Assembly Language: Part 2

Jennifer Rexford

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

Flattened C Code

See Bryant & O’Hallaron book for faster patterns
Agenda

Flattened C code
Control flow with signed integers
Control flow with unsigned integers
Arrays
Structures

if Example

C
int i;
if (i < 0)
    i = -i;
else
    if (i > 0) goto endif1;
    i = -i;
endif1:

Flattened C
int i;
if (i >= 0) goto endif1;
i = -i;
endif1:

Note:

if...else Example

C
int i;
int j;
int smaller;

if (i >= j) goto else1;
smaller = i;
else goto endif1;
else1:
smaller = j;
endif1:

Flattened C
int i;
int j;
int smaller;

if (i > j) goto else1;
smaller = i;
else goto endif1;
else1:
smaller = j;
endif1:

Note:

while Example

C
int fact;
int n;

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

Flattened C
int fact;
int n;

fact = 1;
loop1:
    if (n <= 1) goto endloop1;
    fact *= n;
    n--;
endloop1:

Note:

if...else Example

Assem Lang

section ".bss"
i: .skip 4

section ".text"

cmp $0, i
jge endif1
negl i
endif1:

Note:

while Example

Assem Lang

section ".bss"
i: .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:

if...else Example

Assem Lang

section ".bss"
i: .skip 4

section ".text"

cmp $0, i
jge endif1
negl i
endif1:

Note:

while Example

Assem Lang

section ".bss"

Note:

if...else Example

Assem Lang

section ".bss"
i: .skip 4

section ".text"

cmp $0, i
jge endif1
negl i
endif1:

Note:

while Example

Assem Lang

section ".bss"

Note:
### 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**

```asm
section "bss"
  fact: .skip 4
  n: .skip 4
section "text"
loop1:
  cmpl $1, n
  jle  endloop1
  movl fact, %eax
  imull n
  movl %eax, fact
  decl n
  jmp loop1
endloop1:
```

**Note:**
- `jle` instruction (conditional jump)
- `imull` instruction

### 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**

```asm
section "data"
  power: .long 1
section "bss"
  base: .skip 4
  exp: .skip 4
  i: .skip 4
section "text"
loop1:
  cmpl $0, i
  jge  endloop1
  movl power, %eax
  imull base
  movl %eax, power
  incl i
  jmp loop1
endloop1:
```

**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:
```

### Control Flow with Signed Integers

#### Unconditional jump

- `jmp label` Jump to label

#### Conditional jumps after comparing signed integers

- `ja label` Jump to label if equal
- `jae 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

### Control Flow with Signed Integers

**Comparing signed integers**

- Sets CC bits in the EFLAGS register
- Beware: operands are in counterintuitive order
- Beware: many other instructions set CC bits
- Conditional jump should immediately follow `cmp`
Signed vs. Unsigned Integers

In C
- Integers are signed or unsigned
- Compiler generates assembly 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: imul, idiv
- Unsigned integers: mul, div

Control flow
- Signed integers: cmp + {je, jne, jl, jle, jg, jge}
  - Unsigned integers: "unsigned cmp" + {je, jne, jl, jle, jg, jge} No!!!
  - Unsigned integers: cmp + {je, jne, jb, jbe, ja, 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:
```

Assem Lang
```assembly
section .text
...
 movl $1, fact
 loop1:
 cmpl $1, n
 jbe  endloop1
 movl fact, %eax
 mull n
 movl %eax, fact
 decl n
 jmp loop1
endloop1:
```

While Example

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

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:
```

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:
 mov 1, teax
 cmpl exp, teax
 jae endloop1
 movl power, teax
 mul base
 movl teax, 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
• Same as comparing signed integers

Conditional jumps after comparing unsigned integers

```plaintext
je label  Jump to label if equal
jne label  Jump to label if not equal
jb label  Jump to label if below
jba 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

```assembly
.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

```assembly
.section .text
movl $3, i
movslq i, %rax
salq $2, %rax
addq $a, %rax
movl (%rax), %r10d
movl %r10d, n
```

Memory

RAX    R10
0      1000
1      1004
2      1008
3      1012

100    1396
4      1400
a      1404

Arrays: Indirect Addressing

Assem Lang

```assembly
.section .text
movl $3, i
movslq i, %rax
salq $2, %rax
addq $a, %rax
movl (%rax), %r10d
movl %r10d, n
```

Memory

RAX    R10
0      1000
1      1004
2      1008
3      1012

100    1396
4      1400
a      1404

Arrays: Indirect Addressing

Assem Lang

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

section .text
...
```

Registers

```
RAX 1012
R10 123
```

Memory

```
0 1000
1 1004
2 1008
3 1012
```

Note: Indirect addressing

One step at a time...

Arrays: Base+Disp Addressing

Assem Lang

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

section .text
...
```

Registers

```
RAX 1012
R10 123
```

Memory

```
0 1000
1 1004
2 1008
3 1012
```

Assem Lang

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

Arrays: Base+Disp Addressing

Assem Lang

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

section .text
...
```

Registers

```
RAX 3
R10 123
```

Memory

```
0 1000
1 1004
2 1008
3 1012
```

Assem Lang

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

Arrays: Base+Disp Addressing

Assem Lang

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

section .text
...
```

Registers

```
RAX 1012
R10 123
```

Memory

```
0 1000
1 1004
2 1008
3 1012
```

Assem Lang

```
int a[100];
int i;
int n;
...
i = 3;
...
n = a[i]
```
Arrays: Base+Disp Addressing

Assem Lang

section ".bss"

a: .skip 400
i: .skip 4
n: .skip 4

... 

Assem Lang

section " .text"

... 

movl $3, i
... 

movl i, %eax 

sall $2, %eax

movl a(%eax), %r10d

movl %r10d, n
... 

Note: Base+dispacement addressing

Arrays: Scaled Indexed Addressing

Assem Lang

section " .bss"

a: .skip 400
i: .skip 4
n: .skip 4

... 

Assem Lang

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

\begin{align*}
\text{section ".bss"} & \quad a: .skip 400 \\
& \quad i: .skip 4 \\
& \quad n: .skip 4 \\
\text{section ".text"} & \\
& \quad \text{movl }$3, i \\
& \quad \text{movl }i, \%eax \\
& \quad \text{movl }a(,%eax,4), \%r10d \\
& \quad \text{movl }\%r10d, n \\
\end{align*}

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>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>
</tbody>
</table>

\text{Note:} Scaled indexed addressing

Generalization: Memory Operands

Full form of memory operands:

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

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

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

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 (address denoted by \(i\)) + (contents of RAX) + (contents of R10) \times 4; consider the sum to be an address; load from (or store to) that address

Aside: The \texttt{lea} Instruction
\texttt{lea}: load effective address
• Unique instruction: suppresses memory load/store
Example
• \texttt{movq 5(rax), r10}\n• Compute the sum (5) + (contents of RAX); consider the sum to be an address; load 8 bytes from that address into R10
• \texttt{leaq 5(rax), r10}\n• 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?
\texttt{leaq (r10,rax,4), rax}

Agenda

Flattened C
Control flow with signed integers
Control flow with unsigned integers
Arrays
Structures

Structures: Indirect Addressing

C
\begin{verbatim}
struct S
{ int i;
 int j;
};
struct S myStruct;
myStruct.i = 18;
myStruct.j = 19;
\end{verbatim}

Assem Lang
\begin{verbatim}
.section .bss
myStruct: .skip 8
\section .text
movl $0, %eax
movl $18, myStruct(%eax)
\end{verbatim}

Note:
Indirect addressing

Structures: Base+Disp Addressing

C
\begin{verbatim}
struct S
{ int i;
 int j;
};
struct S myStruct;
myStruct.i = 18;
myStruct.j = 19;
\end{verbatim}

Assem Lang
\begin{verbatim}
.section .bss
myStruct: .skip 8
\section .text
movl $0, %eax
movb $'A', myStruct(%eax)
\end{verbatim}

Note:
Base+displacement addressing

Structures: Padding

C
\begin{verbatim}
struct S
{ char c;
 int i;
};
struct S myStruct;
myStruct.c = 'A';
myStruct.i = 18;
\end{verbatim}

Assem Lang
\begin{verbatim}
.section .bss
myStruct: .skip 8
\section .text
movl $0, %eax
movb %a, myStruct(%eax)
\end{verbatim}

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.
**Condition Code Bits**

**Example:** `subl src, dest`
- Compute sum \((\text{dest} - \text{src})\)
- Assign sum to \(\text{dest}\)
- \(ZF\): set to 1 if sum == 0
- \(SF\): set to 1 if sum < 0
- \(CF\): set to 1 if unsigned overflow
  - Set to 1 if \(\text{dest} < \text{src}\)
- \(OF\): set to 1 if signed overflow
  - Set to 1 if \((\text{dest} > 0 && \text{src} < 0 && \text{sum} < 0) \; \text{||} \; (\text{dest} < 0 && \text{src} > 0 && \text{sum} >= 0)\)

**Example:** `cmpl src, dest`
- Same as `subl`
- But does not affect \(\text{dest}\)

**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: 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>OF ^ SF</td>
</tr>
<tr>
<td>jbe 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: Unsigned**

**Why does \(jb\) jump iff \(CF\)?** Informal explanation:

1. **largeposnum – smallposnum (not below)**
   - Correct result
   - \(\Rightarrow CF=0 \Rightarrow \text{don't jump}\)
2. **smallposnum – largeposnum (below)**
   - Incorrect result
   - \(\Rightarrow CF=1 \Rightarrow \text{jump}\)

---

**Conditional Jumps: Signed**

**Why does \(ji\) jump iff \(OF^SF\)?** Informal explanation:

1. **largeposnum – smallposnum (not less than)**
   - Certainly correct result
   - \(\Rightarrow OF=0, SF=1, OF^SF=1 \Rightarrow \text{don't jump}\)
2. **smallposnum – largeposnum (less than)**
   - Certainly correct result
   - \(\Rightarrow OF=0, SF=1 \Rightarrow (OF^SF)=1 \Rightarrow \text{jump}\)
3. **larenegnum – smallnegnum (less than)**
   - Certainly correct result
   - \(\Rightarrow OF=0, SF=1, OF^SF=1 \Rightarrow \text{jump}\)
4. **smallnegnum – larenegnum (not less than)**
   - Certainly correct result
   - \(\Rightarrow OF=0, SF=0 \Rightarrow (OF^SF)=0 \Rightarrow \text{don't jump}\)
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