Simple CPU

CPU
- condition codes
- program counter
- registers

Memory
- code
- data
- stack

Arrows:
- addresses
- data
- instructions
Simple CPU

CPU

- condition codes
- program counter
- registers

Memory

- code
- data
- stack

Arrows indicate:
- addresses
- data
- instructions
Modern CPU

Instruction Control

Register Update

Prediction OK?

Execution

Operation Results

Branch
Arith
Arith
Arith
Load
Store

Functional Units

Fetch Control
Instruction Decode

Instruction Cache

Address

Operations

Instructions

Retirement Unit
Register File

Data Cache

Addr.

Data

Definitions

**Superscalar processor**

- Issue and execute multiple instructions within a cycle
- Instructions are determined dynamically

**Instruction-level parallelism**

- Some instructions in a program can execute at once
- No explicit declaration needed
Pipelined Functional Units

```
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1*p2;
    return p3;
}
```

Multiplication takes 3 cycles, so... 9 cycles minimum?

No, because

- one new multiplication can start every cycle
- \(a*b\) and \(a*c\) are independent calculations
Pipelined Functional Units

```c
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1*p2;
    return p3;
}
```

![Diagram showing the pipelined execution of the `mult_eg` function](image)

# Instruction Performance

## Haswell:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>latency</th>
<th>cycles/issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>store</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>integer add</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>integer multiply</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>integer divide</td>
<td>3-30</td>
<td>3-30</td>
</tr>
<tr>
<td>FP add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP multiply</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>FP divide</td>
<td>3-15</td>
<td>3-15</td>
</tr>
</tbody>
</table>

## Instruction Performance

### Haswell:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Store</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Integer add</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Integer multiply</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer divide</td>
<td>3-30</td>
<td>3-30</td>
</tr>
<tr>
<td>FP add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP multiply</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>FP divide</td>
<td>3-15</td>
<td>3-15</td>
</tr>
</tbody>
</table>
### Instruction Performance

How long to wait before starting a new one

<table>
<thead>
<tr>
<th>Operation</th>
<th>latency</th>
<th>cycles/issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>store</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>integer add</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>integer multiply</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>integer divide</td>
<td>3-30</td>
<td>3-30</td>
</tr>
<tr>
<td>FP add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP multiply</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>FP divide</td>
<td>3-15</td>
<td>3-15</td>
</tr>
</tbody>
</table>

void combine4(vec_ptr v, data_t *dest) {
    long int i;
    int length = vec_length(v);
    data_t* data = get_vec_start(v);
    data_t acc = IDENT;

    for (i = 0; i < length; i++)
        acc = acc OPER data[i];
    *dest = acc;
}
CPE for Benchmark

void combine4(vec_ptr v, data_t *dest) {
    long int i;
    int length = vec_length(v);
    data_t* data = get_vec_start(v);
    data_t acc = IDENT;
    for (i = 0; i < length; i++)
        acc = acc OPER data[i];
    *dest = acc;
}

.L519:
    imull (%rax,%rdx,4), %ecx # t = t * d[i]
    addq $1, %rdx              # i++
    cmpq %rdx, %rbp            # Compare length:i
    jg    .L519                # If >, loop

for (i = 0; i < length; i++)
    acc = acc OPER data[i];
*dest = acc;
}

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>*</td>
<td>3.01</td>
<td>5.01</td>
</tr>
<tr>
<td>accumulate to local</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>latency bound</td>
<td>3.00</td>
<td>5.00</td>
</tr>
</tbody>
</table>

for (i = 0; i < length; i++)
    acc = acc OPER d[i];
Loop Unrolling

```c
void combine5(vec_ptr v, data_t *dest) {
    long int i;
    int length = vec_length(v);
    int limit = length - 1;
    data_t* data = get_vec_start(v);
    data_t acc = IDENT;

    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i += 2)
        acc = (acc OPER data[i]) OPER data[i+1];

    /* Finish any remaining elements */
    for (; i < length; i++)
        acc = acc OPER data[i];
    *dest = acc;
}
```
void combine5(vec_ptr v, data_t *dest) {
    long int i;
    int length = vec_length(v);
    int limit = length - 1;
    data_t* data = get_vec_start(v);
    data_t acc = IDENT;

    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i += 2)
        acc = (acc OPER data[i]) OPER data[i+1];

    /* Finish any remaining elements */
    for (; i < length; i++)
        acc = acc OPER data[i];

    *dest = acc;
}
for (i = 0; i < limit; i += 2)
    acc = (acc OPER d[i]) OPER d[i+1];
Why Addition Improves, Anyway

Original:
Why Addition Improves, Anyway

Unrolled:
Why Addition Improves, Anyway

Original multiplication:
Reassociation

```c
for (i = 0; i < limit; i += 2)
    acc = (acc OPER d[i]) OPER d[i+1];
```

```c
for (i = 0; i < limit; i += 2)
    acc = acc OPER (d[i] OPER d[i+1]);
```

**Always the same result?**

Not for floating-point, but probably good enough

<table>
<thead>
<tr>
<th>int</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>2x unroll</td>
<td>1.01</td>
</tr>
<tr>
<td>2x + reassoc</td>
<td>1.01</td>
</tr>
<tr>
<td>latency bound</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Reassociated Computation

* for double: 5 cycles
per 2 elements ⇒ 2.5 CPE
Reassociated Computation

**Diagram:**

1. **Integers**
   - 2x unroll: 1.01 + 3.01 = 3.01
   - 2x + reassoc: 1.01 + 1.51 = 2.51
   - latency bound: 1.00 + 3.00 = 3.00

2. **Doubles**
   - 2x unroll: 3.01 + 5.01 = 8.01
   - 2x + reassoc: 3.01 + 2.51 = 5.51
   - latency bound: 3.00 + 5.00 = 8.00
Another Reassociation
Another Reassociation

“2x2” = 2x unrolling with 2 accumulators
Unrolling with Multiple Accumulators

```c
void combine6(vec_ptr v, data_t *dest) {
    long int i;
    int length = vec_length(v);
    int limit = length - 1;
    data_t* data = get_vec_start(v);
    data_t acc0 = IDENT, acc1 = IDENT;

    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i += 2) {
        acc0 = acc0 OPER data[i];
        acc1 = acc1 OPER data[i+1];
    }

    /* Finish any remaining elements */
    for (; i < length; i++)
        acc0 = acc0 OPER data[i];
    *dest = acc0 OPER acc1;
}
```
## Keep Unrolling

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th></th>
<th>double</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>*</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>2x + reassoc</td>
<td>1.01</td>
<td>1.51</td>
<td>1.51</td>
<td>2.51</td>
</tr>
<tr>
<td>2x2 unroll</td>
<td>0.81</td>
<td>1.51</td>
<td>1.51</td>
<td>2.51</td>
</tr>
<tr>
<td>4x4 unroll</td>
<td>0.72</td>
<td>1.07</td>
<td>1.01</td>
<td>1.25</td>
</tr>
<tr>
<td>latency bound</td>
<td>1.00</td>
<td>3.00</td>
<td>3.00</td>
<td>5.00</td>
</tr>
</tbody>
</table>

Why aren’t all 4x4 times half of 2x2 times?
Parallelism by Pipeline

\[
\text{latency} \quad \text{cycles/issue}
\]

integer multiply \hspace{1cm} 3 \hspace{1cm} 1

\[
t = 1 \quad t = 2 \quad t = 3 \quad t = 4 \quad t = 5 \quad t = 6
\]

Stage 1 \hspace{0.5cm} a*b \hspace{0.5cm} c*d \hspace{0.5cm} e*f \hspace{0.5cm} g*h

Stage 2 \hspace{1cm} a*b \hspace{0.5cm} c*d \hspace{0.5cm} c*d \hspace{0.5cm} g*h

Stage 3 \hspace{1cm} a*b \hspace{0.5cm} c*d \hspace{0.5cm} c*d \hspace{0.5cm} g*h
Parallelism by Pipeline × Capacity

\[
\text{latency} \quad \text{cycles/issue} \quad \text{capacity}
\]

integer multiply  \hspace{1cm} 3 \hspace{1cm} 1 \hspace{1cm} 2

\[
t = 1 \quad t = 2 \quad t = 3 \quad t = 4 \quad t = 5
\]

Stage 1  \hspace{1cm} a*b \hspace{0.5cm} c*d \hspace{0.5cm} e*f

Stage 2  \hspace{1cm} a*b \hspace{0.5cm} c*d \hspace{0.5cm} c*d

Stage 3  \hspace{1cm} a*b \hspace{0.5cm} c*d \hspace{0.5cm} c*d

Stage 1  \hspace{1cm} g*h

Stage 2  \hspace{1cm} g*h

Stage 3  \hspace{1cm} g*h
# Throughput Bound

<table>
<thead>
<tr>
<th></th>
<th>latency</th>
<th>cycles/issue</th>
<th>capacity</th>
<th>throughput bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>store</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>integer add</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>0.25</td>
</tr>
<tr>
<td>integer multiply</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>integer divide</td>
<td>3-30</td>
<td>3-30</td>
<td>1</td>
<td>3-30</td>
</tr>
<tr>
<td>FP add</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FP multiply</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>FP divide</td>
<td>3-15</td>
<td>3-15</td>
<td>1</td>
<td>3-15</td>
</tr>
<tr>
<td></td>
<td>latency</td>
<td>cycles/issue</td>
<td>capacity</td>
<td>bound</td>
</tr>
<tr>
<td>----------------</td>
<td>---------</td>
<td>--------------</td>
<td>----------</td>
<td>-------</td>
</tr>
<tr>
<td>load</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>store</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>integer add</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>0.25</td>
</tr>
<tr>
<td>integer multiply</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>integer divide</td>
<td>3-30</td>
<td>3-30</td>
<td>1</td>
<td>3-30</td>
</tr>
<tr>
<td>FP add</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FP multiply</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>FP divide</td>
<td>3-15</td>
<td>3-15</td>
<td>1</td>
<td>3-15</td>
</tr>
</tbody>
</table>

**combine**

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0.50</td>
<td>1.00</td>
</tr>
<tr>
<td>*</td>
<td>1.00</td>
<td>0.50</td>
</tr>
</tbody>
</table>

**throughput bound**

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0.50</td>
<td>1.00</td>
</tr>
<tr>
<td>*</td>
<td>1.00</td>
<td>0.50</td>
</tr>
</tbody>
</table>
## Throughput Bound

<table>
<thead>
<tr>
<th></th>
<th>latency</th>
<th>cycles/issue</th>
<th>capacity</th>
<th>throughput bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>4.0</td>
<td>1.0</td>
<td>2.0</td>
<td>0.5</td>
</tr>
<tr>
<td>store</td>
<td>4.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>integer add</td>
<td>1.0</td>
<td>1.0</td>
<td>4.0</td>
<td>0.25</td>
</tr>
<tr>
<td>integer multiply</td>
<td>3.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>integer divide</td>
<td>3-30</td>
<td>3-30</td>
<td>1.0</td>
<td>3-30</td>
</tr>
<tr>
<td>FP add</td>
<td>3.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>FP multiply</td>
<td>5.0</td>
<td>1.0</td>
<td>2.0</td>
<td>0.5</td>
</tr>
<tr>
<td>FP divide</td>
<td>3-15</td>
<td>3-15</td>
<td>1.0</td>
<td>3-15</td>
</tr>
</tbody>
</table>

**Limited by load instead of +**

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>throughput bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Keep Unrolling?

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2x + reassoc</td>
<td>1.01</td>
<td>1.51</td>
</tr>
<tr>
<td>2x2 unroll</td>
<td>0.81</td>
<td>1.51</td>
</tr>
<tr>
<td>4x4 unroll</td>
<td>0.72</td>
<td>1.01</td>
</tr>
<tr>
<td>latency bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>throughput bound</td>
<td>0.50</td>
<td>1.00</td>
</tr>
</tbody>
</table>

More unrolling:

+ more parallelism
- more register pressure
# Keep Unrolling?

## Floating-point *

<table>
<thead>
<tr>
<th>accumulators</th>
<th>1x</th>
<th>2x</th>
<th>3x</th>
<th>4x</th>
<th>6x</th>
<th>8x</th>
<th>10x</th>
<th>12x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
<td>5.01</td>
</tr>
<tr>
<td>2</td>
<td>2.51</td>
<td></td>
<td>2.51</td>
<td></td>
<td>2.51</td>
<td></td>
<td>2.51</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1.67</td>
<td></td>
<td>1.25</td>
<td></td>
<td>1.26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>0.84</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.63</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.51</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.52</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Keep Unrolling?

Integer +

<table>
<thead>
<tr>
<th>accumulators</th>
<th>1x</th>
<th>2x</th>
<th>3x</th>
<th>4x</th>
<th>6x</th>
<th>8x</th>
<th>10x</th>
<th>12x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.27</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
</tr>
<tr>
<td>2</td>
<td>0.81</td>
<td>0.69</td>
<td>0.54</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>0.74</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>0.69</td>
<td>1.24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>0.56</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.54</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.56</td>
<td></td>
</tr>
</tbody>
</table>

unrolling factor

Summary for Unrolling & Reassociation

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th></th>
<th>double</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>*</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>2x unroll</td>
<td>1.01</td>
<td>3.01</td>
<td>3.01</td>
<td>5.01</td>
</tr>
<tr>
<td>2x + reassoc</td>
<td>1.01</td>
<td>1.51</td>
<td>1.51</td>
<td>2.51</td>
</tr>
<tr>
<td>2x2 unroll</td>
<td>0.81</td>
<td>1.51</td>
<td>1.51</td>
<td>2.51</td>
</tr>
<tr>
<td>4x4 unroll</td>
<td>0.72</td>
<td>1.07</td>
<td>1.01</td>
<td>1.25</td>
</tr>
<tr>
<td>10x10 unroll</td>
<td>0.55</td>
<td>1.00</td>
<td>1.01</td>
<td>0.52</td>
</tr>
<tr>
<td>latency bound</td>
<td>1.00</td>
<td>3.00</td>
<td>3.00</td>
<td>5.00</td>
</tr>
<tr>
<td>throughput bound</td>
<td>0.50</td>
<td>1.00</td>
<td>1.00</td>
<td>0.50</td>
</tr>
</tbody>
</table>
What About Branches?

```c
for (i = 0; i < limit; i += 2) {
    acc0 = acc0 + data[i];
    acc1 = acc1 + data[i+1];
}
```

CPE = 0.81
What About Branches?

```c
for (i = 0; i < limit; i += 2) {
    acc0 = acc0 + data[i];
    acc1 = acc1 + data[i+1];
}
```

```
.L3:
   addl (%rdi,%rax,4), %ecx
   addl 4(%rdi,%rax,4), %r8d
   addq $2, %rax
   cmpq %r9, %rax
   jl .L3
   jmp .L2
...
.L2:
   finish...
```
What About Branches?

```c
for (i = 0; i < limit; i += 2) {
    acc0 = acc0 + data[i];
    acc1 = acc1 + data[i+1];
}
```

```assembly
.L3:
    addl (%rdi,%rax,4), %ecx
    addl 4(%rdi,%rax,4), %r8d
    addq $2, %rax
    cmpq %r9, %rax
    jl .L3
    jmp .L2
...
.L2:
    finish...
```
What About Branches?

```c
for (i = 0; i < limit; i += 2) {
    acc0 = acc0 + data[i];
    acc1 = acc1 + data[i+1];
}
```

```assembly
.L3:
    addl (%rdi,%rax,4), %ecx
    addl 4(%rdi,%rax,4), %r8d
    addq $2, %rax
    cmpq %r9, %rax
    jl    .L3
    jmp   .L2
...
.L2:
    finish...
```

CPE = 0.81

at the same time

Might need to get an early start on the next iteration; can't just wait on a comparison!
Branch Prediction

*Branch prediction* is a guess about whether a jump will be taken or not

- Make a good guess by recording previous experience
- Perform work in parallel based on prediction
- Don’t expose that work (by writing to memory or to registers) until the branch is known
Branch Prediction

```
addl (%rdi,%rax,4), %ecx
addl 4(%rdi,%rax,4), %r8d
addq  $2, %rax
cmpq  %r9, %rax
jl   .L3    i = 0
```
Branch Prediction

\[
\begin{align*}
\text{addl} &\quad (%rdi, %rax, 4), \%ecx \\
\text{addl} &\quad 4(%rdi, %rax, 4), \%r8d \\
\text{addq} &\quad $2, \%rax \\
\text{cmpq} &\quad %r9, %rax \\
\text{jl} &\quad .L3 \quad i = 0
\end{align*}
\]

\[\text{branch taken}\]

\[
\begin{align*}
\text{addl} &\quad (%rdi, %rax, 4), \%ecx \\
\text{addl} &\quad 4(%rdi, %rax, 4), \%r8d \\
\text{addq} &\quad $2, \%rax \\
\text{cmpq} &\quad %r9, %rax \\
\text{jl} &\quad .L3 \quad i = 2
\end{align*}
\]
Prediction: branch is always taken
Branch Prediction

...for length 100...

```
addl (%rdi,%rax,4), %ecx
addl 4(%rdi,%rax,4), %r8d
addq $2, %rax
cmpq %r9, %rax
jl .L3 i = 96
```
Branch Prediction

...for length 100...

```
addl (%rdi,%rax,4), %ecx
addl 4(%rdi,%rax,4), %r8d
addq $2, %rax
cmpq %r9, %rax
jl .L3  i = 96
```

Assume branch taken

```
addl (%rdi,%rax,4), %ecx
addl 4(%rdi,%rax,4), %r8d
addq $2, %rax
cmpq %r9, %rax
jl .L3  i = 98
```
Branch Prediction

...for length 100...

```assembly
addl (%rdi,%rax,4), %ecx
addl 4(%rdi,%rax,4), %r8d
addq $2, %rax
cmpq %r9, %rax
jl .L3
```

$i = 96$

```assembly
addl (%rdi,%rax,4), %ecx
addl 4(%rdi,%rax,4), %r8d
addq $2, %rax
cmpq %r9, %rax
jl .L3
```

$i = 98$

```assembly
addl (%rdi,%rax,4), %ecx
addl 4(%rdi,%rax,4), %r8d
addq $2, %rax
cmpq %r9, %rax
jl .L3
```

$i = 100$

assume branch taken
(oops!)
Branch Prediction

...for length 100...

\[
\begin{align*}
\text{addl} & \quad (%\text{rdi},%\text{rax},4), \quad %\text{ecx} \\
\text{addl} & \quad 4(%\text{rdi},%\text{rax},4), \quad %\text{r8d} \\
\text{addq} & \quad 2, \quad %\text{rax} \\
\text{cmpq} & \quad %r9, \quad %\text{rax} \\
\text{jl} & \quad .L3 & i = 96
\end{align*}
\]

\[
\begin{align*}
\text{addl} & \quad (%\text{rdi},%\text{rax},4), \quad %\text{ecx} \\
\text{addl} & \quad 4(%\text{rdi},%\text{rax},4), \quad %\text{r8d} \\
\text{addq} & \quad 2, \quad %\text{rax} \\
\text{cmpq} & \quad %r9, \quad %\text{rax} \\
\text{jl} & \quad .L3 & i = 98
\end{align*}
\]

\[
\begin{align*}
\text{addl} & \quad (%\text{rdi},%\text{rax},4), \quad %\text{ecx} \\
\text{addl} & \quad 4(%\text{rdi},%\text{rax},4), \quad %\text{r8d} \\
\text{addq} & \quad 2, \quad %\text{rax} \\
\text{cmpq} & \quad %r9, \quad %\text{rax} \\
\text{jl} & \quad .L3 & i = 100
\end{align*}
\]

\[
\begin{align*}
\text{addl} & \quad (%\text{rdi},%\text{rax},4), \quad %\text{ecx} \\
\text{addl} & \quad 4(%\text{rdi},%\text{rax},4), \quad %\text{r8d} \\
\text{addq} & \quad 2, \quad %\text{rax} \\
\text{cmpq} & \quad %r9, \quad %\text{rax} \\
\text{jl} & \quad .L3 & i = 102
\end{align*}
\]

assume branch taken

assume branch taken
(oops!)

assume branch taken
(oops!)
Branch Prediction

...for length 100...

```
addl (%rdi,%rax,4), %ecx
addl 4(%rdi,%rax,4), %r8d
addq $2, %rax
cmpq %r9, %rax
jl .L3
```

i = 96

```
addl (%rdi,%rax,4), %ecx
addl 4(%rdi,%rax,4), %r8d
addq $2, %rax
cmpq %r9, %rax
jl .L3
```

i = 98

(assume branch taken)

branch not taken

```
addl (%rdi,%rax,4), %ecx
addl 4(%rdi,%rax,4), %r8d
addq $2, %rax
cmpq %r9, %rax
jl .L3
```

i = 100

(assume branch taken)

```
addl (%rdi,%rax,4), %ecx
addl 4(%rdi,%rax,4), %r8d
addq $2, %rax
cmpq %r9, %rax
jl .L3
```

i = 102

(assume branch taken)

(b)ooms!
Branch Prediction

...for length 100...

```
addl (%rdi,%rax,4), %ecx
addl 4(%rdi,%rax,4), %r8d
addq $2, %rax
cmpq %r9, %rax
jl .L3  i = 96
```

```
addl (%rdi,%rax,4), %ecx
addl 4(%rdi,%rax,4), %r8d
addq $2, %rax
cmpq %r9, %rax
jl .L3  i = 98
```

```
addl (%rdi,%rax,4), %ecx
addl 4(%rdi,%rax,4), %r8d
addq $2, %rax
cmpq %r9, %rax
jl .L3  i = 100
```

branch not taken

```
addl (%rdi,%rax,4), %ecx
addl 4(%rdi,%rax,4), %r8d
addq $2, %rax
cmpq %r9, %rax
jl .L3  i = 102
```

assume branch taken

assume branch taken (oops!)

assume branch taken (oops!)
Branch Prediction

...for length 100...

```assembly
addl  (%rdi,%rax,4), %ecx
addl  4(%rdi,%rax,4), %r8d
addq  $2, %rax
cmpq  %r9, %rax
jl    .L3
```

\[ i = 96 \]

assume branch taken

```assembly
addl  (%rdi,%rax,4), %ecx
addl  4(%rdi,%rax,4), %r8d
addq  $2, %rax
cmpq  %r9, %rax
jl    .L3
```

\[ i = 98 \]

assume branch taken
(oops!)

\[ i = 100 \]

branch not taken

spend cycles to reload pipeline

```
addl  (%rdi,%rax,4), %ecx
addl  4(%rdi,%rax,4), %r8d
addq  $2, %rax
cmpq  %r9, %rax
jl    .L3
```

\[ i = 102 \]

assume branch taken
(oops!)
Helping the Branch Predictor

Mostly, branch prediction “just works”

- At the assembly level: keep `call` and `ret` balanced
- At the C level: don’t create jumps that are random