Assembly Language: Part 2
Agenda

Flattened C code

Control flow with signed integers
Control flow with unsigned integers
Assembly Language: Defining global data
Arrays
Structures
Flattened C Code

Problem
- Translating from C to assembly language is difficult when the C code contains **nested** statements

Solution
- **Flatten** the C code to eliminate all nesting
C

```c
if (expr)
{  statement1;
    ...
    statementN;
}

if (expr)
{  statementT1;
    ...
    statementTN;
}
else
{  statementF1;
    ...
    statementFN;
}
```

**Flattened C**

```c
if (! expr) goto endif1;
    statement1;
    ...
    statementN;
endif1:

if (! expr) goto else1;
    statement1;
    ...
    statementN;
else1:
    statementF1;
    ...
    statementFN;
endif1:
```
Flattened C Code

C
while (expr)
{  statement1;
    ...
    statementN;
}

for (expr1; expr2; expr3)
{  statement1;
    ...
    statementN;
}

Flattened C

loop1:
    if (! expr) goto endloop1;
    statement1;
    ...
    statementN;
    goto loop1;
endloop1:

expr1;
loop1:
    if (! expr2) goto endloop1;
    statement1;
    ...
    statementN;
    expr3;
    goto loop1;
endloop1:

See Bryant & O’Hallaron book for faster patterns
Agenda

- Flattened C code
- Control flow with signed integers
- Control flow with unsigned integers
- Assembly Language: Defining global data
- Arrays
- Structures
if Example

C

```c
int i;
...
if (i < 0)
    i = -i;
```

Flattened C

```c
int i;
...
if (i >= 0) goto endif1;
    i = -i;
endif1:
```
if Example

Flattened C

```c
int i;
...
    if (i >= 0) goto endif1;
    i = -i;
endif1:
```

Assem Lang

```assembly
.section ".bss"
i: .skip 4
...
.section ".text"
...
    cmpl $0, i
    jge  endif1
    negl i
endif1:
```

Note:
- `cmp` instruction (counterintuitive operand order)
  Sets CC bits in EFLAGS register
- `jge` instruction (conditional jump)
  Examines CC bits in EFLAGS register
**if-else Example**

### C

```
int i;
int j;
int smaller;
...
if (i < j)
    smaller = i;
else
    smaller = j;
```

### Flattened C

```
int i;
int j;
int smaller;
...
if (i >= j) goto else1;
    smaller = i;
    goto endif1;
else1:
    smaller = j;
endif1:
```
if...else Example

Flattened C

```c
int i;
int j;
int smaller;
...
if (i >= j) goto else1;
    smaller = i;
goto endif1;
else1:
    smaller = j;
endif1:
```

Assem Lang

```assembly
.section ".bss"
i:      .skip 4
j:      .skip 4
smaller: .skip 4
...
.section ".text"
...
movl i, %eax
cmpl j, %eax
jge else1
movl i, %eax
movl %eax, smaller
jmp endif1
else1:
    movl j, %eax
    movl %eax, smaller
endif1:
```

Note:

**jmp** instruction
(unconditional jump)
```
int fact;
int n;
...
fact = 1;
while (n > 1)
{ fact *= n;
  n--;
}
```

```
int fact;
int n;
...
fact = 1;
loop1:
  if (n <= 1) goto endloop1;
  fact *= n;
  n--;
  goto loop1;
endloop1:
```
while Example

Flattened C

```c
int fact;
int n;
...
    fact = 1;
loop1:
    if (n <= 1) goto endloop1;
    fact *= n;
    n--;
    goto loop1;
endloop1:
```

Assem Lang

```assembly
.section " .bss"

fact: .skip 4
n:    .skip 4
...

.section " .text"
...

    movl $1, fact
loop1:
    cmpl $1, n
    jle endloop1
    movl fact, %eax
    imull n
    movl %eax, fact
    decl n
    jmp loop1
endloop1:
```

Note:
- **jle** instruction (conditional jump)
- **imul** instruction
for Example

C

```c
int power = 1;
int base;
int exp;
int i;
...
for (i = 0; i < exp; i++)
    power *= base;
```

Flattened C

```c
int power = 1;
int base;
int exp;
int i;
...
i = 0;
loop1:
    if (i >= exp) goto endloop1;
    power *= base;
    i++;
goto loop1;
endloop1:
```
for Example

Flattened C

```c
int power = 1;
int base;
int exp;
int i;
...
    i = 0;
loop1:
    if (i >= exp) goto endloop1;
    power *= base;
    i++;
    goto loop1;
endloop1:
```

Assem Lang

```assembly
.sectio n ".data"
power: .long 1

.sectio n ".bss"
base: .skip 4
exp: .skip 4
i: .skip 4
...
.sectio n ".text"
...
    movl $0, i
loop1:
    movl i, %eax
    cmpl exp, %eax
    jge  endloop1
    movl power, %eax
    imull base
    movl %eax, power
    incl i
    jmp loop1
endloop1:
```
Comparing signed integers

\texttt{cmp\{q,l,w,b\} \ srcIRM, \ destRM} \hspace{1cm} \text{Compare dest with src}

- Sets condition-code bits in the EFLAGS register
- Beware: operands are in counterintuitive order
- Beware: many other instructions set condition-code bits
  - Conditional jump should \textit{immediately} follow \texttt{cmp}
Control Flow with Signed Integers

Unconditional jump

\textbf{jmp label} \quad \text{Jump to label}

Conditional jumps after comparing signed integers

\begin{itemize}
  \item \textbf{je label} \quad \text{Jump to label if equal}
  \item \textbf{jne label} \quad \text{Jump to label if not equal}
  \item \textbf{jl label} \quad \text{Jump to label if less}
  \item \textbf{jle label} \quad \text{Jump to label if less or equal}
  \item \textbf{jg label} \quad \text{Jump to label if greater}
  \item \textbf{jge label} \quad \text{Jump to label if greater or equal}
\end{itemize}

• Examine CC bits in EFLAGS register
Agenda

Flattened C
Control flow with signed integers

Control flow with unsigned integers
Assembly Language: Defining global data
Arrays
Structures
Signed vs. Unsigned Integers

In C
- Integers are signed or unsigned
- Compiler generates assembly language instructions accordingly

In assembly language
- Integers are neither signed nor unsigned
- Distinction is in the instructions used to manipulate them

Distinction matters for
- Multiplication and division
- Control flow
Handling Unsigned Integers

Multiplication and division
- Signed integers: \texttt{imul}, \texttt{idiv}
- Unsigned integers: \texttt{mul}, \texttt{div}

Control flow
- Signed integers: \texttt{cmp} + \{\texttt{je}, \texttt{jne}, \texttt{jl}, \texttt{jle}, \texttt{jg}, \texttt{jge}\}

  Unsigned integers: “unsigned \texttt{cmp}” + \{\texttt{je}, \texttt{jne}, \texttt{jl}, \texttt{jle}, \texttt{jg}, \texttt{jge}\}? \ No!!!
- Unsigned integers: \texttt{cmp} + \{\texttt{je}, \texttt{jne}, \texttt{jb}, \texttt{jbe}, \texttt{ja}, \texttt{jae}\}
C

unsigned int fact;
unsigned int n;
...
fact = 1;
while (n > 1)
{
    fact *= n;
    n--;
}

Flattened C

unsigned int fact;
unsigned int n;
...
    fact = 1;
loop1:
    if (n <= 1) goto endloop1;
    fact *= n;
    n--;
    goto loop1;
endloop1:
while Example

**Flattened C**

```c
unsigned int fact;
unsigned int n;
...
  fact = 1;
loop1:
  if (n <= 1) goto endloop1;
  fact *= n;
  n--;
  goto loop1;
endloop1:
```

**Assem Lang**

```assembly
.section " .bss"
fact: .skip 4
n: .skip 4
...
.section " .text"
...
  movl $1, fact
loop1:
  cmpl $1, n
  jbe endloop1
  movl fact, %eax
  mull n
  movl %eax, fact
  decl n
  jmp loop1
endloop1:
```

Note:
- `jbe` instruction (instead of `jle`)
- `mull` instruction (instead of `imull`)
for Example

C

unsigned int power = 1;
unsigned int base;
unsigned int exp;
unsigned int i;
...
for (i = 0; i < exp; i++)
  power *= base;

Flattened C

unsigned int power = 1;
unsigned int base;
unsigned int exp;
unsigned int i;
...
i = 0;
loop1:
  if (i >= exp) goto endloop1;
  power *= base;
i++;
goto loop1;
endloop1:
for Example

Flattened C

unsigned int power = 1;
unsigned int base;
unsigned int exp;
unsigned int i;
...
i = 0;
loop1:
    if (i >= exp) goto endloop1;
    power *= base;
i++;
goto loop1;
endloop1:

Note:
jae instruction (instead of jge)
mull instruction (instead of imull)

Assem Lang

.section ".data"
power: .long 1
    .section ".bss"
base: .skip 4
exp: .skip 4
i: .skip 4
...
    .section ".text"
...
    movl $0, i
loop1:
    movl i, %eax
    cmpl exp, %eax
    jae endloop1
    movl power, %eax
    mull base
    movl %eax, power
    incl i
    jmp loop1
endloop1:
Control Flow with Unsigned Integers

Comparing unsigned integers

```
cmp{q,l,w,b} srcIRM, destRM  Compare dest with src
```

(Same as comparing signed integers)

Conditional jumps after comparing unsigned integers

```
je label    Jump to label if equal
jne label   Jump to label if not equal
jb label    Jump to label if below
jbe label   Jump to label if below or equal
ja label    Jump to label if above
jae label   Jump to label if above or equal
```

- Examine CC bits in EFLAGS register
Agenda

Flattened C code
Control flow with signed integers
Control flow with unsigned integers
Assembly Language: Defining global data
Arrays
Structures
RAM (Random Access Memory)
Defining Data: DATA Section 1

```c
static char c = 'a';
static short s = 12;
static int i = 345;
static long l = 6789;
```

```c
.section " .data"
c:
    .byte 'a'
s:
    .word 12
i:
    .long 345
l:
    .quad 6789
```

Note:
- `.section` instruction (to announce DATA section)
- Label definition (marks a spot in RAM)
- `.byte` instruction (1 byte)
- `.word` instruction (2 bytes)
- `.long` instruction (4 bytes)
- `.quad` instruction (8 bytes)

Note:
Best to avoid “word” (2 byte) data
Defining Data: DATA Section 2

```c
char c = 'a';
short s = 12;
int i = 345;
long l = 6789;
```

```assembly
.section "".data"
.globl c
c: .byte 'a'
.globl s
s: .word 12
.globl i
i: .long 345
.globl l
l: .quad 6789
```

Note:
Can place label on same line as next instruction
.globl instruction
Defining Data: BSS Section

```
static char c;
static short s;
static int i;
static long l;
```

```
.section ".bss"

.c: .skip 1
.s: .skip 2
.i: .skip 4
.l: .skip 8
```

Note:

- `.section` instruction (to announce BSS section)
- `.skip` instruction
Defining Data: RODATA Section

..."hello\n"...;
...

Note:

```
.section " .rodata"
helloLabel:
 .string "hello\n"
```

.section instruction (to announce RODATA section)
.string instruction
Agenda

Flattened C
Control flow with signed integers
Control flow with unsigned integers
Assembly Language: Defining global data

Arrays
Structures
Arrays: Indirect Addressing

C

```c
int a[100];
int i;
int n;
...
i = 3;
...
n = a[i]
...
```

Assem Lang

```assembly
.section "\.bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section "\.text"
...
movl $3, i
...
movslq i, %rax
salq $2, %rax
addq $a, %rax
movl (%rax), %r10d
movl %r10d, n
...
```

One step at a time…
Arrays: Indirect Addressing

Assem Lang

```
.section ".bss"
  a: .skip 400
  i: .skip 4
  n: .skip 4
...
.section ".text"
...
movl $3, i
...
movslq i, %rax
salq $2, %rax
addq $a, %rax
movl (%rax), %r10d
movl %r10d, n
...
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>0</td>
</tr>
<tr>
<td>R10</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>n</td>
</tr>
</tbody>
</table>
Arrays: Indirect Addressing

Assem Lang

```
.section " .bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section " .text"
...
movl $3, i
...
movslq i, %rax
salq $2, %rax
addq $a, %rax
movl (%rax), %r10d
movl %r10d, n
...
```

Registers

<table>
<thead>
<tr>
<th>RAX</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>1004</td>
</tr>
<tr>
<td></td>
<td>1008</td>
</tr>
<tr>
<td></td>
<td>1012</td>
</tr>
<tr>
<td></td>
<td>1396</td>
</tr>
<tr>
<td></td>
<td>1400</td>
</tr>
<tr>
<td></td>
<td>1404</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1004</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1008</td>
</tr>
<tr>
<td>3</td>
<td>123</td>
<td>1012</td>
</tr>
<tr>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>3</td>
<td>1400</td>
</tr>
<tr>
<td>n</td>
<td></td>
<td>1404</td>
</tr>
</tbody>
</table>
Arrays: Indirect Addressing

Assem Lang

```assembly
section ".bss"
  a:    .skip 400
  i:    .skip 4
  n:    .skip 4
...
section ".text"
...
  movl $3, i
...
  movslq i, %rax
  salq $2, %rax
  addq $a, %rax
  movl (%rax), %r10d
  movl %r10d, n
...
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>12</td>
</tr>
<tr>
<td>R10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>1004</td>
</tr>
<tr>
<td>2</td>
<td>1008</td>
</tr>
<tr>
<td>3</td>
<td>123</td>
</tr>
<tr>
<td>1008</td>
<td>1012</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>1396</td>
</tr>
<tr>
<td>i</td>
<td>3</td>
</tr>
<tr>
<td>n</td>
<td>1404</td>
</tr>
<tr>
<td></td>
<td>1400</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>99</th>
<th>i</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1008</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1012</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1396</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1404</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Arrays: Indirect Addressing

Assem Lang

```plaintext
.section " .bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section " .text"
...
movl $3, i
...
movslq i, %rax
salq $2, %rax
addq $a, %rax
movl (%rax), %r10d
movl %r10d, n
...
```

Registers

<table>
<thead>
<tr>
<th>RAX</th>
<th>1012</th>
</tr>
</thead>
<tbody>
<tr>
<td>R10</td>
<td></td>
</tr>
</tbody>
</table>

Memory

```

| 0  | 1000 |
| 1  | 1004 |
| 2  | 1008 |
| 3  | 1012 |
| n  | 1404 |
```

| 99 | 1396 |
| i  | 1400 |
|    |      |
Arrays: Indirect Addressing

Assem Lang

```assembly
.section " .bss"
a: .skip 400
i: .skip 4
n: .skip 4
...  
.section " .text"
...  
movl $3, i
...  
movslq i, %rax
salq $2, %rax
addq $a, %rax
movl (%rax), %r10d
movl %r10d, n
...  
```

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>1012</td>
</tr>
<tr>
<td>R10</td>
<td>123</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>1004</td>
</tr>
<tr>
<td>2</td>
<td>1008</td>
</tr>
<tr>
<td>3</td>
<td>1012</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>99</td>
<td>1396</td>
</tr>
<tr>
<td>i</td>
<td>1400</td>
</tr>
<tr>
<td>n</td>
<td>1404</td>
</tr>
</tbody>
</table>

Note: Indirect addressing
Arrays: Indirect Addressing

Assem Lang

```assembly
.section " .bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section " .text"
...
movl $3, i
...
movslq i, %rax
salq $2, %rax
addq $a, %rax
movl (%rax), %r10d
movl %r10d, n
...
```

Registers

<table>
<thead>
<tr>
<th>RAX</th>
<th>1012</th>
</tr>
</thead>
<tbody>
<tr>
<td>R10</td>
<td>123</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>0</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1004</td>
</tr>
<tr>
<td>2</td>
<td>1008</td>
</tr>
<tr>
<td>3</td>
<td>1012</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>1396</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>1400</td>
</tr>
<tr>
<td>n</td>
<td>123</td>
</tr>
</tbody>
</table>

```
Arrays: Base+Disp Addressing

C

int a[100];
int i;
int n;
...
i = 3;
...
n = a[i]
...

Assem Lang

[section ".bss"

a: .skip 400
i: .skip 4
n: .skip 4
...
[section ".text"
...
movl $3, i
...
movl i, %eax
sal $2, %eax
movl a(%eax), %r10d
movl %r10d, n
...

One step at a time…
Arrays: Base+Disp Addressing

Assem Lang

```
.section "bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section "text"
...
movl $3, i
...
movl i, %eax
sal $2, %eax
movl a(%eax), %r10d
movl %r10d, n
...
```
Arrays: Base+Disp Addressing

Assem Lang

```asm
.section ".bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section ".text"
...movl $3, i...
movl i, %eax
...movl i, %eax
sall $2, %eax
movl a(%eax), %r10d
movl %r10d, n
...```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>3</td>
</tr>
<tr>
<td>R10</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Arrays: Base+Disp Addressing

Assem Lang

```
.section " .bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section " .text"
...
movl $3, i
...
movl i, %eax
sal $2, %eax
movl a(%eax), %r10d
movl %r10d, n
...
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX 12</td>
<td>0 1000</td>
</tr>
<tr>
<td>R10</td>
<td>1 1004</td>
</tr>
<tr>
<td></td>
<td>2 1008</td>
</tr>
<tr>
<td></td>
<td>3 1012</td>
</tr>
<tr>
<td></td>
<td>a 123</td>
</tr>
<tr>
<td></td>
<td>99 1396</td>
</tr>
<tr>
<td></td>
<td>i 1400</td>
</tr>
<tr>
<td></td>
<td>n 1404</td>
</tr>
</tbody>
</table>
Arrays: Base+Disp Addressing

Assem Lang

```
.section "\text{.bss}\"
\text{a}: \text{.skip 400}
\text{i}: \text{.skip 4}
\text{n}: \text{.skip 4}
...
\text{\texttt{.section \"\text{.text}\"}}
...\text{movl} \text{\$3, i}
...\text{movl i, } %\text{eax}
\text{\texttt{sal}l} \text{\$2, } %\text{eax}
\text{\texttt{movl}} \text{\texttt{a}(\%eax), } \%\text{r10d}
\text{\texttt{movl}} \%\text{r10d, n}
...
```

Registers

<table>
<thead>
<tr>
<th>RAX</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>R10</td>
<td>123</td>
</tr>
</tbody>
</table>

Memory

```
\begin{array}{c|c|c}
\text{a} & 0 & 1000 \\
 & 1 & 1004 \\
 & 2 & 1008 \\
 3 & 123 & 1012 \\
 & ... & \\
 99 & & 1396 \\
 i & 3 & 1400 \\
 n & & 1404 \\
\end{array}
```

Note:

Base+displacement addressing
Arrays: Base+Disp Addressing

Assem Lang

```
.section ".bss"
  a: .skip 400
  i: .skip 4
  n: .skip 4

... .section ".text"
... movl $3, i
... movl i, %eax
    sall $2, %eax
    movl a(%eax), %r10d
    movl %r10d, n
...```

Registers

<table>
<thead>
<tr>
<th>RAX</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>R10</td>
<td>123</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th></th>
<th>1000</th>
<th>1004</th>
<th>1008</th>
<th>1012</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>123</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>123</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
...```

```
Arrays: Scaled Indexed Addressing

---

C

```c
int a[100];
int i;
int n;
...
i = 3;
...
n = a[i]
...
```

Assem Lang

```assembly
.section ".bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section ".text"
...
movl $3, i
...
movl i, %eax
movl a(,%eax,4), %r10d
movl %r10d, n
...
```

One step at a time…
Arrays: Scaled Indexed Addressing

Assem Lang

```
.section ".bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section ".text"
...
movl $3, i
...
movl i, %eax
movl a(,%eax,4), %r10d
movl %r10d, n
...
```

 Registers

<table>
<thead>
<tr>
<th>RAX</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>99</td>
</tr>
<tr>
<td>i</td>
</tr>
<tr>
<td>n</td>
</tr>
</tbody>
</table>
Arrays: Scaled Indexed Addressing

Assem Lang

```assembly
.section "\bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section "\text"
...
movl $3, i
...
movl i, %eax
movl a(,%eax,4), %r10d
movl %r10d, n
...
```

 Registers

<table>
<thead>
<tr>
<th>RAX</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>R10</td>
<td></td>
</tr>
</tbody>
</table>

 Memory

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1000</td>
<td>1004</td>
<td>1008</td>
<td>1012</td>
</tr>
<tr>
<td>i</td>
<td>1396</td>
<td>1400</td>
<td>1404</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Arrays: Scaled Indexed Addressing

Assem Lang

```assembly
.section "bss"
a: .skip 400
i: .skip 4
n: .skip 4

.section "text"
...movl $3, i
...movl i, %eax
movl a(,%eax,4), %r10d
movl %r10d, n
...```

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>3</td>
</tr>
<tr>
<td>R10</td>
<td>123</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>1004</td>
</tr>
<tr>
<td>2</td>
<td>1008</td>
</tr>
<tr>
<td>3</td>
<td>1012</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>1396</td>
</tr>
<tr>
<td>i</td>
<td>1400</td>
</tr>
<tr>
<td>n</td>
<td>1404</td>
</tr>
</tbody>
</table>

Note: Scaled indexed addressing
Arrays: Scaled Indexed Addressing

Assem Lang

```
.section " .bss"
a:  .skip 400
i:  .skip 4
n:  .skip 4
...
.section " .text"
...
movl $3, i
...
movl i, %eax
movl a(, %eax, 4), %r10d
movl %r10d, n
...
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX 12</td>
<td></td>
</tr>
<tr>
<td>R10 123</td>
<td></td>
</tr>
</tbody>
</table>

Memory:

<table>
<thead>
<tr>
<th></th>
<th>1000</th>
<th>1004</th>
<th>1008</th>
<th>1012</th>
<th>1396</th>
<th>1400</th>
<th>1404</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Generalization: Memory Operands

Full form of memory operands:

\( \text{displacement}(\text{base}, \text{index}, \text{scale}) \)

- **displacement** is an integer or a label (default = 0)
- **base** is a 4-byte or 8-byte register
- **index** is a 4-byte or 8-byte register
- **scale** is 1, 2, 4, or 8 (default = 1)

**Meaning**

- Compute the sum
  \((\text{displacement}) + (\text{contents of base}) + ((\text{contents of index}) \times (\text{scale}))\)
- Consider the sum to be an address
- Load from (or store to) that address

**Note:**

- All other forms are subsets of the full form…
Generalization: Memory Operands

Valid subsets:

- Direct addressing
  - displacement

- Indirect addressing
  - (base)

- Base+displacement addressing
  - displacement(base)

- Indexed addressing
  - (base, index)
  - displacement(base,index)

- Scaled indexed addressing
  - (,index, scale)
  - displacement(,index,scale)
  - (base,index,scale)
  - displacement(base,index,scale)
Operand Examples

Immediate operands
• $5 ⇒ use the number 5 (i.e. the number that is available immediately within the instruction)
• $i ⇒ use the address denoted by i (i.e. the address that is available immediately within the instruction

Register operands
• %rax ⇒ read from (or write to) register RAX

Memory operands: direct addressing
• 5 ⇒ load from (or store to) memory at address 5 (silly; seg fault)
• i ⇒ load from (or store to) memory at the address denoted by i

Memory operands: indirect addressing
• (%rax) ⇒ consider the contents of RAX to be an address; load from (or store to) that address
Operand Examples

Memory operands: **base+displacement addressing**
- \(5(\%rax)\) ⇒ compute the sum (5) + (contents of RAX); consider the sum to be an address; load from (or store to) that address
- \(i(\%rax)\) ⇒ compute the sum (address denoted by i) + (contents of RAX); consider the sum to be an address; load from (or store to) that address

Memory operands: **indexed addressing**
- \(5(\%rax,\%r10)\) ⇒ compute the sum (5) + (contents of RAX) + (contents of R10); consider the sum to be an address; load from (or store to) that address
- \(i(\%rax,\%r10)\) ⇒ compute the sum (address denoted by i) + (contents of RAX) + (contents of R10); consider the sum to be an address; load from (or store to) that address
Memory operands: **scaled indexed addressing**

- \(5(\%rax,\%r10,4)\) \(\Rightarrow\) compute the sum \((5) + (\text{contents of RAX}) + ((\text{contents of R10}) \times 4)\); consider the sum to be an address; load from (or store to) that address

- \(i(\%rax,\%r10,4)\) \(\Rightarrow\) compute the sum \((\text{address denoted by } i) + (\text{contents of RAX}) + ((\text{contents of R10}) \times 4)\); consider the sum to be an address; load from (or store to) that address
Aside: The lea Instruction

**lea**: load effective address

- Unique instruction: suppresses memory load/store

Example

- `movq 5(%rax), %r10`
  - Compute the sum (5) + (contents of RAX); consider the sum to be an address; load 8 bytes from that address into R10
- `leaq 5(%rax), %r10`
  - Compute the sum (5) + (contents of RAX); move that sum to R10

Useful for

- Computing an address, e.g. as a function argument
  - See precept code that calls `scanf()`
- Some quick-and-dirty arithmetic

What is the effect of this?

`leaq (%rax,%rax,4),%rax`
Agenda

- Flattened C
- Control flow with signed integers
- Control flow with unsigned integers
- Assembly Language: Defining global data
- Arrays
- Structures
Structures: Indirect Addressing

C

```c
struct S
{
    int i;
    int j;
};
...
struct S myStruct;
...
myStruct.i = 18;
...
myStruct.j = 19;
```

Assem Lang

```assembly
.section ".bss"
myStruct: .skip 8
...
.section ".text"
...
movq $myStruct, %rax
movl $18, (%rax)
...
movq $myStruct, %rax
addq $4, %rax
movl $19, (%rax)
```

Note:
Indirect addressing
C

```c
struct S
{
    int i;
    int j;
};
...
struct S myStruct;
...
myStruct.i = 18;
...
myStruct.j = 19;
```

Assem Lang

```asm
.section ".bss"
myStruct: .skip 8
...
.section ".text"
...
    movq $myStruct, %rax
    movl $18, 0(%rax)
    ...
    movl $19, 4(%rax)
```

rax

RAM

```
18
19
```
**Structures: Padding**

**C**

```c
struct S
{ char c;
  int i;
};
...
struct S myStruct;
...
myStruct.c = 'A';
...
myStruct.i = 18;
```

**Assem Lang**

```asm
.section ".bss"
myStruct: .skip 8
...  
.section ".text"
...  
  movq $myStruct, %rax
  movb $'A', 0(%rax)
...  
  movl $18, 4(%rax)
```

---

**Beware:**

Compiler sometimes inserts padding after fields

---

Three-byte pad here
x86-64/Linux rules

<table>
<thead>
<tr>
<th>Data type</th>
<th>Within a struct, must begin at address that is evenly divisible by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(unsigned) char</td>
<td>1</td>
</tr>
<tr>
<td>(unsigned) short</td>
<td>2</td>
</tr>
<tr>
<td>(unsigned) int</td>
<td>4</td>
</tr>
<tr>
<td>(unsigned) long</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>16</td>
</tr>
<tr>
<td>any pointer</td>
<td>8</td>
</tr>
</tbody>
</table>

- Compiler may add padding after last field if struct is within an array
Summary

Intermediate aspects of x86-64 assembly language…

Flattened C code

Control transfer with signed integers

Control transfer with unsigned integers

Arrays
  • Full form of instruction operands

Structures
  • Padding
Setting and using CC bits in EFLAGS register
Setting Condition Code Bits

Question
• How does $\text{cmp}\{q,l,w,b\}$ set condition code bits in EFLAGS register?

Answer
• (See following slides)
Condition Code Bits

Condition code bits

- ZF: zero flag: set to 1 iff result is zero
- SF: sign flag: set to 1 iff result is negative
- CF: carry flag: set to 1 iff unsigned overflow occurred
- OF: overflow flag: set to 1 iff signed overflow occurred
Condition Code Bits

Example: `addq src, dest`

- Compute sum `(dest+src)`
- Assign sum to `dest`
- ZF: set to 1 iff `sum == 0`
- SF: set to 1 iff `sum < 0`
- CF: set to 1 iff unsigned overflow
  - Set to 1 iff `sum<src`
- OF: set if signed overflow
  - Set to 1 iff
    - `(src>0 && dest>0 && sum<0) ||
      (src<0 && dest<0 && sum>=0)`
Condition Code Bits

Example: \texttt{subq src, dest}
- Compute sum (\texttt{dest+(-src)})
- Assign sum to \texttt{dest}
- ZF: set to 1 iff sum == 0
- SF: set to 1 iff sum < 0
- CF: set to 1 iff unsigned overflow
  - Set to 1 iff \texttt{dest<src}
- OF: set to 1 iff signed overflow
  - Set to 1 iff
    \[
    (\texttt{dest}>0 \&\& \texttt{src}<0 \&\& \texttt{sum}<0) \text{ || } \\
    (\texttt{dest}<0 \&\& \texttt{src}>0 \&\& \texttt{sum}\geq 0)
    \]

Example: \texttt{cmpq src, dest}
- Same as \texttt{subq}
- But does not affect \texttt{dest}
Using Condition Code Bits

Question
• How do conditional jump instructions use condition code bits in EFLAGS register?

Answer
• (See following slides)
### Conditional Jumps: Unsigned

After comparing **unsigned** data

<table>
<thead>
<tr>
<th>Jump Instruction</th>
<th>Use of CC Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>je label</td>
<td>ZF</td>
</tr>
<tr>
<td>jne label</td>
<td>~ZF</td>
</tr>
<tr>
<td>jb label</td>
<td>CF</td>
</tr>
<tr>
<td>jae label</td>
<td>~CF</td>
</tr>
<tr>
<td>jbe label</td>
<td>CF</td>
</tr>
<tr>
<td>ja label</td>
<td>~(CF</td>
</tr>
</tbody>
</table>

**Note:**
- If you can understand why `jb` jumps iff CF
- … then the others follow
Conditional Jumps: Unsigned

Why does jb jump iff CF? Informal explanation:

(1) largenum – smallnum (not below)
   • Correct result
   • ⇒ CF=0 ⇒ don’t jump

(2) smallnum – largenum (below)
   • Incorrect result
   • ⇒ CF=1 ⇒ jump
## Conditional Jumps: Signed

After comparing *signed* data

<table>
<thead>
<tr>
<th>Jump Instruction</th>
<th>Use of CC Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>je label</td>
<td>ZF</td>
</tr>
<tr>
<td>jne label</td>
<td>~ZF</td>
</tr>
<tr>
<td>jl label</td>
<td>OF ^ SF</td>
</tr>
<tr>
<td>jge label</td>
<td>~(OF ^ SF)</td>
</tr>
<tr>
<td>jle label</td>
<td>(OF ^ SF)</td>
</tr>
<tr>
<td>jg label</td>
<td>~((OF ^ SF)</td>
</tr>
</tbody>
</table>

**Note:**

- If you can understand why `jl` jumps iff OF^SF
- ... then the others follow
Conditional Jumps: Signed

Why does `jl` jump iff `OF^SF`? Informal explanation:

1. `largeposnum` – `smallposnum` (not less than)
   - Certainly correct result
   - \( \Rightarrow \) OF=0, SF=0, OF^SF==0 \( \Rightarrow \) don’t jump

2. `smallposnum` – `largeposnum` (less than)
   - Certainly correct result
   - \( \Rightarrow \) OF=0, SF=1, OF^SF==1 \( \Rightarrow \) jump

3. `largenegnum` – `smallnegnum` (less than)
   - Certainly correct result
   - \( \Rightarrow \) OF=0, SF=1 \( \Rightarrow \) (OF^SF)==1 \( \Rightarrow \) jump

4. `smallnegnum` – `largenegnum` (not less than)
   - Certainly correct result
   - \( \Rightarrow \) OF=0, SF=0 \( \Rightarrow \) (OF^SF)==0 \( \Rightarrow \) don't jump
Conditional Jumps: Signed

(5) posnum – negnum (not less than)
  • Suppose correct result
  • ⇒ OF=0, SF=0 ⇒ (OF^SF)==0 ⇒ don't jump

(6) posnum – negnum (not less than)
  • Suppose incorrect result
  • ⇒ OF=1, SF=1 ⇒ (OF^SF)==0 ⇒ don't jump

(7) negnum – posnum (less than)
  • Suppose correct result
  • ⇒ OF=0, SF=1 ⇒ (OF^SF)==1 ⇒ jump

(8) negnum – posnum (less than)
  • Suppose incorrect result
  • ⇒ OF=1, SF=0 ⇒ (OF^SF)==1 ⇒ jump
Appendix

Big-endian vs little-endian byte order
Byte Order

x86-64 is a **little endian** architecture

- **Least** significant byte of multi-byte entity is stored at lowest memory address
- “Little end goes first”

Some other systems use **big endian**

- **Most** significant byte of multi-byte entity is stored at lowest memory address
- “Big end goes first”

The int 5 at address 1000:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>00000101</td>
</tr>
<tr>
<td>1001</td>
<td>00000000</td>
</tr>
<tr>
<td>1002</td>
<td>00000000</td>
</tr>
<tr>
<td>1003</td>
<td>00000000</td>
</tr>
</tbody>
</table>

The int 5 at address 1000:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>00000000</td>
</tr>
<tr>
<td>1001</td>
<td>00000000</td>
</tr>
<tr>
<td>1002</td>
<td>00000000</td>
</tr>
<tr>
<td>1003</td>
<td>00000101</td>
</tr>
</tbody>
</table>
Byte Order Example 1

```c
#include <stdio.h>
int main(void)
{
    unsigned int i = 0x003377ff;
    unsigned char *p;
    int j;
    p = (unsigned char *)&i;
    for (j=0; j<4; j++)
        printf("Byte %d: %2x\n", j, p[j]);
}
```

Output on a little-endian machine:
- Byte 0: ff
- Byte 1: 77
- Byte 2: 33
- Byte 3: 00

Output on a big-endian machine:
- Byte 0: 00
- Byte 1: 33
- Byte 2: 77
- Byte 3: ff
Byte Order Example 2

Note:
Flawed code; uses “b” instructions to manipulate a four-byte memory area

x86-64 is **little** endian, so what will be the value of grade?

What would be the value of grade if x86-64 were **big** endian?

```assembly
.section ".data"
grade: .long 'B'
...

.section ".text"
...
# Option 1
movb grade, %al
subb $1, %al
movb %al, grade
...
# Option 2
subb $1, grade
```

Note: x86-64 is **little** endian, so what will be the value of grade?

x86-64 is **little** endian, so what will be the value of grade?
Note:
Flawed code; uses “l” instructions to manipulate a one-byte memory area

What would happen?