3810 Review Session

Spring 2024

Reminders

Student course evals

$17/hr

- Hit Record
- 2 sheets of notes (front + back), plus MIPS reference sheet
- 80% based on post-midterm material
- No phones – simple calculators allowed – ok to leave solution as an expression
- Studying strategy: focus on homeworks, then go through annotated slides, refer videos when things are unclear
- My office hours: Thurs 9-11, Fri 10:30-11:30

	A	A-	B+	B	B-	C+/C/C-	D+ or worse
Percentile	Top 16%	32%	45%	58%	71%	86%	
Rank	47	94	133	172	211	256	295
Midterm Score	83.5	76.25	69.5	64	57	48	
HW 1-9 (top 7)	103	99	95.7	93	88.5	80	

Disk Basics

- 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
- 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)

RAID

- 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!

Unpipelined processor

CPI: 1

Clock speed: $\frac{1}{\text{cycle time}} = \frac{1}{5\,ns} = 0.2\,GHz$

Throughput:

clk speed x IPC = 0.2 GHz x 1

= 0.2 BIPS

Pipelined processor

CPI: 1

Clock speed: $\frac{1}{\text{cycle time}} = \frac{1}{0.8\,ns} = 1.25\,GHz$

Throughput:

= 1.25 GHz x 1

= 1.25 BIPS

Speedup = $\frac{1.25}{0.2} = 6.25x$

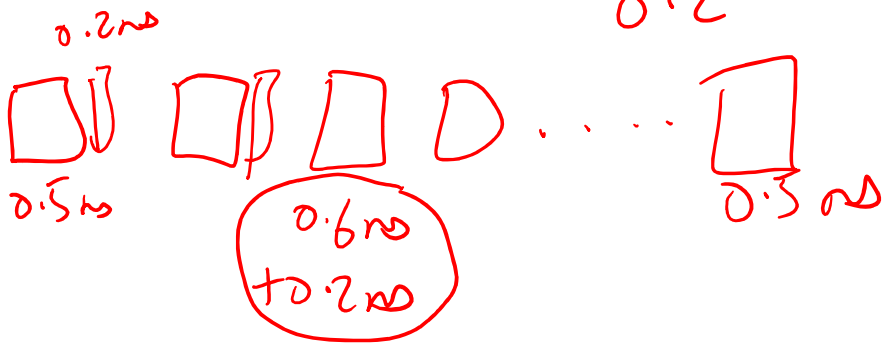
Circuit Assumptions

Length of full circuit:

Length of each stage:

No hazards

10-stage pipe



Pipeline Performance

Data Hazards

No Bypassing

(for the 5-stage pipeline)

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
```

Assumptions

100 instructions
20 branches
14 Not-Taken, 6 Taken
Branch resolved in 6th cycle (penalty of 5)

Approach 1: Panic and wait

Cycle = 100 + 20 x 5 = 200 cyc

CPI = 2.0

Approach 2: Fetch-next-instr

PC+4

Cycle = 100 + 6 x 5 = 130 cyc

CPI = 1.3

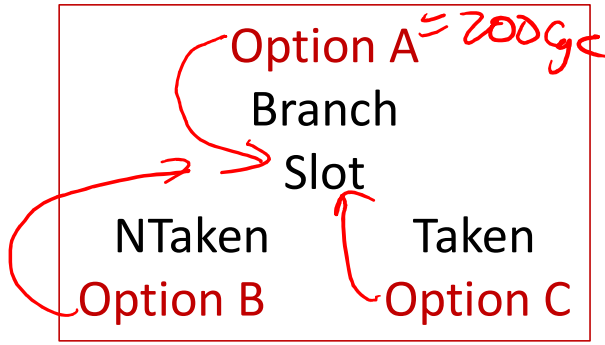
Approach 3: Branch Delay Slot

Option A: always useful

Option B: useful when the branch goes along common fork

Option C: useful when the branch goes along uncommon fork

Option D: no-op, always non-useful



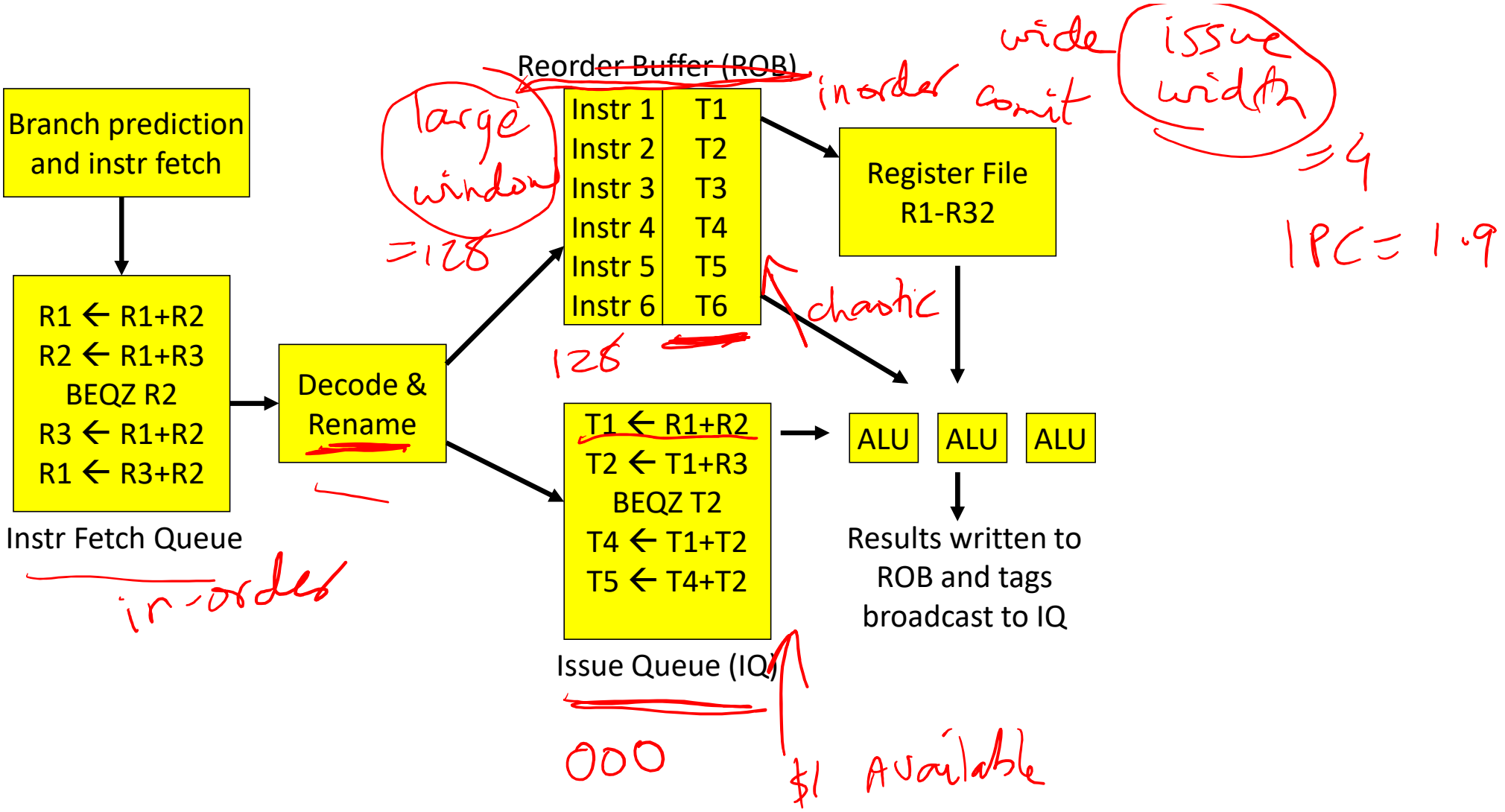
Approach 4: Branch predictor

Accuracy of 90%

Cycle = 100 + 20 x 10% x 5
= 100 + 10 = 110 cyc

CPI = 1.1

Control Hazards



Out of Order Processor

Assumptions

1000 instructions, 1000 cycles, no stalls with L1 hits
loads/stores: 40% 400 lds/sts
% of loads/stores that show up at L2: 10%
% of loads/stores that show up at L3: 5%
% of loads/stores that show up at mem: 2%
L2 acc = 10 cyc, L3 acc = 25 cyc, mem acc = 200 cyc

$$1000 + (10\% \text{ of } 400) \times 10 + (5\% \text{ of } 400) \times 25 + (2\% \text{ of } 400) \times 200$$

$$= 1000 + 400 + 500 + (8 \times 200)$$

$$= 3500$$

$$CPI = 3.5$$

Cache Latency

Assumptions

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

Data array size = #sets x #ways x blocksize

Tag array size = #sets x #ways x tagsize

Offset bits = log(blocksize)

Index bits = log(#sets)

Tag bits + index bits + offset bits = addresswidth

$1024 \times 8 \times 16b = 128Kb = 16KB$

$= \log_2(64) = 6$

$= \log 1024 = 10$

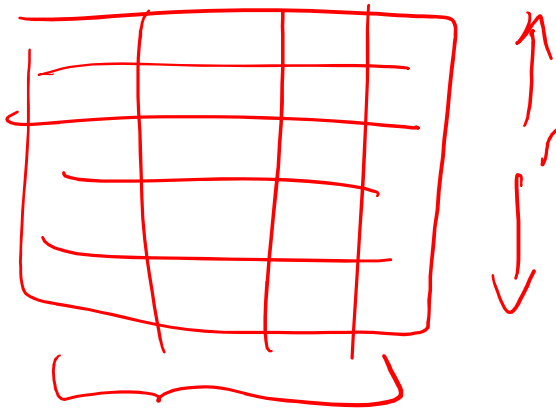
16b tags

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

$2^9 \times 2^{10} = sets \times 2^3 \times 2^6$

$sets = 2^{9+10-3-6} = 2^{10}$

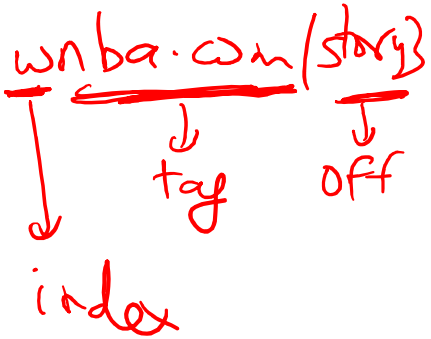
$= 1024$



Assumptions

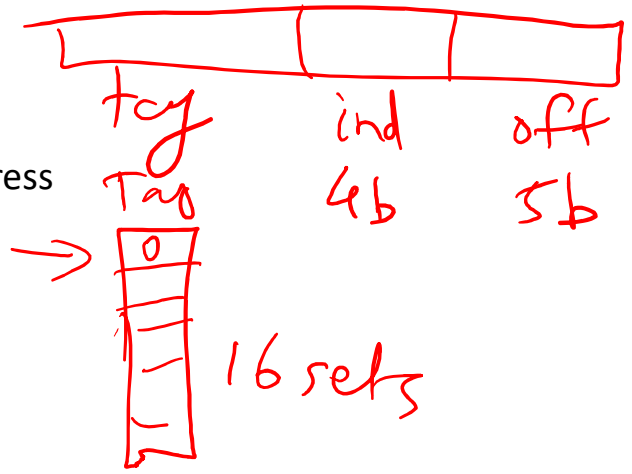
16 sets, 1 way, 32-byte blocks

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)

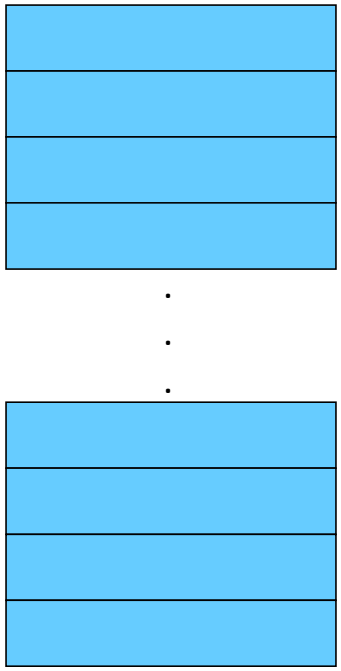
	32-bit address				
	23 bits tag	4 bits index	5 bits offset	H/M	Evicted address
4:	0	0	4	M	Inv
40:	0	1	8	M	Inv
400:	0	12	16	M	Inv
480:	0	15	0	M	Inv
512:	1	0	0	M	0
520:	1	0	8	H	-
1032:	2	0	8	M	512
1540:	3	0	4	M	1024



Cache Hits/Misses

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

Consider a 3-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 17 of lecture 25, fill in the following table to indicate what happens for every request. Also indicate if/when memory writeback is performed. (8 points)

P2: Read X
P1: Read X
P2: Write X
P3: Read X

Request	Cache Hit/Miss	Request on the bus	Who responds	State in Cache 1	State in Cache 2	State in Cache 3	State in Cache 4
				Inv	Inv	Inv	Inv
P2: Rd X	Data Miss	Rd Miss	Mem		S		
P1: Rd X	Data miss	Rd Miss	Mem	S	S		
P2: Wr X	Perms miss	Upgrade	No one	I	M		
P3: Rd X	Data miss	Rd Miss	Cache 2 responds		S	S	
			Mem writeblk	(M → S)			

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?

Recall that Meltdown and Spectre both rely on a sequence of priming the cache by the attacker, a secret-dependent footprint in the cache, and the attacker finally doing a timing probe in the cache to detect a cache miss and the location of the footprint. In Meltdown, the attacker does a load to an illegal location which will eventually be squashed. But before the squashing, it uses the secret value as the address for a load and leaves a footprint in the cache. In Spectre, the attacker runs alongside a program that is naturally leaking secrets. The attacker can also force the victim program to leak more secrets by training the branch predictor and forcing the victim to execute a leaky sequence (R1 ← secret; lw …, [R1]).

Questions to ask yourself:
What does the programmer/compiler deal with?
What does the OS deal with?
How is translation done efficiently?

The programmer/compiler have no idea who will run alongside a program. They have no option but to assume that all the memory belongs to them. So they assume the abstraction of virtual memory, i.e., each program has access to a contiguous 4GB memory. In reality, 100 programs might run together and the system may only support (say) 16 GB of physical memory. The OS maps each program's virtual memory pages to physical memory pages. It keeps track of a page table per program to remember this mapping. Since the page table is itself large, it needs to be cached. The TLB is a hardware structure that serves as a page table cache and is accessed within a cycle while performing loads/stores. The cache is virtually indexed and physically tagged so that the TLB access can happen in parallel with cache access (while also not suffering from aliasing – multiple threads referring to the same physical memory location with different names).

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?

What consistency model is assumed by a programmer?

Why is it slow?

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

Multi-threaded applications have many threads spreak across many cores that access shared data. Like in the atm-deposit example, threads should coordinate or synchronize when accessing shared data. This is usually done with locks to ensure that only one thread is messing with data at a time. Locks are implemented with a special hardware instruction, the test-and-set. Coherence ensures that updates to cached values are also reflected in other caches and exposed to other threads. The consistency model specifies the programming model and hardware optimizations. For example, sequential consistency is a simple programming model, but it requires the hardware to not do any re-ordering. Relaxed consistency is a better consistency model since it imposes a small programming burden (programmers must avoid races by using locks around shared data access) while allowing the hardware to do re-orderings for the most part.

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?

A GPU is essentially parallel vector operations on steroids. So the GPU chip is packed with number-crunching units and very little cache. When there is a cache miss (which happens often), the GPU just switches the task (warp) and works on something else because it's trying to be a number-crunching beast. So, context switches between warps are very common in a GPU. To make the context switch lightweight, the register file has to be large enough to accommodate the register operands of several warps (otherwise, the context switch would require copies between the register file and memory, which would take way too long).

GPUs, Disks