Most of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer's Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.
Supplied by CMU.
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)

Supplied by CMU.
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 %r8d, %r9d
  imulq %rcx, %r12d
  addq $1, %rdx

Note that the first three instructions are floating-point instructions, and %xmm0 is a floating-point register. We will discuss x86 floating point in an upcoming lecture.
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
Supplied by CMU.

“Nehalem” is Intel’s code name for its Core I7 processor design.
x86-64 Compilation of Combine4

- Inner loop (case: integer multiply)

```assembly
.L519:
imull (%rax, %rdx, 4), %ecx  # Loop:
addq $1, %rdx  # %t = %t * d[i]
addq %rdx, %rbp  # 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>2.0</td>
<td>3.0</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>

Supplied by CMU.
This is Figure 5.13 of Bryant and O'Hallaron. It shows the code for the single-precision floating-point version of our example.

```
mulss (%rax,%rdx,4), %xmm0
addq $1,%rdx
cmpq %rdx,%rbp
jg loop
```
These are Figures 5.14 a and b of Bryant and O’Hallaron.
Here we modify the graph of the previous slide to show the relative times required of \textit{mul}, \textit{load}, and \textit{add}.
This is Figure 5.15 of Bryant and O'Hallaron.
Without pipelining, the data flow would appear as shown in the slide.
Pipelined Data-Flow Over Multiple Iterations
Since the loads can be pipelined, it’s clear that the multiplies form the critical path. (Note that the multiplies cannot be pipelined since each subsequent multiply depends on the result of the previous.)
Since the multiplies form the critical path, here we focus only on them.
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
Supplied by CMU.

Loop Unrolling

```c
void unroll2x(vec_ptr_t vec, data_t *dest)
{
    int length = vec_length(vec);
    int limit = length-1;
    data_t *d = get_vec_start(vec);
    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

Quiz 1
Does it speed things up?

a) yes
b) no
What the compiler does for the case of integer multiplication is to apply reassociation, discussed in the next slide.

- 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?*

Supplied by CMU.
**Effect of Reassociation**

<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>Unroll 2x, reassociate</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>

- 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)

Supplied by CMU.
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:
    - \( CPE = D/2 \)
  - measured CPE slightly worse for FP mult

Supplied by CMU.
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>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>Unroll 2x parallel 2x</td>
<td>1.5</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>

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

```plaintext
x0 = x0 OP d[i];
x1 = x1 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 \((N/2+1)D\) cycles:
    \[
    \text{CPE} = \frac{D}{2}
    \]
  - CPE matches prediction!

---

**What Now?**

Supplied by CMU.
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>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Scalar optimum</td>
<td>1.00</td>
<td>1.00</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>
</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>Operation</td>
<td>Add</td>
<td>Mult</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

Supplied by CMU.

We’ll look at vector instructions in an upcoming lecture.
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

Supplied by CMU, converted to x86-64.
Supplied by CMU.
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

```assembly
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)
```

Supplied by CMU.
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  0xffffffff8(%rbp),%esp
8048a2f:  movl  %ecx,(%rax)
```

Supplied by CMU.
Branch Prediction Through Loop

Supplied by CMU.
### Branch Misprediction Invalidation

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>80488b1</td>
<td>movl (%rcx,%rdx,4),%eax</td>
<td>Predict taken (OK)</td>
<td>Assume vector length = 100</td>
</tr>
<tr>
<td>80488b4</td>
<td>addl %eax,(%rdi)</td>
<td>Predict taken (OK)</td>
<td></td>
</tr>
<tr>
<td>80488b6</td>
<td>incl %edx</td>
<td>Predict taken (OK)</td>
<td></td>
</tr>
<tr>
<td>80488b7</td>
<td>cmp %esi,%edx</td>
<td>Predict taken (OK)</td>
<td></td>
</tr>
<tr>
<td>80488b9</td>
<td>jl 80488b1</td>
<td>Predict taken (OK)</td>
<td></td>
</tr>
<tr>
<td>80488b1</td>
<td>movl (%rcx,%rdx,4),%eax</td>
<td>Predict taken (oops)</td>
<td></td>
</tr>
<tr>
<td>80488b4</td>
<td>addl %eax,(%rdi)</td>
<td>Predict taken (oops)</td>
<td></td>
</tr>
<tr>
<td>80488b6</td>
<td>incl %edx</td>
<td>Predict taken (oops)</td>
<td></td>
</tr>
<tr>
<td>80488b7</td>
<td>cmp %esi,%edx</td>
<td>Predict taken (oops)</td>
<td></td>
</tr>
<tr>
<td>80488b9</td>
<td>jl 80488b1</td>
<td>Predict taken (oops)</td>
<td></td>
</tr>
<tr>
<td>80488b1</td>
<td>movl (%rcx,%rdx,4),%eax</td>
<td>Invalidate</td>
<td></td>
</tr>
<tr>
<td>80488b4</td>
<td>addl %eax,(%rdi)</td>
<td>Invalidate</td>
<td></td>
</tr>
<tr>
<td>80488b6</td>
<td>incl %edx</td>
<td>Invalidate</td>
<td></td>
</tr>
<tr>
<td>80488b7</td>
<td>cmp %esi,%edx</td>
<td>Invalidate</td>
<td></td>
</tr>
<tr>
<td>80488b9</td>
<td>jl 80488b1</td>
<td>Invalidate</td>
<td></td>
</tr>
</tbody>
</table>

Supplied by CMU.
Branch Misprediction Recovery

80488b1:  movl  (%ecx,%rdx,4),%eax
80488b4:  addl  %eax,(%rdi)
80488b6:  incl  %edx
80488b7:  cmpq  %esi,%edx
80488b9:  jle  80488b1  
80488bb:  leal  0xffffffe8(%ebp),%esp
80488be:  popl  %ebx
80488bf:  popl  %esi
80488c0:  popl  %edi

i = 99
Defineitely not taken

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

Supplied by CMU.
<table>
<thead>
<tr>
<th>Conditional Moves</th>
<th>Compiled code uses conditional branch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• 13.5 CPE for random data</td>
</tr>
<tr>
<td></td>
<td>• 2.5 – 3.5 CPE for predictable data</td>
</tr>
</tbody>
</table>

This example is from the textbook. Note that in `minmax1`, a conditional move cannot be used, since the compiler does not know whether `a` and `b` are aliased. In `minmax2`, since both `min` and `max` are computed, the compiler is assured that aliasing doesn't matter.
This example is from the textbook (Figure 5.31). Here we can’t execute the loads in parallel, since each load is dependent on the result of the previous load. The point is that loads (fetching data from memory) have a latency of 4 cycles.
This is adapted from Figure 5.32 of the textbook. There are no data dependencies and thus the stores can be pipelined.
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;
    }
}
```

This code is from the textbook.
This is Figure 5.33 of the textbook. Performance depends upon whether src and dest are the same location.
This is Figure 5.34 of the textbook.
This is Figure 5.35 of the textbook.
This is Figure 5.36 of the textbook.
This is adapted from Figure 5.37 of the textbook.
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
Multiple Cores

Chip

Instruction Control

Fetch Control

Operands

Integer

Float

Functional

Units

Operation Results

Data Cache

Execution

Other Stuff

More Cache

Other Stuff