Assembly Language: Part 2

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

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

#### Flattened C

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

#### Assem Lang

```assembly
.section "bss"
i: .skip 4
...$
.section "text"
...$
```

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

### if Example

#### Flattened C

```c
int i;
...$
```

#### Assem Lang

```assembly
.section "bss"
i: .skip 4
...$
```

#### Note:
- `jmp` instruction (unconditional jump)

### if...else Example

#### 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;
else1:
  smaller = j;
endif1:
```

#### Assem Lang

```assembly
.section "bss"
i: .skip 4
j: .skip 4
smaller: .skip 4
...$
```

#### 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

#### Flattened C

```c
int i;
int j;
int smaller;
...$
```

#### Assem Lang

```assembly
.section "bss"
i: .skip 4
j: .skip 4
smaller: .skip 4
...$
```

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

### 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:
```

#### Assem Lang

```assembly
.section "bss"
fact: .skip 4
n: .skip 4
...$
```

#### Note:
- `jmp` instruction (unconditional jump)
- `imull` instruction

### 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
...$
```

#### Note:
- `cmp` instruction (counterintuitive operand order)
- Sets CC bits in EFLAGS register
- `jle` instruction (conditional jump)
- `imul` instruction
for Example

```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;
if (i >= exp) goto endloop1;
power *= base;
i++;
endloop1:
```

Assem Lang

```asm
section "data"
    power: .long 1
    base: .skip 4
    exp: .skip 4
    i: .skip 4

section "text"
    movl $0, i
    jmp loop1
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

- `cmp{q,l,w,b} srcIRM, destRM` 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 immediately follow `cmp`

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: imul, idiv
- Unsigned integers: mul, div

Control flow
- Signed integers: cmp + (je, jae, jl, jle, jg, jge)
  Unsigned integers: "unsigned cmp" + (je, jae, jl, jle, jg, jge)? NaN!
- Unsigned integers: cmp + (je, jae, jb, jbe, ja, jae)

while Example

C

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

Flattened C

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

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

Control Flow with Unsigned Integers

Comparing unsigned integers

[je, jae, jl, jle, jg, jge] (Same as comparing signed integers)

Conditional jumps after comparing unsigned integers

[je label Jump to label if equal
[jbe label Jump to label if below or equal
[jae label Jump to label if not equal
[jl label Jump to label if below
[jg label Jump to label if above or equal
[jge label Jump to label if above

• Examine CC bits in EFLAGS register
Agenda

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

Arrays: Indirect Addressing

Assem Lang

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

Arrays: Indirect Addressing

Assem Lang

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

Arrays: Indirect Addressing

Assem Lang

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

Arrays: Indirect Addressing

Assem Lang

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

Arrays: Indirect Addressing

Assem Lang

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

Arrays: Indirect Addressing

Assem Lang

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

Arrays: Indirect Addressing

Assem Lang

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

Arrays: Indirect Addressing

Assem Lang

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

Assem Lang

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

Registers

```
RAX  1012
R10  123
```

Memory

```
0  1000
1  1004
2  1008
3  1012
...
99 1396
100 1400
101 1404
```

Note:
Indirect addressing

---

Arrays: Base+Disp Addressing

Assem Lang

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

C

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

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

Assem Lang

Registers

Memory

Note: Base+displacement addressing

Assem Lang

Registers

Memory

Note: Base+displacement addressing

Arrays: Scaled Indexed Addressing

C

Assem Lang

Registers

Memory

One step at a time...

Assem Lang

Registers

Memory

Note: Scaled indexed addressing

Assem Lang

Registers

Memory

Note: Scaled indexed addressing
**Arrays: Scaled Indexed Addressing**

Assem Lang

<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 3008</td>
</tr>
<tr>
<td></td>
<td>3 1012</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5 1396</td>
</tr>
<tr>
<td></td>
<td>6 1400</td>
</tr>
<tr>
<td></td>
<td>7 1404</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...

**Operand Examples**

**Memory operands: base+displacement addressing**

- \(5(\text{RAX})\) ⇒ compute the sum \((5) + (\text{contents of RAX})\); consider the sum to be an address; load from (or store to) that address
- \(i(\text{RAX})\) ⇒ compute the sum \((\text{address denoted by } i) + (\text{contents of RAX})\); consider the sum to be an address; load from (or store to) that address

**Memory operands: indexed addressing**

- \(5(\text{RAX}, \text{R10})\) ⇒ compute the sum \((5) + (\text{contents of RAX}) + (\text{contents of R10})\); consider the sum to be an address; load from (or store to) that address
- \(i(\text{RAX}, \text{R10})\) ⇒ compute the sum \((\text{address denoted by } i) + (\text{contents of RAX}) + (\text{contents of R10})\); consider the sum to be an address; load from (or store to) that address

**Memory operands: scaled indexed addressing**

- \(5(\text{RAX}, \text{R10}, 4)\) ⇒ 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
- \(i(\text{RAX}, \text{R10}, 4)\) ⇒ 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

What is the effect of this?
- `leaq (%rax, %rax, 4), %rax`

---

**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
 movq $myStruct, %rax
 movl $18, 0(%rax)
 movl $19, 4(%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**
```assembly
 movq $myStruct, %rax
 movl $18, (%rax)
 movl $19, 4(%rax)
```

---

**Structures: Padding**

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

**Assem Lang**
```assembly
 movb $'A', 0(%rax)
 movl $18, 4(%rax)
```

**Beware:** Compiler sometimes inserts padding after fields

---

**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

Appendix

Setting and using CC bits in EFLAGS register

Setting Condition Code Bits

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

Answer
  • (See following slides)

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)

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
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 if 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>~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 if OF^SF
• ... then the others follow

Conditional Jumps: Signed

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

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

(1) largeposnum – smallposnum (not less than)
• Certainly correct result
  • ⇒ OF=0, SF=0, OF^SF=0 ⇒ don’t jump
(2) smallposnum – largeposnum (less than)
• Certainly correct result
  • ⇒ OF=0, SF=1, OF^SF=1 ⇒ jump
(3) largenegnum – smallnegnum (less than)
• Certainly correct result
  • ⇒ OF=0, SF=1 ⇒ (OF^SF)=1 ⇒ jump
(4) smallnegnum – largenegnum (not less than)
• Certainly correct result
  • ⇒ OF=0, SF=0 ⇒ (OF^SF)=0 ⇒ 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)=1 ⇒ 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