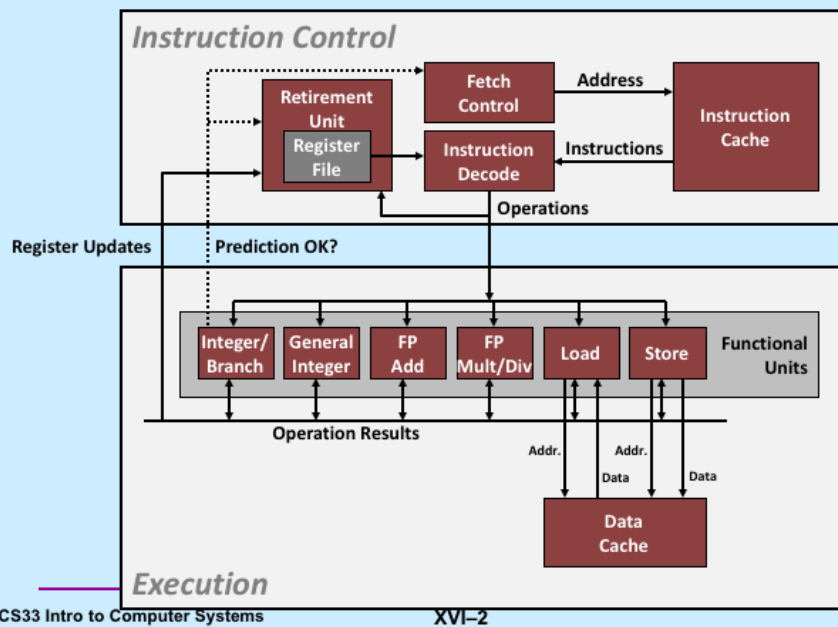


# CS 33

## 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,” 2<sup>nd</sup> 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.

# 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
  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 processor design.



## 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

Supplied by CMU.

“Haswell” is Intel’s code name for recent versions of its Core I7 processor design.

## Haswell CPU Performance Bounds

	Integer		Floating Point	
	+	*	+	*
Latency	1.00	3.00	3.00	5.00
Throughput	0.50	1.00	1.00	0.50

## x86-64 Compilation of Combine4

- Inner loop (case: integer multiply)

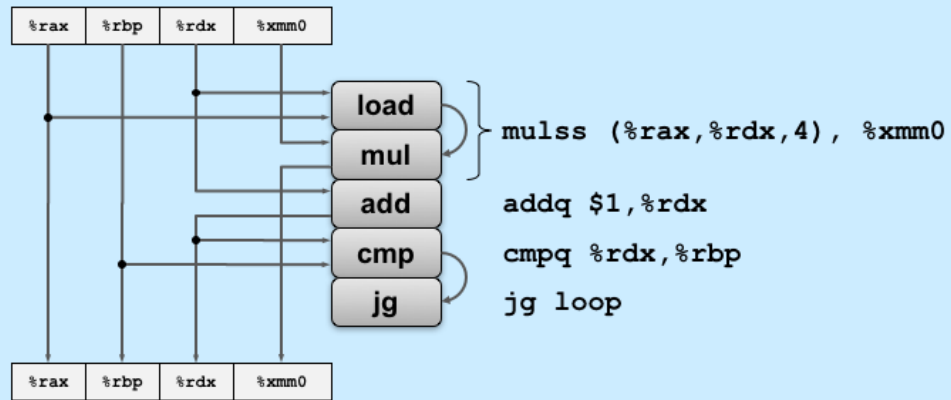
```
.L519:                # Loop:
    imull (%rax,%rdx,4), %ecx    # t = t * d[i]
    addq $1, %rdx               # i++
    cmpq %rdx, %rbp             # Compare length:i
    jg    .L519                 # If >, goto Loop
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Latency bound	1.00	3.00	3.00	5.0
Throughput bound	0.50	1.00	1.00	0.50

Supplied by CMU.

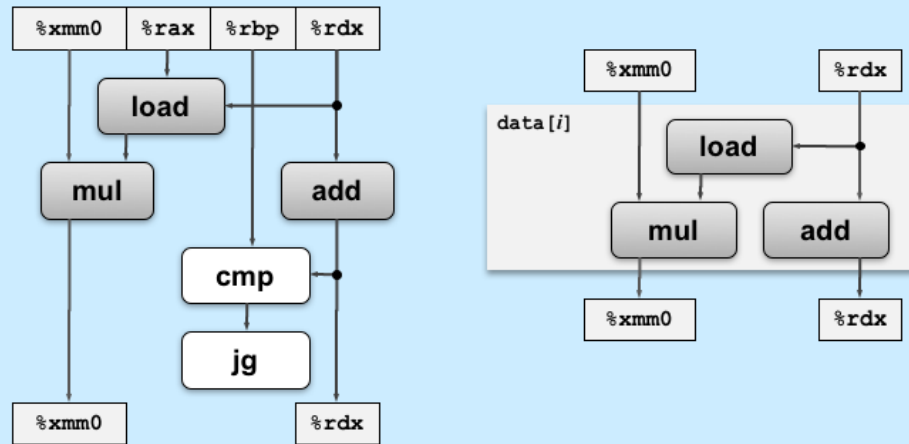
These numbers are for the Haswell CPU.

## Inner Loop



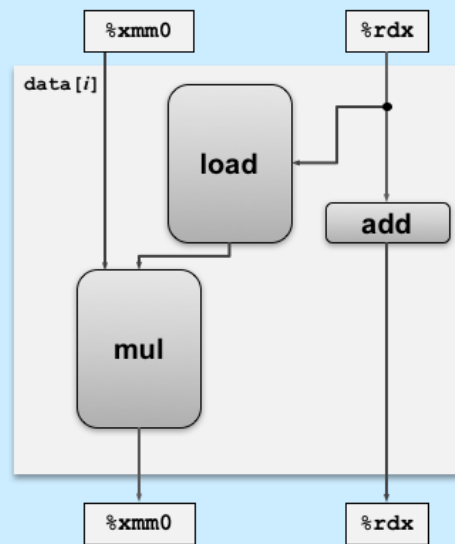
This is Figure 5.13 of Bryant and O'Hallaron. It shows the code for the single-precision floating-point version of our example.

## Data-Flow Graphs of Inner Loop



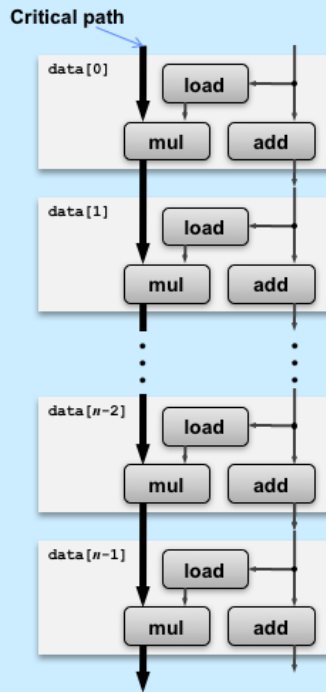
These are Figures 5.14 a and b of Bryant and O'Hallaron.

## Relative Execution Times



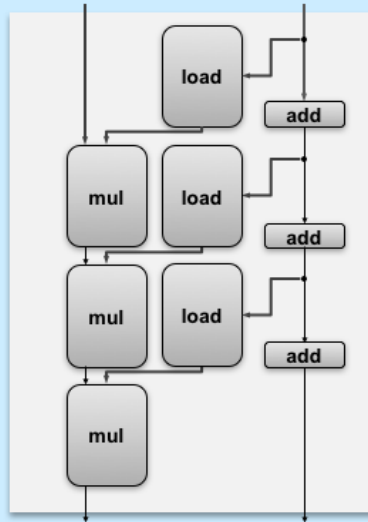
Here we modify the graph of the previous slide to show the relative times required of *mul*, *load*, and *add*.

## Data Flow Over Multiple Iterations



This is Figure 5.15 of Bryant and O'Hallaron.

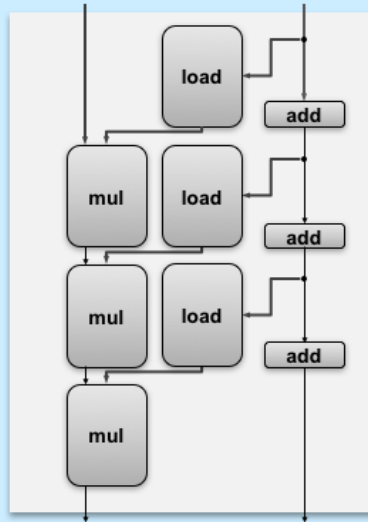
## Pipelined Data-Flow Over Multiple Iterations



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



## Pipelined Data-Flow Over Multiple Iterations

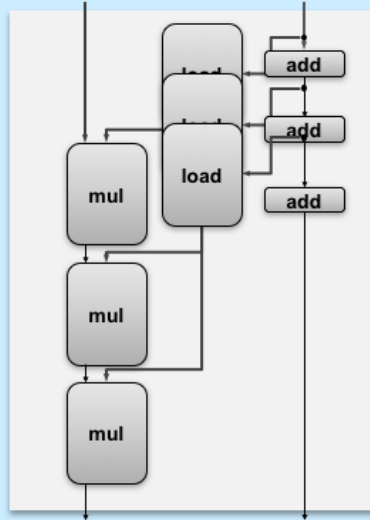


# Pipelined Data-Flow Over Multiple Iterations

CS33 Intro to Computer Systems

XVI-18

Copyright © 2017 Thomas W. Doepfner. All rights reserved.



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.)

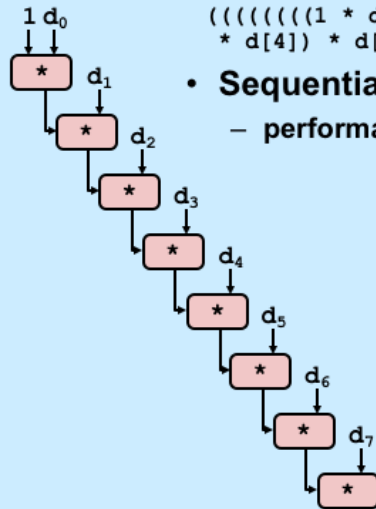
## Combine4 = Serial Computation (OP = \*)

- **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.

# 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.

## Loop Unrolling

```
void unroll2x(vec_ptr_t v)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_data(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

- **Perform 2x more useful work per iteration**

## Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x	1.01	3.01	3.01	5.01
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	0.5	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];
```

## 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?*

## Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x	1.01	3.01	3.01	5.01
Unroll 2x, reassociate	1.01	1.51	1.51	2.51
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	.5	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]);
```

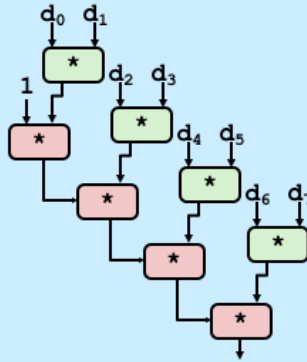
- why is that? (next slide)

Supplied by CMU.



# 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

Supplied by CMU.

How much time is required to compute the products shown in the slide? The multiplications in the upper right, 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. 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  $(N/2 + 1)*D$ .

## Loop Unrolling with Separate Accumulators

```
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**

Supplied by CMU.

## Effect of Separate Accumulators

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x	1.01	3.01	3.01	5.01
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	.5	1.0	1.0	.5

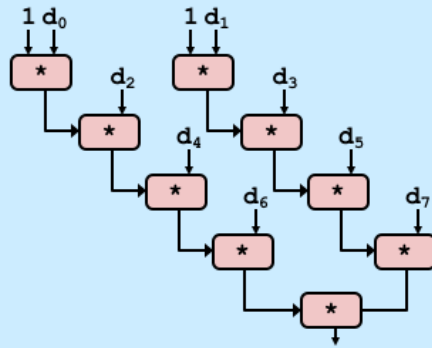
- 2x speedup (over unroll 2x) for int \*, FP +, FP \*
  - breaks sequential dependency in a “cleaner,” more obvious way

```
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?***

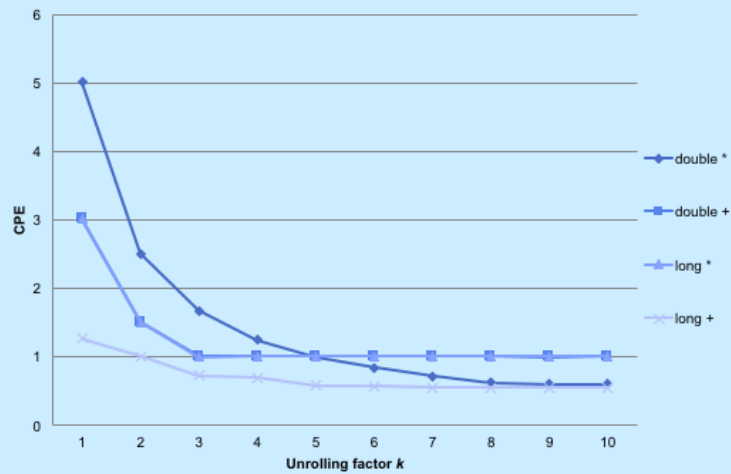
## Quiz 2

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

## 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
Achievable scalar	.54	1.01	1.01	.520
Latency bound	1.00	3.00	3.00	5.00
Throughput bound	.5	1.00	1.00	.5

Supplied by CMU.

## Using Vector Instructions

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Achievable Scalar	.54	1.01	1.01	.520
Latency bound	1.00	3.00	3.00	5.00
Throughput bound	.5	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

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

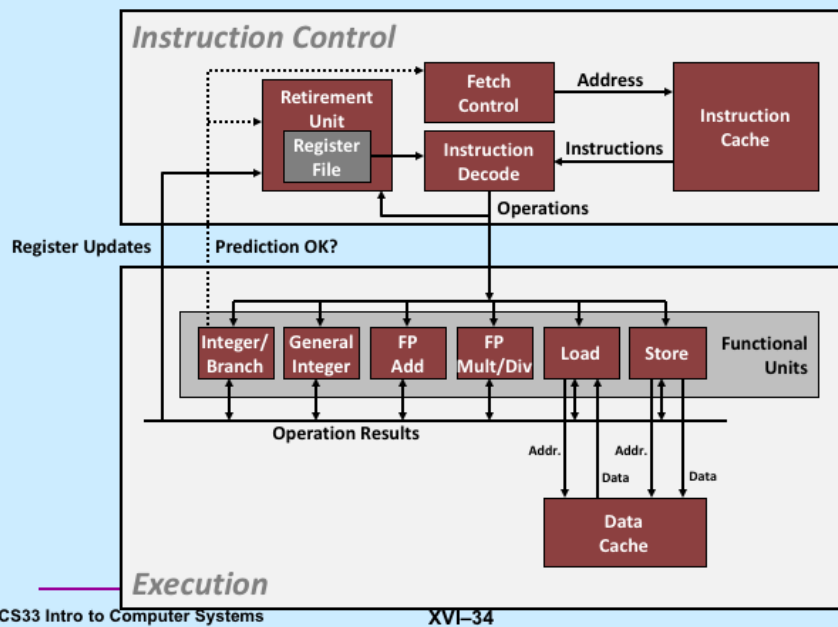
```
80489f3: movl    $0x1,%ecx
80489f8: xorq    %rdx,%rdx
80489fa: cmpq    %rsi,%rdx
80489fc: jnl     8048a25
80489fe: movl    %esi,%edi
8048a00: imull   (%rax,%rdx,4),%ecx
```

} Executing

← How to continue?

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

# 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    0xfffffffffe8(%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
. . .
```

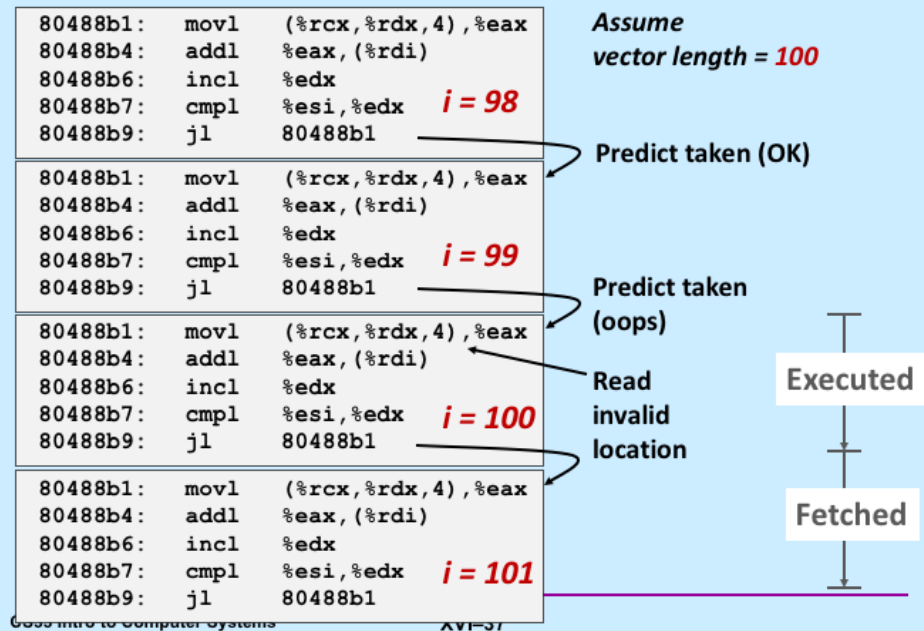
Predict taken

```
8048a25: cmpq    %rdi,%rdx
8048a27: jl      8048a20
8048a29: movl    0xc(%rbp),%eax
8048a2c: leal    0xffffffff8(%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: jl      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

## 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;
        }
    }
}
```

- Compiled code uses conditional branch
  - 13.5 CPE for random data
  - 2.5 – 3.5 CPE for predictable data

```
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 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.



## 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 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

```
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

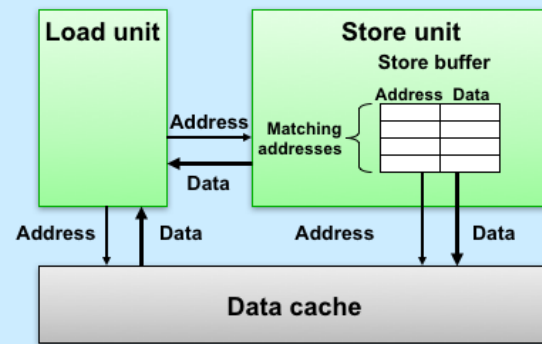
Example B: `write_read(&a[0], &a[0], 3)`

	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

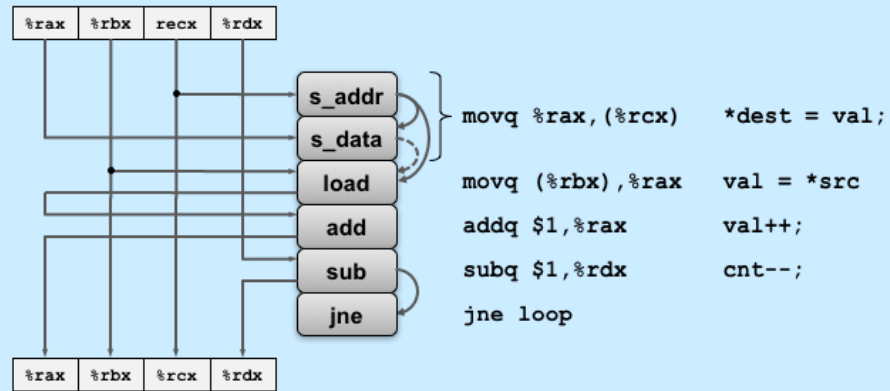
This is Figure 5.33 of the textbook. Performance depends upon whether *src* and *dest* are the same location.

## Some Details of Load and Store



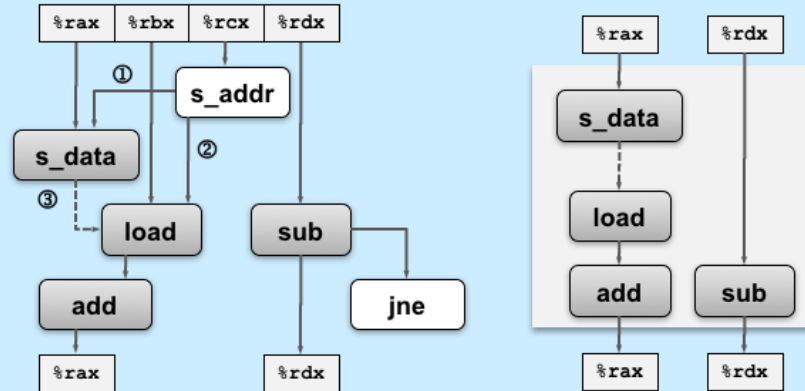
This is Figure 5.34 of the textbook.

## Inner-Loop Data Flow of Write\_Read



This is Figure 5.35 of the textbook.

## Inner-Loop Data Flow of Write\_Read

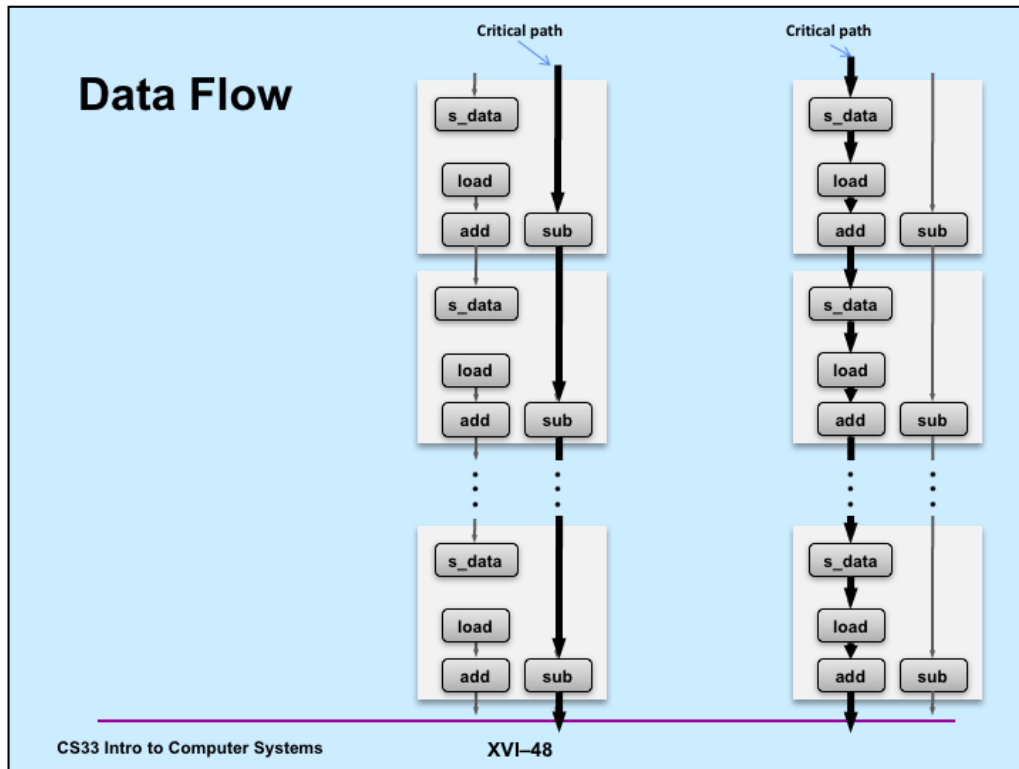


This is Figure 5.36 of the textbook.

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.

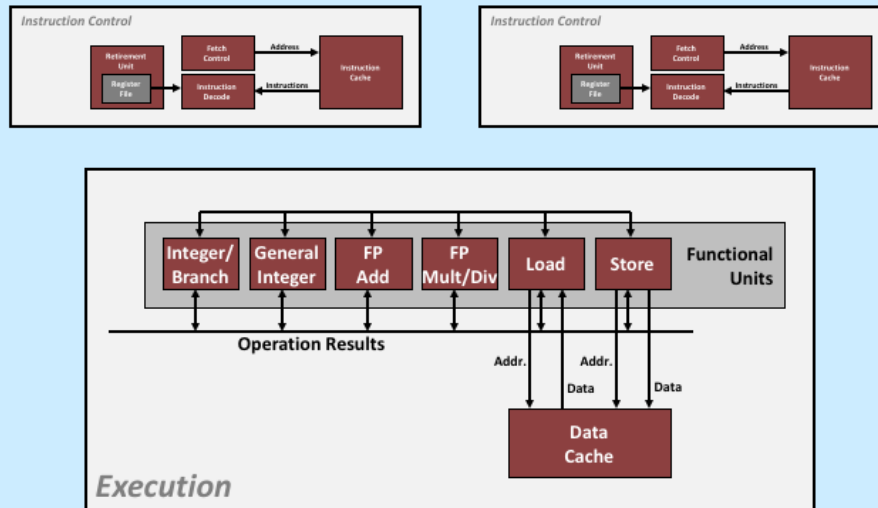


## Getting High Performance

- **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.

# Hyper Threading



# Multiple Cores

## Chip

