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

else
{
    statementF1;
    ...
    statementFN;
}

Flattened C

if (! expr) goto else1;
statement1;
...
statementN;
endif1:

else1:

Flattened C Code

C
while (expr)
{
    statement1;
    ...
    statementN;
}
loop1:
if (! expr) goto endloop1;
statement1;
...
statementN;
goto loop1;
endloop1:

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
- Arrays
- Structures

if Example

C

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

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 Example

C

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

Assem Lang

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

Note:
- `cmp` instruction (counterintuitive operand order)
- `jge` instruction (conditional jump)
- `negl` instruction (unconditional jump)

if...else Example

C

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

Flattened C

```c
int i, j;
int smaller;
if (i >= j) goto else1;
smaller = i;
```

Note:
- `jmp` instruction (unconditional jump)

while Example

C

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

Flattened C

```c
int fact, 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:
```

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

Assem Lang

```asm
.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:
```

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"
basex: .skip 4
exp: .skip 4
-
.section ".text"
-    movl $0, i
    loop1: cmpl $0, i
        jge endloop1
        movl exp, %eax
        imull base
        movl %eax, power
        incl i
        jmp loop1
endloop1:
```

for Example

Control Flow with Signed Integers

### Comparing signed integers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cmp</strong></td>
<td>Compare with src</td>
</tr>
</tbody>
</table>

- 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

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, 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
```
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:
```

Assem Lang
```
.section ".bss"
fact: .skip 4
n: .skip 4
-
.section ".text"
loop1:
  cmp $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:
```

Assem Lang
```
.section ".data"
power: .long 1
.
.section ".bss"
base: .skip 4
exp: .skip 4
i: .skip 4
-
.section ".text"
loop1:
  cmpl $0, i
  jae endloop1
  movl %eax
  mull base
  movl %eax, base
  incl i
  jmp loop1
endloop1:
```

Note:
- `jae` instruction (instead of `jge`)
- `mull` instruction (instead of `imull`
Control Flow with Unsigned Integers

Comparing unsigned integers

```
cmp(q,l,w,b) srcIRM, destRM  // Compare 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

```
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
...
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
...
```

Registers

```
RAX  R10
```

Memory

```
a  
0  1000
1  1004
2  1008
3  123  1012
...
99  1396
i  3  1400
n  1404
```

Arrays: Indirect Addressing

Assem Lang

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

Registers

```
RAX  R10
```

Memory

```
a  
0  1000
1  1004
2  1008
3  123  1012
...
99  1396
i  3  1400
n  1404
```
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
- RAX 1012
- R10 123

Memory
- a
- 0 1000
- 1 1004
- 2 1008
- 3 123 1012
- 99 1396
- i 3 1400
- n 123 1404

Note:
- Indirect addressing

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
- RAX 1012
- R10 123

Memory
- a
- 0 1000
- 1 1004
- 2 1008
- 3 123 1012
- 99 1396
- i 3 1400
- n 123 1404

Arrays: Base+Disp Addressing

Assem Lang
```
section `.bss'
a: .skip 400
i: .skip 4
n: .skip 4
  - section `.text'
  - movl $3, i
  - movslq i, %eax
  - salq $2, %eax
  - movl a(%eax), %r10d
  - movl %r10d, n
```

Registers
- RAX 1012
- R10 123

Memory
- a
- 0 1000
- 1 1004
- 2 1008
- 3 123 1012
- 99 1396
- i 3 1400
- n 123 1404

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

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
  - movslq i, %eax
  - salq $2, %eax
  - movl a(%eax), %r10d
  - movl %r10d, n
```

Registers
- RAX 1012
- R10 123

Memory
- a
- 0 1000
- 1 1004
- 2 1008
- 3 123 1012
- 99 1396
- i 3 1400
- n 123 1404
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

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

Memory

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>1396</td>
</tr>
<tr>
<td>i</td>
<td>3</td>
</tr>
<tr>
<td>n</td>
<td>1404</td>
</tr>
</tbody>
</table>

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
-```

Registers

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>3</td>
</tr>
<tr>
<td>R10</td>
<td>12</td>
</tr>
</tbody>
</table>

Memory

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>1396</td>
</tr>
<tr>
<td>i</td>
<td>3</td>
</tr>
<tr>
<td>n</td>
<td>1404</td>
</tr>
</tbody>
</table>

C

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

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>3</td>
</tr>
<tr>
<td>R10</td>
<td>12</td>
</tr>
</tbody>
</table>

Memory

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>1396</td>
</tr>
<tr>
<td>i</td>
<td>3</td>
</tr>
<tr>
<td>n</td>
<td>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
```

Registers

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

<table>
<thead>
<tr>
<th>R10</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>99 1396</td>
</tr>
<tr>
<td></td>
<td>i 3 1400</td>
</tr>
<tr>
<td></td>
<td>n 1404</td>
</tr>
</tbody>
</table>

Note:
- Scaled indexed addressing

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

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

Register operands
- \texttt{%rax} \rightarrow read from (or write to) register RAX

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

Memory operands: indirect addressing
- \texttt{%rax} \rightarrow consider the contents of RAX to be an address; load from (or store to) that address

Memory operands: base+displacement addressing
- 5(%rax) \rightarrow compute the sum (5) + (contents of RAX); consider the sum to be an address; load from (or store to) that address
- i(%rax) \rightarrow 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) \rightarrow 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) \rightarrow 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
- \(5(\text{rax}, \%r10, 4)\) ⇒ 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
- \(i(\text{rax}, \%r10, 4)\) ⇒ 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
- \(\text{movq } 5(\text{rax}), \%r10\)
- Compute the sum (5) + (contents of RAX); consider the sum to be an address; load 8 bytes from that address into R10
- \(\text{leaq } 5(\text{rax}), \%r10\)
- 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?

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

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
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, (%rax)

movq $myStruct, %rax
.addq $4, %rax
.movl $19, (%rax)

Note:
Indirect addressing

Structures: Base+Disp Addressing

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

Assem Lang
.section "text"
.movq $myStruct, %rax
.movl $18, 0(%rax)

.movl $19, 4(%rax)

Notes:
C

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

Beware:

Compiler sometimes inserts padding after fields

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 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

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: \texttt{addq 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 sum<\texttt{src}
- OF: set if signed overflow
  - Set to 1 iff \( (\texttt{src}>0 \&\& \texttt{dest}>0 \&\& \texttt{sum}<0) \) \( \| \)
  - \( (\texttt{src}<0 \&\& \texttt{dest}<0 \&\& \texttt{sum}>0) \)

**Example: 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}<\texttt{src}
- OF: set to 1 iff signed overflow
  - Set to 1 iff \( (\texttt{dest}>0 \&\& \texttt{src}<0 \&\& \texttt{sum}<0) \) \( \| \)
  - \( (\texttt{dest}<0 \&\& \texttt{src}>0 \&\& \texttt{sum}>0) \)

**Example: 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 \texttt{unsigned} data

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

**Note:**
- If you can understand why \texttt{jb} jumps iff CF
- \( \ldots \) then the others follow

**Conditional Jumps: Signed**

After comparing \texttt{signed} data

<table>
<thead>
<tr>
<th>Jump Instruction</th>
<th>Use of CC Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{je label}</td>
<td>ZF</td>
</tr>
<tr>
<td>\texttt{jne label}</td>
<td>\sim{ZF}</td>
</tr>
<tr>
<td>\texttt{jl label}</td>
<td>OF ^ SF</td>
</tr>
<tr>
<td>\texttt{jg label}</td>
<td>\sim{(OF ^ SF)}</td>
</tr>
<tr>
<td>\texttt{je label}</td>
<td>(OF ^ SF)</td>
</tr>
<tr>
<td>\texttt{jg label}</td>
<td>\sim{((OF ^ SF)</td>
</tr>
</tbody>
</table>

**Note:**
- If you can understand why \texttt{jl} jumps iff \texttt{OF^SF}
- \( \ldots \) then the others follow

---

**Conditional Jumps: Unsigned**

Why does \texttt{jb} jump iff CF? Informal explanation:

1. \texttt{largenum} – \texttt{smallnum} (not below)
   - Correct result
   - \( \Rightarrow \) \texttt{CF}=0 \( \Rightarrow \) don’t jump
2. \texttt{smallnum} – \texttt{largenum} (below)
   - Incorrect result
   - \( \Rightarrow \) \texttt{CF}=1 \( \Rightarrow \) jump
Conditional Jumps: Signed

Why does \( \text{jl} \) jump iff \( \text{OF}^\text{SF} \)? Informal explanation:

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

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

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

(4) smallnegnum – largenegnum (not less than)
   - Certainly correct result
   - \( \Rightarrow \text{OF}=0, \text{SF}=0 \Rightarrow (\text{OF}^\text{SF})=0 \Rightarrow \text{don't jump} \)

(5) posnum – negnum (not less than)
   - Suppose correct result
   - \( \Rightarrow \text{OF}=0, \text{SF}=0 \Rightarrow (\text{OF}^\text{SF})=0 \Rightarrow \text{don't jump} \)

(6) posnum – negnum (not less than)
   - Suppose incorrect result
   - \( \Rightarrow \text{OF}=1, \text{SF}=1 \Rightarrow (\text{OF}^\text{SF})=1 \Rightarrow \text{don't jump} \)

(7) negnum – posnum (less than)
   - Suppose correct result
   - \( \Rightarrow \text{OF}=0, \text{SF}=1 \Rightarrow (\text{OF}^\text{SF})=1 \Rightarrow \text{jump} \)

(8) negnum – posnum (less than)
   - Suppose incorrect result
   - \( \Rightarrow \text{OF}=1, \text{SF}=0 \Rightarrow (\text{OF}^\text{SF})=1 \Rightarrow \text{jump} \)