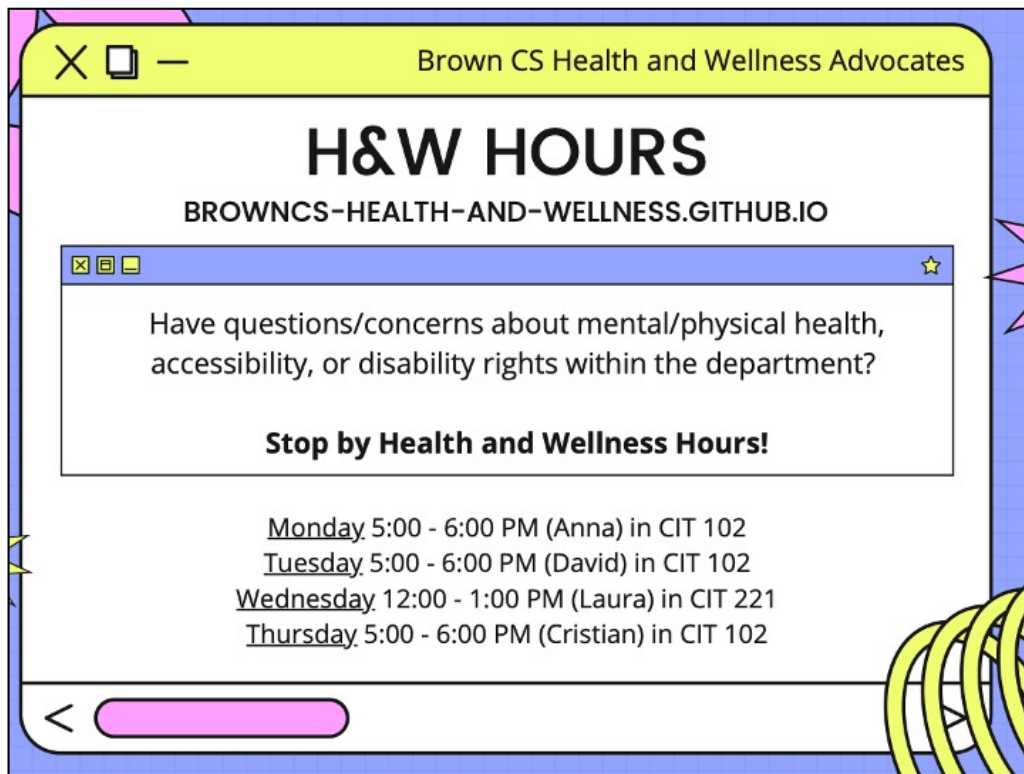


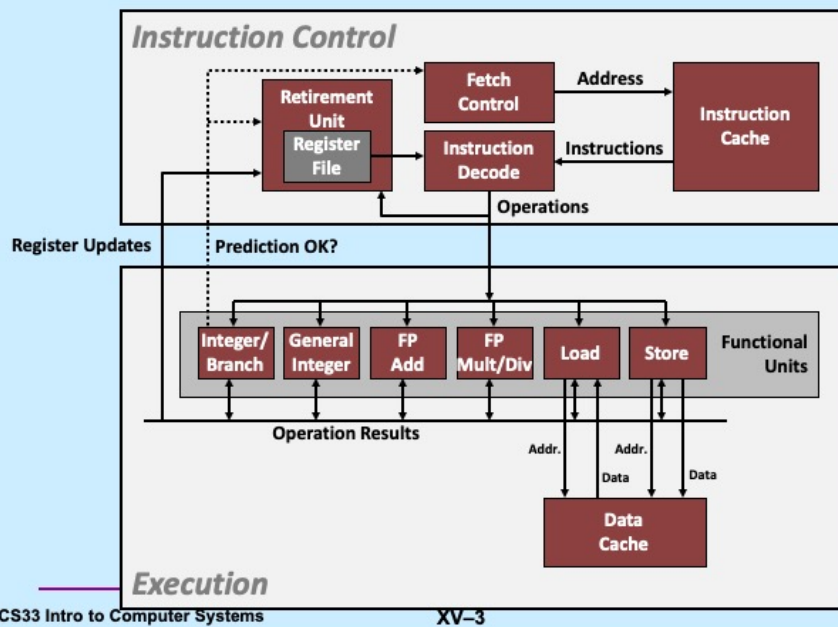
CS 33

Architecture and Optimization (2)

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.



Modern CPU Design



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 relatively 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**

<i>Instruction</i>	<i>Latency</i>	<i>Cycles/Issue</i>	<i>Capacity</i>
Integer Add	1	1	4
Integer Multiply	3	1	1
Integer/Long Divide	3-30	3-30	1
Single/Double FP Add	3	1	1
Single/Double FP Multiply	5	1	2
Single/Double FP Divide	3-15	3-15	1
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 from the start of execution of one instruction to the start of execution to the next. The reciprocal of this value is the throughput: the number of instructions (typically a fraction) that can be completed per cycle.

"Capacity" is the number of functional units that can do the indicated operations.

The figures for load and store assume the data is coming from/going 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.

Haswell CPU Performance Bounds

	Integer		Floating Point	
	+	*	+	*
Latency	1.00	3.00	3.00	5.00
Throughput	4.00	1.00	1.00	2.00

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, for a throughput of 4 instructions/cycle.

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, for a throughput of 2 instructions/cycle.

x86-64 Compilation of Combine4

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

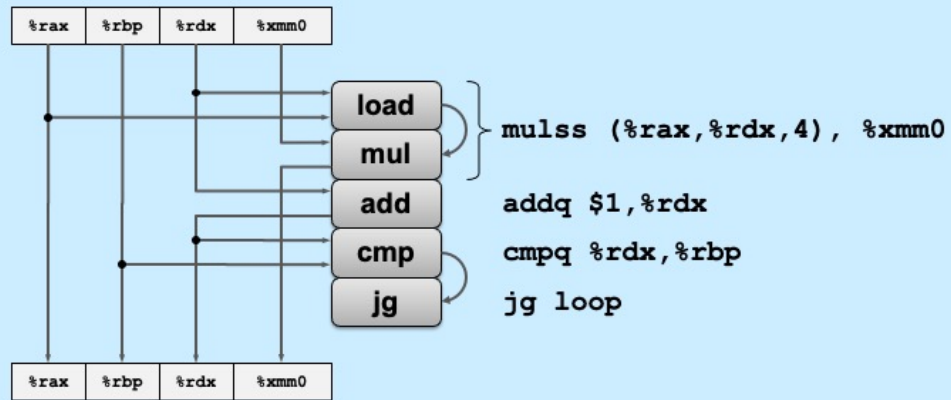
```
.L519:                # Loop:
    mullss (%rax,%rdx,4), %xmm0 # t = t * d[i]
    addq $1, %rdx           # i++
    cmpq %rdx, %rbp         # Compare length:i
    jg .L519                # If >, goto Loop
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.00	3.00	5.00
Latency bound	1.00	3.00	3.00	5.0
Throughput bound	0.25	1.00	1.00	0.50

Supplied by CMU.

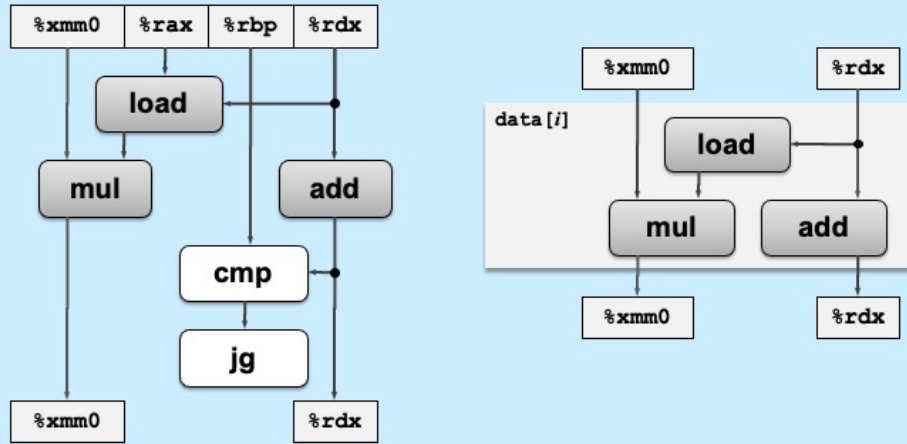
These numbers are for the Haswell CPU. The row labelled "Combine4" gives the actual time, in clock cycles, taken by each execution of the loop. The row labelled "Latency bound" gives the time required for the arithmetic instruction (integer add or multiply, double-precision floating-point add or multiply) in each execution of the loop. The last row, "Throughput bound", gives the time required for the arithmetic instruction if they can be executed without delays by the multiple execution units – i.e., there are no data hazards (as explained in the previous lecture).

Inner Loop



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

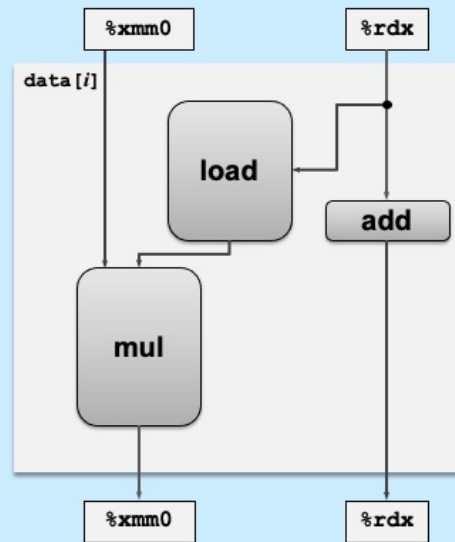
Data-Flow Graphs of Inner Loop



These are Figures 5.14 a and b of Bryant and O'Hallaron.

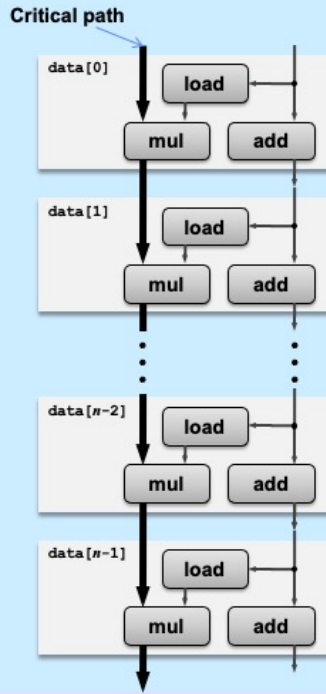
Since the values in `%rax` and `%rbp` don't change during the execution of the inner loop, they're not critical to the scheduling and timing of the instructions. Assuming the branch is taken, the **cmp** and **lg** instructions also aren't a factor in determining the timing of the instructions. We focus on what's shown in the righthand portion of the slide.

Relative Execution Times



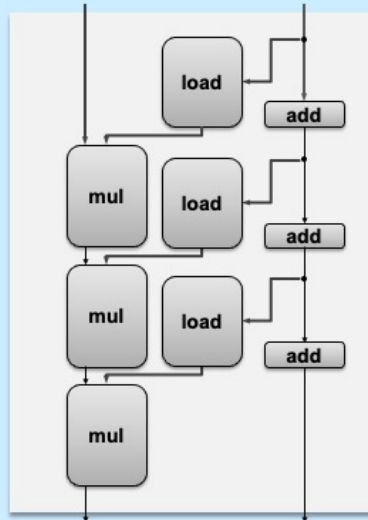
Here we modify the graph of the previous slide to show the relative times required of **mul**, **load**, and **add**.

Data Flow Over Multiple Iterations



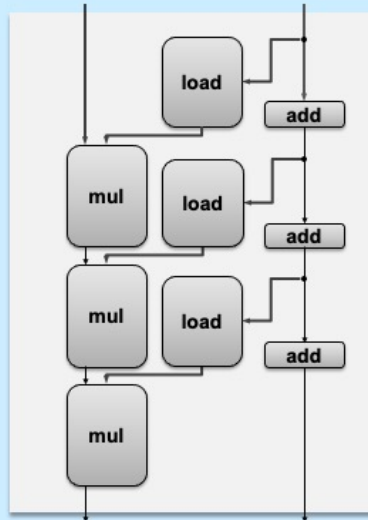
This is Figure 5.15 of Bryant and O'Hallaron.

Pipelined Data-Flow Over Multiple Iterations



Without pipelining, the data flow would appear as shown in the slide.

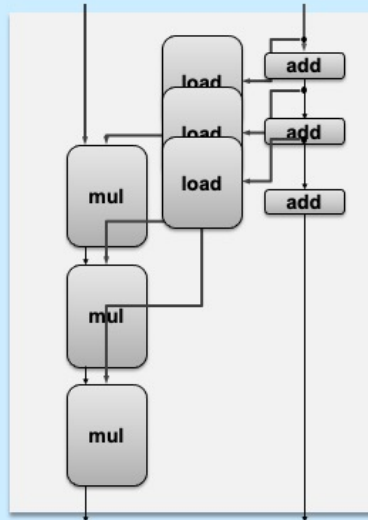
Pipelined Data-Flow Over Multiple Iterations



The loads depend only on the computation of the array index, which is quickly done by addition units. Thus, the loads can be pipelined.

It's clear that the multiplies form the critical path, since they use the results of the previous multiplies.

Pipelined Data-Flow Over Multiple Iterations



The loads depend only on the computation of the array index, which is quickly done by addition units. Thus, the loads can be pipelined.

It's clear that the multiplies form the critical path, since they use the results of the previous multiplies.

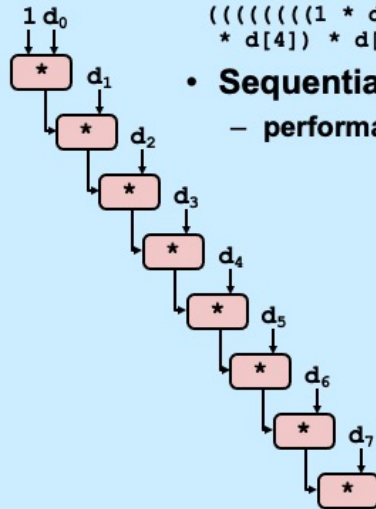
Combine4 = Serial Computation (OP = *)

- **Computation (length=8)**

$(((((1 * d[0]) * d[1]) * d[2]) * d[3]) * d[4]) * d[5]) * d[6]) * d[7])$

- **Sequential dependence**

– performance: determined by latency of OP



Supplied by CMU.

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

```
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.

Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.00	3.00	5.00
Unroll 2x	1.01	3.00	3.00	5.00
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	0.25	1.0	1.0	0.5

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

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

Supplied by CMU.

Loop Unrolling with Reassociation

```
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;
}
```

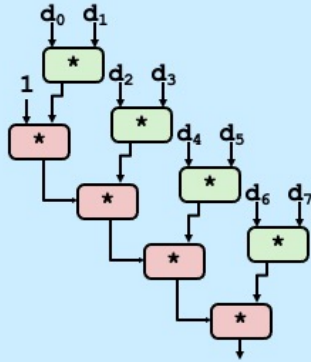
Compare to before

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

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

Reassociated Computation

```
x = x OP (d[i] 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 integer addition (there are other things going on)

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

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.00	3.00	5.00
Unroll 2x	1.01	3.00	3.00	5.00
Unroll 2x, reassociate	1.01	1.51	1.51	2.51
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	.25	1.0	1.0	.5

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

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

Supplied by CMU.

Loop Unrolling with Separate Accumulators

```
void unroll2x2x(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**

Supplied by CMU.

Here one "accumulator" (x0) is summing the array elements with even indices, the other (x1) is summing array elements with odd indices.

Effect of Separate Accumulators

Method	Integer		Double FP	
	Add	Mult	Add	Mult
Combine4	1.27	3.00	3.00	5.00
Unroll 2x	1.01	3.00	3.00	5.00
Unroll 2x, reassociate	1.01	1.51	1.51	2.01
Unroll 2x parallel 2x	.81	1.51	1.51	2.51
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	.25	1.0	1.0	.5

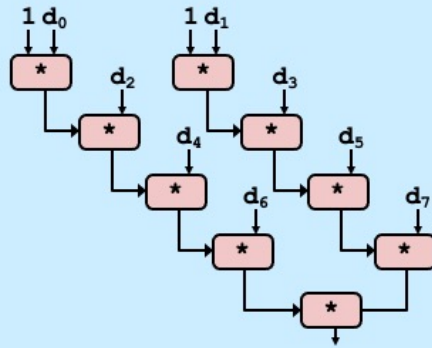
- 2x speedup (over unroll 2x) for int *, FP +, FP *
 - breaks sequential dependency in a “cleaner,” more obvious way

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

Supplied by CMU.

Separate Accumulators

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

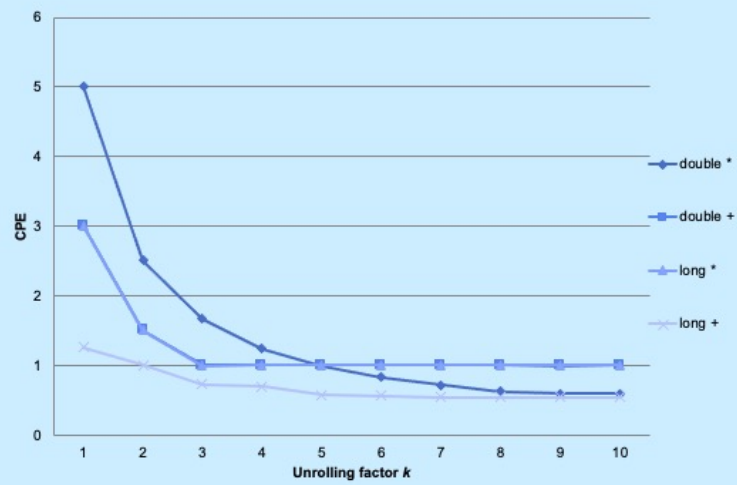
What Now?

Quiz 1

We're making progress. With two accumulators we get a two-fold speedup. With three accumulators, we can get a three-fold speedup. How much better performance can we expect if we add even more accumulators?

- a) It keeps on getting better as we add more and more accumulators**
- b) It's limited by the latency bound**
- c) It's limited by the throughput bound**
- d) It's limited by something else**

Performance



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

This is Figure 5.30 from the textbook.

Achievable Performance

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.0	3.0	5.0
Achievable scalar	.52	1.01	1.01	.54
Latency bound	1.00	3.00	3.00	5.00
Throughput bound	.25	1.00	1.00	.5

Based on a slide supplied by CMU.

Using Vector Instructions

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.0	3.0	5.0
Achievable Scalar	.52	1.01	1.01	.54
Latency bound	1.00	3.00	3.00	5.00
Throughput bound	.25	1.00	1.00	.5
Achievable Vector	.05	.24	.25	.16
Vector throughput bound	.06	.12	.25	.12

- **Make use of SSE Instructions**
 - parallel operations on multiple data elements

Based on a slide supplied by CMU.

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

```
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
```

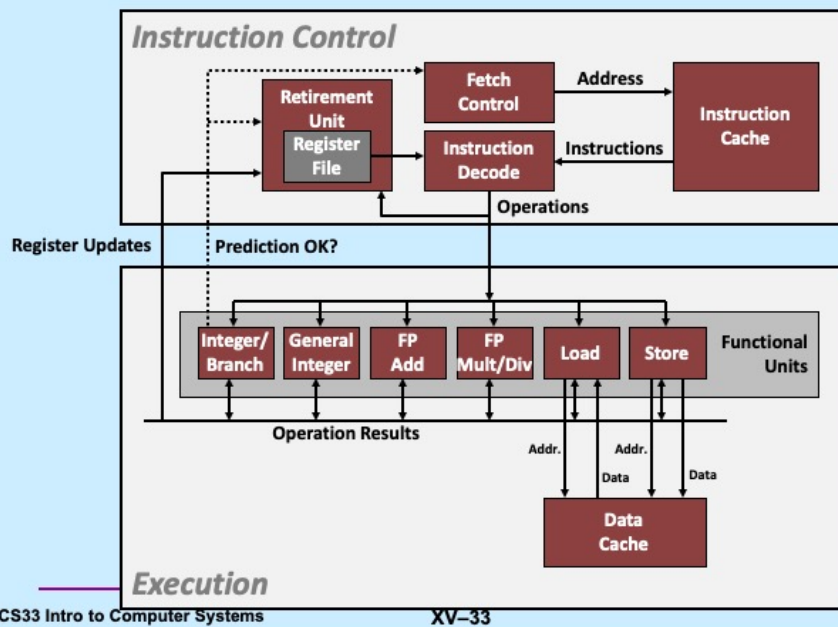
} Executing

How to continue?

- when it encounters conditional branch, cannot reliably determine where to continue fetching

Supplied by CMU, converted to x86-64.

Modern CPU Design



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

Branch not-taken

Branch taken

```
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

- 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
. . .
```

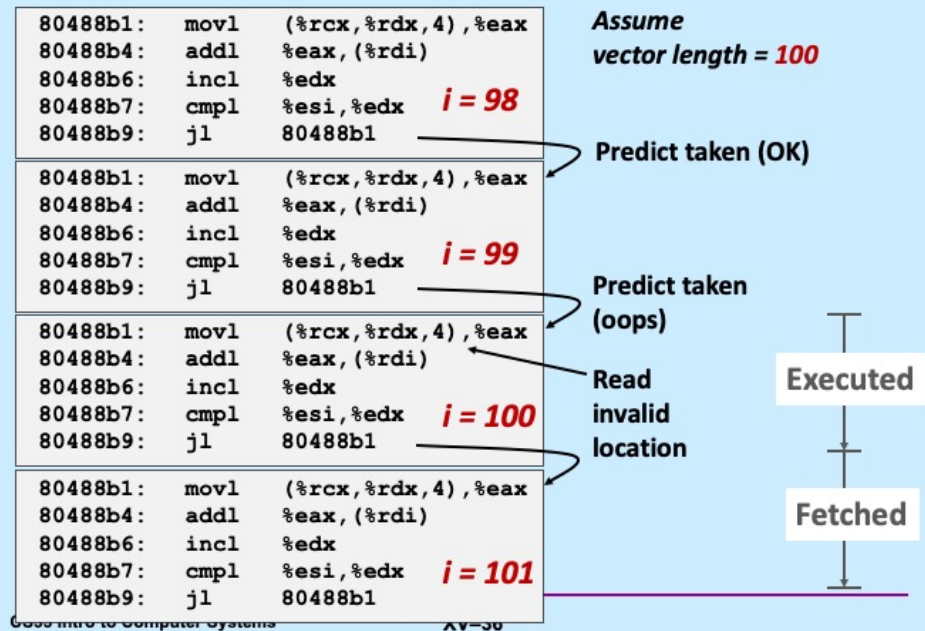
Predict taken

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

Begin
execution

Supplied by CMU.

Branch Prediction Through Loop



Supplied by CMU.

Branch Misprediction Invalidation

```
80488b1: movl    (%rcx,%rdx,4),%eax
80488b4: addl    %eax, (%rdi)
80488b6: incl    %edx
80488b7: cmpl    %esi,%edx    i = 98
80488b9: j1l     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    i = 99
80488b9: j1l     80488b1
```

Predict taken (oops)

```
80488b1: movl    (%rcx,%rdx,4),%eax
80488b4: addl    %eax, (%rdi)
80488b6: incl    %edx
80488b7: cmpl    %esi,%edx    i = 100
80488b9: j1l     80488b1
```

Invalidate

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

Branch Misprediction Recovery

```
80488b1: movl    (%rcx,%rdx,4),%eax
80488b4: addl    %eax,(%rdi)
80488b6: incl    %edx
80488b7: cmpl    %esi,%edx    i = 99
80488b9: jnl     80488b1
80488bb: leal    0xffffffffe8(%rbp),%esp
80488be: popl    %ebx
80488bf: popl    %esi
80488c0: popl    %edi
```

Definitely not taken

- **Performance Cost**

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

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

This example is from the textbook (Figure 5.31). Here we can't execute the loads (of `ls->next`) 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.

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;
    }
}
```

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

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

	Initial	Iter. 1	Iter. 2	Iter. 3
cnt	3	2	1	0
a	-10 17	-10 0	-10 -9	-10 -9
val	0	-9	-9	-9

• CPE 1.3

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

	Initial	Iter. 1	Iter. 2	Iter. 3
cnt	3	2	1	0
a	-10 17	0 17	1 17	2 17
val	0	1	2	3

• CPE 7.3

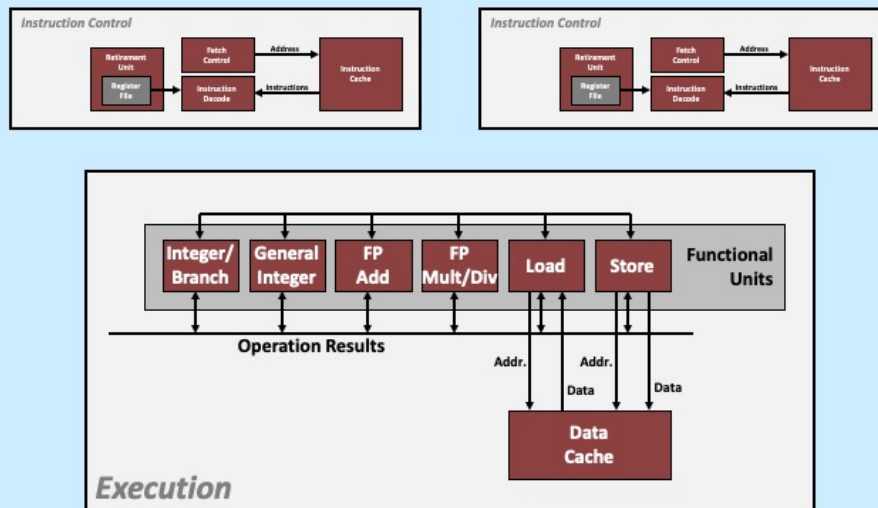
This is Figure 5.33 of the textbook. Performance depends upon whether **src** and **dest** are the same location. If they are different locations, they don't interact, and loads and stores can be pipelined. If they are the same locations, then they do interact, and pipelining is not possible.

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:
function 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)

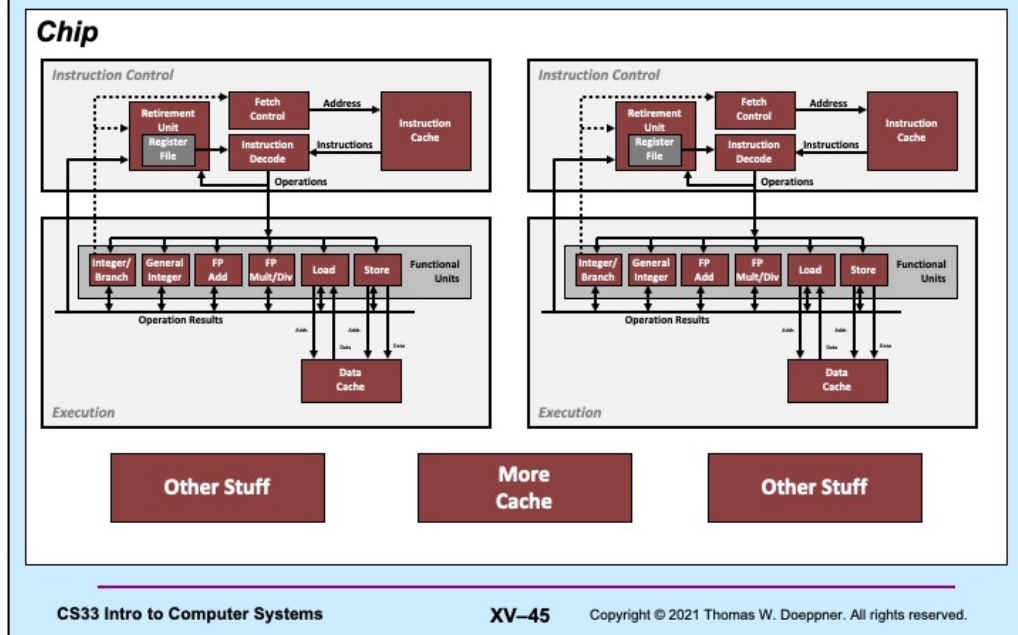
Supplied by CMU.

Hyper Threading



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.

Multiple Cores



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.