



A4 Recitation 9? Test 2 Prep

I have no idea what number we're on
Still Sanjana and Jacob
Turn in HW at the front!



Pipeline Hazards

1. Compare/Contrast Structural Hazards, Control Hazards, and Data Hazards. Give an example of each.
2. I1: $R1 \leftarrow R2 + R3$; I2: $R4 \leftarrow R1 + R6$. If I2 is immediately following I1 in the pipeline with no forwarding, how many bubbles will there be? How about with register forwarding?
3. I1: LOAD R1, 0(R2); I2: $R4 \leftarrow R1 + R6$. If I2 is immediately following I1 in the pipeline with no forwarding, how many bubbles will there be? How about with register forwarding?
4. How are discontinuities handled in a pipelined processor? List the steps, starting from the time the processor receives the interrupt to the beginning of the execution of the handler code.



Pipeline Hazards - Answers

1. Structural hazards - Caused by hardware limitations. Ex: BEQ causing a bubble due to only 1 ALU in EX stage.
Data Hazards - RAW, WAW, WAR. Dependencies between instructions.
Control Hazards - Don't know the outcome of a branch instruction until EX, but we have to keep fetching instructions.
2. 3 without forwarding, 0 with.
3. 3 without forwarding, 1 with.
4. Steps are as follows:
 - a. Halt fetching of instructions by IF (IF generates bubbles)
 - b. Let partially executed instructions already in the pipeline finish (drain the pipeline)
 - c. Save PC into \$k0, PC <- IVT[Interrupt Vector], Disable Interrupts
 - d. Save mode on kernel stack, switch to kernel mode
 - e. Resume Execution




Pipeline Hazards

Assume we have a five stage pipelined processor as described in the textbook **with no data forwarding**.

1. Identify any and all data hazards in this program, and their type.
2. Complete the waterfall diagram for this program until the last instruction has reached the WB stage.
3. Compute the CPI for these instructions.

```
ADD R1, R3, R4
ADDI R2, R1, 4
SUB R3, R8, R5
NAND R3, R6, R7
ADD R6, R5, R4
```



IF	ID/RR	EX	MEM	WB
ADD				
ADDI	ADD			
SUB	ADDI	ADD		
SUB	ADDI	NOP	ADD	
SUB	ADDI	NOP	NOP	ADD
SUB	ADDI	NOP	NOP	NOP
NAND	SUB	ADDI	NOP	NOP
ADD	NAND	SUB	ADDI	NOP
-	ADD	NAND	SUB	ADDI
-	-	ADD	NAND	SUB
-	-	-	ADD	NAND
-	-	-	-	ADD

1.

- ADDI: RAW,
- NAND: WAW
- SUB, 2nd ADD: WARs

2.

On the left
(RAW is the only one of all of the hazards here that results in bubbles. Need to wait til write back **completes** -> 3 bubbles)

3.

Num cycles / num instructions =
 $12 / 5 = 2.4$



Branching Methods

1. Which method has the compiler find instructions that are benign from the p.o.v. of branch semantics to execute immediately after a branch until the decision is certain?
 - a. Conservative branching
 - b. Delayed branching
 - c. Branch prediction (no target buffer)
 - d. Stalled branching
2. Why is the above method not a great idea?
3. With the Branch Target Buffer, in which pipeline stage is the BTB looked up? Why? How many pipeline bubbles are incurred upon an incorrect prediction?



Branching Methods - Answers

1. Delayed Branching
2. With the increase in depth of pipelines of modern processors, it becomes increasingly difficult to fill the delay slots by the compiler; limits microarchitecture evolution due to backward compatibility restrictions; it makes the compiler not just ISA-specific but implementation specific.

****Pros of delayed branching:** No need for any additional hardware for either stalling or flushing instructions; It involves the compiler by exposing the pipeline delay slots and takes its help to achieve good performance.

3. IF stage. If the lookup is successful (i.e., the table contains the address of the branch instruction currently being fetched and sent down the pipeline), then the IF stage can start fetching the target of the branch immediately. In this case, there will be no bubbles in the pipeline due to the branch instruction (if the prediction is correct).. Upon an incorrect prediction, instructions in IF and ID/RR are flushed when the BEQ decision is made in EX → 2 NOPs

****Pros of conservative branching:** no hardware needed for flushing

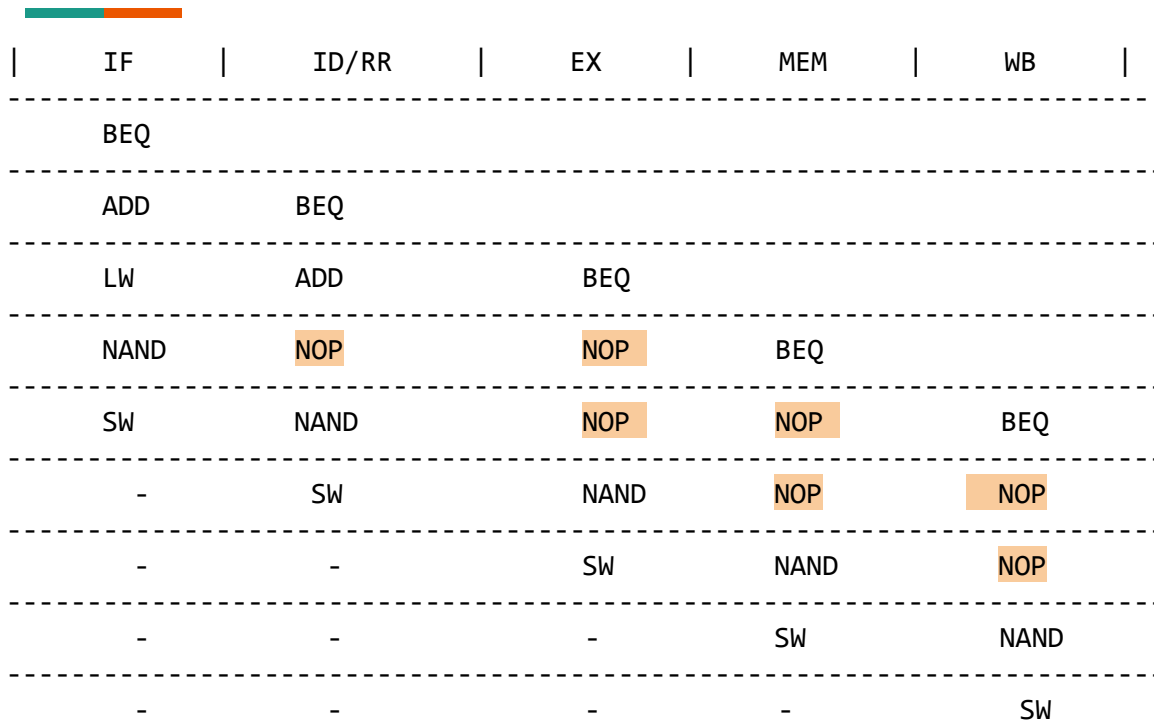


Branch Prediction

Assume we have a processor which uses **branch prediction** assuming **the sequential path to be the probable one**. In the actual execution of the code snippet below, **the prediction turns out to be false**.


1. Fill out the waterfall diagram, completing it until the last above instruction reaches the WB stage.
2. What is the observed CPI for the instructions that actually get executed?

```
BEQ L1
ADD
LW
...
L1 NAND
SW
```

IF	ID/RR	EX	MEM	WB
BEQ				
ADD	BEQ			
LW	ADD	BEQ		
NAND	NOP	NOP	BEQ	
SW	NAND	NOP	NOP	BEQ
-	SW	NAND	NOP	NOP
-	-	SW	NAND	NOP
-	-	-	SW	NAND
-	-	-	-	SW

2 NOPs because **after** BEQ's decision is resolved in the EX stage, the pipeline flushes LW and AND and fetches the actual instruction (NAND), which begins execution in the **next** cycle.

- 
1. The phenomena where a single long running process can cause adverse performance by blocking shorter running processes is called ____ **Convoy effect** ____.
 2. Which algorithm determines which process to run based on arrival time? ____ **FCFS** ____.
 3. Which algorithm determines which process to run based on running time? ____ **SJF** ____.
 4. Which algorithm determines which process to run based on some measure of criticality? **Priority**.
 5. The phenomena where a process may never get run because the scheduling algorithm always selects someone else to run is called ____ **Starvation** ____.
 6. Which algorithms we've discussed result in starvation? ____ **SJF, SRTF, Priority** ____.
 7. Which algorithms we've discussed result in overhead with context switching? ____ **Round robin** ____.
 8. T/F:
 - a. the dispatcher balances the mix of processes in memory to avoid thrashing. **_F_**
 - b. the scheduler populates the CPU registers with the selected algorithm's process state. **_F_**



Memory Management/Paging

1. Quick Maths - Imagine we have a system with a 48 bit virtual address and a 32 bit physical address. The pagesize is 4KB. How many entries will be in each page table? How many physical frames will there be?
2. What additional information needs to be carried along in a pipelined processor in order for demand paging to work?
3. Why does CPU utilization increase as we increase the degree of multiprogramming? Why does it decrease past a certain 'inflection point'?
4. What is the TLB a special case of? What is it used for? How does it help performance?
5. Does paging suffer from external fragmentation or internal fragmentation? Which is preferred?



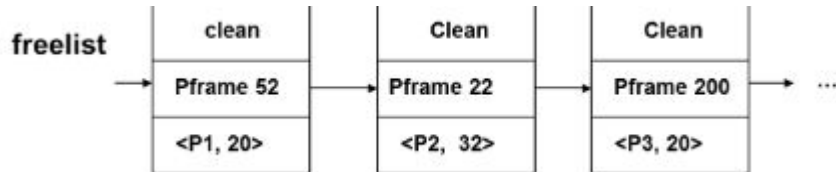
Memory Management/Paging - Answers

1. 4KB = 4096. $\log(4096) = 12$, so 36 bits for vpn, 20 bits for physical frame. 2^{36} page table entries, 2^{20} physical frames.
2. The PC value of the instruction, so the OS knows where to restart the process when a page fault occurs
3. Increases because processes normally switch between CPU and IO bursts, so having more processes ready to go on the CPU will increase utilization. Beyond a certain point, however, there is not enough memory to go around to keep every process's working set in memory. This will cause a lot of page faults, which results in thrashing = not much work getting done on the CPU
4. TLB is a special case of a CACHE. It stores recent virtual address -> physical address translations. Without a TLB, we would have to make two trips to memory for every memory access (remember that page tables are in memory). Memory trips are expensive, so we can take advantage of the principle of locality to improve performance
5. Internal fragmentation. However, this isn't as big of a deal as external fragmentation, because the pagesize is a parameter of the operating system we can optimize. External fragmentation is harder to deal with - would have to re-arrange processes in memory (very expensive!)

Memory Management/Paging

Inspect the freelist shown below. Each entry in the freelist, shows the page frame number that is available for allocation as well as the reverse mapping, i.e., the process-ID and the VPN of that process that was hosted in this page frame.

Imagine P3 is running on the processor, and incurs a page fault at VPN 20. Explain how the memory manager will service this page fault.



Answers:

Memory manager sees 200 contains the contents of P3's VPN 20

The node is removed from the freelist

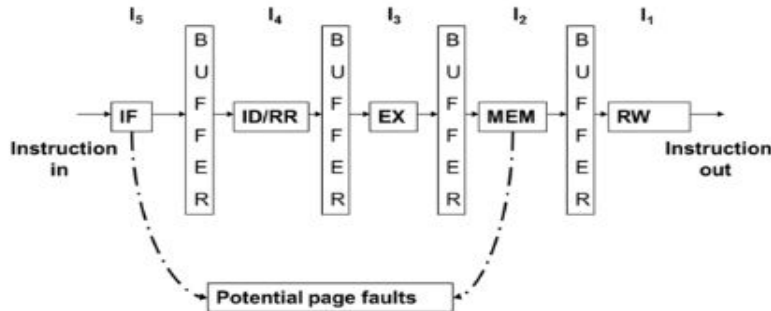
Page table of VPN = 20 is set to 200

Valid bit of VPN = 20 is set

Memory Management/Paging

Examine the state of a pipelined processor shown below. Assume this processor demand-paging virtual memory management. **Instruction I2 incurs a page fault.**

1. What will happen to the current instructions in the pipeline?
2. From which instruction will execution be resumed after the page fault is serviced?



Answers:

1. I1 will finish, I3, I4, and I5 will be squashed
2. I2



Caching

1. What is the principle of locality? What does spatial locality suggest? Temporal locality?
2. In how many cache blocks can a particular memory location appear in a direct mapped cache?
3. In a direct mapped cache with a t bit tag, how many comparators will there be? How many bits will each be?
4. In how many cache blocks can a particular memory location appear in a fully associative cache?
5. In a fully associative cache with 64K bytes of data, 64 bytes per block, and a t -bit tag, how many comparators will there be? How many bits will each be?
6. In how many cache blocks can a particular memory location appear in a n -way set associative cache?
7. In a 4-way set associative cache with 64K bytes of data, 64 bytes per block, and a t -bit tag, how many comparators will there be? How many bits will each be?



Caching - Answers

1. Principle of locality simply states that programs tend to operate in a small region of memory at any given time. Spatial locality suggests that programs will likely access adjacent memory locations, and thus we should bring in adjacent memory locations in the cache upon a miss. Temporal locality suggests that programs will likely access the area of memory it is currently accessing again in the near future, and thus we should keep it in the cache as long as possible.
2. 1. There's a 1 to 1 mapping of memory locations to cache locations in a direct mapped cache.
3. There will be a single t-bit comparator.
4. It can appear in any of the cache blocks. There is no specified cache location for any memory location in a fully associative cache.
5. There will be 1000 (one for each block) t-bit comparators. Since an entry can be in any block in the cache, we have to check all of them.
6. It can appear in n different locations.
7. There will be 4 t-bit comparators.



Assume the cache configuration:

- Total Size : 1 MB (2^{20} bytes)
- 4-way set associativity
- Each block is 256 B

Questions:

1. How many cache blocks are there in total?
2. How many blocks are in each cache line?
3. How many cache lines are there?

Answers:

1. Total size / block size = 2^{12}
2. Blocks in a line = associativity = 4
3. Num Lines =
num total blocks / associativity =
 2^{10}



In a pipelined processor

- Average CPI = 1.3 without stalls
- I-cache hit rate = 98%
- D-cache hit rate = 99%
- Memory reference instructions = 30% of all instructions
- Out of these memory reference instructions 80% are loads, 20% are stores
- Read miss penalty (instruction or data) = 100 cycles
- Write miss penalty = 5 cycles

What is the effective CPI of the processor?



Answer to prev. slide

Effective CPI = Average CPI + Memory Stalls

$$\text{EMAT} = T_c + m * T_m$$

Memory Stalls = I-cache miss penalty + D-cache miss penalty

I-cache miss penalty = I-Cache miss rate * read-miss penalty

$$= 0.02 * 100 = 2$$

D-cache miss penalty = fraction of memory reference instructions


* D-cache miss rate * (fraction of loads * read-miss penalty +
fraction of stores * write-miss penalty)

$$= 0.3 * 0.01 * (0.8 * 100 + 0.2 * 5)$$

$$= 0.003 * (81)$$

$$= 0.243$$

$$\text{Effective CPI} = 1.3 + 2 + 0.243 = 3.543$$



Consider a 16-way set associative cache with the following parameters:

- Cache size (i.e, the amount of actual data it can hold) of 1 Mbytes
- 32-bit byte-addressable memory
- Each memory word contains 4 bytes
- cache block size is 64 bytes
- Write-back policy
- One dirty bit per **word**.
- One valid bit per **block**.
- MRU field that records the most recently used cache for that cache line

1. How many cache lines are there?
2. How many tag / index / offset bits are there?
3. How many meta-data bits are there TOTALLY for the ENTIRE cache?



Answer to prev. slide

Total number of cache blocks = cache size/block size = 1 Mbyte/64 bytes = 2^{14}

Number of lines in each cache = total number of cache blocks/associativity = $2^{14} / 2^4 = 2^{10}$
= 1024

Index = log cache lines = 10

Offset = log(base2) 64 = 6

Tag = 32 - 16 = 16

MRU bits per line = log 16 = 4

Each block: 1 valid, 16 dirty (64/4), 16 tag

Each line = $(16 * 33) + 4$

Entire cache = $1024 * ((16 * 33) + 4) = 544,768$



TLB + Cache

Consider the following memory hierarchy:

- A 128 entry fully associative TLB split into 2 halves; one-half for user processes and the other half for the kernel.
- The TLB has an access time of 1 cycle.
- The hit rate for the TLB is 95%.
- A miss results in a main memory access to complete the address translation.
- An L1 cache with a 1 cycle access time, and 99% hit rate.
- An L2 cache with a 4 cycle access time, and a 90% hit rate.
- An L3 cache with a 10 cycle access time, and a 70% hit rate.
- A physical memory with a 100 cycle access time

Compute the average memory access time for this memory hierarchy. Note that the page table entry may itself be in the cache.



Answer to prev. slide

Recall from Section 9.4:

$$\text{EMAT}_i = T_i + m_i * \text{EMAT}_{i+1}$$

$\text{EMAT}_{\text{physical memory}} = 100$ cycles.

$$\text{EMATL3} = 10 + (1 - 0.7) * 100 = 40 \text{ cycles}$$

$$\text{EMATL2} = (4) + (1 - 0.9) * (40) = 8 \text{ cycles}$$

$$\text{EMATL1} = (1) + (1 - 0.99) * (8) = 1.08 \text{ cycles}$$

$$\text{EMATTLB} = (1) + (1 - 0.95) * (1.08) = 1.054 \text{ cycles}$$

$$\text{EMATHierarchy} = \text{EMATTLB} + \text{EMATL1} = 1.054 + 1.08 = 2.134 \text{ cycles.}$$



Info slide: Cache Writes

Write Through:

Every time you write to a cache block you also write to actual memory using a hardware write buffer

1. On every write, update the tag field and set the valid bit
2. CPU sends address and data to memory into write buffer. If buffer is full, write stall incurred.
3. Write buffer completes the write to Mem independent of the CPU.

Write Back:

Update only the cache upon a write

1. Cache update tag field and set the valid bit and dirty bit.
 - a. Clears the dirty bit when bringing a memory location into the block
 - b. Sets the bit on a write
 - c. Writes back the data upon replacement.

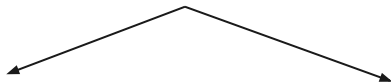
****notice the similarity of all this to virtual memory?**



Cache Writes Cont.

Write Through:

Every time you write to a cache block you also write to actual memory using a hardware write buffer



Write Allocate (usual)

On a write miss, assume the data being written to will be needed soon and bring the block into the cache

No-write allocate (unusual)

On a write miss, just put the data in the write buffer and don't bring the block into the cache.

Write Back:

Update only the cache upon a write
Cache update tag field and set the valid bit and dirty bit.

- Clears the dirty bit when bringing a memory location into the block
 - Sets the bit on a write
 - Writes back the data upon replacement.
- **notice the similarity of all this to virtual memory?**