Pipelining

The processor can complete 1 instruction each cycle

CPI == 1 if everything works perfectly!
Recap: Pipeline hazards

- Even though we perfectly divide pipeline stages, it’s still hard to achieve CPI \( \geq 1 \).

- Pipeline hazards:
  - Structural hazard
    - The hardware does not allow two pipeline stages to work concurrently
  - Data hazard
    - A later instruction in a pipeline stage depends on the outcome of an earlier instruction in the pipeline
  - Control hazard
    - The processor is not clear about what’s the next instruction to fetch
Recap: solutions for hazards

• Structural hazards
  • Stall
  • Modify the hardware design to support concurrent accesses

• Data Hazards
  • Stall
  • Data forwarding
  • Code optimization — limited

• Control Hazards
  • Stall
  • Branch prediction
    • Static prediction
    • Dynamic prediction
      • Local
      • Global history
Dynamic branch prediction

- A 2-bit counter for each branch
- Predict taken if the counter value >= 2
- If the prediction in taken states, fetch from target PC, otherwise, use PC+4
  - If we guess right — no penalty
  - If we guess wrong — flush (clear pipeline registers) for mis-predicted instructions that are currently in IF and ID stages and reset the PC

<table>
<thead>
<tr>
<th>Branch Address</th>
<th>Target Address</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400420</td>
<td>0x8048324</td>
<td>11</td>
</tr>
<tr>
<td>0x400464</td>
<td>0x8048392</td>
<td>10</td>
</tr>
<tr>
<td>0x400578</td>
<td>0x804850a</td>
<td>00</td>
</tr>
<tr>
<td>0x41000C</td>
<td>0x8049624</td>
<td>01</td>
</tr>
</tbody>
</table>

PC = 0x400420

Taken!
Recap: Local 2-bit predictor

```c
i = 0;
do {
    if (i % 2 != 0) // Branch X, taken if i % 2 == 0
        a[i] *= 2;
    a[i] += i;
} while (++i < 100) // Branch Y
```

<table>
<thead>
<tr>
<th>i</th>
<th>branch?</th>
<th>state</th>
<th>prediction</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>11</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>11</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

For branch Y, almost 100%.
For branch X, only 50%.
2-level global predictor

- Instead of using the PC to choose the predictor, use a bit vector (global history register, GHR) made up of the previous branch outcomes.
- Global predictor: predictor using results from all branches
- Local predictor: predictor tracking states/history for each branch
- Each entry in the history table has its own counter.

First level

3-bit GHR = 101 (T, NT, T)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

2nd level 2^3 entries

Pentium Pro uses this predictor
Performance of the 2-bit global predictor

```
i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100) // Branch Y
```

Nearly perfect after this branch?
Demo revisited

• Why the sorting the array speed up the code despite the increased instruction count?

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

for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
        if (data[i] >= threshold)
            sum ++;
    }
}
```
Branch performance

- 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
B. 1
C. 2
D. 3
E. 4

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

for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
        if (data[i] >= threshold)
            sum ++;
    }
}
```

<table>
<thead>
<tr>
<th></th>
<th>Without sorting</th>
<th>With sorting</th>
</tr>
</thead>
<tbody>
<tr>
<td>The prediction accuracy of X before threshold</td>
<td>50%</td>
<td>100%</td>
</tr>
<tr>
<td>The prediction accuracy of X after threshold</td>
<td>50%</td>
<td>100%</td>
</tr>
</tbody>
</table>
Demo: popcount

- How many 1s in binary representations
- Applications
  - Hamming weight
  - Encryption/decryption

```c
int main(int argc, char *argv[]) {
    uint64_t key = 0xdeadbeef;
    int count = 1000000000;
    uint64_t sum = 0;
    for (int i=0; i < count; i++)
    {
        sum += popcount (RandLFSR(key));
    }
    printf("Result: %lu\n", sum);
    return sum;
}
```
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;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        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, 2, 3, 1, 2, 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, 3, 3, 4};
    for (uint64_t i = 0; i < 16; i++) {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```
The last why about branch & your code
Why is D better than C

• How many of the following statements explains the main reason why B outperforms C with compiler optimizations

  ① D has lower dynamic instruction count than C
  ② D has significantly lower branch mis-predictions than C
  ③ D has significantly fewer branch instructions than C
  ④ D can incur fewer data hazards than C

A. 0
B. 1
C. 2
D. 3
E. 4

— potentially could be more, but aggressive compiler can do loop unrolling
— Could be
— maybe eliminated through loop unrolling...
— about the same

you know exactly the number of iterations — compiler can do loop unrolling!
Loop unrolling eliminates all branches!

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

- Because popcount is important, both Intel and AMD added a POPCNT instruction in their processors with SSE4.2 and SSE4a.
- In C/C++, you may use the intrinsic “_mm_popcnt_u64” to get # of “1”s in an unsigned 64-bit number.
  - You need to compile the program with -m64 -msse4.2 flags to enable these new features.

```c
#include <smmintrin.h>
inline int popcount(uint64_t x) {
    int c = _mm_popcnt_u64(x);
    return c;
}
```
von Neumann architecture

Processor
PC

memory

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 ldi t0,-23508(t1)
120007a44: 130020e4 beq t0,120007a94
120007a48: 00003d24 ldah t0,0(gp)
120007a4c: 2ca4e2b3 stl zero,-23508(t1)
120007a50: 0004ff47 clr v0
120007a54: 28a4e5b3 stl zero,-23512(t4)
120007a58: 20a421a4 ldq t0,-23520(t0)
120007a5c: 0e0020e4 beq t0,120007a98
120007a60: 0204e147 mov t0,t1
120007a64: 0304ff47 clr t2
120007a68: 0500e0c3 br 120007a80
Outline

- Memory wall/gap problem
- Memory hierarchy
- Cache organization
Memory wall problem
The memory gap problem

CPU

DRAM-based main memory

lw $t2, 0($a0)
add $t3, $t2, $a1
addi $a0, $a0, 4
subi $a1, $a1, 1
bne $a1, LOOP
lw $t2, 0($a0)
add $t3, $t2, $a1

The access time of DRAM is around 50ns
100x to the cycle time of a 2GHz processor!

SRAM is as fast as the processor, but

$500–$1000
$10–$20
$0.75–$1.00
$0.05–$0.10

The access time of DRAM is around 50ns
100x to the cycle time of a 2GHz processor!
Memory’s impact

• Assume your processor takes only 1 cycle to process an instruction if the DRAM is the same speed as the processor. In fact, the latency of DRAM is 100 cycles.
  • Ignore all virtual memory overheads and processor optimizations
  • The application contains 20% instructions that perform data memory accesses
• How many cycles do you need to process an instruction on average?
  A. ~ 10
  B. ~ 20
  C. ~ 100
  D. ~ 120
  E. ~ 200

average = 1 + \( \frac{1}{100} \times 100 \) + \( \frac{0.2}{100} \times 100 \) = 121
Why is C better than B

- How many of the following statements explains the reason why B outperforms C with compiler optimizations
  1. C has lower dynamic instruction count than B
  2. C has significantly lower branch mis-predictions than B
  3. C has significantly fewer branch instructions than B
  4. C can incur fewer data hazards

A. 0  
B. 1  
C. 2  
D. 3  
E. 4

Does this make sense if memory is so slow?
Memory hierarchy
The memory hierarchy

- **CPU**
- **Main Memory**
- **Secondary Storage**

**Access time**:
- **< 1ns**
- **< 1ns ~ 20 ns**
- **100ns**
- **10,000,000ns**

**Cache**
- L1: 16KB-64KB
- L2: 128KB-512KB
- L3: Several MBs

**Fastest, Most Expensive**
- 32* 64-bit registers
- Several GBs
- 500+ GB

**Biggest**
By how much can cache help?

- Assume your processor takes only 1 cycle to process an instruction if the DRAM is the same speed as the processor. In fact, the latency of DRAM is 100 cycles.
  - Now, if we add a cache between CPU and DRAM. If 90% of time, the data/instruction can be found in the cache — no additional cycles is wasted to fetch the data/instruction.
  - If the data is missing in the cache, it takes 100 cycles to retrieve data from the DRAM.
  - You may ignore all virtual memory overheads and processor optimizations.
  - The application contains 20% instructions that perform data memory accesses.
  - How many cycles do you need to process an instruction on average?

A. ~ 10
B. ~ 20
C. ~ 100
D. ~ 120
E. ~ 200


calculate

\[
\text{average} = 1 + (1 \times 100 + 0.2 \times 100) \times (1 - 90\%) = 13
\]
Why can a small, fast SRAM help?
Localities in your code

- 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
Localities in your code

• Spatial locality: programs tend to access neighboring data/instructions
  • Data structures (e.g. arrays) demonstrate strong spatial locality
  • Especially effective for code/instructions — you usually just move to the next instruction or loop back to the small piece of code

• Temporal locality: programs tend to have frequently accessed data
  • You may update/reference the same set of memory locations many times in your code
Cache organization
Architecting caches to capture localities

• To capture spatial locality
  • We need to put not only just a “word” or small piece of data/instructions, but a “block” of data/instructions

• To capture temporal locality
  • We need to keep frequently used data
Organizing memory locations into blocks

Processor

PC

A
B
C
D
E

page
block

0x0
0x1000
0x2000
0x3000
0x4000
0x5000
0x6000
0x7000
0x8000

0x1000
0x2000
0x3000
0x4000
0x5000
0x6000
0x7000
0x8000
0xFFF
0x1FFF
0x2FFF
0x3FFF
0x4FFF
0x5FFF
0x6FFF
0x7FFF
0xFFFF
0x1FFFFF
0x2FFFFF
0x3FFFFF
0x4FFFFF
0x5FFFFF
0x6FFFFF
0x7FFFFF
0x8FFFFF

A
B
C
D
E
Architecting caches to capture localities

- To capture spatial locality
  - We need to put not only just a “word” or small piece of data/instructions, but a “block” of data/instructions
  - How to distinguish each block?
- To capture temporal locality
  - We need to keep frequently used data
How do you make a cheatsheet?

- Go through your homework
- Write down the topic and content
- If running out of space: kick out the least recently used content

1. Performance equation
2. Amdahl’s law
3. MIPS
4. Power consumption
5. Performance equation 😊
6. Amdahl’s law 😊
7. MFLOPS

Tag: the address prefix of data in the cacheline/block

<table>
<thead>
<tr>
<th>Performance equation</th>
<th>ET=IC<em>CPI</em>CT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amdahl’s law</td>
<td>ET_after = ET_affected/Speedup + ET_unaffected</td>
</tr>
<tr>
<td>MFLOPS</td>
<td>MIPS = IC*(ET*10^6)</td>
</tr>
<tr>
<td>Power consumption</td>
<td>P = aCV^2f</td>
</tr>
</tbody>
</table>

Cacheline/block: data with the same prefix in their addresses
A simple cache: now with tags associated with blocks

- Assume each block contains 16B data
- A total of 4 blocks

<table>
<thead>
<tr>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>content of 0b00000000 - 0b00001111</td>
</tr>
<tr>
<td>0b0100</td>
<td>content of 0b01000000 - 0b01001111</td>
</tr>
<tr>
<td>0b1100</td>
<td>content of 0b11000000 - 0b11001111</td>
</tr>
<tr>
<td>0b1111</td>
<td>content of 0b11110000 - 0b11111111</td>
</tr>
</tbody>
</table>
Architecting caches to capture localities

• To capture spatial locality
  • We need to put not only just a “word” or small piece of data/instructions, but a “block” of data/instructions
  • A tag associated with each block

• To capture temporal locality
  • A cache replacement policy to keep most frequently used data (e.g. LRU)
  • LRU — kick out the least recently used block when we need to kick out one
A simple cache: a block can go anywhere

- Assume each block contains 16B data
- A total of 4 blocks
- LRU — kick out the least recently used whenever we need to

<table>
<thead>
<tr>
<th></th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x4</td>
<td>0b00000100</td>
</tr>
<tr>
<td>2</td>
<td>0x48</td>
<td>0b01001000</td>
</tr>
<tr>
<td>3</td>
<td>0xC4</td>
<td>0b11000100</td>
</tr>
<tr>
<td>4</td>
<td>0xFC</td>
<td>0b11111100</td>
</tr>
<tr>
<td>5</td>
<td>0x12</td>
<td>0b00001100</td>
</tr>
<tr>
<td>6</td>
<td>0x44</td>
<td>0b01000100</td>
</tr>
<tr>
<td>7</td>
<td>0x68</td>
<td>0b01100100</td>
</tr>
</tbody>
</table>

- Too slow if the number of entries/blocks/cachelines is huge
Architecting caches to capture localities

• To capture spatial locality
  • We need to put not only just a “word” or small piece of data/instructions, but a “block” of data/instructions
  • A tag associated with each block

• To capture temporal locality
  • A cache replacement policy to keep most frequently used data (e.g. LRU)
  • LRU — kick out the least recently used block when we need to kick out one

• Performance needs to be better than linear search
  • Make cache a hardware hash table!
  • The hash function takes memory addresses as inputs
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**

**data**

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

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

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

0x8 0 0 0 0 1 5 8

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
How many bits in each field?

- \( \text{lg(number of sets)} \)
- \( \text{lg(block size)} \)

- valid
- dirty
- tag
- index
- offset
- data

hit?

valid
dirty
=?

hit?
C = ABS

- C: Capacity in data arrays
- A: Way-Associativity
  - 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
Corollary of $C = \text{ABS}$

- offset bits: $\lg(B)$
- index bits: $\lg(S)$
- tag bits: $\text{address}_\text{length} - \lg(S) - \lg(B)$
  - $\text{address}_\text{length}$ is 32 bits for 32-bit machine
- $(\text{address} / \text{block}_\text{size}) \% S = \text{set index}$
AMD Phenom II

• 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

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