Final Review

Hung-Wei Tseng
Final is cumulative

• Don’t forget to go through midterm & midterm review again.
Basic steps of execution

- Instruction fetch: fetch an instruction from memory
- Decode:
  - What’s the instruction?
  - Where are the operands?
- Execute
- Memory access
  - Where is my data? (The data memory address)
- Write back
  - Where to put the result
- Determine the next PC
The pipelined processor

memory accesses
Recap: why memory hierarchy?

The access time of DDR3-1600 DRAM is around 50ns, 100x to the cycle time of a 2GHz processor! SRAM is as fast as the processor, but $$$
Memory & Cache
Memory hierarchy

- **CPU**
  - Fastest, Most Expensive
  - Access time: < 1ns

- **Cache**
  - $\sim 1\text{ ns} - 20\text{ ns}$

- **Main Memory**
  - Access time: 50-60ns

- **Secondary Storage**
  - Access time: 10,000,000ns
  - Biggest
Locality

- **Temporal Locality**
  - Referenced item tends to be referenced again soon.

- **Spatial Locality**
  - Items close by referenced item tends to be referenced soon.
    - example: consecutive instructions, arrays
Where is locality?

• 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

**Spatial locality:**
A[0], A[1], A[2], A[3], ....  
sum[0], sum[1], ... , sum[9]

**Temporal locality:**
reuse of sum[0], sum[1], ... , sum[9]
The structure of a cache

**Set:** cache blocks/lines sharing the same index. A cache is called **N-way** set associative cache if N blocks share the same set/index (this one is a 2-way set cache).

**Tag:** the high order memory address bits stored along with the data in a block to identify the actual address of the cache line.

**Block / Cacheline:** The basic unit of data storage in cache. Contains all data with the same tag/prefix and index in their memory addresses.

**valid:** if the data is meaningful
**dirty:** if the block is modified
Accessing the cache

Hit: The data was found in the cache
Miss: The data was not found in the cache

Offset: The position of the requesting word in a cache block

Hit? Miss?

tag: tells us if that’s the same block we are asking for
index: tells us which set?

hit? miss?

memory address:

0x8 0 0 0 0 1 5 8

memory address:

0 100 0001 0000 1000 0000

0 100 0000 0000 0000 0000

valid
dirty
tag
data

valid
dirty
tag
data

index: tells us which set?
tag: tells us if that’s the same block we are asking for

1000 0000 0000 0000 0000 0001 0101 1000

0x8   0   0   0   0   1   5   8

Offset:
The position of the requesting word in a cache block
C = ABS

- **C**: Capacity
- **A**: Way-Associativity
  - How many blocks in a set
  - 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
  - offset bits: \( \log_2(B) \)
  - index bits: \( \log_2(S) \)
  - tag bits: \( \text{address\_length} - \log_2(S) - \log_2(B) \)
    - address\_length is 32 bits for 32-bit machine
L1 data (D-L1) cache configuration of Athlon 64
- 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

\[ C = \text{ABS} \]
\[ 64KB = 2 \times 64 \times S \]
\[ S = 512 \]
\[ \text{offset} = \lg(64) = 6 \text{ bits} \]
\[ \text{index} = \lg(512) = 9 \text{ bits} \]
\[ \text{tag} = 64 - \lg(512) - \lg(64) = 49 \text{ bits} \]
• 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

\[
\begin{align*}
  C &= \text{ABS} \\
  32\text{KB} &= 8 \times 64 \times S \\
  S &= 64 \\
  \text{offset} &= \lg(64) = 6 \text{ bits} \\
  \text{index} &= \lg(64) = 6 \text{ bits} \\
  \text{tag} &= 64 - \lg(64) - \lg(64) = 52 \text{ bits}
\end{align*}
\]
What happens on a read?

- Read hit
  - hit time
- Read miss?
  - Select victim block
    - LRU, random, FIFO, ...
  - Write back if dirty
- Fetch Data from Lower Memory Hierarchy
  - As a unit of a cache block
    - Data with the same “block address” will be fetch
  - Miss penalty
What happens on a write? (Write Allocate, write back)

- **Write hit?**
  - Update in-place
  - Set dirty bit (Write-Back Policy)

- **Write miss?**
  - Select victim block
    - LRU, random, FIFO, ...
    - Write back to lower memory hierarchy if dirty
  - Fetch Data from Lower Memory Hierarchy
    - As a unit of a cache block
    - Miss penalty
3Cs of misses

- Compulsory 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 2-way cache

- Consider a small 256B 2-way cache with 16 byte blocks, and the application repeatedly reading the following memory addresses:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000

- \( C = A B S \)
- \( S = 256/(16*2) = 8 \)
- \( \lg(8) = 3 : 3 \) bits are used for the index
- \( \lg(16) = 4 : 4 \) bits are used for the byte offset
- The tag is 32 - (3 + 4) = 25 bits
- For example: 0b\text{1000 0000 0000 0000 0000 0000 0000 0001 0000}
Simulate a 2-way cache

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0b100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0b100</td>
<td>1</td>
<td>0b110</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10 0000 0000</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>0b10 0000 1000</td>
<td>hit!</td>
</tr>
<tr>
<td>0b10 0001 0000</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>0b10 0001 0100</td>
<td>hit!</td>
</tr>
<tr>
<td>0b11 0001 0000</td>
<td>compulsory 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>
A direct-mapped (1-way) cache

Tag: the high order address bits stored along with the data to identify the actual address of the cache line.

Block (cacheline): The basic unit of data storage in cache. Contains all data with the same tag and index in their address block / cacheline.

memory address: 1000 0000 0000 0000 0000 0001 0101 1000

Tag: 1000 0000 0000 0000 0000

Hit: The data was found in the cache
Miss: The data was not found in the cache
Simulate a direct-mapped cache

- Consider a 256B direct mapped (1-way) cache with 16 byte blocks, and the application repeatedly reading the following memory addresses:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000

- \( C = A \ B \ S \)
- \( S = \frac{256}{(16 \times 1)} = 16 \)
- \( \lg(16) = 4 \) : 4 bits are used for the index
- \( \lg(16) = 4 \) : 4 bits are used for the byte offset
- The tag is \( 32 - (4 + 4) = 24 \) bits
- For example: 0b\textbf{1000 0000 0000 0000 0000 0000 0000 1000 0000}
Simulate a direct-mapped cache

<table>
<thead>
<tr>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0b10</td>
<td>0000 1000</td>
</tr>
<tr>
<td>0b10</td>
<td>0001 0000</td>
</tr>
<tr>
<td>0b10</td>
<td>0001 0100</td>
</tr>
<tr>
<td>0b11</td>
<td>0001 0000</td>
</tr>
<tr>
<td>0b10</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0b10</td>
<td>0000 1000</td>
</tr>
<tr>
<td>0b10</td>
<td>0001 0000</td>
</tr>
<tr>
<td>0b10</td>
<td>0001 0100</td>
</tr>
</tbody>
</table>
• If you have multiple blocks need to compete the same set, the **next time** you access a block that is kicked out, it’s a **conflict miss**

• In a direct-mapped cache, if these two block are usually used back-to-back, one will kick out the other all the time. You will see lots of conflict misses
Tips for cache simulation

- Figure out the memory access patterns
  - Address sequences from your code
  - The behavior/locality of the variables/arrays
- Partition the address
  - Use C=ABS
  - Find out tag, index
- Check your current cache content
  - Hit: for the same index, if you can find the same tag there.
  - Otherwise, miss
    - Compulsory misses: you never accessed the same (tag,index) pair before
    - Conflict misses: the tag appeared in the same index before
    - Replace the least recently used block with the requesting block
Athlon 64

- Size 64KB, 2-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[i], load b[i], store c[i]*/
```

<table>
<thead>
<tr>
<th>Address in hex</th>
<th>Tag</th>
<th>Address in Binary</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>10 0000 0000 0000 0000</td>
<td>0x4</td>
<td>0</td>
<td>compulsory miss</td>
<td></td>
</tr>
<tr>
<td>load b[0]</td>
<td>0x20000</td>
<td>10 0000 0000 0000 0000</td>
<td>0x6</td>
<td>0</td>
<td>compulsory miss</td>
<td></td>
</tr>
<tr>
<td>store c[0]</td>
<td>0x10000</td>
<td>1 0000 0000 0000 0000</td>
<td>0x2</td>
<td>0</td>
<td>compulsory miss, evict 0x4</td>
<td></td>
</tr>
<tr>
<td>load a[1]</td>
<td>0x20004</td>
<td>10 0000 0000 0000 0000</td>
<td>0x4</td>
<td>0</td>
<td>conflict miss, evict 0x6</td>
<td></td>
</tr>
<tr>
<td>load b[1]</td>
<td>0x20004</td>
<td>10 0000 0000 0000 0000</td>
<td>0x6</td>
<td>0</td>
<td>conflict miss, evict 0x2</td>
<td></td>
</tr>
<tr>
<td>store c[1]</td>
<td>0x10004</td>
<td>1 0000 0000 0000 0000</td>
<td>0x2</td>
<td>0</td>
<td>conflict miss, evict 0x4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>load a[15]</td>
<td>0x2003C</td>
<td>10 0000 0000 0111 1100</td>
<td>0x4</td>
<td>0</td>
<td>conflict miss, evict 0x6</td>
<td></td>
</tr>
<tr>
<td>load b[15]</td>
<td>0x2003C</td>
<td>10 0000 0000 0111 1100</td>
<td>0x6</td>
<td>0</td>
<td>conflict miss, evict 0x2</td>
<td></td>
</tr>
<tr>
<td>store c[15]</td>
<td>0x1003C</td>
<td>1 0000 0000 0111 1100</td>
<td>0x2</td>
<td>0</td>
<td>conflict miss, evict 0x4</td>
<td></td>
</tr>
<tr>
<td>load a[16]</td>
<td>0x20040</td>
<td>10 0000 0000 0100 0000</td>
<td>0x4</td>
<td>1</td>
<td>compulsory miss</td>
<td></td>
</tr>
<tr>
<td>load b[16]</td>
<td>0x20040</td>
<td>10 0000 0000 0100 0000</td>
<td>0x6</td>
<td>1</td>
<td>compulsory miss</td>
<td></td>
</tr>
<tr>
<td>store c[16]</td>
<td>0x10040</td>
<td>1 0000 0000 0100 0000</td>
<td>0x2</td>
<td>1</td>
<td>compulsory miss, evict 0x4</td>
<td></td>
</tr>
</tbody>
</table>

\[
C = \text{ABS} \\
64KB = 2 * 64 * S \\
S = 512 \\
\text{offset} = \text{lg}(64) = 6 \text{ bits} \\
\text{index} = \text{lg}(512) = 9 \text{ bits} \\
\text{tag} = \text{the rest bits}
\]
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 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>
</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 0100 0000</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>load</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>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>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>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>b[1]</td>
<td>0x40008</td>
<td>100 0000 0000 0000 1000</td>
<td>0x8</td>
<td>0</td>
<td>hit</td>
</tr>
</tbody>
</table>

load a[128], store b[*128+j] (j = 2 ~ 7), For A, it's always a "compulsory miss". For B it's always a hit except for compulsory misses when j%8==0 until...

<table>
<thead>
<tr>
<th>load</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>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>b[8]</td>
<td>0x40040</td>
<td>100 0000 0000 0100 0000</td>
<td>0x8</td>
<td>0x1</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>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>b[9]</td>
<td>0x40048</td>
<td>100 0000 0000 0100 1000</td>
<td>0x8</td>
<td>0x1</td>
<td>hit</td>
</tr>
</tbody>
</table>

load a[128], store b[*128+j] (j = 10 ~ 31), For A, it's always a "compulsory miss". For B it's always a hit except for compulsory misses when j%8==0 until...

<table>
<thead>
<tr>
<th>load</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>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>a[8192]</td>
<td>0x30000</td>
<td>11 0000 0000 0000 0000</td>
<td>0x6</td>
<td>0</td>
<td>compulsory miss, evict something in index 0</td>
</tr>
</tbody>
</table>

load a[128], store b[*128+j] (j = 33 ~ 63), For A, it's always a "compulsory miss". For B it's always a hit except for compulsory misses when j%8==0 until...

<table>
<thead>
<tr>
<th>load</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>a[8192]</td>
<td>0x30000</td>
<td>11 0000 0000 0000 0000</td>
<td>0x6</td>
<td>0</td>
<td>compulsory miss, evict something in index 0</td>
</tr>
<tr>
<td>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>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>
</tbody>
</table>

load a[128], store b[*128+j] For A, it's always a "miss". For B it's always a hit except for compulsory misses when j%8==0 until the end!
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
    - Victim cache to reduce the miss penalty of conflict misses
- Prefetch: let misses occur before we need data
- Write buffers: reduce the penalty of a write miss
- Or modify the memory access pattern of your program!
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?

\[
CPI_{\text{Average}} = CPI_{\text{base}} + \text{miss}_rate \times \text{miss}_\text{penalty} \\
= 1 + 100\% \times (5\% \times (10 + 20\% \times ((1 + 10\%) \times 100))) \\
+ 20\% \times (10\% \times (1 + 20\%) \times (10 + 20\% \times ((1 + 10\%) \times 100))) \\
= 3.368
\]
Virtual memory
Why Virtual Memory?

- Every process runs on “virtual memory space”
  - Process: a running program in the operating system
- Physical memory “caches” memory “pages” from virtual memory
- Single program can exceed the size of physical memory.
- Multiple processes can share a single main memory.
Main memory is a cache for “Virtual Memory”
- A: Fully Associative
- B: page size!
- S: 1 (Since it’s FA)
- Replacement policies?
  - LRU, random...
- Operating system manages the mapping between physical and virtual addresses
Address translation

- Processor uses virtual addresses, 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 32-bit virtual address space, each page is 4KB, each page table entry is 4 bytes, how big is the page table for a process?

A. 1MB
B. 2MB
C. 4MB
D. 8MB
E. 16MB

What if we have 16 processes?

4 MB * 16 = 64MB

What if it’s 64-bit address space?

(2^{64}B/4KB)*4B = 2^4PB
Hierarchical page table

- Break the virtual page number into several pieces
- If one piece has $N$ bits, build an $2^N$-ary tree
- Only store the part of the tree that contain valid pages
- Walk down the tree to translate the virtual address
Hierarchical page table

- Only store the valid second level pages.
Cache + Virtual Memory

- TLB: Translation Look-aside Buffer
  - a cache of page table
    - small, high-associativity
    - miss penalty: access to page table in main memory

Too slow!

50 ns+ latency
Cache+Virtual Memory

- Virtual Cache
  - The cache also uses virtual addresses
  - Address translation is required only when miss.

- What if two processes (different running programs) accessing the same virtual address (map to different physical locations)?
Virtually indexed, physically tagged cache

- Avoid the aliasing problem of virtual caches
- Force aliasing virtual addresses mapped to the same cache location.
  - The cache uses the “index” field to place data blocks
  - Page offset remains the same in virtual and physical addresses
  - Index field must be inside the page offset to guarantee that aliasing are mapped to the same place
- Cache stores tag fields of “physical addresses”
Virtually indexed, physically tagged cache

C = ABS
\[ \lg(S) + \lg(B) = 12 \]
if A = 1 (DM cache)
\[ C = 1 \times (2^{12}) = 4KB \]
If you want to build a virtual indexed, physical tagged cache with 32KB capacity, which of the following configuration is possible? Assume the system use 4K pages.

A. 32B blocks, 2-way
B. 32B blocks, 4-way
C. 64B blocks, 4-way
D. 64B blocks, 8-way

The page offset of 4KB-pages is \( \lg(4K) = \lg(4096) = 12 = \lg(B) + \lg(S) \)

\[ C = \text{ABS} \]
\[ \lg(C) = \lg(A) + \lg(B) + \lg(S) \]
\[ 15 = \lg(A) + 12 \]
Cache & Performance

- The processor runs @ 2GHz. 20% are L/S
  - L1 I-cache miss rate: 5%, hit time: 1 cycle
  - L1 D-cache miss rate: 10%, hit time: 1 cycle, 10% evicted blocks are dirty
  - L2 U-Cache miss rate: 20%, hit time: 10 cycles, 20% evicted blocks are dirty
  - L1 TLB miss rate: 1%, hit time < 1 cycle
    - 200 cycles penalty
  - Main memory hit time: 100 cycles
  - All caches are write-back, write-allocate

\[ \text{CPI}_{\text{average}} = 1 + 20\% \times (1\% \times 200 + 10\% \times (1 + 10\%) \times (10 + 20\% \times (1 + 20\%) \times (100))) \]
\[ + 1\% \times 200 + 1 \times (5\% \times (10 + 20\% \times (1 + 20\%) \times (100))) \]
\[ = 5.85 \]
SuperScalar
Running compiler optimized code

- We can use compiler optimization to reorder the instruction sequence
- Compiler optimization requires no hardware change

3 cycles if the processor predicts branch perfectly, CPI = 0.75
Limitations of compiler optimizations

- Compiler can only see/optimize **static instructions**, instructions in the compiled binary
- Compiler cannot optimize **dynamic instructions**, the real instruction sequence when executing the program
  - Compiler cannot re-order 3, 5 or 4,5
  - Compiler cannot predict cache misses
- Compiler optimization is constrained by **false dependencies** due to limited number of registers (even worse for x86)
  - Instructions `lw $t1, 0($a0)` and `addi $a0, $a0, 4` do not depend on each other
- Compiler optimizations do not work for all architectures
  - The code optimization in the previous example works for single pipeline, but not for superscalar

<table>
<thead>
<tr>
<th>Static instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP: lw $t1, 0($a0)</td>
</tr>
<tr>
<td>addi $a0, $a0, 4</td>
</tr>
<tr>
<td>add $v0, $v0, $t1</td>
</tr>
<tr>
<td>bne $a0, $t0, LOOP</td>
</tr>
<tr>
<td>lw $t0, 0($sp)</td>
</tr>
<tr>
<td>lw $t1, 4($sp)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dynamic instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: lw $t1, 0($a0)</td>
</tr>
<tr>
<td>2: addi $a0, $a0, 4</td>
</tr>
<tr>
<td>3: add $v0, $v0, $t1</td>
</tr>
<tr>
<td>4: bne $a0, $t0, LOOP</td>
</tr>
<tr>
<td>5: lw $t1, 0($a0)</td>
</tr>
<tr>
<td>6: addi $a0, $a0, 4</td>
</tr>
<tr>
<td>7: add $v0, $v0, $t1</td>
</tr>
<tr>
<td>8: bne $a0, $t0, LOOP</td>
</tr>
</tbody>
</table>
Dynamic out-of-order execution
Scheduling instructions: based on data dependencies

• Draw the data dependency graph, put an arrow if an instruction depends on the other.
  • RAW (Read after write)

1: lw $t1, 0($a0)  
2: addi $a0, $a0, 4  
3: add $v0, $v0, $t1  
4: bne $a0, $t0, LOOP  
5: lw $t1, 0($a0)  
6: addi $a0, $a0, 4  
7: add $v0, $v0, $t1  
8: bne $a0, $t0, LOOP

• In theory, instructions without dependencies can be executed in parallel or out-of-order
• Instructions with dependencies can never be reordered
Consider the following dynamic instructions:

1: lw $t1, 0($a0)
2: addi $a0, $a0, 4
3: add $v0, $v0, $t1
4: bne $a0, $t0, LOOP
5: lw $t1, 0($a0)
6: addi $a0, $a0, 4
7: add $v0, $v0, $t1
8: bne $a0, $t0, LOOP

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

A. 1 and 2
B. 3 and 5
C. 3 and 6
D. 4 and 5
E. 4 and 6
False dependencies

- We are still limited by **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
    - 1 and 2, 3 and 5, 5 and 6
  - WAW (Write After Write): a later instruction overwrites the output of an earlier one
    - 1 and 5

```
1: lw  $t1, 0($a0)
2: addi $a0, $a0, 4
3: add  $v0, $v0, $t1
4: bne  $a0, $t0, LOOP
5: lw  $t1, 0($a0)
6: addi $a0, $a0, 4
7: add  $v0, $v0, $t1
8: bne  $a0, $t0, LOOP
```
Consider the following dynamic instructions:

1: lw $t2, 0($a0)
2: add $t2, $t0, $t2
3: sub $t8, $t2, $t0
4: lw $t2, 4($a0)
5: add $t4, $t8, $t2
6: add $t8, $t4, $t4
7: sw $t4, 8($a0)
8: addi $a0, $a0, 4

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

A. 1 and 4               WAW
B. 1 and 8               WAR
C. 5 and 7               True dependency (RAW)
D. 4 and 8               WAR
E. 7 and 8               WAR
Register renaming

• We can remove false dependencies if we can store each new output in a different register
• Architectural registers: an abstraction of registers visible to compilers and programmers
  • Like MIPS $0 -- $31
• Physical registers: the internal registers used for execution
  • Larger number than architectural registers
  • Modern processors have 128 physical registers
  • Invisible to programmers and compilers
• Maintains a mapping table between “physical” and “architectural” registers
Register renaming

Original code
1: lw $t1, 0($a0)
2: addi $a0, $a0, 4
3: add $v0, $v0, $t1
4: bne $a0, $t0, LOOP
5: lw $t1, 0($a0)
6: addi $a0, $a0, 4
7: add $v0, $v0, $t1
8: bne $a0, $t0, LOOP

After renamed
1: lw $p5, 0($p1)
2: addi $p6, $p1, 4
3: add $p7, $p4, $p5
4: bne $p6, $p2, LOOP
5: lw $p8, 0($p6)
6: addi $p9, $p6, 4
7: add $p10, $p7, $p8
8: bne $p9, $p2, LOOP

Register map

<table>
<thead>
<tr>
<th>cycle</th>
<th>$a0</th>
<th>$t0</th>
<th>$t1</th>
<th>$v0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p1</td>
<td>p2</td>
<td>p3</td>
<td>p4</td>
</tr>
<tr>
<td>1</td>
<td>p1</td>
<td>p2</td>
<td>p5</td>
<td>p4</td>
</tr>
<tr>
<td>2</td>
<td>p6</td>
<td>p2</td>
<td>p5</td>
<td>p4</td>
</tr>
<tr>
<td>3</td>
<td>p6</td>
<td>p2</td>
<td>p5</td>
<td>p7</td>
</tr>
<tr>
<td>4</td>
<td>p6</td>
<td>p2</td>
<td>p5</td>
<td>p7</td>
</tr>
<tr>
<td>5</td>
<td>p6</td>
<td>p2</td>
<td>p8</td>
<td>p7</td>
</tr>
<tr>
<td>6</td>
<td>p9</td>
<td>p2</td>
<td>p8</td>
<td>p7</td>
</tr>
<tr>
<td>7</td>
<td>p9</td>
<td>p2</td>
<td>p8</td>
<td>p10</td>
</tr>
<tr>
<td>8</td>
<td>p9</td>
<td>p2</td>
<td>p8</td>
<td>p10</td>
</tr>
</tbody>
</table>
OoO SuperScalar Processor

- Fetch instructions in the instruction window
- Register renaming to eliminate false dependencies
- Schedule an instruction to execution stage (issue) whenever all data inputs are ready for the instruction
- Put the instruction in reorder buffer and commit the instruction if the instruction is (1) not mis-predicted and (2) all the instruction prior to this instruction are committed
Dynamic execution with register naming

- Register renaming with unlimited physical registers, dynamical scheduling with 2-issue pipeline
- Assume that we fetch/decode/renaming/retire 4 instructions into/from instruction window each cycle
- Assume load needs 2 cycles to execute (one cycle address calculation and one cycle memory access)

Assumptions:
- Instruction fetch/decode/renaming/retire each cycle:
  1. lw $p5, 0($p1)
  2. addi $p6, $p1, 4
  3. add $p7, $p4, $p5
  4. bne $p6, $p2, LOOP
  5. lw $p8, 0($p6)
  6. addi $p9, $p6, 4
  7. add $p10, $p7, $p8
  8. bne $p9, $p2, LOOP

Graphical representation:
- After renamed, 1-4 and 5 are issues before 3
- 4 and 5 cannot issue because the issue width is only 2
- Only 2 instructions can be issued per cycle
Dynamic execution with register naming

- Consider the following dynamic instructions
  1: lw   $t1, 0($a0)
  2: lw   $a0, 4($a0)
  3: add  $v0, $v0, $t1
  4: bne  $a0, $zero, LOOP
  5: lw   $t1, 0($a0)
  6: lw   $t2, 4($a0)
  7: add  $v0, $v0, $t1
  8: bne  $t2, $zero, LOOP

Assume a superscalar processor with 4-issue width & unlimited physical registers that can fetch up to 4 instructions per cycle, 2 cycles to execute a memory instruction how many cycles it takes to issue all instructions?

A. 1  
B. 2  
C. 3  
D. 4  
E. 5
Simultaneous Multi-Threading (SMT)
Simplified SMT-OOO pipeline
Assume a superscalar/OoO/SMT processor with 4-issue width & unlimited physical registers that can fetch up to 4 instructions from each thread per cycle, 2 cycles to execute a memory instruction how many cycles it takes to issue all instructions?

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lw $t1, 0($a0)</td>
<td>lw $t1, 0($a0)</td>
</tr>
<tr>
<td>2</td>
<td>lw $a0, 4($a0)</td>
<td>lw $a0, 4($a0)</td>
</tr>
<tr>
<td>3</td>
<td>add $v0, $v0, $t1</td>
<td>add $v0, $v0, $t1</td>
</tr>
<tr>
<td>4</td>
<td>bne $a0, $zero, LOOP</td>
<td>bne $a0, $zero, LOOP</td>
</tr>
<tr>
<td>5</td>
<td>lw $t1, 0($a0)</td>
<td>lw $t1, 0($a0)</td>
</tr>
<tr>
<td>6</td>
<td>lw $t2, 4($a0)</td>
<td>lw $t2, 4($a0)</td>
</tr>
<tr>
<td>7</td>
<td>add $v0, $v0, $t1</td>
<td>add $v0, $v0, $t1</td>
</tr>
<tr>
<td>8</td>
<td>bne $t2, $zero, LOOP</td>
<td>bne $t2, $zero, LOOP</td>
</tr>
</tbody>
</table>

Cannot issue B3, B4 in cycle 3 because the issue width is only 4.
Chip multiprocessor (CMP)
A wide-issue processor or multiple narrower-issue processors

What can you do within a 21 mm * 21 mm area?

A 6-issue superscalar processor
3 integer ALUs
3 floating point ALUs
3 load/store units

4 2-issue superscalar processor
4*1 integer ALUs
4*1 floating point ALUs
4*1 load/store units

You will have more ALUs if you choose this!

Figure 2. Floorplan for the six-issue dynamic superscalar processor

Figure 3. Floorplan for the four-way single-chip superscalar multiprocessor.
Die photo of a CMP processor
Memory hierarchy on CMP

- Each processor has its own local cache
Parallel programming

- To exploit CMP/SMT parallelism you need to break your computation into multiple “processes” or multiple “threads”
- Processes (in OS/software systems)
  - Separate programs actually running (not sitting idle) on your computer at the same time.
  - Each process will have its own virtual memory space and you need explicitly exchange data using inter-process communication APIs
- Threads (in OS/software systems)
  - Independent portions of your program that can run in parallel
  - All threads share the same virtual memory space
- We will refer to these collectively as “threads”
  - A typical user system might have 1-8 actively running threads.
  - Servers can have more if needed (the sysadmins will hopefully configure it that way)
Cache on Multiprocessor

• Coherency
  • Guarantees all processors see the same value for a variable/memory address in the system when the processors need the value at the same time
    • What value should be seen

• Consistency
  • All threads see the change of data in the same order
    • When the memory operation should be done
What happens when core 0 modifies 0x1000?, which belongs to the same cache block as 0x1000?

Cache coherency practice
Cache coherency practice

- Then, what happens when core 2 reads 0x1000?
Cache coherency practice

- Now, what happens when core 2 writes 0x1004, which belongs the same block as 0x1000?

- Then, if Core 0 accesses 0x1000, it will be a miss!

```
Core 0  Core 1  Core 2  Core 3
Shared 0x1000  |  Invalid 0x1000  |  Shared 0x1000  |  Invalid 0x1000

Local $  Invalidation path:

Invalid 0x1000  |  Invalid 0x1000  |  Invalid 0x1000

Bus

Shared $  Write miss 0x1004
```

Invalidate all 0x1000 because 0x1000 and 0x1004 belong to the same cache block!
4C model

- **3Cs:**
  - Compulsory, Conflict, Capacity
- **Coherency miss:**
  - A “block” invalidated because of the sharing among processors.
    - True sharing
      - Processor A modifies X, processor B also want to access X.
    - False Sharing
      - Processor A modifies X, processor B also want to access Y. However, Y is invalidated because X and Y are in the same block!
Sample questions
Format of finals

• Multiple choices * 16
  • They’re like your clicker/midterm multiple choices questions
  • Accumulative, don’t forget your midterm and midterm review
• Homework style calculation/operation based questions
• Brief discussion
  • Explain your answer using less than 100 words
  • May not have a standard answer. You need to understand the concepts to provide a good answer
How many of the following descriptions about pipelining is correct?

① You can always divide stages into short stages with latches
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
⑤ Pipelining a stage can always improve cycle time

A. 1
B. 2
C. 3
D. 4
E. 5
Where in our code has locality?

• Which description about locality of arrays sum and A in the following code is the most accurate?
  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
• 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
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
CMP advantages

• How many of the following are advantages of CMP over traditional superscalar processor
  ① CMP can provide better energy-efficiency within the same area
  ② CMP can deliver better instruction throughput within the same die area (chip size)
  ③ CMP can achieve better ILP for each running thread
  ④ CMP can improve the performance of a single-threaded application without modifying code

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
Other concepts that are important

- When do we need to stall the pipeline?
- Why do we need branch prediction?
- What’s the impact of deep pipeline?
- How cache exploit each type of locality
- Why we need tag, index, offset in cache?
- What are three Cs of cache misses? For each C, how can we improve?
- Why we need virtual memory? Pros & Cons?
- Why hierarchical page tables? Pros & Cons?
- What is TLB? Are TLB misses more expensive than L1 misses? Why?
- What’s coherence miss?
- What are reasons of performance differences behind the demos?
Single-issue pipeline

```assembly
LOOP: lw $t1, 0($a0)
       add $v0, $v0, $t1
       addi $a0, $a0, 4
       bne $a0, $t0, LOOP
       lw $t0, 0($sp)
       lw $t1, 4($sp)
```

If the current value of $a0 is **0x10000000** and $t0 is **0x10001000**, what are the dynamic instructions that the processor will execute?

Can you draw the pipeline diagram?
The processor has a 8KB, 256B blocked, 2-way L1 cache. Consider the following code:

```c
for(i=0;i<256;i++) {
    a[i] = b[i] + c[i];
    // load b[i] and load a[i], store to a[i]
    // &a[0] = 0x10000, &b[0] = 0x20000, &c[0] = 0x30000
}
```

What’s the total miss rate? How many of the misses are compulsory misses? How many of the misses are conflict misses?

How can you improve the cache performance of the above code through changing hardware?

How can you improve the performance without changing hardware?
Performance evaluation with cache

- Consider the following cache configuration on a MIPS 5-stage MIPS processor:

<table>
<thead>
<tr>
<th></th>
<th>I-L1</th>
<th>D-L1</th>
<th>L2</th>
<th>DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>32K</td>
<td>32K</td>
<td>256K</td>
<td>Big enough</td>
</tr>
<tr>
<td>block size</td>
<td>64 Bytes</td>
<td>64 Bytes</td>
<td>64 Bytes</td>
<td>4KB pages</td>
</tr>
<tr>
<td>associativity</td>
<td>2-way</td>
<td>2-way</td>
<td>8-way</td>
<td></td>
</tr>
<tr>
<td>access time</td>
<td>1 cycle (no penalty if it's a hit)</td>
<td>1 cycle (no penalty if it's a hit)</td>
<td>10 cycles</td>
<td>100 cycles</td>
</tr>
<tr>
<td>local miss rate</td>
<td>2%</td>
<td>10%, 20% dirty</td>
<td>15% (i.e., 15% of L1 misses, also miss in the L2), 30% dirty</td>
<td></td>
</tr>
<tr>
<td>Write policy</td>
<td>N/A</td>
<td>Write-back, write allocate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replacement</td>
<td>LRU replacement policy</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The application has 20% branches, 10% loads/stores, 70% integer instructions. Assume that TLB miss rate is 2% and it requires 100 cycles to handle a TLB miss. Also assume that the branch predictor has a hit rate of 87.5%, what’s the CPI of branch, L/S, and integer instructions? What is the average CPI?
Dynamic scheduling

- Please identify the RAW, WAW, WAR dependencies for the following code

```
lw   $t1, 0($a0)
add  $t0, $t0, $t1
addi $a0, $a0, 4
lw   $t1, 0($a0)
add  $t0, $t0, $t1
addi $a0, $a0, 4
```

- Please draw the data dependency graph for the above code

- If we can eliminate the false dependencies, and can issue up to 2 instructions each cycle with memory accesses taking 3 cycles, how many cycles it takes to empty the instruction queue?
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?
Dynamic branch prediction

- Consider the following code, which branch predictor (always-taken, 2-bit local, 2-bit global history with 4-bit GHR) works the best?

```c
int i = 0, a = 0, b = 0;
do {
    if((i%4)==0) // branch X: i%4 != 0 means taken
    {
        a = i;
        if((i%8)==0) // branch Y: i%8 != 0 means taken
            b = i;
    }
    i++;
} while(i<INT_MAX); // i < INT_MAX means taken
```
Other open-ended brief discussion

- 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?
- If we have hardware dynamic scheduling, do we still need compiler optimizations?
- What are false dependencies? How can we remove false dependencies?
- If the OoO pipeline is highly optimized, do we still care about the ISA design?
Announcement

- CAPE (Course Evaluation) / TA Evaluation
  - Will drop one more homework grade if the response rate is more than 70%
- Hung-Wei’s last office hour — 3p — 6pm @ EBU3B B270
- Final exam — Saturday @ PCYNH 120
  - 8a-11a
  - Bring a calculator
  - Get good grades
  - Then .. eat pizza!
- Will have a small award session (should be less than 30 minutes) after our final
- Bonus point — if you complete both the pre-class survey and end-of-class survey — there is a link on piazza