Lecture 5: Memory Hierarchy

Some slides adapted from Mary Jane Irwin at Penn State University for Computer Organization and Design, Patterson & Hennessy, © 2005
Watch this

Click the chip
Review: Instruction Type vs # of Required Cycles

- **IFetch**: Instruction Fetch and Update PC
- **Dec**: Instruction Decode, Register Read, Sign Extend Offset
- **Exec**: Execute R-type; Calculate Memory Address; Branch Comparison; Branch and Jump Completion
- **Mem**: Memory Read; Memory Write Completion; R-type Completion (RegFile write)
- **WB**: Memory Read Completion (RegFile write)

*INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!*
Review: Single Cycle Pros & Cons

- uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the **slowest** instruction
  - especially problematic for more complex instructions like floating point multiply

![Clock Cycle Diagram]

- may be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle
  - but
- is simple and easy to understand
Review: Multicycle Pros & Cons

- uses the clock cycle efficiently – the clock cycle is timed to accommodate the slowest instruction step

- multicycle implementations allow functional units to be used more than once per instruction as long as they are used on different clock cycles

but

- requires additional internal state registers, more muxes, and more complicated (FSM) control
Review: Single Cycle vs. Multiple Cycle Timing

single cycle implementation:

cycle 1

lw

cycle 2

sw

waste

multicycle clock slower than 1/5\textsuperscript{th} of single cycle clock due to state register overhead

multiple cycle implementation:

clk cycle 1 cycle 2 cycle 3 cycle 4 cycle 5 cycle 6 cycle 7 cycle 8 cycle 9 cycle 10

lw IFetch Dec Exec Mem WB

sw IFetch Dec Exec Mem

R-type IFetch
Review: Will multicycle design be faster?

- let's assume $t_{\text{setup}} + t_{\text{cq}}$ time for registers = 0.1 ns
  - single cycle design:
    - clock cycle time = $4.7 + 0.1 = 4.8$ ns
    - time/inst = 1 cycle/inst $\times$ 4.8 ns/cycle = 4.8 ns/inst
  - multicycle design:
    - clock cycle time = $1.0 + 0.1 = 1.1$
    - time/inst = CPI $\times$ 1.1 ns/cycle (depends on the types or mixture of instructions!)

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>Fetch</th>
<th>Decode, R-Read</th>
<th>ALU</th>
<th>PC update</th>
<th>D Memory</th>
<th>R-Write</th>
<th>Total (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td></td>
<td></td>
<td></td>
<td>.8</td>
<td>3.7</td>
</tr>
<tr>
<td>Load</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td></td>
<td></td>
<td>1</td>
<td>.8</td>
<td>4.7</td>
</tr>
<tr>
<td>Store</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td></td>
<td></td>
<td>1</td>
<td>-</td>
<td>3.9</td>
</tr>
<tr>
<td>beq</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>.1</td>
<td></td>
<td></td>
<td>-</td>
<td>3.0</td>
</tr>
</tbody>
</table>
so far, we’ve viewed memory as a black box that you can put data and programs into for later access
Types of Memories

- SSD: $0.75 /GB, HD: $0.1-0.2/GB
- DRAM: $20-25/GB
- more on bandwidth later

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Latency</th>
<th>Cost/bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>&lt; 1KB</td>
<td>&lt; 1ns</td>
<td>$$$$$</td>
</tr>
<tr>
<td>On-chip SRAM</td>
<td>8KB-6MB</td>
<td>&lt; 2ns</td>
<td>$$$</td>
</tr>
<tr>
<td>Off-chip SRAM</td>
<td>1Mb – 16Mb</td>
<td>&lt; 10ns</td>
<td>$$</td>
</tr>
<tr>
<td>DRAM</td>
<td>64MB – 1TB</td>
<td>&lt; 100ns</td>
<td>$</td>
</tr>
<tr>
<td>Disk (SSD, HD)</td>
<td>40GB – 1PB</td>
<td>&lt; 20ms</td>
<td>&lt; $1/GB</td>
</tr>
</tbody>
</table>
Memory Hierarchy

Registers
On-Chip SRAM
Off-Chip SRAM
DRAM
Disk

CAPACITY vs. SPEED and COST
Why Does a Hierarchy Work?

- locality of reference
  - ✓ temporal locality
    - reference same memory location many times (close together, in time)
  - ✓ spatial locality
    - reference near neighbors around the same time
Example of Locality

```c
int A[100], B[100], C[100], D;
for (i=0; i<100; i++) {
}
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>D</th>
<th>C[99]</th>
<th>C[98]</th>
<th>C[97]</th>
<th>C[96]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a cache line (one fetch)
Memory Hierarchy

- Temporal locality
  - Keep recently referenced items at higher levels
  - Future references satisfied quickly

- Spatial locality
  - Bring neighbors of recently referenced to higher levels
  - Future references satisfied quickly

Diagram:

- CPU
- I & D L1 cache
- Shared L2 cache
- Main memory
- Disk
Example: Intel Nehalem Memory Hierarchy

1. 4-way set associative instruction cache
2. 8-way set associative L1 data cache (32 KB)
3. 8-way set associative L2 data cache (256 KB)
4. 16-way shared L3 cache (8 MB)
5. 3 DDR3 memory connections
Typical Memory Organization

- row decoder
- memory cell core Array
- sense amps
- column latches
- mux
- pin interface
- off-chip data

row addr

column addr
Basic Cache Operation

Diagram showing basic cache operation with arrows indicating data flow from processor to cache and then to lower level memory.
Cache Terminology

- **hit**: data appears in some block
  - **hit rate**: the fraction of accesses found in the level
  - **hit time**: time to access the level (consists of RAM access time + time to determine hit)

- **miss**: data needs to be retrieved from a block in the lower level (e.g., block Y)
  - **miss rate**: $1 - \text{hit rate}$
  - **miss penalty**: time to replace a block in the upper level + time to deliver the block to the processor

- hit time $\ll$ miss penalty
Average Memory Access Time

- average memory-access time
  = hit time + miss rate x miss penalty

- miss penalty: time to fetch a block from lower memory level
  - access time: function of latency
  - transfer time: function of bandwidth b/w levels
    - transfer one “cache line/block” at a time
    - transfer at the size of the memory-bus width
Memory Hierarchy Performance

- **Average Memory Access Time (AMAT)**
  - $AMAT = \text{hit time} + \text{miss rate} \times \text{miss penalty}$
  - $AMAT = T_{hit}(L1) + \text{miss\%}(L1) \times T(\text{memory})$

- **example:**
  - cache hit = 1 cycle
  - miss rate = 10\% = 0.1
  - miss penalty = 300 cycles
  - $AMAT = 1 + 0.1 \times 300 = 31 \text{ cycles}$

- **can we improve it?**
Reducing Penalty: Multi-Level Cache

- Average Memory Access Time (AMAT)

\[
\text{AMAT} = T_{\text{hit}}(L1) + \text{miss}\%(L1) \times (T_{\text{hit}}(L2) + \text{miss}\%(L2) \times (T_{\text{hit}}(L3) + \text{miss}\%(L3) \times T(\text{memory}))) \\
= T_{\text{hit}}(L1) + \text{miss}\%(L1) \times T_{\text{miss}}(L1) \\
= T_{\text{hit}}(L1) + \text{miss}\%(L1) \times \{ T_{\text{hit}}(L2) + \text{miss}\%(L2) \times T_{\text{miss}}(L2) \} \\
= T_{\text{hit}}(L1) + \text{miss}\%(L1) \times \{ T_{\text{hit}}(L2) + \text{miss}\%(L2) \times [ T_{\text{hit}}(L3) + \text{miss}\%(L3) \times T(\text{memory}) ] \}
\]
AMAT Example

\[ T_{hit}(L1) + \text{miss}\%(L1) \times (T_{hit}(L2) + \text{miss}\%(L2) \times (T_{hit}(L3) + \text{miss}\%(L3) \times T(\text{memory}) ) ) \]

- **Example:**
  - miss rate L1 = 10%, \( T_{hit}(L1) = 1 \) cycle
  - miss rate L2 = 5%, \( T_{hit}(L2) = 10 \) cycles
  - miss rate L3 = 1%, \( T_{hit}(L3) = 20 \) cycles
  - \( T(\text{memory}) = 300 \) cycles

- **AMAT = ?**
  - 2.115 (compare to 31 with no multi-levels)
  - 14.7× speed-up!
# Types of Caches

<table>
<thead>
<tr>
<th>type of cache</th>
<th>mapping of data from memory to cache</th>
<th>complexity of searching the cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>direct mapped (DM)</td>
<td>a memory value can be placed at a <strong>single corresponding</strong> location in the cache</td>
<td>fast indexing mechanism</td>
</tr>
<tr>
<td>set-associative (SA)</td>
<td>a memory value can be placed in any location in the cache</td>
<td>extensive hardware resources required to search (CAM)</td>
</tr>
<tr>
<td>fully-associative (FA)</td>
<td>a memory value can be placed at a single corresponding location in the cache</td>
<td>extensive hardware resources required to search (CAM)</td>
</tr>
</tbody>
</table>

DM and FA can be thought as special cases of SA (e.g., DM → 1-way SA, FA → All-way SA)
Direct Mapping

- direct mapping:
  - a memory value can only be placed at a single corresponding location in the cache
Set Associative Mapping (2-Way)

- set-associative mapping:
  - a memory value can be placed in any location of a set in the cache

![Diagram of set-associative mapping]

- Way 0:
  - Tag: 0000 0
  - Index: 0
  - Data: 0x55

- Way 0:
  - Tag: 0000 1
  - Index: 1
  - Data: 0x0F

- Way 1:
  - Tag: 1111 0
  - Index: 0
  - Data: 0xAA

- Way 1:
  - Tag: 1111 1
  - Index: 1
  - Data: 0xF0
Fully Associative Mapping

- fully-associative mapping:
  - a memory value can be placed anywhere in the cache
Direct Mapped Cache

- location 0 is occupied by data from (0, 4, 8, and C)
  - which one should we place in the cache?
  - how can we tell which one is in the cache?

```plaintext
memory
dm cache
```
Three Cs (Cache Miss Terms)

- compulsory misses:
  - cold start misses (caches do not have valid data at the start of the program)
Three Cs (Cache Miss Terms)

- capacity misses:
  - increase cache size
Three Cs (Cache Miss Terms)

- conflict misses:
  - increase cache size and/or associativity.
  - associative caches reduce conflict misses
Four Central Questions in Designing a Cache

- P-I-R-W:
  - placement: where can a block of memory go?
  - identification: how do I find a block of memory?
  - replacement: how do I make space for new blocks?
  - write policy: how do I propagate changes?

- Need to consider these for all levels of the memory hierarchy
  - L1/L2/L3 caches now

- Main memory, disks have similar issues, addressed later
Describing Caches: 7 Parameters

- access time: $T_{hit}$
- capacity
  - total amount of data the cache can hold
    - # of blocks $\times$ block size
- block (line) size
  - the amount of data that gets moved into or out of the cache as a chunk
    - analogous to page size in virtual memory
- replacement policy
  - what data is replaced on a miss?
- associativity
  - how many locations in the cache is a given address eligible to be placed in?
- unified, instruction, data
  - what type of data is kept in the cache? We’ll cover this in more detail later
Example: 1KB DM Cache, 32-byte Lines

- lowest M bits are offset (Line Size = 2M)
- index = log2 (# of sets)
Example of Caches

- given a 2MB, direct-mapped physical caches, line size=64bytes, and 52-bit physical address
  - tag size?

- now change it to 16-way, tag size?

- how about if it’s fully associative, tag size?
Announcement

- today’s lecture: cache basics
  ✔ Ch. 5.1 – 5.4 (HP1)

- next lecture: other cache topics (e.g., replacement policy)
  ✔ Ch. 5.4 – 5.8 (HP1)

- MP assignment
  ✔ MP1 due on 9/17 5pm