

Organisation und Architektur von Rechnern

Lecture 16

Instructor:

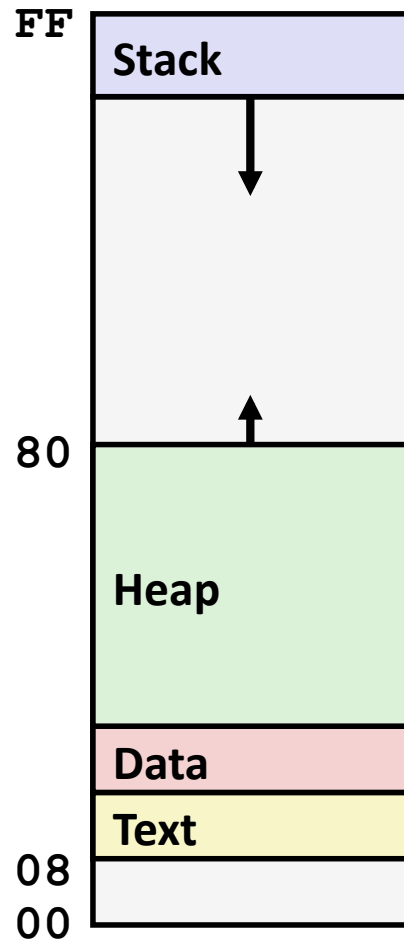
Reinhard v. Hanxleden

<http://www.informatik.uni-kiel.de/rtsys/teaching/v-sysinf2>

These slides are used with kind permission from the Carnegie Mellon University

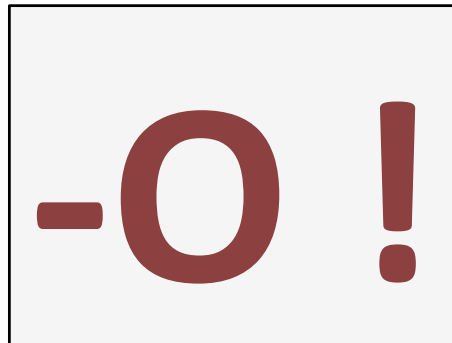
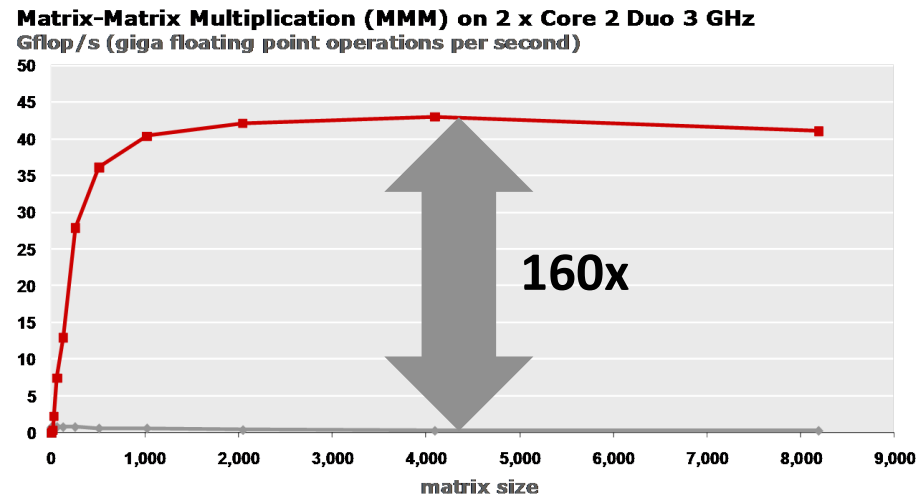
Last Time

■ Memory layout



Last Time

■ Program Optimization



Last Time

■ Program optimization

- Overview
- Removing unnecessary procedure calls
- Code motion/precomputation
- Strength reduction
- Sharing of common subexpressions
- Optimization blocker: Procedure calls

```
for (i = 0; i < n; i++) {  
    get_vec_element(v, i, &val);  
    *res += val;  
}
```

```
void lower(char *s)  
{  
    int i;  
    for (i = 0; i < strlen(s); i++)  
        if (s[i] >= 'A' && s[i] <= 'Z')  
            s[i] -= ('A' - 'a');  
}
```

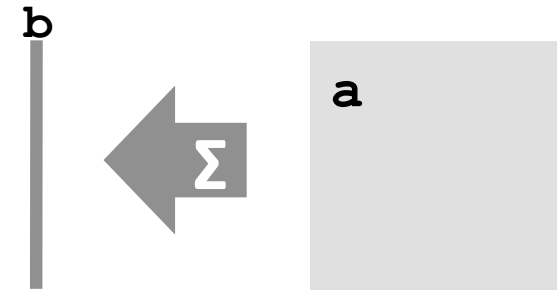
Today

- **Program optimization**

- Optimization blocker: Memory aliasing
- Out of order processing: Instruction level parallelism
- Understanding branch prediction

Optimization Blocker: Memory Aliasing

```
/* Sums rows of n x n matrix a
   and stores in vector b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```



```
# sum_rows1 inner loop
.L53:
    addsd    (%rcx), %xmm0          # FP add
    addq    $8, %rcx
    decq    %rax
    movsd   %xmm0, (%rsi,%r8,8)    # FP store
    jne     .L53
```

- Code updates `b[i]` (= memory access) on every iteration
- Why couldn't compiler optimize this away?

Reason

- If memory is accessed, compiler assumes the possibility of side effects
- Example:

```
/* Sums rows of n x n matrix a
   and stores in vector b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
    { 0,  1,  2,
      4,  8, 16,
      32, 64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

Value of B:

init: [4, 8, 16]

i = 0: [3, 8, 16]

i = 1: [3, 22, 16]

i = 2: [3, 22, 224]

Removing Aliasing

```
/* Sums rows of n x n matrix a
   and stores in vector b */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

```
# sum_rows2 inner loop
.L66:
    addsd    (%rcx), %xmm0    # FP Add
    addq    $8, %rcx
    decq    %rax
    jne     .L66
```

■ Scalar replacement:

- Copy array elements **that are reused** into temporary variables
- 8 ■ Assumes no memory aliasing (otherwise possibly incorrect)

Unaliased Version When Aliasing Happens

```
/* Sum rows is of n X n matrix a
   and store in vector b */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

```
double A[9] =
    { 0,  1,  2,
      4,  8, 16},
    { 32, 64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

Value of B:

init: [4, 8, 16]

i = 0: [3, 8, 16]

i = 1: [3, 27, 16]

i = 2: [3, 27, 224]

- Aliasing still creates interference
- Result different than before

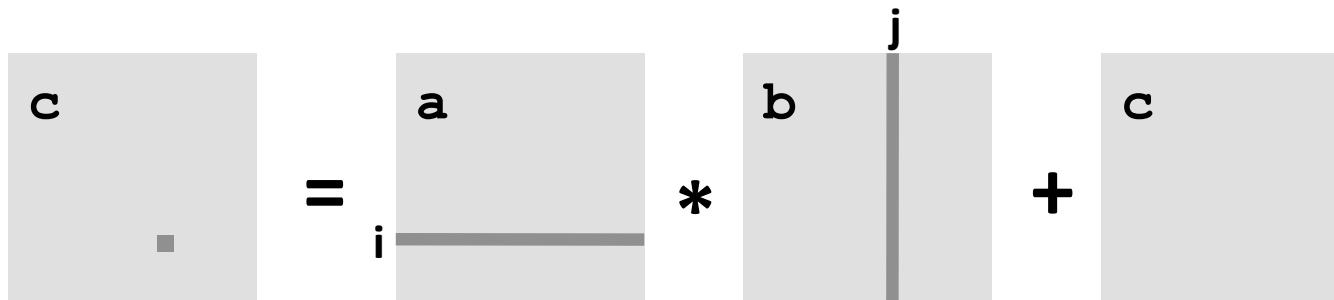
Optimization Blocker: Memory Aliasing

- **Memory aliasing: Two different memory references write to the same location**
- **Easy to have happen in C**
 - Since allowed to do address arithmetic
 - Direct access to storage structures
- **Hard to analyze = compiler cannot figure it out**
 - Hence is conservative
- **Solution: Scalar replacement in innermost loop**
 - Copy memory variables **that are reused** into local variables
 - Basic scheme:
 - **Load:** $t1 = a[i], t2 = b[i+1], \dots$
 - **Compute:** $t4 = t1 * t2; \dots$
 - **Store:** $a[i] = t12, b[i+1] = t7, \dots$

More Difficult Example

■ Matrix multiplication: $C = A * B + C$

```
c = (double *) calloc(sizeof(double), n*n);  
  
/* Multiply n x n matrices a and b */  
void mmm(double *a, double *b, double *c, int n) {  
    int i, j, k;  
    for (i = 0; i < n; i++)  
        for (j = 0; j < n; j++)  
            for (k = 0; k < n; k++)  
                c[i*n+j] += a[i*n + k]*b[k*n + j];  
}
```

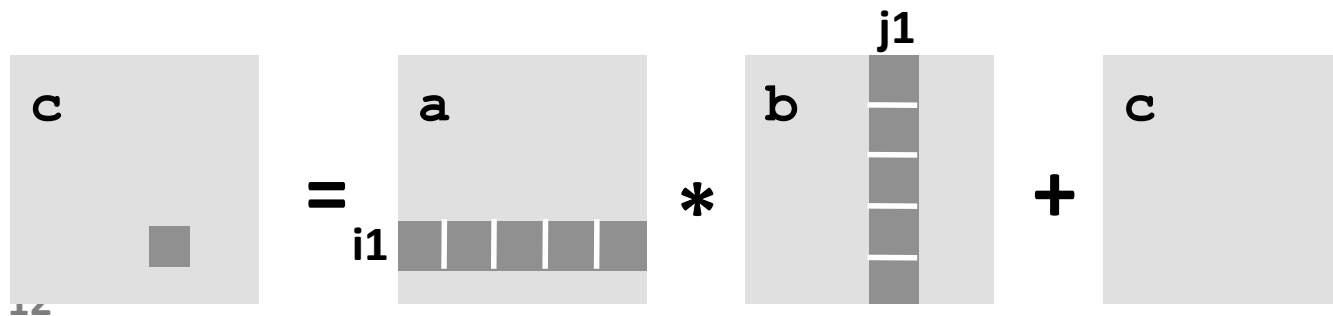


- Which array elements are reused?
- All of them! *But how to take advantage?*

Step 1: Blocking (Here: 2 x 2)

- **Blocking, also called tiling = partial unrolling + loop exchange**
 - Assumes associativity (= compiler will never do it)

```
c = (double *) calloc(sizeof(double), n*n);  
  
/* Multiply n x n matrices a and b */  
void mmm(double *a, double *b, double *c, int n) {  
    int i, j, k;  
    for (i = 0; i < n; i+=2)  
        for (j = 0; j < n; j+=2)  
            for (k = 0; k < n; k+=2)  
                for (i1 = i; i1 < i+2; i1++)  
                    for (j1 = j; j1 < j+2; j1++)  
                        for (k1 = k; k1 < k+2; k1++)  
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];  
}
```



Step 2: Unrolling Inner Loops

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=2)
        for (j = 0; j < n; j+=2)
            for (k = 0; k < n; k+=2)
                <body>
}
```

<body>

```
c[i*n + j]           = a[i*n + k]*b[k*n + j] + a[i*n + k+1]*b[(k+1)*n + j]
                    + c[i*n + j]
c[(i+1)*n + j]       = a[(i+1)*n + k]*b[k*n + j] + a[(i+1)*n + k+1]*b[(k+1)*n + j]
                    + c[(i+1)*n + j]
c[i*n + (j+1)]       = a[i*n + k]*b[k*n + (j+1)] + a[i*n + k+1]*b[(k+1)*n + (j+1)]
                    + c[i*n + (j+1)]
c[(i+1)*n + (j+1)] = a[(i+1)*n + k]*b[k*n + (j+1)]
                    + a[(i+1)*n + k+1]*b[(k+1)*n + (j+1)] + c[(i+1)*n + (j+1)]
```

- Every array element a [...], b [...], c [...] used twice
- Now scalar replacement can be applied

Today

- **Program optimization**

- Optimization blocker: Memory aliasing
- **Out of order processing: Instruction level parallelism**
- Understanding branch prediction

Example: Compute Factorials

```
int rfact(int n)
{
    if (n <= 1)
        return 1;
    return n * rfact(n-1);
}
```

```
int fact(int n)
{
    int i;
    int result = 1;

    for (i = n; i > 0; i--)
        result = result * i;
    return result;
}
```

■ Machines

- Intel Pentium 4 Nocona, 3.2 GHz
- Intel Core 2, 2.7 GHz

■ Compiler Versions

- GCC 3.4.2

Cycles per element (or per mult)

Machine	Nocona	Core 2
rfact	15.5	6.0
fact	10.0	3.0

Something changed from Pentium 4 to Core: Details later

Optimization 1: Loop Unrolling

```
int fact_u3a(int n)
{
    int i;
    int result = 1;

    for (i = n; i >= 3; i-=3) {
        result =
            result * i * (i-1) * (i-2);
    }
    for (; i > 0; i--)
        result *= i;
    return result;
}
```

Cycles per element (or per mult)

Machine	Nocona	Core 2
rfact	15.5	6.0
fact	10.0	3.0
fact_u3a	10.0	3.0

- Compute more values per iteration
- Does not help here
- Why? Branch prediction – details later

Optimization 2: Multiple Accumulators

```
int fact_u3b(int n)
{
    int i;
    int result0 = 1;
    int result1 = 1;
    int result2 = 1;

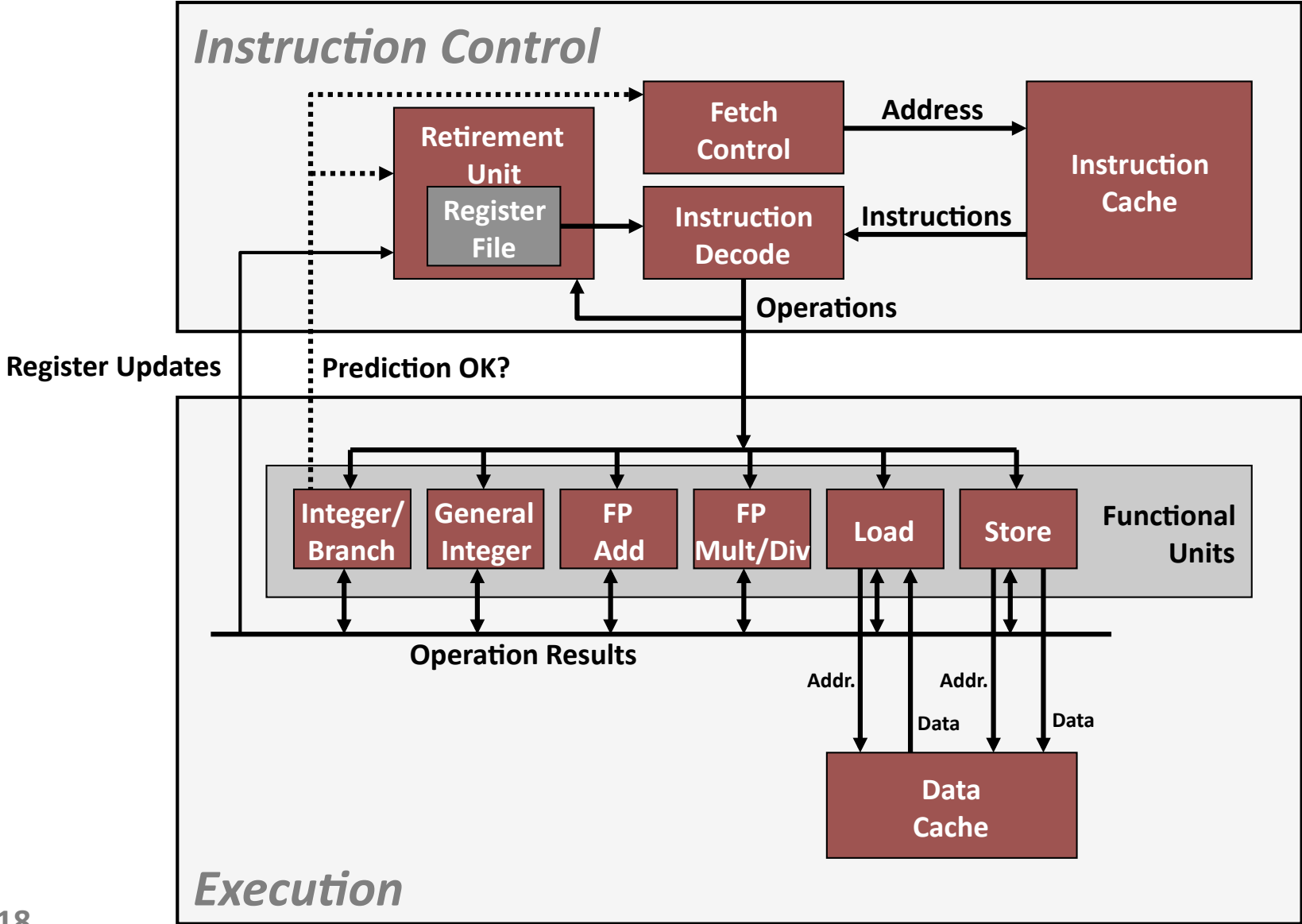
    for (i = n; i >= 3; i-=3) {
        result0 *= i;
        result1 *= (i-1);
        result2 *= (i-2);
    }
    for (; i > 0; i--)
        result0 *= i;
    return result0 * result1 * result2;
}
```

Cycles per element (or per mult)

Machine	Nocona	Core 2
rfact	15.5	6.0
fact	10.0	3.0
fact_u3a	10.0	3.0
fact_u3b	3.3	1.0

- That seems to help. Can one get even faster?
- Explanation: instruction level parallelism – details later

Modern CPU Design



Superscalar Processor

- **Definition:** A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.
- **Benefit:** without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have
- Most CPUs since about 1998 are superscalar.
- Intel: since Pentium Pro

Pentium 4 Nocona 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/SSE3 unit
- 1 FP move (does all conversions)

■ Some instructions take > 1 cycle, but can be pipelined

<i>Instruction</i>	<i>Latency</i>	<i>Cycles/Issue</i>
Load / Store	5	1
Integer Multiply	10	1
Integer/Long Divide	36/106	36/106
Single/Double FP Multiply	7	2
Single/Double FP Add	5	2
Single/Double FP Divide	32/46	32/46

Latency versus Throughput

■ Last slide:

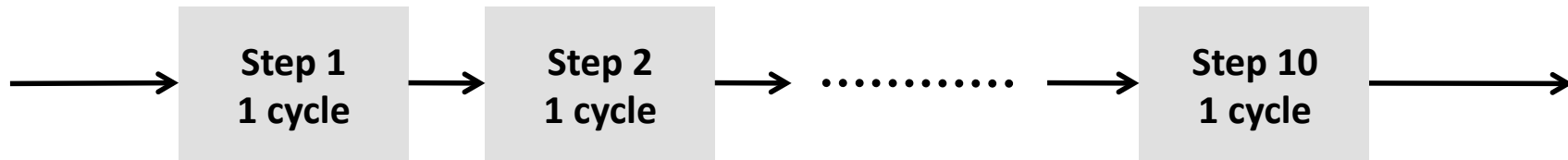
Integer Multiply

latency

10

cycles/issue

1



■ Consequence:

- How fast can 10 independent int mults be executed?

$t1 = t2*t3; t4 = t5*t6; \dots$

- How fast can 10 sequentially dependent int mults be executed?

$t1 = t2*t3; t4 = t5*t1; t6 = t7*t4; \dots$

- Major problem for fast execution: **Keep pipelines filled**

Hard Bounds

■ Latency and throughput of instructions

<i>Instruction</i>	<i>Latency</i>	<i>Cycles/Issue</i>
Load / Store	5	1
Integer Multiply	10	1
Integer/Long Divide	36/106	36/106
Single/Double FP Multiply	7	2
Single/Double FP Add	5	2
Single/Double FP Divide	32/46	32/46

■ How many cycles at least if

- Function requires n int mults?
- Function requires n float adds?
- Function requires n float ops (adds and mults)?

Performance in Numerical Computing

- Numerical computing =
computing dominated by floating point operations
- Example: Matrix multiplication
- Performance measure:
Floating point operations per second (flop/s)
 - Counting only floating point adds and mults
 - Higher is better
 - Like inverse runtime
- Theoretical scalar (no vector SSE) peak performance of
Nocona (3.2 GHz)?
 - 3.2 Gflop/s = 3200 Mflop/s. *Why?*

Nocona vs. Core 2

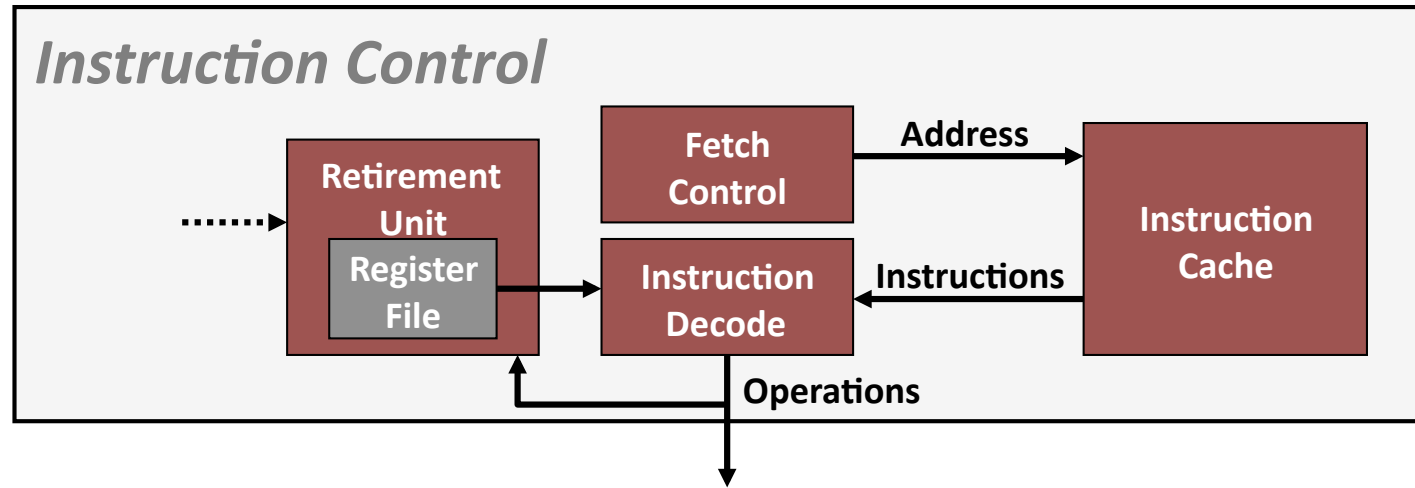
■ Nocona (3.2 GHz)

<i>Instruction</i>	<i>Latency</i>	<i>Cycles/Issue</i>
Load / Store	5	1
Integer Multiply	10	1
Integer/Long Divide	36/106	36/106
Single/Double FP Multiply	7	2
Single/Double FP Add	5	2
Single/Double FP Divide	32/46	32/46

■ Core 2 (2.7 GHz) (Recent Intel microprocessors)

<i>Instruction</i>	<i>Latency</i>	<i>Cycles/Issue</i>
Load / Store	5	1
Integer Multiply	3	1
Integer/Long Divide	18/50	18/50
Single/Double FP Multiply	4/5	1
Single/Double FP Add	3	1
Single/Double FP Divide	18/32	18/32

Instruction Control



- **Grabs instruction bytes from memory**
 - Based on current PC + predicted targets for predicted branches
 - Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target
- **Translates instructions into *micro-operations* (for CISC style CPUs)**
 - Micro-op = primitive step required to perform instruction
 - Typical instruction requires 1–3 operations
- **Converts register references into *tags***
 - Abstract identifier linking destination of one operation with sources of later operations

Translating into Micro-Operations

```
imulq %rax, 8(%rbx,%rdx,4)
```

- **Goal: Each operation utilizes single functional unit**

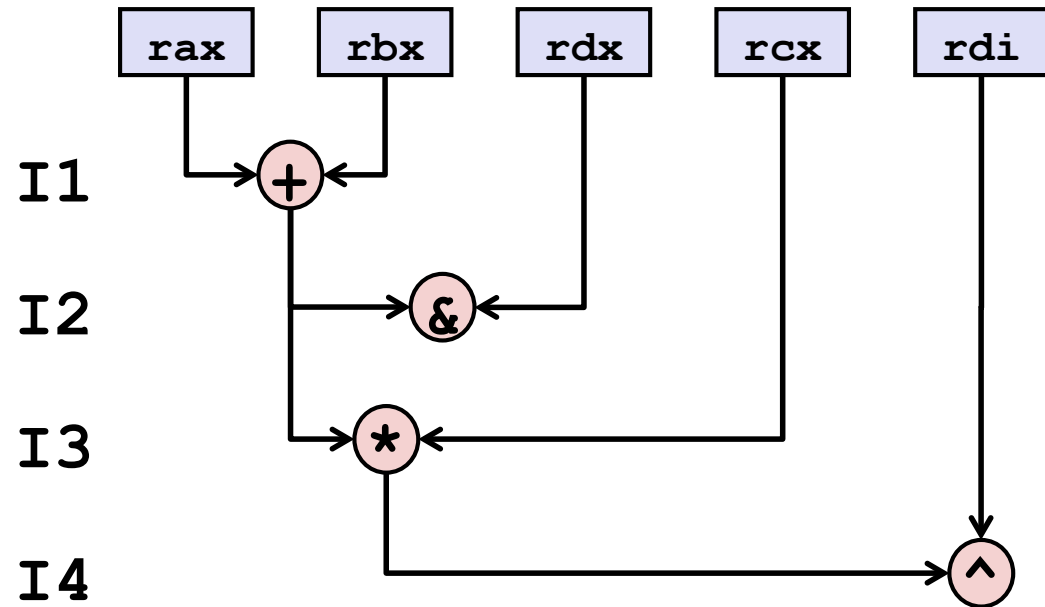
- Requires: Load, integer arithmetic, store

```
load 8(%rbx,%rdx,4)      →   temp1
imulq %rax, temp1        →   temp2
store temp2, 8(%rbx,%rdx,4)
```

- Exact form and format of operations is trade secret

Traditional View of Instruction Execution

```
addq %rax, %rbx # I1
andq %rbx, %rdx # I2
mulq %rcx, %rbx # I3
xorq %rbx, %rdi # I4
```

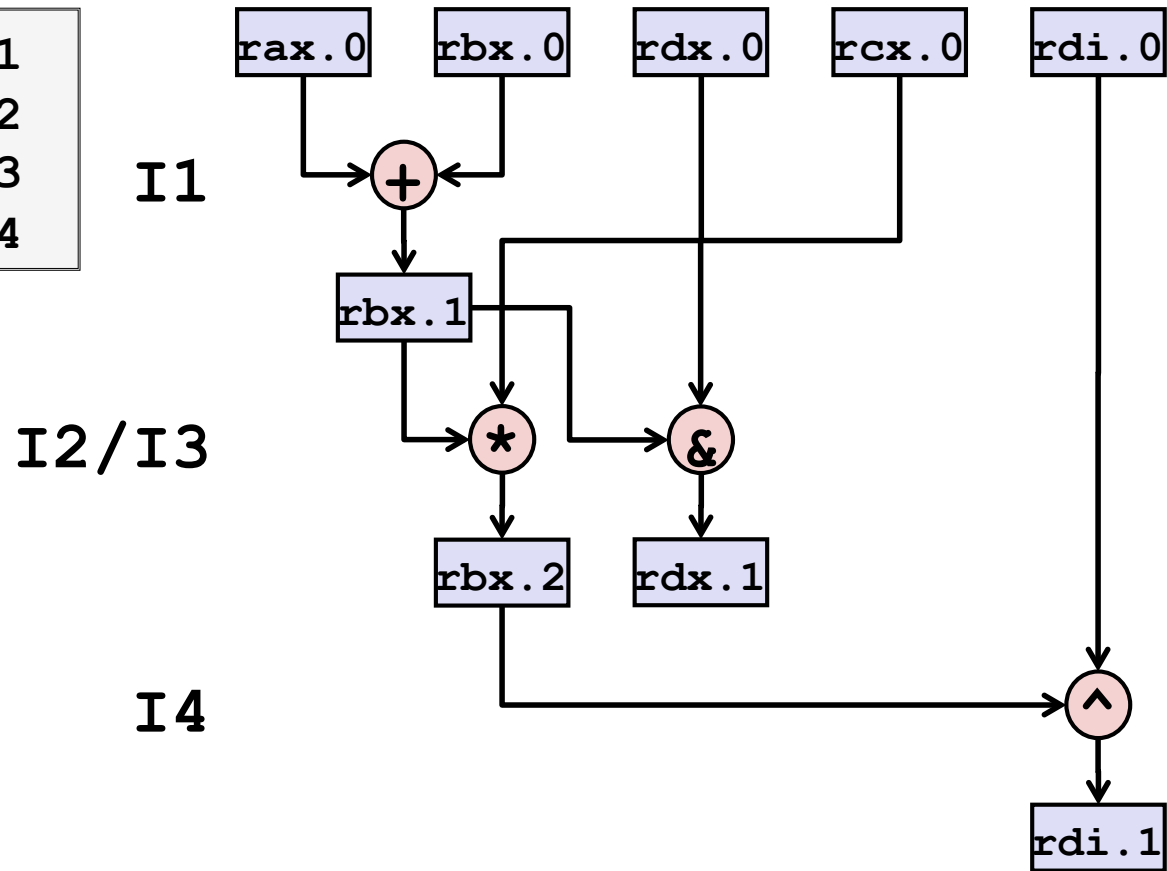


■ Imperative View

- Registers are fixed storage locations
 - Individual instructions read & write them
- Instructions must be executed in specified sequence to guarantee proper program behavior

Dataflow View of Instruction Execution

```
addq %rax, %rbx # I1
andq %rbx, %rdx # I2
mulq %rcx, %rbx # I3
xorq %rbx, %rdi # I4
```



■ Functional View

- View each write as creating new instance of value
- Operations can be performed as soon as operands available
- No need to execute in original sequence

Example Computation

```
void combine4(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

d[1] OP d[2] OP d[3] OP ... OP d[length-1]

■ Data Types

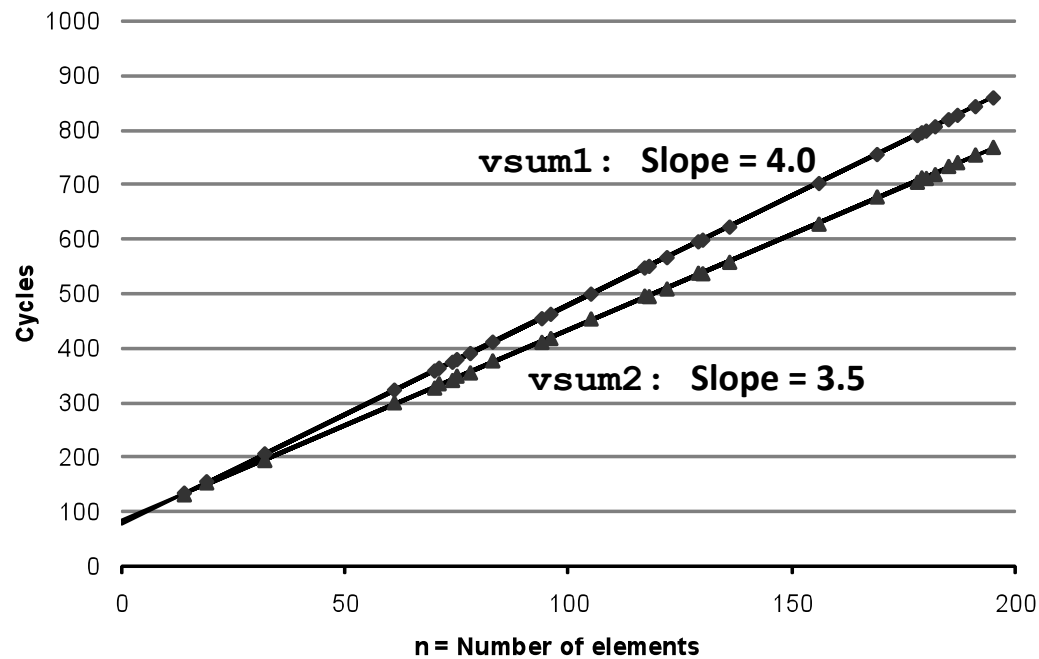
- Use different declarations for **data_t**
- **int**
- **float**
- **double**

■ Operations

- Use different definitions of **OP** and **IDENT**
- **+ / 0**
- *** / 1**

Cycles Per Element (CPE)

- Convenient way to express performance of program that operators on vectors or lists
- Length = n
- In our case: **CPE = cycles per OP** (gives hard lower bound)
- $T = \text{CPE} * n + \text{Overhead}$



x86-64 Compilation of Combine4

```
void combine4(vec_ptr v,
             data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

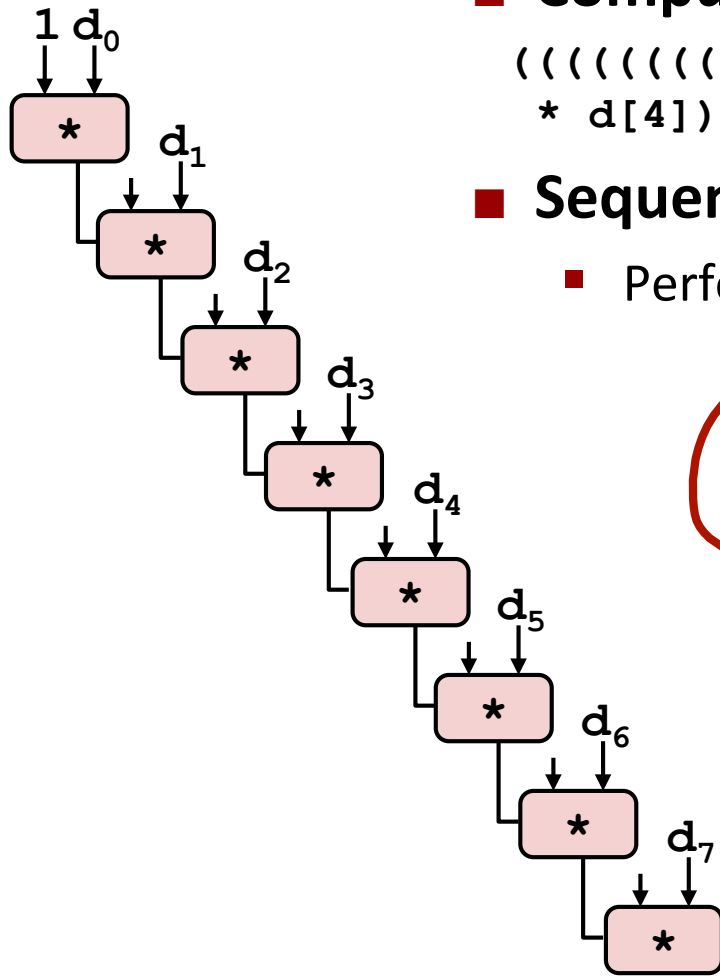
Cycles per element (or per OP)

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
bound	1.0	1.0	2.0	2.0

■ Inner Loop (Case: Integer Multiply)

```
L33:                                # Loop:
    movl  (%eax,%edx,4), %ebx        # temp = d[i]
    incl  %edx                       # i++
    imull %ebx, %ecx                 # x *= temp
    cmpl  %esi, %edx                 # i:length
    jl    L33                        # if < goto Loop
```

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! Hence,**

- Performance: determined by latency of OP!

Cycles per element (or per OP)

Method	Int (add/mult)	Float (add/mult)
combine4	2.2	10.0
bound	1.0	1.0

Loop Unrolling

```
void unroll2a_combine(vec_ptr 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

Effect of Loop Unrolling

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
bound	1.0	1.0	2.0	2.0

- Helps integer sum
- Others don't improve. *Why?*
 - Still sequential dependency

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

Loop Unrolling with Reassociation

```
void unroll2aa_combine(vec_ptr 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?

3 ■ Yes, for FP. *Why?*

Effect of Reassociation

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
unroll2-ra	1.56	5.0	2.75	3.62
bound	1.0	1.0	2.0	2.0

■ Nearly 2x speedup for Int *, FP +, FP *

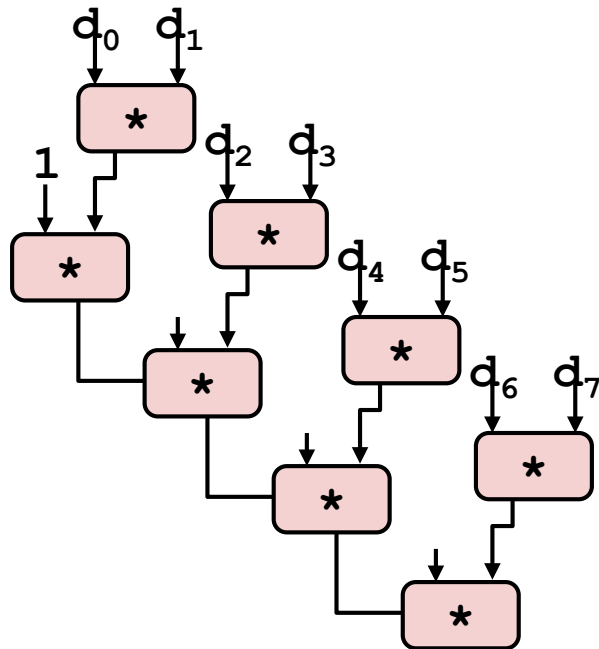
- Reason: Breaks sequential dependency

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

- Why is that? (next slide)

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 FP

Loop Unrolling with Separate Accumulators

```
void unroll2a_combine(vec_ptr 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

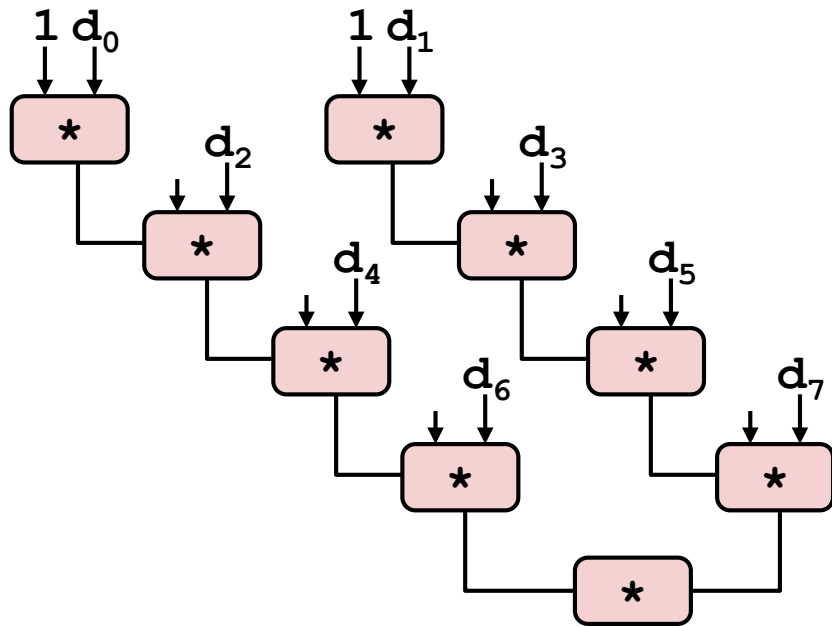
Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
unroll2-ra	1.56	5.0	2.75	3.62
unroll2-sa	1.50	5.0	2.5	3.5
bound	1.0	1.0	2.0	2.0

- **Almost exact 2x speedup (over unroll2) 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?

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

Unrolling & Accumulating: Intel FP *

■ Case

- Intel Nocona
- FP Multiplication
- Theoretical Limit: 2.00

Accumulators	FP *	Unrolling Factor L							
	K	1	2	3	4	6	8	10	12
	1	7.00	7.00		7.01		7.00		
	2		3.50		3.50		3.50		
	3			2.34					
	4				2.01		2.00		
	6					2.00			2.01
	8						2.01		
	10							2.00	
	12								2.00

Unrolling & Accumulating: Intel FP +

■ Case

- Intel Nocona
- FP Addition
- Theoretical Limit: 2.00

FP +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	5.00	5.00		5.02		5.00		
2		2.50		2.51		2.51		
3			2.00					
4				2.01		2.00		
6					2.00			1.99
8						2.01		
10							2.00	
12								2.00

Unrolling & Accumulating: Intel Int *

■ Case

- Intel Nocona
- Integer Multiplication
- Theoretical Limit: 1.00

Int *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	10.00	10.00		10.00		10.01		
2		5.00		5.01		5.00		
3			3.33					
4				2.50		2.51		
6					1.67			1.67
8						1.25		
10							1.09	
12								1.14

Unrolling & Accumulating: Intel Int +

■ Case

- Intel Nocona
- Integer addition
- Theoretical Limit: 1.00 (unrolling enough)

Int +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	2.20	1.50		1.10		1.03		
2		1.50		1.10		1.03		
3			1.34					
4				1.09		1.03		
6					1.01			1.01
8						1.03		
10							1.04	
12								1.11

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	7.00	7.00		7.01		7.00		
2		3.50		3.50		3.50		
3			2.34					
4				2.01		2.00		
6					2.00			2.01
8						2.01		
10							2.00	
12								2.00

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	4.00	4.00		4.00		4.01		
2		2.00		2.00		2.00		
3			1.34					
4				1.00		1.00		
6					1.00			1.00
8						1.00		
10							1.00	
12								1.00

FP *: Nocona versus Core 2

■ Machines

- Intel Nocona
 - 3.2 GHz
- Intel Core 2
 - 2.7 GHz

■ Performance

- Core 2 lower latency
& fully pipelined
(1 cycle/issue)

Can We Go Faster?

- Yes, SSE!
 - But not in this class 😊

Today

- **Program optimization**

- Optimization blocker: Memory aliasing
- Out of order processing: Instruction level parallelism
- **Understanding branch prediction**

What About Branches?

■ Challenge

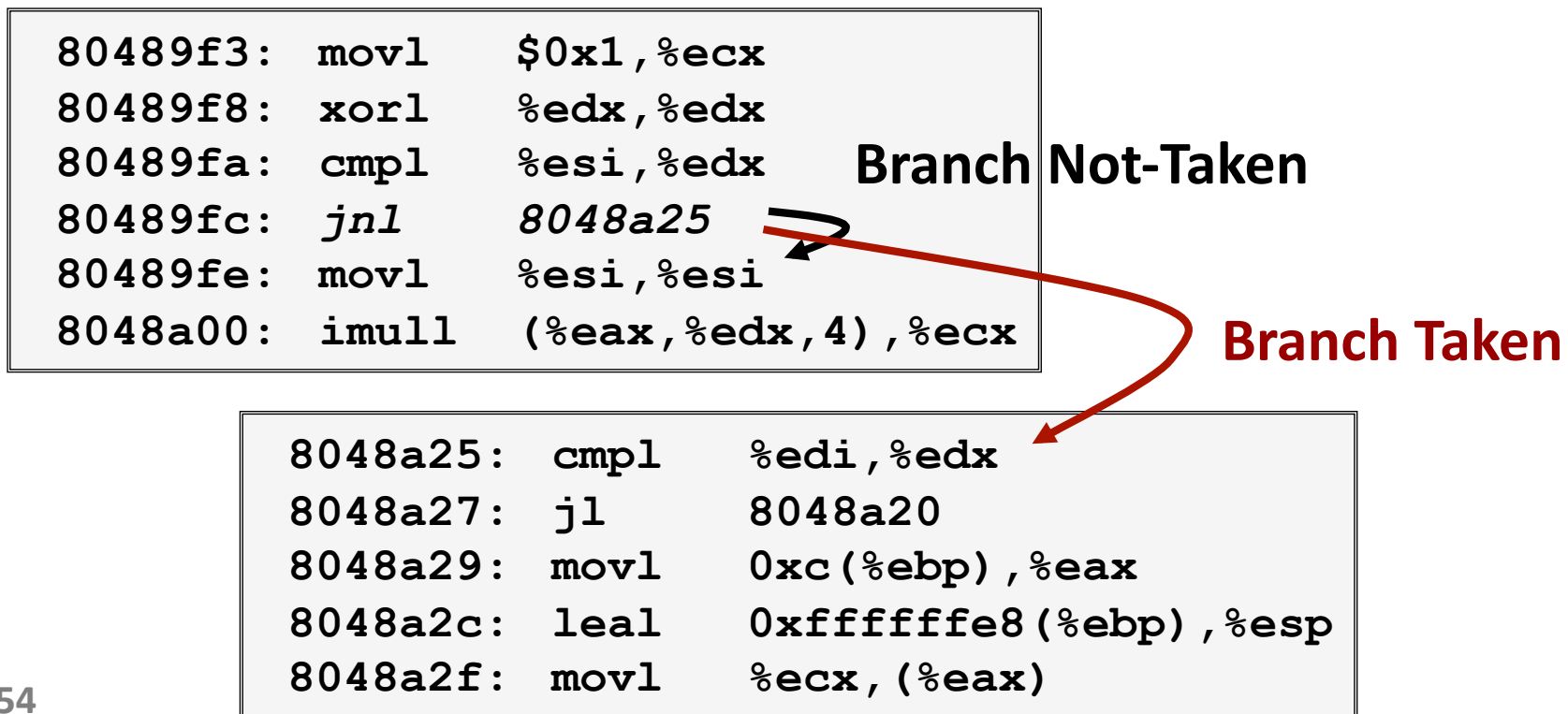
- **Instruction Control Unit** must work well ahead of **Execution Unit** to generate enough operations to keep EU busy

80489f3:	movl	\$0x1, %ecx	} Executing
80489f8:	xorl	%edx, %edx	
80489fa:	cmpl	%esi, %edx	
80489fc:	jnl	8048a25 ← How to continue?	
80489fe:	movl	%esi, %esi	
8048a00:	imull	(%eax, %edx, 4), %ecx	

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

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



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   0xffffffff(%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**

Predict Taken (OK)

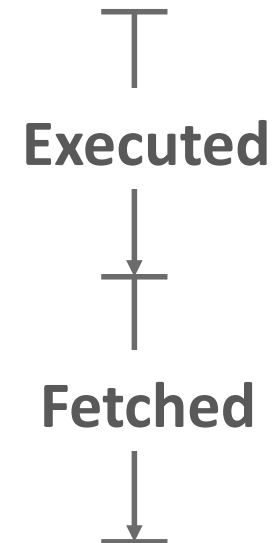
```
80488b1:  movl  (%ecx,%edx,4),%eax
80488b4:  addl  %eax,(%edi)
80488b6:  incl  %edx
80488b7:  cmpl  %esi,%edx  i = 99
80488b9:  jl    80488b1
```

Predict Taken
(Oops)

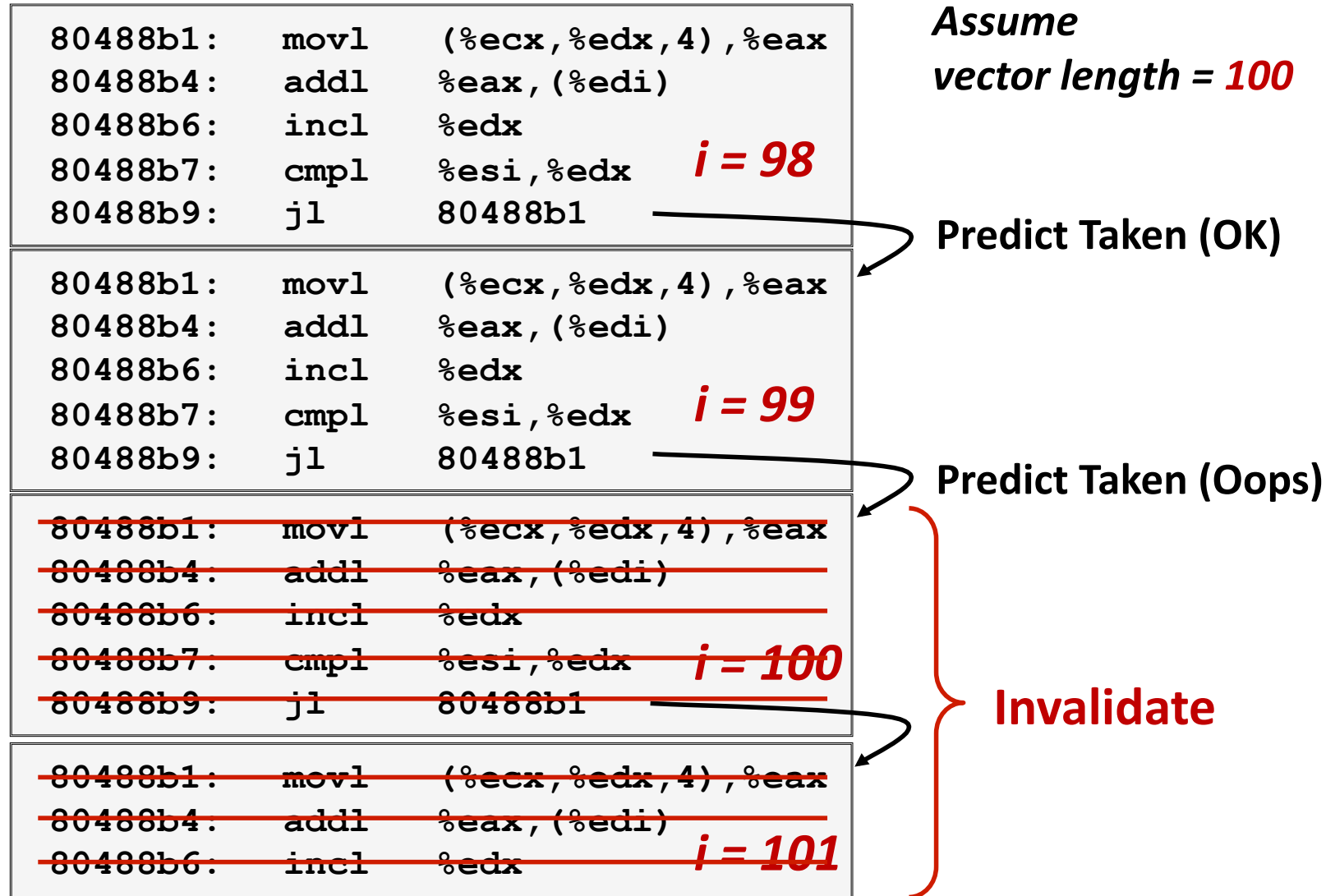
```
80488b1:  movl  (%ecx,%edx,4),%eax
80488b4:  addl  %eax,(%edi)
80488b6:  incl  %edx
80488b7:  cmpl  %esi,%edx  i = 100
80488b9:  jl    80488b1
```

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
```



Branch Misprediction Invalidation




Branch Misprediction Recovery

```
80488b1:  movl    (%ecx,%edx,4),%eax
80488b4:  addl    %eax,(%edi)
80488b6:  incl    %edx
80488b7:  cmpl   %esi,%edx
80488b9:  jl      80488b1
80488bb:  leal   0xffffffe8(%ebp),%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

Determining Misprediction Penalty

```
int cnt_gt = 0;
int cnt_le = 0;
int cnt_all = 0;

int choose_cmov(int x, int y)
{
    int result;
    if (x > y) {
        result = cnt_gt;
    } else {
        result = cnt_le;
    }
    ++cnt_all;
    return result;
}
```

- **GCC/x86-64 tries to minimize use of Branches**
 - Generates conditional moves when possible/sensible

```
choose_cmov:
    cmpl    %esi, %edi        # x:y
    movl   cnt_le(%rip), %eax # r = cnt_le
    cmovg  cnt_gt(%rip), %eax # if >= r=cnt_gt
    incl   cnt_all(%rip)     # cnt_all++
    ret                                # return r
```

Forcing Conditional

```
int cnt_gt = 0;
int cnt_le = 0;

int choose_cond(int x, int y)
{
    int result;
    if (x > y) {
        result = ++cnt_gt;
    } else {
        result = ++cnt_le;
    }
    return result;
}
```

- Cannot use conditional move when either outcome has side effect

```
choose_cond:
    cmpl  %esi, %edi
    jle   .L8
    movl  cnt_gt(%rip), %eax
    incl  %eax
    movl  %eax, cnt_gt(%rip)
    ret
.L8:
    movl  cnt_le(%rip), %eax
    incl  %eax
    movl  %eax, cnt_le(%rip)
    ret
```

If

Then

Else

Testing Methodology

■ Idea

- Measure procedure under two different prediction probabilities
 - $P = 1.0$: Perfect prediction
 - $P = 0.5$: Random data

■ Test Data

- $x = 0, y = \pm 1$

Case +1: $y = [+1, +1, +1, \dots, +1, +1]$

Case -1: $y = [-1, -1, -1, \dots, -1, -1]$

Case A: $y = [+1, -1, +1, \dots, +1, -1]$ (alternate)

Case R: $y = [+1, -1, -1, \dots, -1, +1]$ (random)

Testing Outcomes

Intel Nocona

Case	cmov	cond
+1	12.3	18.2
-1	12.3	12.2
A	12.3	15.2
R	12.3	31.2

AMD Opteron

Case	cmov	cond
+1	8.05	10.1
-1	8.05	8.1
A	8.05	9.2
R	8.05	15.7

Intel Core 2

Case	cmov	cond
+1	7.17	9.2
-1	7.17	8.2
A	7.17	8.7
R	7.17	17.7

■ Observations:

- Conditional move insensitive to data
- Perfect prediction for regular patterns
 - Else case requires 6 (Nocona), 2 (AMD), or 1 (Core 2) extra cycles
 - Averages to 15.2
- Branch penalties: (for R, processor will get it right half of the time)
 - Nocona: $2 * (31.2 - 15.2) = 32$ cycles
 - AMD: $2 * (15.7 - 9.2) = 13$ cycles
 - Core 2: $2 * (17.7 - 8.7) = 18$ cycles

Getting High Performance So Far

- **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
 - Careful with implemented abstract data types
 - 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 later in course)