**Review: Processor Pipelining (1/2)**

- “Pipeline registers” are added to the datapath/controller to neatly divide the single cycle processor into “pipeline stages”.

- **Optimal Pipeline**
  - Each stage is executing part of an instruction each clock cycle.
  - One inst. finishes during each clock cycle.
  - On average, execute far more quickly.

- **What makes this work well?**
  - Similarities between instructions allow us to use same stages for all instructions (generally).
  - Each stage takes about the same amount of time as all others: little wasted time.
Review: Pipeline (2/2)

• Pipelining is a BIG IDEA
  • widely used concept

• What makes it less than perfect?
  • Structural hazards: Conflicts for resources. Suppose we had only one cache?
    \[ \Rightarrow \] Need more HW resources
  • Control hazards: Branch instructions effect which instructions come next.
    \[ \Rightarrow \] Delayed branch
  • Data hazards: Data flow between instructions. \[ \Rightarrow \] Forwarding
Review

• Some fixes to hazards
  • Illusion of two memories
  • Register file convention
  • Forwarding
  • Load delay slot
  • All else fails, bubble/stall

• Latency vs throughput

• What prevents us from getting n-times speedup, where n is the number of pipeline stages?

Graphical Pipeline Representation

(In Reg, right half highlight read, left half write)

Time (clock cycles)
Control Hazard: Branching (1/8)

Where do we do the compare for the branch?

Control Hazard: Branching (2/8)

• We had put branch decision-making hardware in ALU stage
  • therefore two more instructions after the branch will *always* be fetched, whether or not the branch is taken

• Desired functionality of a branch
  • if we do not take the branch, don’t waste any time and continue executing normally
  • if we take the branch, don’t execute any instructions after the branch, just go to the desired label
Control Hazard: Branching (3/8)

• Initial Solution: Stall until decision is made
  • insert “no-op” instructions (those that accomplish nothing, just take time) or hold up the fetch of the next instruction (for 2 cycles).
  • Drawback: branches take 3 clock cycles each (assuming comparator is put in ALU stage)

Control Hazard: Branching (4/8)

• Optimization #1:
  • insert special branch comparator in Stage 2
  • as soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
  • Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
  • Side Note: This means that branches are idle in Stages 3, 4 and 5.
Control Hazard: Branching (5/8)

- Branch comparator moved to Decode stage.

Control Hazard: Branching (6a/8)

- User inserting no-op instruction

  • Impact: 2 clock cycles per branch instruction ⇒ slow
Control Hazard: Branching (6b/8)

- Controller inserting a single bubble

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>IS → Reg</td>
</tr>
<tr>
<td>beq</td>
<td>IS → Reg</td>
</tr>
<tr>
<td>lw</td>
<td>IS → Reg, D$ → Reg</td>
</tr>
</tbody>
</table>

- Impact: 2 clock cycles per branch instruction ⇒ slow

Control Hazard: Branching (7/8)

- Optimization #2: Redefine branches
  - Old definition: if we take the branch, none of the instructions after the branch get executed by accident
  - New definition: whether or not we take the branch, the single instruction immediately following the branch gets executed (called the branch-delay slot)

- The term “Delayed Branch” means we always execute inst after branch

- This optimization is used on the MIPS
Control Hazard: Branching (8/8)

- Notes on Branch-Delay Slot
  - Worst-Case Scenario: can always put a no-op in the branch-delay slot
  - Better Case: can find an instruction preceding the branch which can be placed in the branch-delay slot without affecting flow of the program
    - re-ordering instructions is a common method of speeding up programs
    - compiler must be very smart in order to find instructions to do this
    - usually can find such an instruction at least 50% of the time

Jumps also have a delay slot...

Example: Nondelayed vs. Delayed Branch

Nondelayed Branch

```
or $8, $9, $10
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
xor $10, $1, $11
```

Delayed Branch

```
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
or $8, $9, $10
xor $10, $1, $11
```

Exit:

Exit:
**Out-of-Order Laundry: Don’t Wait**

A depends on D; rest continue; need more resources to allow out-of-order

**Superscalar Laundry: Parallel per stage**

More resources, HW to match mix of parallel tasks?
Superscalar Laundry: Mismatch Mix

<table>
<thead>
<tr>
<th>Time</th>
<th>Task Order</th>
<th>Task Mix</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 PM</td>
<td>30 30 30 30 30 30</td>
<td>(light clothing)</td>
</tr>
<tr>
<td>7 PM</td>
<td>30 30 30 30 30 30</td>
<td>(light clothing)</td>
</tr>
<tr>
<td>8 PM</td>
<td>30 30 30 30 30 30</td>
<td>(light clothing)</td>
</tr>
<tr>
<td>9 PM</td>
<td>30 30 30 30 30 30</td>
<td>(dark clothing)</td>
</tr>
<tr>
<td>10 PM</td>
<td>30 30 30 30 30 30</td>
<td>(light clothing)</td>
</tr>
<tr>
<td>11 PM</td>
<td>30 30 30 30 30 30</td>
<td>(light clothing)</td>
</tr>
<tr>
<td>12 AM</td>
<td>30 30 30 30 30 30</td>
<td>(light clothing)</td>
</tr>
</tbody>
</table>

Task mix underutilizes extra resources

Real-world pipelining problem

- You’re the manager of a HUGE assembly plant to build computers.
- Main pipeline
  - 10 minutes/pipeline stage
  - 60 stages
  - Latency: 10hr

Problem: need to run 2 hr test before done..help!
Real-world pipelining problem solution 1

• You remember: “a pipeline frequency is limited by its slowest stage”, so...

• Main pipeline
  - 2 hours/pipeline stage
  - 60 stages
  - Latency: 120 hours

Problem: need to run 2 hr test before done..help!

Real-world pipelining problem solution 2

• Create a sub-pipeline!

• Main pipeline
  - 10 minutes/pipeline stage
  - 60 stages

2 hr test (12 CPUs in this pipeline)
Peer Instruction (1/2)

Assume 1 instr/clock, delayed branch, 5 stage pipeline, forwarding, interlock on unresolved load hazards (after $10^3$ loops, so pipeline full)

Loop:

1. lw $t0, 0($s1)
2. addu $t0, $t0, $s2
3. sw $t0, 0($s1)
4. addiu $s1, $s1, -4
5. bne $s1, $zero, Loop
6. nop

• How many pipeline stages (clock cycles) per loop iteration to execute this code?

1 2 3 4 5 6 7 8 9 10

Peer Instruction Answer (1/2)

• Assume 1 instr/clock, delayed branch, 5 stage pipeline, forwarding, interlock on unresolved load hazards. $10^3$ iterations, so pipeline full.

Loop:

1. lw $t0, 0($s1)
2. (data hazard so stall)
3. addu $t0, $t0, $s2
4. sw $t0, 0($s1)
5. addiu $s1, $s1, -4
6. (!= in DCD)
7. bne $s1, $zero, Loop
8. nop (delayed branch so exec. nop)

• How many pipeline stages (clock cycles) per loop iteration to execute this code?

1 2 3 4 5 6 7 8 9 10
Assume 1 instr/clock, delayed branch, 5 stage pipeline, forwarding, interlock on unresolved load hazards (after $10^3$ loops, so pipeline full).

Rewrite this code to reduce pipeline stages (clock cycles) per loop to as few as possible.

Loop:
1. lw $t0, 0($s1)
2. addu $t0, $t0, $s2
3. sw $t0, 0($s1)
4. addiu $s1, $s1, -4
5. bne $s1, $zero, Loop

How many pipeline stages (clock cycles) per loop iteration to execute this code?

1 2 3 4 5 6 7 8 9 10

Rewrite this code to reduce clock cycles per loop to as few as possible:

Loop:
1. lw $t0, 0($s1) (no hazard since extra cycle)
2. addiu $s1, $s1, -4
3. addu $t0, $t0, $s2
4. bne $s1, $zero, Loop
5. sw $t0, +4($s1) (modified sw to put past addiu)

How many pipeline stages (clock cycles) per loop iteration to execute your revised code? (assume pipeline is full)

1 2 3 4 5 6 7 8 9 10
Administrivia

• HW5 due TODAY 7/29
• Quiz9 due Wednesday 7/30
• HW6 due Friday 8/1
• Proj3 out soon, due next Tuesday 8/5
  • Will be hand graded in person, signups will be posted soon
• Midterm regrades due TODAY 7/29
• Proj1 grades out, proj2 hopefully soon
  • appeals due 7/31

Administrivia

• Lab on polling/interrupts is cancelled
  • We will give everyone 4 pts on that lab

• Drop or grading option deadline
  • August 1
  • summer.berkeley.edu for more details
The Big Picture

Memory Hierarchy

Storage in computer systems:

- **Processor**
  - holds data in register file (~100 Bytes)
  - Registers accessed on nanosecond timescale

- **Memory (we’ll call “main memory”)**
  - More capacity than registers (~Gbytes)
  - Access time ~50-100 ns
  - Hundreds of clock cycles per memory access?!

- **Disk**
  - HUGE capacity (virtually limitless)
  - VERY slow: runs ~milliseconds
Motivation: Why We Use Caches (written $)

- 1989 first Intel CPU with cache on chip
- 1998 Pentium III has two levels of cache on chip

Memory Caching

- Mismatch between processor and memory speeds leads us to add a new level: a memory cache
- Implemented with same IC processing technology as the CPU (usually integrated on same chip): faster but more expensive than DRAM memory.
- Cache is a copy of a subset of main memory.
- Most processors have separate caches for instructions and data.
Memory Hierarchy

- As we move to deeper levels the latency goes up and price per bit goes down.

If level closer to Processor, it is:
- smaller
- faster
- subset of lower levels (contains most recently used data)

Lowest Level (usually disk) contains all available data (or does it go beyond the disk?)

Memory Hierarchy presents the processor with the illusion of a very large very fast memory.
Memory Hierarchy Analogy: Library (1/2)

• You’re writing a term paper (Processor) at a table in Doe

• **Doe Library** is equivalent to **disk**
  - essentially limitless capacity
  - very slow to retrieve a book

• **Table** is **main memory**
  - smaller capacity: means you must return book when table fills up
  - easier and faster to find a book there once you’ve already retrieved it

Memory Hierarchy Analogy: Library (2/2)

• **Open books on table** are **cache**
  - smaller capacity: can have very few open books fit on table; again, when table fills up, you must close a book
  - much, much faster to retrieve data

• Illusion created: whole library open on the tabletop
  - Keep as many recently used books open on table as possible since likely to use again
  - Also keep as many books on table as possible, since faster than going to library
Memory Hierarchy Basis

• Cache contains copies of data in memory that are being used.

• Memory contains copies of data on disk that are being used.

• Caches work on the principles of temporal and spatial locality.
  • Temporal Locality: if we use it now, chances are we’ll want to use it again soon.
  • Spatial Locality: if we use a piece of memory, chances are we’ll use the neighboring pieces soon.

Cache Design

• How do we organize cache?

• Where does each memory address map to?
  (Remember that cache is subset of memory, so multiple memory addresses map to the same cache location.)

• How do we know which elements are in cache?

• How do we quickly locate them?
**Direct-Mapped Cache (1/4)**

- In a **direct-mapped cache**, each memory address is associated with one possible **block** within the cache
  - Therefore, we only need to look in a single location in the cache for the data if it exists in the cache
  - **Block** is the unit of transfer between cache and memory

---

**Direct-Mapped Cache (2/4)**

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache Index</th>
<th>4 Byte Direct Mapped Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**Cache Location 0 can be occupied by data from:**
- Memory location 0, 4, 8, ...
- 4 blocks ⇒ any memory location that is multiple of 4

**What if we wanted a block to be bigger than one byte?**
### Direct-Mapped Cache (3/4)

- When we ask for a byte, the system finds out the right block, and loads it all!
  - How does it know right block?
  - How do we select the byte?

- E.g., Mem address 11101?
- How does it know WHICH colored block it originated from?
  - What do you do at baggage claim?

### Direct-Mapped Cache (4/4)

- What should go in the tag?
  - Do we need the entire address?
    - What do all these tags have in common?
  - What did we do with the immediate when we were branch addressing, always count by bytes?

- Why not count by cache #?
  - It's useful to draw memory with the same width as the block size.
Issues with Direct-Mapped

• Since multiple memory addresses map to same cache index, how do we tell which one is in there?

• What if we have a block size > 1 byte?

• Answer: divide memory address into three fields

$$\text{tttttttttttttttt} \ | \ \text{i} \ | \ \text{oo} \ | \ \text{oo}$$

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
<th>byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>to check</td>
<td>to</td>
<td>offset</td>
</tr>
<tr>
<td>if have</td>
<td>select</td>
<td>within</td>
</tr>
<tr>
<td>correct block</td>
<td>block</td>
<td>block</td>
</tr>
</tbody>
</table>

Direct-Mapped Cache Terminology

• All fields are read as unsigned integers.

• **Index**: specifies the cache index (which “row”/block of the cache we should look in)

• **Offset**: once we’ve found correct block, specifies which byte within the block we want

• **Tag**: the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location
**TIO Dan’s great cache mnemonic**

\[
\text{AREA (cache size, B)} = \text{HEIGHT (\# of blocks)} \times \text{WIDTH (size of one block, B/block)}
\]

\[2^{(H+W)} = 2^H \times 2^W\]

- **Tag**
- **Index**
- **Offset**

---

**Direct-Mapped Cache Example (1/3)**

- Suppose we have a 16KB of data in a direct-mapped cache with 4 word blocks
- Determine the size of the tag, index and offset fields if we’re using a 32-bit architecture
- **Offset**
  - need to specify correct byte within a block
  - block contains 4 words
    \[= 16 \text{ bytes}\]
    \[= 2^4 \text{ bytes}\]
  - need 4 bits to specify correct byte
**Direct-Mapped Cache Example (2/3)**

- **Index:** (~index into an “array of blocks”)
  - need to specify correct block in cache
  - cache contains 16 KB = $2^{14}$ bytes
  - block contains $2^4$ bytes (4 words)
  - \# blocks/cache
    \[
    \text{bytes/cache} = \frac{\text{bytes}}{\text{block}} = \frac{2^{14}}{2^4} = 2^{10} \text{ blocks/cache}
    \]
  - need **10 bits** to specify this many blocks

**Direct-Mapped Cache Example (3/3)**

- **Tag:** use remaining bits as tag
  - tag length = addr length – offset - index
    \[
    = 32 - 4 - 10 \text{ bits}
    = 18 \text{ bits}
    \]
  - so tag is leftmost **18 bits** of memory address

- **Why not full 32 bit address as tag?**
  - All bytes within block need same address (4b)
  - Index must be same for every address within a block, so it’s redundant in tag check, thus can leave off to save memory (here 10 bits)
Caching Terminology

• When we try to read memory, 3 things can happen:

1. **cache hit**: cache block is valid and contains proper address, so read desired word

2. **cache miss**: nothing in cache in appropriate block, so fetch from memory

3. **cache miss, block replacement**: wrong data is in cache at appropriate block, so discard it and fetch desired data from memory (cache always copy)

Peer instruction

• Consider an address split into fields for cache access as follows:

```

```

• How big are the cache blocks in words?
  4  8  16  32

• How many entries does the cache have?
  4  8  16  32

• How big is a cache entry?
  24  33  55  56
In Conclusion

• Pipeline challenge is hazards
  • Forwarding helps w/many data hazards
  • Delayed branch helps with control hazard in 5 stage pipeline
  • Load delay slot / interlock necessary

• More aggressive performance:
  • Superscalar
  • Out-of-order execution

• Use caches to simulate fast large memory