3810 Review Session

Spring 2023

Hit Record!

Reminders:

- Practice exam, annotated slides, class notes, homework solutions
- Friday April 28, 8-10am. Room assignments: Last names A-R in WEB L104. Last names S-Z in WEB 2230.
- 80-20 post-pre midterm material
- Office hours: today until 11:30am, Wed and Thurs 9-11am
- No laptops/textbooks. 6 sheets + green sheet. Calculators ok.
- SoC code of conduct

Disks Basics

Non-volatile storage

- Disk access remains very slow – mechanical head that has to move to the correct “ring” of data – order of milli-seconds – high enough that a context-switch is best
- Focus on other metrics, especially reliability
- A sector on the disk is associated with a cyclic redundancy code (CRC) – a hash that tells us if the read data is correct or not – it is simply an error detector, not an error corrector

Redundant array of inexpensive disks

SSD

- To correct the error, RAID is commonly used
- Reliability measures continuous service accomplishment and is usually expressed as mean time to failure (MTTF)
- Availability is measured as MTTF/(MTTF+MTTRecovery)

Flash

$$f(D) = \underline{CRC}$$

RAID

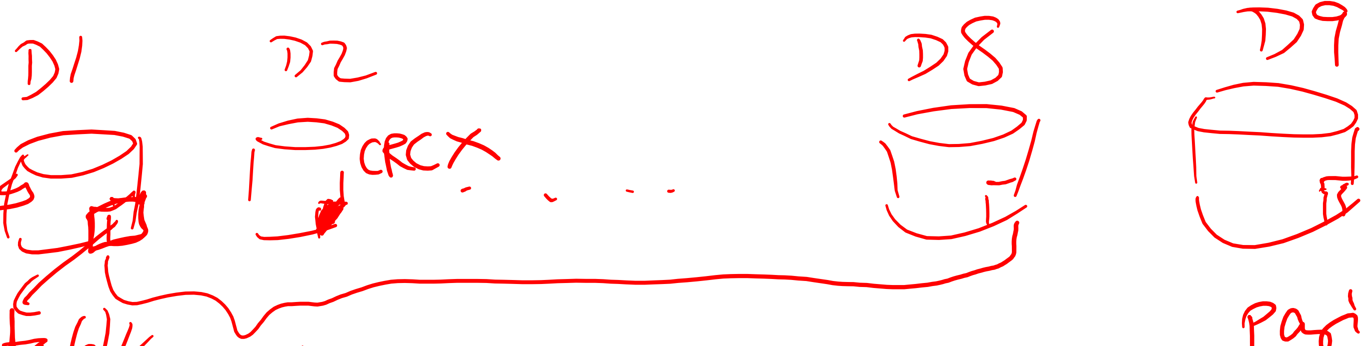
	D1	D2	D3	D4	D5	D6	D7	D8	P
	1	1	0	1	1	1	0	0	1
Rd	1	CRCX	0	1	1	1	0	0	1

100% overhead

- RAID 0: no redundancy
- RAID 1: mirroring
- RAID 2 and 6: memory-style ECC and rarely deployed
- RAID 3: bit-interleaved, lower cost, but no query-level parallelism
- RAID 4: block-interleaved, lower cost, query-level parallelism, but write bottleneck
- RAID 5: block-interleaved, lower cost, query-level parallelism, write parallelism
- Parity and XOR!

R - 8D
12.5% → W - 9D
R - 1D
W - 2D Rds
2 DWRs

parity is rotated across disks



1 data blk

Read entirely in 1 disk

Parity disk

parity = # of 1s in the data

Parity with XOR

on a write,
Rd old data & old parity
Calculate Diff. Apply diff to parity in RAID-4/5

Pipeline Performance

Unpipelined processor

CPI: 1

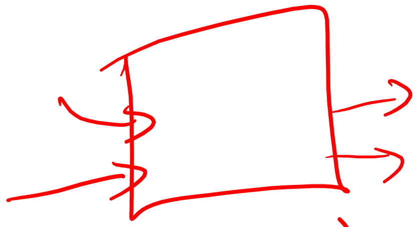
Clock speed: $\frac{1}{6ns} = 0.167GHz$ ✓

Throughput:

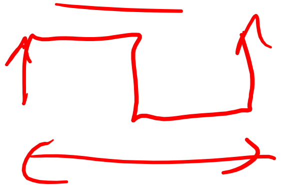
clk speed × IPC

= 0.167 GHz × 1

= 0.167 B Instr per sec (BIPS)



6ns = cycle time



Circuit Assumptions

Length of full circuit: uniform

Length of each stage: no latch overhead

No hazards stalls

latch ovhd = 0.2ns

6ns → 6.2ns (speedup ~ 7.8x)

0.6ns → 0.8ns

10-stage pipeline

Pipelined processor

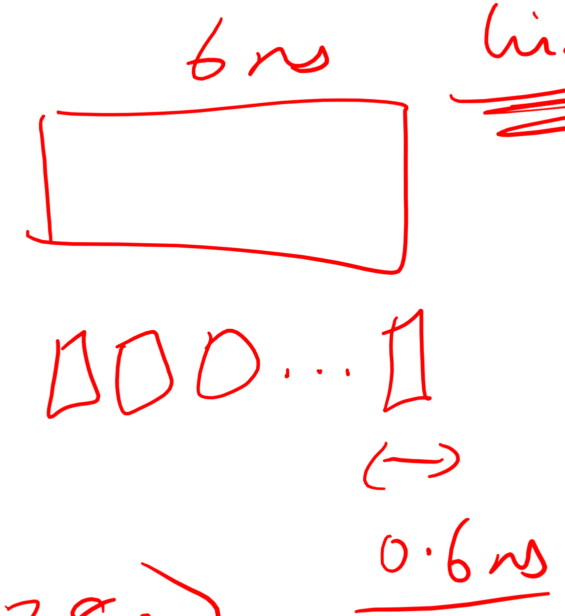
CPI: 1

Clock speed: $\frac{1}{0.6ns} = 1.67GHz$

Throughput:

1.67GHz × 1 = 1.67 BIPS

10x higher



Data Hazards

No Bypassing

(for the 5-stage pipeline) → $\frac{1}{2}$ cycle Reg Rd & Wr

Point of production: always RW middle

Point of consumption: always D/R middle

					* PoP			
I1	add:	IF	DR	AL	DM	RW		
I2	add:		IF	DR	DR	DR	AL	DM	RW
						* PoC		

2 stalls

Bypassing

Point of production:

- add, sub, etc.: end of ALU
- lw: end of DM

Point of consumption:

- add, sub, lw: start of ALU
- sw $1, 8($2): start of ALU for $2, start of DM for $1

					* PoP		
I1	add:	IF	DR	AL	DM	RW	
I2	add:		IF	DR	AL	DM	RW
					* PoC		

Control Hazards

Assumptions

ideal → 100 cycles

100 instructions

20 branches

14 Not-Taken, 6 Taken

Branch resolved in 6th cycle (penalty of 5)

Approach 1: Panic and wait

br

↳ PC+4

T

Exec time = 100 cycles (idealized) + 20 br × 5 cyc penalty = 100 + 100 = 200 cyc

Approach 2: Fetch-next-instr

Exec time = 100 + 6 Tak br × 5 cyc pen = 130 cycles

Approach 3: Branch Delay Slot

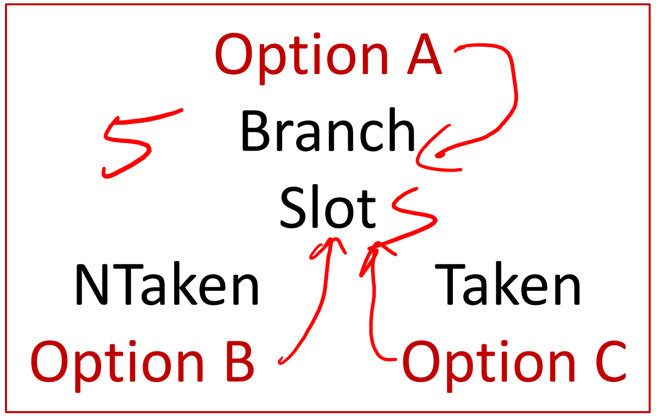
- hw/compiler

Option A: always useful — 100 cyc

Option B: useful when the branch goes along common fork — 100 + 6 × 5 = 130 cyc

Option C: useful when the branch goes along uncommon fork — 100 + 14 × 5 = 170 cyc

Option D: no-op, always non-useful — 100 + 20 × 5 = 200


Option A

5

Branch

Slot

5

NTaken

Taken

Option B

Option C

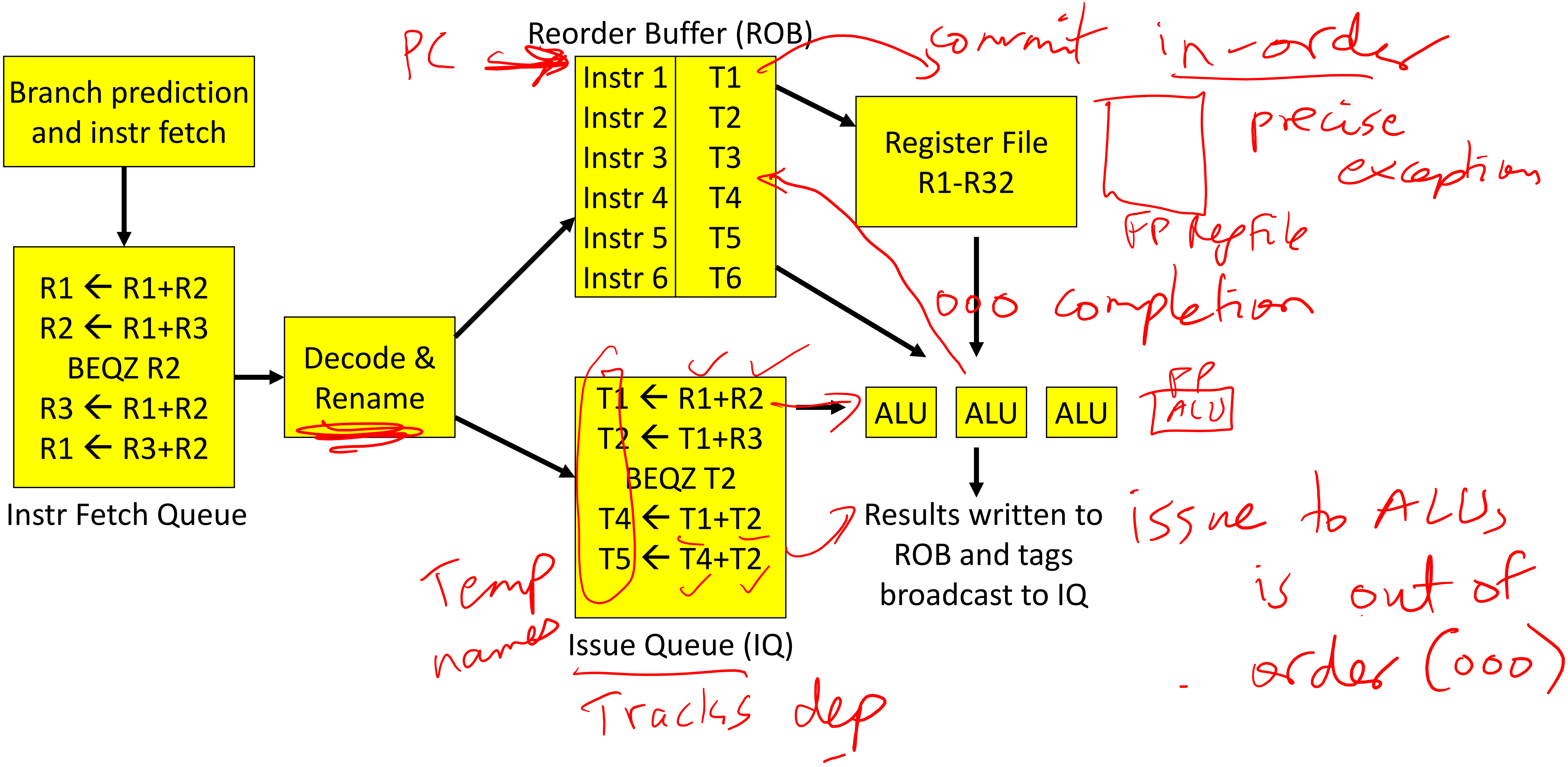
Approach 4: Branch predictor

hw-only

Accuracy of 90%

100 + 20 × 10% misses × 5 cyc

$= 100 + 20 \times \frac{1}{10} \times 5 = 110$ cyc



Branch prediction and instr fetch

R1 ← R1+R2
R2 ← R1+R3
BEQZ R2
R3 ← R1+R2
R1 ← R3+R2

Instr Fetch Queue

Decode & Rename

Reorder Buffer (ROB)

Instr 1	T1
Instr 2	T2
Instr 3	T3
Instr 4	T4
Instr 5	T5
Instr 6	T6

Register File
R1-R32

T1 ← R1+R2
T2 ← T1+R3
BEQZ T2
T4 ← T1+T2
T5 ← T4+T2

Issue Queue (IQ)

ALU ALU ALU

Results written to ROB and tags broadcast to IQ

PC

commit in-order

precise exception

FP RegFile

OOO completion

FP ALU

issue to ALUs is out of order (OOO)

Temp names

Tracks dep

Out of Order Processor

Assumptions

idealized (all L1 hits) (L1 latency already captured

1000 instructions, 1000 cycles, no stalls with L1 hits

# loads/stores: 400

% of loads/stores that show up at L2: 5%

% of loads/stores that show up at L3: 3%

% of loads/stores that show up at mem: 1%

L2 acc = 10 cyc, L3 acc = 25 cyc, mem acc = 200 cyc

5% of 400 show up in L2 = 20

3% of 400 in L3 = 12

1% of 400 in mem = 4

L1 acc = 2cyc

1000 instrs

Exec time = 1000 cyc (idealized) + 20×10 (L2) + 12×25 (spent in L3) + 4×200 (mem

= 1000 + 200 + 300 + 800 = 2300

$CPI = \frac{2300}{1000} = 2.3$

Cache Latency

Assumptions

512KB cache, 8-way set-associative, 64-byte blocks, 32-bit addresses

Cache size

Data array size = #sets x #ways x blocksize

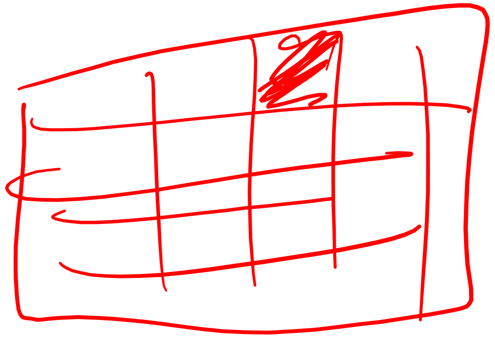
Tag array size = #sets x #ways x tagsize = ~~16~~ $2^{10} \times 2^{3} \times 16b$ $= 2B$ $= 2^{4}$ KBytes

Offset bits = log(blocksize) = 6

Index bits = log(#sets) = 10

Tag bits + index bits + offset bits = addresswidth = 16

Capacity = rows x cols x blksize



$$512KB = \#sets \times 8 \times 64B$$

$$\frac{512KB}{8 \times 64} = \#sets = \frac{2^{19}}{2^{3} \times 2^{6}} = 2^{10} = 1024 \text{ sets}$$

Cache Size

Cache Hits/Misses

Assumptions

4 index 5 off

16 sets, 1 way, 32-byte blocks

Access pattern: 4 40 400 480 512 520 1032 1540

Offset = address % 32 (address modulo 32, extract last 5)

Index = (address/32) % 16 (shift right by 5, extract last 4)

Tag = address/512 (shift address right by 9)

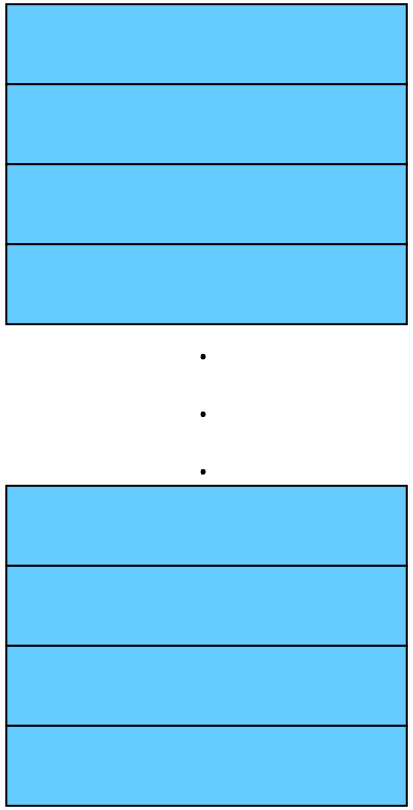
4b 5b

32-bit address

	23 bits tag	4 bits index	5 bits offset	H/M	Evicted address
4:	0	0	4	M	Inv
40:	0	1	8	M	Inv
400:	0	12	16	M	Inv
480:	0	15	0	M	Inv
512:	1	0	0	M	0
520:	1	0	8	H	-
1032:	2	0	8	M	512
1540:	3	0	4	M	1024

Example 0b

Show how the following addresses map to the cache and yield hits or misses.
The cache is direct-mapped, has 16 sets, and a 64-byte block size.
Addresses: 8, 96, 32, 480, 976, 1040, 1096



Offset = address % 64 (address modulo 64, extract last 6)
Index = address/64 % 16 (shift right by 6, extract last 4)
Tag = address/1024 (shift address right by 10)

32-bit address

	22 bits tag	4 bits index	6 bits offset	
8:	0	0	8	M
96:	0	1	32	M
32:	0	0	32	H
480:	0	7	32	M
976:	0	15	16	M
1040:	1	0	16	M
1096:	1	1	8	M

6. Consider a 4-processor multiprocessor connected with a shared bus that has the following properties: (i) centralized shared memory accessible with the bus, (ii) snooping-based MSI cache coherence protocol, (iii) write-invalidate policy. Also assume that the caches have a writeback policy. Initially, the caches all have invalid data. The processors issue the following three requests, one after the other. Similar to slide 4 of lecture 25, fill in the following table to indicate what happens for every request. Also indicate if/when memory writeback is performed. **(12 points)**

(a) P3: Read X

(b) P3: Write X

(c) P2: Write X

Request	Cache Hit/Miss	Request on bus	Who responds	State Cache 1	State Cache 2	State Cache 3	State Cache 4
				Inv	Inv	Inv	Inv
P3: Rd X	Rd Miss	Rd MissX	Mem	I	I	Sh	I
P3: Wr X	Perms Miss	Upgrade X	No one	I	I	~~Sh~~ M	I
P2: Wr X	Wr Miss	Wr Miss x	P3 responds	I	M	I	I
P1: Rdx	Rd miss	Rd MissX	P2 resp mem wb	Sh	Sh	I	I

M → S

⇒ Mem writeback

Questions to ask yourself:

How does Meltdown work?

How does Spectre work?

How can you force a footprint? (the relevant code sequence)

How can you examine footprints? (exploiting the side channel)

How can you defend against these attacks?

lw R1 ← ill

lw [R1]

prime + probe

Meltdown

↓

Disable speculation after illegal access

Spectre

↓

Partition resources

Security

Questions to ask yourself:

What does the programmer/compiler deal with?

What does the OS deal with?

How is translation done efficiently?

Page tables TLB

Virtual Memory

Questions to ask yourself:

Why do multiprocs need to deal with prog. models, coherence, synchronization, consistency?

What are race conditions?

What is an example synchronization primitive and how is it implemented?

test & set

What consistency model is assumed by a programmer?

seq cons

Why is it slow? no reordering

How do I make life easier for the programmer and provide high performance?

use locks

reorderings in most places

Synchronization, Consistency

Questions to ask yourself:

What are the central philosophies in a GPU?

In what ways does the GPU design differ from a CPU?

What are the different ways that disks provide high reliability?

Can you explain how parity is used to recover lost data?

GPUs, Disks