Unpipelined processor
CPI:
Clock speed:
Throughput:

Pipelined processor
CPI:
Clock speed:
Throughput:

Circuit Assumptions
Length of full circuit:
Length of each stage:
No hazards

Pipeline Performance
No Bypassing

Point of production: always RW middle
Point of consumption: always D/R middle

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

Data Hazards
Control Hazards

Assumptions

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

Approach 1: Panic and wait

Approach 2: Fetch-next-instr

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%
Out of Order Processor

Branch prediction and instr fetch

Instr Fetch Queue

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

Decode & Rename

Instr 1
Instr 2
Instr 3
Instr 4
Instr 5
Instr 6

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

Reorder Buffer (ROB)

Instr 1
Instr 2
Instr 3
Instr 4
Instr 5
Instr 6

T1
T2
T3
T4
T5
T6

Register File R1-R32

Out of Order Processor
Assumptions

1000 instructions, 1000 cycles, no stalls with L1 hits

# loads/stores:
% of loads/stores that show up at L2:
% of loads/stores that show up at L3:
% of loads/stores that show up at mem:
L2 acc = 10 cyc,  L3 acc = 25 cyc,  mem acc = 200 cyc
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
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)

<table>
<thead>
<tr>
<th>32-bit address</th>
<th>23 bits tag</th>
<th>4 bits index</th>
<th>5 bits offset</th>
<th>H/M</th>
<th>Evicted address</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>M</td>
<td>Inv</td>
</tr>
<tr>
<td>40:</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>M</td>
<td>Inv</td>
</tr>
<tr>
<td>400:</td>
<td>0</td>
<td>12</td>
<td>16</td>
<td>M</td>
<td>Inv</td>
</tr>
<tr>
<td>480:</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td>M</td>
<td>Inv</td>
</tr>
<tr>
<td>512:</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>M</td>
<td>0</td>
</tr>
<tr>
<td>520:</td>
<td>1</td>
<td>0</td>
<td>8</td>
<td>H</td>
<td>-</td>
</tr>
<tr>
<td>1032:</td>
<td>2</td>
<td>0</td>
<td>8</td>
<td>M</td>
<td>512</td>
</tr>
<tr>
<td>1540:</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>M</td>
<td>1024</td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>Address</th>
<th>32-bit address</th>
<th>22 bits tag</th>
<th>4 bits index</th>
<th>6 bits offset</th>
<th>Cache State</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>0</td>
<td>1</td>
<td>32</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>0</td>
<td>32</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>480</td>
<td>0</td>
<td>7</td>
<td>32</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>976</td>
<td>0</td>
<td>15</td>
<td>16</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>1040</td>
<td>1</td>
<td>0</td>
<td>16</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>1096</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>M</td>
<td></td>
</tr>
</tbody>
</table>
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) snoop-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

<table>
<thead>
<tr>
<th>Request</th>
<th>Cache Hit/Miss</th>
<th>Request on bus</th>
<th>Who responds</th>
<th>State Cache 1</th>
<th>State Cache 2</th>
<th>State Cache 3</th>
<th>State Cache 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3: Rd X</td>
<td></td>
<td></td>
<td></td>
<td>Inv</td>
<td>Inv</td>
<td>Inv</td>
<td>Inv</td>
</tr>
<tr>
<td>P3: Wr X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Wr X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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?
Questions to ask yourself:
What does the programmer/compiler deal with?
What does the OS deal with?
How is translation done efficiently?
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?
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?