Assembly Language:
Part 2
Goals of this Lecture

Help you learn:

- Intermediate aspects of x86-64 assembly language…
- Control flow with signed integers
- Control flow with unsigned integers
- Arrays
- Structures
Agenda

Flattened C code

Control flow with signed integers
Control flow with unsigned integers
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
if (expr)
{
  statement1;
  ...
  statementN;
}
else
{
  statementF1;
  ...
  statementFN;
}

Flattened C
if (! expr) goto endif1;
  statement1;
  ...
  statementN;
endif1:

if (! expr) goto else1;
  statementT1;
  ...
  statementTN;
else1:
  statementF1;
  ...
  statementFN;
endif1:
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
Flattened C code

Control flow with signed integers

Control flow with unsigned integers

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
C

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

Flattened C

```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)
while Example

C

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

Flattened C

```c
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
.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
jge endloop1
movl power, %eax
imull base
movl %eax, power
incl i
jmp loop1
endloop1:
```
Control Flow with Signed Integers

Comparing signed integers

\[
\text{cmp\{q,l,w,b\} srcIRM, destRM} \quad \text{Compare dest with src}
\]

- 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

jmp label  Jump to label

Conditional jumps after comparing signed integers

je  label  Jump to label if equal
jne label  Jump to label if not equal
jl  label  Jump to label if less
jle label  Jump to label if less or equal
jg  label  Jump to label if greater
jge label  Jump to label if greater or equal

• Examine CC bits in EFLAGS register
Agenda

Flattened C

Control flow with signed integers

Control flow with unsigned integers

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}\}
while Example

C

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

Flattened C

```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**

```c
.section ".bss"
... fact: .skip 4
... n: .skip 4
...
    .section ".text"
... movl $1, fact
loop1:
    cmp1 $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`)

---

22
for Example

C

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

Flattened C

```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**

```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:
```

**Assem Lang**

```assembly
.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:
```

Note:

* jae instruction (instead of jge)
* mull instruction (instead of imull)
Control Flow with Unsigned Integers

Comparing unsigned integers

```plaintext
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
Control flow with signed integers
Control flow with unsigned integers

Arrays

Structures
Arrays: Indirect Addressing

C

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

Assem Lang

```asm
.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

```asm
.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>i</td>
<td>3</td>
</tr>
<tr>
<td>n</td>
<td></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     Memory

<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>

0  1  2  3  99  i  n
### 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>12</td>
<td></td>
</tr>
</tbody>
</table>

| R10 | 3 123 1012 |

#### Memory

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1000</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>1396</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**

```
.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 1012</td>
<td>0 1000</td>
</tr>
<tr>
<td>R10</td>
<td>1 1004</td>
</tr>
<tr>
<td>a 3 123</td>
<td>2 1008</td>
</tr>
<tr>
<td>a 99</td>
<td>3 1012</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>a 1012</td>
<td>99 1396</td>
</tr>
<tr>
<td>i 3</td>
<td>10004</td>
</tr>
<tr>
<td>n</td>
<td>10008</td>
</tr>
<tr>
<td></td>
<td>1400</td>
</tr>
<tr>
<td></td>
<td>1404</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
...
```

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
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>1012</td>
</tr>
<tr>
<td>R10</td>
<td>123</td>
</tr>
<tr>
<td>0</td>
<td>1008</td>
</tr>
<tr>
<td>3</td>
<td>1012</td>
</tr>
<tr>
<td>99</td>
<td>1396</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

```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
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
...
```

Registers

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

Memory

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1000</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td></td>
<td>1396</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: Base+Disp Addressing

Assem Lang

```assembly
.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>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>3</td>
</tr>
<tr>
<td>R10</td>
<td></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>99</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>1400</td>
</tr>
<tr>
<td>n</td>
<td>1404</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
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>12</td>
</tr>
<tr>
<td>R10</td>
<td></td>
</tr>
</tbody>
</table>

```
...
...
...movl $3, i
...movl i, %eax
sall $2, %eax
movl a(%eax), %r10d
movl %r10d, n
...```
Arrays: Base+Disp Addressing

Assem Lang

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

[section ".text"
  movl $3, i
  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>12</td>
</tr>
<tr>
<td>R10</td>
<td>123</td>
</tr>
</tbody>
</table>

<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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td></td>
<td>1396</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>

Note:

Base+displacement addressing
Arrays: Base+Disp Addressing

Assem Lang

```assembly
.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
...
```

<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 123</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>99 1396</td>
</tr>
<tr>
<td></td>
<td>i 1400</td>
</tr>
<tr>
<td></td>
<td>n 1404</td>
</tr>
</tbody>
</table>
Arrays: Scaled Indexed Addressing

C

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

Assem Lang

```asm
.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
...
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</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>3 123  1012</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>99 1396</td>
</tr>
<tr>
<td></td>
<td>1 1400</td>
</tr>
<tr>
<td></td>
<td>n 1404</td>
</tr>
</tbody>
</table>
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</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></td>
</tr>
<tr>
<td></td>
<td></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>

RAX = 3
R10 = a

Assem Lang code:

```
movl $3, i
movl i, %eax
movl a(%eax,4), %r10d
movl %r10d, n
```
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

- **RAX**: 3
- **R10**: 123

### Memory

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

Registers

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

Memory

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1000</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td></td>
<td>1396</td>
</tr>
<tr>
<td>i</td>
<td>3</td>
<td>1400</td>
</tr>
<tr>
<td>n</td>
<td>123</td>
<td>1404</td>
</tr>
</tbody>
</table>
Generalization: Memory Operands

Full form of memory operands:

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

- \textit{displacement} is an integer or a label (default = 0)
- \textit{base} is a 4-byte or 8-byte register
- \textit{index} is a 4-byte or 8-byte register
- \textit{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
Operand Examples

Memory operands: **scaled indexed addressing**

- \( 5(%rax, %r10, 4) \) ⇒ 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

- \( i(%rax, %r10, 4) \) ⇒ 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
iClicker Question

Q: What is the effect of the following code?

\[
\text{leaq} (\%rax,\%rax,4),\%rax
\]

A. Move the quad at 4*(contents of RAX) to RAX
B. Move the quad at 5*(contents of RAX) to RAX
C. Multiply RAX by 4
D. Multiply RAX by 5
E. Double RAX and add 4
Agenda

Flattened C
Control flow with signed integers
Control flow with unsigned integers
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
Structures: Base+Disp Addressing

C

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

Assem Lang

```
.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

```assembly
.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
## Structures: Padding

### 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 \texttt{cmp\{q,l,w,b\}} set condition code bits in EFLAGS register?

Answer
  • (See following slides)
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
Example: `addq src, dest`

- Compute sum \((\text{dest}+\text{src})\)
- Assign sum to \text{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<\text{src}
- OF: set if signed overflow
  - Set to 1 iff
    \[(\text{src}>0 \&\& \text{dest}>0 \&\& \text{sum}<0) \; \| \; (\text{src}<0 \&\& \text{dest}<0 \&\& \text{sum}\geq0)\]
Condition Code Bits

Example: `subq 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 `dest<src`
- OF: set to 1 iff signed overflow  
  - Set to 1 iff  
    - `(dest>0 && src<0 && sum<0) ||`
    - `(dest<0 && src>0 && sum>=0)`

Example: `cmpq src, dest`
- Same as `subq`
- But does not affect `dest`
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 `j1` 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
   • $\Rightarrow$ OF=0, SF=0 $\Rightarrow$ (OF^SF)==0 $\Rightarrow$ don't jump

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

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

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