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 %r8, %r9
- imulq %rcx, %r12
- addq $1, %rdx

these can be executed without waiting for the first three to finish

Note that the first three instructions are floating-point instructions, and %xmm0 is a floating-point register.
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

Supplied by CMU.

“Haswell” is Intel’s code name for recent versions of its Core I7 and Core I5 processor design. Most of the computers in Brown CS employ Core I5 processors.
# 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>

| Load                 | 4       | 1            | 2        |
| Store                | -       | 1            | 2        |

Supplied by CMU.

These figures are for those cases in which the operands are either in registers or are immediate. For the other cases, additional time is required to load operands from memory or store them to memory.

"Cycles/Issue" is the number of clock cycles that must occur between the start of execution of one instruction to the start of execution of the next. The reciprocal of this value is the throughput: the number of instructions (typically a fraction) that can be completed per cycle.

The figures for load and store assume the data is coming from/go to the data cache. Much more time is required if the source or destination is RAM.

The latency for stores is a bit complicated – we discuss it later in this lecture.
Derived from a slide provided by CMU.

We assume that the source and destination are either immediate (source only) or registers. Thus any bottlenecks due to memory access do not arise.

Each integer add requires one clock cycle of latency. It’s also the case that, for each functional unit doing integer addition, the time required between add instructions is one clock cycle. However, since there are four such functional units, all four can be kept busy with integer add instructions and thus the aggregate throughput can be as good as one integer add instruction completing, on average, every .25 clock cycles.

Each integer multiply requires three clock cycles. But since a new multiply instruction can be started every clock cycle (i.e., they can be pipelined), the aggregate throughput can be as good as one integer multiply completing every clock cycle.

Each floating point multiply requires five clock cycles, but they can be pipelined with one starting every clock cycle. Since there are two functional units that can perform floating point multiply, the aggregate throughput can be as good as one completing every .5 clock cycles.
x86-64 Compilation of Combine4

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

```
.L519:     # Loop:
mulss (%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>Add</td>
<td>Mult</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>

Supplied by CMU.

These numbers are for the Haswell CPU.
This is Figure 5.13 of Bryant and O'Hallaron. It shows the code for the single-precision floating-point version of our example.
These are Figures 5.14 a and b of Bryant and O'Hallaron.

Since the values in %rax and %rbp don't change during the execution of the inner loop, they're not critical to the scheduling and timing of the instructions. Assuming the branch is taken, the cmp and jg instructions also aren't a factor in determining the timing of the instructions. We focus on what's shown in the righthand portion of the slide.
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. In what’s shown here, only one multiply can be done at a time, since the result of the one multiply is needed for the next.
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

```c
void unroll2x(vec_ptr_t vec, data_t *dest)
{
    int length = vec_length(vec);
    int limit = length-1;
    data_t *d = get_vec_data(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 by allowing more parallelism?

a) yes  
b) no

Supplied by CMU.
### 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>1.27</td>
<td>3.00</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>1.01</td>
<td>3.00</td>
</tr>
<tr>
<td>Latency bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>0.25</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- Helps integer add
  - reduces loop overhead
- Others don’t improve. *Why?*
  - still sequential dependency

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

Supplied by CMU.
**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.
How much time is required to compute the products shown in the slide? The multiplications in the upper right of the tree, directly involving the \( d_i \), could all be done at once, since there are no dependencies; thus computing them can be done in \( D \) cycles, where \( D \) is the latency required for multiply. This assumes we have a sufficient number of functional units to do this, thus this is a lower bound. The multiplications in the lower left must be done sequentially, since each depends on the previous; thus computing them requires \((N/2)*D\) cycles. Since first of the top right multiplies must be completed before the bottom left multiplies can start, the overall performance has a lower bound of \((N/2 + 1)*D\).
Effect of Reassociation

<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>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>Unroll 2x, reassociate</td>
<td>1.01</td>
<td>1.51</td>
<td>1.51</td>
<td>2.51</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>.25</td>
<td>1.0</td>
<td>1.0</td>
<td>.5</td>
</tr>
</tbody>
</table>

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

```c
x = x 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 = IDENTITY;
    data_t x1 = IDENTITY;
    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>3.00</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>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>.25</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

  ```
  x0 = x0 OP d[i];
  x1 = x1 OP d[i+1];
  ```

Supplied by CMU.
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:
    \[
    \text{CPE} = \frac{D}{2}
    \]
  - 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
This is Figure 5.30 from the textbook.
Based on a slide supplied by CMU.
Based on a slide supplied by CMU.

We’ll look at vector instructions in an upcoming lecture.
SSE stands for “streaming SIMD extensions”. SIMD stands for “single instruction multiple data” – these are instructions that operate on vectors.
What About Branches?

- **Challenge**
  - Instruction control unit must work well ahead of execution unit to generate enough operations to keep EU busy

```plaintext
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.
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  # Branch not-taken
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

Supplied by CMU.
Branch Prediction Through Loop

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

Assume
vector length = 100

Predict taken (OK)

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

i = 98

Predict taken (oops)

Read invalid location

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

i = 99

Executed

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

i = 100

Fetched

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

i = 101

Supplied by CMU.
Branch Misprediction Invalidation

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

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

Assume
vector length = 100

Predict taken (OK)

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

Predict taken (oops)

```
80488b1:  movl  (%rcx,%rdx,4),%eax  
80488b4:  addl  %eax,%rdi          
80488b6:  incl  %edx              
80488b7:  cmpl  %esi,%edx         
80488b9:  jl  80488b1             
     i = 101
```

Invalidate

Supplied by CMU.
Branch Misprediction Recovery

80488b1: movl (%rcx,%rdx,4),%eax
80488b4: addl %eax,(%rdi)
80488b6: incl %edx
80488b7: cmpl %esi,%edx
80488b9: jle 80488b1
80488bb: leal 0xffffffffe8(%rbp),%esp
80488be: popl %ebx
80488bf: popl %esi
80488c0: popl %edi

\( i = 99 \) Definitely not taken

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

Supplied by CMU.
### Conditional Moves

**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

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.
Clearing an Array ...

```
#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

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

    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. If they are different locations, they don't interact so that loads and stores can be pipelined. If they are the same locations, then they do interact and pipelining is not possible.
This is Figure 5.34 of the textbook.

The processor's store unit maintains a small store buffer containing the results of recent store operations. This is queried by the load unit before it accesses memory (via the data cache). Thus the load unit can get the results of recent stores relatively quickly via the store buffer.
This is Figure 5.35 of the textbook.
The solid lines represent dependencies. The dashed line represents a possible dependency (if the address used by the load is the same as the address used by the store).
This is Figure 5.36 of the textbook.

The arc labelled 1 represents the dependency that the address of where data must be stored must be computed before the data is stored.

The arc labelled 2 represents the check the processor must make to test whether the store will be to the location of the load.

The arc labelled 3 represents the dependency of the load on the stored data if they use the same address.
This is adapted from Figure 5.37 of the textbook.

On the left is the case in which the store is to a different location than the store. On the right is the case in which they are to the same location.
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)

Supplied by CMU.
One way of improving the utilization of the functional units of a processor is hyperthreading. The processor supports multiple instruction streams ("hyper threads"), each with its own instruction control. But all the instruction streams share the one set of functional units.
Going a step further, one can pack multiple complete processors onto one chip. Each processor is known as a core and can execute instructions independently of the other cores (each has its private set of functional units). In addition to each core having its own instruction and data cache, there are caches shared with the other cores on the chip. We discuss this in more detail in a subsequent lecture.

In many of today’s processor chips, hyperthreading is combined with multiple cores. Thus, for example, a chip might have four cores each with four hyperthreads. Thus the chip might handle 16 instruction streams.