LECTURE 3

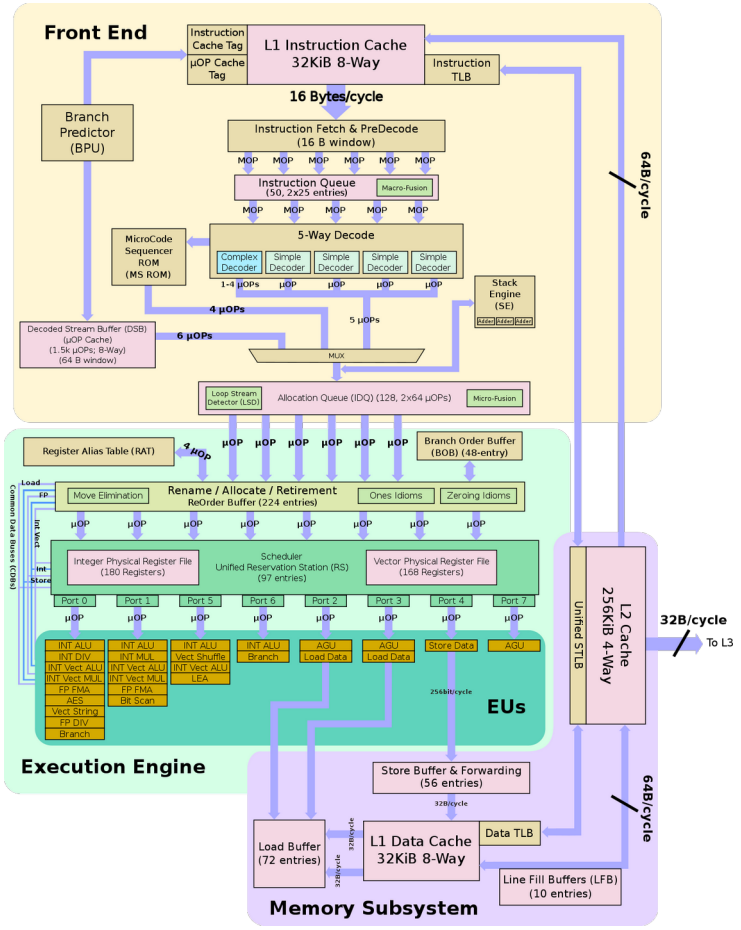
OUT-OF-ORDER EXECUTION AND MULTILEVEL CACHE HIERARCHIES

Miquel Pericàs

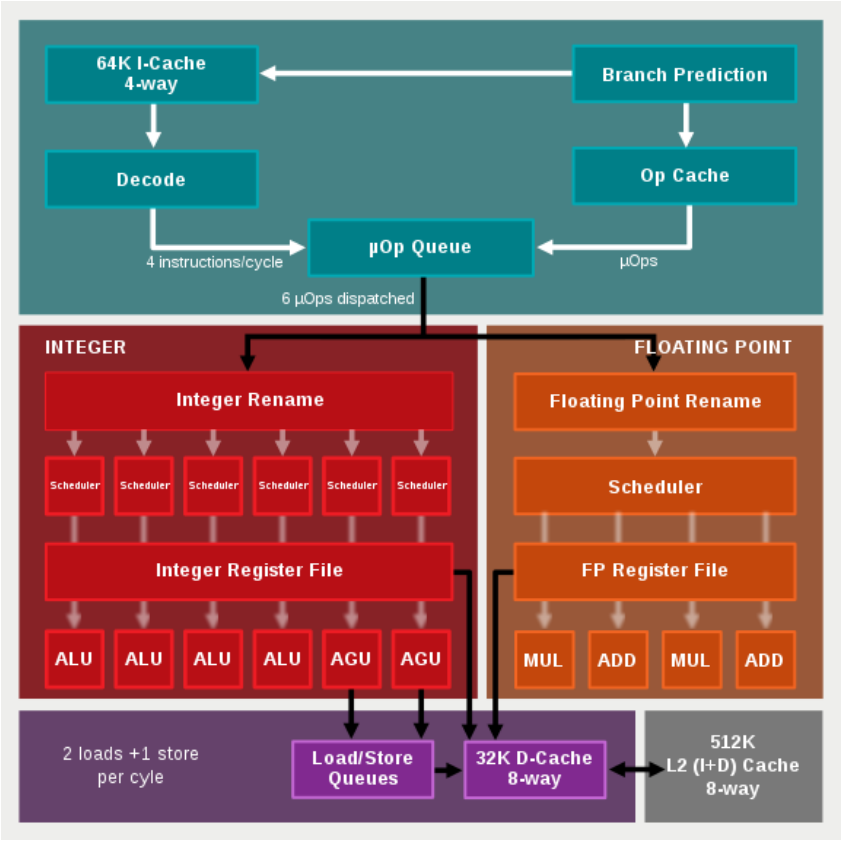
EDA284/DIT361 - 2019/2020 SP3

WHAT IS INSIDE MODERN PROCESSORS?

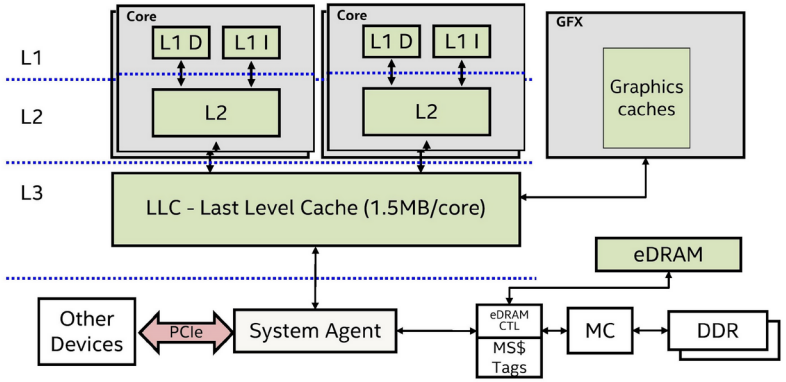
INTEL SKYLAKE



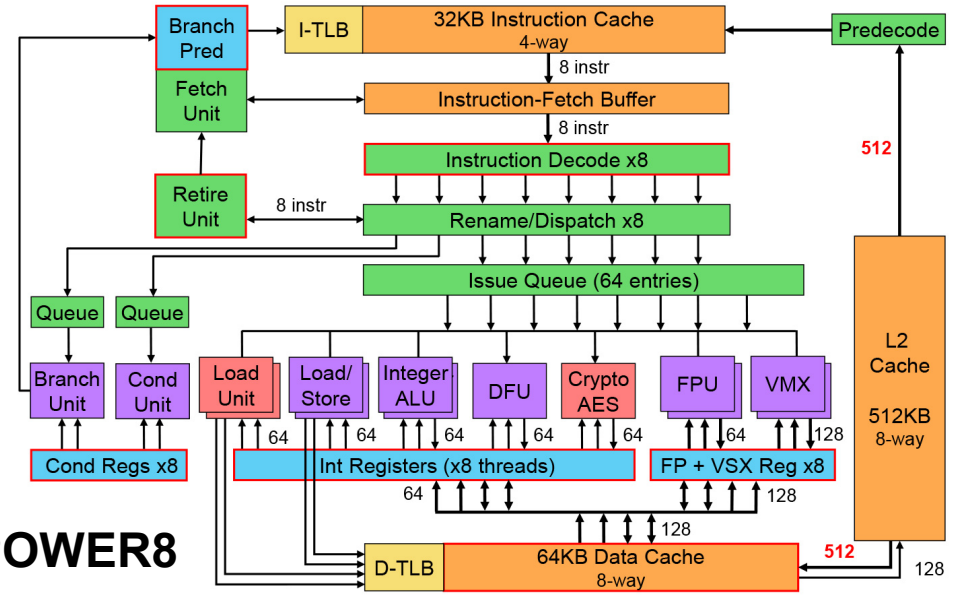
AMD ZEN



Skylake-H (Core i7-6770HQ) Memory Hierarchy



IBM POWER8



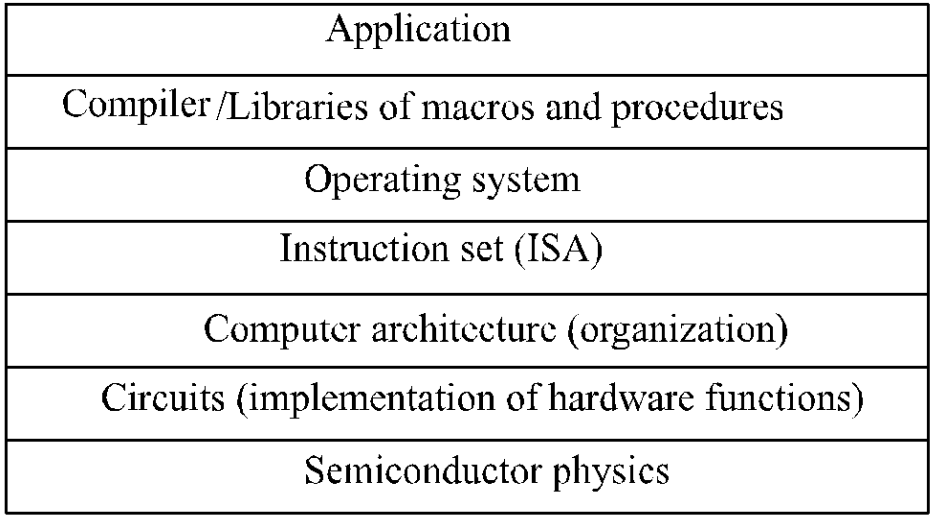
Dynamic Scheduling (OoO Microarchitecture)

- **OUT-OF-ORDER EXECUTION**
  - huge and non-trivial topic
  - for all the details read chapter 3.4 “dynamically scheduled pipelines”.
  - OoO is motivated by limitations of statically scheduled pipelines.
    - recommended reading: chapter 3.3

- **TODAY’S LECTURE:**
  - out-of-order scheduling
  - multi-level caches
  - non-blocking caches

INSTRUCTION SET ARCHITECTURE (ISA)

- **The ISA is the interface between software and hardware**



- **DESIGN OBJECTIVES**
  - functionality and flexibility for OS and compilers
  - implementation efficiency in available technology
  - backward compatibility

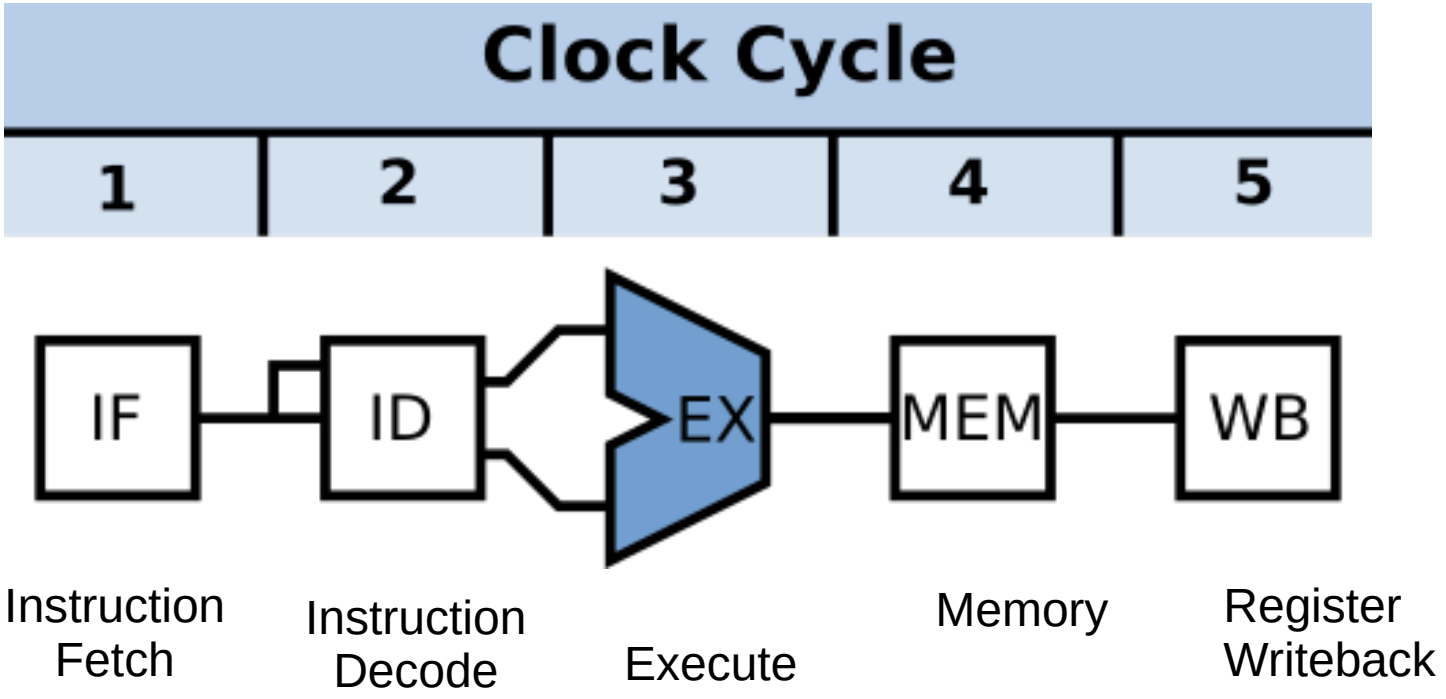
ISAs are typically designed to last through trends of changes in usage and technology.

AS TIME GOES BY THEY TEND TO GROW

(https://software.intel.com/en-us/isa-extensions)

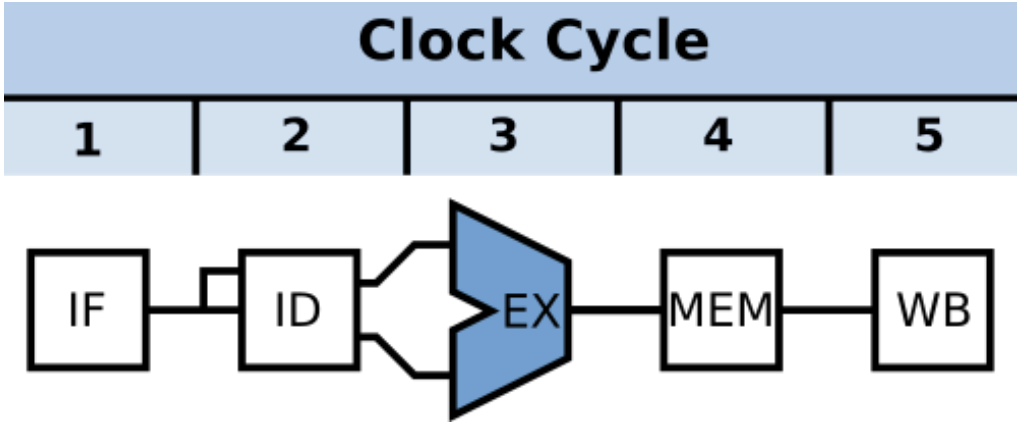
Static Scheduling

- Static Scheduling: In-order instruction dispatch & execution
- Simple & Low Power: All hazards resolved at ID (Instr. Decode)
  - Data Hazards: Only read-after-write (RAW) hazards
  - No Structural Hazards (if no multicycle ops, no superpipelining)
  - Control Hazards: Squash IF+ID stages if branch is taken
- Instruction Scheduling is the job of the compiler



Static Scheduling: Limitations

- Not portable: one schedule for each pipeline
- Cannot handle unpredictable latencies



EX MEM

RAW (read-after-write)

```
In: Op    Dst, Src1, Src2
I1: L.S   F0, 0(R1)  Miss (1+10 cycles)  Exec Cycle: 0
I2: ADD.S F1, F1, F0     (1 cycle)       Exec Cycle: 12
I3: L.S   F0, 0(R2)  Hit  (1+1 cycles)   Exec Cycle: 13
I4: ADD.S F2, F2, F0     (1 cycle)       Exec Cycle: 15
```

- `I3`, `I4` do not depend on `I1`, `I2`, but cannot execute early
- In-order dispatch under-utilizes resources

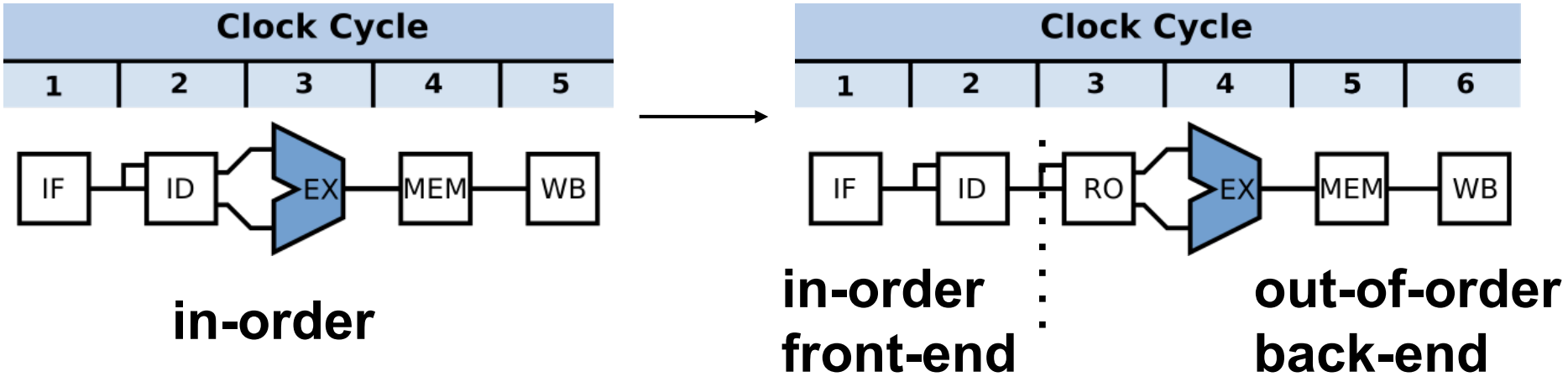
Dynamic Scheduling: The idea

Start execution of instructions in any order that respects the dependencies implied by the program

Instruction scheduling is the job of the processor

Dynamically Scheduled Pipeline

- Split Inst. Decode stage (ID) into two parts:
  - *Dispatch:* Decode + check for structural hazards
  - *Read operands:* Wait until no data hazards, then read operands and proceed to execute
- Dispatch in-order, Read Operands out-of-order



Statically Scheduled

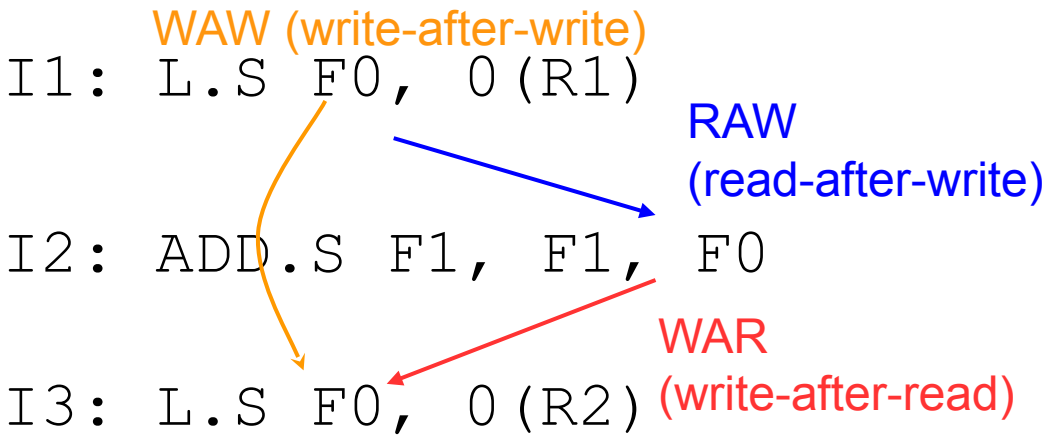
Data Hazards: RAW

Dynamically Scheduled

Data Hazards: RAW + WAR + WAW

Data Hazards (Review)

- Possible Data Hazards with Dynamic Scheduling



- **RAW**: subsequent instruction needs the result of the current instruction (RAW is called *true dependency*)
- **WAR**: subsq. instruction writes to operand of current instruction
- **WAW**: subsq. instruction writes into the same operand as current
- **WAW + WAR** are *name dependencies* (no data is transmitted)
  - *Can be avoided if a different register can be used*

Dynamic Scheduling: Implementations

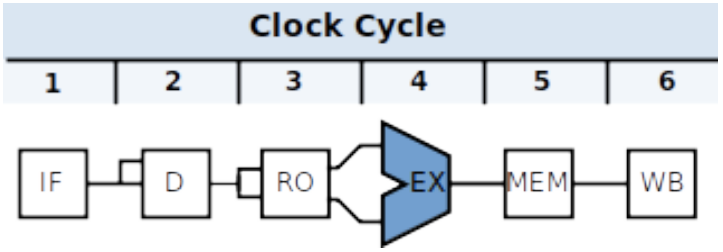
a) Scoreboarding (*not covered*)

- CDC 6600 (1965)
- handles RAW dependencies
- stalls on WAR + WAW hazards

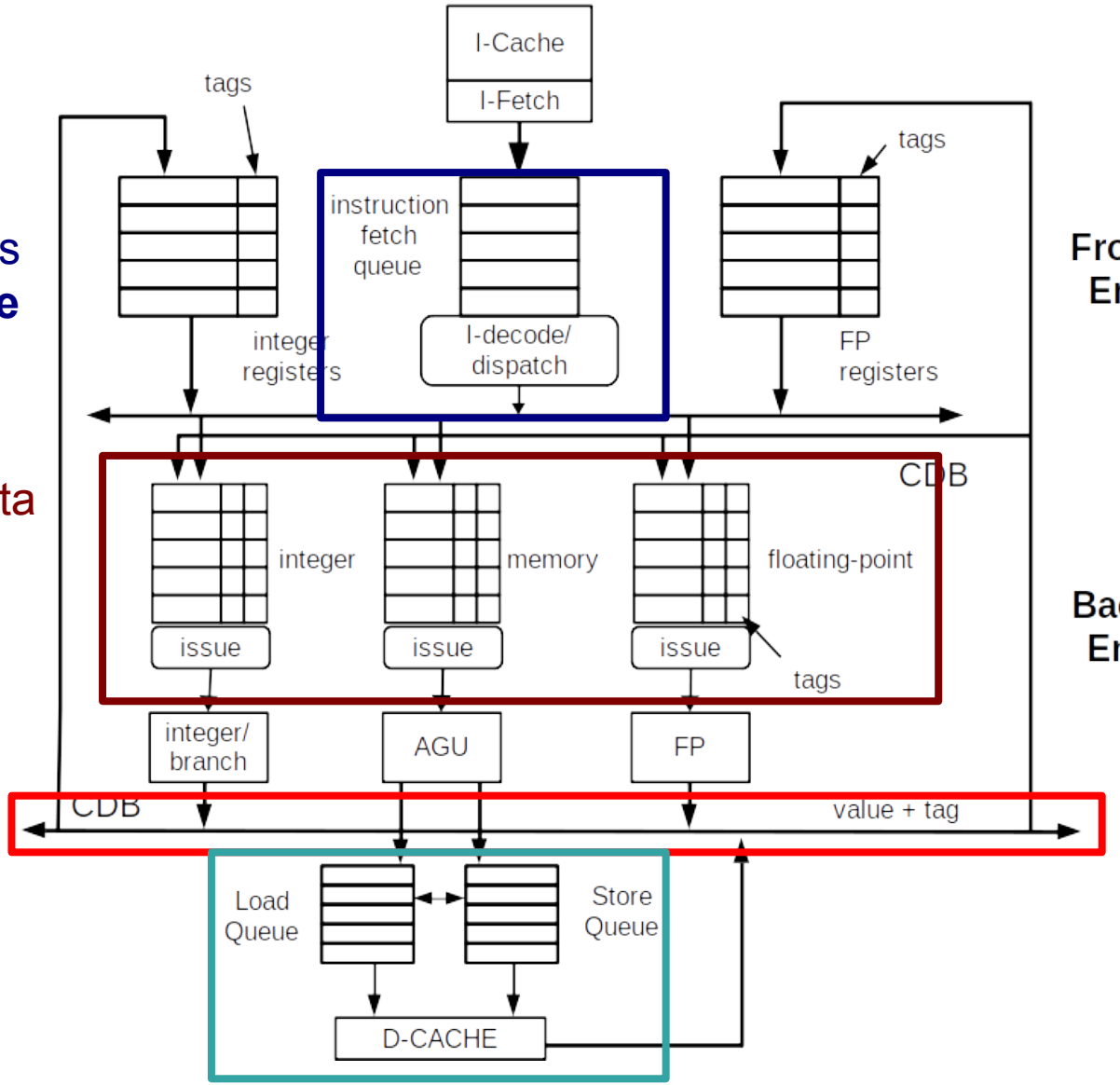
b) **Tomasulo's Algorithm**

- IBM System/360 Model 91 (1967)
- handles RAW dependencies
- overcomes WAW + WAR dependencies (via *register renaming*)

Tomasulo's Algorithm: Overview

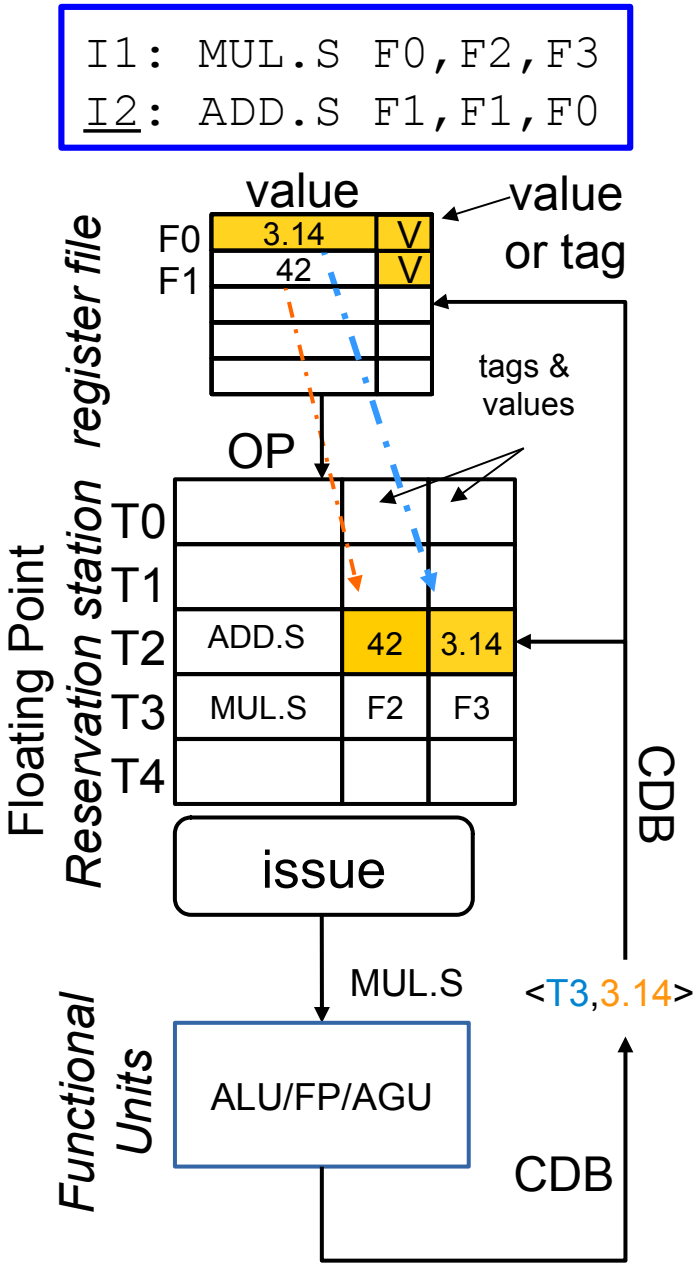


1. Dispatch (D) to Reservation Stations (RS) from **Instruction Fetch Queue** (resolves structural hazards)
2. Wait for operands (RO) in **Reservation Stations** (resolves data hazards)
3. Writeback (WB) results through a **Common Data Bus** (CDB)
4. Manage Memory Hazards (RAW, WAR, WAW) via **Load & Store Queues** (M)



Managing RAW Dependencies: Reservation Stations

1. Read and buffer available operands from Register File (F1=42)
2. Unavailable operands store the #entry (tag) of the reservation station that writes the value (F0=T3)
3. Results+tags are broadcast on the CDB, which is monitored by the reservation stations and RF. Matching tags are updated with corresponding values (<T3, 3.14>)
4. Instructions with all operand values become ready to execute



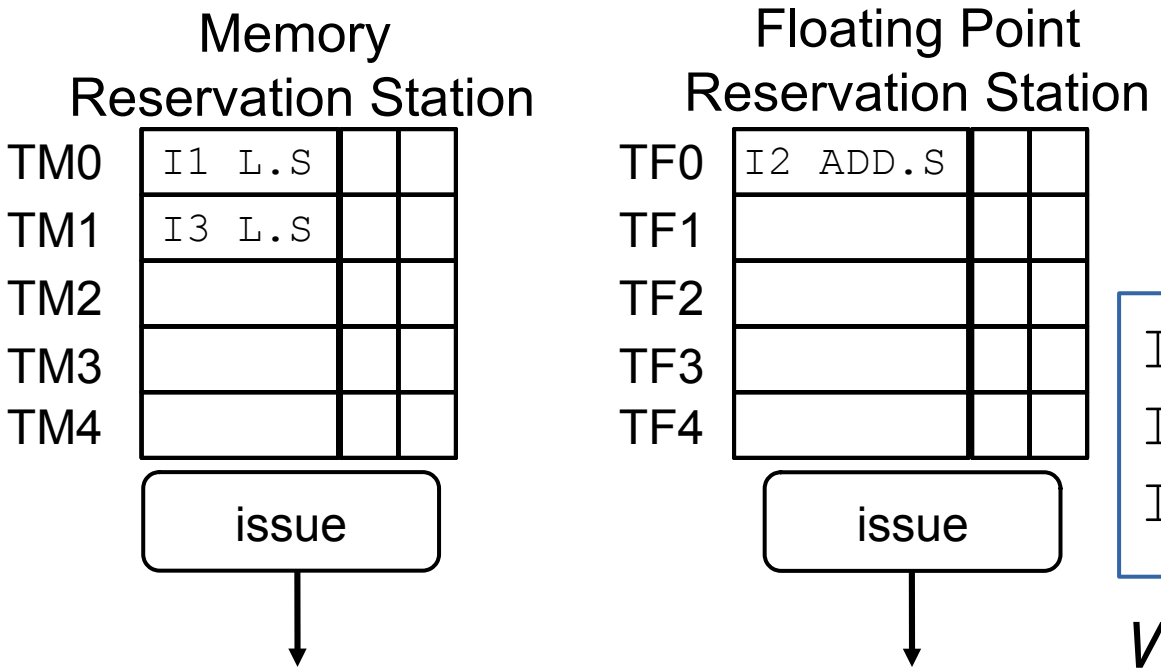
Managing WAR + WAW Dependencies: Register Renaming

WAW (write-after-write)

```
I1: L.S  F0,  0(R1)
I2: ADD.S F1, F1, F0
I3: L.S  F0,  0(R2)
```

RAW (read-after-write)

WAR (write-after-read)



At Dispatch, rename destination register in Register File with RS entry name (= <*tag*>)

```
I1 L.S:    F0 → TM0
I2 ADD.S:  F1 → TF0
I3 L.S:    F0 → TM1
```

```
I1 L.S    <TM0>,  0(R1)
I2 ADD.S  <TF0>,F1,<TM0>
I3 L.S    <TM1>,  0(R2)
```

WAW+WAR hazards eliminated!

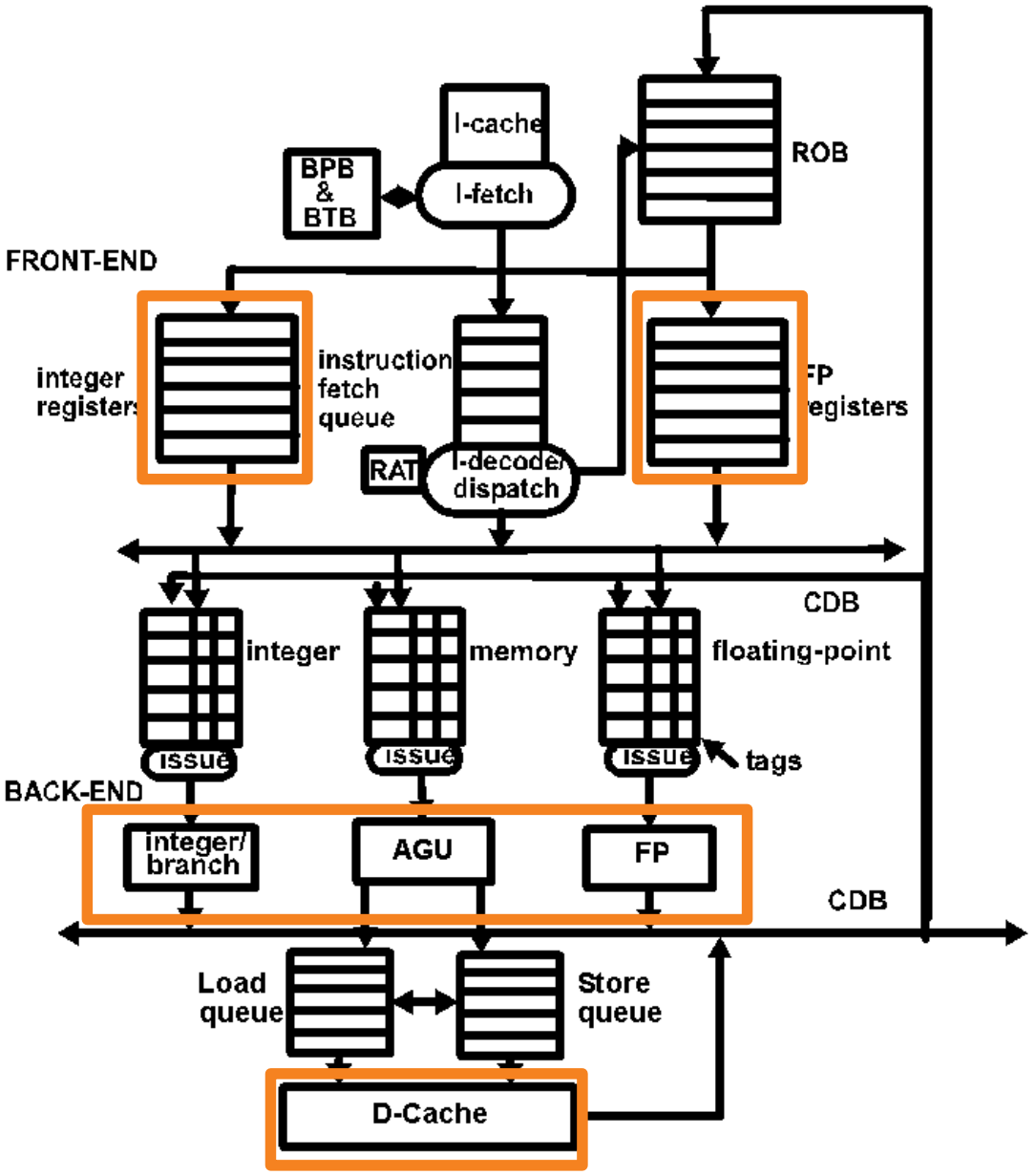
Summary

- **Static scheduling**: Simple & low power, but not portable and does not tolerate unpredictable delays
- **Dynamic Scheduling**: makes observation that a program is just a specification of dependencies that need to be respected (not a schedule!)
  - *Dispatch* in-order: check for structural hazards
  - *Read Operands* out-of-order: respect dependencies

Summary

- **Tomasulo's Algorithm**:
  - Reservation Stations: wait for operands, then execute
  - Register renaming: overcomes name dependencies
  - Limitations: stalls at branches, no precise exceptions
- **Solved by speculative Tomasulo (see 3.4.4)**
  - Branch prediction to speculate past branches
  - Re-Order Buffer (ROB) to separate speculative state from non-speculative state
    - enables exception management
  - All modern microprocessors feature some variation of HW-supported speculation

SPECULATIVE TOMASULO



NEW STRUCTURES:

- REORDER BUFFER (ROB)
- BRANCH PREDICTION BUFFER (BPB)
- BRANCH TARGET BUFFER (BTB)

ROB:

- KEEPS TRACK OF PROCESS ORDER (FIFO)
- HOLDS SPECULATIVE RESULTS
- NO MORE SNOOPING BY REGISTERS

REGISTER VALUES

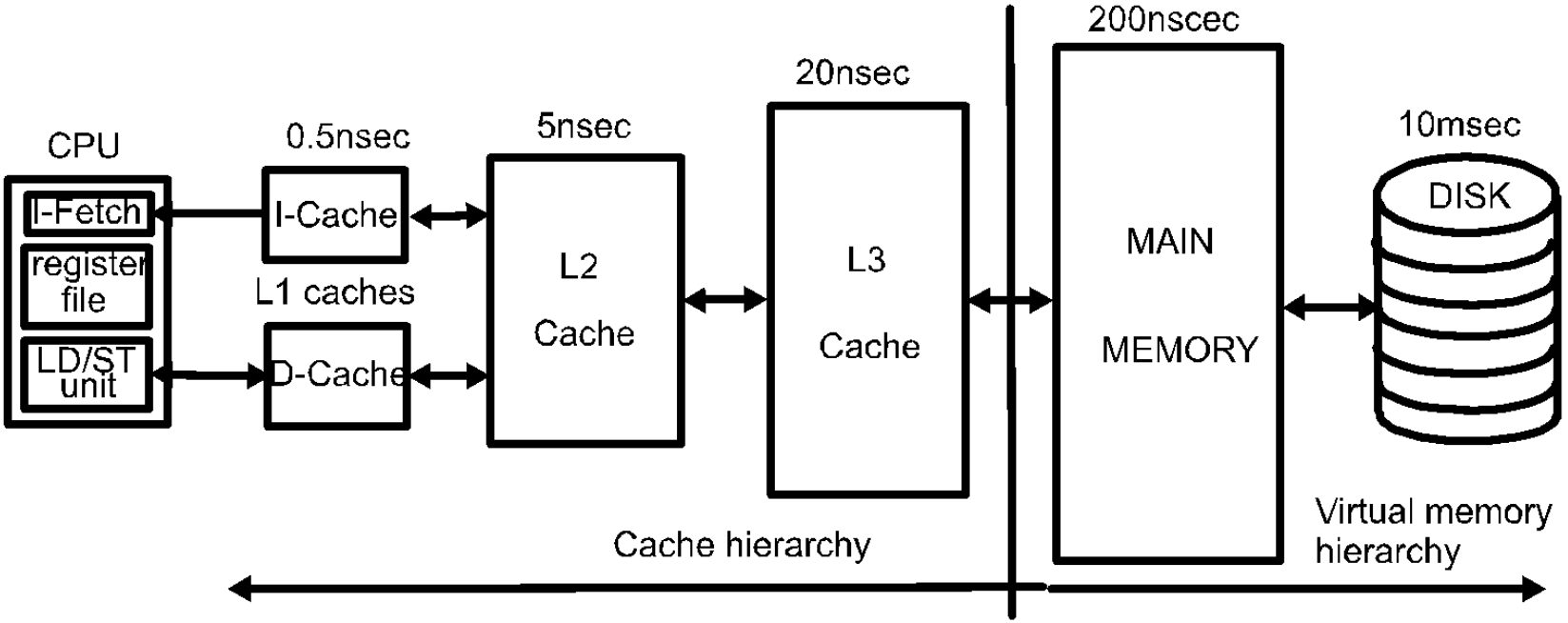
- PENDING IN BACK-END
- SPECULATIVE IN ROB
- COMMITTED IN THE REGISTER FILE

USE ROB ENTRY # AS TAG TO RENAME REGISTER

How much of this hardware actually contributes to execution?

MULTI-LEVEL CACHE HIERARCHY

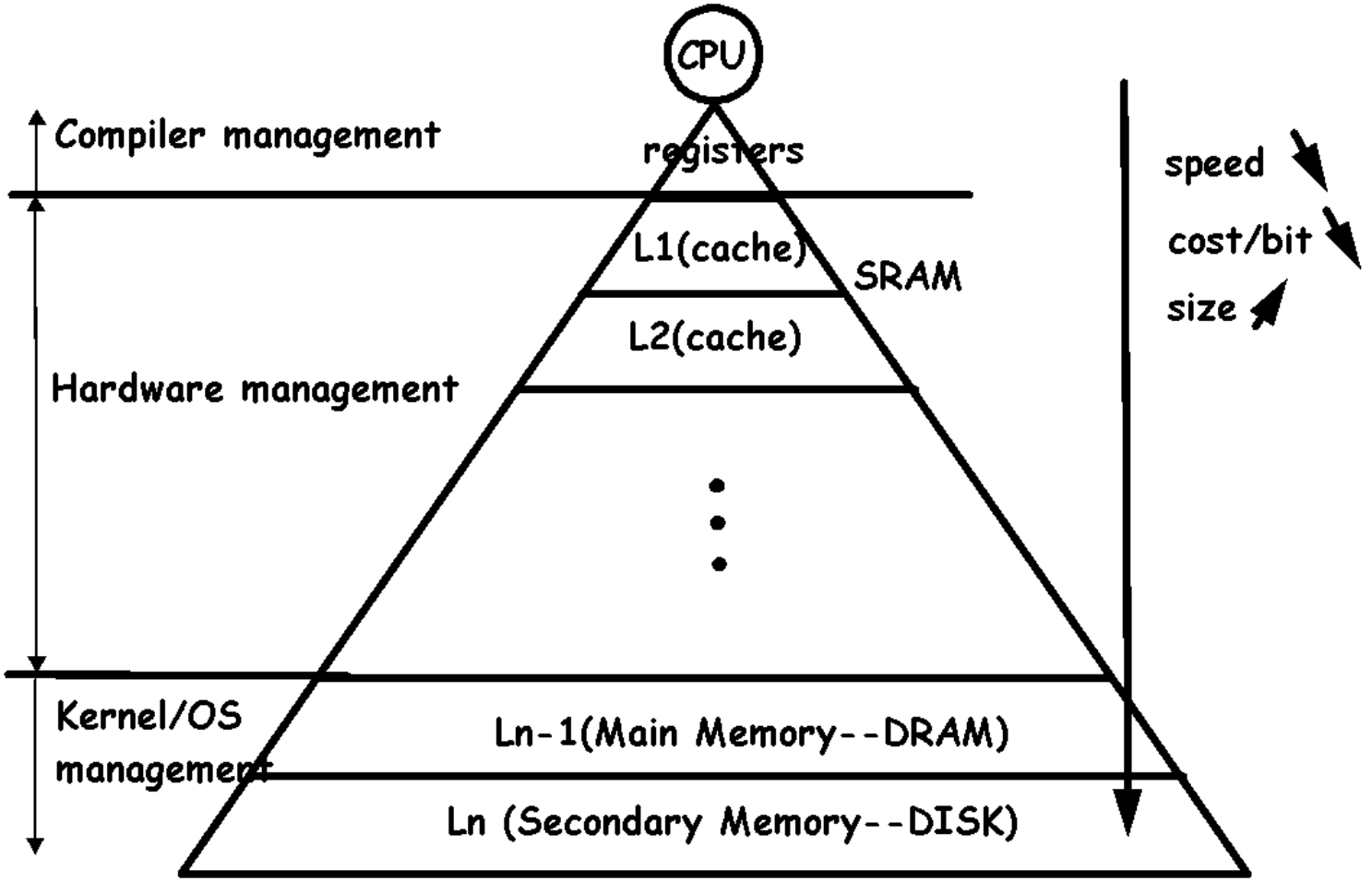
TYPICAL MEMORY HIERARCHY



- **Principle of locality:**

  a program accesses a relatively small portion of the address space at a time
- **two different types of locality:**
  - temporal locality: if an item is referenced, it will tend to be referenced again soon
  - spatial locality: if an item is referenced, items whose addresses are close tend to be referenced soon
  - spatial locality turns into temporal locality in blocks (cache) / pages (disk)

TYPICAL MEMORY HIERARCHY: THE PYRAMID



GOALS: HIGH SPEED, LOW COST, HIGH CAPACITY

COHERENCE := illusion that there is only one copy of each data element

CACHE PERFORMANCE METRICS (4.3.4)

- **AVERAGE MEMORY ACCESS TIME (AMAT)**

  AMAT= hit time + miss rate x miss penalty

- **MISS RATE: FRACTION OF ACCESSES NOT SATISFIED AT THE HIGHEST LEVEL**
  - number of misses in L1 divided by the number of processor references
  - *also* Hit rate = 1 - Miss rate

- **MISSES PER INSTRUCTIONS (MPI)**
  - Number of misses in L1 divided by number of instructions
  - Easier to use than miss rate: $CPI = CPI_0 + MPI*\text{miss penalty}$

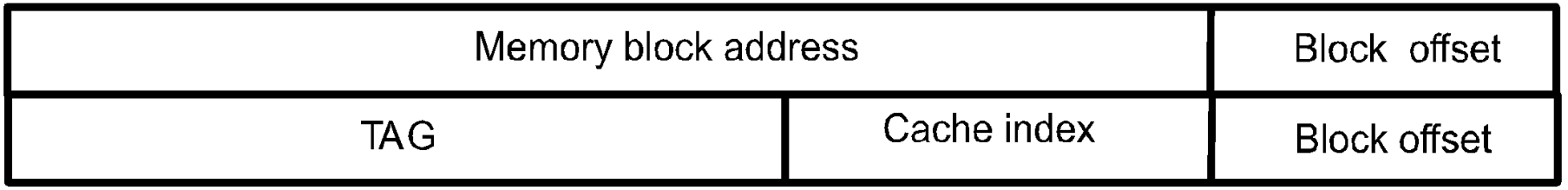
- **MISS PENALTY: AVERAGE DELAY PER MISS CAUSED IN THE PROCESSOR**
  - if processor blocks on misses, then this is simply the number of clock cycles to bring a block from memory or miss latency
  - in a OoO processor, the penalty of a miss cannot be measured directly
    - different from miss latency

- **MISS RATE AND PENALTY CAN BE DEFINED AT EVERY CACHE LEVELS**
  - usually normalized to the number of processor references
  - or to the number of accesses from the upper level

CACHE MAPPING (4.3.1)

- **Memory blocks are mapped to cache lines**
- **Mapping can be direct, set-associative or fully associative**

Physical Address



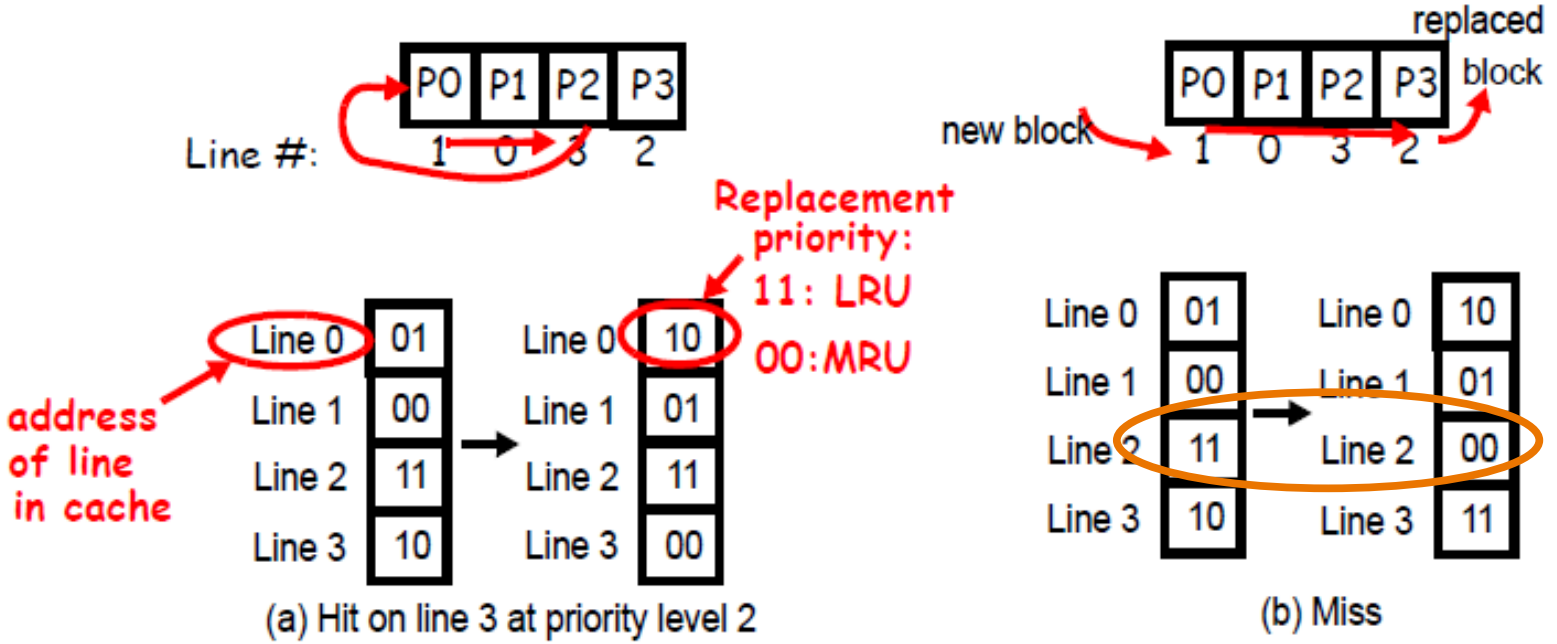
- **DIRECT-MAPPED:** each memory block can be mapped to only one cache line: block address modulo the number of lines in cache
- **SET-ASSOCIATIVE:** each memory block can be mapped to a set of lines in cache; set number is block address modulo the number of cache sets
- **FULLY ASSOCIATIVE:** each memory block can be in any cache line

- **Cache is made of directory+ data memory, one entry per cache line**
  - DIRECTORY: tag + status (state) bits: valid, dirty, reference, cache coherence

- **Cache access has two phases**
  1) **Use index bits to fetch the tags and data from the set (cache index)**
  2) **Check tags to detect hit/miss**

REPLACEMENT POLICIES (4.3.2)

- Which block to evict when a new cache line is filled?
- random, LRU, fifo, Pseudo-LRU
  - maintains replacement bits
- **EXAMPLE:** least-recently used (LRU)



Update of history bits in LRU

DIRECT-MAPPED: no need

SET-ASSOCIATIVE: per-set replacement

FULLY ASSOCIATIVE: cache-level replacement

CLASSIFICATION OF CACHE MISSES (4.3.5)

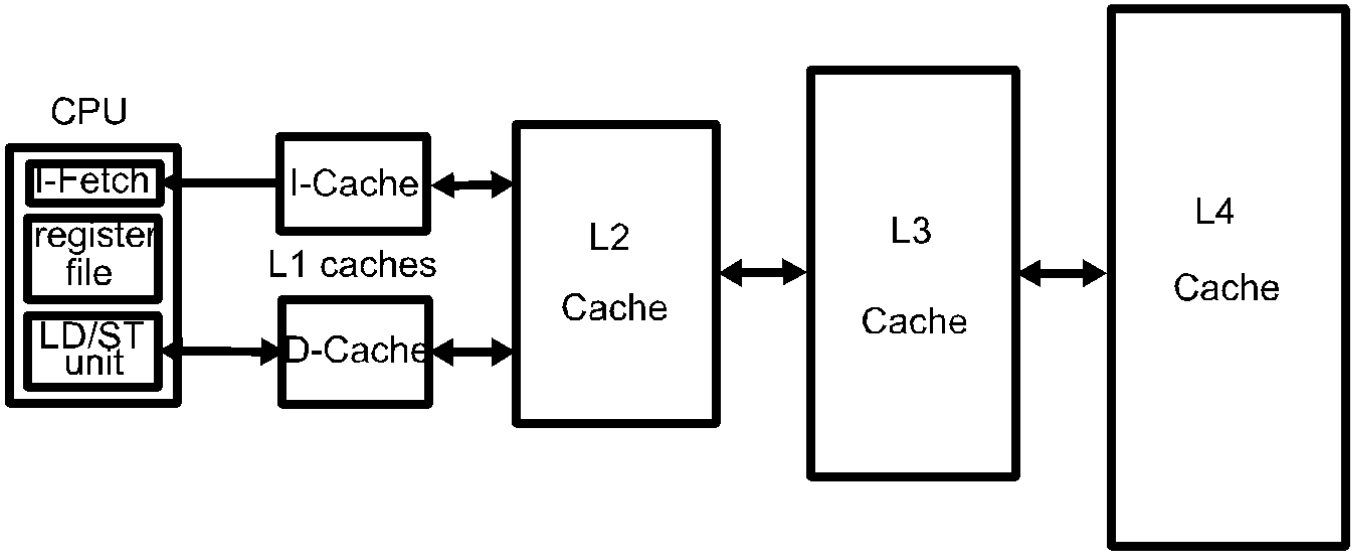
- **THE 3 C's**
  - **compulsory** (cold) misses: on the 1st reference to a block
  - **capacity** misses: space is not sufficient to host data or code
  - **conflict** misses: happen when two memory blocks map on the same cache block in direct-mapped or set-associative caches

    - LATER ON: COHERENCE MISSES → 4C's CLASSIFICATION

- **HOW TO FIND OUT?**
  - cold misses: simulate infinite cache size
  - capacity misses: simulate fully associative cache then deduct cold misses
  - conflict misses: simulate cache then deduct cold and capacity misses

CLASSIFICATION IS USEFUL TO UNDERSTAND HOW TO ELIMINATE MISSES

MULTI-LEVEL CACHE HIERARCHIES



- **1st-level, 2nd and 3rd level are usually on-chip; 4th level mostly off-chip**
- **usually, cache inclusion is maintained** ("inclusive cache hierarchy")
  - Lower cache levels "include" (contain) the data blocks in higher cache levels
  - when a block misses in L1 then it must be brought into all Li.
  - when a block is replaced in Li, then it must be removed from all Lj, j<i
- **also: exclusive cache** ("exclusive cache hierarchy")
  - if a block is in Li then it is not in any other cache level
  - if a block misses in L1 then all copies are removed from all Li's, i>1
  - if a block is replaced in Li then it is allocated in Li+1
- **or no policy (**eg., "non-inclusive cache hierarchy"**)**

THINK: what are the advantages / disadvantages of each scheme?

MULTI-LEVEL CACHE HIERARCHIES

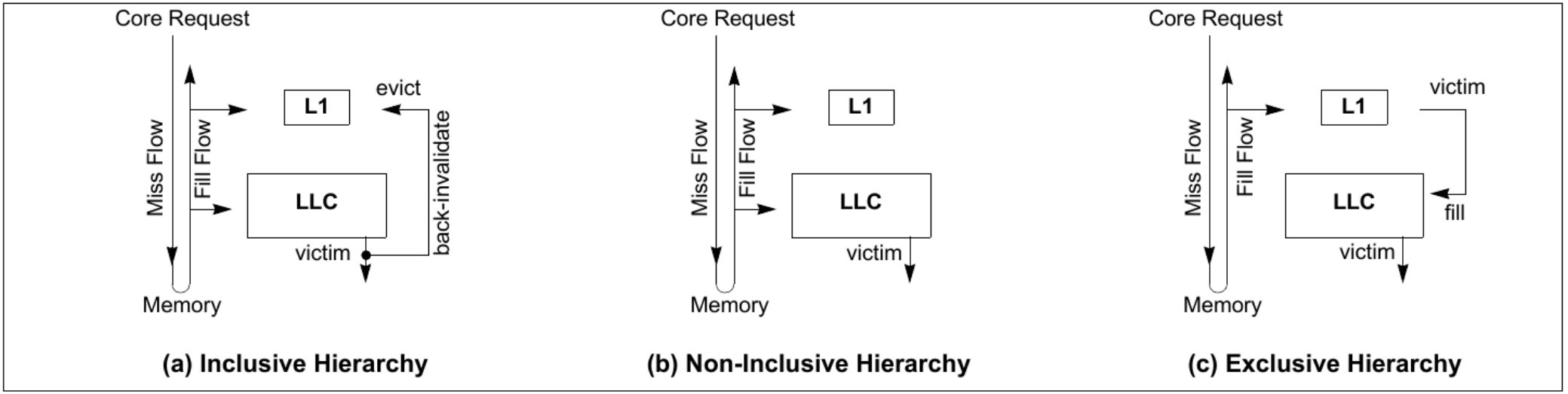
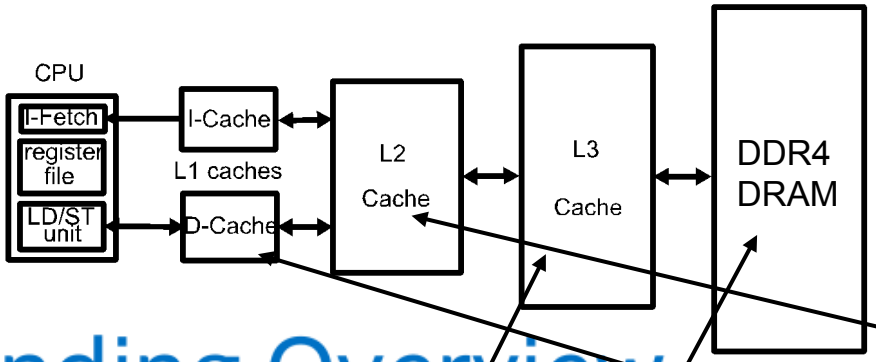


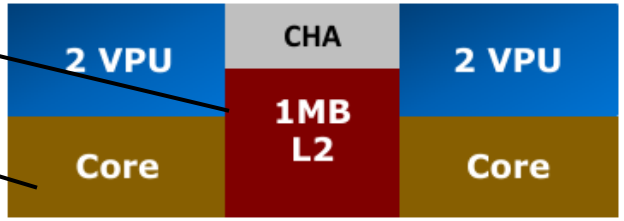
Figure 1: Summary of Cache Hierarchies.

- Miss and Fill flow for different policies
  - Taken from Jaleel et al. "*Achieving Non-Inclusive Cache Performance with Inclusive Caches. Temporal Locality Aware (TLA) Cache Management Policies*" (MICRO 2010)

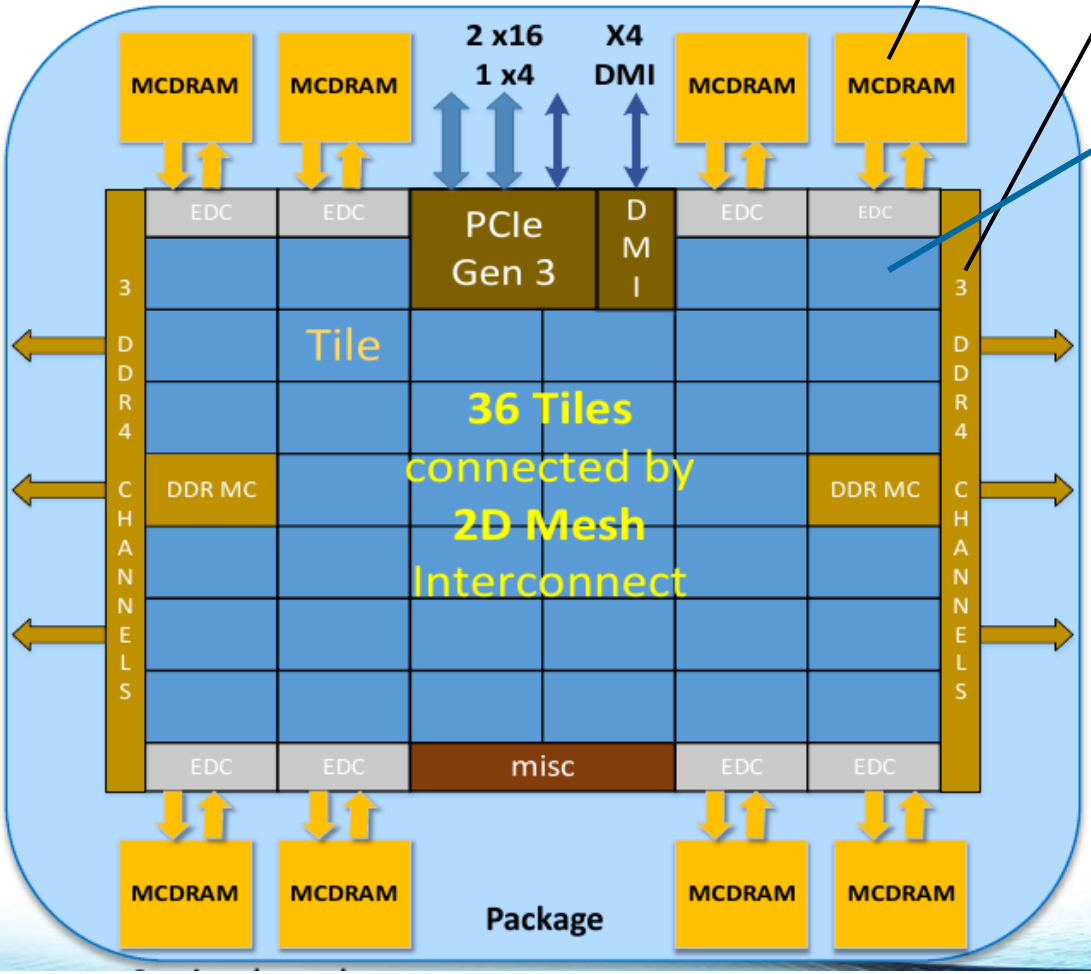
- AMD processors have traditionally used an exclusive hierarchy
- Intel processor traditionally used an inclusive hierarchy. Since Skylake, the hierarchy has been updated to include non-inclusive cache hierarchy

EXAMPLE OF INTEL KNIGHTS LANDING





Knights Landing Overview



Chip: 36 Tiles interconnected by **2D Mesh**

Tile: 2 Cores + 2 VPU/core + 1 MB L2

Memory: MCDRAM: 16 GB on-package; High BW

DDR4: 6 channels @ 2400 up to 384GB

IO: 36 lanes PCIe Gen3. 4 lanes of DMI for chipset

Node: 1-Socket only

Fabric: Omni-Path on-package (not shown)

Vector Peak Perf: 3+TF DP and 6+TF SP Flops

Scalar Perf: ~3x over Knights Corner

Streams Triad (GB/s): MCDRAM : 400+; DDR: 90+

Source Intel: All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice. KNL data are preliminary based on current expectations and are subject to change without notice. 1Binary Compatible with Intel Xeon processors using Haswell Instruction Set (except TSX). 2Bandwidth numbers are based on STREAM-like memory access pattern when MCDRAM used as flat memory. Results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system

EFFECT OF CACHE PARAMETERS

- **LARGER CACHES**
  - slower,
  - more complex,
  - less capacity misses

- **LARGER BLOCK SIZE**
  - exploit spatial locality
  - too big a block increases capacity misses
  - big blocks also increase miss penalty

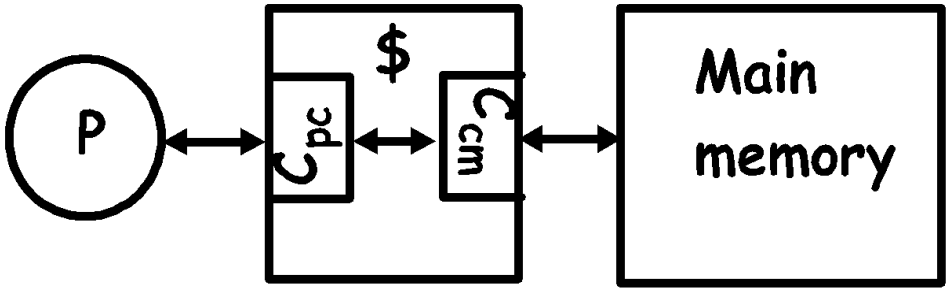
- **HIGHER ASSOCIATIVITY**
  - addresses conflict misses
  - in practice, 8-16 way S.A. is as good as fully associative
  - a 2-way S.A. cache of size N has a similar miss rate as a direct mapped cache of size 2N
  - higher hit time

LOCKUP-FREE (NON-BLOCKING CACHES)

- **CACHE IS A 2-PORTED DEVICE: MEMORY & PROCESSOR**

C_{cm}: cache to memory interface

C_{pc}: processor to cache interface



- if a lockup-free cache misses, it does not block
- rather, it handles the miss and keeps accepting accesses from the processor
  - allows for the concurrent processing of multiple misses and hits

LOCKUP-FREE (NON-BLOCKING CACHES)

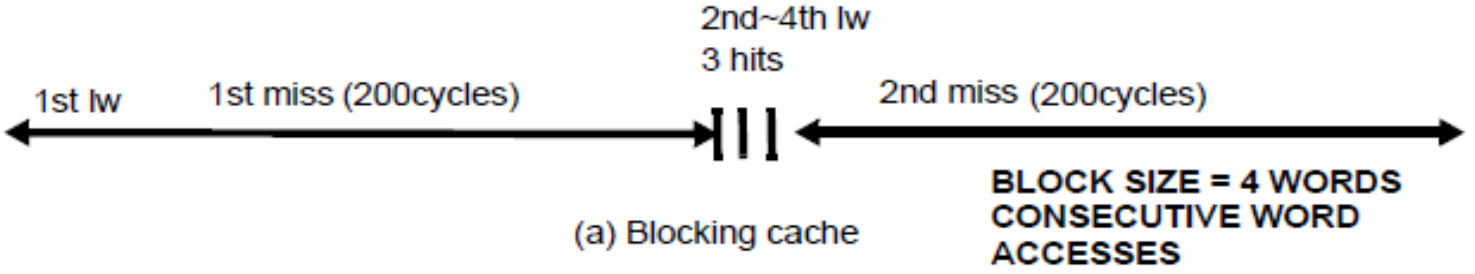
- cache has to bookkeep all pending misses
  - MSHRs (miss status handling registers) contain the address of pending miss, the destination block in cache, and the destination register
  - number of MSHRs limits the number of pending misses
- data dependencies eventually block the processor
- non-blocking caches are required:
  1. dynamically scheduled processors
  2. to support prefetching
  3. for multithreading (L1)
  4. shared L2 caches in multicore

LOCKUP-FREE CACHES: PRIMARY/SECONDARY MISSES

- **PRIMARY: the first miss to a block**
- **SECONDARY: following accesses to blocks pending due to primary miss**
  - Many more misses (blocking cache only has primary misses)
  - Needs MSHRs for both primary and secondary misses
  - Misses are overlapped with computation and other misses

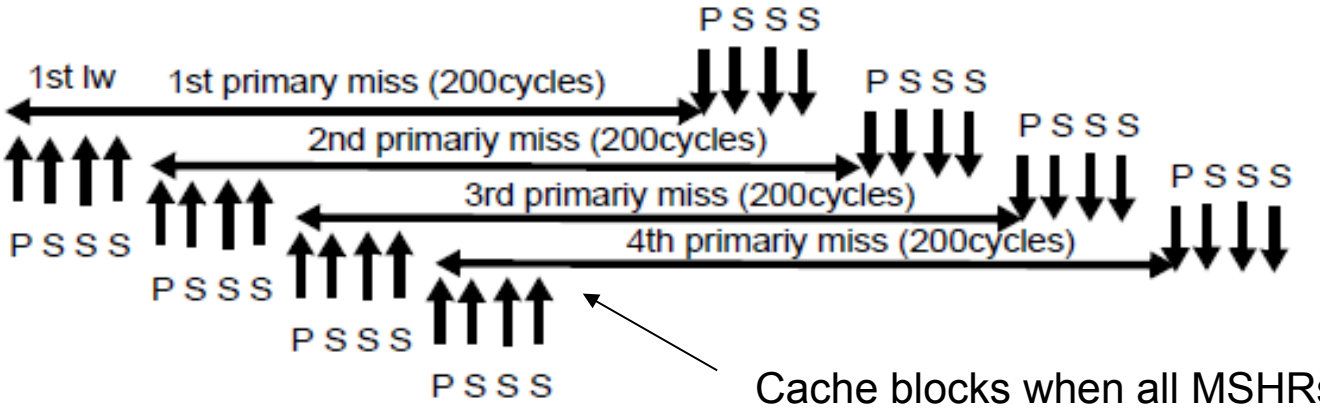
```
TOY: LW R1,0(R2)          Miss latency = 200 cycles
     ADDI R2, R2, #4      #MSHRs = 16
     BNE R2, R4, TOY      Block Size = 4 words
```

BLOCKING CACHE



(a) Blocking cache

NON-BLOCKING CACHE



(b) Non-blocking cache with 16 MSHRs

SUMMARY

- **Out-of-order (Tomasulo) overcomes limits of static scheduling**
  - But introduces considerable control overhead!
- **Multi-level caches**
  - Inclusive vs exclusive, coherence across levels
- **Non-blocking caches**
  - Necessary for chip thread level parallelism

- **Next lecture (Feb 4th): Multiprocessor hardware (i)**
  - Main topic: coherence across MP caches