CS 33

Architecture and Optimization (2)
Modern CPU Design

**Instruction Control**

- **Retirement Unit**
  - **Register File**

- **Fetch Control**
  - **Instruction Decode**

- **Instruction Cache**

**Execution**

- **Functional Units**
  - **Integer/Branch**
  - **General Integer**
  - **FP Add**
  - **FP Mul/Div**
  - **Load**
  - **Store**

- **Data Cache**

**Data**

**Addr.**

**Operation Results**

**Register Updates**

**Prediction OK?**
Superscalar Processor

• **Definition:** A superscalar processor can issue and execute *multiple instructions in one cycle*
  – instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically
    » instructions may be executed *out of order*

• **Benefit:** without programming effort, superscalar processors can take advantage of the *instruction-level parallelism* that most programs have

• Most CPUs since about 1998 are superscalar
• Intel: since Pentium Pro (1995)
Multiple Operations per Instruction

• `addq %rax, %rdx`
  – a single operation

• `addq %rax, 8(%rdx)`
  – three operations
    » load value from memory
    » add to it the contents of %rax
    » store result in memory
Instruction-Level Parallelism

• addq 8(%rax), %rax
  addq %rbx, %rdx
  – can be executed simultaneously: completely independent

• addq 8(%rax), %rbx
  addq %rbx, %rdx
  – can also be executed simultaneously, but some coordination is required
Out-of-Order Execution

- `movss (%rbp), %xmm0`
- `mulss (%rax, %rdx, 4), %xmm0`
- `movss %xmm0, (%rbp)`
- `addq %r8, %r9`
- `imulq %rcx, %r12`
- `addq $1, %rdx`

These can be executed without waiting for the first three to finish.
Speculative Execution

80489f3:  movl    $0x1,%ecx
80489f8:  xorq    %rdx,%rdx
80489fa:  cmpq    %rsi,%rdx
80489fc:  jnl     8048a25
80489fe:  movl    %esi,%edi
8048a00:  imull   (%rax,%rdx,4),%ecx

perhaps execute these instructions
Haswell CPU

• Functional Units
  1) Integer arithmetic, floating-point multiplication, integer and floating-point division, branches
  2) Integer arithmetic, floating-point addition, integer and floating-point multiplication
  3) Load, address computation
  4) Load, address computation
  5) Store
  6) Integer arithmetic
  7) Integer arithmetic, branches
  8) Store, address computation
Haswell CPU

- Instruction characteristics

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Add</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>3-30</td>
<td>3-30</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>3-15</td>
<td>3-15</td>
<td>1</td>
</tr>
<tr>
<td>Load</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Store</td>
<td>-</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
## Haswell CPU Performance Bounds

<table>
<thead>
<tr>
<th></th>
<th>Integer</th>
<th></th>
<th>Floating Point</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>*</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>Latency</td>
<td>1.00</td>
<td>3.00</td>
<td>3.00</td>
<td>5.00</td>
</tr>
<tr>
<td>Throughput</td>
<td>0.25</td>
<td>1.00</td>
<td>1.00</td>
<td>0.50</td>
</tr>
</tbody>
</table>
x86-64 Compilation of Combine4

- Inner loop (case: SP floating-point multiply)

```assembly
.L519:
mullss (%rax,%rdx,4), %xmm0  # t = t * d[i]
addq $1, %rdx                # i++
cmpq %rdx, %rbp             # Compare length:i
jg   .L519                  # If >, goto Loop
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.00</td>
</tr>
<tr>
<td>Latency bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>0.25</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Inner Loop

%rax %rbp %rdx %xmm0

load
mul
add
cmp
jg

mulss (%rax,%rdx,4), %xmm0
addq $1,%rdx
cmpq %rdx,%rbp
jg loop
Data-Flow Graphs of Inner Loop

\[ \text{load} \rightarrow \text{mul} \rightarrow \text{cmp} \rightarrow \text{jg} \]

\[ \text{load} \rightarrow \text{mul} \rightarrow \text{add} \]

\[ \text{data}[i] \rightarrow \text{load} \rightarrow \text{mul} \rightarrow \text{add} \]
Relative Execution Times

data[i] → load → mul → %xmm0

%rdx → add → %rdx

%xmm0 → %rdx

Data Flow Over Multiple Iterations

Critical path

```
data[0]  
  load  
  mul  add

data[1]  
  load  
  mul  add

...  

data[n-2]  
  load  
  mul  add

data[n-1]  
  load  
  mul  add
```
Pipelined Data-Flow Over Multiple Iterations
Pipelined Data-Flow Over Multiple Iterations

Diagram showing the flow of operations over multiple iterations, including load, multiply (mul), and add operations.
Pipelined Data-Flow Over Multiple Iterations
Combine4 = Serial Computation (OP = *)

- **Computation (length=8)**
  \[(((((1 \times d[0]) \times d[1]) \times d[2]) \times d[3]) \times d[4]) \times d[5]) \times d[6]) \times d[7]]

- **Sequential dependence**
  - performance: determined by latency of OP
Loop Unrolling

```c
void unroll2x(vec_ptr_t v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length - 1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration
Loop Unrolling

- Perform 2x more useful work per iteration

```c
void unroll2x(vec_ptr_t v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length - 1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Quiz 1

Does it speed things up by allowing more parallelism?

a) yes  
b) no
Effect of Loop Unrolling

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th>Double FP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.00</td>
<td>3.00</td>
<td>5.00</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>1.01</td>
<td>3.00</td>
<td>3.00</td>
<td>5.00</td>
</tr>
<tr>
<td>Latency bound</td>
<td>1.0</td>
<td>3.0</td>
<td>3.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>0.25</td>
<td>1.0</td>
<td>1.0</td>
<td>0.5</td>
</tr>
</tbody>
</table>

• Helps integer add
  – reduces loop overhead

• Others don’t improve. Why?
  – still sequential dependency

\[ x = (x \text{ OP } d[i]) \text{ OP } d[i+1]; \]
Loop Unrolling with Reassociation

```c
void unroll2xra(vec_ptr_t v, data_t *dest)
{
  int length = vec_length(v);
  int limit = length-1;
  data_t *d = get_vec_start(v);
  data_t x = IDENT;
  int i;
  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i+=2) {
    x = x OP (d[i] OP d[i+1]);
  }
  /* Finish any remaining elements */
  for (; i < length; i++) {
    x = x OP d[i];
  }
  *dest = x;
}
```

- Can this change the result of the computation?
- Yes, for FP. **Why?**
Reass ociated Computation

\[ x = x \text{ OP} (d[i] \text{ OP} d[i+1]); \]

- **What changed:**
  - ops in the next iteration can be started early (no dependency)

- **Overall Performance**
  - N elements, D cycles latency/op
  - should be \((N/2+1)*D\) cycles:
    \[ \text{CPE} = \frac{D}{2} \]
  - measured CPE slightly worse for integer addition (there are other things going on)
Effect of Reassociation

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.00</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>1.01</td>
<td>3.00</td>
</tr>
<tr>
<td>Unroll 2x, reassociate</td>
<td>1.01</td>
<td>1.51</td>
</tr>
<tr>
<td>Latency bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>.25</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- Nearly 2x speedup for int *, FP +, FP *
  - reason: breaks sequential dependency

\[ x = x \text{ OP (d[i] OP d[i+1])}; \]
Loop Unrolling with Separate Accumulators

```c
void unroll2xp2x(vec_ptr_t v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length - 1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- Different form of reassociation
Effect of Separate Accumulators

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.00</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>1.01</td>
<td>3.00</td>
</tr>
<tr>
<td>Unroll 2x, reassociate</td>
<td>1.01</td>
<td>1.51</td>
</tr>
<tr>
<td>Unroll 2x parallel 2x</td>
<td>.81</td>
<td>1.51</td>
</tr>
<tr>
<td>Latency bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>.25</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- 2x speedup (over unroll 2x) for int *, FP +, FP *
  - breaks sequential dependency in a “cleaner,” more obvious way

\[
x0 = x0 \text{ OP } d[i];
\]
\[
x1 = x1 \text{ OP } d[i+1];
\]
Separate Accumulators

- **What changed:**
  - two independent “streams” of operations

- **Overall Performance**
  - N elements, D cycles latency/op
  - should be (N/2+1)*D cycles:
    \[ CPE = \frac{D}{2} \]
  - Integer addition improved, but not yet at predicted value

**What Now?**
Quiz 2

With 3 accumulators there will be 3 independent streams of instructions; with 4 accumulators 4 independent streams of instructions, etc.
Thus with n accumulators we can have a speedup of $O(n)$, as long as n is no greater than the number of available registers.

a) true
b) false
Performance

- **K-way loop unrolling with K accumulators**
  - limited by number and throughput of functional units
## Achievable Performance

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th>Double FP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.0</td>
<td>3.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Achievable scalar</td>
<td>.52</td>
<td>1.01</td>
<td>1.01</td>
<td>.54</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>.25</td>
<td>1.00</td>
<td>1.00</td>
<td>.5</td>
</tr>
</tbody>
</table>
Using Vector Instructions

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th>Double FP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.0</td>
<td>3.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Achievable Scalar</td>
<td>.52</td>
<td>1.01</td>
<td>1.01</td>
<td>.54</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>.25</td>
<td>1.00</td>
<td>1.00</td>
<td>.5</td>
</tr>
<tr>
<td>Achievable Vector</td>
<td>.05</td>
<td>.24</td>
<td>.25</td>
<td>.16</td>
</tr>
<tr>
<td>Vector throughput bound</td>
<td>.06</td>
<td>.12</td>
<td>.25</td>
<td>.12</td>
</tr>
</tbody>
</table>

- Make use of SSE Instructions
  - parallel operations on multiple data elements
What About Branches?

• Challenge

  – *instruction control unit* must work well ahead of *execution unit* to generate enough operations to keep EU busy

```
80489f3:  movl $0x1,%ecx  
80489f8:  xorq %rdx,%rdx  
80489fa:  cmpq %rsi,%rdx  
80489fc:  jnl 8048a25  
80489fe:  movl %esi,%edi  
8048a00:  imull (%rax,%rdx,4),%ecx  
```

  – when it encounters conditional branch, cannot reliably determine where to continue fetching
Modern CPU Design

Instruction Control

Execution

Instruction Cache

Fetch Control

Instruction Decode

Retirement Unit

Register File

Address

Operations

Prediction OK?

Register Updates

Integer/Branch

General Integer

FP Add

FP Mult/Div

Load

Store

Functional Units

Operation Results

Addr.

Data

Data

Data

Addr.

Addr.
Branch Outcomes

- When encounter conditional branch, cannot determine where to continue fetching
  - branch taken: transfer control to branch target
  - branch not-taken: continue with next instruction in sequence
- Cannot resolve until outcome determined by branch/integer unit

```
80489f3: movl $0x1,%ecx
80489f8: xorq %rdx,%rdx
80489fa: cmpq %rsi,%rdx
80489fc: jnl 8048a25
80489fe: movl %esi,%esi
8048a00: imull (%rax,%rdx,4),%ecx

8048a25: cmpq %rdi,%rdx
8048a27: jl 8048a20
8048a29: movl 0xc(%rbp),%eax
8048a2c: leal 0xffffffff (%rbp),%esp
8048a2f: movl %ecx,(%rax)
```
Branch Prediction

• Idea
  – guess which way branch will go
  – begin executing instructions at predicted position
    » but don’t actually modify register or memory data

```assembly
80489f3:  movl  $0x1,%ecx
80489f8:  xorq  %edx,%edx
80489fa:  cmpq  %rsi,%rdx
80489fc:  jnl   8048a25
...
```

**Predict taken**

```assembly
8048a25:  cmpq  %rdi,%rdx
8048a27:  jl    8048a20
8048a29:  movl  0xc(%rbp),%eax
8048a2c:  leal  0xfffffffffe8(%rbp),%esp
8048a2f:  movl  %ecx,(%rax)
```

**Begin execution**
## Branch Prediction Through Loop

### Code Snippet

```assembly
80488b1:  movl (%rcx,%rdx,4),%eax
80488b4:  addl %eax,(%rdi)
80488b6:  incl %edx
80488b7:  cmpl %esi,%edx
80488b9:  jl 80488b1

Assume vector length = 100
```

### Branch Predictions

- **i = 98**
  - Predict taken (OK)

- **i = 99**
  - Predict taken (oops)

- **i = 100**
  - Read invalid location

- **i = 101**
  - Executed

- **Fetched**
### Branch Misprediction Invalidation

Assume vector length = 100

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>80488b1</td>
<td><code>movl (%rcx, %rdx, 4), %eax</code></td>
</tr>
<tr>
<td>80488b4</td>
<td><code>addl %eax, (%rdi)</code></td>
</tr>
<tr>
<td>80488b6</td>
<td><code>incl %edx</code></td>
</tr>
<tr>
<td>80488b7</td>
<td><code>cmpeq %esi, %edx</code></td>
</tr>
<tr>
<td>80488b9</td>
<td><code>jl 80488b1</code></td>
</tr>
</tbody>
</table>

\(i = 98\)

Predict taken (OK)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>80488b1</td>
<td><code>movl (%rcx, %rdx, 4), %eax</code></td>
</tr>
<tr>
<td>80488b4</td>
<td><code>addl %eax, (%rdi)</code></td>
</tr>
<tr>
<td>80488b6</td>
<td><code>incl %edx</code></td>
</tr>
<tr>
<td>80488b7</td>
<td><code>cmpeq %esi, %edx</code></td>
</tr>
<tr>
<td>80488b9</td>
<td><code>jl 80488b1</code></td>
</tr>
</tbody>
</table>

\(i = 99\)

Predict taken (oops)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>80488b1</td>
<td><code>movl (%rcx, %rdx, 4), %eax</code></td>
</tr>
<tr>
<td>80488b4</td>
<td><code>addl %eax, (%rdi)</code></td>
</tr>
<tr>
<td>80488b6</td>
<td><code>incl %edx</code></td>
</tr>
<tr>
<td>80488b7</td>
<td><code>cmpeq %esi, %edx</code></td>
</tr>
<tr>
<td>80488b9</td>
<td><code>jl 80488b1</code></td>
</tr>
</tbody>
</table>

\(i = 100\)

Invalidate

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>80488b1</td>
<td><code>movl (%rcx, %rdx, 4), %eax</code></td>
</tr>
<tr>
<td>80488b4</td>
<td><code>addl %eax, (%rdi)</code></td>
</tr>
<tr>
<td>80488b6</td>
<td><code>incl %edx</code></td>
</tr>
<tr>
<td>80488b7</td>
<td><code>cmpeq %esi, %edx</code></td>
</tr>
<tr>
<td>80488b9</td>
<td><code>jl 80488b1</code></td>
</tr>
</tbody>
</table>

\(i = 101\)
### Branch Misprediction Recovery

The code snippet illustrates a simple branch misprediction scenario:

```
80488b1:  movl (%rcx,%rdx,4),%eax  \[\text{\textit{i} = 99}\]  
80488b4:  addl %eax,(%rdi)        
80488b6:  incl %edx              
80488b7:  cmp %esi,%edx          
80488b9:  jl 80488b1            
80488bb:  leal 0xfffffffffe8(%rbp),%esp  
80488be:  popl %ebx             
80488bf:  popl %esi             
80488c0:  popl %edi             
```

- **Performance Cost**
  - multiple clock cycles on modern processor
  - can be a major performance limiter
Conditional Moves

```c
void minmax1(long *a, long *b, long n) {
    long i;
    for (i=0; i<n; i++) {
        if (a[i] > b[i]) {
            long t = a[i];
            a[i] = b[i];
            b[i] = t;
        }
    }
}
```

```c
void minmax2(long *a, long *b, long n) {
    long i;
    for (i=0; i<n; i++) {
        long min = a[i] < b[i]?
            a[i] : b[i];
        long max = a[i] < b[i]?
            b[i] : a[i];
        a[i] = min;
        b[i] = max;
    }
}
```

- Compiled code uses conditional branch
  - 13.5 CPE for random data
  - 2.5 – 3.5 CPE for predictable data

- Compiled code uses conditional move instruction
  - 4.0 CPE regardless of data’s pattern
Latency of Loads

typedef struct ELE {
    struct ELE *next;
    long data;
} list_ele, *list_ptr;

int list_len(list_ptr ls) {
    long len = 0;
    while (ls) {
        len++;
        ls = ls->next;
    }
    return len;
}

• 4 CPE
#define ITERS 100000000

void clear_array() {
    long dest[100];
    int iter;
    for (iter=0; iter<ITERS; iter++) {
        long i;
        for (i=0; i<100; i++)
            dest[i] = 0;
    }
}

• 1 CPE
Store/Load Interaction

```c
void write_read(long *src, long *dest, long n) {
    long cnt = n;
    long val = 0;

    while(cnt--) {
        *dest = val;
        val = (*src)+1;
    }
}
```
Store/Load Interaction

long a[] = {-10, 17};

Example A: write_read(&a[0],&a[1],3)

<table>
<thead>
<tr>
<th>cnt</th>
<th>Initial</th>
<th>Iter. 1</th>
<th>Iter. 2</th>
<th>Iter. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-10 17</td>
<td>-10 0</td>
<td>-10 -9</td>
<td>-10 -9</td>
</tr>
<tr>
<td>val</td>
<td>0</td>
<td>-9</td>
<td>-9</td>
<td>-9</td>
</tr>
</tbody>
</table>

Example B: write_read(&a[0],&a[0],3)

<table>
<thead>
<tr>
<th>cnt</th>
<th>Initial</th>
<th>Iter. 1</th>
<th>Iter. 2</th>
<th>Iter. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-10 17</td>
<td>0 17</td>
<td>1 17</td>
<td>2 17</td>
</tr>
<tr>
<td>val</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

```c
void write_read(long *src,
               long *dest, long n){
    long cnt = n;
    long val = 0;
    while(cnt--) {
        *dest = val;
        val = (*src)+1;
    }
}
```

• CPE 1.3

• CPE 7.3
Some Details of Load and Store

Load unit

Address

Data

Store unit

Store buffer

Address

Data

Matching addresses

Address

Data

Data cache
Inner-Loop Data Flow of Write_Read

\[
\begin{align*}
\text{movq } \%rax, (\%rcx) & \quad \text{\( \ast \text{dest} = \text{val}; \)} \\
\text{movq } (\%rbx), \%rax & \quad \text{val} = \ast \text{src} \\
\text{addq } \$1, \%rax & \quad \text{val}++; \\
\text{subq } \$1, \%rdx & \quad \text{cnt}--; \\
\text{jne loop} & \\
\end{align*}
\]
Inner-Loop Data Flow of Write_Read

%rax %rbx %rcx %rdx

s_addr

s_data

load

sub

add

%rax

%rdx

%rax

%rdx

%rax

%rdx

s_data

load

add

sub
Data Flow
Getting High Performance

• Good compiler and flags
  • Don’t do anything stupid
    – watch out for hidden algorithmic inefficiencies
    – write compiler-friendly code
      » watch out for optimization blockers: procedure calls & memory references
    – look carefully at innermost loops (where most work is done)

• Tune code for machine
  • exploit instruction-level parallelism
  • avoid unpredictable branches
  • make code cache friendly (covered soon)
Hyper Threading

Execution

Instruction Control
- Retirement Unit
- Fetch Control
- Instruction Decode
- Instruction Cache

Instruction Control
- Retirement Unit
- Fetch Control
- Instruction Decode
- Instruction Cache

Functional Units
- Integer/Branch
- General Integer
- FP Add
- FP Mult/Div
- Load
- Store

Data Cache

Operation Results

Addr.
Data
Data
Addr.
Multiple Cores

Chip

Instruction Control

Retirement Unit
Register File
Fetch Control
Instruction Decode
Instruction Cache

Operations

Integer/Branch
General Integer
FP Add
FP Mult/Div
Load
Store
Functional Units

Operation Results

Data Cache

Instruction Control

Retirement Unit
Register File
Fetch Control
Instruction Decode
Instruction Cache

Operations

Integer/Branch
General Integer
FP Add
FP Mult/Div
Load
Store
Functional Units

Operation Results

Data Cache

Other Stuff

More Cache

Other Stuff