Assembly Language

Part 2
Goals of this Lecture

Help you learn:

• Intermediate aspects of AARCH64 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 doesn’t proceed in consecutive lines

Solution

• Flatten the C code to eliminate all nesting
Flattened C Code

C

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

if (expr)
{
    statementT1;
    ...
    statementTN;
}
else
{
    statementF1;
    ...
    statementFN;
}

Flattened C

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

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

### C

```c
while (expr)
{
    statement1;
    ...
    statementN;
}
```

### Flattened C

```
loop1:
    if (! expr) goto endloop1;
    statement1;
    ...
    statementN;
    goto loop1;
endloop1:
```

```c
for (expr1; expr2; expr3)
{
    statement1;
    ...
    statementN;
}
```

```
loop1:
    expr1;
    loop1:
        if (! expr2) goto endloop1;
        statement1;
        ...
        statementN;
        expr3;
        goto loop1;
endloop1:
```
## 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:
```
if Example

Flattened C

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

Assembly

```assembly
.section ".bss"
i: .skip 4
...
.section ".text"
...
    adr x0, i
    ldr w1, [x0]
    cmp w1, 0
    bge endif1
    neg w1, w1
endif1:
```

Notes:

- `cmp` instruction: compares operands, sets condition flags
- `bge` instruction (conditional branch if greater than or equal): Examines condition flags in PSTATE 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;
else 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:
```

**Assembly**

```assembly
... 
    adr x0, i 
    ldr w1, [x0] 
    adr x0, j 
    ldr w2, [x0] 
    cmp w1, w2 
    bge else1 
    adr x0, smaller 
    str w1, [x0] 
    b endif1 
else1:
    adr x0, smaller 
    str w2, [x0] 
endif1:
```

**Note:**

b instruction (unconditional branch)
while Example

C

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

Flattened C

```c
int n;
int fact;
...
fact = 1;
loop1:
    if (n <= 1) goto endloop1;
    fact *= n;
    n--;
    goto loop1;
endloop1:
```
### Flattened C

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

### Assembly

```asm
...  
    adr x0, n
    ldr w1, [x0]
    mov w2, 1
loop1:
    cmp w1, 1
    ble endloop1
    mul w2, w2, w1
    sub w1, w1, 1
    b loop1
endloop1:
# str w2 into fact
```

**Note:**

*ble* instruction (conditional branch if less than or equal)
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:
```
What goes where?

Q: Which section(s) would power, base, exp, i go into?

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

A. All on stack
B. power in .data and rest in .rodata
C. All in .data
D. power in .bss and rest in .data
E. power in .data and rest in .bss

E

none are string literals: not RODATA

all are file scope, process duration: not STACK

power is initialized: DATA

the rest are not: BSS
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:
```

Assembly

```assembly
.section ".data"
power: .word 1
...
.section ".bss"
base: .skip 4
exp: .skip 4
i: .skip 4
...
```
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:
```

Assembly

```assembly
adr x0, power
ldr w1, [x0]
adr x0, base
ldr w2, [x0]
adr x0, exp
ldr w3, [x0]
mov w4, 0
loop1:
    cmp w4, w3
    bge endloop1
mul w1, w1, w2
add w4, w4, 1
b loop1
endloop1:
    # str w1 into power
```
Control Flow with Signed Integers

Unconditional branch

\[ b \text{ label} \text{ Branch to label} \]

Compare

\[ \text{cmp } X_m, X_n \text{ Compare } X_m \text{ to } X_n \]
\[ \text{cmp } W_m, W_n \text{ Compare } W_m \text{ to } W_n \]

- Set condition flags in PSTATE register

Conditional branches after comparing signed integers

\[ \text{beq } \text{label} \text{ Branch to label if equal} \]
\[ \text{bne } \text{label} \text{ Branch to label if not equal} \]
\[ \text{blt } \text{label} \text{ Branch to label if less than} \]
\[ \text{ble } \text{label} \text{ Branch to label if less or equal} \]
\[ \text{bgt } \text{label} \text{ Branch to label if greater than} \]
\[ \text{bge } \text{label} \text{ Branch to label if greater or equal} \]

- Examine condition flags in PSTATE register
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
• Division (sdiv vs. udiv)
• Control flow
  • Which is the larger 32-bit integer value?
    11111111111111111111111111111111
    00000000000000000000000000000000
    (Yes, there are 32 bits there. You don’t have to count)
Control Flow with Unsigned Integers

Unconditional branch

\[ \texttt{b label} \quad \texttt{b label} \quad \text{Branch to label} \]

Compare

\begin{align*}
\texttt{cmp Xm, Xn} & \quad \texttt{cmp Xm, Xn} \quad \text{Compare Xm to Xn} \\
\texttt{cmp Wm, Wn} & \quad \texttt{cmp Wm, Wn} \quad \text{Compare Wm to Wn}
\end{align*}

• Set condition flags in PSTATE register

Conditional branches after comparing \textbf{unsigned} integers

\begin{align*}
\texttt{beq label} & \quad \texttt{beq label} \quad \text{Branch to label if equal} \\
\texttt{bne label} & \quad \texttt{bne label} \quad \text{Branch to label if not equal} \\
\texttt{blt label} & \quad \texttt{blo label} \quad \text{Branch to label if lower} \\
\texttt{ble label} & \quad \texttt{bhs label} \quad \text{Branch to label if higher or same} \\
\texttt{bgt label} & \quad \texttt{bhi label} \quad \text{Branch to label if higher} \\
\texttt{bge label} & \quad \texttt{bhs label} \quad \text{Branch to label if higher or same}
\end{align*}

• Examine condition flags in PSTATE register
while Example

Flattened C

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

Assembly: Signed → Unsigned

```assembly
... 
    adr x0, n
    ldr w1, [x0]
    mov w2, 1
loop1:
    cmp w1, 1
    ble endloop1
    mul w2, w2, w1
    sub w1, w1, 1
    b loop1
endloop1: 
    # str w2 into fact
... 
    adr x0, n
    ldr w1, [x0]
    mov w2, 1
loop1:
    cmp w1, 1
    bls endloop1
    mul w2, w2, w1
    sub w1, w1, 1
    b loop1
endloop1:
    # str w2 into fact
```

Note:
bls instruction (instead of ble)
Alternative Control Flow: CBZ, CBNZ

Special-case, all-in-one compare-and-branch instructions
• DO NOT examine condition flags in PSTATE register

\[
\text{cbz Xn, label} \quad \text{Branch to label if Xn is zero} \\
\text{cbz Wn, label} \quad \text{Branch to label if Wn is zero} \\
\text{cbnz Xn, label} \quad \text{Branch to label if Xn is nonzero} \\
\text{cbnz Wn, label} \quad \text{Branch to label if Wn is nonzero}
\]
Agenda

- Flattened C
- Control flow with signed integers
- Control flow with unsigned integers
- Arrays
- Structures
Arrays: Brute Force

C

```c
int a[100];
long i;
int n;
...
i = 2;
...
n = a[i]
...```

Assembly

```assembly
.section "bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
.section "text"
...
mov x1, 2
...
adr x0, a
lsl x1, x1, 2
add x0, x0, x1
ldr w2, [x0]
adr x0, n
str w2, [x0]
...
```

To do array lookup, need to compute address of `a[i] = *(a+i)`

Let’s take it one step at a time...
Arrays: Brute Force

Assembly

```
.section ".bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
.section ".text"
...
mov x1, 2
...
adr x0, a
ls1 x1, x1, 2
add x0, x0, x1
ldr w2, [x0]
adr x0, n
str w2, [x0]
...
```

Registers

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>w2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Memory

```
a
2  42  1008
99  1396
i  1400
n  1404
```
Arrays: Brute Force

Assembly

[section "bss"
  a: .skip 400
  i: .skip 8
  n: .skip 4
...
  .section "text"
  ...
  mov x1, 2
...|

  addr x0, a
  lsl x1, x1, 2
  add x0, x0, x1
  ldr w2, [x0]
  addr x0, n
  str w2, [x0]
...|

Registers

<table>
<thead>
<tr>
<th>x0</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>2</td>
</tr>
<tr>
<td>w2</td>
<td></td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1004</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1008</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>1396</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>1400</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>1404</td>
<td></td>
</tr>
</tbody>
</table>
Arrays: Brute Force

Assembly

```
.sect .bss
a: .skip 400
i: .skip 8
n: .skip 4
...
.sect .text
...
mov x1, 2
...
adr x0, a
lsl x1, x1, 2
add x0, x0, x1
ldr w2, [x0]
adr x0, n
str w2, [x0]
...
```

Registers

<table>
<thead>
<tr>
<th>x0</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>8</td>
</tr>
<tr>
<td>w2</td>
<td></td>
</tr>
</tbody>
</table>

Memory

```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1000</th>
<th>1004</th>
<th>1008</th>
<th>1396</th>
<th>1400</th>
<th>1404</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>42</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

---

29
Arrays: Brute Force

Assembly

```
[section ".bss"]
a: .skip 400
i: .skip 8
n: .skip 4
...
[section ".text"]
...
mov x1, 2
...
adr x0, a
lsl x1, x1, 2
add x0, x0, x1
ldr w2, [x0]
adr x0, n
str w2, [x0]
...
```

Registers

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>w2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1008</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Memory

```
 a 0 1 2 ...
1000 1004 1008 ...
99 1396
i 1400
n 1404
```
## Arrays: Brute Force

### Assembly

```
.section ".bss"
  a: .skip 400
  i: .skip 8
  n: .skip 4
...
.section ".text"
  ...
    mov x1, 2
...
    adr x0, a
    lsl x1, x1, 2
    add x0, x0, x1
    ldr w2, [x0]
    adr x0, n
    str w2, [x0]
...
```

### Registers

<table>
<thead>
<tr>
<th>x0</th>
<th>1008</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>8</td>
</tr>
<tr>
<td>w2</td>
<td>42</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1000</th>
<th>1004</th>
<th>1008</th>
<th>1396</th>
<th>1400</th>
<th>1404</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>42</td>
<td>0</td>
<td>1008</td>
<td>1008</td>
<td>1008</td>
<td>1396</td>
<td>1396</td>
<td>1396</td>
</tr>
<tr>
<td>i</td>
<td></td>
<td>99</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Arrays: Brute Force

**Assembly**

```assembly
.section " .bss"
a: .skip 400
i: .skip 8
n: .skip 4
...

.section " .text"
...
mov x1, 2
...
adr x0, a
lsr x1, x1, 2
add x0, x0, x1
ldr w2, [x0]
adr x0, n
str w2, [x0]
...```

**Registers**

- `x0`: 1404
- `x1`: 8
- `w2`: 42

**Memory**

- `a`: 1000
- `i`: 1004
- `n`: 1008
- `99`: 1396
- `i`: 1400
- `n`: 1404
Arrays: Brute Force

Assembly

```
.section ".bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
.section ".text"
...
  mov x1, 2
...
  adr x0, a
  lsl x1, x1, 2
  add x0, x0, x1
  ldr w2, [x0]
  adr x0, n
  str w2, [x0]
...
```

Registers

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>w2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1404</td>
<td>8</td>
<td>42</td>
</tr>
</tbody>
</table>

Memory

```
0 | 1000
1 | 1004
2 | 42  1008
99 | 1396
i | 1400
n | 42  1404
```
This uses a different addressing mode for the load
Memory Addressing Modes

Address loaded:

- `ldr Wt, [Xn, offset]`  \[ Xn+offset \] \((-2^8 \leq \text{offset} < 2^{14})\)
- `ldr Wt, [Xn]`  \[ Xn \]  (shortcut for offset=0)
- `ldr Wt, [Xn, Xm]`  \[ Xn+Xm \]
- `ldr Wt, [Xn, Xm, LSL n]`  \[ Xn+(Xm<<n) \]  \((n = 3 \text{ for } 64\text{-bit}, 2 \text{ for } 32\text{-bit})\)

All these addressing modes also available for 64-bit loads:

- `ldr Xt, [Xn, offset]`  \[ Xn+offset \]

etc.
Agenda

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

Structures
Structures: Brute Force

C

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

Assembly

```assembly
.section "bss"
myStruct: .skip 8
...
.section "text"
...
adr x0, myStruct
...
mov w1, 18
str w1, [x0]
...
mov w1, 19
str ???
```
Which mode is à la mode?

Q: Which addressing mode is most appropriate for the last store?

A. `str Wt, [Xn, offset]`
B. `str Wt, [Xn]`
C. `str Wt, [Xn, Xm, LSL n]`
D. `str Wt, [Xn, Xm]`

A is the simplest option: the only one that requires no additional setup.
Structures: Offset Addressing

C

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

Brute-Force

```assembly
.section ".bss"
myStruct: .skip 8
...
.section ".text"
...
adr x0, myStruct
...
mov w1, 18
str w1, [x0]
...
mov w1, 19
add x0, x0, 4
str w1, [x0]
```

Offset

```assembly
.section ".bss"
myStruct: .skip 8
...
.section ".text"
...
adr x0, myStruct
...
mov w1, 18
str w1, [x0]
...
mov w1, 19
str w1, [x0, 4]
```
C

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

Assembly

```assembly
.section " .bss"
myStruct: .skip 8
...
.section " .text"
...
adr x0, myStruct
...
mov w1, 'A'
strb w1, [x0]
...
mov w1, 18
str w1, [x0, 4]
```

Beware:

As we've seen, Compiler sometimes inserts padding after fields
## Structures: Padding

### AARCH64 rules

<table>
<thead>
<tr>
<th>Data type</th>
<th>Within a struct, field 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 AARCH64 assembly language...

Flattened C code

Control transfer with signed integers

Control transfer with unsigned integers

Arrays
  • Addressing modes

Structures
  • Padding
Setting and using condition flags in PSTATE register
Setting Condition Flags

Question
• How does cmp (or arithmetic instructions with “s” suffix) set condition flags?
Condition flags

- **N**: negative flag: set to 1 iff result is **negative**
- **Z**: zero flag: set to 1 iff result is **zero**
- **C**: carry flag: set to 1 iff carry/borrow from msb (unsigned overflow)
- **V**: overflow flag: set to 1 iff **signed overflow** occurred
Condition Flags

Example: `adds dest, src1, src2`

- Compute sum `(src1+src2)`
- Assign sum to `dest`
- `N`: set to 1 iff sum < 0
- `Z`: set to 1 iff sum == 0
- `C`: set to 1 iff unsigned overflow: sum < `src1` or `src2`
- `V`: set to 1 iff signed overflow:
  
  \[
  (src1 > 0 \&\& src2 > 0 \&\& sum < 0) \text{ or } (src1 < 0 \&\& src2 < 0 \&\& sum >= 0)
  \]
Condition Flags

Example: `cmp src1, src2`
- Recall that this is a shorthand for `subs xzr, src1, src2`
- Compute sum `(src1+(-src2))`
- Throw away result
- N: set to 1 iff `sum < 0`
- Z: set to 1 iff `sum == 0` (i.e., `src1 == src2`)
- C: set to 1 iff unsigned overflow (i.e., `src1 >= src2`)
- V: set to 1 iff signed overflow:
  
  
  (src1 > 0 && src2 < 0 && sum < 0) ||
  
  (src1 < 0 && src2 > 0 && sum >= 0)
Unsigned comparison

Why is carry bit set if src1 >= src2? Informal explanation:

1. largenum - smallnum
   - largenum + (two’s complement of smallnum) does cause carry
   - \( C = 1 \)

2. smallnum - largenum (below)
   - smallnum + (two’s complement of largenum) does not cause carry
   - \( C = 0 \)
Question
• How do conditional branch instructions use the condition flags?

Answer
• (See following slides)
Conditional Branches: Unsigned

After comparing unsigned data

<table>
<thead>
<tr>
<th>Branch instruction</th>
<th>Use of condition flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq label</td>
<td>Z</td>
</tr>
<tr>
<td>bne label</td>
<td>~Z</td>
</tr>
<tr>
<td>blo label</td>
<td>~C</td>
</tr>
<tr>
<td>bhs label</td>
<td>C</td>
</tr>
<tr>
<td>bhs label</td>
<td>(~C)</td>
</tr>
<tr>
<td>bhi label</td>
<td>C &amp; (~Z)</td>
</tr>
</tbody>
</table>

Note:
- If you can understand why blo branches iff ~C
- ... then the others follow
Conditional Branches: Unsigned

Why does blo branch iff C? Informal explanation:

(1) largenum – smallnum (not below)

• largenum + (two’s complement of smallnum) does cause carry
• ⇒ C=1 ⇒ don’t branch

(2) smallnum – largenum (below)

• smallnum + (two’s complement of largenum) does not cause carry
• ⇒ C=0 ⇒ branch
After comparing signed data

<table>
<thead>
<tr>
<th>Branch instruction</th>
<th>Use of condition flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq label</td>
<td>Z</td>
</tr>
<tr>
<td>bne label</td>
<td>~Z</td>
</tr>
<tr>
<td>blt label</td>
<td>V ^ N</td>
</tr>
<tr>
<td>bge label</td>
<td>~(V ^ N)</td>
</tr>
<tr>
<td>ble label</td>
<td>(V ^ N)</td>
</tr>
<tr>
<td>bgt label</td>
<td>~(V ^ N)</td>
</tr>
</tbody>
</table>

Note:
- If you can understand why blt branches iff V^N
- ... then the others follow
Conditional Branches: Signed

Why does blt branch iff V^N?
Informal explanation:

(1) largeposnum – smallposnum (not less than)
- Certainly correct result
- \( V=0, N=0, V^N==0 \) ⇒ don’t branch

(2) smallposnum – largeposnum (less than)
- Certainly correct result
- \( V=0, N=1, V^N==1 \) ⇒ branch

(3) largenegnum – smallnegnum (less than)
- Certainly correct result
- \( V=0, N=1 \) ⇒ \((V^N)==1\) ⇒ branch

(4) smallnegnum – largenegnum (not less than)
- Certainly correct result
- \( V=0, N=0 \) ⇒ \((V^N)==0\) ⇒ don't branch
Conditional Branches: Signed

(5) posnum – negnum (not less than)
• Suppose correct result
  • $\Rightarrow V=0, N=0 \Rightarrow (V^N)==0 \Rightarrow$ don't branch

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

(7) negnum – posnum (less than)
• Suppose correct result
  • $\Rightarrow V=0, N=1 \Rightarrow (V^N)==1 \Rightarrow$ branch

(8) negnum – posnum (less than)
• Suppose incorrect result
  • $\Rightarrow V=1, N=0 \Rightarrow (V^N)==1 \Rightarrow$ branch