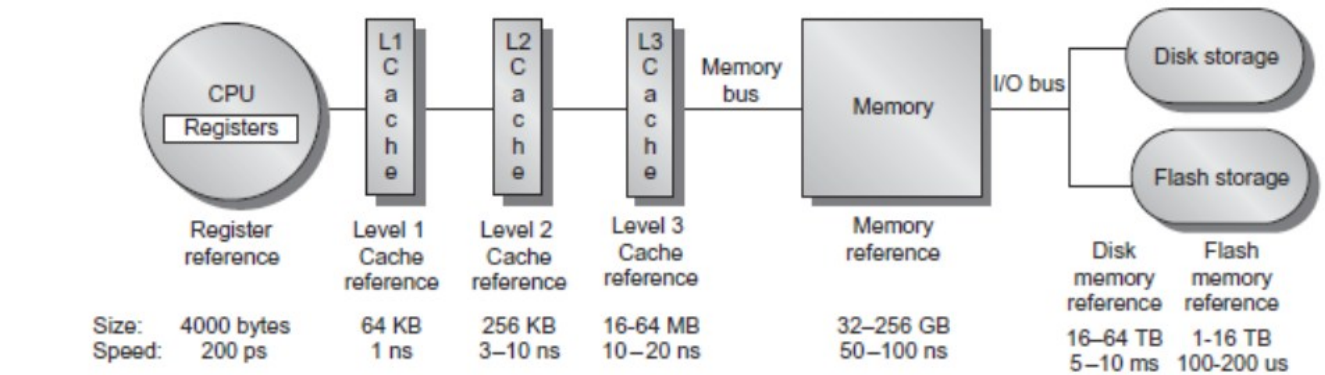
# Recall: Hierarchy




(A) Memory hierarchy for a personal mobile device



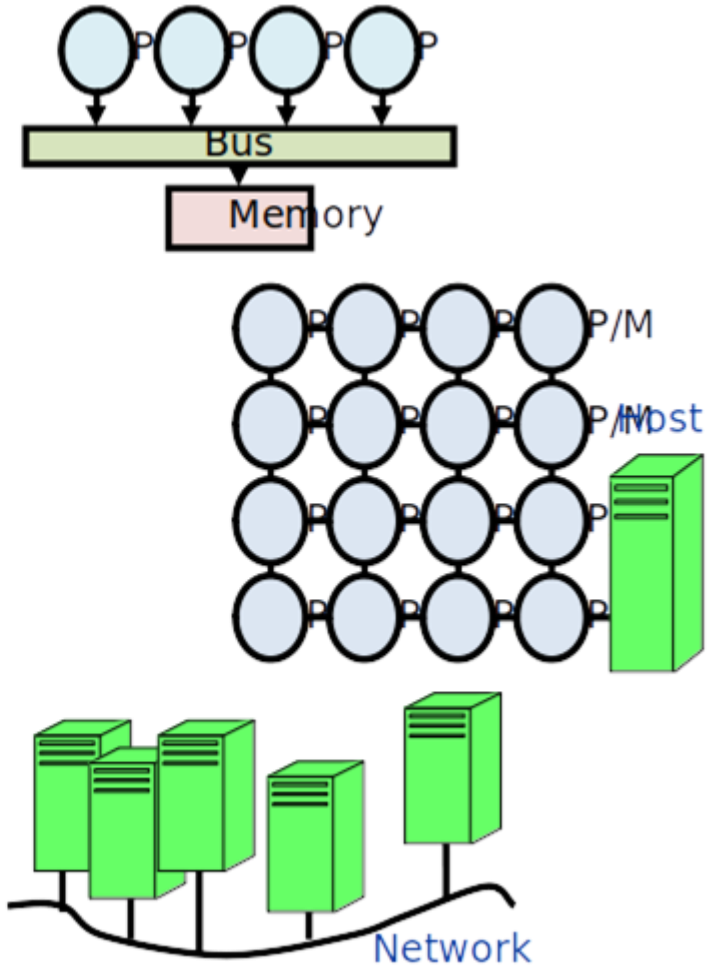

(B) Memory hierarchy for a laptop or a desktop




(C) Memory hierarchy for server

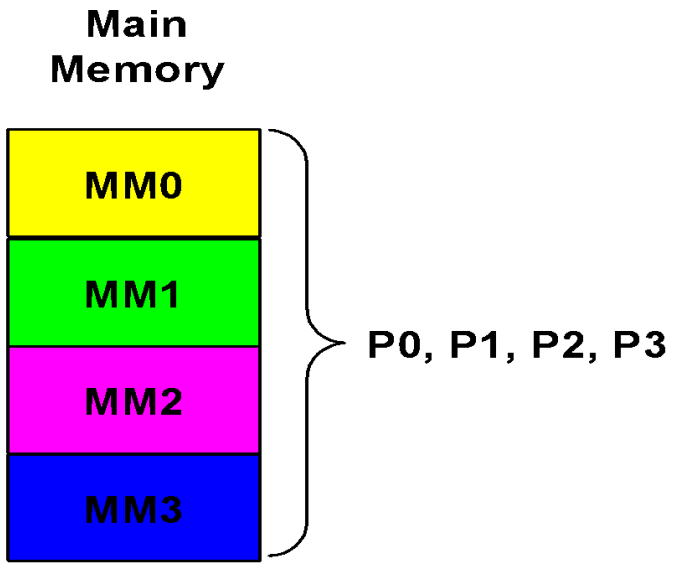
# Recall: MIMD Machines

- Symmetric Multiprocessor
  - Multiple processors in box with shared memory communication
  - Current MultiCore chips like this
  - Every processor runs copy of OS
- Non-uniform shared-memory with separate I/O through host
  - Multiple processors
    - Each with local memory
    - general scalable network
  - Extremely light "OS" on node provides simple services
    - Scheduling/synchronization
  - Network-accessible host for I/O
- Cluster
  - Many independent machine connected with general network
  - Communication through messages

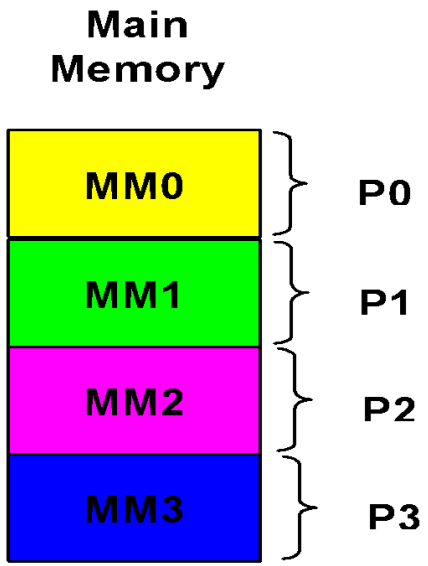


# Recall: **Memory Address Space Model**

Single logically shared address space



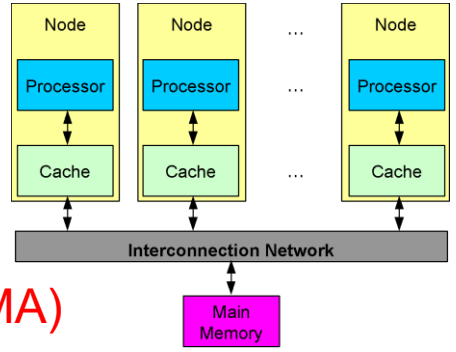

Multiple and private address spaces



# Recall: **Physical Memory Organization**

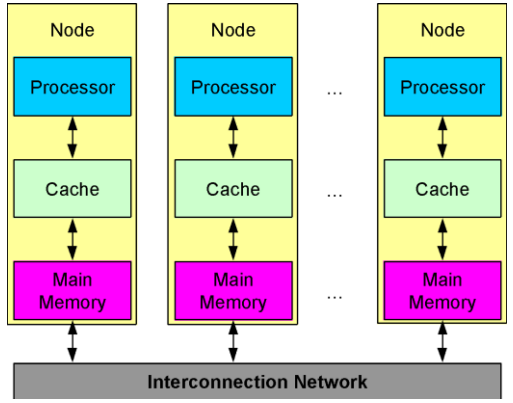
## Centralized shared-memory architectures

- at most few dozen processor chips (< 100 cores)
- Large caches, single memory multiple banks
- Often called symmetric multiprocessors (SMP) and the style of architecture called Uniform Memory Access (UMA)



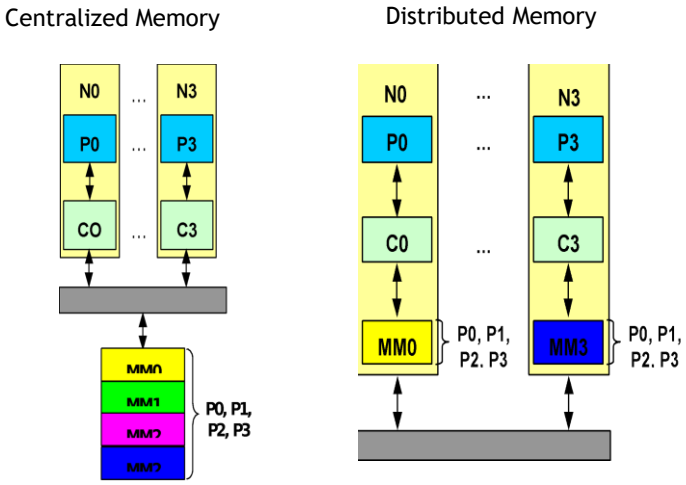

## Distributed memory architectures

- To support large processor counts
- Requires high-bandwidth interconnect
- Cons: data communication among processors
- Non Uniform Memory Access (NUMA)

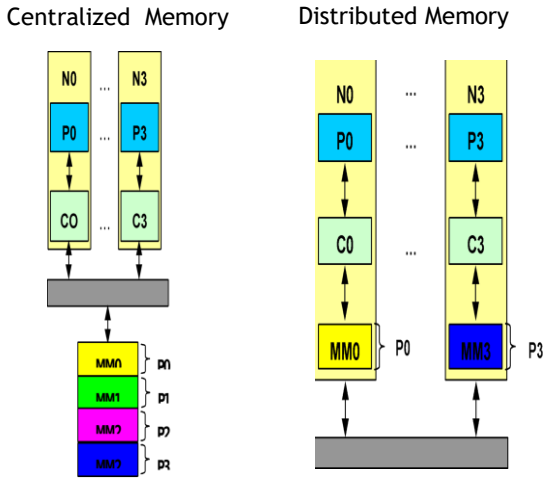


# Recall: Address Space vs. Physical Memory Org.

## Single Logically Shared Address Space (Shared-Memory Architectures)

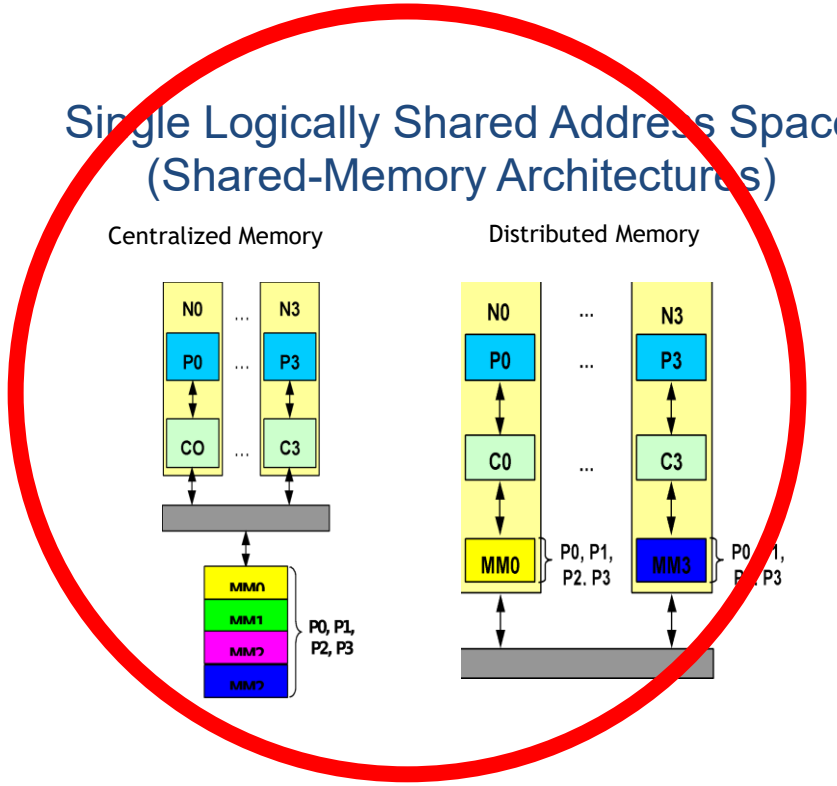



## Multiple and Private Address Space (Message Passing Architectures)

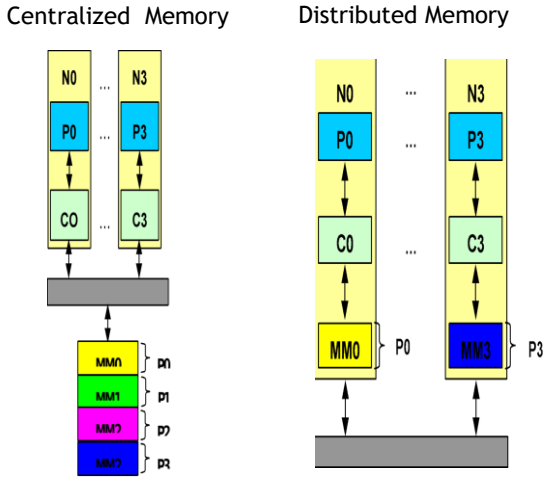


# Recall: Address Space vs. Physical Memory Org.

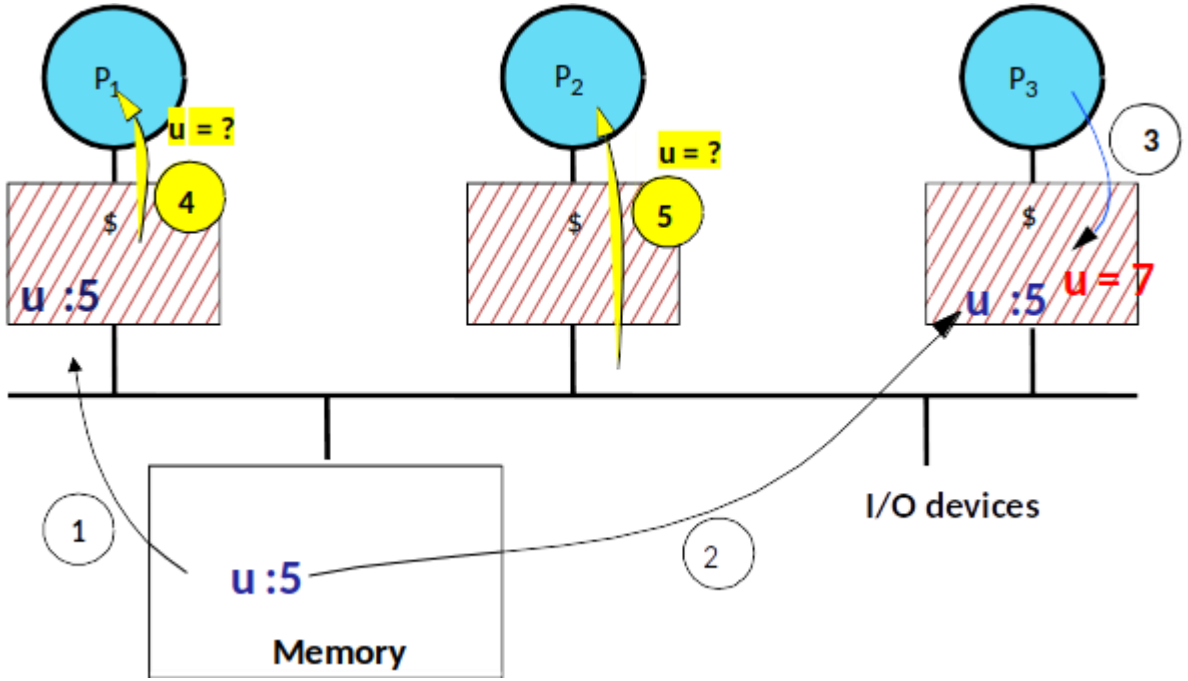
## Single Logically Shared Address Space (Shared-Memory Architectures)




## Multiple and Private Address Space (Message Passing Architectures)



# Recall: Example Cache Coherency Problem



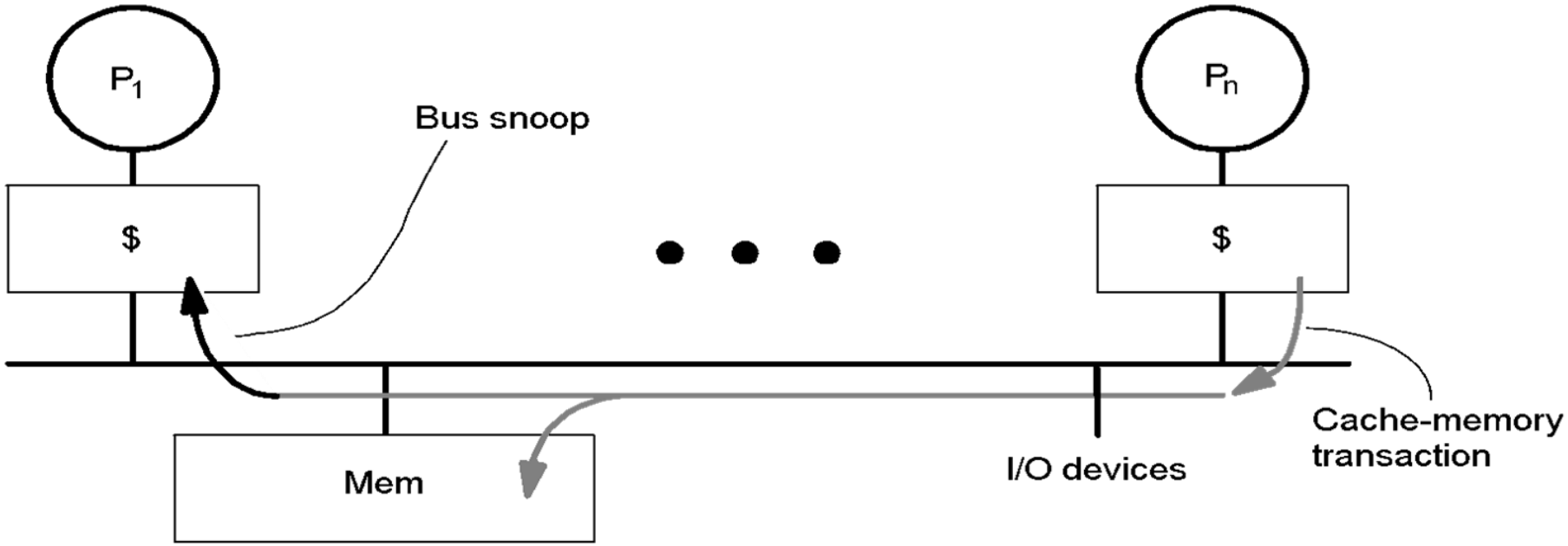

Things to note:

- Processors see different values for u after event 3
- With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value
  - Processes accessing main memory may see very stale value
- Unacceptable to programs, and frequent!

# Recall: **Potential Solutions**

- HW-based solutions to maintain coherency: Cache-Coherence Protocols
- Key issues to implement a cache coherent protocol in multiprocessors is tracking the status of any sharing of a data block.
- Two classes of protocols:
  - **Snooping Protocols**
  - **Directory-Based Protocols**

# Recall: **Snooping protocols**



# Recall: Basic Snooping Protocols

Snooping Protocols are of two types depending on what happens on a write operation:

- Write-Invalidate Protocol
- Write-Update or Write-Broadcast Protocol

# Recall: MESI Protocol

MESI Protocol: Write-Invalidate
Each cache block can be in one of four states:

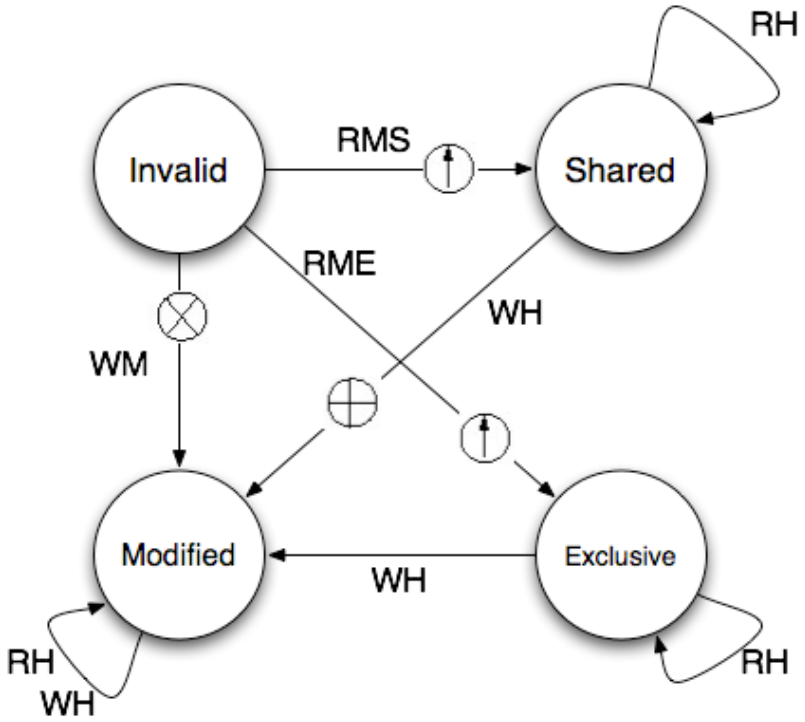
- Modified: the block is dirty and cannot be shared; cache has only copy, its writeable.
- Exclusive: the block is clean and cache has only copy;
- Shared: the block is clean and other copies of the block are in cache;
- Invalid: block contains no valid data

Add exclusive state to distinguish exclusive (writable) and owned (written)

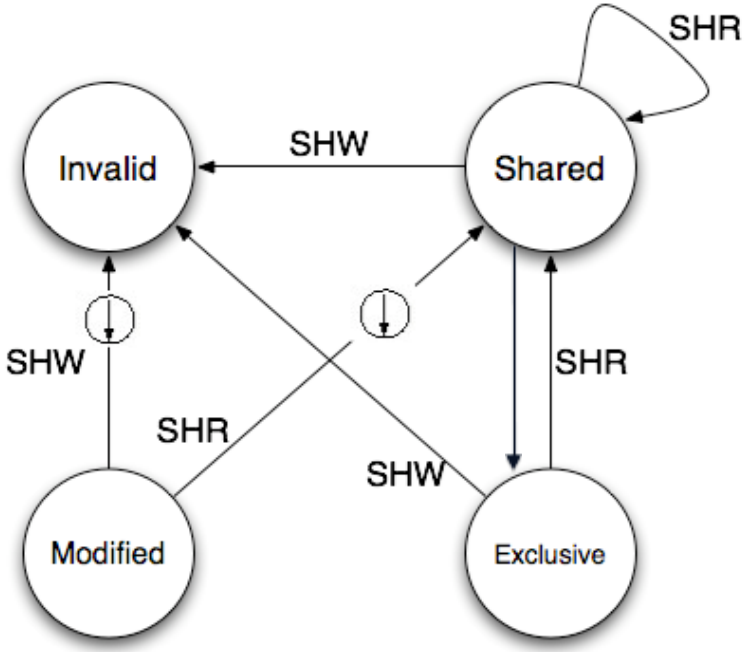


61

# Recall: MESI State Transition Diagram




Cache line in the "acting" processor




Transaction due to events snooped on the common BUS

BUS Transactions

RH = Read Hit
RMS = Read Miss, Shared
RME = Read Miss, Exclusive
WH = Write Hit
WM = Write Miss
SHR = Snoop Hit on a Read
SHW = Snoop Hit on a Write or Read-with-Intent-to-Modify

= Snoop Push
= Invalidate Transaction
= Read-with-Intent-to-Modify
= Cache Block Fill



62

# Exe1: Cache Coherency

Consider the following access pattern on a two-processor system with a direct-mapped, write-back cache with **one cache block and a two cache block memory.**

Assume the MESI protocol is used, with **write-back caches, write-allocate, and invalidation** of other caches on write (instead of updating the value in the other caches).

# Recall: Write Policy

- Write-through: all writes update cache and underlying memory/cache
  - Can always discard cached data - most up-to-date data is in memory
  - Cache control bit: only a *valid* bit
- Write-back: all writes simply update cache
  - Can't just discard cached data - may have to write it back to memory
  - Cache control bits: both *valid* and *dirty* bits

- How to identify the most recent data value of a cache block in case of cache miss?
  - It can be in a cache rather in a memory
- Can use the same snooping scheme both for cache misses and writes
  - Each processor snoops every address placed on the bus
  - If a processor finds that it has a dirty copy of the requested cache block, it provides the cache block in response to the read request
  - memory access is aborted

# Recall: Write Policy

- **What happens on write miss?**
- Write allocate: allocate new cache line in cache
  - Usually means that you have to do a “read miss” to fill in rest of the cache-line!
  - Alternative: per/word valid bits
- Write non-allocate (or “write-around”):
  - Simply send write data through to underlying memory/cache - don’t allocate new cache line!

# Exe 1: Cache Coherency

| Cycle | After Operation | P0 cache block state | P1 cache block state | Memory at block 0 up to date? | Memory at block 1 up to date? |
|---|---|---|---|---|---|
| 0 | P0: read block 1 | Exclusive (1) | Invalid | Yes | Yes |
| 1 | P1: read block 0 | | | | |
| 2 | P0: write block 1 | | | | |
| 3 | P0: write block 0 | | | | |
| 4 | P1: read block 0 | | | | |
| 5 | P1: write block 0 | | | | |
| 6 | P0: read block 1 | | | | |
| 7 | P1: read block 1 | | | | |
| 8 | P0: write block 1 | | | | |
| 9 | P1: write block 1 | | | | |
| 10 | P0: read block 0 | | | | |
| 11 | P1: write block 1 | | | | |
| 12 | P1: read block 1 | | | | |
| 13 | P0: read block 1 | | | | |
| 14 | P1: write block 1 | | | | |

# Thanks for your attention

Davide Conficconi <davide.conficconi@polimi.it>

**Acknowledgements**

E. Del Sozzo, Marco D. Santambrogio, D. Sciuto

Part of this material comes from:

- “Computer Organization and Design” and “Computer Architecture A Quantitative Approach” Patterson and Hennessy books
- “Digital Design and Computer Architecture” Harris and Harris
- Elsevier Inc. online materials
- Papers/news  cited in this lecture