

15-213

“The course that gives CMU its Zip!”

Code Optimization II

Dec. 2, 2008

Topics

- Machine Dependent Optimizations
 - Understanding Processor Operations
 - Branches and Branch Prediction

Motivate: 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
 - Fish Machines
- Intel Core 2, 2.7 GHz

Compiler Versions

- GCC 3.4.2 (current on Fish machines)
- GCC 4.1.2 (most recent available)

Cycles Per Element

| Machine | Nocona | Core 2 | |
|----------|--------|--------|-----|
| Compiler | 3.4 | 3.4 | 4.1 |
| rfact | 15.5 | 6.0 | 3.0 |
| fact | 10.0 | 3.0 | 3.0 |

Faster Versions (1)

```
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

| Machine | Nocona | Core 2 | |
|----------|--------|--------|-----|
| | | | |
| Compiler | 3.4 | 3.4 | 4.1 |
| rfact | 15.5 | 6.0 | 3.0 |
| fact | 10.0 | 3.0 | 3.0 |
| fact_u3a | 10.0 | 3.0 | 3.0 |

Loop Unrolling

- Compute more values per iteration
- Does not help here

Faster Versions (2)

```
int fact_u3b(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

| Machine | Nocona | Core 2 | |
|----------|--------|--------|--------|
| | | Core 2 | Core 2 |
| Compiler | 3.4 | 3.4 | 4.1 |
| rfact | 15.5 | 6.0 | 3.0 |
| fact | 10.0 | 3.0 | 3.0 |
| fact_u3a | 10.0 | 3.0 | 3.0 |
| fact_u3b | 3.3 | 1.0 | 3.0 |

Loop Unrolling + Reassociation

- ~3X drop for GCC 3.4
- No improvement for GCC 4.1
 - Very strange!

Faster Versions (3)

```
int fact_u3c(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

| Machine | Nocona | Core 2 | |
|----------|--------|--------|-----|
| Compiler | 3.4 | 3.4 | 4.1 |
| rfact | 15.5 | 6.0 | 3.0 |
| fact | 10.0 | 3.0 | 3.0 |
| fact_u3a | 10.0 | 3.0 | 3.0 |
| fact_u3b | 3.3 | 1.0 | 3.0 |
| fact_u3c | 3.3 | 1.0 | 1.0 |

Loop Unrolling + Multiple Accumulators

- ~3X drop for all machines

Getting High Performance

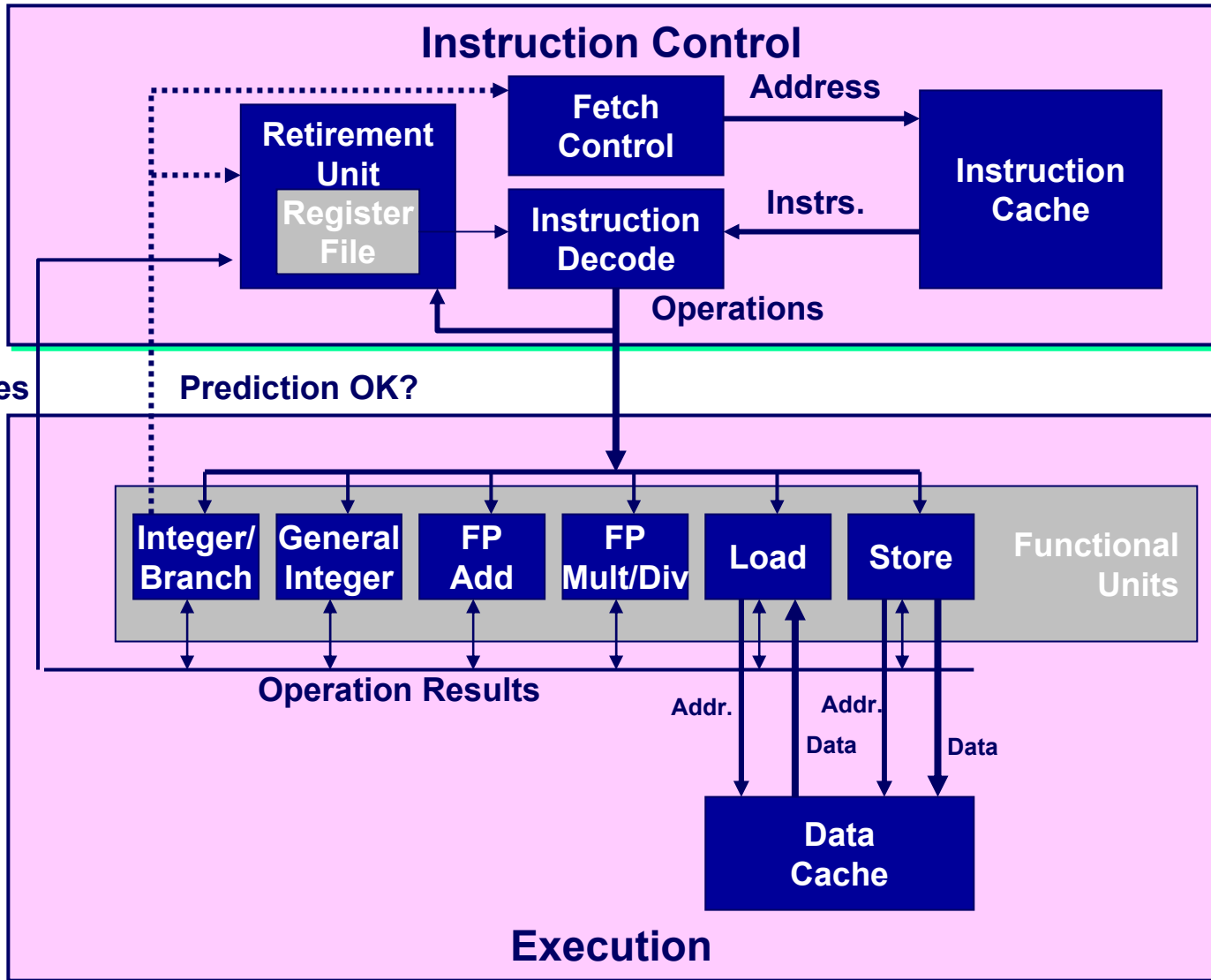
Don't Do Anything Stupid

- Watch out for hidden algorithmic inefficiencies
- Write compiler-friendly code
 - Help compiler past optimization blockers: function calls & memory refs.

Tune Code For Machine

- Exploit instruction-level parallelism
- Avoid unpredictable branches
- Make code cache friendly
 - Covered later in course

Modern CPU Design



Pentium IV 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

| Instruction | Latency | Cycles/Issue |
|-----------------------------|---------|--------------|
| ■ 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 |

CPUs: Nocona vs. Core 2

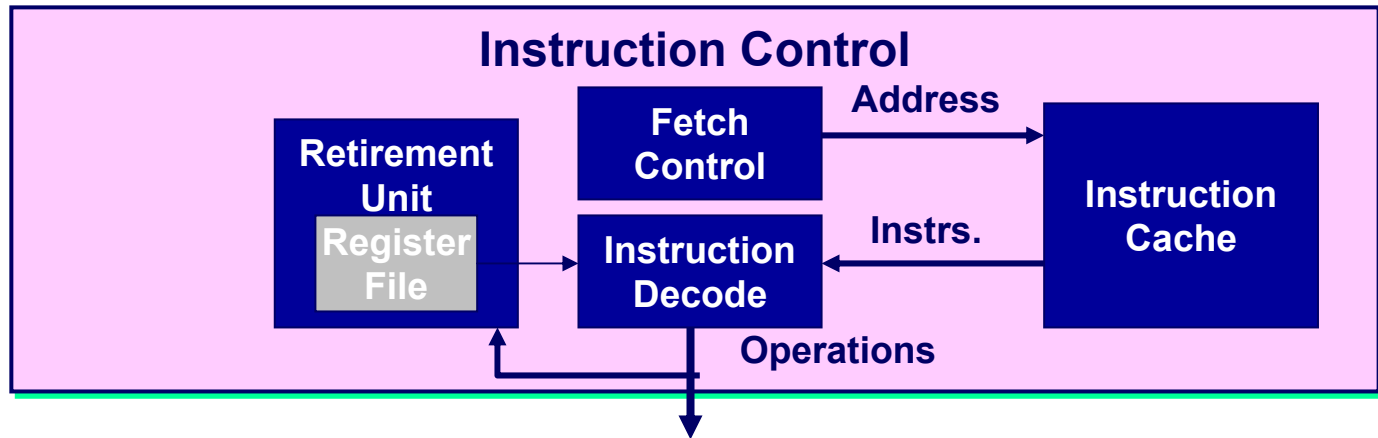
Nocona (3.2 GHz) (Saltwater fish machines)

| Instruction | Latency | Cycles/Issue |
|-----------------------------|---------|--------------|
| ■ Load / Store | 10 | 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)

| | | |
|-----------------------------|-------|-------|
| ■ Load / Store | 5 | 1 |
| ■ Integer Multiply | 3 | 1 |
| ■ Integer/Long Divide | 18/50 | 18/50 |
| ■ Single/Double FP Multiply | 4 | 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 *Operations* (for CISC style CPUs)

- Primitive steps 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 Operations

Goal: Each Operation Utilizes Single Functional Unit

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

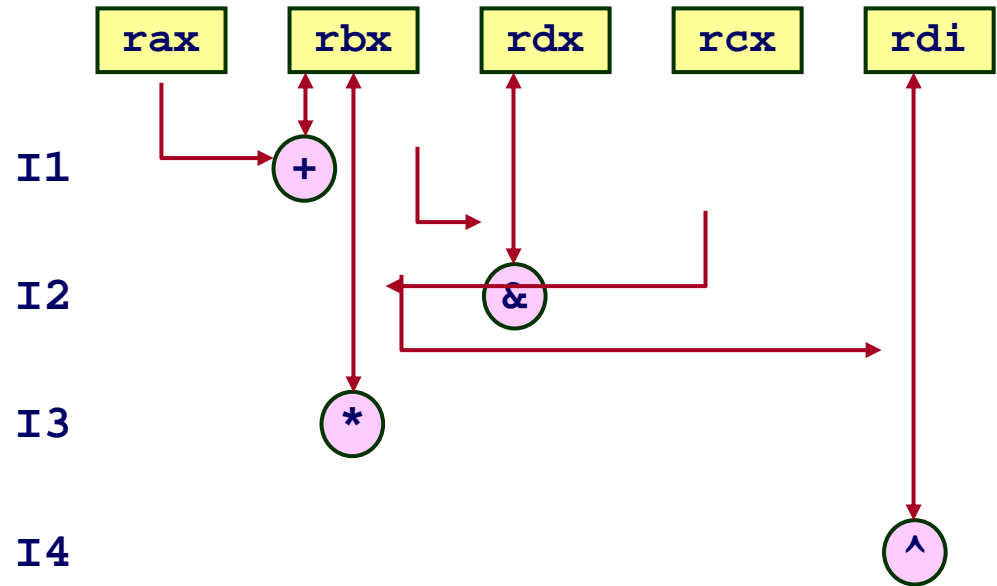
- Requires: Load, Integer arithmetic, Store

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

- Exact form and format of operations is trade secret
- Operations: split up instruction into simpler pieces
- Devise temporary names to describe how result of one operation gets used by other operations

Traditional View of Instruction Execution

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

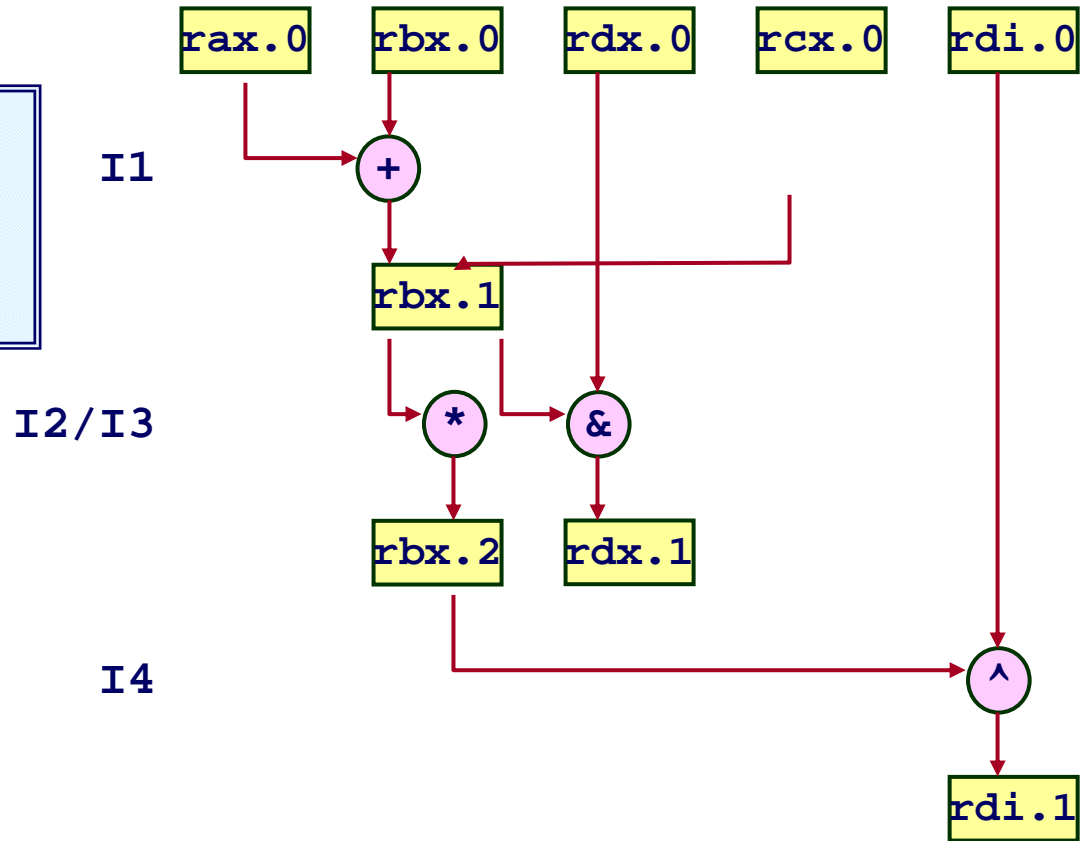


Imperative View

- 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;
}
```

Data Types

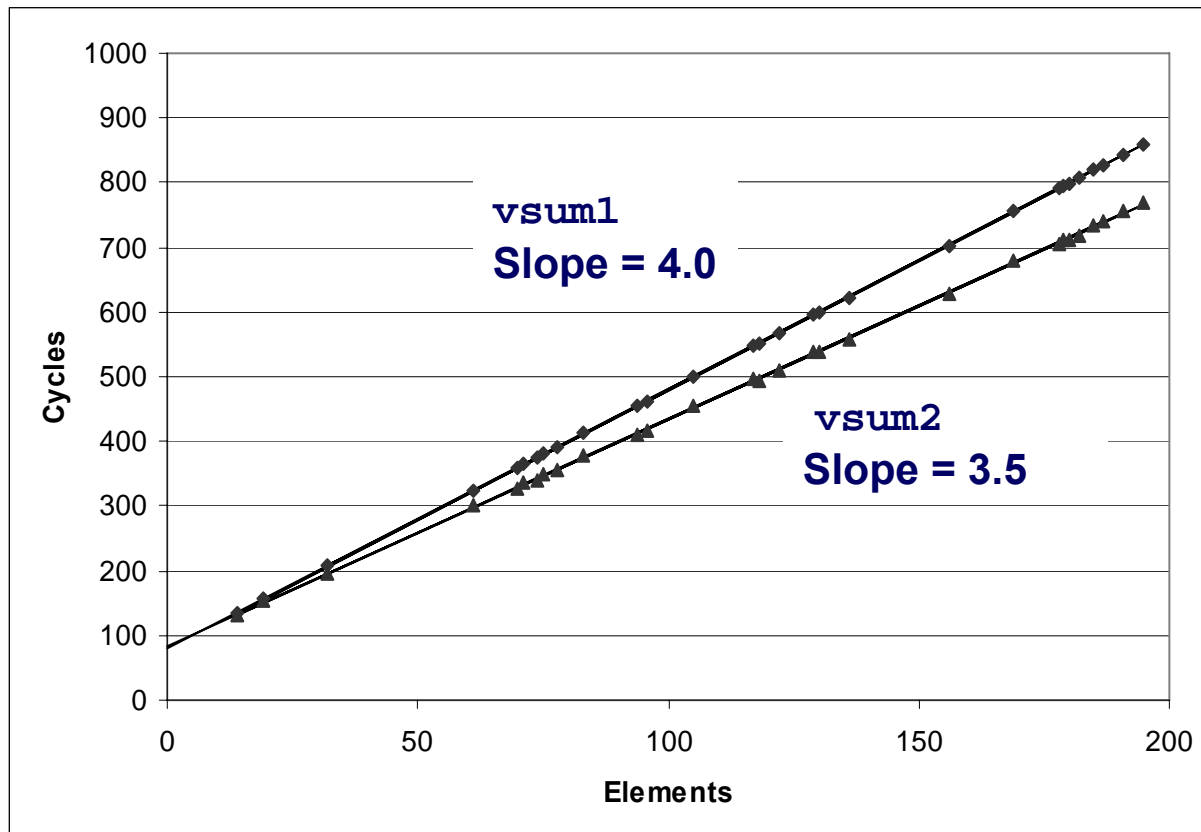
- Use different declarations for `data_t`
- `int`
- `float`
- `double`

Operations

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

Cycles Per Element

- Convenient way to express performance of program that operators on vectors or lists
- Length = n
- $T = CPE * n + \text{Overhead}$



x86-64 Compilation of Combine4

Inner Loop (Integer Multiply)

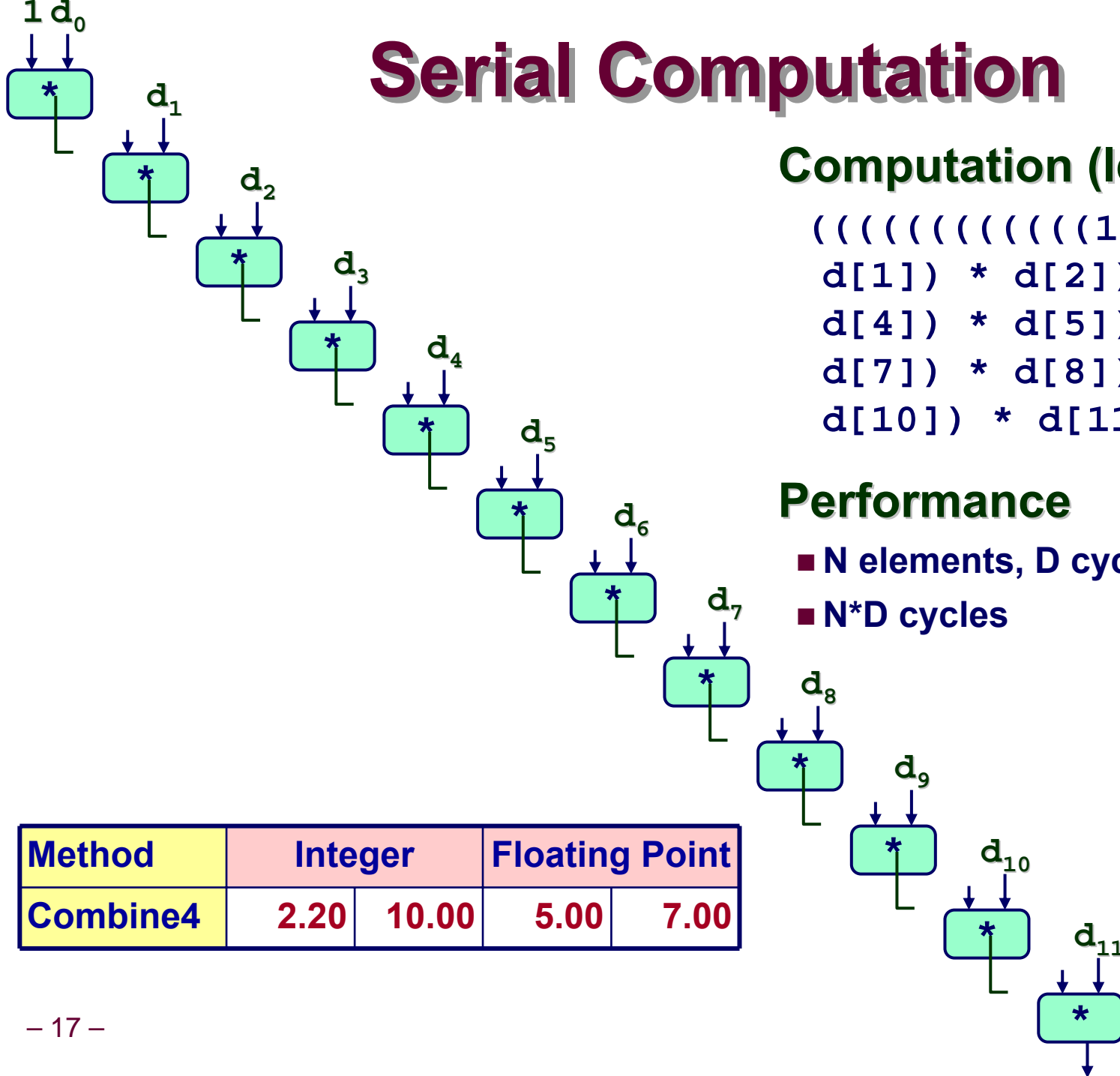
```
L33:                # Loop:
  movl  (%eax,%edx,4), %ebx # temp = d[i]
  incl  %edx              # i++
  imull %ebx, %ecx        # x *= temp
  cmpl  %esi, %edx        # i:length
  jl    L33               # if < goto Loop
```

Performance

- 5 instructions in 2 clock cycles

| Method | Integer | | Floating Point | |
|----------|---------|-------|----------------|------|
| Combine4 | 2.20 | 10.00 | 5.00 | 7.00 |

Serial Computation



Computation (length=12)

```

(((((((((((1 * d[0]) *
d[1]) * d[2]) * d[3]) *
d[4]) * d[5]) * d[6]) *
d[7]) * d[8]) * d[9]) *
d[10]) * d[11])
    
```

Performance

- N elements, D cycles/operation
- N*D cycles

| Method | Integer | | Floating Point | |
|--------|----------|------|----------------|------|
| | Combine4 | 2.20 | 10.00 | 5.00 |

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 OPER d[i]) OPER d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OPER d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration

Effect of Loop Unrolling

| Method | Integer | | Floating Point | |
|----------|---------|-------|----------------|------|
| Combine4 | 2.20 | 10.00 | 5.00 | 7.00 |
| Unroll 2 | 1.50 | 10.00 | 5.00 | 7.00 |

Helps Integer Sum

- Before: 5 operations per element
- After: 6 operations per 2 elements
 - = 3 operations per element

Others Don't Improve

- Sequential dependency
 - Each operation must wait until previous one completes

```
x = (x OPER d[i]) OPER 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 OPER (d[i] OPER d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OPER d[i];
    }
    *dest = x;
}
```

- Could change numerical results for FP

Effect of Reassociation

| Method | Integer | | Floating Point | |
|-------------------|---------|-------|----------------|------|
| Combine4 | 2.20 | 10.00 | 5.00 | 7.00 |
| Unroll 2 | 1.50 | 10.00 | 5.00 | 7.00 |
| 2 X 2 reassociate | 1.56 | 5.00 | 2.75 | 3.62 |

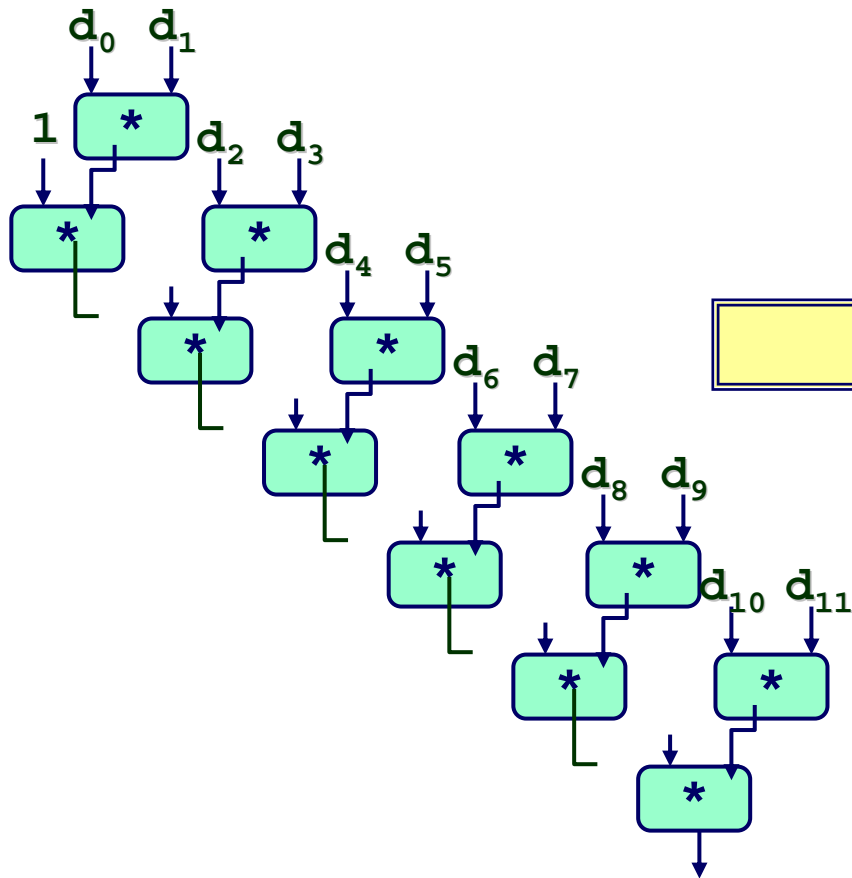
Nearly 2X speedup for Int *, FP +, FP *

- Breaks sequential dependency

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

- While computing result for iteration i , can precompute $d[i+2]*d[i+3]$ for iteration $i+2$

Reassociated Computation



Performance

- N elements, D cycles/operation
- Should be $(N/2+1)*D$ cycles
 - CPE = $D/2$
- Measured CPE slightly worse for FP

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

Loop Unrolling with Separate Accum.

```
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 OPER d[i];
        x1 = x1 OPER d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OPER d[i];
    }
    *dest = x0 OPER x1;
}
```

■ Different form of reassociation

Effect of Reassociation

| Method | Integer | | Floating Point | |
|-----------------------|---------|-------|----------------|------|
| Combine4 | 2.20 | 10.00 | 5.00 | 7.00 |
| Unroll 2 | 1.50 | 10.00 | 5.00 | 7.00 |
| 2 X 2 reassociate | 1.56 | 5.00 | 2.75 | 3.62 |
| 2 X 2 separate accum. | 1.50 | 5.00 | 2.50 | 3.50 |

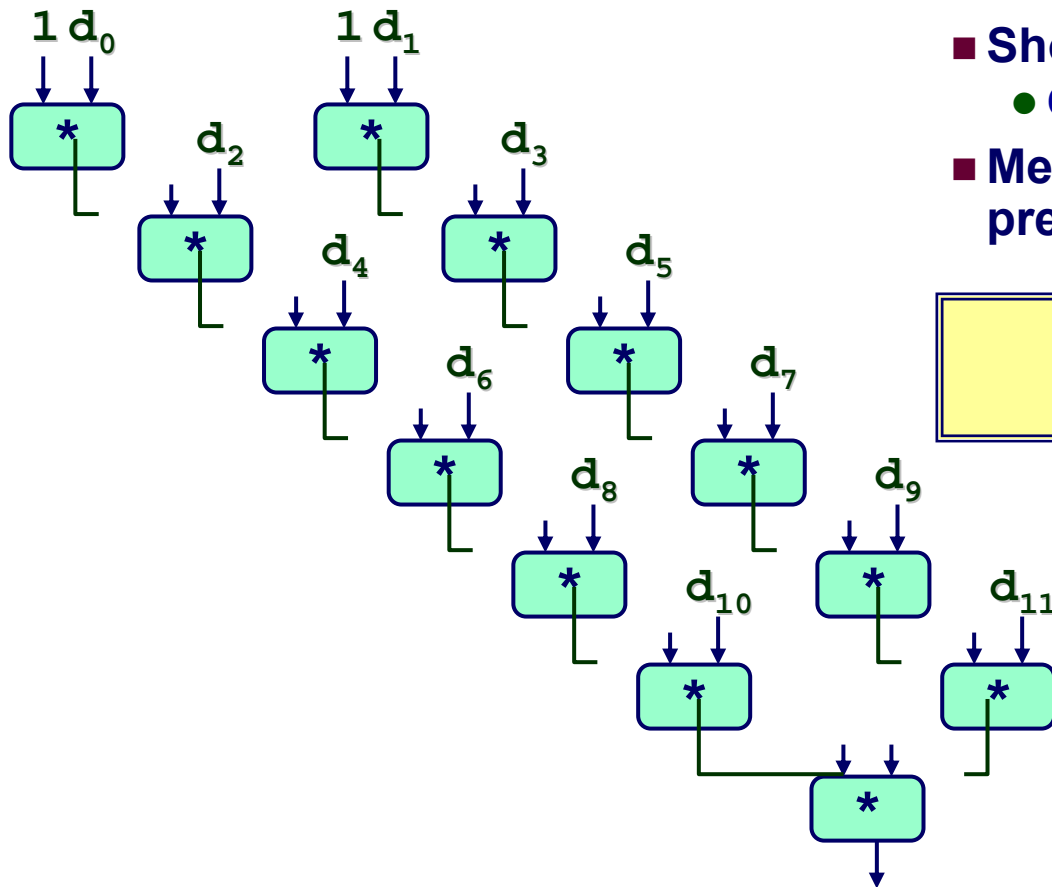
Nearly 2X speedup for Int *, FP +, FP *

- Breaks sequential dependency

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

- Computation of even elements independent of odd ones

Separate Accum. Computation



Performance

- N elements, D cycles/operation
- Should be $(N/2+1)*D$ cycles
 - CPE = $D/2$
- Measured CPE matches prediction!

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

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 pipelining limitations of execution units
- Large overhead
 - Finish off iterations sequentially
 - Especially for shorter lengths

Unrolling & Accumulating: Intel FP *

Case

- Intel Nocona (Saltwater fish machines)
- FP Multiplication
- Theoretical Limit: 2.00

| 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 (Saltwater fish machines)
- 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 (Saltwater fish machines)
- 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 (Saltwater fish machines)
- 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 |

Nocona vs. Core 2 FP *

| 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 |

Machines

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

Performance

- Core 2 lower latency & better pipelining
- But slower clock rate

Nocona vs. Core 2 Int *

| 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 |

Performance

- Newer version of GCC does reassociation
- Why for int's and not for float's?

| Int * | Unrolling Factor L | | | | | | | |
|-------|--------------------|------|------|------|------|------|------|------|
| K | 1 | 2 | 3 | 4 | 6 | 8 | 10 | 12 |
| 1 | 3.00 | 1.50 | | 1.00 | | 1.00 | | |
| 2 | | 1.50 | | 1.00 | | 1.00 | | |
| 3 | | | 1.00 | | | | | |
| 4 | | | | 1.00 | | 1.00 | | |
| 6 | | | | | 1.00 | | | 1.00 |
| 8 | | | | | | 1.00 | | |
| 10 | | | | | | | 1.00 | |
| 12 | | | | | | | | 1.33 |

Intel vs. AMD

FP *

| 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 |

Machines

- Intel Nocona
 - 3.2 GHz
- AMD Opteron
 - 2.0 GHz

Performance

- AMD lower latency & better pipelining
- But slower clock rate

Intel vs. AMD

Int *

Performance

- AMD multiplier much lower latency
- Can get high performance with less work
- Doesn't achieve as good an optimum

| 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 |

| Int * | Unrolling Factor L | | | | | | | |
|-------|--------------------|------|------|------|------|------|------|------|
| K | 1 | 2 | 3 | 4 | 6 | 8 | 10 | 12 |
| 1 | 3.00 | 3.00 | | 3.00 | | 3.00 | | |
| 2 | | 2.33 | | 2.0 | | 1.35 | | |
| 3 | | | 2.00 | | | | | |
| 4 | | | | 1.75 | | 1.38 | | |
| 6 | | | | | 1.50 | | | 1.50 |
| 8 | | | | | | 1.75 | | |
| 10 | | | | | | | 1.30 | |
| 12 | | | | | | | | 1.33 |

Intel vs. AMD

Int +

| 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 |

| Int + | Unrolling Factor L | | | | | | | |
|-------|--------------------|------|------|------|------|------|------|------|
| K | 1 | 2 | 3 | 4 | 6 | 8 | 10 | 12 |
| 1 | 2.32 | 1.50 | | 0.75 | | 0.63 | | |
| 2 | | 1.50 | | 0.83 | | 0.63 | | |
| 3 | | | 1.00 | | | | | |
| 4 | | | | 1.00 | | 0.63 | | |
| 6 | | | | | 0.83 | | | 0.67 |
| 8 | | | | | | 0.63 | | |
| 10 | | | | | | | 0.60 | |
| 12 | | | | | | | | 0.85 |

Performance

- AMD gets below 1.0
- Even just with unrolling

Explanation

- Both Intel & AMD can “double pump” integer units
- Only AMD can load two elements / cycle

Can We Go Faster?

Fall 2005 Lab #4

Performance Lab Class Status Page

[Home](#) | [Messages](#) | [Grace](#) | [Jobs](#) | [Update](#) | [Logout](#) | [Help](#)

| # | Nickname | Handin Version | Submission Date | ACPE | Grade Points |
|---|----------|----------------|------------------|------|--------------|
| 1 | Anton | 4 | Sun Oct 23 21:10 | 1.69 | 100 |
| 2 | edanaher | 1 | Thu Oct 13 22:41 | 2.37 | 100 |
| 3 | mjrosenb | 4 | Tue Oct 18 08:40 | 2.37 | 100 |
| 4 | Old Prof | 3 | Tue Oct 18 13:38 | 2.42 | 100 |
| 5 | mcwillia | 4 | Thu Oct 20 01:17 | 2.46 | 100 |

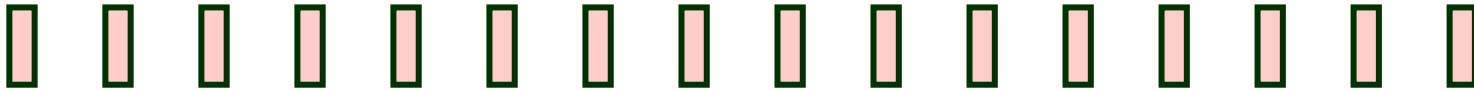
- Floating-point addition & multiplication gives theoretical optimum CPE of 2.00
- What did Anton do?

Programming with SSE3

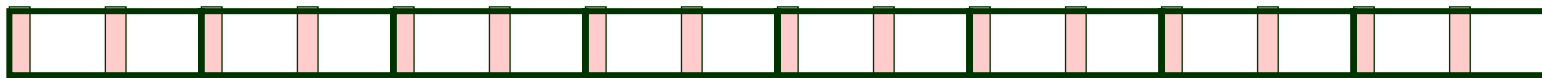
XMM Registers

- 16 total, each 16 bytes

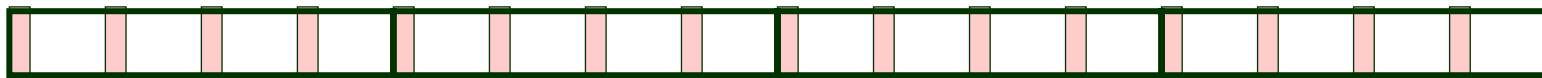
- 16 single-byte integers



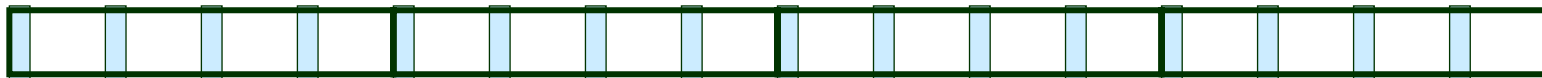
- 8 16-bit integers



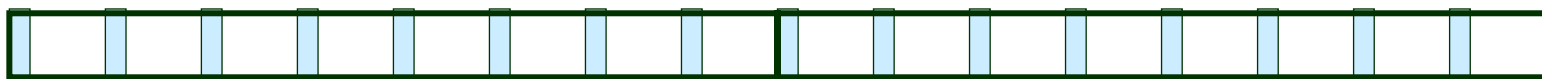
- 4 32-bit integers



- 4 single-precision floats



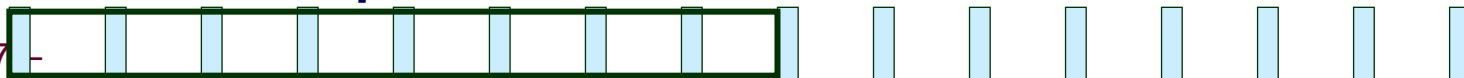
- 2 double-precision floats



- 1 single-precision float

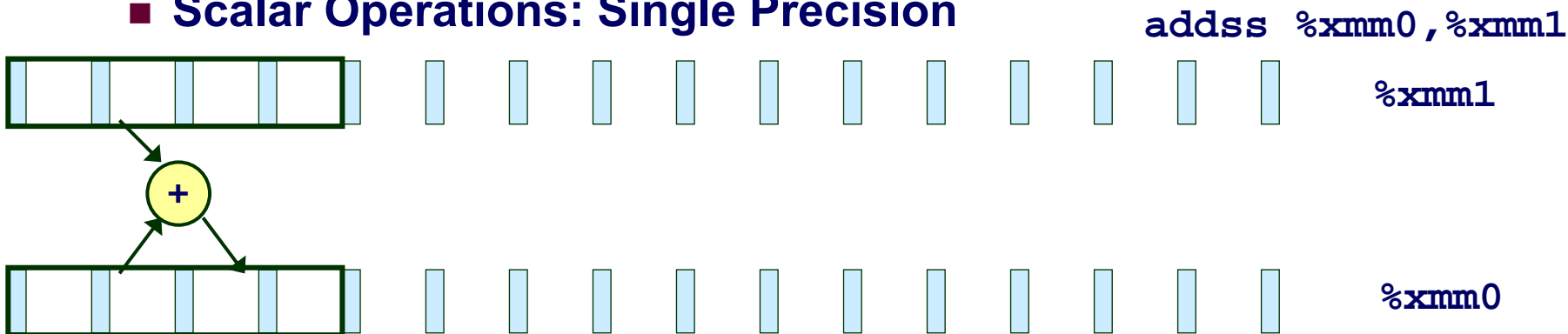


- 1 double-precision float

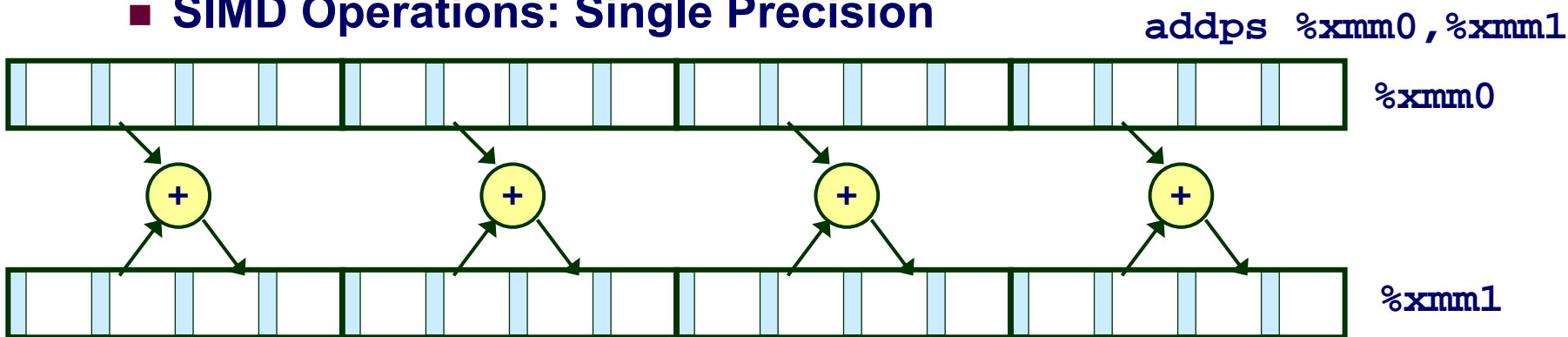


Scalar & SIMD Operations

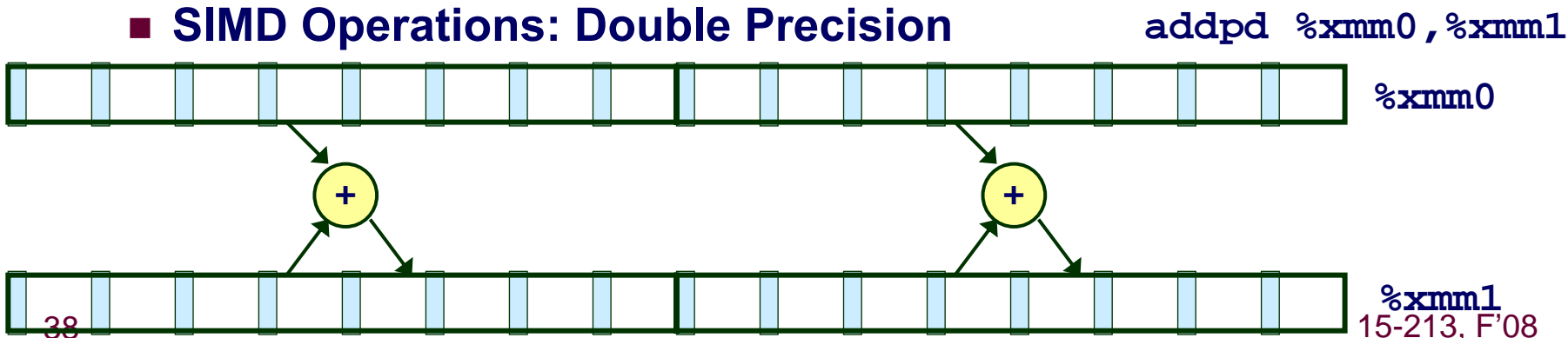
■ Scalar Operations: Single Precision



■ SIMD Operations: Single Precision



■ SIMD Operations: Double Precision



Getting GCC to Use SIMD Operations

Declarations

```
typedef float vec_t __attribute__((vector_size(16)));  
typedef union {  
    vec_t v;  
    float d[4];  
} pack_t
```

Accessing Vector Elements

```
pack_t xfer;  
vec_t accum;  
for (i = 0; i < 4; i++)  
    xfer.d[i] = IDENT;  
accum = xfer.v;
```

Invoking SIMD Operations

```
vec_t chunk = *((vec_t *) d);  
accum = accum OPER chunk;
```

Implementing Combine

```
void SSEx1_combine(vec_ptr v, float *dest)
{
    pack_t xfer;
    vec_t accum;
    float *d = get_vec_start(v);
    int cnt = vec_length(v);
    float result = IDENT;

    /* Initialize vector of 4 accumulators */

    /* Step until d aligned to multiple of 16 */

    /* Use packed operations with 4X parallelism */

    /* Single step to finish vector */

    /* Combine accumulators */
}
```


Getting Started

Create Vector of 4 Accumulators

```
/* Initialize vector of 4 accumulators */
int i;
for (i = 0; i < 4; i++)
    xfer.d[i] = IDENT;
accum = xfer.v;
```

Single Step to Meet Alignment Requirements

- Memory address of vector must be multiple of 16

```
/* Step until d aligned to multiple of 16 */
while (((long) d)%16 && cnt) {
    result = result OPER *d++;
    cnt--;
}
```

SIMD Loop

- Similar to 4-way loop unrolling
- Express with single arithmetic operation
 - Translates into single `addps` or `mulps` instruction

```
/* Use packed operations with 4X parallelism */  
while (cnt > 4) {  
    vec_t chunk = *((vec_t *) d);  
    accum = accum OPER chunk;  
    d += 4;  
    cnt -= 4;  
}
```

Completion

Finish Off Final Elements

- Similar to standard unrolling

```
/* Single step to finish vector */  
while (cnt) {  
    result = result OPER *d++;  
    cnt--;  
}
```

Combine Accumulators

- Use union to reference individual elements

```
/* Combine accumulators */  
xfer.v = accum;  
for (i = 0; i < 4; i++)  
    result = result OPER xfer.d[i];  
*dest = result;
```

SIMD Results

Intel Nocona

| | Unrolling Factor L | | | |
|-------|--------------------|-------|-------|-------|
| | 4 | 8 | 16 | 32 |
| FP + | 1.25 | 0.82 | 0.50 | 0.58 |
| FP * | 1.90 | 1.24 | 0.90 | 0.57 |
| Int + | 0.84 | 0.70 | 0.51 | 0.58 |
| Int * | 39.09 | 37.65 | 36.75 | 37.44 |

AMD Opteron

| | Unrolling Factor L | | | |
|-------|--------------------|------|------|------|
| | 4 | 8 | 16 | 32 |
| FP + | 1.00 | 0.50 | 0.50 | 0.50 |
| FP * | 1.00 | 0.50 | 0.50 | 0.50 |
| Int + | 0.75 | 0.38 | 0.28 | 0.27 |
| Int * | 9.40 | 8.63 | 9.32 | 9.12 |

Nocona Results

- FP approaches theoretical optimum of 0.50
- Int + shows speed up
- For int *, compiler does not generate SIMD code

Portability

- GCC can target other machines with this code

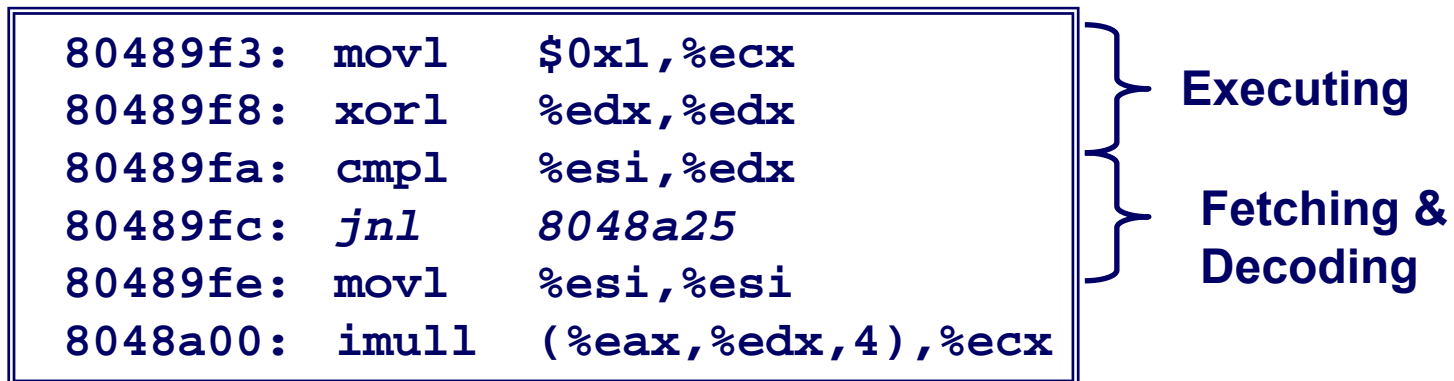
Intel Core 2

| | Unrolling Factor L | | | |
|-------|--------------------|------|------|------|
| | 4 | 8 | 16 | 32 |
| FP + | 0.75 | 0.40 | 0.25 | 0.24 |
| FP * | 1.00 | 0.53 | 0.24 | 0.24 |
| Int + | 0.50 | 0.32 | 0.24 | 0.25 |
| Int * | 3.69 | 2.18 | 1.64 | 1.58 |

What About Branches?

Challenge

- Instruction Control Unit must work well ahead of Exec. Unit
 - To generate enough operations to keep EU busy



- 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

```
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

Branch Taken

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

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>i = 98</i> |
| 80488b9: | j1 | 80488b1 | |
| 80488b1: | movl | (%ecx,%edx,4),%eax | |
| 80488b4: | addl | %eax,(%edi) | |
| 80488b6: | incl | %edx | |
| 80488b7: | cmpl | %esi,%edx | <i>i = 99</i> |
| 80488b9: | j1 | 80488b1 | |
| 80488b1: | movl | (%ecx,%edx,4),%eax | |
| 80488b4: | addl | %eax,(%edi) | |
| 80488b6: | incl | %edx | |
| 80488b7: | cmpl | %esi,%edx | <i>i = 100</i> |
| 80488b9: | j1 | 80488b1 | |
| 80488b1: | movl | (%ecx,%edx,4),%eax | |
| 80488b4: | addl | %eax,(%edi) | |
| 80488b6: | incl | %edx | |
| 80488b7: | cmpl | %esi,%edx | <i>i = 101</i> |
| 80488b9: | j1 | 80488b1 | |

Assume vector length = 100

Predict Taken (OK)

Predict Taken (Oops)

Read invalid location



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

Invalidate

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

Branch Misprediction Recovery

Assume vector length = 100

```
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
- One of the major performance limiters

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

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:
 - 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

Role of Programmer

How should I write my programs, given that I have a good, optimizing compiler?

Don't: Smash Code into Oblivion

- Hard to read, maintain, & assure correctness

Do:

- Select best algorithm
- Write code that's readable & maintainable
 - Procedures, recursion, without built-in constant limits
 - Even though these factors can slow down code
- Eliminate optimization blockers
 - Allows compiler to do its job

Focus on Inner Loops

- Do detailed optimizations where code will be executed repeatedly
- Will get most performance gain here
- Understand how enough about machine to tune effectively

Fact Revisited

```
int fact_u3b(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

| Machine | Nocona | Core 2 | |
|----------|--------|--------|--------|
| | | Core 2 | Core 2 |
| Compiler | 3.4 | 3.4 | 4.1 |
| rfact | 15.5 | 6.0 | 3.0 |
| fact | 10.0 | 3.0 | 3.0 |
| fact_u3a | 10.0 | 3.0 | 3.0 |
| fact_u3b | 3.3 | 1.0 | 3.0 |

Loop Unrolling + Reassociation

- ~3X drop for GCC 3.4
- No improvement for GCC 4.1
 - Does reassociation, but in wrong direction!

Preventing Reassociation

```
int fact_u3x(int n)
{
    int i;
    int result = 1;

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

Cycles Per Element

| Machine | Nocona | Core 2 | |
|----------|--------|--------|-----|
| | | | |
| Compiler | 3.4 | 3.4 | 4.1 |
| rfact | 15.5 | 6.0 | 3.0 |
| fact | 10.0 | 3.0 | 3.0 |
| fact_u3a | 10.0 | 3.0 | 3.0 |
| fact_u3b | 3.3 | 1.0 | 3.0 |
| fact_u3x | 3.3 | 1.0 | 1.0 |

- Obfuscate code enough to prevent reassociation