CS211
Advanced Computer Architecture

L20 Summary

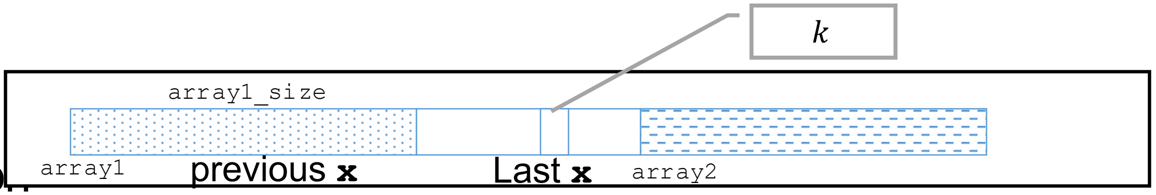
Chundong Wang

December 6th, 2024

上海科技大学 ShanghaiTech University

Spectre: Speculation

- Speculative executio
  - Example: branch prediction



Prerequisites:
i. array1[**x**], with an out-of-bound **x** larger than array1_size, resolves to a secret byte k that is cached;
ii. array1_size and array2 uncached.
iii. Previous **x** values have been valid.

```
// x is controlled by attacker.
1. if (x < array1_size)
2.    y = array2[array1[x] * 4096]
```

← cache miss, so run next line due to prediction history

← array1[**x**] cache hit, as k is cached, so load array2[k * 4096]

Regarding a misprediction with an illegal **x**, array2[k * 4096] will not be used, but has been loaded into CPU cache.
We can use Flush+Reload to guess k with array2.

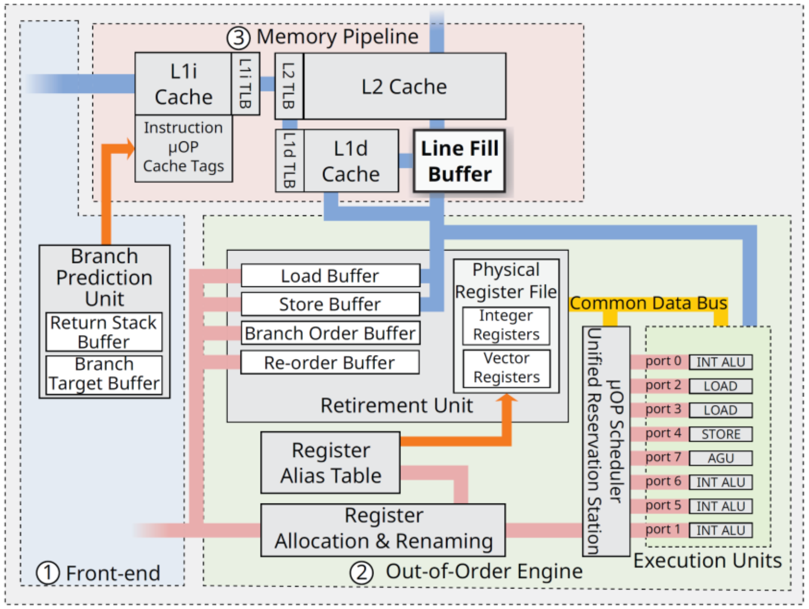
The aim of Spectre:
to read out a victim's sensitive information

Source: https://spectreattack.com/spectre.pdf

RIDL: Rogue In-Flight Data Load

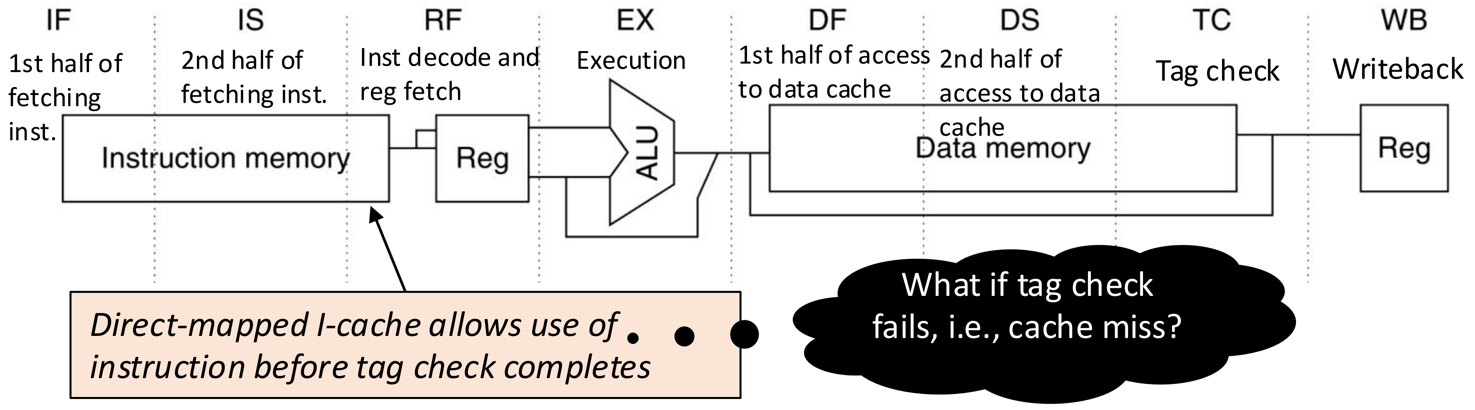
- Not data in cache, but in line fill buffer (LFB)

Line Fill Buffers (LFBs) are internal buffers that the CPU uses to keep track of outstanding memory requests and perform a number of optimizations such as merging multiple in-flight stores.



An overview of the Intel Skylake microarchitecture from *RIDL: Rogue In-Flight Data Load,* S&P '19

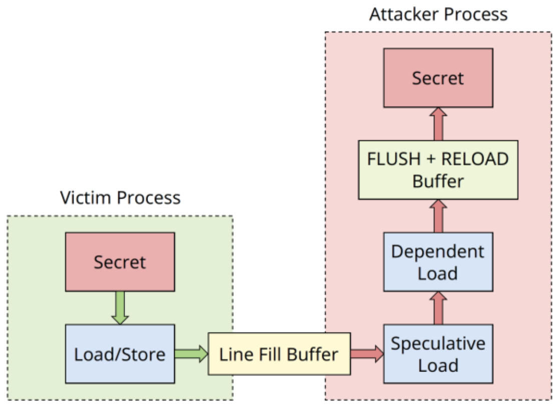
Speculative load (L05)


IF
1st half of fetching inst.
IS
2nd half of fetching inst.
RF
Inst decode and reg fetch
EX
Execution
DF
1st half of access to data cache
DS
2nd half of access to data cache
TC
Tag check
WB
Writeback
Instruction memory
Reg
ALU
Data memory
Reg
Direct-mapped I-cache allows use of instruction before tag check completes
What if tag check fails, i.e., cache miss?

The eight-stage pipeline structure of the MIPS R4000 (1991) uses pipelined instruction and data caches.

RIDL: an example

Any problems with RIDL?



An overview of the RIDL attack from *RIDL: Rogue In-Flight Data Load,* S&P '19

```
 1  /* Flush flush & reload buffer entries. */
 2  for (k = 0; k < 256; ++k)
 3    flush(buffer + k * 1024);
 4
 5  /* Speculatively load the secret. */
 6  char value = *(new_page);
 7  /* Calculate the corresponding entry. */
 8  char *entry_ptr = buffer + (1024 * value);
 9  /* Load that entry into the cache. */
10  *(entry_ptr);
11
12  /* Time the reload of each buffer entry to
13     see which entry is now cached. */
14  for (k = 0; k < 256; ++k) {
15    t0 = cycles();
16    *(buffer + 1024 * k);
17    dt = cycles() - t0;
18
19    if (dt < 100)
20      ++results[k];
21  }
```

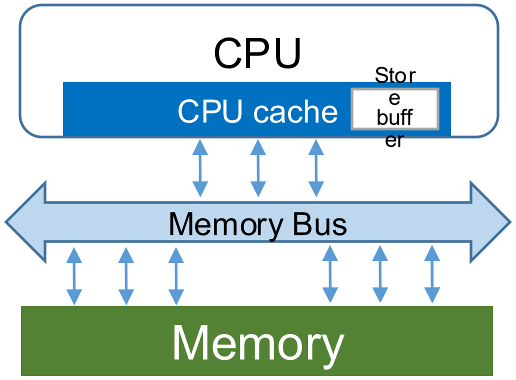
when executing Line 6, the CPU speculatively loads a value from memory in the hope it is from our newly allocated page, while really it is in-flight data from the LFBs belonging to an arbitrarily different security domain

More MDS attacks

- Microarchitectural data sampling (MDS)
- Fallout
    - *Fallout: Leaking Data on Meltdown-resistant CPUs*, CCS ’19
- ZombieLoad
    - *ZombieLoad: Cross-Privilege-Boundary Data Sampling,* CCS ’19
- Medusa
    - *Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis,* USENIX Security ’20
- CacheOut
    - *CacheOut: Leaking Data on Intel CPUs via Cache Evictions,* S&P ’21

LVI: Value Injection

CPU thinks this one in store buffer seems to be the one it needs.

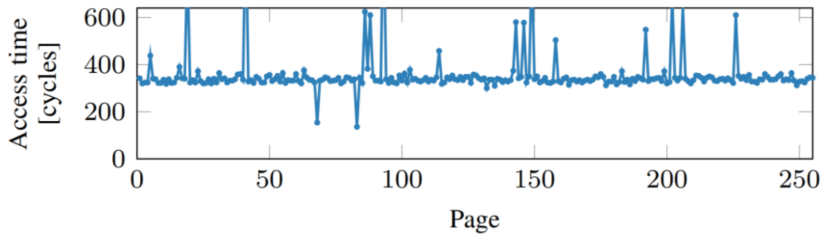


```
void call_victim(size_t untrusted_arg)
{
        *arg_copy = untrusted_arg;
        array[**trusted_ptr * 4096];
}
```

Dereference: but at 1st level, page fault.

An attacker-controlled value is fed, and some micro-architectural state change happens.

CPU may use data in the store buffer upon a cache miss of memory load, although the two are completely unrelated.

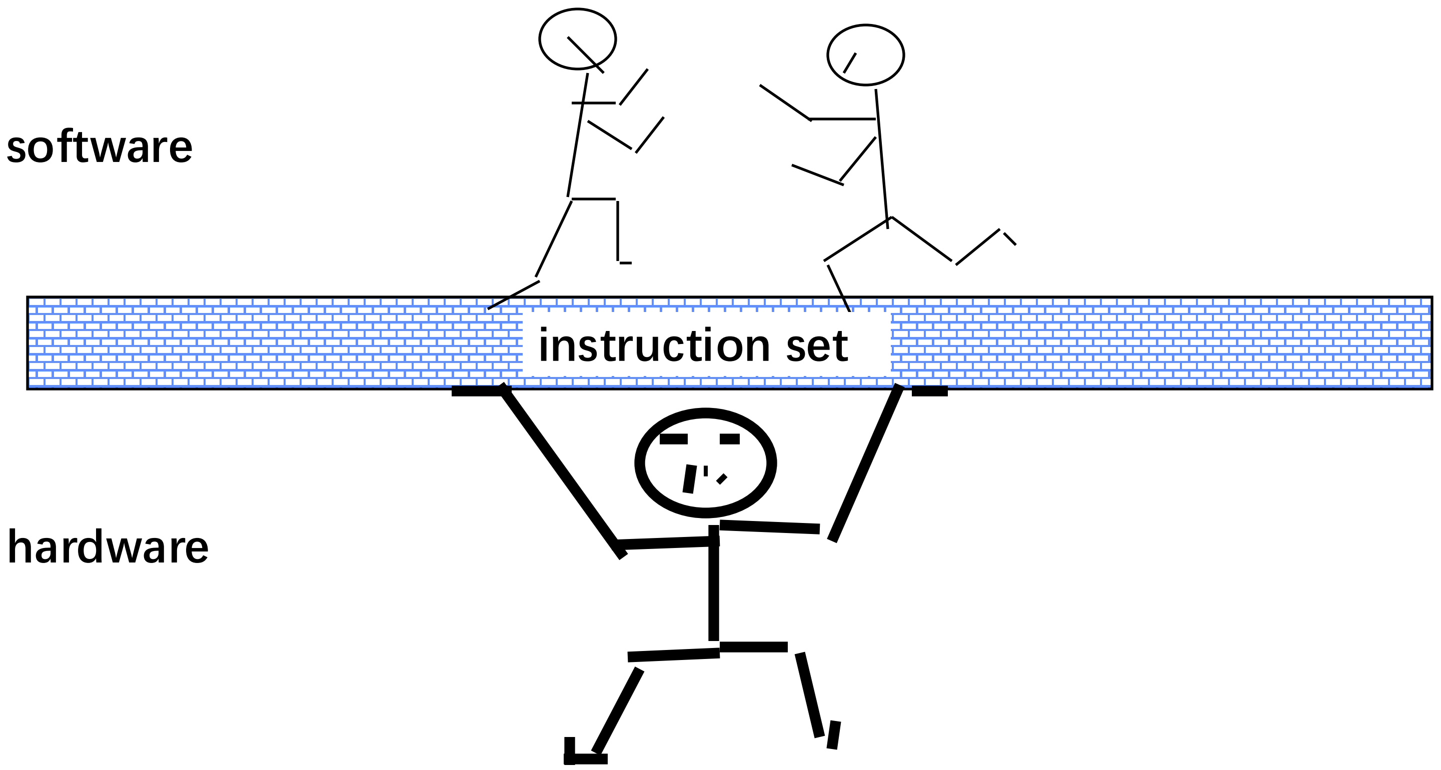


LVI - Hijacking Transient Execution with Load Value Injection, S&P '20

Let’s start refreshing.

ISA: instruction set architecture

An instruction is a single operation, with an opcode and zero or more operands, of a processor, defined by the processor instruction set.



ISA is the actual programmer-visible instruction set, a critical interface/boundary/contract between software and hardware

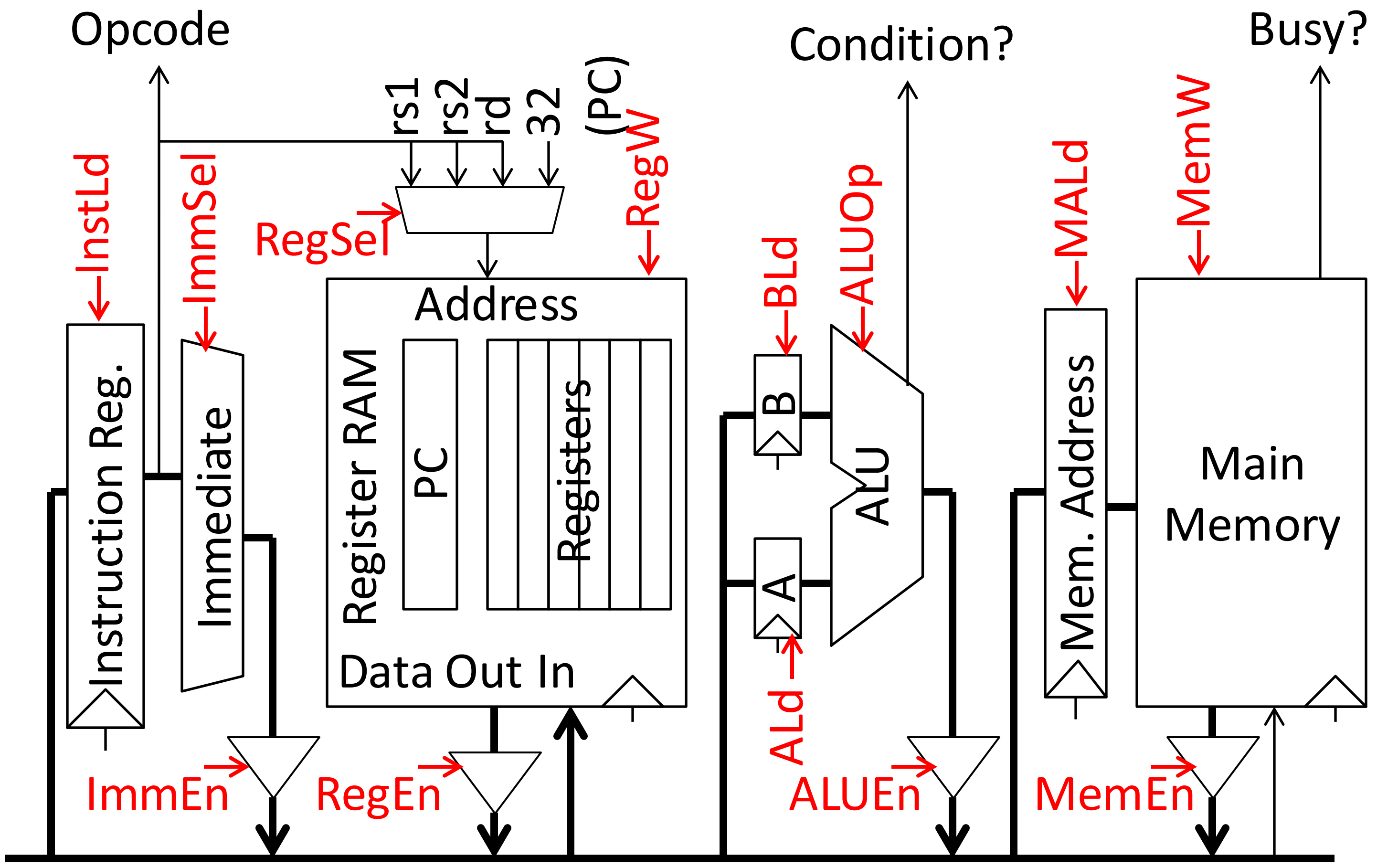
From architecture to microarchitecture

- Instructions are visible to programmers
- Two processors may have the same ISA but different microarchitectures
  - e.g., AMD Opteron and Intel Core i7, with the same 80x86 ISA, have very different pipelines and cache organizations
- Stack and accumulator
- CISC and RISC
- An instruction is partitioned into multiple stages
- Five classic stages of executing an instruction
  - Instruction fetch
  - Instruction decode/register fetch
  - Execute
  - Memory access
  - Write back

Microcode vs. Hardwired

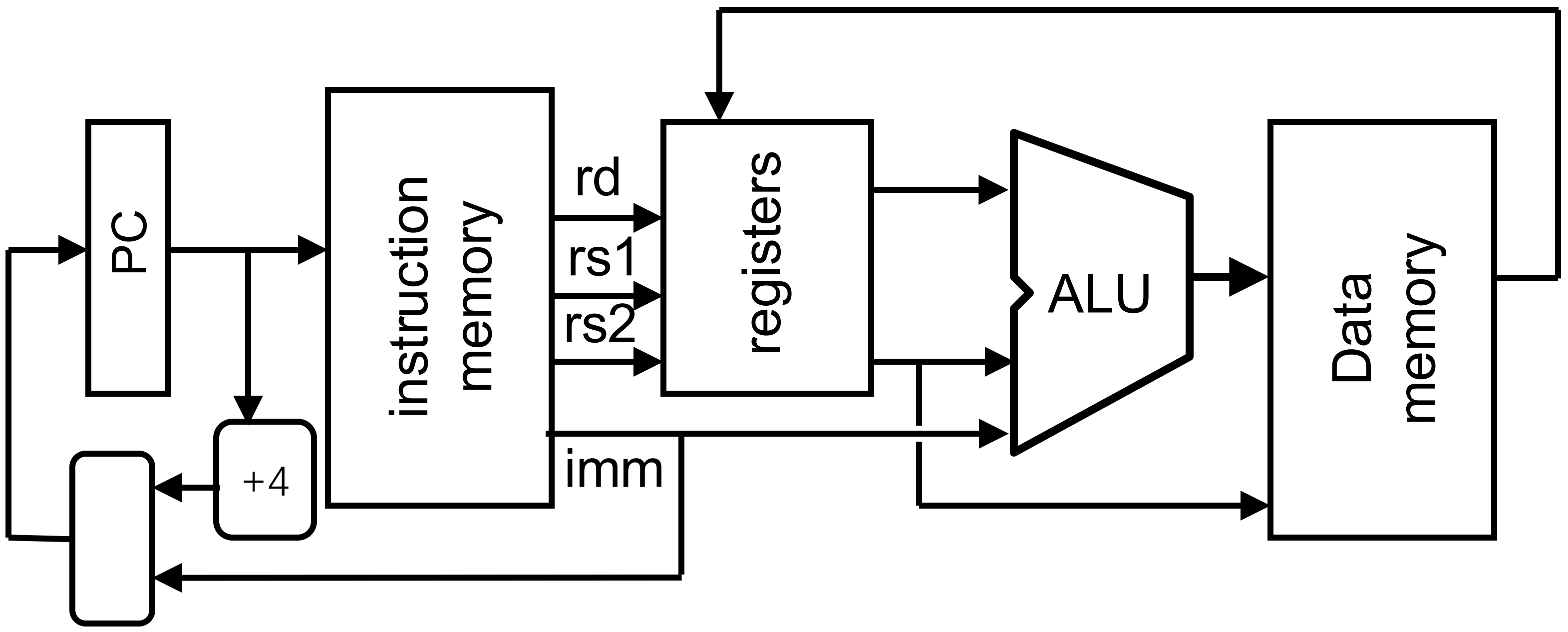
- Data path and control
- Microcoded control
  - Implemented using ROMs/RAMs
  - Indirect next_state function: “here’s how to compute next state”
  - Slower … but can do complex instructions
  - Multi-cycle execution (of control)
- Hardwired control
  - Implemented using logic (“hardwired” can’t re-program)
  - Direct next_state function: “here is the next state”
  - Faster … for simple instructions (speed is function of complexity)
  - Single-cycle execution (of control)

Single-Bus Datapath for Microcoded RISC-V



Datapath unchanged for complex instructions!

Stages of Execution on Datapath

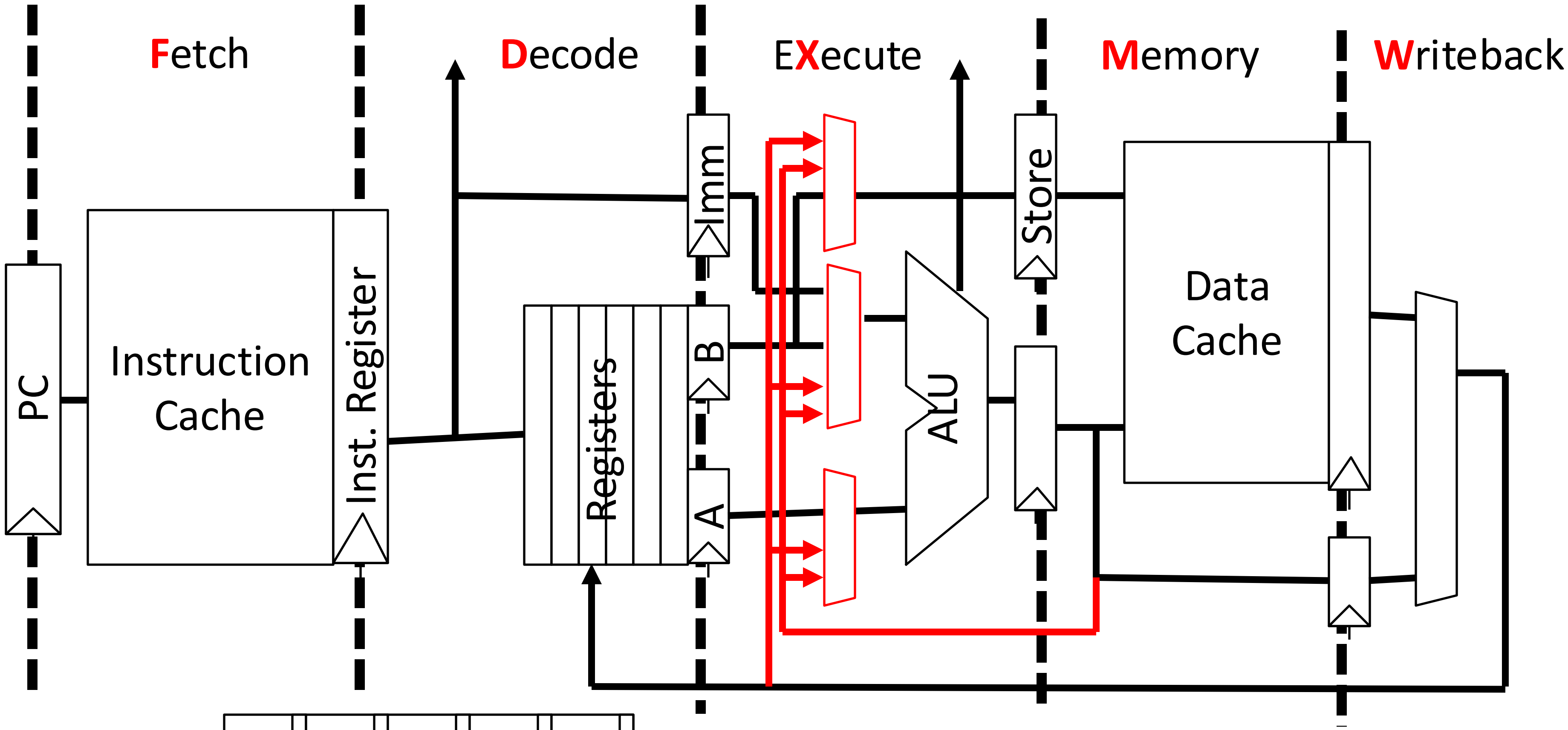


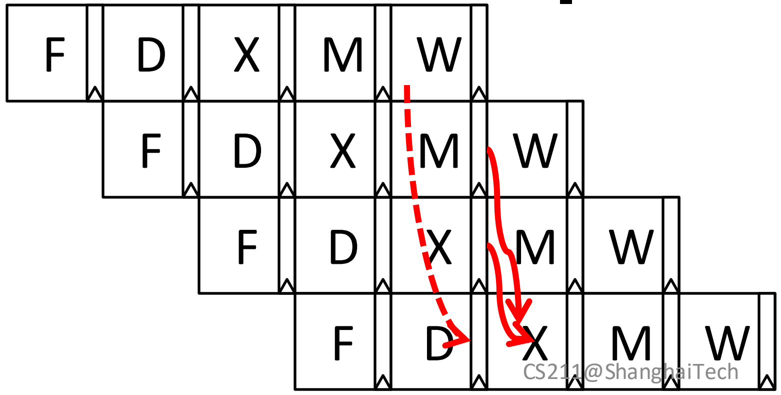
1. Instruction Fetch
2. Decode/ Register Read
3. Execute
4. Memory
5. Write Back

Hazards

- Structural Hazard
  - A required hardware resource is busy
- Data hazard
  - An instruction depends on the result(s) of a previous instruction
- Control Hazard
  - Branches, jumps, etc.
  - Exception
    - Exception handling

Pipeline with Fully Bypassed Data Path





[Assumes data written to registers in a W cycle is readable in parallel D cycle (dotted line). Extra write data register and bypass paths required if this is not possible.]

Types of Data Hazards

Consider executing a sequence of register-register instructions of type:

$$r_k \leftarrow r_i \ \mathrm{op}\ r_j$$

Data-dependence

$$\begin{tikzcd}
r_3 \arrow[drr] & \leftarrow & r_1 & \mathrm{op}\ r_2 \\
r_5 & \leftarrow & r_3 & \mathrm{op}\ r_4
\end{tikzcd}$$

Read-after-Write (RAW) hazard

Anti-dependence

$$\begin{tikzcd}
r_3 & \leftarrow & r_1 \arrow[dll] & \mathrm{op}\ r_2 \\
r_1 & \leftarrow & r_4 & \mathrm{op}\ r_5
\end{tikzcd}$$

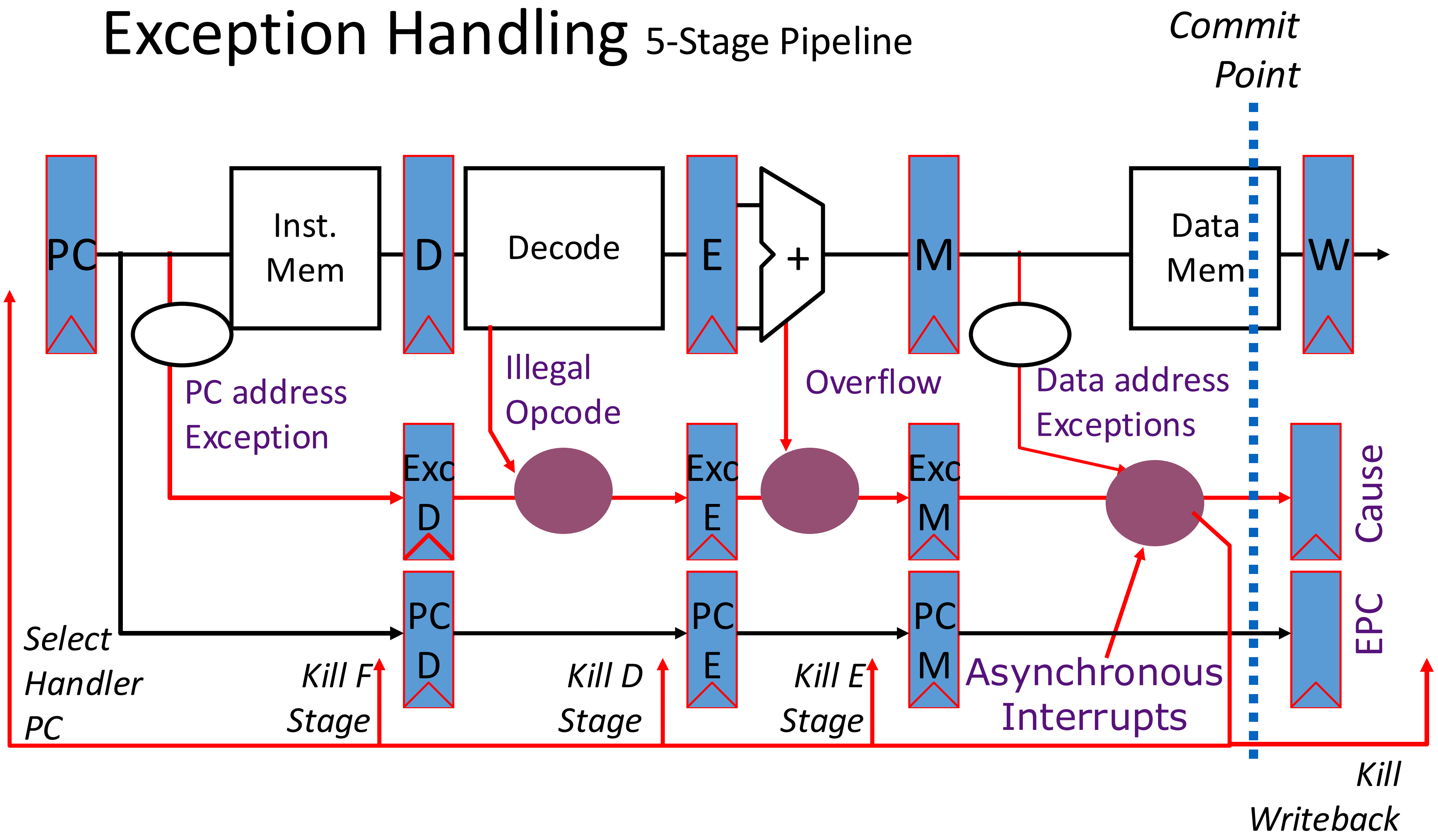
Write-after-Read (WAR) hazard

Output-dependence

$$\begin{tikzcd}
r_3 \arrow[d, bend right] & \leftarrow & r_1 & \mathrm{op}\ r_2 \\
r_3 & \leftarrow & r_6 & \mathrm{op}\ r_7
\end{tikzcd}$$

Write-after-Write (WAW) hazard

Exception Handling 5-Stage Pipeline



Pipeline scheduling

- Loop unrolling
  - Long latency loads and floating-point operations limit parallelism within a single loop iteration
  - Loop unrolled to expose more parallelism
    - Available registers, code generation, etc. to be considered
    - With an unrolling factor
- Decoupling access and execute
  - Separate control and memory access operations from data computations

  + Execute stream can run ahead of the access stream and vice versa

  + Limited out-of-order execution without wakeup/select complexity

  - Compiler support to partition the program and manage queues

  - Branch instructions require synchronization between A and E

More Complex Pipeline: Scoreboard

- When is it safe to issue an instruction?
  - Use a data structure to keep track of all the instructions in all the functional units
- The following checks need to be made before the Issue stage can dispatch an instruction
  - Is the required function unit (FU) available?
  - Is the input data available? (RAW?)
  - Is it safe to write the destination? (WAR? WAW?)
  - Is there a structural conflict at the WB stage?

an instruction is not dispatched by the Issue stage if a RAW hazard exists or the required functional unit (FU) is busy, and that operands are latched by FU on issue

- Scoreboard for In-order Issues
  - Issue checks the instruction (opcode dest src1 src2) against the scoreboard (Busy & WP) to dispatch

FU available?	Busy[FU#]	
RAW?	WP[src1] or WP[src2]	
WAR?	*cannot arise*	*NO: Operands read at issue*
WAW?	WP[dest]	*YES: Out-of-order completion*

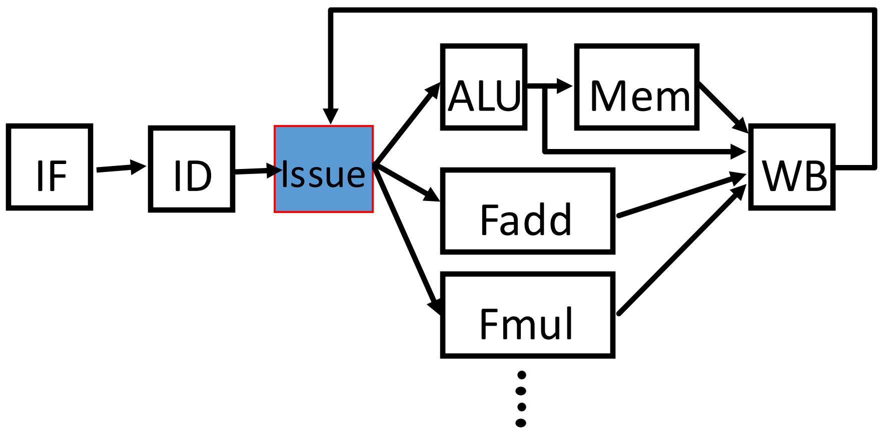
Scoreboard Dynamics

	Functional Unit Status										Registers Reserved for Writes	
	Int(1)	Add(1)	Mult(3)			Div(4)				WB		
t0	I_1					f6					f6	
t1	I_2 f2						f6				f6, f2	
t2								f6		f2	f6, f2	$\underline{I_2}$
t3	I_3		f0						f6		f6, f0	
t4				f0						f6	f6, f0	$\underline{I_1}$
t5	I_4				f0	f8					f0, f8	
t6							f8			f0	f0, f8	$\underline{I_3}$
t7	I_5	f10						f8			f8, f10	
t8									f8	f10	f8, f10	$\underline{I_5}$
t9										f8	f8	$\underline{I_4}$
t10	I_6	f6									f6	
t11										f6	f6	$\underline{I_6}$

I_1	FDIV.D	f6,	f6,	f4	
I_2	FLD	f2,	45(x3)		
I_3	FMULT.D	f0,	f2,	f4	
I_4	FDIV.D	f8,	f6,	f2	
I_5	FSUB.D		f10,	f0,	f6
I_6	FADD.D		f6,	f8,	f2

In-order issue

Register renaming and ROB



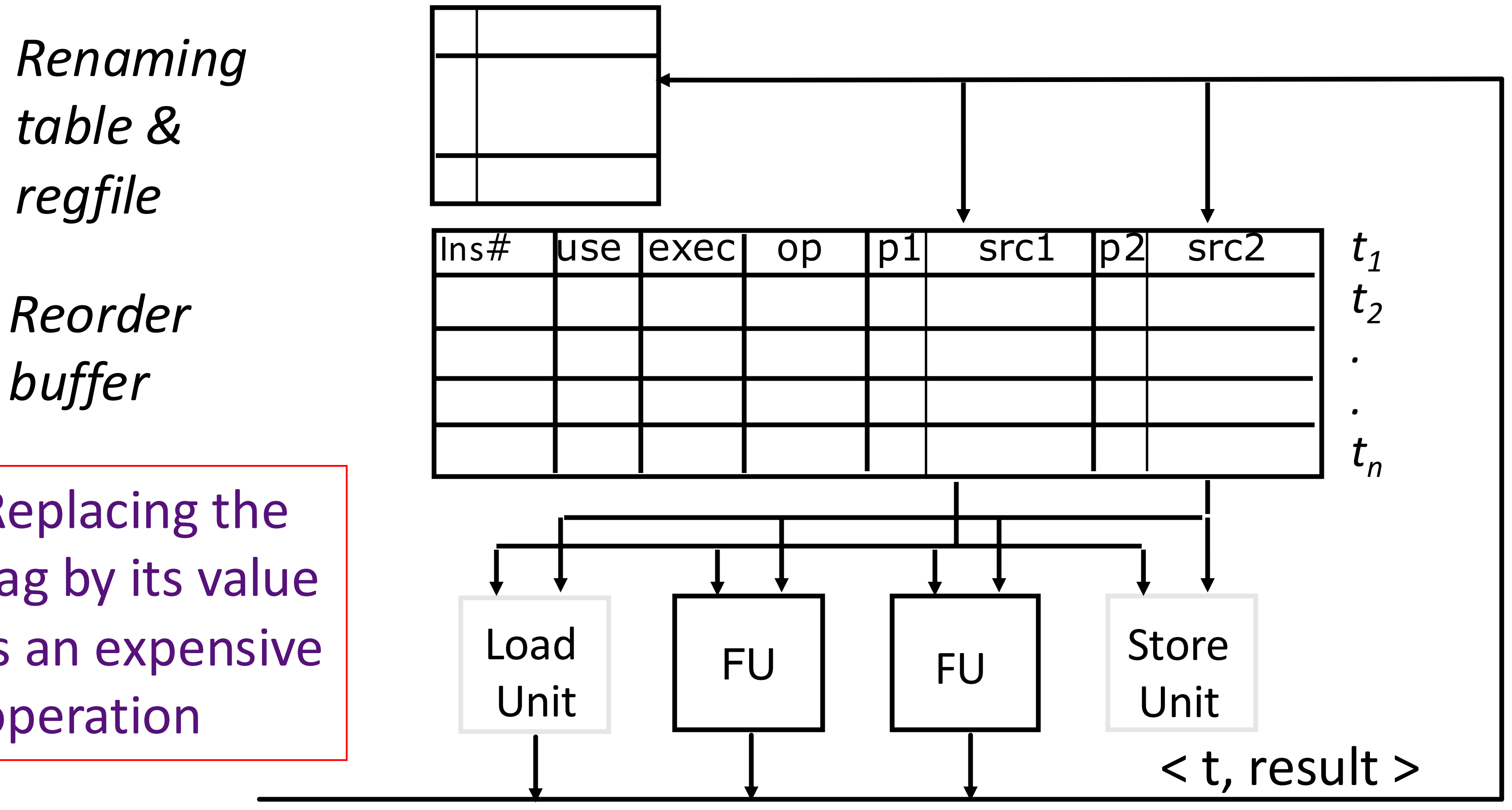
- Decode does register renaming and adds instructions to the issue-stage instruction reorder buffer (ROB)

  ➔ renaming makes WAR or WAW hazards impossible

- Any instruction in ROB whose RAW hazards have been satisfied can be dispatched

  ➔ Out-of-order or dataflow execution

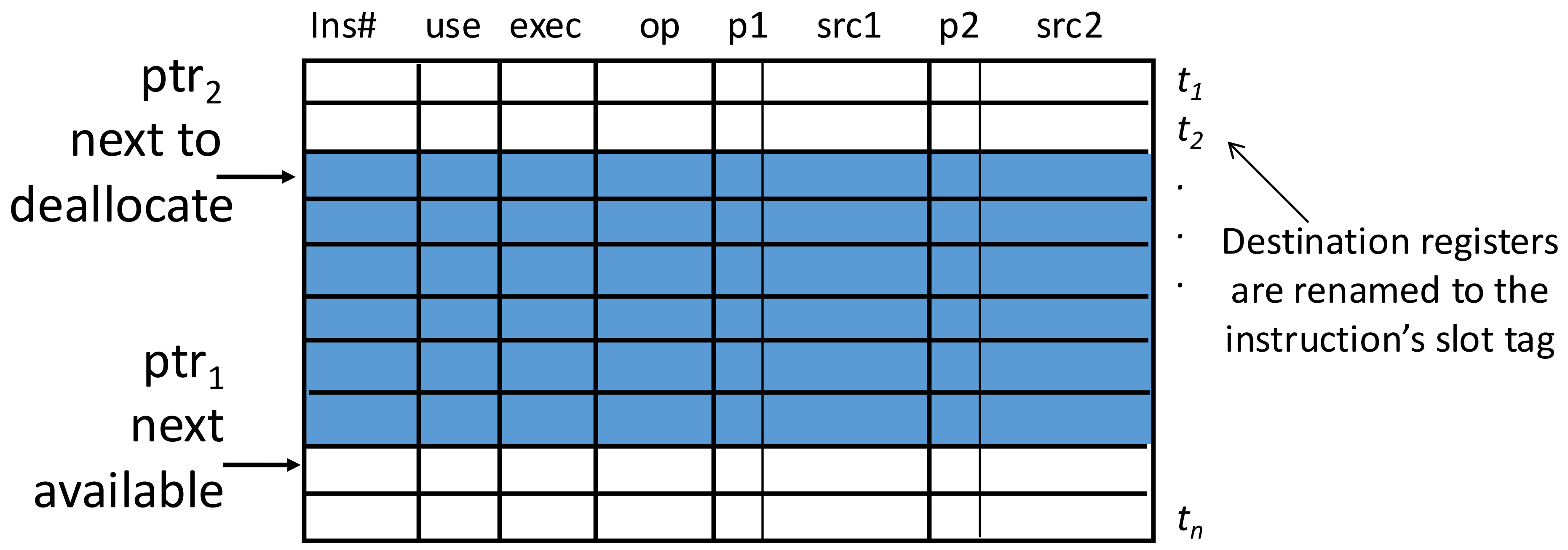
Renaming Structures



Replacing the tag by its value is an expensive operation

- Instruction template (i.e., tag t) is allocated by the Decode stage, which also associates tag with register in regfile
- When an instruction completes, its tag is deallocated

Reorder Buffer Management


Ins#
use
exec
op
p1
src1
p2
src2
ptr$_2$ next to deallocate
ptr$_1$ next available
t_1
t_2
.
.
.
t_n
Destination registers are renamed to the instruction's slot tag

Instruction slot is candidate for execution when:

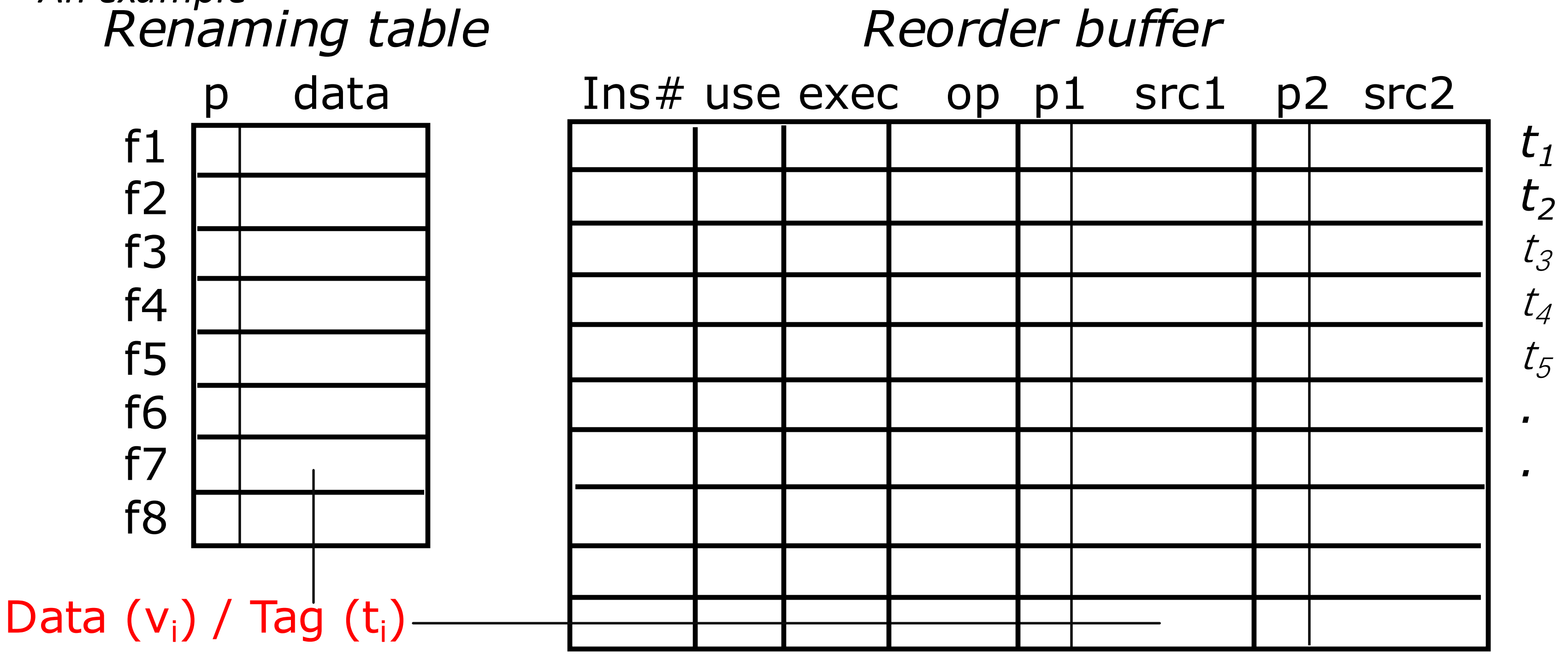
- It holds a valid instruction ("use" bit is set)
- It has not already started execution ("exec" bit is clear)
- Both operands are available (p1 and p2 are set)

Is it obvious where an architectural register value is?

No

Renaming & Out-of-order Issue

An example



1 FLD	f2,	34(x2)	
2 FLD	f4,	45(x3)	
3 FMULT.D	f6,	f4,	f2
4 FSUB.D	f8,	f2,	f2
5 FDIV.D	f4,	f2,	f8
6 FADD.D	f10,	f6,	f4

- *When are tags in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

Renaming & Out-of-order Issue

An example

Renaming table

	p	data
f1		
f2		t1 v1
f3		
f4		t2 t5
f5		
f6		t3
f7		
f8		t4 v4

Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2	
1	0 1	0 1	LD					t_1
2	1 0	0 1	LD					t_2
3	1	0	MUL	0 1	t2 v2	1	v1	t_3
4	1 0	0 1	SUB	1	v1	1	v1	t_4
5	1	0	DIV	1	v1	0 1	t4 v4	t_5
								.
								.

Data (v_i) / Tag (t_i)

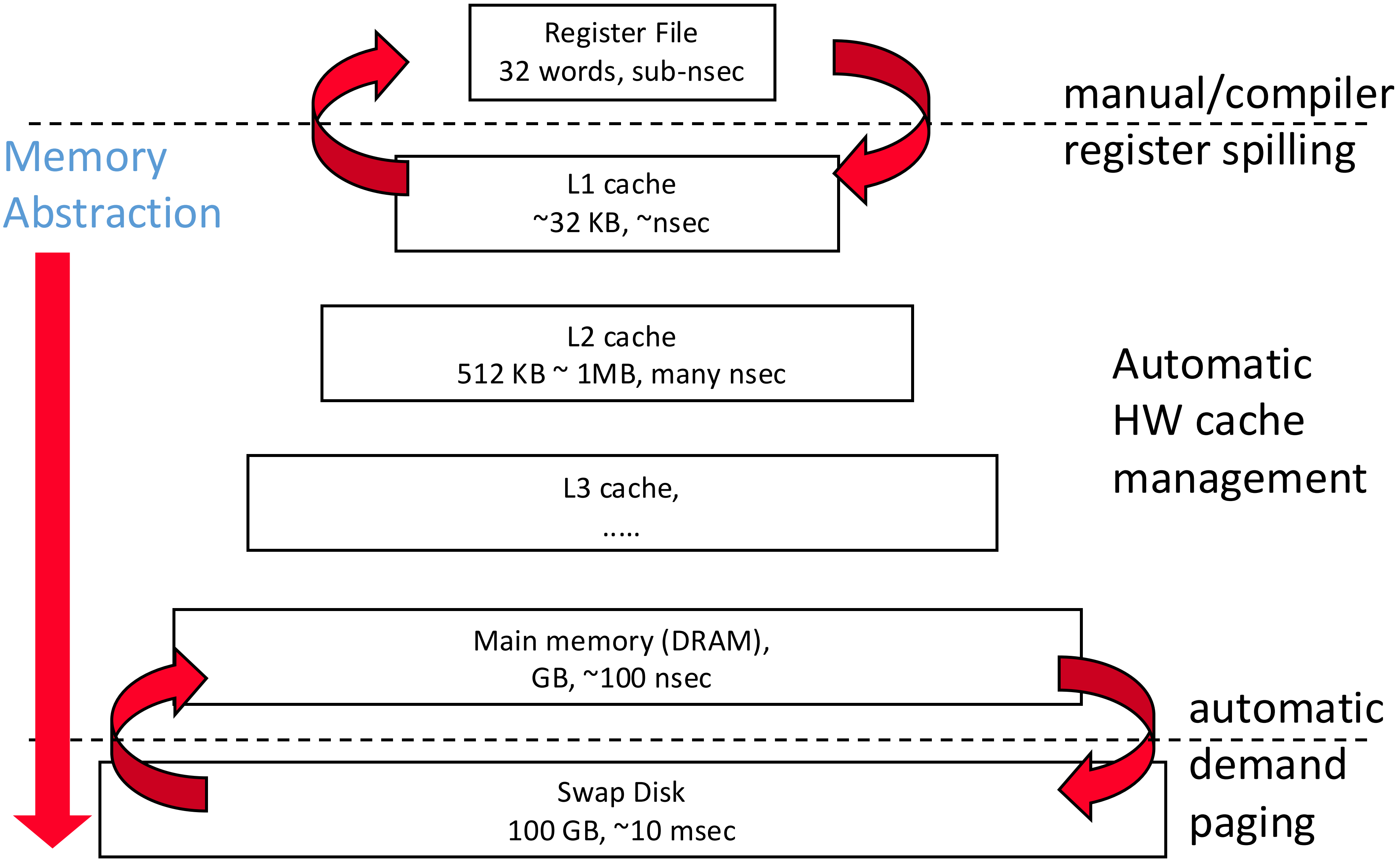
1 FLD	f2,	34(x2)	
2 FLD	f4,	45(x3)	
3 FMULT.D	f6,	f4,	f2
4 FSUB.D	f8,	f2,	f2
5 FDIV.D	f4,	f2,	f8
6 FADD.D	f10,	f6,	f4

- Insert instruction in ROB
- Issue instruction from ROB
- Complete instruction
- Empty ROB entry

In-order vs. Out-of-order

- In-order issue vs. out-of-order issue
  - Dependence, available registers, available FU, etc.
- In-order completion vs. out-of-order completion
  - All but the simplest machines have out-of-order completion
- In-order commit vs. out-of-order commit
  - In-order commit supports precise traps

A Modern Memory Hierarchy



CPU cache

- Store recently accessed data in automatically managed (hardware controlled) fast memory
  - Temporal locality
    - Recently accessed data will be again accessed in the near future
  - Spatial locality
    - A program tends to reference a cluster of memory locations at a time
    - Cache block/line
- Direct-mapped, set-associative, fully-associative
- Hierarchical multi-level cache
  - Cache hit/miss rate, and how to calculate latency accordingly
  - Inclusive, exclusive, or non-inclusive

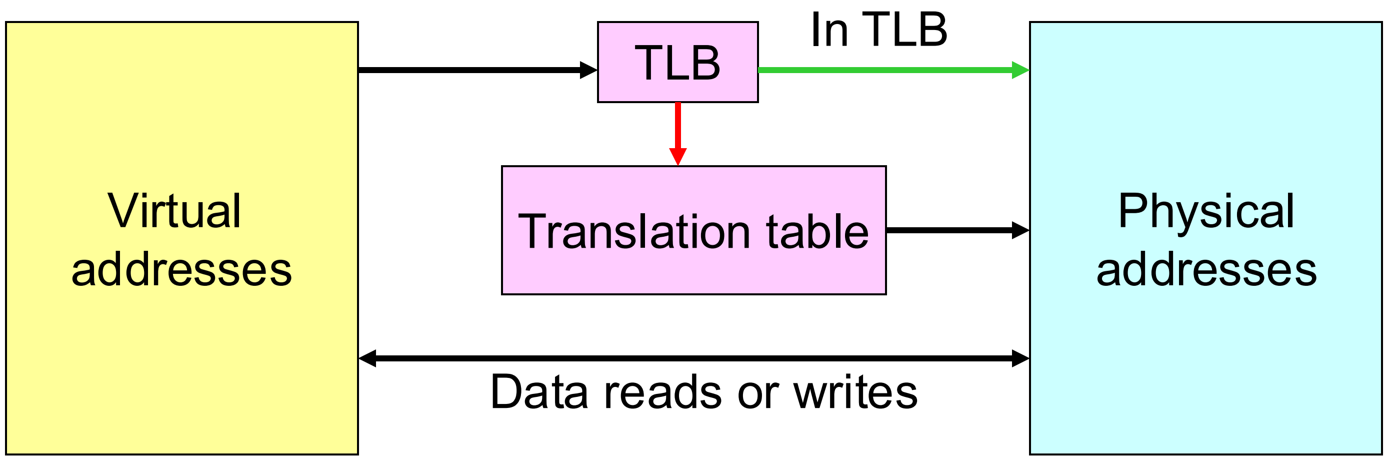
CPU Cache

- Write-back/write-through
- Write-allocate or not
- Unified cache or separate instruction/data caches
- LRU/NRU/NMRU/… replacement policy
- Address used for caching
  - Virtually-Indexed Physically-Tagged (VIPT)
  - PIPT and VIVT

Cache Coherence

- Coherence: What values can a read return?
  - Concerns reads/writes to a single memory location
  - Write propagation: Writes eventually become visible to all processors
  - Write serialization: Writes to the same location are serialized (all processors see them in the same order)
- Snoopy Cache-Coherence Protocols
  - MSI, MESI, MOESI, etc.
  - False sharing
    - Coherence miss
- Buses don’t scale, directory cache protocol
  - Write miss vs. read miss

Virtual memory and TLB, NVM



- Virtual to physical page-level mapping
- Translation lookaside table (TLB)
    - A cache of frequently used page table entries
- DRAM’s problems
- Non-volatile memory
    - Flash memory
    - Byte-addressable NVM

Memory Consistency

- *Consistency* describes properties across *all* memory addresses
  - When writes to *X* propagate to other processors, relative to reads and writes to other addresses
  - A memory consistency model is a contract between the hardware and software
- Sequential Consistency (SC)
  - Arbitrary *order-preserving interleaving* of memory references of sequential programs over a single shared memory, in some sequential order
- Relaxed consistency
  - TSO: total store ordering
  - Fences (memory barrier), with overheads
- Multi-Copy Atomic, and Non-Multi-Copy Atomic

Branch Prediction

- Usefulness
  - Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly in pipeline

- Static and dynamic prediction
  - *Required hardware support*
  - Static: profile-, program-, and programmer-based
  - Dynamic: 1-bit, 2-bit, BHT, BTB, Spatial Correlation

- Misprediction recovery mechanisms:
  - *Keep result computation separate from commit*
  - Kill instructions following branch in pipeline
  - Restore state to that following branch

Superscalar and VLIW: instruction-level parallelism

- Superscalar
  - N-wide superscalar → fetch, decode, execute, retire N instructions per cycle
  - Hardware performs the dependence checking between concurrently-fetched instructions
- VLIW
  - Very Long Instruction Word
  - Multiple operations packed into one instruction
  - Compiler
    - Schedule operations to maximize parallel execution
    - Guarantee intra-instruction parallelism and avoid data hazards in one instruction
  - Loop unrolling vs. software pipelining
    - *Software pipelining pays startup/wind-down costs only once per loop, not once per iteration*
  - Trace scheduling
    - Trace selection and compaction
  - Predicated execution
    - Control dependence converted to data dependence

Multithreading: thread-level parallelism

- Have multiple thread contexts in a single processor
  - Latency tolerance, hardware utilization, single-thread performance, etc.
- Fine-grained
  - Cycle by cycle
  - Simpler to implement, but low single thread performance
- Coarse-grained
  - Switch on event (e.g., cache miss)
  - Switch on quantum/timeout
  - Fairness among threads
- Simultaneous
  - Instructions from multiple threads executed concurrently in the same cycle
  - Dispatch instructions from multiple threads in the same cycle (to keep multiple execution units utilized)
    - Utilize functional units with independent operations from the same or different threads
  - Fetch policies
    - Round-robin, ICOUNT, etc.

Vector:

Single instruction operates on multiple data elements (SIMD)

- A vector processor is one whose instructions operate on vectors rather than scalar (single data) values
- A vector is a one-dimensional array of numbers
- No dependencies within a vector
  - Pipelining & parallelization work really well
- Each instruction generates a lot of work
  - Reduces instruction fetch bandwidth requirements
  - Highly regular memory access pattern
    - Memory (bandwidth) can easily become a bottleneck
- Vector instruction parallelism
  - Overlapped execution of multiple vector instructions
- Vector Chaining
- Vector Conditional Execution
- Vector Scatter/Gather

Synchronization

- Mutual Exclusion
  - Dekker's algorithm with shared variables for two processes
  - ISA Support for Mutual-Exclusion Locks
    - Atomic Memory Operations (AMOs)
    - Test and set, swap, acquire & release
- Nonblocking Synchronization
  - Compare and Swap
  - ABA problem
  - Load-linked & Store-conditional
- Hardware Transactional Memory (TM)
  - Operations in a TM either all succeed or are all squashed
  - Data versioning
    - Eager (undo-log) vs. lazy (write-buffer)
  - Conflict detection
    - Pessimistic detection vs. Optimistic detection

Virtual Machines

- User virtual machine = ISA + Environment
- Software interpreter
- Binary translation
- Dynamic translation
  - Transmeta Crusoe with “Code Morphing”
  - x86 → VLIW
  - Handling exception
    - Shadow registers and store buffer
- System VMs: Supporting Multiple OSs on Same Hardware
  - Hypervisor

Security & Privacy

- ROP attacks
- Side-channel attacks
  - Sound, time, power, etc.
- How to communicate between processes (victim/attacker)
  - Page faults/cache evictions/cache coherence protocols/…
  - Flush+Reload, Prime+Probe

- Meltdown
  - Leveraging out-of-order execution to dump memory
- Spectre
  - Leveraging speculative execution to leak secrets

Other topics

- I/O
  - Memory-mapped vs. I/O channels, DMA
  - Polling vs. interrupt
  - Different buses
- Disk
  - The components and mechanism of hard disk
- ECC
  - Hamming distance
  - Single-bit correction, double-bit detection
- RAID
  - RAID 0, 1, 01, 3, 4, 5, 10

Conclusion

- CA is much more complicated than what we studied in this course