CS 33

Architecture and Optimization (2)
Modern CPU Design

Instruction Control

- Instruction Cache
  - Fetch Control
  - Instruction Decode
  - Address
  - Instructions

Instruction Execution

- Functional Units: Integer/Branch, General Integer, FP Add, FP Mult/Div, Load, Store
- Operation Results
- Data Cache
  - Data
  - Addr.

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

• \texttt{addq} %rax, %rdx
  – a single operation
• \texttt{addq} %rax, 8(%rdx)
  – three operations
    » load value from memory
    » add to it the contents of %rax
    » store result in memory
Instruction-Level Parallelism

- \texttt{addq} 8(\%rax), \%rax
  \texttt{addq} \%rbx, \%rdx
  \quad \text{– can be executed simultaneously: completely independent}

- \texttt{addq} 8(\%rax), \%rbx
  \texttt{addq} \%rbx, \%rdx
  \quad \text{– 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

\begin{verbatim}
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
\end{verbatim}

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>
</tbody>
</table>
## Haswell CPU Performance Bounds

<table>
<thead>
<tr>
<th></th>
<th>Integer</th>
<th></th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>Throughput</td>
<td>0.50</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>
x86-64 Compilation of Combine4

- Inner loop (case: integer multiply)

```assembly
.L519:                # Loop:
imull (%rax,%rdx,4), %ecx   # 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>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</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>
Inner Loop

\[
\begin{align*}
\text{load} & \quad \text{mulss} (\%rax, \%rdx, 4), \ %xmm0 \\
\text{mul} & \quad \text{addq} \ $1, \ %rdx \\
\text{add} & \quad \text{cmpq} \ %rdx, \ %rbp \\
\text{cmp} & \quad \text{jg} \ \text{loop} \\
\text{jg} & \quad \text{loop}
\end{align*}
\]
Data-Flow Graphs of Inner Loop

%xmm0  %rax  %rbp  %rdx
load
mul
add
cmp
jg
%xmm0
%rdx

data[i]
load
mul
add
%xmm0
%rdx
Relative Execution Times

data[i] → load → mul → %xmm0
%rdx ← add
Data Flow Over Multiple Iterations

Critical path

data[0] → load → mul → add

data[1] → load → mul → add

... (for n-2 iterations)

data[n-2] → load → mul → add

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

- **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;
}
```
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  

Quiz 1
Effect of Loop Unrolling

- 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?

Compare to before

```c
x = (x OP d[i]) OP d[i+1];
```
Effect of Reassociation

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

```
x = x OP (d[i] OP d[i+1]);
```

- why is that? (next slide)
Reassociated 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:
    \[ \text{CPE} = D/2 \]
  - measured CPE slightly worse for integer addition
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></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>1.01</td>
<td>3.01</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>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>.5</td>
<td>1.0</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
  x_0 &= x_0 \text{ OP } d[i]; \\
  x_1 &= x_1 \text{ OP } d[i+1];
\end{align*}
\]
Separate Accumulators

\[
x_0 = x_0 \text{ OP } d[i]; \\
x_1 = x_1 \text{ OP } d[i+1];
\]

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

- **Overall Performance**
  - N elements, D cycles latency/op
  - should be \((N/2+1)\)*D cycles:
    \[ CPE = D/2 \]
  - 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
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>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Achievable scalar</td>
<td>.54</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>.5</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Using Vector Instructions

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Achievable Scalar</td>
<td>.54</td>
<td>1.01</td>
<td>1.01</td>
</tr>
<tr>
<td>Latency bound</td>
<td>1.00</td>
<td>3.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>.5</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Achievable Vector</td>
<td>.05</td>
<td>.24</td>
<td>.25</td>
</tr>
<tr>
<td>Vector throughput bound</td>
<td>.06</td>
<td>.12</td>
<td>.25</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

Fetch Control
Instruction Decode
Instruction Cache

Retirement Unit
Register File

Register Updates
Prediction OK?

Operation Results

Integer/Branch
General Integer
FP Add
FP Mul/Div
Load
Store

Functional Units

Data Cache

Addr.
Data

CS33 Intro to Computer Systems
XVI–34
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 0xfffffffffe8(%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

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

8048a25:  cmpq %rdi,%rdx
8048a27:  jl 8048a20
8048a29:  movl 0xc(%rbp),%eax
8048a2c:  leal 0xffffffffe8(%rbp),%esp
8048a2f:  movl %ecx,(%rax)
```

Predict taken

Begin execution
Branch Prediction Through Loop

Assume
vector length = 100

Read invalid location

Executed

Fetched
Branch Misprediction Invalidation

Assume
vector length = 100

Predict taken (OK)

Predict taken (oops)

Invalidate
Branch Misprediction Recovery

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>80488b1:</td>
<td>movl (%rcx,%rdx,4),%eax</td>
<td></td>
</tr>
<tr>
<td>80488b4:</td>
<td>addl %eax,(%rdi)</td>
<td></td>
</tr>
<tr>
<td>80488b6:</td>
<td>incl %edx</td>
<td></td>
</tr>
<tr>
<td>80488b7:</td>
<td>cmpl %esi,%edx</td>
<td>$i = 99$</td>
</tr>
<tr>
<td>80488b9:</td>
<td>jl 80488b1</td>
<td>Definitely not taken</td>
</tr>
<tr>
<td>80488bb:</td>
<td>leal 0xfffffffffe8(%rbp),%esp</td>
<td></td>
</tr>
<tr>
<td>80488be:</td>
<td>popl %ebx</td>
<td></td>
</tr>
<tr>
<td>80488bf:</td>
<td>popl %esi</td>
<td></td>
</tr>
<tr>
<td>80488c0:</td>
<td>popl %edi</td>
<td></td>
</tr>
</tbody>
</table>

- 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;
        }
    }
}

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;
}

# len in %rax, ls in %rdi

.L11:
    # loop:
    addq $1, %rax          # incr len
    movq (%rdi), %rdi     # ls = ls->next
    testq %rdi, %rdi      # test ls
    jne .L11              # if != 0
    # go to loop

• 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

```c
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>cnt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</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>cnt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</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>

- CPE 1.3
- CPE 7.3
Some Details of Load and Store

Load unit

Address

Data

Store unit

Store buffer

Address

Data

Matching addresses

Data cache

Address

Data
Inner-Loop Data Flow of Write_Read

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

%rax  %rbx  %rcx  %rdx

1. s_addr
2. s_data
3. load

add
%rax
%rdx

sub
%rax
%rdx

jne

%rax
%rdx

load
s_data
add
%rax
%rdx

sub
%rax
%rdx
Data Flow

```
Critical path

s_data
load
add
sub

s_data
load
add
sub

s_data
load
add
sub

s_data
load
add
sub
```

CS33 Intro to Computer Systems

XVI–48
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**
  - Fetch Control
  - Instruction Decode
  - Instruction Cache
  - Retirement Unit
  - Register File

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

- **Data Cache**
  - Address
  - Data

Operation Results
Multiple Cores

Chip

Instruction Control
- Fetch Control
- Instruction Decode
- Instruction Cache
- Register File
- Operations
- Address
- Instructions

Execution
- Integer/Branch
- General Integer
- FP Add
- FP Mul/Div
- Load
- Store
- Functional Units
- Operation Results
- Data Cache
- Add
- Data
- Data

Other Stuff
More Cache
Other Stuff