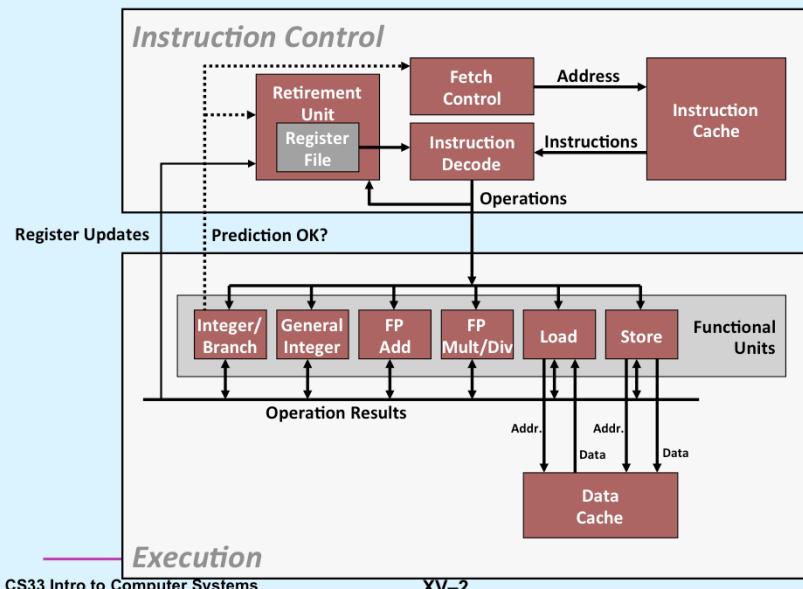


CS 33

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.

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

- **addl %eax, %edx**
 - a single operation
- **addl %eax, 4(%edx)**
 - three operations
 - » load value from memory
 - » add to it the contents of %eax
 - » store result in memory

Instruction-Level Parallelism

- **addl 4(%eax), %eax
addl %ebx, %edx**
 - can be executed simultaneously: completely independent
- **addl 4(%eax), %ebx
addl %ebx, %edx**
 - 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
```

} 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. We will discuss x86 floating point in an upcoming lecture.

Speculative Execution

```
80489f3:    movl    $0x1,%ecx
80489f8:    xorl    %edx,%edx
80489fa:    cmpl    %esi,%edx
80489fc:    jnl     8048a25
80489fe:    movl    %esi,%esi
8048a00:    imull   (%eax,%edx,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**

<i>Instruction</i>	<i>Latency</i>	<i>Cycles/Issue</i>
Load / Store	4	1
Integer Add	1	.33
Integer Multiply	3	1
Integer/Long Divide	11–21	11–21
Single/Double FP Multiply	4/5	1
Single/Double FP Add	3	1
Single/Double FP Divide	10–23	10–23

Supplied by CMU.

“Nehalem” is Intel’s code name for its Core I7 processor design.

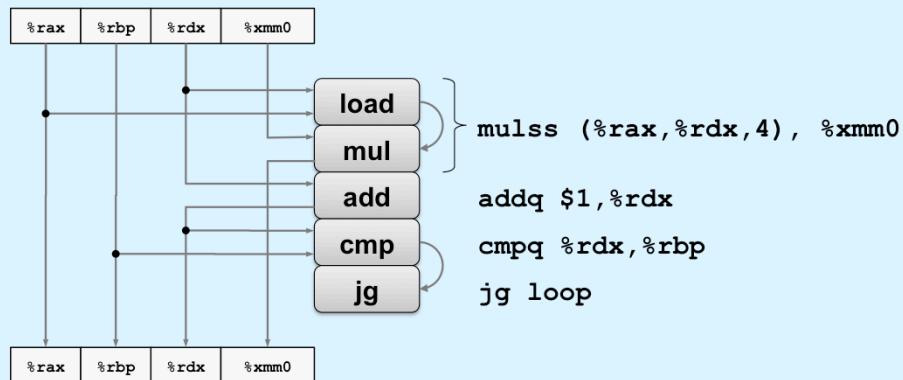
x86-64 Compilation of Combine4

- Inner loop (case: integer multiply)

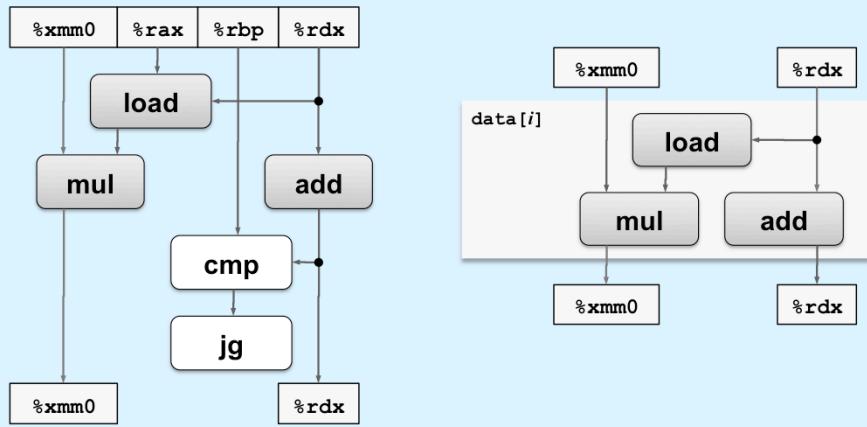
```
.L519:  
    imull  (%rax,%rdx,4), %ecx    # Loop:  
    addq   $1, %rdx               # t = t * d[i]  
    cmpq   %rdx, %rbp             # i++  
    jg     .L519                 # Compare length:i  
                                # If >, goto Loop
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	1.0	1.0	1.0	1.0

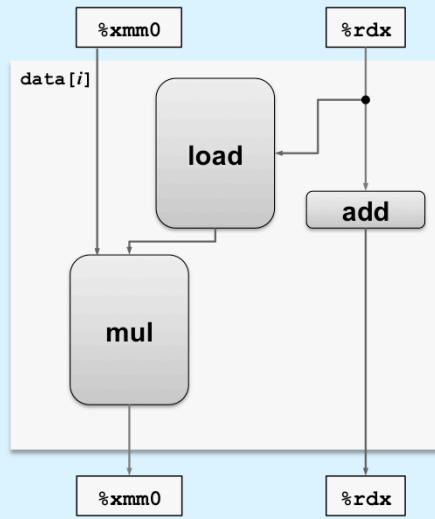
Inner Loop



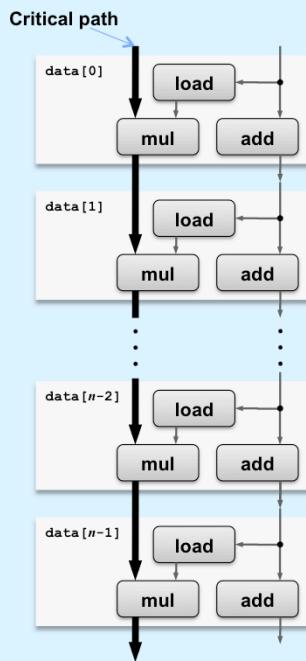
Data-Flow Graphs of Inner Loop



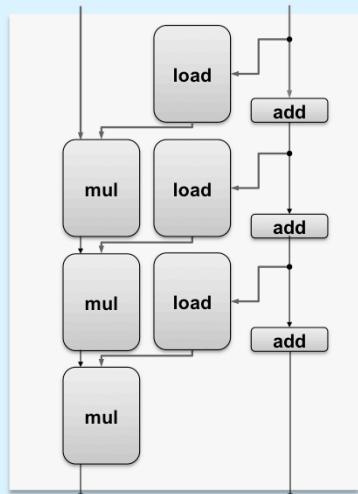
Relative Execution Times



Data Flow Over Multiple Iterations

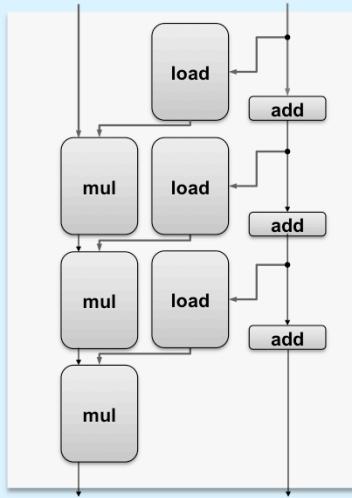


Pipelined Data-Flow Over Multiple Iterations

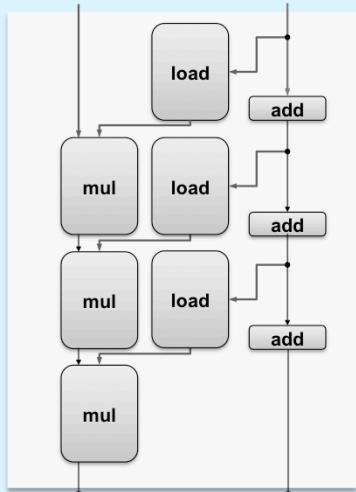


Without pipelining, the data flow would appear as shown in the slide.

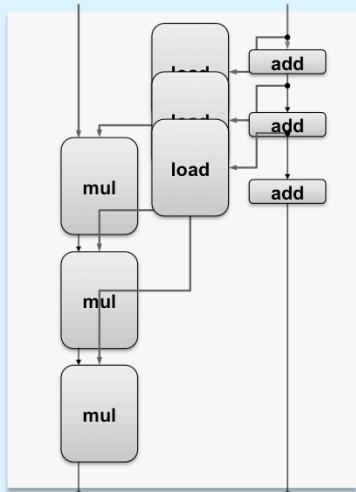
Pipelined Data-Flow Over Multiple Iterations



Pipelined Data-Flow Over Multiple Iterations



Pipelined Data-Flow Over Multiple Iterations



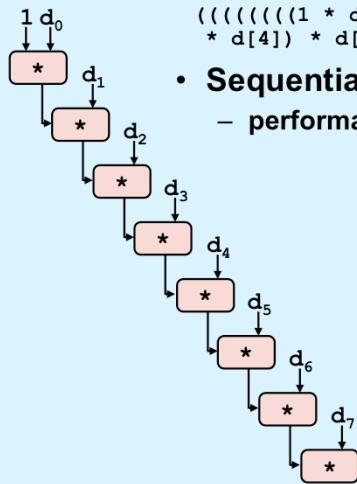
Combine4 = Serial Computation (OP = *)

- **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 multipliers 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

Loop Unrolling

```
void unroll2x(vec_ptr_t v
{
    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

- Perform 2x more useful work per iteration

Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	1.0	1.0	1.0	1.0

- Helps integer multiply
 - below latency bound
 - compiler does clever optimization
- Others don't improve. **Why?**
 - still sequential dependency

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

Supplied by CMU.

What the compiler does for the case of integer multiplication is to apply reassociation, discussed in the next slide.

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	
	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Unroll 2x, reassociate	2.0	1.5	1.5	3.0
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	1.0	1.0	1.0	1.0

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

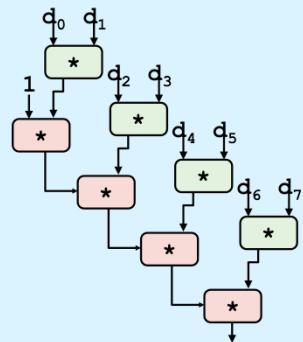
```
x = x OP (d[i] OP d[i+1]);
```

- 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 $(N/2+1) \times D$ cycles:
CPE = $D/2$
 - measured CPE slightly worse for FP mult

Supplied by CMU.

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

Effect of Separate Accumulators

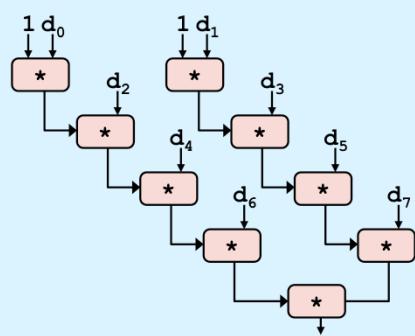
Method	Integer		Double FP	
	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Unroll 2x, reassociate	2.0	1.5	1.5	3.0
Unroll 2x parallel 2x	1.5	1.5	1.5	2.5
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	1.0	1.0	1.0	1.0

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

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

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
 - 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.

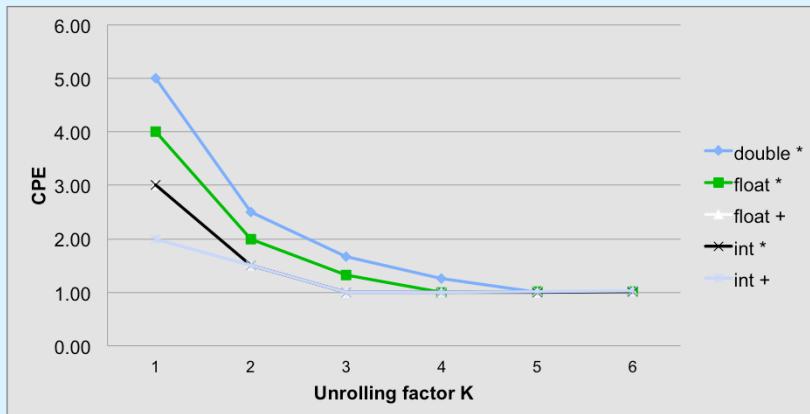
- a) true
- b) false

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

Supplied by CMU.

Performance



- K-way loop unrolling with K accumulators

Achievable Performance

Method	Integer		Double FP	
	Add	Mult	Add	Mult
Scalar optimum	1.00	1.00	1.00	1.00
Latency bound	1.00	3.00	3.00	5.00
Throughput bound	1.00	1.00	1.00	1.00

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

Supplied by CMU.

Using Vector Instructions

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar optimum	1.00	1.00	1.00	1.00
Vector optimum	0.25	0.53	0.53	0.57
Latency bound	1.00	3.00	3.00	5.00
Throughput bound	1.00	1.00	1.00	1.00
Vec throughput bound	0.25	0.50	0.50	0.50

- 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

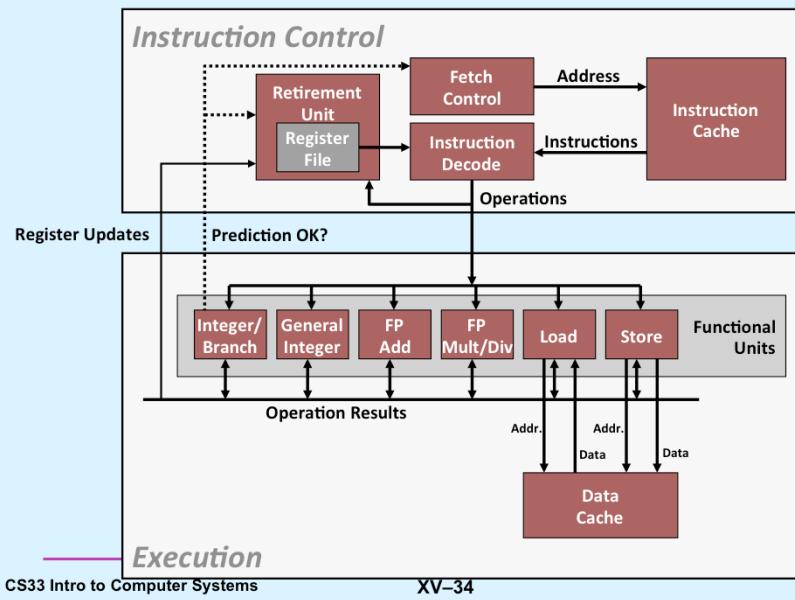
–instruction control unit must work well ahead of execution unit to generate enough operations to keep EU busy

```
80489f3: movl    $0x1,%ecx
80489f8: xorl    %edx,%edx
80489fa: cmpl    %esi,%edx
80489fc: jnl     8048a25 ←
80489fe: movl    %esi,%esi
8048a00: imull   (%eax,%edx,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: xorl    %edx,%edx  
80489fa: cmpl    %esi,%edx  
80489fc: jnl     8048a25  
80489fe: movl    %esi,%esi  
8048a00: imull   (%eax,%edx,4),%ecx
```

Branch not-taken

```
8048a25: cmpl    %edi,%edx  
8048a27: jl      8048a20  
8048a29: movl    0xc(%ebp),%eax  
8048a2c: leal    0xfffffff8(%ebp),%esp  
8048a2f: movl    %ecx,(%eax)
```

Branch taken

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: xorl    %edx,%edx  
80489fa: cmpl    %esi,%edx  
80489fc: jnl     8048a25  
. . .
```

Predict taken

```
8048a25: cmpl    %edi,%edx  
8048a27: jl     8048a20  
8048a29: movl    0xc(%ebp),%eax  
8048a2c: leal    0xffffffe8(%ebp),%esp  
8048a2f: movl    %ecx,(%eax)
```

} Begin execution

Branch Prediction Through Loop

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

Assume
vector length = 100

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

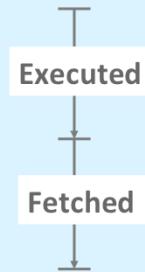
Predict taken (OK)

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

Predict taken
(oops)

Read
invalid
location

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



Supplied by CMU.

Branch Misprediction Invalidation

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

Assume
vector length = 100

Predict taken (OK)

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

Predict taken (oops)

```
80488b1: movi (%ecx,%edx,4),%eax  
80488b4: addi %eax,(%edi)  
80488b6: inci %edx  
80488b7: cmpl %esi,%edx i = 100  
80488b9: jl 80488b1
```

Invalidate

```
80488b1: movi (%ecx,%edx,4),%eax  
80488b4: addi %eax,(%edi)  
80488b6: inci %edx  
80488b7: cmpl %esi,%edx i = 101  
80488b9: jl 80488b1
```

Branch Misprediction Recovery

```
80488b1:    movl    (%ecx,%edx,4),%eax
80488b4:    addl    %eax,(%edi)
80488b6:    incl    %edx
80488b7:    cmpl    %esi,%edx      i = 99
80488b9:    jl     80488b1
80488bb:    leal    0xffffffe8(%ebp),%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(int *a, int *b, int n {
    int i;
    for (i=0; i<n; i++) {
        if (a[i] > b[i]) {
            int t = a[i];
            a[i] = b[i];
            b[i] = t;
        }
    }
}
```

```
void minmax2(int *a, int *b, int n {
    int i;
    for (i=0; i<n; i++) {
        int min = a[i] < b[i]?
                    a[i] : b[i];
        int max = a[i] < b[i]?
                    b[i] : a[i];
        a[i] = min;
        b[i] = max;
    }
}
```

- Compiled code uses conditional branch
 - 14.5 CPE for random data
 - 2.0 – 4.0 CPE for predictable data
- Compiled code uses conditional move instruction
 - 5.0 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;
    int data;
} list_ele, *list_ptr;

int list_len(list_ptr ls) {
    int len = 0;
    while (ls) {
        len++;
        ls = ls->next;
    }
    return len;
}

# len in %eax, ls in %rdi
.L11:           # loop:
    addl $1, %eax      # 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
int main() {
    volatile int dest[100];
    int iter;
    for (iter=0; iter<ITERS; iter++) {
        long i;
        for (i=0; i<100; i++)
            dest[i] = 0;
    }
}
```

Clearing an Array ... Unwound

```
#define ITERS 100000000
int main() {
    volatile int dest[100];
    int iter;
    for (iter=0; iter<ITERS; iter++) {
        long i;
        for (i=0; i<97; i+=4) {
            dest[i] = 0;
            dest[i+1] = 0;
            dest[i+2] = 0;
            dest[i+3] = 0;
        }
    }
}
```

This is adapted from Figure 5.32 of the textbook. By unrolling the loop four times, the stores can be pipelined and thus a store (which takes four cycles to complete) can be started every cycle.

Store/Load Interaction

```
void write_read(int *src, int *dest, int n) {
    int cnt = n;
    int val = 0;

    while(cnt--) {
        *dest = val;
        val = (*src)+1;
    }
}
```

This code is from the textbook.

Store/Load Interaction

```
int 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

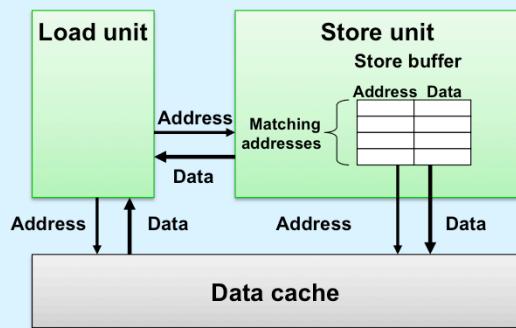
Example B: write_read(&a[0], &a[0], 3)

	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

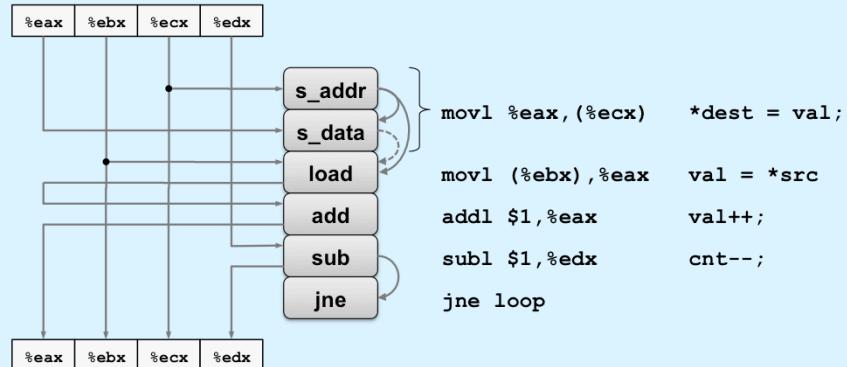
• CPE 2.0

• CPE 6.0

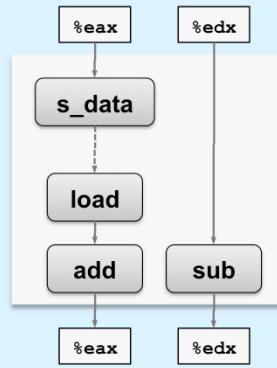
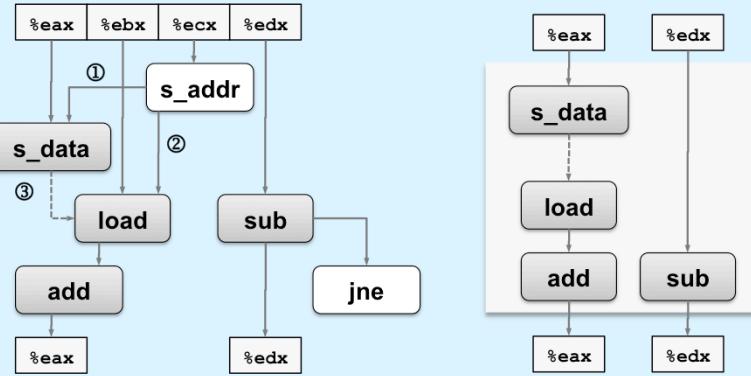
Some Details of Load and Store



Inner-Loop Data Flow of Write_Read



Inner-Loop Data Flow of Write_Read



Data Flow

Critical path

Critical path

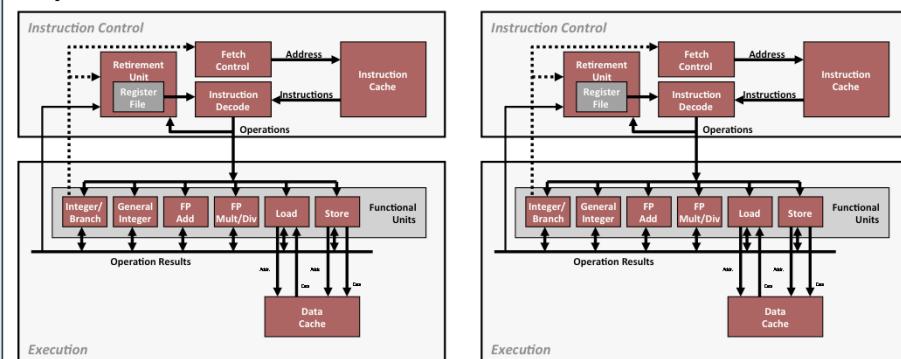
Getting High Performance

- **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.

Multiple Cores

Chip



Other Stuff

More Cache

Other Stuff

Hyper Threading

