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

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

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

C
```
else
    { statementF1;
      ...
      statementFN;
    }
```

Flattened C
```
if (! expr) goto else1;
statement1;
...
statementN;
goto 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:
```

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:
```

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

```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:
```

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:
```

Note:
- `jle` instruction (conditional jump)
- `imul` instruction
### Flattened C

```c
int power = 1;
int base;
int exp;
int i;

for (i = 0; i < exp; i++)
    power *= base;
```

### Assem Lang

```asm
.section .data
power: .long 1
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**

- `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`

### 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

### Signed vs. Unsigned Integers

- In C
  - Integers are signed or unsigned
  - Compiler generates assem lang 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: `cmp {je, jne, jb, jbe, ja, jae}`

Unsigned integers: "unsigned cmp" + {je, jne, jl, jle, jg, jge}?
No!!!
- Unsigned integers: `cmp {je, jne, jb, jbe, ja, jae}`
Agenda

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

RAM

RAM (Random Access Memory)

```
... TEXT
RODATA
DATA
BSS
HEAP
STACK
...
```

Defining Data: DATA Section 1

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

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

Note:
- `.section` instruction (to announce BSS section)
- `.skip` instruction

Defining Data: RODATA Section

```
... "hello\n"
helloLabel:
```

Note:
- `.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
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, %rax
salq $2, %rax
addq $a, %rax
movl (%rax), %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, %rax
salq $2, %rax
addq $a, %rax
movl (%rax), %r10d
movl %r10d, n
```

Arrays: Indirect Addressing
Arrays: Indirect Addressing

Assem Lang

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

.section "text"
movl $3, i
movl $4, n

movl (rax), %r10d
movl %r10d, n
```

Registers

- RAX: 1012
- R10: 123

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>
</tbody>
</table>

Note: Indirect addressing

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 $4, n

movl a(%eax), %r10d
movl %r10d, n
```

Registers

- RAX
- R10: 123

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>
</tbody>
</table>

One step at a time...

Arrays: Base+Disp Addressing

Assem Lang

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

.section "text"
movl $3, i
movl $4, n

movl a(%eax), %r10d
movl %r10d, n
```

Registers

- RAX
- R10

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>
</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 $4, n

movl a(%eax), %r10d
movl %r10d, n
```

Registers

- RAX
- R10

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>
</tbody>
</table>

Arrays: Base+Disp 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
sal $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

```
0 1000
1 1004
2 1008
3 123 1012
```

Note:

Base+displacement 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
```

C

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

One step at a time...

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

Memory

```
0 1000
1 1004
2 1008
3 123 1012
```

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

Memory

```
0 1000
1 1004
2 1008
3 123 1012
```

Note:

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

Generalization: Memory Operands

Full form of memory operands:

displacement(base,index,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 (displacement) + (contents of base) + ((contents of index) * (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

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) ⇒ compute the sum (5) + (contents of RAX) + ((contents of R10) * 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) * 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

<table>
<thead>
<tr>
<th>C</th>
<th>Assem Lang</th>
</tr>
</thead>
</table>
| struct S { int i; int j; };  
  - struct S myStruct;  
  - myStruct.i = 18;  
  - myStruct.j = 19; | .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

<table>
<thead>
<tr>
<th>C</th>
<th>Assem Lang</th>
</tr>
</thead>
</table>
| struct S { int i; int j; };  
  - struct S myStruct;  
  - myStruct.i = 18;  
  - myStruct.j = 19; | .section ".bss"  
myStruct: .skip 8  
- .section ".text"  
- movq $myStruct, %rax  
- movl $18, 0(%rax)  
- movl $19, 4(%rax) |

Structures: Padding

<table>
<thead>
<tr>
<th>C</th>
<th>Assem Lang</th>
</tr>
</thead>
</table>
| struct S { char c; int i; };  
  - struct S myStruct;  
  - myStruct.c = 'A';  
  - myStruct.i = 18; | .section ".bss"  
myStruct: .skip 8  
- .section ".text"  
- movq $myStruct, %rax  
- movb $'A', 0(%rax)  
- movq $myStruct, %rax  
- movl $18, 0(%rax)  
- movl $19, 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 \mid ZF</td>
</tr>
<tr>
<td>ja label</td>
<td>¬(CF \mid ZF)</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 \( \rightarrow \) smallnum (not below)
• Correct result
  \( \Rightarrow CF=0 \Rightarrow \) don’t jump

(2) smallnum \( \rightarrow \) largenum (below)
• Incorrect result
  \( \Rightarrow CF=1 \Rightarrow \) 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 \mid SF</td>
</tr>
<tr>
<td>jge label</td>
<td>¬(OF \mid SF)</td>
</tr>
<tr>
<td>jle label</td>
<td>(OF \mid SF) \mid ZF</td>
</tr>
<tr>
<td>jg label</td>
<td>¬((OF \mid SF) \mid ZF)</td>
</tr>
</tbody>
</table>

Note:
• If you can understand why \( jl \) jumps iff \( OF^\mid SF \)
• … then the others follow

Conditional Jumps: Signed

Why does \( jl \) jump iff \( OF^\mid SF \)? Informal explanation:

(1) largeposnum \( \rightarrow \) smallposnum (not less than)
• Certainly correct result
  \( \Rightarrow OF=0, SF=0 \Rightarrow (OF^\mid SF)==0 \Rightarrow \) don’t jump

(2) smallposnum \( \rightarrow \) largeposnum (less than)
• Certainly correct result
  \( \Rightarrow OF=0, SF=1, OF^\mid SF==1 \Rightarrow \) jump

(3) largenegnum \( \rightarrow \) smallnegnum (less than)
• Certainly correct result
  \( \Rightarrow OF=0, SF=1 \Rightarrow (OF^\mid SF)==1 \Rightarrow \) jump

(4) smallnegnum \( \rightarrow \) largenegnum (not less than)
• Certainly correct result
  \( \Rightarrow OF=0, SF=0 \Rightarrow (OF^\mid SF)==0 \Rightarrow \) don’t jump

(5) posnum \( \rightarrow \) negnum (not less than)
• Suppose correct result
  \( \Rightarrow OF=0, SF=0 \Rightarrow (OF^\mid SF)==0 \Rightarrow \) don’t jump

(6) posnum \( \rightarrow \) negnum (not less than)
• Suppose incorrect result
  \( \Rightarrow OF=1, SF=1 \Rightarrow (OF^\mid SF)==1 \Rightarrow \) don’t jump

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

(8) negnum \( \rightarrow \) posnum (less than)
• Suppose incorrect result
  \( \Rightarrow OF=1, SF=0 \Rightarrow (OF^\mid SF)==1 \Rightarrow \) 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”

```
#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: 00
- Byte 1: 33
- Byte 2: 77
- Byte 3: ff

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

```
.section ".data"
grade:

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

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?

```
.section ".data"
grade:

.section ".text"
# Option 1
movl grade, %eax
subl $1, %eax
movl %eax, grade
# Option 2
subl $1, grade
```

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

What would happen?