Lecture 14:
Program Optimization
Computer Architecture and
Systems Programming
(252-0061-00)

Timothy Roscoe
Herbstsemester 2012
Last time

Making the pipelined processor work!

• Data Hazards
  – Instruction having register R as source follows shortly after instruction having register R as destination
  – Common condition, don’t want to slow down pipeline

• Control Hazards
  – Mispredict conditional branch
    • Our design predicts all branches as being taken
    • Naïve pipeline executes two extra instructions
  – Getting return address for ret instruction
    • Naïve pipeline executes three extra instructions

• Making sure it really works
  – What if multiple special cases happen simultaneously? Must analyze carefully
  – First version had subtle bug
    • Only arises with unusual instruction combination
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:

```c
/* 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};
sum_rows1(A, B, 3);

Value of B:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>[4, 8, 16]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i = 0</td>
<td>[3, 8, 16]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i = 1</td>
<td>[3, 22, 16]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i = 2</td>
<td>[3, 22, 224]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
  - Assumes no memory aliasing (otherwise possibly incorrect)
Unaliased version when aliasing happens

```c
/* 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;
    }
}
```

```c
double A[9] =
{ 0, 1, 2,
  4, 8, 16},
32, 64, 128};
sum_rows1(A, B, 3);
```

- Aliasing still creates interference
- Result different from before

Value of B:

init: [4, 8, 16]
i = 0: [3, 8, 16]
i = 1: [3, 27, 16]
i = 2: [3, 27, 224]
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: \( t_1 = a[i], t_2 = b[i+1], \ldots \)
    • Compute: \( t_4 = t_1 \times t_2; \ldots \)
    • Store: \( a[i] = t_{12}, b[i+1] = t_7, \ldots \)
More difficult example

- Matrix multiplication: \( C = A \times B + C \)

```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
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

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

```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+=2)
      for (j = 0; j < n; j+=2)
        for (k = 0; k < n; k+=2)
          <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)];
          </body>
  }
```
Today

- Program optimization
  - Optimization blocker: Memory aliasing
  - **Out of order processing: Instruction level parallelism**
  - Understanding branch prediction
Example: Compute factorials

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

```c
int fact(int n)
{
    int i;
    int result = 1;
    for (i = n; i > 0; i--)
        result = result * i;
    return result;
}
```

- **Machines**
  - Intel P. 4 Nocona, 3.2 GHz
  - Intel Core 2, 2.7 GHz
- **Compiler versions**
  - GCC 3.4.2

Cycles per element (or per mult)

<table>
<thead>
<tr>
<th>Machine</th>
<th>Nocona</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfact</td>
<td>15.5</td>
<td>6.0</td>
</tr>
<tr>
<td>fact</td>
<td>10.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Something changed from Pentium 4 to Core: Details later
Optimization 1: Loop unrolling

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

```c
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;
}
```

<table>
<thead>
<tr>
<th></th>
<th>Nocona</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfact</td>
<td>15.5</td>
<td>6.0</td>
</tr>
<tr>
<td>fact</td>
<td>10.0</td>
<td>3.0</td>
</tr>
<tr>
<td>fact_u3a</td>
<td>10.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>
Optimization 2: 
Multiple accumulators

```c
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;
}
```

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

<table>
<thead>
<tr>
<th>Machine</th>
<th>Nocona</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfact</td>
<td>15.5</td>
<td>6.0</td>
</tr>
<tr>
<td>fact</td>
<td>10.0</td>
<td>3.0</td>
</tr>
<tr>
<td>fact_u3a</td>
<td>10.0</td>
<td>3.0</td>
</tr>
<tr>
<td>fact_u3b</td>
<td>3.3</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Modern CPU design

Instruction Control

- Instruction Cache
  - Instruction Decode
  - Fetch Control
  - Retirement Unit
    - Register File
  - Address
  - Operations
  - Instructions

Execution

- Functional Units
  - Integer/Branch
  - General Integer
  - FP Add
  - FP Mult/Div
  - Load
  - Store
  - Operation Results
  - Addr.
  - Addr.
  - Data
  - Data

Register Updates

Prediction OK?
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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td><strong>Integer/Long Divide</strong></td>
<td><strong>36/106</strong></td>
<td><strong>36/106</strong></td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td><strong>Single/Double FP Divide</strong></td>
<td><strong>32/46</strong></td>
<td><strong>32/46</strong></td>
</tr>
</tbody>
</table>
Latency versus Throughput

• Last slide: \( latency \) \( cycles/issue \)

| Integer Multiply | 10 | 1 |

• Consequence:
  - How fast can 10 independent int mults be executed?
    \[ t_1 = t_2 \times t_3; \quad t_4 = t_5 \times t_6; \quad \ldots \]
  - How fast can 10 sequentially dependent int mults be executed?
    \[ t_1 = t_2 \times t_3; \quad t_4 = t_5 \times t_1; \quad t_6 = t_7 \times t_4; \quad \ldots \]

• Major problem for fast execution: Keep pipelines filled
Hard Bounds

• How many cycles at least if
  – Function requires \( n \) int mults?
  – Function requires \( n \) float adds?
  – Function requires \( n \) float ops (adds and mults)?
• Latency and throughput of instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>36/106</td>
<td>36/106</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>32/46</td>
<td>32/46</td>
</tr>
</tbody>
</table>
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 on Nocona?
  - 3.2 Gflop/s = 3200 Mflop/s. *Why?*
## Nocona vs. Core 2

### Nocona (3.2 GHz)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>36/106</td>
<td>36/106</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>32/46</td>
<td>32/46</td>
</tr>
</tbody>
</table>

### Core 2 (2.7 GHz) (More recent Intel microprocessors)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>18/50</td>
<td>18/50</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>4/5</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>18/32</td>
<td>18/32</td>
</tr>
</tbody>
</table>
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

\[
\text{imulq } %rax, 8(\%rbx,\%rdx,4)
\]

• Goal: Each operation utilizes single functional unit
  – Requires: Load, integer arithmetic, store

\[
\begin{align*}
\text{load } 8(\%rbx,\%rdx,4) & \rightarrow \text{ temp1} \\
\text{imulq } %rax, \text{ temp1} & \rightarrow \text{ temp2} \\
\text{store temp2, } 8(\%rbx,\%rdx,4) &
\end{align*}
\]

  – Exact form and format of operations is trade secret
Traditional view of instruction execution

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

```
addq %rax, %rbx  # I1
andq %rbx, %rdx  # I2
mulq %rcx, %rbx  # I3
xorq %rbx, %rdi  # I4
```
Dataflow view of instruction execution

- 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

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

```c
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 (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 = CPE*n + Overhead

![Graph showing vsum1: Slope = 4.0 and vsum2: Slope = 3.5](image)

$n =$ Number of elements
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;
}

• Inner loop (Case: integer multiply)

L33:

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>10.0</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>10.0</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>2.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>
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

- 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 unroll2ra_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?
• Yes, for FP. Why?
Effect of Reassociation

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add/mult)</th>
<th>Float (add/mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2</td>
<td>1.5</td>
<td>10.0</td>
</tr>
<tr>
<td>unroll2-ra</td>
<td>1.56</td>
<td>5.0</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- Nearly 2x speedup for Int *, FP +, FP *
  - Reason: Breaks sequential dependency

\[ x = x \text{ OP } (d[i] \text{ OP } d[i+1]); \]

- Why is that? (next slide)
Reassociated Computation

\[ x = x \text{ OP } (d[i] \text{ OP } d[i+1]); \]

- **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:
    \[ \text{CPE} = D/2 \]
  - Measured CPE slightly worse for FP
Loop unrolling with separate accumulators

```c
void unroll2sa_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

<table>
<thead>
<tr>
<th>Method</th>
<th>Int (add-mult)</th>
<th>Float (add-mult)</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine4</td>
<td>2.2</td>
<td>5.0</td>
</tr>
<tr>
<td>unroll2</td>
<td>1.5</td>
<td>5.0</td>
</tr>
<tr>
<td>unroll2-ra</td>
<td>1.56</td>
<td>2.75</td>
</tr>
<tr>
<td>unroll2-sa</td>
<td>1.50</td>
<td>2.5</td>
</tr>
<tr>
<td>bound</td>
<td>1.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

- Almost exact 2x speedup (over unroll2) for Int *, FP +, FP *
  - Breaks sequential dependency in a “cleaner,” more obvious way
    
    \[
    \begin{align*}
    x_0 &= x_0 \text{ OP } d[i]; \\
    x_1 &= x_1 \text{ OP } d[i+1];
    \end{align*}
    \]
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 floating point multiply

- Case
  - Intel Nocona
  - FP Multiplication
  - Theoretical Limit: 2.00

<table>
<thead>
<tr>
<th>Accumulators</th>
<th>FP *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>7.00</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3.50</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Unrolling & Accumulating: Intel floating point addition

- Case
  - Intel Nocona
  - FP Addition
  - Theoretical Limit: 2.00

<table>
<thead>
<tr>
<th>FP +</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>K</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>5.00</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Unrolling & Accumulating: Intel integer multiplication

- Case
  - Intel Nocona
  - Integer Multiplication
  - Theoretical Limit: 1.00

<table>
<thead>
<tr>
<th>Int *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>10.00</td>
</tr>
<tr>
<td>2</td>
<td>5.00</td>
</tr>
<tr>
<td>3</td>
<td>3.33</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Unrolling & Accumulating: Intel integer addition

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

<table>
<thead>
<tr>
<th>Int +</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>K</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2.20</td>
</tr>
<tr>
<td>2</td>
<td>1.50</td>
</tr>
<tr>
<td>3</td>
<td>1.34</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
### Nocona va. Core 2, FP *

- **Machines**
  - Intel Nocona
    - 3.2 GHz
  - Intel Core 2
    - 2.7 GHz
  - **Performance**
    - Core 2 lower latency & fully pipelined (1 cycle/issue)

<table>
<thead>
<tr>
<th>FP *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K 1  2  3  4  6  8  10 12</td>
</tr>
<tr>
<td>1</td>
<td>7.00 7.00 7.01 7.00</td>
</tr>
<tr>
<td>2</td>
<td>3.50 3.50 3.50</td>
</tr>
<tr>
<td>3</td>
<td>2.34</td>
</tr>
<tr>
<td>4</td>
<td>2.01 2.00</td>
</tr>
<tr>
<td>6</td>
<td>2.00 2.01</td>
</tr>
<tr>
<td>8</td>
<td>2.01</td>
</tr>
<tr>
<td>10</td>
<td>2.00</td>
</tr>
<tr>
<td>12</td>
<td>2.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FP *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K 1  2  3  4  6  8  10 12</td>
</tr>
<tr>
<td>1</td>
<td>4.00 4.00 4.00 4.01</td>
</tr>
<tr>
<td>2</td>
<td>2.00 2.00 2.00</td>
</tr>
<tr>
<td>3</td>
<td>1.34</td>
</tr>
<tr>
<td>4</td>
<td>1.00 1.00</td>
</tr>
<tr>
<td>6</td>
<td>1.00 1.00</td>
</tr>
<tr>
<td>8</td>
<td>1.00</td>
</tr>
<tr>
<td>10</td>
<td>1.00</td>
</tr>
<tr>
<td>12</td>
<td>1.00</td>
</tr>
<tr>
<td>Int *</td>
<td>Unrolling Factor L</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------</td>
</tr>
<tr>
<td></td>
<td>K</td>
</tr>
<tr>
<td>-------</td>
<td>----</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Int *</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K</td>
</tr>
<tr>
<td>-------</td>
<td>----</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

**Nocona vs. Core 2, Int ***

- Performance
  - Newer version of GCC does reassociation
  - Why for int’s and not for float’s?
**Intel vs. AMD**

**FP * Unrolling Factor L**

<table>
<thead>
<tr>
<th>K</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.00</td>
<td>7.00</td>
<td>7.01</td>
<td>7.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3.50</td>
<td>3.50</td>
<td>3.50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2.34</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2.01</td>
<td>2.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>2.00</td>
<td>2.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>2.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FP * Unrolling Factor L**

<table>
<thead>
<tr>
<th>K</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
<td>4.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1.34</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1.00</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>1.00</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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 * Unrolling Factor L**

<table>
<thead>
<tr>
<th>K</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10.00</td>
<td>10.00</td>
<td>10.00</td>
<td>10.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5.00</td>
<td>5.01</td>
<td>5.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3.33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2.50</td>
<td></td>
<td></td>
<td>2.51</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1.67</td>
<td></td>
<td></td>
<td></td>
<td>1.67</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.14</td>
<td></td>
</tr>
</tbody>
</table>

**Int * Unrolling Factor L**

<table>
<thead>
<tr>
<th>K</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.00</td>
<td>3.00</td>
<td>3.00</td>
<td></td>
<td>3.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2.33</td>
<td>2.00</td>
<td>1.35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1.75</td>
<td>1.38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1.50</td>
<td></td>
<td>1.50</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>1.75</td>
<td></td>
<td></td>
<td>1.75</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.33</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Int +</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2.20</td>
</tr>
<tr>
<td>2</td>
<td>1.50</td>
</tr>
<tr>
<td>3</td>
<td>1.34</td>
</tr>
<tr>
<td>4</td>
<td>1.09</td>
</tr>
<tr>
<td>6</td>
<td>1.01</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Int +</th>
<th>Unrolling Factor L</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2.32</td>
</tr>
<tr>
<td>2</td>
<td>1.50</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Can We Go Faster?

• Yes, using 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

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>80489f3</td>
<td>movl</td>
<td>$0x1,%ecx</td>
</tr>
<tr>
<td>80489f8</td>
<td>xorl</td>
<td>%edx,%edx</td>
</tr>
<tr>
<td>80489fa</td>
<td>cmpl</td>
<td>%esi,%edx</td>
</tr>
<tr>
<td>80489fc</td>
<td>jnl</td>
<td>8048a25</td>
</tr>
<tr>
<td>80489fe</td>
<td>movl</td>
<td>%esi,%esi</td>
</tr>
<tr>
<td>8048a00</td>
<td>imull</td>
<td>(%eax,%edx,4),%ecx</td>
</tr>
</tbody>
</table>

– 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

---

### Code Snippet

```
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
8048a25:  cmpl  %edi,%edx
8048a27:  jl    8048a20
8048a29:  movl  0xc(%ebp),%eax
8048a2c:  leal  0xfffffffffe8(%ebp),%esp
8048a2f:  movl  %ecx,(%eax)
```

---

Branch Taken: 8048a25

Branch Not-Taken: 80489fc
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 0xfffffffff8(%ebp),%esp
8048a2f:  movl %ecx,(%eax)
```

Begin Execution
### Branch prediction through loop

**Assume**

\[ \text{vector length} = 100 \]

- **Predict Taken (OK)**
  - \( i = 98 \)
  - \( i = 99 \)
  - \( i = 100 \)
  - \( i = 101 \)

- **Predict Taken (Oops)**
  - \( i = 98 \)
  - \( i = 99 \)
  - \( i = 100 \)
  - \( i = 101 \)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>80488b1</td>
<td>movl (%ecx,%edx,4),%eax</td>
<td>98</td>
</tr>
<tr>
<td>80488b4</td>
<td>addl %eax,(%edi)</td>
<td></td>
</tr>
<tr>
<td>80488b6</td>
<td>incl %edx</td>
<td></td>
</tr>
<tr>
<td>80488b7</td>
<td>cmpl %esi,%edx</td>
<td></td>
</tr>
<tr>
<td>80488b9</td>
<td>jl 80488b1</td>
<td></td>
</tr>
</tbody>
</table>

- **Read invalid location**
- **Executed**
- **Fetched**
### Branch misprediction invalidation

Assume vector length = 100

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>80488b1: movl (%ecx,%edx,4),%eax</td>
<td>Predict Taken (OK)</td>
</tr>
<tr>
<td>80488b4: addl %eax,(%edi)</td>
<td></td>
</tr>
<tr>
<td>80488b6: incl %edx</td>
<td></td>
</tr>
<tr>
<td>80488b7: cmpl %esi,%edx</td>
<td></td>
</tr>
<tr>
<td>80488b9: jl 80488b1</td>
<td></td>
</tr>
<tr>
<td>80488b1: movl (%ecx,%edx,4),%eax</td>
<td>Predict Taken (Oops)</td>
</tr>
<tr>
<td>80488b4: addl %eax,(%edi)</td>
<td></td>
</tr>
<tr>
<td>80488b6: incl %edx</td>
<td></td>
</tr>
<tr>
<td>80488b7: cmpl %esi,%edx</td>
<td></td>
</tr>
<tr>
<td>80488b9: jl 80488b1</td>
<td></td>
</tr>
<tr>
<td>80488b1: movl (%ecx,%edx,4),%eax</td>
<td>Invalidate</td>
</tr>
<tr>
<td>80488b4: addl %eax,(%edi)</td>
<td></td>
</tr>
<tr>
<td>80488b6: incl %edx</td>
<td></td>
</tr>
<tr>
<td>80488b7: cmpl %esi,%edx</td>
<td></td>
</tr>
<tr>
<td>80488b9: jl 80488b1</td>
<td></td>
</tr>
</tbody>
</table>

- Assume vector length = 100
- Predict Taken (OK)
- Predict Taken (Oops)
- Invalidate
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 0xfffffffffe8(%ebp),%esp
80488be: popl %ebx
80488bf: popl %esi
80488c0: popl %edi

• Performance Cost
  – Multiple clock cycles on modern processor
  – Can be a major performance limiter

Definitely not taken
Determining misprediction penalty

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

```c
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;
}
```

```
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

- Cannot use conditional move when either outcome has side effect

```c
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;
}
```

```assembly
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
```
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 = ±1
    Case +1: y = [+1, +1, +1, ..., +1, +1]
    Case −1: y = [−1, −1, −1, ..., −1, −1]
    Case A: y = [+1, −1, +1, ..., +1, −1] (alternate)
    Case R: y = [+1, −1, −1, ..., −1, +1] (random)
Testing Outcomes

<table>
<thead>
<tr>
<th>Case</th>
<th>cmov</th>
<th>cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>12.3</td>
<td>18.2</td>
</tr>
<tr>
<td>−1</td>
<td>12.3</td>
<td>12.2</td>
</tr>
<tr>
<td>A</td>
<td>12.3</td>
<td>15.2</td>
</tr>
<tr>
<td>R</td>
<td>12.3</td>
<td>31.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>cmov</th>
<th>cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>8.05</td>
<td>10.1</td>
</tr>
<tr>
<td>−1</td>
<td>8.05</td>
<td>8.1</td>
</tr>
<tr>
<td>A</td>
<td>8.05</td>
<td>9.2</td>
</tr>
<tr>
<td>R</td>
<td>8.05</td>
<td>15.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>cmov</th>
<th>cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>7.17</td>
<td>9.2</td>
</tr>
<tr>
<td>−1</td>
<td>7.17</td>
<td>8.2</td>
</tr>
<tr>
<td>A</td>
<td>7.17</td>
<td>8.7</td>
</tr>
<tr>
<td>R</td>
<td>7.17</td>
<td>17.7</td>
</tr>
</tbody>
</table>

### 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: $2 \times (31.2 - 15.2) = 32$ cycles
  - AMD: $2 \times (15.7 - 9.2) = 13$ cycles
  - Core 2: $2 \times (17.7 - 8.7) = 18$ 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)