Final Review

Hung-Wei Tseng
By loading different programs into memory, your computer can perform different functions.
How does a processor execute instructions?

- **Instruction Fetch (IF)**
  - Fetch the *instruction* pointed by PC from *memory*
- **Instruction Decode (ID)**
  - Decode the instruction for the desired operation and operands
  - Reading source *register* values
- **Execution (EX)**
  - ALU instructions: Perform ALU operations
  - Conditional Branch: Determine the branch outcome (taken/not taken)
  - Memory instructions: Determine the effective address for data memory access
- **Data Memory Access (MEM)** — Read/write *data memory*
- **Write Back (WB)** — Present ALU result/read value in the target *register*
- **Update PC**
  - If the branch is taken — set to the branch target address
  - Otherwise — advance to the next instruction — current PC + 4

---

**Program**

- 120007a30: `0f00bb27 ldah gp,15(t12)
- 120007a34: `509cbd23 lda   gp,-25520(gp)
- 120007a38: `00005d24 ldah  t1,0(gp)
- 120007a3c: `0000bd24 ldah  t4,0(gp)
- 120007a40: `2ca422a0 ldl   t0,-23508(t1)
- 120007a44: `130020e4 beq   t0,120007a94
- 120007a48: `00003d24 ldah  t0,0(gp)
- 120007a4c: `2ca4e2b3 stl   zero,-23508(t1)

---

**ALU**

- Processor
- Registers
- Instructions

---

**Cycles / Instruction ** $\times$ **Seconds / Cycle**

**How long is it take to execution each of these?**
CPU Performance Equation

\[
\text{Performance} = \frac{1}{\text{Execution Time}}
\]

\[
\text{Execution Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}
\]

\[
ET = IC \times CPI \times CT
\]

\[
1GHz = 10^9 Hz = \frac{1}{10^9} \text{ sec per cycle} = 1 \text{ ns per cycle}
\]

Frequency(\(i.e.,\) clock rate)
Amdahl’s Law

\[ \text{Speedup}_{\text{enhanced}}(f, s) = \frac{1}{(1-f) + \frac{f}{s}} \]

- \( f \) — The fraction of time in the original program
- \( s \) — The speedup we can achieve on \( f \)

Execution Time_{baseline} = 1

Execution Time_{enhanced} = (1-f) + \( \frac{f}{s} \)
Pipelined processor with Data Forwarding
<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Usage</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td></td>
<td>addi</td>
<td>addi $s1,$s2, 20</td>
<td>$s1 = $s2 + 20</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>Logical</td>
<td>and</td>
<td>and $s1, $s2, $s3</td>
<td>$s1 = $s2 &amp; $s3</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>or $s1, $s2, $s3</td>
<td>$s1 = $s2</td>
</tr>
<tr>
<td></td>
<td>andi</td>
<td>andi $s1, $s2, 20</td>
<td>$s1 = $s2 &amp; 20</td>
</tr>
<tr>
<td></td>
<td>sll</td>
<td>sll $s1, $s2, 10</td>
<td>$s1 = $s2 * 2^10</td>
</tr>
<tr>
<td></td>
<td>srl</td>
<td>srl $s1, $s2, 10</td>
<td>$s1 = $s2 / 2^10</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>lw</td>
<td>lw $s1, 4($s2)</td>
<td>$s1 = mem[$s2+4]</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>sw $s1, 4($s2)</td>
<td>mem[$s2+4] = $s1</td>
</tr>
<tr>
<td>Branch</td>
<td>beq</td>
<td>beq $s1, $s2, 25</td>
<td>if($s1 == $s2), PC = PC + 100</td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td>bne $s1, $s2, 25</td>
<td>if($s1 != $s2), PC = PC + 100</td>
</tr>
<tr>
<td>Jump</td>
<td>jal</td>
<td>jal 25</td>
<td>$ra = PC + 4, PC = 100</td>
</tr>
<tr>
<td></td>
<td>jr</td>
<td>jr $ra</td>
<td>PC = $ra</td>
</tr>
</tbody>
</table>
A basic dynamic branch predictor

Branch Target Buffer

<table>
<thead>
<tr>
<th>branch PC</th>
<th>target PC</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400048</td>
<td>0x400032</td>
<td>10</td>
</tr>
<tr>
<td>0x400080</td>
<td>0x400068</td>
<td>11</td>
</tr>
<tr>
<td>0x401080</td>
<td>0x401100</td>
<td>00</td>
</tr>
<tr>
<td>0x4000F8</td>
<td>0x400100</td>
<td>01</td>
</tr>
</tbody>
</table>
Global history (GH) predictor

Branch Target Buffer

<table>
<thead>
<tr>
<th>branch PC</th>
<th>target PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400048</td>
<td>0x400032</td>
</tr>
<tr>
<td>0x400080</td>
<td>0x400068</td>
</tr>
<tr>
<td>0x401080</td>
<td>0x401100</td>
</tr>
<tr>
<td>0x4000F8</td>
<td>0x400100</td>
</tr>
</tbody>
</table>

Predict Taken = (NT, T, NT, NT)
Which of the following implementations will perform the best on modern pipeline processors?

A

```c
inline int popcount(uint64_t x) {
    int c = 0;
    while(x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

B

```c
inline int popcount(uint64_t x) {
    int c = 0;
    while(x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

C

```c
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1, 2, 3, 1, 2, 2, 3, 2, 3, 3, 3, 4};
    while(x) {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```

D

```c
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    for (uint64_t i = 0; i < 16; i++) {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```
Recap: Performance gap between Processor/Memory
Recap: Four implementations

- Which of the following implementations will perform the best on modern pipeline processors?

```
inline int popcount(uint64_t x){
    int c = 0;
    while(x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

```
inline int popcount(uint64_t x) {
    int c = 0;
    while(x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

```
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1, 2, 3, 2, 3, 3, 4};
    while(x) {
        c += table[x & 0xF];
        x = x >> 4;
    }
    return c;
}
```

```
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    for (uint64_t i = 0; i < 16; i++)
        c += table[(x & 0xF)];
    x = x >> 4;
    return c;
}
```

Not going to work out if memory is that slow
Make Memory Great Again
Memory Hierarchy

Processor

- Core
  - Registers

SRAM

DRAM

Storage

L1$

L2$

L3$

Fastest

< 1ns

A few ns

Tens of ns

Tens of us

TBs
The impact of “slow” memory

• Assume that we have a processor running @ 2 GHz and a program with 30% of load/store instructions. If the computer has “perfect” memory, the CPI is just 1. Now, consider we have DDR4 and the program is well-behaved that precharge is never necessary — the access latency is simply 26 ns. What’s the average CPI (pick the most close one)?

A. 9
B. 17
C. 27
D. 35
E. 69

\[1 + 100\% \times (52) + 30\% \times 52 = 68.6 \text{ cycles}\]
How can deeper memory hierarchy help in performance?

- Assume that we have a processor running @ 2 GHz and a program with 30% of load/store instructions. If the computer has “perfect” memory, the CPI is just 1. Now, in addition to DDR4, whose latency 26 ns, we also got a 2-level SRAM caches with
  - it’s 1st-level one at latency of 0.5ns and can capture 90% of the desired data/instructions.
  - the 2nd-level at latency of 5ns and can capture 60% of the desired data/instructions

What’s the average CPI (pick the most close one)?

A. 2
B. 4
C. 8
D. 16
E. 32

\[ 1 + (1 - 90\%) \times [10 + (1 - 60\%) \times 52 + 30\% \times (10 + (1 - 60\%) \times 52)] = 5 \text{ cycle}. \]
Locality — why cache works

• Spatial locality — application tends to visit nearby stuffs in the memory
  • Code — the current instruction, and then PC + 4

Most of time, your program is just visiting a very small amount of data/instructions within a given window

• Temporal locality — application revisit the same thing again and again
  • Code — loops, frequently invoked functions

• Data — the same data can be read/write many times
What happens when we read data

- Processor sends load request to L1-$
  - if hit
    - return data
  - if miss
    - Select a victim block
      - If the target “set” is not full — select an empty/invalidated block as the victim block
      - If the target “set is full — select a victim block using some policy
        - LRU is preferred — to exploit temporal locality!
        - If the victim block is “dirty” & “valid”
          - Write back the block to lower-level memory hierarchy
          - Fetch the requesting block from lower-level memory hierarchy and place in the victim block
          - If write-back or fetching causes any miss, repeat the same process
What happens when we write data

- Processor sends load request to L1-$
  - if hit
    - return data — set DIRTY
  - if miss
    - Select a victim block
      - If the target “set” is not full — select an empty/invalidated block as the victim block
      - If the target “set” is full — select a victim block using some policy
        - LRU is preferred — to exploit temporal locality!
    - If the victim block is “dirty” & “valid”
      - Write back the block to lower-level memory hierarchy
    - Fetch the requesting block from lower-level memory hierarchy and place in the victim block
    - If write-back or fetching causes any miss, repeat the same process
  - Present the write “ONLY” in L1 and set DIRTY
How to evaluate cache performance

• CPI_{Average} : the average CPI of a memory instruction

  \[ CPI_{\text{average}} = CPI_{\text{base}} + miss\_rate_{L1} \times miss\_penalty_{L1} \]

  \[ miss\_penalty_{L1} = CPI_{\text{accessing}_{L2}} + miss\_rate_{L2} \times miss\_penalty_{L2} \]

  \[ miss\_penalty_{L2} = CPI_{\text{accessing}_{L3}} + miss\_rate_{L3} \times miss\_penalty_{L3} \]

  \[ miss\_penalty_{L3} = CPI_{\text{accessing}_{\text{DRAM}}} + miss\_rate_{\text{DRAM}} \times miss\_penalty_{\text{DRAM}} \]

• If the problem is asking for average memory access time, transform the CPI values into/from time by multiplying with CPU cycle time!
Cache & Performance

- Application: 80% ALU, 20% Load/Store
- L1 I-cache miss rate: 5%, hit time: 1 cycle
- L1 D-cache miss rate: 10%, hit time: 1 cycle, 20% dirty
- L2 U-Cache miss rate: 20%, hit time: 10 cycles, 10% dirty
- Main memory hit time: 100 cycles

What's the average CPI?

CPI_{average} = CPI_{base} + \text{miss rate} \times \text{miss penalty}

= 1 + 100\% \times (5\% \times (10 + 20\% \times ((1 + 10\%) \times 100)))

+ 20\% \times (10\% \times (1 + 20\%) \times (10 + 20\% \times ((1 + 10\%) \times 100)))

= 3.368
\( C = \text{ABS} \)

- **C**: Capacity in data arrays
- **A**: Way-Associativity — how many blocks within a set
  - N-way: N blocks in a set, \( A = N \)
  - 1 for direct-mapped cache
- **B**: Block Size (Cacheline)
  - How many bytes in a block
- **S**: Number of Sets:
  - A set contains blocks sharing the same index
  - 1 for fully associate cache
L1 data (D-L1) cache configuration of AMD Phenom II
- Size 64KB, 2-way set associativity, 64B block
- Assume 64-bit memory address

Which of the following is correct?

A. Tag is 49 bits
B. Index is 8 bits
C. Offset is 7 bits
D. The cache has 1024 sets
E. None of the above

C = ABS
64KB = 2 * 64 * S
S = 512
offset = lg(64) = 6 bits
index = lg(512) = 9 bits
tag = 64 - lg(512) - lg(64) = 49 bits
intel Core i7

• L1 data (D-L1) cache configuration of Core i7
  • Size 32KB, 8-way set associativity, 64B block
  • Assume 64-bit memory address
  • Which of the following is NOT correct?
    A. Tag is 52 bits
    B. Index is 6 bits
    C. Offset is 6 bits
    D. The cache has 128 sets

\[ C = \text{ABS} \]
\[ 32\text{KB} = 8 \times 64 \times S \]
\[ S = 64 \]
\[ \text{offset} = \lg(64) = 6 \text{ bits} \]
\[ \text{index} = \lg(64) = 6 \text{ bits} \]
\[ \text{tag} = 64 - \lg(64) - \lg(64) = 52 \text{ bits} \]
Simulate a 2-way cache

- Consider a 2-way cache with 256 bytes total capacity, a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000, 0b1000010100, 0b1100010000

  - C = A B S
  - S = 256 / (16 * 2) = 8
  - 8 = 2^3 : 3 bits are used for the index
  - 16 = 2^4 : 4 bits are used for the byte offset
  - The tag is 32 - (3 + 4) = 25 bits
  - For example: 0b1000 0000 0000 0000 0000 0000 0000 0001 0000
Simulate a 2-way cache

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0b100</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0b100</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0b100</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0b110</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0b110</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10 0000 0000</td>
<td>miss</td>
</tr>
<tr>
<td>0b10 0000 1000</td>
<td>hit!</td>
</tr>
<tr>
<td>0b10 0001 0000</td>
<td>miss</td>
</tr>
<tr>
<td>0b10 0001 0100</td>
<td>hit!</td>
</tr>
<tr>
<td>0b11 0001 0000</td>
<td>miss</td>
</tr>
<tr>
<td>0b10 0000 0000</td>
<td>hit!</td>
</tr>
<tr>
<td>0b10 0000 1000</td>
<td>hit!</td>
</tr>
<tr>
<td>0b10 0001 0000</td>
<td>hit</td>
</tr>
<tr>
<td>0b10 0001 0100</td>
<td>hit!</td>
</tr>
</tbody>
</table>
• D-L1 Cache configuration of AMD Phenom II
  • Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.

```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

What’s the data cache miss rate for this code?

A. 6.25%
B. 56.25%
C. 66.67%
D. 68.75%
E. 100%
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/

100% miss rate!

- Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 48-bit address.

<table>
<thead>
<tr>
<th>address in hex</th>
<th>tag</th>
<th>index</th>
<th>offset</th>
<th>tag</th>
<th>index</th>
<th>hit? miss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>load a[0]</td>
<td>0x20000</td>
<td>0b10 0000 0000 0000 0000</td>
<td>0x4</td>
<td>0</td>
<td>miss</td>
<td></td>
</tr>
<tr>
<td>load b[0]</td>
<td>0x30000</td>
<td>0b11 0000 0000 0000 0000</td>
<td>0x6</td>
<td>0</td>
<td>miss</td>
<td></td>
</tr>
<tr>
<td>store c[0]</td>
<td>0x10000</td>
<td>0b01 0000 0000 0000 0000</td>
<td>0x2</td>
<td>0</td>
<td>miss, evict 0x4</td>
<td></td>
</tr>
<tr>
<td>load a[1]</td>
<td>0x20004</td>
<td>0b10 0000 0000 0000 0100</td>
<td>0x4</td>
<td>0</td>
<td>miss, evict 0x6</td>
<td></td>
</tr>
<tr>
<td>load b[1]</td>
<td>0x30004</td>
<td>0b11 0000 0000 0000 0100</td>
<td>0x6</td>
<td>0</td>
<td>miss, evict 0x2</td>
<td></td>
</tr>
<tr>
<td>store c[1]</td>
<td>0x10004</td>
<td>0b01 0000 0000 0000 0100</td>
<td>0x2</td>
<td>0</td>
<td>miss, evict 0x4</td>
<td></td>
</tr>
</tbody>
</table>

C = ABS
64KB = 2 * 64 * S
S = 512
offset = lg(64) = 6 bits
index = lg(512) = 9 bits
tag = the rest bits

C = ABS
offset = lg(64) = 6 bits
index = lg(512) = 9 bits
tag = the rest bits
D-L1 Cache configuration of intel Core i7 processor
- Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

What’s the data cache miss rate for this code?
A. 6.25%
B. 56.25%
C. 66.67%
D. 68.75%
E. 100%
```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
{
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
}
```

<table>
<thead>
<tr>
<th>address</th>
<th>tag</th>
<th>index</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>load a[0]</td>
<td>0x20000</td>
<td>0x20</td>
<td>0</td>
</tr>
<tr>
<td>load b[0]</td>
<td>0x30000</td>
<td>0x30</td>
<td>0</td>
</tr>
<tr>
<td>store c[0]</td>
<td>0x10000</td>
<td>0x10</td>
<td>0</td>
</tr>
<tr>
<td>load a[1]</td>
<td>0x20004</td>
<td>0x20</td>
<td>0</td>
</tr>
<tr>
<td>load b[1]</td>
<td>0x30004</td>
<td>0x30</td>
<td>0</td>
</tr>
<tr>
<td>store c[1]</td>
<td>0x10004</td>
<td>0x10</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>load a[15]</td>
<td>0x2003C</td>
<td>0x20</td>
<td>0</td>
</tr>
<tr>
<td>load b[15]</td>
<td>0x3003C</td>
<td>0x30</td>
<td>0</td>
</tr>
<tr>
<td>store c[15]</td>
<td>0x1003C</td>
<td>0x10</td>
<td>0</td>
</tr>
<tr>
<td>load a[16]</td>
<td>0x20040</td>
<td>0x20</td>
<td>1</td>
</tr>
<tr>
<td>load b[16]</td>
<td>0x30040</td>
<td>0x30</td>
<td>1</td>
</tr>
<tr>
<td>store c[16]</td>
<td>0x1003C</td>
<td>0x10</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
32*3/(512*3) = 1/16 = 6.25\% (93.75\% hit rate!)
\]
Cause of cache misses
3Cs of misses

- Compulsory miss
  - Cold start miss. First-time access to a block
- Capacity miss
  - The working set size of an application is bigger than cache size
- Conflict miss
  - Required data replaced by block(s) mapping to the same set
  - Similar collision in hash
Simulate a direct-mapped cache

- Consider a direct mapped (1-way) cache with 256 bytes total capacity, a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000

- \( C = A B S \)
- \( S = 256/(16*1) = 16 \)
- \( \lg(16) = 4 \): 4 bits are used for the index
- \( \lg(16) = 4 \): 4 bits are used for the byte offset
- The tag is 48 - (4 + 4) = 40 bits
- For example: 0b1000 0000 0000 0000 0000 0000 0000 1000 0000
### Simulate a direct-mapped cache

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Tag and Index

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0000</td>
</tr>
<tr>
<td>0b10</td>
<td>0001</td>
</tr>
<tr>
<td>0b10</td>
<td>0001</td>
</tr>
<tr>
<td>0b11</td>
<td>0001</td>
</tr>
<tr>
<td>0b10</td>
<td>0001</td>
</tr>
<tr>
<td>0b10</td>
<td>0001</td>
</tr>
<tr>
<td>0b10</td>
<td>0001</td>
</tr>
</tbody>
</table>

- **compulsory miss**
- **hit!**
- **conflict miss**
Simulate a 2-way cache

- Consider a 2-way cache with 256 bytes total capacity, a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000, 0b1000010100, 0b1100010000

- \( C = A B S \)
- \( S = 256 / (16 \times 2) = 8 \)
- 8 = \(2^3\) : 3 bits are used for the index
- 16 = \(2^4\) : 4 bits are used for the byte offset
- The tag is 32 - (3 + 4) = 25 bits
- For example: 0b1000 0000 0000 0000 0000 0000 0001 0000
Simulate a 2-way cache

```
<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0b100</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0b100</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Compulsory miss

hit!

Compulsory miss

hit!

Compulsory miss

hit!

Hit!

Hit!

Hit!

Hit!
```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/

• Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

<table>
<thead>
<tr>
<th>address in hex</th>
<th>address in binary</th>
<th>hit? miss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>load a[0]</td>
<td>0x20000</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>load b[0]</td>
<td>0x30000</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>store c[0]</td>
<td>0x10000</td>
<td>compulsory miss, evict</td>
</tr>
<tr>
<td>load a[1]</td>
<td>0x20004</td>
<td>conflict miss, evict 0x6</td>
</tr>
<tr>
<td>load b[1]</td>
<td>0x30004</td>
<td>conflict miss, evict 0x2</td>
</tr>
<tr>
<td>store c[1]</td>
<td>0x10004</td>
<td>conflict miss, evict 0x4</td>
</tr>
<tr>
<td>load a[15]</td>
<td>0x2003C</td>
<td>miss, evict 0x6</td>
</tr>
<tr>
<td>load b[15]</td>
<td>0x3003C</td>
<td>miss, evict 0x2</td>
</tr>
<tr>
<td>store c[15]</td>
<td>0x1003C</td>
<td>miss, evict 0x4</td>
</tr>
<tr>
<td>load a[16]</td>
<td>0x20040</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>load b[16]</td>
<td>0x30040</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>store c[16]</td>
<td>0x10040</td>
<td>compulsory miss, evict</td>
</tr>
</tbody>
</table>

C = \text{ABS}

64KB = 2\times 64 \times S
S = 512

\text{offset} = \lg(64) = 6 \text{ bits}
\text{index} = \lg(512) = 9 \text{ bits}
tag = \text{the rest bits}

\text{hit} \Rightarrow \text{tag} = 0
\text{miss} \Rightarrow \text{tag} = 1
\text{evict} \Rightarrow \text{evicted address} = \text{index of miss}
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
{
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
}

<table>
<thead>
<tr>
<th></th>
<th>address</th>
<th>tag</th>
<th>index</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>load a[0]</td>
<td>0x20000</td>
<td>0x20</td>
<td>0</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>load b[0]</td>
<td>0x30000</td>
<td>0x30</td>
<td>0</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>store c[0]</td>
<td>0x10000</td>
<td>0x10</td>
<td>0</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>load a[1]</td>
<td>0x20004</td>
<td>0x20</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td>load b[1]</td>
<td>0x30004</td>
<td>0x30</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td>store c[1]</td>
<td>0x10004</td>
<td>0x10</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td>load a[15]</td>
<td>0x2003C</td>
<td>0x20</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td>load b[15]</td>
<td>0x3003C</td>
<td>0x30</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td>store c[15]</td>
<td>0x1003C</td>
<td>0x10</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td>load a[16]</td>
<td>0x20040</td>
<td>0x20</td>
<td>1</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>load b[16]</td>
<td>0x30040</td>
<td>0x30</td>
<td>1</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>store c[16]</td>
<td>0x1003C</td>
<td>0x10</td>
<td>1</td>
<td>compulsory miss</td>
</tr>
</tbody>
</table>

32*3/(512*3) = 1/16 = 6.25% (93.75% hit rate!)
Matrix transpose

double A[16384], B[16384];
int N=128;
for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
        B[i+N+j] = A[j*N+i];
// assume load A[j*N+i] and then store B[i*N+j]
// &A[0] is 0x20000, &B[0] is 0x40000

What’s the access sequence of A[] looks like?
A[0], A[128], A[256], ..., A[127*128], A[1], A[129]..., A[127*128+1], ...

What’s the access sequence of B[] looks like?
B[0], B[1], B[2], ......

If the cache is 64KB-sized, 2-way, 64B-blocked
Each block can hold B[0]-B[7] or B[8]-B[15] or B[16]-B[23], or and so on
Every “first” time you access an element in a block, it will incur a compulsory miss

Given this code will go through every elements, and compulsory misses occurs every 8 elements.
For array A, 128*128/8 = 2048 compulsory misses, and
for array B, 128*128/8 = 2048 compulsory misses
Matrix transpose (cont.)

double A[16384], B[16384];
int N=128;
for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
        B[i*N+j] = A[j*N+i];
// assume load A[j*N+i] and then store B[i*N+j]
// &A[0] is 0x20000, &B[0] is 0x40000

What’s the access sequence of A[] looks like?
A[0], A[128], A[256], ..., A[127*128], A[1], A[129], ..., A[127*128+1], ...
Since the cache is 64KB-sized, 2-way, 64B-blocked

<table>
<thead>
<tr>
<th>address hex</th>
<th>address in binary</th>
<th>tag</th>
<th>index</th>
<th>hit? miss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>load a[0]</td>
<td>0x20000</td>
<td>10 0000 0000 0000 0000</td>
<td>0x4</td>
<td>0</td>
</tr>
<tr>
<td>load a[128]</td>
<td>0x20400</td>
<td>10 0000 0100 0000 0000</td>
<td>0x4</td>
<td>0x10</td>
</tr>
<tr>
<td>load a[4096]</td>
<td>0x28000</td>
<td>10 1000 0000 0000 0000</td>
<td>0x5</td>
<td>0</td>
</tr>
<tr>
<td>load a[8192]</td>
<td>0x30000</td>
<td>11 0000 0000 0000 0000</td>
<td>0x6</td>
<td>0</td>
</tr>
<tr>
<td>load a[1]</td>
<td>0x20004</td>
<td>10 0000 0000 0000 0000 0100</td>
<td>0x4</td>
<td>0</td>
</tr>
</tbody>
</table>

Very unlikely in index 0 given we only have
2 blocks in set index 0

For array A, always a miss.
128*128/8 = 2048 compulsory misses, and
128*128-2048 conflict misses.
### Matrix transpose

<table>
<thead>
<tr>
<th></th>
<th>address in hex</th>
<th>address in binary</th>
<th>tag</th>
<th>index</th>
<th>hit? miss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>load a[0]</td>
<td>0x20000</td>
<td>10 0000 0000 0000 0000</td>
<td>0x4</td>
<td>0</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>store b[0]</td>
<td>0x40000</td>
<td>100 0000 0000 0000 0000</td>
<td>0x8</td>
<td>0</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>load a'[128]'</td>
<td>0x20400</td>
<td>10 0000 0100 0000 0000</td>
<td>0x4</td>
<td>0x10</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>store b'[1]'</td>
<td>0x40008</td>
<td>100 0000 0000 0000 1000</td>
<td>0x8</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>load a[1024]</td>
<td>0x22000</td>
<td>10 0010 0000 0000 0000</td>
<td>0x4</td>
<td>0x80</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>store b[8]</td>
<td>0x40400</td>
<td>100 0000 0000 0100 0000</td>
<td>0x8</td>
<td>0x1</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>load a[1152]</td>
<td>0x22400</td>
<td>10 0010 0100 0000 0000</td>
<td>0x4</td>
<td>0x90</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>store b[9]</td>
<td>0x4048</td>
<td>100 0000 0000 0100 1000</td>
<td>0x8</td>
<td>0x1</td>
<td>hit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>load a[4096]</td>
<td>0x28000</td>
<td>10 1000 0000 0000 0000</td>
<td>0x5</td>
<td>0</td>
<td>compulsory miss, evict something in index 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>load a'[128]'</td>
<td>0x20004</td>
<td>10 0000 0000 0000 0100</td>
<td>0x4</td>
<td>0</td>
<td>conflict miss, evict something in index 0</td>
</tr>
<tr>
<td>store b'[128]'</td>
<td>0x40400</td>
<td>100 0000 0100 0000 0000</td>
<td>0x8</td>
<td>0x10</td>
<td>compulsory miss, evict something in index 0x10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Improvement of 3Cs

• 3Cs and A, B, C of caches
  • Compulsory miss
    • Increase B: increase miss penalty (more data must be fetched from lower hierarchy)
  • Capacity miss
    • Increase C: increase cost, access time, power
  • Conflict miss
    • Increase A: increase access time and power

• Or modify the memory access pattern of your program!
### What data structure is performing better

<table>
<thead>
<tr>
<th></th>
<th>Array of objects</th>
<th>object of arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>struct grades</strong></td>
<td>{ int id; double *homework; double average; }</td>
<td><strong>struct grades</strong></td>
</tr>
<tr>
<td></td>
<td>};</td>
<td></td>
</tr>
<tr>
<td><strong>average of each homework</strong></td>
<td>for(i=0;i&lt;homework_items; i++) { gradesheet[total_number_students].homework[i] = 0.0; for(j=0;j&lt;total_number_students;j++) gradesheet[total_number_students].homework[i] +=gradesheet[j].homework[i]; gradesheet[total_number_students].homework[i] /= (double)total_number_students; }</td>
<td>for(i = 0;i &lt; homework_items; i++) { gradesheet.homework[i][total_number_students] = 0.0; for(j = 0; j &lt;total_number_students;j++) { gradesheet.homework[i][total_number_students] += gradesheet.homework[i][j]; } gradesheet.homework[i][total_number_students] /= total_number_students; }</td>
</tr>
</tbody>
</table>

• Considering your workload would like to calculate the average score of one of the homework for all students, which data structure would deliver better performance? **What if we want to calculate average scores for each student?**

  A. Array of objects  
  **B. Object of arrays**
Column-store or row-store

• If you’re designing an in-memory database system, will you be using

<table>
<thead>
<tr>
<th>RowId</th>
<th>EmpId</th>
<th>Lastname</th>
<th>Firstname</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>Smith</td>
<td>Joe</td>
<td>40000</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>Jones</td>
<td>Mary</td>
<td>50000</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>Johnson</td>
<td>Cathy</td>
<td>44000</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>Jones</td>
<td>Bob</td>
<td>55000</td>
</tr>
</tbody>
</table>

• column-store — stores data tables column by column

10:001,12:002,11:003,22:004;
Smith:001,Jones:002,Johnson:003,Jones:004;
Joe:001,Mary:002,Cathy:003,Bob:004;
40000:001,50000:002,44000:003,55000:004;

if the most frequently used query looks like —

select Lastname, Firstname from table

• row-store — stores data tables row by row

001:10,Smith,Joe,40000;
002:12,Jones,Mary,50000;
003:11,Johnson,Cathy,44000;
004:22,Jones,Bob,55000;
What if the code look like this?

- D-L1 Cache configuration of AMD Phenom II
  - Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.

```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    c[i] = a[i]; //load a and then store to c
for(i = 0; i < 512; i++)
    c[i] += b[i]; //load b, load c, add, and then store to c
```

What’s the data cache miss rate for this code?

A. 6.25%
B. 56.25%
C. 66.67%
D. 68.75%
E. 100%
Case study: Matrix Multiplication

Algorithm class tells you it’s $O(n^3)$
If $n=1024$, it takes about 1 sec
How long is it take when $n=2048$?
Matrix Multiplication

- If each dimension of your matrix is 2048
  - Each row takes 2048*8 bytes = 16KB
  - The L1 $ of intel Core i7 is 32KB, 8-way, 64-byte blocked
  - You can only hold at most 2 rows/columns of each matrix!
  - You need the same row when j increase!

```c
for(i = 0; i < ARRAY_SIZE; i++) {
    for(j = 0; j < ARRAY_SIZE; j++) {
        for(k = 0; k < ARRAY_SIZE; k++) {
            c[i][j] += a[i][k]*b[k][j];
        }
    }
}
```
Block algorithm for matrix multiplication

• Discover the cache miss rate
  • `valgrind --tool=cachegrind cmd`
    • cachegrind is a tool profiling the cache performance
• Performance counter
  • Intel® Performance Counter Monitor http://www.intel.com/software/pcm/
Block algorithm for matrix multiplication

```c
for(i = 0; i < ARRAY_SIZE; i++) {
    for(j = 0; j < ARRAY_SIZE; j++) {
        for(k = 0; k < ARRAY_SIZE; k++) {
            c[i][j] += a[i][k]*b[k][j];
        }
    }
}
```

You only need to hold these sub-matrices in your cache
Recap: optimizations

• Software
  • Data layout — capacity miss, conflict miss, compulsory miss
  • Blocking — capacity miss, conflict miss
  • Loop fission — conflict miss — when $ has limited way associativity
  • Loop fusion — capacity miss — when $ has enough way associativity
  • Loop interchange — conflict/capacity miss

• Hardware
  • Increase block size — compulsory miss, but increase miss penalty
  • Increase way associativity — conflict miss, but increase hit time
  • Increase capacity — capacity miss, but $$$
Virtual memory
Virtual memory

- An abstraction of memory space available for programs/software/programmer
- Programs execute using virtual memory address
- The operating system and hardware work together to handle the mapping between virtual memory addresses and real/physical memory addresses
- Virtual memory organizes memory locations into “pages”
The virtual memory abstraction

### Virtual Memory Space

<table>
<thead>
<tr>
<th>Page #1</th>
<th>Page #1</th>
<th>Page #1</th>
<th>Page #1</th>
<th>Page #1</th>
<th>Page #1</th>
<th>Page #1</th>
<th>Page #1</th>
<th>Page #1</th>
<th>Page #1</th>
<th>Page #1</th>
<th>Page #1</th>
<th>Page #1</th>
<th>Page #1</th>
<th>Page #1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x1000</td>
<td>0x2000</td>
<td>0x3000</td>
<td>0x4000</td>
<td>0x5000</td>
<td>0x6000</td>
<td>0x7000</td>
<td>0x8000</td>
<td>0x0000</td>
<td>0x1000</td>
<td>0x2000</td>
<td>0x3000</td>
<td>0x4000</td>
<td>0x5000</td>
</tr>
<tr>
<td>AAA</td>
<td>AAA</td>
<td>AAA</td>
<td>AAA</td>
<td>AAA</td>
<td>AAA</td>
<td>AAA</td>
<td>AAA</td>
<td>AAA</td>
<td>AAA</td>
<td>AAA</td>
<td>AAA</td>
<td>AAA</td>
<td>AAA</td>
<td>AAA</td>
</tr>
<tr>
<td>BBB</td>
<td>BBB</td>
<td>BBB</td>
<td>BBB</td>
<td>BBB</td>
<td>BBB</td>
<td>BBB</td>
<td>BBB</td>
<td>BBB</td>
<td>BBB</td>
<td>BBB</td>
<td>BBB</td>
<td>BBB</td>
<td>BBB</td>
<td>BBB</td>
</tr>
<tr>
<td>CCC</td>
<td>CCC</td>
<td>CCC</td>
<td>CCC</td>
<td>CCC</td>
<td>CCC</td>
<td>CCC</td>
<td>CCC</td>
<td>CCC</td>
<td>CCC</td>
<td>CCC</td>
<td>CCC</td>
<td>CCC</td>
<td>CCC</td>
<td>CCC</td>
</tr>
<tr>
<td>DDD</td>
<td>DDD</td>
<td>DDD</td>
<td>DDD</td>
<td>DDD</td>
<td>DDD</td>
<td>DDD</td>
<td>DDD</td>
<td>DDD</td>
<td>DDD</td>
<td>DDD</td>
<td>DDD</td>
<td>DDD</td>
<td>DDD</td>
<td>DDD</td>
</tr>
<tr>
<td>EEE</td>
<td>EEEE</td>
<td>EEEE</td>
<td>EEEE</td>
<td>EEEE</td>
<td>EEEE</td>
<td>EEEE</td>
<td>EEEE</td>
<td>EEEE</td>
<td>EEEE</td>
<td>EEEE</td>
<td>EEEE</td>
<td>EEEE</td>
<td>EEEE</td>
<td>EEEE</td>
</tr>
<tr>
<td>FFF</td>
<td>FFFF</td>
<td>FFFF</td>
<td>FFFF</td>
<td>FFFF</td>
<td>FFFF</td>
<td>FFFF</td>
<td>FFFF</td>
<td>FFFF</td>
<td>FFFF</td>
<td>FFFF</td>
<td>FFFF</td>
<td>FFFF</td>
<td>FFFF</td>
<td>FFFF</td>
</tr>
<tr>
<td>GGG</td>
<td>GGG</td>
<td>GGG</td>
<td>GGG</td>
<td>GGG</td>
<td>GGG</td>
<td>GGG</td>
<td>GGG</td>
<td>GGG</td>
<td>GGG</td>
<td>GGG</td>
<td>GGG</td>
<td>GGG</td>
<td>GGG</td>
<td>GGG</td>
</tr>
<tr>
<td>HHH</td>
<td>HHH</td>
<td>HHH</td>
<td>HHH</td>
<td>HHH</td>
<td>HHH</td>
<td>HHH</td>
<td>HHH</td>
<td>HHH</td>
<td>HHH</td>
<td>HHH</td>
<td>HHH</td>
<td>HHH</td>
<td>HHH</td>
<td>HHH</td>
</tr>
</tbody>
</table>
```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

#define _GNU_SOURCE

double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    fprintf(stderr, "Process %d is using CPU: %d. Value of a is %lf and address of a is %p\n", getpid(), cpu, a, &a);
    sleep(10);
    fprintf(stderr, "Process %d is using CPU: %d. Value of a is %lf and address of a is %p\n", getpid(), cpu, a, &a);
    return 0;
}
```

**Demo revisited**

\[&a = 0x601090\]
Address translation

- Processor receives virtual addresses from the running code, main memory uses physical memory addresses
- Virtual address space is organized into “pages”
- The system references the **page table** to translate addresses
  - Each process has its own page table
  - The page table content is maintained by OS
Size of page table

• Assume that we have 64-bit virtual address space, each page is 4KB, each page table entry is 8 Bytes, what magnitude in size is the page table for a process?

A. MB — $2^{20}$ Bytes
B. GB — $2^{30}$ Bytes
C. TB — $2^{40}$ Bytes
D. PB — $2^{50}$ Bytes
E. EB — $2^{60}$ Bytes

\[
\frac{2^{64} \text{ Bytes}}{4 \text{ KB}} \times 8 \text{ Bytes} = 2^{55} \text{ Bytes} = 32 \text{ PB}
\]

If you still don’t know why — you need to take CSE120
Do we really need a large table?

Dynamic allocated data: `malloc()`

Local variables, arguments

Virtual memory

Virtual memory

0x0000000000000000

0xFFFFFFFFFFFFFFFF
Address translation in x86-64

<table>
<thead>
<tr>
<th>63:48 (16)</th>
<th>47:39 (9 bits)</th>
<th>38:30 (9 bits)</th>
<th>29:21 (9 bits)</th>
<th>20:12 (9 bits)</th>
<th>11:0 (12 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SignExt</td>
<td>L4 index</td>
<td>L3 index</td>
<td>L2 index</td>
<td>L1 index</td>
<td>page offset</td>
</tr>
</tbody>
</table>

X86 Processor

CR3 Reg.

512 entries

512 entries

512 entries

512 entries

11:0 (12 bits)

physical page #

page offset
Address translation in x86-64

May have 10 memory accesses for a “MOV” instruction! — 5 for instruction fetch and 5 for data access
TLB: Translation Look-aside Buffer

- TLB — a small SRAM stores frequently used page table entries
- Good — A lot faster than having everything going to the DRAM
- Bad — Still on the critical path
TLB + Virtual cache

- L1 $ accepts virtual address — you don’t need to translate
- Good — you can access both TLB and L1-$ at the same time and physical address is only needed if L1-$ misses
- Bad — it doesn’t work in practice
  - Many applications have the same virtual address but should be pointing different physical addresses
  - An application can have “aliasing virtual addresses” pointing to the same physical address

You really need “physical address” to judge if that’s what you want
Virtually indexed, physically tagged cache

- Can we find physical address directly in the virtual address — Not everything — but the page offset isn’t changing!
- Can we indexing the cache using the “partial physical address”? — Yes — Just make set index + block set to be exactly the page offset
Virtually indexed, physically tagged cache

<table>
<thead>
<tr>
<th>virtual page #</th>
<th>physical page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x29 0x45</td>
</tr>
<tr>
<td>1</td>
<td>0xDE 0x68</td>
</tr>
<tr>
<td>1</td>
<td>0x10 0xA1</td>
</tr>
<tr>
<td>0</td>
<td>0x8A 0x98</td>
</tr>
</tbody>
</table>

memory address: 0x0 0x8 0x2 0x4

V D tag

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0x00</td>
<td>AABBCDDEEGGFFHH</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x10</td>
<td>IIJJKLLMMNNOOPP</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0xA1</td>
<td>QQRRSTTUUVVVWMXX</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0x10</td>
<td>YYZZAABBCDDEEFF</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x31</td>
<td>AABBCDDEEGGFFHH</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x45</td>
<td>IIJJKLLMMNNOOPP</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0x41</td>
<td>QQRRSTTUUVVVWMXX</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0x68</td>
<td>YYZZAABBCDDEEFF</td>
</tr>
</tbody>
</table>

Hit?
Virtually indexed, physically tagged cache

- If page size is 4KB —

\[ \log(B) + \log(S) = \log(4096) = 12 \]

\[ C = ABS \]

\[ C = A \times 2^{12} \]

if \( A = 1 \)

\[ C = 4KB \]
Dynamic instruction scheduling/
Out-of-order (OoO) execution
Tips of drawing a pipeline diagram

• Each instruction has to go through all 5 pipeline stages: IF, ID, EXE, MEM, WB in order — only valid if it’s single-issue, MIPS 5-stage pipeline

• An instruction can enter the next pipeline stage in the next cycle if
  • No other instruction is occupying the next stage
  • This instruction has completed its own work in the current stage
  • The next stage has all its inputs ready

• Fetch a new instruction only if
  • We know the next PC to fetch
  • We can predict the next PC
  • Flush an instruction if the branch resolution says it’s mis-predicted.
What do you need to execute an instruction?

• Whenever the instruction is decoded — put decoded instruction somewhere
• Whenever the inputs are ready — all data dependencies are resolved
• Whenever the target functional unit is available

• This instruction has completed its own work in the current stage
• No other instruction is occupying the next stage
• The next stage has all its inputs ready
Scheduling instructions: based on data dependencies

- Draw the data dependency graph, put an arrow if an instruction depends on the other.
  ① lw $6,0($10)
  ② add $7, $6,$12
  ③ sw $7,0($10)
  ④ addi $10,$10, 8
  ⑤ bne $10, $5, LOOP
  ⑥ lw $6,0($10)
  ⑦ add $7, $6,$12
  ⑧ sw $7,0($10)
  ⑨ addi $10,$10, 8
  ⑩ bne $10, $5, LOOP

- **In theory**, instructions without dependencies can be executed in parallel or out-of-order
- Instructions with dependencies can never be reordered
If we can predict the future ...

Consider the following dynamic instructions:

1. lw $6, 0($10)
2. add $7, $6, $12
3. sw $7, 0($10)
4. addi $10, $10, 8
5. bne $10, $5, LOOP
6. lw $6, 0($10)
7. add $7, $6, $12
8. sw $7, 0($10)
9. addi $10, $10, 8
10. bne $10, $5, LOOP

Which of the following pair can we reorder without affecting the correctness if the branch prediction is perfect?

A. (2) and (4)
B. (3) and (5)
C. (5) and (6)
D. (6) and (9)
E. (9) and (10)

We still can only reorder (5) and (6) even though (2) & (4) are not depending on each other!
False dependencies

• They are not “true” dependencies because they don’t have an arrow in data dependency graph
  • WAR (Write After Read): a later instruction overwrites the source of an earlier one
    • 4 and 1 4 and 3, 6 and 2, 7 and 3, 9 and 5, 9 and 6, 9 and 8
  • WAW (Write After Write): a later instruction overwrites the output of an earlier one
    • 6 and 1, 7 and 2

• False dependencies coming from the sharing/competition of registers

① lw $6,0($10)
② add $7, $6,$12
③ sw $7,0($10)
④ addi $10,$10, 8
⑤ bne $10, $5, LOOP
⑥ lw $6,0($10)
⑦ add $7, $6,$12
⑧ sw $7,0($10)
⑨ addi $10,$10, 8
⑩ bne $10, $5, LOOP

① lw $6,0($10)
② add $7, $6,$12
③ sw $7,0($10)
④ addi $11,$10, 8
⑤ bne $11, $5, LOOP
⑥ lw $20,0($10)
⑦ add $21, $6,$12
⑧ sw $20,0($10)
⑨ addi $22,$10, 8
⑩ bne $22, $5, LOOP
Register renaming

• Provide a set of “physical registers” and a mapping table mapping “architectural registers” to “physical registers”
• Allocate a physical register for a new output
• Eliminate all false dependencies
• Stages
  • Dispatch (D) — allocate a “physical” for the output of a decoded instruction
  • Issue (I) — collect pending values/branch outcome from common data bus
  • Execute (INT, AQ/AQ/MEM, M1/M2/M3, BR) — send the instruction to its corresponding pipeline if no structural hazards
  • Write Back (WB) — broadcast the result through CDB
Overview of a processor supporting register renaming

Fetch/decode instruction ➔ Renaming logic

Instruction Queue ➔ Unresolved Branch

Physical Registers ➔ Register mapping table

Physical Registers ➔ Valid value

Register mapping table ➔ X1 register #

Address Resolution ➔_addr_

Integer ALU ➔ Addr.

Floating-Point Adder ➔ Value

Floating-Point Mul/Div ➔ Dest

Branch ➔ Load Queue

Memory ➔ Store Queue

Load Queue ➔ Address

Store Queue ➔ Data
Register renaming in motion

Takes 12 cycles to issue all instructions
Through data flow graph analysis

R  I  AQ  AR  MEM  WB

INT — 2 cycles for depending instruction to start
MEM — 4 cycles for the depending instruction to start
MUL/DIV — 4 cycles for the depending instruction to start
BR — 2 cycles to resolve
Super Scalar
Superscalar

• Since we have more functional units now, we should fetch/decode more instructions each cycle so that we can have more instructions to issue!

• Super-scalar: fetch/decode/issue more than one instruction each cycle
  • Fetch width: how many instructions can the processor fetch/decode each cycle
  • Issue width: how many instructions can the processor issue each cycle
Recap: Pipeline SuperScalar/OoO/ROB

Front-end:
- Instruction Fetch
- Instruction Decode
- Register renaming logic

Issue/Schedule:
- ALU
- MUL/DIV 1
- Address Resolution

Issue Width:
- FP1
- MUL/DIV 2
- Address Queue

Back-end:
- FP2
- MEM

Fetch Width:
- Branch predictor
Overview of a processor supporting register renaming

What if we widen the pipeline to fetch/issue two instructions at the same time?
2-issue RR processor in motion

```assembly
lw $6,0($10)
add $7,$6,$12
sw $7,0($10)
addi $10,$10,8
bne $10,$5,LOOP
lw $6,0($10)
add $7,$6,$12
sw $7,0($10)
addi $10,$10,8
bne $10,$5,LOOP
```

Renamed instruction

1 lw P1, 0($10)
2 add P2, P1, $12
3 sw P2, 0($10)
4 addi P3, $10, 8
5 bne P9, $5, LOOP
6 lw P4, 0(P3)
7 add P5, P1, $12
8 sw P5, 0(P3)
9 addi P6, P3, 8
10 bne P6, 0($10)

Physical Register

<table>
<thead>
<tr>
<th>Valid</th>
<th>Value</th>
<th>In use</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P10</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Recap: What about “linked list”

Static instructions
- LOOP: `ld X10, 8(X10)`
- `addi X7, X7, 1`
- `bne X10, X0, LOOP`

Dynamic instructions
1. `ld X10, 8(X10)`
2. `addi X7, X7, 1`
3. `bne X10, X0, LOOP`
4. `ld X10, 8(X10)`
5. `addi X7, X7, 1`
6. `bne X10, X0, LOOP`
7. `ld X10, 8(X10)`
8. `addi X7, X7, 1`
9. `bne X10, X0, LOOP`

ILP is low because of data dependencies

Wasted slots

Instruction Queue
SuperScalar Processor w/ ROB

Fetch/decode instruction

Renaming logic

Instruction Queue

Unresolved Branch

Register mapping table

Physical Registers

Address Resolution

Integer ALU

Floating-Point Adder

Floating-Point Mul/Div

Branch

Load Queue

Store Queue

Memory

Address

Dest Reg.

Addr.

Addr.

Value

Addr.

Data

85
Power v.s. Energy

- Power is the direct contributor of “heat”
  - Packaging of the chip
  - Heat dissipation cost
- Energy = P * ET
  - The electricity bill and battery life is related to energy!
  - Lower power does not necessary means better battery life if the processor slow down the application too much
Dynamic/Active Power

- The power consumption due to the switching of transistor states
- Dynamic power per transistor
  \[ P_{\text{dynamic}} \sim \alpha \times C \times V^2 \times f \times N \]
  - \( \alpha \): average switches per cycle
  - \( C \): capacitance
  - \( V \): voltage
  - \( f \): frequency, usually linear with \( V \)
  - \( N \): the number of transistors
The “power/energy cost” of doubling the clocking rate

\[ \text{Power}_{\text{new}} = \text{Power}_{\text{old}} \times \left( \frac{f_{\text{new}}}{f_{\text{old}}} \right)^3 \]

\[ \text{Power}_{\text{new}} = \text{Power}_{\text{old}} \times (2)^3 = \text{Power}_{\text{old}} \times 8 \]

\[ \text{Speedup} = \frac{\text{Execution Time}_{\text{baseline}}}{\text{Execution Time}_{\text{enhanced}}} = \frac{5}{4} = 1.25 \]

\[ \text{Energy}_{\text{new}} = \text{Power}_{\text{new}} \times ET_{\text{new}} \]

\[ = \text{Power}_{\text{new}} \times \frac{ET_{\text{old}}}{\text{Speedup}} \]

\[ = \text{Power}_{\text{old}} \times 8 \times \frac{ET_{\text{old}}}{1.25} = 6.4 \times \text{Power}_{\text{old}} \times ET_{\text{old}} \]

\[ = 6.4 \times \text{Energy}_{\text{old}} \]
Recap: Amdahl’s Law on Multicore Architectures

- Symmetric multicore processor with $n$ cores (if we assume the processor performance scales perfectly)

\[
\text{Speedup}_\text{parallel}(f_{\text{parallelizable}}, n) = \frac{1}{(1 - f_{\text{parallelizable}}) + \frac{f_{\text{parallelizable}}}{n}}
\]
What if we double the number of cores?

\[ \text{Power}_{new} = \text{Power}_{old} \times \text{number\_of\_cores} = \text{Power}_{old} \times 2 \]

Assume 40% of execution time can be parallelized

\[ \text{Speedup}_{\text{parallel}}(f_{\text{parallelizable}}, n) = \frac{1}{(1 - f_{\text{parallelizable}}) + \frac{f_{\text{parallelizable}}}{n}} = \frac{1}{(1 - 40\%) + \frac{40\%}{2}} = 1.25 \]

\[ \text{Energy}_{new} = \text{Power}_{new} \times \text{ET}_{new} = \text{Power}_{new} \times \frac{\text{ET}_{old}}{\text{Speedup}} = \text{Power}_{old} \times 2 \times \frac{\text{ET}_{old}}{1.25} = 1.6 \times \text{Power}_{old} \times \text{ET}_{old} \]

A better deal in terms of energy!  

\[ = 1.6 \times \text{Energy}_{old} \]
Concept of CMP

Processor

Core
Registers
L1-$_$
L2-$_$

Core
Registers
L1-$_$
L2-$_$

Core
Registers
L1-$_$
L2-$_$

Core
Registers
L1-$_$
L2-$_$

Last-level $ (LLC)$
What software thinks about “multiprogramming” hardware
What software thinks about “multiprogramming” hardware

Others do not see the updated value in the cache and keep working — incorrect result!
Coherent way-associative cache

memory address: $0x0$

memory address: $0b0000100000100100$

hit?

hit?
Snooping Protocol

- **Invalid**
  - Write miss (processor)
  - Write miss (bus) to **Shared**
  - Write request (processor) to **Exclusive**
  - Write miss (bus)

- **Shared**
  - Read miss (processor) to **Invalid**
  - Write miss (bus) to **Exclusive**
  - Write request (processor) to **Invalid**

- **Exclusive**
  - Write hit to **Shared**
  - Read miss (bus) to **Shared**
  - Write miss (bus) to **Invalid**

- Read/write miss (bus)
False sharing
Possible scenarios

Thread 1
\[ a=1; \]
\[ x=b; \]

Thread 2
\[ b=1; \]
\[ y=a; \]

Thread 1
\[ a=1; \]
\[ x=b; \]

Thread 2
\[ b=1; \]
\[ y=a; \]

Thread 1
\[ a=1; \]
\[ x=b; \]

Thread 2
\[ b=1; \]
\[ y=a; \]

(0,1)

Thread 1
\[ a=1; \]
\[ x=b; \]

Thread 2
\[ y=a; \]
\[ b=1; \]

(0,0)

(1,1)

(1,0)
• x86 provides an “mfence” instruction to prevent reordering across the fence instruction

• x86 only supports this kind of “relaxed consistency” model. You still have to be careful enough to make sure that your code behaves as you expected.

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=1; mfence</td>
<td>b=1; mfence</td>
</tr>
<tr>
<td>a=1 must occur/update before mfence</td>
<td>b=1 must occur/update before mfence</td>
</tr>
<tr>
<td>x=b;</td>
<td>y=a;</td>
</tr>
</tbody>
</table>
Power consumption to light on all transistors

<table>
<thead>
<tr>
<th>Chip</th>
<th>Dennardian Scaling</th>
<th>Dennardian Broken</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

=49W

=50W

=100W!
For final

- No cheating allowed
- Make sure you have stable internet connections
- 3-hour slots between 9/4 8a-6p.
- Format
  - Multiple choice question * 20
  - Free answer — calculation/operation intensive * 3 sets
  - Essay style questions * 10
Sample Final
Which description about locality of arrays sum and A in the following code is the most accurate?

```c
for(i = 0; i< 100000; i++)
{
    sum[i%10] += A[i];
}
```

A. Access of A has temporal locality, sum has spatial locality
B. Both A and sum have temporal locality, and sum also has spatial locality
C. Access of A has spatial locality, sum has temporal locality
D. Both A and sum have spatial locality
E. Both A and sum have spatial locality, and sum also has temporal locality
• L1 data (D-L1) cache configuration of AMD Phenom II
  • Size 64KB, 2-way set associativity, 64B block
  • Assume 64-bit memory address
Which of the following is correct?

A. Tag is 49 bits
B. Index is 8 bits
C. Offset is 7 bits
D. The cache has 1024 sets
E. None of the above
3Cs and A, B, C

• Regarding 3Cs: compulsory, conflict and capacity misses and A, B, C: associativity, block size, capacity

How many of the following are correct?

① Increasing associativity can reduce conflict misses
② Increasing associativity can reduce hit time
③ Increasing block size can increase the miss penalty
④ Increasing block size can reduce compulsory misses

A. 0
B. 1
C. 2
D. 3
E. 4
### What data structure is performing better

<table>
<thead>
<tr>
<th>Array of objects</th>
<th>object of arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct grades</td>
<td></td>
</tr>
<tr>
<td>```</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>int id;</td>
<td></td>
</tr>
<tr>
<td>double *homework;</td>
<td></td>
</tr>
<tr>
<td>double average;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>```</td>
<td></td>
</tr>
<tr>
<td>struct grades</td>
<td></td>
</tr>
<tr>
<td>```</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>int *id;</td>
<td></td>
</tr>
<tr>
<td>double **homework;</td>
<td></td>
</tr>
<tr>
<td>double *average;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>```</td>
<td></td>
</tr>
</tbody>
</table>

#### average of each homework

```c
for(i=0;i<homework_items; i++)
{
  gradesheet[total_number_students].homework[i] = 0.0;
  for(j=0;j<total_number_students; j++)
    gradesheet[total_number_students].homework[i] += gradesheet[j].homework[i];
  gradesheet[total_number_students].homework[i] /= (double)total_number_students;
}
```

```c
for(i = 0; i < homework_items; i++)
{
  gradesheet.homework[i][total_number_students] = 0.0;
  for(j = 0; j < total_number_students; j++)
    gradesheet.homework[i][j] += gradesheet.homework[i][j];
  gradesheet.homework[i][total_number_students] /= total_number_students;
}
```

- Considering your workload would like to calculate the average score of **one of the homework** for **all students**, which data structure would deliver better performance?
  A. Array of objects
  B. Object of arrays
Comparing the naive algorithm and block algorithm on matrix multiplication, what kind of misses does block algorithm help to remove? (assuming an intel Core i7)

A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. Capacity & conflict miss
E. Compulsory & conflict miss
If there is no abstraction between the processor and memory, the processor/cache needs to directly use main memory’s byte address to read/write data. How many of the following would be happening?

① The program’s memory footprint, including instructions/data, cannot exceed the capacity of the installed DRAM
② There is no guarantee the compiled program can execute on another machine if both machine have the same processor but different memory capacities
③ Two programs cannot run simultaneously if they use the same memory addresses
④ One program can maliciously access data from other concurrently executing programs

A. 0
B. 1
C. 2
D. 3
E. 4
Size of page table

Assume that we have 64-bit virtual address space, each page is 4KB, each page table entry is 8 Bytes, what magnitude in size is the page table for a process?

A. MB — $2^{20}$ Bytes
B. GB — $2^{30}$ Bytes
C. TB — $2^{40}$ Bytes
D. PB — $2^{50}$ Bytes
E. EB — $2^{60}$ Bytes
When we have virtual memory...

- If an x86 processor supports virtual memory through the basic format of the page table as shown in the previous slide, how many memory accesses can a `mov` instruction that access data memory once incur?
  
  A. 2  
  B. 4  
  C. 6  
  D. 8  
  E. 10
Virtual indexed, physical tagged cache limits the cache size

- If you want to build a virtual indexed, physical tagged cache with 32KB capacity, which of the following configuration is possible? Assume the operating system uses 4K pages.
  - A. 32B blocks, 2-way
  - B. 32B blocks, 4-way
  - C. 64B blocks, 4-way
  - D. 64B blocks, 8-way
False dependencies

- Consider the following dynamic instructions
  ① lw $12, 0($20)
  ② add $12, $10, $12
  ③ sub $18, $12, $10
  ④ lw $12, 8($20)
  ⑤ add $14, $18, $12
  ⑥ add $18, $14, $14
  ⑦ sw $14, 16($20)
  ⑧ addi $20, $20, 8

Which of the following pair is not a “false dependency”

A. (1) and (4)
B. (1) and (8)
C. (5) and (7)
D. (4) and (8)
E. (7) and (8)
What about “linked list”

- For the following C code and it’s translation in MIPS, how many cycles it takes the processor to issue all instructions? Assume the current PC is already at the first instruction and this linked list has only three nodes. This processor only fetches 1 instruction per cycle, with exactly the same register renaming hardware and pipeline as we showed previously.

```c
do {
    number_of_nodes++;
    current = current->next;
} while ( current != NULL )
```

```
LOOP: lw $10, 8($10)
      addi $7, $7, 1
      bne $10, $0, LOOP
```

A. 9  
B. 10  
C. 11  
D. 12  
E. 13  

Why does an Intel Core i7 @ 3.5 GHz usually perform better than an Intel Core i5 @ 3.5 GHz or AMD FX-8350@4GHz?

**Identify the limiting factor**

- A. Because the instruction count of the program are different
- B. Because the clock rate of AMD FX is higher
- C. Because the CPI of Core i7 is better
- D. Because the clock rate of AMD FX is higher and CPI of Core i7 is better
- E. None of the above

Why the performance is better when option is not “0”

1. The amount of dynamic instructions needs to execute is a lot smaller
2. The amount of branch instructions to execute is smaller
3. The amount of branch mis-predictions is smaller
4. The amount of data accesses is smaller

A. 0
   
   ```cpp
   if (option)
       std::sort(data, data + arraySize);
   ```

B. 1
   
   ```cpp
   for (unsigned i = 0; i < 100000; ++i) {
   ```

C. 2
   
   ```cpp
   int threshold = std::rand();
   for (unsigned i = 0; i < arraySize; ++i) {
       if (data[i] >= threshold)
           sum ++;
   ```

D. 3
   
   ```cpp
   ```

E. 4
Limitations of pipelining

• How many of the following descriptions about pipelining is correct?
  ① You can always divide stages into short stages with latches to improve performance
  ② Pipeline registers incur overhead for each pipeline stage
  ③ The latency of executing an instruction in a pipeline processor is longer than a single-cycle processor
  ④ The throughput of a pipeline processor is usually better than a single-cycle processor

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
Practicing Amdahl’s Law (2)

- Final Fantasy XV spends lots of time loading a map — within which period that 95% of the time on the accessing the H.D.D., the rest in the operating system, file system and the I/O protocol. If we replace the H.D.D. with a flash drive, which provides 100x faster access time and a better processor to accelerate the software overhead by 2x. By how much can we speed up the map loading process?
  
  A. ~7x
  B. ~10x
  C. ~17x
  D. ~29x
  E. ~100x
MIPS v.s. x86

Comparing x86 and MIPS ISAs, how many of the following statements is/are “generally” correct?

① x86 provides more instructions than MIPS
② x86 usually needs more instructions to express the same program
③ An x86 instruction may access memory for 3 times
④ An x86 instruction may be shorter than a MIPS instruction
⑤ An x86 instruction may be longer than a MIPS instruction

A. 1  
B. 2  
C. 3  
D. 4  
E. 5
Assuming that we are running the following code on a CMP with a cache coherency protocol, how many of the following outputs are possible? (a is initialized to 0 as assume we will output more than 10 numbers)

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>while(1) printf(&quot;%d &quot;,a);</td>
<td>while(1) a++;</td>
</tr>
</tbody>
</table>

① 0 1 2 3 4 5 6 7 8 9
② 1 2 5 9 3 6 8 10 12 13
③ 1 1 1 1 1 1 1 64 100
④ 1 1 1 1 1 1 1 1 100

A. 0
B. 1
C. 2
D. 3
E. 4
How many of the following about SMT are correct?

① SMT makes processors with deep pipelines more tolerable to mis-predicted branches
② SMT can improve the throughput of a single-threaded application
③ SMT processors can better utilize hardware during cache misses comparing with superscalar processors with the same issue width
④ SMT processors can have higher cache miss rates comparing with superscalar processors with the same cache sizes when executing the same set of applications.

A. 0
B. 1
C. 2
D. 3
E. 4
What happens if power doesn’t scale with process technologies?

- If we are able to cram more transistors within the same chip area (Moore’s law continues), but the power consumption per transistor remains the same. Right now, if put more transistors in the same area because the technology allows us to. How many of the following statements are true?
  ① The power consumption per chip will increase
  ② The power density of the chip will increase
  ③ Given the same power budget, we may not able to power on all chip area if we maintain the same clock rate
  ④ Given the same power budget, we may have to lower the clock rate of circuits to power on all chip area

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
Power & Energy

• Regarding power and energy, how many of the following statements are correct?

① Lowering the power consumption helps extending the battery life
② Lowering the power consumption helps reducing the heat generation
③ Lowering the energy consumption helps reducing the electricity bill
④ A CPU with 10% utilization can still consume 33% of the peak power

A. 0
B. 1
C. 2
D. 3
E. 4
D-L1 Cache configuration of AMD Phenom II

- Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.

```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

- What’s the overall cache miss rate?
- How many of the cache misses are **conflict** misses?
- Can you rewrite the code to eliminate all conflict misses?
Cache & Performance

• What’s the average CPI?
  • Application: 80% ALU, 20% Load/Store
  • L1 I-cache miss rate: 5%, hit time: 1 cycle
  • L1 D-cache miss rate: 10%, hit time: 1 cycle, 20% dirty
  • L2 U-Cache miss rate: 20%, hit time: 10 cycles, 10% dirty
  • Main memory hit time: 100 cycles
Consider the following dynamic instructions:

1. `lw  $1, 0($10)`
2. `addi $10, $10, 8`
3. `add  $20, $20, $1`
4. `bne  $10, $2, LOOP`
5. `lw  $1, 0($10)`
6. `addi $10, $10, 8`
7. `add  $20, $20, $1`
8. `bne  $10, $2, LOOP`

- Can you draw the data dependency graph?

Assume a MIPS 5-stage pipeline processor with 2-bit branch prediction and full data forwarding.

- Can you draw the pipeline diagram?

Assume a superscalar processor with issue width as 2 & unlimited physical registers that can fetch up to 4 instructions per cycle, 2 cycles for an integer instruction, 3 cycles to execute a memory instruction.

- Can you identify all false dependencies?

- How many cycles it takes to issue all instructions?
Recap: Four implementations

- Which of the following implementations will perform the best on modern pipeline processors?

### A
```c
inline int popcount(uint64_t x){
    int c=0;
    while(x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

### B
```c
inline int popcount(uint64_t x) {
    int c = 0;
    while(x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

### C
```c
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1, 2, 3, 1, 2, 2, 3, 3, 3, 3, 4};
    while(x) {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```

### D
```c
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    for(uint64_t i = 0; i < 16; i++) {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```

Why C and D outperforms A and B?
With what architecture component could this happen?
1. By adding the “sort” in the following code snippet, what the programmer changes in the performance equation to achieve better performance?

```cpp
std::sort(data, data + arraySize);

for (unsigned c = 0; c < arraySize * 1000; ++c) {
    if (data[c % arraySize] >= INT_MAX / 2)
        sum ++;
}

```

A. CPI
B. IC
C. CT
D. IC & CPI

2. What in the processor make this code outperform the same code without sorting?
• Assuming both application X and application Y have similar instruction combination, say 60% ALU, 20% load/store, and 20% branches. Consider two processors:

P1: CMP with a 2-issue pipeline on each core. Each core has a private L1 32KB D-cache

P2: SMT with a 4-issue pipeline. 64KB L1 D-cache

Which one do you think is better?
Essay style questions

• What’s SMT? What problem SMT solves? What are the pros & cons of SMT?
• What’s CMP? What’s the benefit of CMP? What’s the limitation of CMP?
• What’s coherence miss? When will it appear?
• Why do we need hardware dynamic scheduling given we have compiler optimizations?
• What are false dependencies? How can we remove false dependencies?
Essay style questions (cont.)

• What’s Amdahl’s Law implication on parallelism? How does that guide future software design?
• What’s Dark Silicon Problem? What are the new design trends in addressing the problem?
• If the OoO pipeline is highly optimized, do we still care about the ISA design?
• What’s volatile? What’s inline? Why we need them? The pros and cons?
• What is speculative execution? What is out-of-order execution? Can you potentially create buggy code without taking care of these processor features?
Announcements

• CAPE/Survey
  • Screenshot of your CAPE
  • Fill the survey
  • Count as a full-credit assignment and we’re dropping your lowest two assignments now.

• Assignment 5 is up — a mini final
  • Given that we’re dropping 2 lowest assignment grade and give you a full credit one once you submitted your post-CAPE survey, you probably don’t need to turn in that.
  • Strongly encourage to practice that since it covers the material for the last week of class

• Regarding final exam —
  • 9/4 8am—6pm — any consecutive, non-stop 3-hour slot you pick
  • Open books, open notes, but it’s going to be twice longer than the midterm
  • Not using Lockdown browser — since some of us claim difficulties with that
  • No zoom, no response to piazza posts for fairness
The End