Lecture 4
Computer Architecture and Performance
Saman Amarasinghe
2013
Outline

X86–64 Assembly Primer
Overview of Computer Architecture
Profiling a Program
Set of Example Programs
Today

- Registers, Instruction format, Opcodes, data types, addressing modes

More to come later…
# X86–64 Registers

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>64-bit</td>
<td>general-purpose registers</td>
</tr>
<tr>
<td>6</td>
<td>16-bit</td>
<td>segment registers</td>
</tr>
<tr>
<td>1</td>
<td>64-bit</td>
<td>RFLAGS register</td>
</tr>
<tr>
<td>1</td>
<td>64-bit</td>
<td>instruction pointer register (%rip)</td>
</tr>
<tr>
<td>7</td>
<td>64-bit</td>
<td>control register</td>
</tr>
<tr>
<td>8</td>
<td>64-bit</td>
<td>MMX registers</td>
</tr>
<tr>
<td>16</td>
<td>128-bit</td>
<td>XMM registers (for SSE)</td>
</tr>
<tr>
<td>1</td>
<td>32-bit</td>
<td>MXCSR register (SSE2 control register)</td>
</tr>
<tr>
<td>16</td>
<td>256-bit</td>
<td>YMM registers (for AVX)</td>
</tr>
<tr>
<td>8</td>
<td>80-bit</td>
<td>x87 FPU data registers</td>
</tr>
<tr>
<td>1</td>
<td>16-bit</td>
<td>x87 FPU control register</td>
</tr>
<tr>
<td>1</td>
<td>16-bit</td>
<td>x87 FPU status register</td>
</tr>
<tr>
<td>1</td>
<td>48-bit</td>
<td>x87 FPU instruction pointer register</td>
</tr>
<tr>
<td>1</td>
<td>48-bit</td>
<td>x87 FPU data operant pointer register</td>
</tr>
<tr>
<td>1</td>
<td>16-bit</td>
<td>x87 FPU tag register</td>
</tr>
<tr>
<td>1</td>
<td>11-bit</td>
<td>x87 FPU opcode register</td>
</tr>
</tbody>
</table>
### x86–64 General Registers

<table>
<thead>
<tr>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
<td>%ax</td>
<td>%al</td>
<td></td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%bx</td>
<td>%bl</td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%cx</td>
<td>%cl</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%dx</td>
<td>%dl</td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%si</td>
<td>%sil</td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%di</td>
<td>%sil</td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%bp</td>
<td>%bpl</td>
<td></td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
<td>%r8w</td>
<td>%r8b</td>
<td></td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
<td>%r9w</td>
<td>%r9b</td>
<td></td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
<td>%r10w</td>
<td>%r10b</td>
<td></td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
<td>%r11w</td>
<td>%r11b</td>
<td></td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
<td>%r12w</td>
<td>%r12b</td>
<td></td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
<td>%r13w</td>
<td>%r13b</td>
<td></td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
<td>%r14w</td>
<td>%r14b</td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
<td>%r15w</td>
<td>%r15b</td>
<td></td>
</tr>
</tbody>
</table>

Also, the high–order bytes of %ax, %bx, %cx, and %dx are available as %ah, %bh, %ch, and %dh.
Instruction Format

\[ \langle \text{opcode} \rangle \langle \text{operand\_list} \rangle \]

- \langle \text{opcode} \rangle \text{ is a short mnemonic identifying the type of instruction with a single-character suffix indicating the data type.}
  
  - **Example:** `movq -16(%rbp), %rax`
  
  - If the suffix is missing, it can usually be inferred from the sizes of operand registers.

- \langle \text{operand\_list} \rangle \text{ is 0, 1, 2, or (rarely) 3 operands separated by commas.}
  
  - One of the operands (the final operand in AT&T assembly format) is the destination.
  
  - The other operands are read-only (const).
## x86–64 Data Types

<table>
<thead>
<tr>
<th>C declaration</th>
<th>C constant</th>
<th>x86–64 size in bytes</th>
<th>Assembly suffix</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>'c'</td>
<td>1</td>
<td>b</td>
<td>Byte</td>
</tr>
<tr>
<td>short</td>
<td>172</td>
<td>2</td>
<td>w</td>
<td>Word</td>
</tr>
<tr>
<td>int</td>
<td>172</td>
<td>4</td>
<td>l</td>
<td>Double word</td>
</tr>
<tr>
<td>unsigned int</td>
<td>172U</td>
<td>4</td>
<td>l</td>
<td>Double word</td>
</tr>
<tr>
<td>long</td>
<td>172L</td>
<td>8</td>
<td>q</td>
<td>Quad word</td>
</tr>
<tr>
<td>unsigned long</td>
<td>172UL</td>
<td>8</td>
<td>q</td>
<td>Quad word</td>
</tr>
<tr>
<td>char *</td>
<td>&quot;6.172&quot;</td>
<td>8</td>
<td>q</td>
<td>Quad word</td>
</tr>
<tr>
<td>float</td>
<td>6.172F</td>
<td>4</td>
<td>s</td>
<td>Single precision</td>
</tr>
<tr>
<td>double</td>
<td>6.172</td>
<td>8</td>
<td>d</td>
<td>Double precision</td>
</tr>
<tr>
<td>long double</td>
<td>6.172L</td>
<td>16(10)</td>
<td>t</td>
<td>Extended precision</td>
</tr>
</tbody>
</table>
Arithmetic and logical: add, sub, mult, and, or, not, cmp, ...
- subq %rdx, %rax (%rax = %rax - %rdx)

Shift/rotate instructions: sar, sal, shr, shl ...
- sar and sal are arithmetic (signed) shift.

Control transfer: call, ret, jmp, j<condition>, ...

Data-transfer: mov, push, pop, ...
- Careful: Results of 32-bit operations are implicitly zero-extended to 64-bit values, unlike results of 8- and 16-bit operations.
- movl $-1, %eax ≡ movq 0x00000000ffffffff, %rax
- To preserve the sign bit:
  movslq %eax, %rdx (move sign extended)

See http://www.x86-64.org/documentation/assembly.html.
X86–64 Addressing Modes

Register:
  addq %rbx, %rax // value in register rax

Direct:
  movq 0x172, %rdi // contents at address 172 hex (370 in dec)

Immediate:
  movq $172, %rdi // value 172

Register indirect:
  movq %rbx, (%rax) // data at addr in reg

Register indexed:
  movq $6, 172(%rax) // address is 172+%rax

Base indexed scale displacement:
  - base and index are registers
  - scale is 2, 4, or 8 (absent implies 1)
  - displacement is 8-, 16-, or 32-bit value
    addq 172(%rdi,%rdx,8), %rax // address is %rdi+8*rdx+172

Instruction-pointer relative:
  movq 6(%rip), %rax

Only one operand may address memory.
Outline

X86–64 Assembly Primer
Overview of Computer Architecture
Profiling a Program
Set of Example Programs
Computer Architecture Overview

Instructions
Memory System
Processor Bus and IO Subsystem
Disk System
GPU and Graphics System
Network

courtesy of Intel Corp.

© 2013 Charles E. Leiserson, Saman P. Amarasinghe and I-Ting Angelina Lee
Instructions
Memory System

courtesy of Intel Corp.

© 2013 Charles E. Leiserson, Saman P. Amarasinghe and I–Ting Angelina Lee 12
20–24 stage Pipeline
Instruction Execution

<table>
<thead>
<tr>
<th>Instruction #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instruction Execution

IF: Instruction fetch  ID: Instruction decode
EX: Execution        MEM: Memory access
WB: Write back

Cycles

<table>
<thead>
<tr>
<th>Instruction #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i+3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Instruction i+4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IF</td>
<td>ID</td>
</tr>
</tbody>
</table>

© 2013 Charles E. Leiserson, Saman P. Amarasinghe and I-Ting Angelina Lee
Pipelining Execution

IF: Instruction fetch  ID : Instruction decode
EX : Execution        MEM: Memory access
WB : Write back

Cycles

Instruction #  1  2  3  4  5  6  7  8  9  10

Instruction i  IF  ID  EX  MEM  WB
Instruction i+1 IF  ID  EX  MEM  WB
Instruction i+2 IF  ID  EX  MEM  WB
Instruction i+3 IF  ID  EX  MEM  WB
Instruction i+4 IF  ID  EX  MEM  WB
Hazards prevent next instruction from executing during its designated clock cycle

- **Structural hazards**: attempt to use the same hardware to do two different things at once
- **Data hazards**: Instruction depends on result of prior instruction still in the pipeline
- **Control hazards**: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).
Data Hazards: True Dependence

Instr\textsubscript{j} is data dependent (aka true dependence) on Instr\textsubscript{i}:

\begin{align*}
\text{addl } & \text{rbx, rax} \\
\text{J: subl } & \text{rax, rcx}
\end{align*}

If two instructions are data dependent, they cannot execute simultaneously, be completely overlapped or execute in out-of-order.

If data dependence caused a hazard in pipeline, called a Read After Write (RAW) hazard.
**Benefits of Unrolling**

```c
int A[1000000];
int B[1000000];
test()
{
    int i;
    for(i=0; i < 1000000; i++)
}

xorl %edx, %edx

..B1.2:
    movl B(%rdx), %eax
    addl %eax, A(%rdx)
    addq $4, %rdx
    cmpq $4000000, %rdx
    jl ..B1.2

..B1.3:
    ret
```

For(i=0; i<N; i += 4) {
    A[i+1] = A[i+1] + 1
}

```c
xorl %edi, %edi

..B1.2:
    movl B(%rdi), %eax
    movl 4+B(%rdi), %edx
    movl 8+B(%rdi), %ecx
    movl 12+B(%rdi), %esi
    addl %eax, A(%rdi)
    addl %edx, 4+A(%rdi)
    addl %ecx, 8+A(%rdi)
    addl %esi, 12+A(%rdi)
    addq $16, %rdi
    cmpq $4000000, %rdi
    jl ..B1.2

..B1.3:
    ret
```
Name dependence: when 2 instructions use same register or memory location, called a name, but no flow of data between the instructions associated with that name; 2 versions of name dependence

Instr\textsubscript{j} writes operand \textit{before} Instr\textsubscript{i} reads it

\[
\begin{align*}
\text{subl} & \quad \textit{rax}, \text{rbx} \\
\text{addl} & \quad \textit{rcx}, \textit{rax}
\end{align*}
\]

Called an “anti-dependence” by compiler writers. This results from reuse of the name “rax”

If anti-dependence caused a hazard in the pipeline, called a Write After Read (WAR) hazard
Name Dependence #2: Output dependence

Instr\(_j\) writes operand *before* Instr\(_i\) writes it.

\[
\begin{align*}
\text{subl} & \text{ rcx, } rax \\
\text{addl} & \text{ rbx, } rax
\end{align*}
\]

Called an “output dependence” by compiler writers. This also results from the reuse of name “rax”

If anti-dependence caused a hazard in the pipeline, called a Write After Write (WAW) hazard

Instructions involved in a name dependence can execute simultaneously if name used in instructions is changed so instructions do not conflict

- Register renaming resolves name dependence for registers
- Renaming can be done either by compiler or by HW
Control Hazards

Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order.

```plaintext
if p1 {
    S1;
} if p2 {
    S2;
}
```

S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1.

Control dependence need not be preserved
- willing to execute instructions that should not have been executed, thereby violating the control dependences, if can do so without affecting correctness of the program.

Speculative Execution
### Superscalar Execution

#### 2-issue super-scalar machine

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2013 Charles E. Leiserson, Saman P. Amarasinghe and I-Ting Angelina Lee
Finds Instruction Level Parallelism

- Multiple instructions issued in parallel

**HW/SW** must preserve program order: order instructions would execute in if executed sequentially as determined by original source program
- Dependences are a property of programs

**Importance of the data dependencies**
- 1) indicates the possibility of a hazard
- 2) determines order in which results must be calculated
- 3) sets an upper bound on how much parallelism can possibly be exploited

**Goal:** exploit parallelism by preserving program order only where it affects the outcome of the program
Can execute 6 Ops per cycle

3 Memory Operations
1 Load
1 Store address
1 Store data

3 Computational Operations
The floating-point hardware of a modern microprocessor incorporates vector hardware to process data in single-instruction stream, multiple-data stream (SIMD) fashion.

A vector unit with vector length $k$ consists of
- $k$ vector lanes, each containing scalar floating-point hardware, and
- a set of vector registers, each comprising $k$ words distributed across the lanes.

- Intel calls these MMX/SSE/SSE2/AVX
Multimedia Instructions

Packed data type

- Separate register file

![Diagram of packed data type and SIMD operation]

Single Instruction on Multiple Data (SIMD)

Figure 4: Packed Operation

© 2013 Charles E. Leiserson, Saman P. Amarasinghe and I-Ting Angelina Lee  courtesy of x86.org
Vector instructions generally operate in an elementwise fashion:

- The $i$th element of one vector register can only take part in operations with the $i$th element of other vector registers.
- All lanes perform exactly the same operation on their respective slices of the vector.
- Cross-lane operations are generally not allowed, although some architectures support access to unaligned contiguous memory at some cost in performance.
- Others support scatter/gather instructions which can access memory in random fashion, also generally at reduced performance.
Multimedia Instructions

Vectorization Converts Loops

```
for (l=0; l<=MAX; l++)
c[l]=a[l]+b[l];
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

|------|------|------|------|

Multimedia Instructions

```c
int A[1000000];
int B[1000000];
test()
{
    int i;
    for(i=0; i < 1000000; i++)
}
```

```

xorl  %edx, %edx
..B1.2:
    movq   A(%rax), %xmm0
    paddq B(%rax), %xmm0
    movsd  16+A(%rax), %xmm1
    paddq  16+B(%rax), %xmm1
    movsd  32+A(%rax), %xmm2
    paddq  32+B(%rax), %xmm2
    movsd  48+A(%rax), %xmm3
    paddq  48+B(%rax), %xmm3
    movsd  64+A(%rax), %xmm4
    paddq  64+B(%rax), %xmm4
    movsd  80+A(%rax), %xmm5
    paddq  80+B(%rax), %xmm5
    movsd  96+A(%rax), %xmm6
    paddq  96+B(%rax), %xmm6
    movsd  112+A(%rax), %xmm7
    paddq  112+B(%rax), %xmm7
    movdqa %xmm0, A(%rax)
    movdqa %xmm1, 16+A(%rax)
    movdqa %xmm2, 32+A(%rax)
    movdqa %xmm3, 48+A(%rax)
    movdqa %xmm4, 64+A(%rax)
    movdqa %xmm5, 80+A(%rax)
    movdqa %xmm6, 96+A(%rax)
    movdqa %xmm7, 112+A(%rax)
    addq  128, %rax
    cmpq  $4000000, %rdx
    jl    ..B1.2
..B1.3:
    ret
```

© 2013 Charles E. Leiserson, Saman P. Amarasinghe and I–Ting Angelina Lee
Vectorizable Loops

```c
void scale(double *A, size_t n, double x) {
    for (size_t i = 0; i < n; ++i) {
        A[i] *= x;
    }
}
```

- The loop’s bounds are known at runtime.
- All iterations of the loop do the same thing: no branches or function calls.
- The iterations of the loop are \textit{independent}: any given iteration \( i \) does not need any results from a previous loop iteration \( j < i \) in order to execute.
- If you specify the compiler switch 
  
  \texttt{-ftree-vectorizer-verbose=1}
  
  to GCC, it will produce a vectorization report telling which loops it vectorized.

© 2013 Charles E. Leiserson, Saman P. Amarasinghe and I-Ting Angelina Lee
Cilk Plus Vector Notation

void scale(double *A, size_t n, double x) {
    for (size_t i = 0; i < n; ++i) {
        A[i] *= x;
    }
}

void scale(double *A, size_t n, double x) {
    A[0:n] *= x;
}

• The notation $A[0:n]$ denotes an array section of length $n$ starting at index 0 of $A$.
• Operations performed on an array section are performed elementwise on every element of the array section.
Unvectorizable Loops

```c
double reduce(double *A, size_t n) {
    double sum = 0;
    for (size_t i=0; i<n; i++)
        sum += A[i];
    return sum;
}
```

• The iterations are not independent: each iteration \(i > 0\) depends on the previous iteration \(i - 1\) to update the variable `sum` before iteration \(i\) can.
• The compiler will not reorder the additions, because floating-point addition is not associative.
• The switch `–ffast-math` allows the GCC compiler to reorder the additions and vectorize the code using a technique called `strip mining`. 
**Strip Mining**

```c
double reduce(double *A, size_t n) {
    double sum = 0;
    for (size_t i=0; i<n; i++)
        sum += A[i];
    return sum;
}
```

**Original code**

**IDEA:** Add down using vector ops, and then sum subtotals across:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

```c
double reduce(double *A, size_t n) {
    // Assume that n is a multiple of 4
    double temp[4];
    for (size_t j=0; j<4; j++)
        temp[j] = 0;
    for (size_t i =0; i < n; i+=4) {
        for (size_t j =0 ;j <4; ++j) {
            temp[j] += A[i+j];
        }
        double sum = 0;
        // Sum the temporaries
        for (size_t j=0; j<4; ++j) {
            sum += temp[j];
        }
        return sum;
    }
}
```

**Strip mined for 4 lanes**
Out of Order Execution

Issue varying numbers of instructions per clock

- dynamically scheduled
  - Extracting ILP by examining 100’s of instructions
  - Scheduling them in parallel as operands become available
  - Rename registers to eliminate anti and dependences
  - out-of-order execution
  - Speculative execution
Out of order Execution

\[ A[i] = B[i] \]
\[ X[i] = Y[i] \]

I in %rdx
B in %rsi
A in %rdi
Y in %r8
X in %rcx

\[
\begin{align*}
\text{movl} & \quad (%rsi, %rdx, 4), %eax \\
\text{movl} & \quad %eax, (%rdi, %rdx, 4) \\
\text{movl} & \quad (%r8, %rdx, 4), %eax \\
\text{movl} & \quad %eax, (%rcx, %rdx, 4)
\end{align*}
\]
Out of order Execution

\[ A[I] = B[I] \]
\[ X[I] = Y[I] \]

I in %rd
B in %rsi
A in %rdi
Y in %r8
X in %rcx

\begin{verbatim}
movl (%rsi, %rdx, 4), %eax
movl %eax, (%rdi, %rdx, 4)
movl (%r8, %rdx, 4), %eax
movl %eax, (%rcx, %rdx, 4)
\end{verbatim}
Out of order Execution

\[ A[I] = B[I] \]
\[ X[I] = Y[I] \]

Register Renaming

1 in %rd
B in %rsi
A in %rdi
Y in %r8
X in %rcx

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source 1</th>
<th>Source 2</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl</td>
<td>(%rsi, %rdx, 4), %eax</td>
<td>%eax, (%rdi, %rdx, 4)</td>
<td>(%r8, %rdx, 4), %ebx</td>
</tr>
<tr>
<td></td>
<td>(%r8, %rdx, 4), %ebx</td>
<td>%ebx, (%rcx, %rdx, 4)</td>
<td></td>
</tr>
</tbody>
</table>
Out of order Execution

A[I] = B[I]
X[I] = Y[I]

I in %rdx
B in %rsi
A in %rdi
Y in %r8
X in %rcx

\[
\begin{array}{c}
\text{movl} \quad \%\text{rsi}, \%\text{rdx}, 4, \%\text{eax} \\
\text{movl} \quad \%\text{eax}, (\%\text{rdi}, \%\text{rdx}, 4) \\
\text{movl} \quad (\%\text{rdx}, \%\text{rdx}, 4), \%\text{ebx} \\
\text{movl} \quad \%\text{ebx}, (\%\text{rcx}, \%\text{rdx}, 4)
\end{array}
\]
Out of order Execution

\[ A[I] = B[I] \]
\[ X[I] = Y[I] \]

I in %rdx
B in %rsi
A in %rdi
Y in %r8
X in %rcx

\textbf{B[I] is a cache miss}

\[
\begin{align*}
\text{movl} & \quad (%rsi, %rdx, 4), %eax \\
\text{movl} & \quad (%r8, %rdx, 4), %ebx \\
\text{movl} & \quad %ebx, (%rcx, %rdx, 4) \\
\text{movl} & \quad %eax, (%rdi, %rdx, 4)
\end{align*}
\]
Out of order Execution

\[
\begin{align*}
A[i] &= B[i] \\
X[i] &= Y[i]
\end{align*}
\]

Assume \( A[i] = Y[i] \)

I in %rdx
B in %rsi
A in %rdi
Y in %r8
X in %rcx

\[
\begin{align*}
\text{movl} & \quad (%rsi, %rdx, 4), %eax \\
\text{movl} & \quad (%r8, %rdx, 4), %ebx \\
\text{movl} & \quad %ebx, (%rcx, %rdx, 4) \\
\text{movl} & \quad %eax, (%rdi, %rdx, 4)
\end{align*}
\]
Out of order Execution

\[ A[I] = B[I] \]
\[ X[I] = Y[I] \]

I in %rdx
B in %rsi
A in %rdi
Y in %r8
X in %rcx

Assume \( A[I] = Y[I] \)
Write buffer
either delay
or predict and squash
Speculation

Different predictors
- Branch Prediction
- Value Prediction
- Prefetching (memory access pattern prediction)

Greater ILP: Overcome control dependence by hardware speculating on outcome of branches and executing program as if guesses were correct
- Speculation $\Rightarrow$ fetch, issue, and execute instructions as if branch predictions were always correct
- Dynamic scheduling $\Rightarrow$ only fetches and issues instructions

Essentially a data flow execution model: Operations execute as soon as their operands are available
20 to 24 stage Pipeline

6 micro-ops issued at a time

128 micro-ops waiting to be executed
Branch Prediction and Speculative Execution

The diagram illustrates the execution of multiple instructions over several cycles, with a branch instruction indicated at the top. The execution stages are labeled as IF, ID, EX, MEM, and WB. Instruction #1 through #6 are shown, with the branch target decided at cycle 7.

© 2013 Charles E. Leiserson, Saman P. Amarasinghe and I–Ting Angelina Lee
Branch Prediction and Speculative Execution

<table>
<thead>
<tr>
<th>Instruction #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction i (branch)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>stall</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stall</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stall</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stall</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branch target decided
Branch Prediction and Speculative Execution

Build a predictor to figure out which direction branch is going
  • Today we have complex predictors with 99+% accuracy
  • Even predict the address in indirect branches / returns

Fetch and speculatively execute from the predicted address
  • No pipeline stalls

When the branch is finally decided, the speculative execution is confirmed or squashed
Complex predictor
Multiple predictors
- Use branch history
- Different algorithms
- Vote at the end

Indirect address predictor
Return address predictor

Nehalem is even more complicated!
The Principle of Locality:
- Program access a relatively small portion of the address space at any instant of time.

Two Different Types of Locality:
- **Temporal Locality** (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
- **Spatial Locality** (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straight-line code, array access)

Last 30 years, HW relied on locality for memory perf.
Levels of the Memory Hierarchy

**CPU Registers**
- 100s Bytes
- 300 – 500 ps (0.3-0.5 ns)

**L1 and L2 Cache**
- 10s-100s K Bytes
- ~1 ns - ~10 ns
- $1000s/ GByte

**Main Memory**
- G Bytes
- 80ns- 200ns
- ~ $100/ GByte

**Disk**
- 10s T Bytes, 10 ms (10,000,000 ns)
- ~ $1 / GByte

**Tape**
- infinite sec-min
- ~$1 / GByte
Cache Issues

Cold Miss
- The first time the data is available
- Prefetching may be able to reduce the cost

Capacity Miss
- The previous access has been evicted because too much data touched in between
- “Working Set” too large
- Reorganize the data access so reuse occurs before getting evicted.
- Prefetch otherwise

Conflict Miss
- Multiple data items mapped to the same location. Evicted even before cache is full
- Rearrange data and/or pad arrays

True Sharing Miss
- Thread in another processor wanted the data, it got moved to the other cache
- Minimize sharing/locks

False Sharing Miss
- Other processor used different data in the same cache line. So the line got moved
- Pad data and make sure structures such as locks don’t get into the same cache line
## Intel® Nehalem™ Microarchitecture – Memory Sub-system

### Intel 6 Core Processor

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 Data Cache</strong></td>
<td>32 KB</td>
<td>64 bytes</td>
<td>4 ns</td>
<td>8-way</td>
</tr>
<tr>
<td><strong>L1 Instruction Cache</strong></td>
<td>32 KB</td>
<td>64 bytes</td>
<td>4 ns</td>
<td>4-way</td>
</tr>
<tr>
<td><strong>L2 Cache</strong></td>
<td>256 KB</td>
<td>64 bytes</td>
<td>10 ns</td>
<td>8-way</td>
</tr>
<tr>
<td><strong>L3 Cache</strong></td>
<td>12 MB</td>
<td>64 bytes</td>
<td>50 ns</td>
<td>16-way</td>
</tr>
<tr>
<td><strong>Main Memory</strong></td>
<td>64 bytes</td>
<td>75 ns</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Memory Consistency

\[ A[I] = B[I] \]
\[ X[I] = Y[I] \]

I in %rdx
B in %rsi
A in %rdi
Y in %r8
X in %rcx

```asm
movl (%rsi, %rdx, 4), %eax
movl %eax, (%rdi, %rdx, 4)
movl (%r8, %rdx, 4), %ebx
movl %ebx, (%rcx, %rdx, 4)
```

© 2013 Charles E. Leiserson, Saman P. Amarasinghe and I–Ting Angelina Lee
Memory Consistency

A[I] = B[I]
X[I] = Y[I]

I in %rdx
B in %rsi
A in %rdi
Y in %r8
X in %rcx

What happens if another core is looking?

movl (%rsi, %rdx, 4), %eax
movl %eax, (%rdi, %rdx, 4)
movl (%r8, %rdx, 4), %ebx
movl %ebx, (%rcx, %rdx, 4)

rd B
rd Y
wr A
wr X
Memory Consistency

A[i] = .... while(!done);
done = 1 B = A[i]
A[i] = ....
done = 1

while(!done);
B = A[i]

wr A[i]
rd done
rd done
rd done
rd A[i]
A[i] = …. while(!done);
done = 1

while(!done);
B = A[i]

wr A[i]

rd done

rd done

rd done

rd A[i]
Memory Consistency

- Reordering of reads and writes in OOO can create an inconsistent view from the outside.
- If it matters to the outside make sure that all the memory references are done before the communication is initiated.
- Memory fence instructions
  - Pros: get rid of the inconsistency
  - Cons: expensive

```plaintext
A[i] = ... while(!done);
mem fence
B = A[i]
done = 1
```
Outline

X86–64 Assembly Primer
Overview of Computer Architecture
Profiling a Program
Set of Example Programs
Helps you identify and characterize performance issues by:

- Collecting performance data from the system running your application.
- Organizing and displaying the data in a variety of interactive views, from system-wide down to source code or processor instruction perspective.
- Identifying potential performance issues and suggesting improvements.

- Example: Intel Vtune, gprof, oprofile, perf
What Is a Hotspot?

Where in an application or system there is a significant amount of activity

- **Where** = address in memory
  - => OS process
  - => OS thread
  - => executable file or *module*
  - => user function (requires symbols)
  - => line of source code (requires symbols with line numbers) or assembly instruction

- **Significant** = activity that occurs infrequently probably does not have much impact on system performance

- **Activity** = time spent or other internal processor event
  - Examples of other events: Cache misses, branch mispredictions, floating-point instructions retired, partial register stalls, and so on.
Two Ways to Track Location

Problem: I need to know where you spend most of your time.

Statistical Solution: I call you on your cellular phone every 30 minutes and ask you to report your location. Then I plot the data as a histogram.

Instrumentation Solution: I install a special phone booth at the entrance of every site you plan to visit. As you enter or exit every site, you first go into the booth, call the operator to get the exact time, and then call me and tell me where you are and when you got there.
Sampling Collector

Periodically interrupt the processor to obtain the execution context

- **Time-based sampling (TBS)** is triggered by:
  - Operating system timer services.
  - Every $n$ processor clockticks.
- **Event-based sampling (EBS)** is triggered by processor event counter overflow.
  - These events are processor-specific, like L2 cache misses, branch mispredictions, floating-point instructions retired, and so on.
No Installation Required

- No need to install a phone everywhere you want a man in the field to make a report.

Wide Coverage

- Assuming all his territory has cellular coverage, you can track him wherever he goes.

Low Overhead

- Answering his cellular telephone once in a while, reporting his location, and returning to other tasks do not take much of his time.
The Statistical Solution: Disadvantages

Approximate Precision:
- A number of factors can influence exactly how long he takes to answer the phone.

Limited Report:
- Insufficient time to find out how he got to where he is or where he has been since you last called him.

Statistical Significance: There are some places you might not locate him, if he does not go there often or he does not stay very long. Does that really matter?
Perfect Accuracy

- I know where you were immediately before and after your visit to each customer.
- I can calculate how much time you spent at each customer site.
- I know how many times you visited each customer site.
The Instrumentation Solution: Disadvantages

Low Granularity
- Too coarse; the site is the site.

High Overhead
- You spend valuable time going to phone booths, calling operators, and calling me.

High Touch
- I have to build all those phone booths, which expands the space in each site you visit.
Events

Intel provide 100’s of types of events

- Can be very confusing (ex: “number of bogus branches”)
- Some useful event categories
  - Total instruction count and mix
  - Branch events
  - Load/store events
  - L1/L2 cache events
  - Prefetching events
  - TLB events
  - Multicore events
Use Event Ratios

In isolation, events may not tell you much.

Event ratios are dynamically calculated values based on events that make up the formula.

- Cycles per instruction (CPI) consists of clockticks and instructions retired.

There are a wide variety of predefined event ratios.
Outline

X86–64 Assembly Primer
Overview of Computer Architecture
Profiling a Program
Set of Example Programs
#define MAXA 10000
int maxa_half = MAXA/2;
int32_t  A[MAXA];
   //     [0, 1, 2, 3, 4,...] 
int32_t  incA[MAXA];
   //       [0..MAXA−1 randomly] 
int32_t  rndA[MAXA];
   // test j < maxa_half
for(j=0; j<MAXA; j++) {
    if(j < maxa_half) 
}
   // test div by 4
for(j=0; j<MAXA; j++) {
    if((j & 0x03) == 0) 
}
   // test incA[i] < maxa_half
for(j=0; j<MAXA; j++) {
    if(incA[j] < maxa_half) 
}
   // test rndA[i] < maxa_half
for(j=0; j<MAXA; j++) {
    if(rndA[j] < maxa_half) 
}
multiple passes over data

```c
for(j = 0; j < MAXA; j++)
```

```assembly
movl $A, %eax
movl $A+40000, %edx
```
```assembly
..B3.3:
    incl (%rax)
    addq $4, %rax
    cmpq %rdx, %rax
    jl ..B3.3
```
```assembly
movl maxa_half(%rip), %edx
xorl %ecx, %ecx
```
```assembly
..B3.2:
    cmpl %edx, %ecx
    jge ..B3.4
```
```assembly
..B3.3:
    movslq %ecx, %rax
    incl A(,%rax,4)
```
```assembly
..B3.4:
    incl %ecx
    cmpl $10000, %ecx
    jl ..B3.2
```
```assembly
test j < maxa_half
```
```c
for(j = 0; j < MAXA; j++) {
    if(j < maxa_half)
}
```
```assembly
movl maxa_half(%rip), %edx
```
```assembly
..B3.2:
    cmpl %edx, %ecx
    jge ..B3.4
```
```assembly
..B3.3:
    movslq %ecx, %rax
    incl A(,%rax,4)
```
```assembly
..B3.4:
    incl %ecx
    cmpl $10000, %ecx
    jl ..B3.2
```
Assembly listings

**test div by 4**

```c
for(j=0; j<MAXA; j++) {
    if((j & 0x03) == 0)
}
```

```assembly
xorl %edx, %edx
```

```c
for(j=0; j<MAXA; j++) {
    if(incA[j] < maxa_half)
}
```

```assembly
movslq maxa_half(%rip), %rdx
xorl %ecx, %ecx
xorl %eax, %eax
```

```c
..B2.2:
testb $3, %dl
jne ..B2.4
```

```assembly
..B2.3:
movslq %edx, %rax
incl A(%rax,4)
```

```c
..B2.4:
incl %edx
cmpl $10000, %edx
jl ..B2.2
```

```assembly
..B2.3:
icl A(%rax)
```

```c
..B2.4:
addq $4, %rax
addq $1, %rcx
cmpq $10000, %rcx
jl ..B1.2
```

```assembly
..B1.2:
cmpq incA(%rcx,8), %rdx
jle ..B1.4
```

```assembly
..B1.3:
icl A(%rax)
```

```assembly
..B1.4:
addq $4, %rax
addq $1, %rcx
cmpq $10000, %rcx
jl ..B1.2
```
## results

<table>
<thead>
<tr>
<th></th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>multi pass over</td>
<td>1.00</td>
</tr>
<tr>
<td>test j &lt; maxa_half</td>
<td>1.26</td>
</tr>
<tr>
<td>test div by 4</td>
<td>2.21</td>
</tr>
<tr>
<td>test incA[i] &lt; maxa_half</td>
<td>1.33</td>
</tr>
<tr>
<td>test rndA[i] &lt; maxa_half</td>
<td>6.80</td>
</tr>
</tbody>
</table>

### multiple passes over data
```c
for(j=0; j<MAXA; j++)
```

### test j < maxa_half
```c
for(j=0; j<MAXA; j++) {
    if(j < maxa_half)
}
```

### test div by 4
```c
for(j=0; j<MAXA; j++) {
    if((j & 0x03) == 0)
}
```

### test incA[i] < maxa_half
```c
for(j=0; j<MAXA; j++) {
    if(incA[j] < maxa_half)
}
```

### test rndA[i] < maxa_half
```c
for(j=0; j<MAXA; j++) {
    if(rndA[j] < maxa_half)
}
```
## Results

<table>
<thead>
<tr>
<th>Multi Pass Over</th>
<th>Runtime (ms)</th>
<th>INST_RETIRED.ANY events</th>
</tr>
</thead>
<tbody>
<tr>
<td>test j &lt; maxa_half</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>test div by 4</td>
<td>1.26</td>
<td>1.50</td>
</tr>
<tr>
<td>test incA[i] &lt; maxa_half</td>
<td>2.21</td>
<td>1.37</td>
</tr>
<tr>
<td>test rndA[i] &lt; maxa_half</td>
<td>1.33</td>
<td>1.63</td>
</tr>
</tbody>
</table>

**INST_RETIRED.ANY ** Instructions retired.

This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.

Multiple passes over data

```c
for(j=0; j<MAXA; j++)

for(j=0; j<MAXA; j++)
    if(j < maxa_half)
```

Test div by 4

```c
for(j=0; j<MAXA; j++)
    if((j & 0x03) == 0)
```

Test incA[i] < maxa_half

```c
for(j=0; j<MAXA; j++)
    if(incA[j] < maxa_half)
```

Test rndA[i] < maxa_half

```c
for(j=0; j<MAXA; j++)
    if(rndA[j] < maxa_half)
```
### Results

<table>
<thead>
<tr>
<th></th>
<th>Runtime (ms)</th>
<th>INST RETIRED.ANY</th>
<th>INST RETIRED.LOADS</th>
<th>BR_INST_RETIRED.ANY</th>
<th>Inst Retired (ANY - LOAD - BR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>multi pass over</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>test j &lt; maxa_half</td>
<td>1.26</td>
<td>1.50</td>
<td>0.50</td>
<td>1.99</td>
<td>1.75</td>
</tr>
<tr>
<td>test div by 4</td>
<td>2.21</td>
<td>1.37</td>
<td>0.25</td>
<td>1.99</td>
<td>1.62</td>
</tr>
<tr>
<td>test incA[i] &lt; maxa_half</td>
<td>1.33</td>
<td>1.63</td>
<td>1.50</td>
<td>2.00</td>
<td>1.50</td>
</tr>
<tr>
<td>test rndA[i] &lt; maxa_half</td>
<td>6.80</td>
<td>1.63</td>
<td>1.50</td>
<td>1.99</td>
<td>1.50</td>
</tr>
</tbody>
</table>

**INST RETIRED.LOADS**  Instructions retired, contain a load  
**INST RETIRED.STORE**  Instructions retired, contain a store  
**BR_INST_RETIRED.ANY**  Number of branch instructions retired

```c
for(j=0; j<MAXA; j++)
```

```c
test j < maxa_half
for(j=0; j<MAXA; j++) {
    if(j < maxa_half)
}
```

```c
test div by 4
for(j=0; j<MAXA; j++) {
    if((j & 0x03) == 0)
}
```

```c
test incA[i] < maxa_half
for(j=0; j<MAXA; j++) {
    if(incA[j] < maxa_half)
}
```

```c
test rndA[i] < maxa_half
for(j=0; j<MAXA; j++) {
    if(rndA[j] < maxa_half)
}
```
## results

<table>
<thead>
<tr>
<th></th>
<th>Runtime (ms)</th>
<th>INST_RETIRED.ANY</th>
<th>Clocks per Instructions Retired - CPI</th>
<th>CPI*Tot Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>multi pass over</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>test j &lt; maxa_half</td>
<td>1.26</td>
<td>1.50</td>
<td>0.84</td>
<td>1.25</td>
</tr>
<tr>
<td>test div by 4</td>
<td>2.21</td>
<td>1.37</td>
<td>1.60</td>
<td>2.19</td>
</tr>
<tr>
<td>test incA[i] &lt; maxa_half</td>
<td>1.33</td>
<td>1.63</td>
<td>0.82</td>
<td>1.33</td>
</tr>
<tr>
<td>test rndA[i] &lt; maxa_half</td>
<td>6.80</td>
<td>1.63</td>
<td>4.17</td>
<td>6.78</td>
</tr>
</tbody>
</table>

**CPI**  
\[ \text{CPI} = \frac{\text{CPU_CLK_UNHALTED.CORE}}{\text{INST_RETIRED.ANY}} \]

High CPI indicates that instructions require more cycles to execute than they should. In this case there may be opportunities to modify your code to improve the efficiency with which instructions are executed within the processor. CPI can get as low as 0.25 cycles per instructions.
### BR_INST_RETIRED.MISPRED

This event counts the number of retired branch instructions that were mispredicted by the processor. A branch misprediction occurs when the processor predicts that the branch would be taken, but it is not, or vice-versa. ....

<table>
<thead>
<tr>
<th></th>
<th>Runtime (ms)</th>
<th>BR_INST_RETIRED.MISPRED %</th>
</tr>
</thead>
<tbody>
<tr>
<td>multi pass over</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>test j &lt; maxa_half</td>
<td>1.26</td>
<td>2.50</td>
</tr>
<tr>
<td>test div by 4</td>
<td>2.21</td>
<td>400.00</td>
</tr>
<tr>
<td>test incA[i] &lt; maxa_half</td>
<td>1.33</td>
<td>2.00</td>
</tr>
<tr>
<td>test rndA[i] &lt; maxa_half</td>
<td>6.80</td>
<td>2134.00</td>
</tr>
</tbody>
</table>

#### Multiple passes over data

```c
for(j=0; j<MAXA; j++)

test j < maxa_half
for(j=0; j<MAXA; j++) {
    if(j < maxa_half)
}

test div by 4
for(j=0; j<MAXA; j++) {
    if((j & 0x03) == 0)
}

test incA[i] < maxa_half
for(j=0; j<MAXA; j++) {
    if(incA[j] < maxa_half)
}

test rndA[i] < maxa_half
for(j=0; j<MAXA; j++) {
    if(rndA[j] < maxa_half)
}
```
## results

<table>
<thead>
<tr>
<th></th>
<th>Runtime (m.s)</th>
<th>INST_RETIRED.ANY events</th>
<th>BR_INST_RETIRED.MIS PRED %</th>
<th>&quot;Instructions wasted&quot; of mispredicted branches</th>
<th>Total &quot;Cost&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>multi pass over</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>test j &lt; maxa_half</td>
<td>1.26</td>
<td>1.50</td>
<td>2.50</td>
<td>4.97</td>
<td>1.50</td>
</tr>
<tr>
<td>test div by 4</td>
<td>2.21</td>
<td>1.37</td>
<td>400.00</td>
<td>797.51</td>
<td>2.21</td>
</tr>
<tr>
<td>test incA[i] &lt; maxa_half</td>
<td>1.33</td>
<td>1.63</td>
<td>2.00</td>
<td>3.99</td>
<td>1.63</td>
</tr>
<tr>
<td>test rndA[i] &lt; maxa_half</td>
<td>6.80</td>
<td>1.63</td>
<td>2134.00</td>
<td>4254.69</td>
<td>6.10</td>
</tr>
</tbody>
</table>

Assume the cost of a mispredicted branch is 21 "instructions wasted"

- Number 21 got the closest answer