CS 33

Architecture and Optimization (2)
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`
- `movvss %xmm0, (%rbp)`
- `addq %r8d, %r9d`
- `imulq %rcx, %r12d`
- `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
Nehalem CPU

- **Multiple instructions can execute in parallel**
  1 load, with address computation
  1 store, with address computation
  2 simple integer (one may be branch)
  1 complex integer (multiply/divide)
  1 FP Multiply
  1 FP Add

- **Some instructions take > 1 cycle, but can be pipelined**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Integer Add</td>
<td>1</td>
<td>.33</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>11–21</td>
<td>11–21</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>4/5</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>10–23</td>
<td>10–23</td>
</tr>
</tbody>
</table>
x86-64 Compilation of Combine4

• Inner loop (case: integer multiply)

```assembly
.L519:
imull (%rax,%rdx,4), %ecx  # Loop:
# 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></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>2.0</td>
<td>3.0</td>
<td>3.0</td>
<td>5.0</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>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</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

\[
\begin{align*}
\%xmm0 & \rightarrow \text{load} & \%rax & \rightarrow \text{mul} & \%rbp & \rightarrow \text{add} & \%rdx & \rightarrow \text{cmp} & \%xmm0 & \rightarrow \text{jg} \\
\%xmm0 & \rightarrow \text{mul} & \%rax & \rightarrow \text{add} & \%rbp & \rightarrow \text{cmp} & \%rdx & \rightarrow \text{jg} \\
\%xmm0 & \rightarrow \text{load} & \%xmm0 & \rightarrow \text{mul} & \%xmm0 & \rightarrow \text{add} & \%rdx & \rightarrow \text{load} & \%xmm0 & \rightarrow \text{mul} & \%rdx & \rightarrow \text{add} \\
\%xmm0 & \rightarrow \text{load} & \%xmm0 & \rightarrow \text{mul} & \%xmm0 & \rightarrow \text{add} & \%rdx & \rightarrow \text{load} & \%xmm0 & \rightarrow \text{mul} & \%rdx & \rightarrow \text{add} \\
\end{align*}
\]
Relative Execution Times

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

\[\text{data}[i] \rightarrow \%xmm0 \rightarrow \%rdx\]
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
Pipelined Data-Flow Over Multiple Iterations
Combine4 = Serial Computation (OP = *)

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

- 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?

a) yes  
b) no
## Effect of Loop Unrolling

<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>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>Latency bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- Helps integer multiply
  - below latency bound
  - compiler does clever optimization

- 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?**
### Effect of Reassociation

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>Unroll 2x, reassociate</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td><strong>Latency bound</strong></td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td><strong>Throughput bound</strong></td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- Nearly 2x speedup for int *, FP +, FP *
  - reason: breaks sequential dependency
  
  \[ x = x \text{ OP} (d[i] \text{ OP} d[i+1]); \]

- why is that? (next slide)
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 \((N/2+1)\times D\) cycles:
  \[ \text{CPE} = \frac{D}{2} \]
- measured CPE slightly worse for FP mult
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></th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>2.0</td>
<td>1.5</td>
<td>3.0</td>
</tr>
<tr>
<td>Unroll 2x, reassociate</td>
<td>2.0</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Unroll 2x parallel 2x</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Latency bound</td>
<td>1.0</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- 2x speedup (over unroll2x) for int *, FP +, FP *
  - breaks sequential dependency in a “cleaner,” more obvious way

\[
x_0 = x_0 \text{ OP } d[i];
\]
\[
x_1 = x_1 \text{ OP } d[i+1];
\]
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 \((N/2+1) \times D\) cycles:
    \[ \text{CPE} = \frac{D}{2} \]
  - CPE matches prediction!

**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
Unrolling & Accumulating

• Idea
  – can unroll to any degree L
  – can accumulate K results in parallel
  – L must be multiple of K

• Limitations
  – diminishing returns
    » cannot go beyond throughput limitations of execution units
  – large overhead for short lengths
    » finish off iterations sequentially
Performance

- K-way loop unrolling with K accumulators
Achievable Performance

<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>Scalar optimum</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</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>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- Limited only by throughput of functional units
- Up to 29X improvement over original, unoptimized code
Using Vector Instructions

<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>Scalar optimum</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Vector optimum</td>
<td>0.25</td>
<td>0.53</td>
</tr>
<tr>
<td>Latency bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Vec throughput bound</td>
<td>0.25</td>
<td>0.50</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

```c
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

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

Execution

- Instruction
- Functional Units
- Load
- Store
- Data Cache

- Integer/Branch
- General Integer
- FP Add
- FP Mult/Div
- Operations
- Prediction OK?
- Register Updates
- Data
- Addr.
- Data
- 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
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

Assume
vector length = 100

Predict taken (OK)

Read
invalid
location

Executed

Fetched
Branch Misprediction Invalidation

Assume
vector length = 100

Predict taken (OK)

Predict taken (oops)

Invalidate

```c
80488b1:  movl  (%rcx,%rdx,4),%eax
80488b4:  addl  %eax,(%rdi)
80488b6:  incl  %edx
80488b7:  cmpl  %esi,%edx  i = 98
80488b9:  jl   80488b1

80488b1:  movl  (%rcx,%rdx,4),%eax
80488b4:  addl  %eax,(%rdi)
80488b6:  incl  %edx
80488b7:  cmpl  %esi,%edx  i = 99
80488b9:  jl   80488b1

80488b1:  movl  (%rcx,%rdx,4),%eax
80488b4:  addl  %eax,(%rdi)
80488b6:  incl  %edx
80488b7:  cmpl  %esi,%edx  i = 100
80488b9:  jl   80488b1

80488b1:  movl  (%rcx,%rdx,4),%eax
80488b4:  addl  %eax,(%rdi)
80488b6:  incl  %edx  i = 101
```
Branch Misprediction Recovery

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

```assembly
80488b1:  movl  (%rcx,%rdx,4),%eax
80488b4:  addl  %eax,(%rdi)
80488b6:  incl  %edx
80488b7:  cmpl  %esi,%edx
80488b9:  jl    80488b1
80488bb:  leal  0xfffffffffe8(%rbp),%esp
80488be:  popl  %ebx
80488bf:  popl  %esi
80488c0:  popl  %edi
```

\[ i = 99 \]

Definitely not taken
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

```c
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;
}
```

```
.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
Clearing an Array ...

```c
#include <stdio.h>

int main() {
    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>Iter. 1</th>
<th>Iter. 2</th>
<th>Iter. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>val</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-10</td>
<td>-10</td>
<td>-10</td>
<td>-10</td>
</tr>
<tr>
<td>17</td>
<td>0</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>Iter. 1</th>
<th>Iter. 2</th>
<th>Iter. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>val</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-10</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
</tbody>
</table>

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

Load unit

Store unit

Data cache

Address

Data

Address

Data

Matching addresses

Address

Data

Store buffer
Inner-Loop Data Flow of Write_Read

```
%eax  %ebx  %ecx  %edx
\hline
\hline
\hline
movq %rax,(%rcx)  *dest = val;
\hline
movq (%rbx),%rax  val = *src
\hline
addq $1,%rax     val++;  
\hline
subq $1,%rdx     cnt--;  
\hline
jne loop
```
Inner-Loop Data Flow of Write_Read

\begin{align*}
\%rax & \quad \%rbx \quad \%rcx \quad \%rdx \\
\text{s\_addr} & \quad \text{s\_data} \\
\text{load} & \quad \text{sub} \\
\text{add} & \quad \text{jne} \\
\%rax & \quad \%rdx \\
\text{s\_data} & \quad \text{load} \\
\%rax & \quad \%rdx \\
\%rax & \quad \%rdx
\end{align*}
Data Flow

Critical path

{s_data} -> {load} -> {add} -> {sub}

{s_data} -> {load} -> {add} -> {sub}

{s_data} -> {load} -> {add} -> {sub}

{s_data} -> {load} -> {add} -> {sub}

...
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

[Diagram of Hyper Threading with blocks labeled as follows:
- Instruction Control
- Functional Units
- Execution

Blocks include:
- Integer/Branch
- General Integer
- FP Add
- FP Mult/Div
- Load
- Store
- Data Cache

Arrows indicate the flow of information between blocks.]
Multiple Cores

Chip

Instruction Control

Fetch Control

Instruction Decode

Instruction Cache

Retirement Unit
Register File

Operations

Instructions

Execution

Operation Results

Instruction

Cache

Data

Cache

Other Stuff

More Cache

Other Stuff

Other Stuff

More Cache

Other Stuff

Instruction Control

Fetch Control

Instruction Decode

Instruction Cache

Retirement Unit
Register File

Operations

Instructions

Execution

Operation Results

Instruction

Cache

Data

Cache

Other Stuff

More Cache

Other Stuff

More Cache

Other Stuff

More Cache

Other Stuff