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
Flattened C code

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

C

\[
\text{if (expr)} \\
\{ \text{statement1;} \\
\quad \ldots \\
\quad \text{statementN;} \\
\} \\
\text{endif1:}
\]

\[
\text{if (expr)} \\
\{ \text{statementT1;} \\
\quad \ldots \\
\quad \text{statementTN;} \\
\} \\
\text{else} \\
\{ \text{statementF1;} \\
\quad \ldots \\
\quad \text{statementFN;} \\
\}
\]

Flattened C

\[
\text{if (! expr) goto endif1;} \\
\quad \text{statement1;} \\
\quad \ldots \\
\quad \text{statementN;} \\
\text{endif1:}
\]

\[
\text{if (! expr) goto else1;} \\
\quad \text{statementT1;} \\
\quad \ldots \\
\quad \text{statementTN;} \\
\quad \text{goto endif1;} \\
\text{else1:}
\]

\[
\text{statementF1;} \\
\quad \ldots \\
\quad \text{statementFN;} \\
\text{endif1:}
\]
C

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

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

Flattened C

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

loop1:
    expr1;
    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;
    goto 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:
```
while Example

Flattened C

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

Assembly

```assembly
...  
adr x0, n
1dr w1, [x0]
mov w2, 1
loop1:
    cmp w1, 1
    ble endloop1
mul w2, w2, w1
sub w1, w1, 1
b loop1
endloop1:
```

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:
```
iClicker Question

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

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

A. All in .data
B. All in .bss
C. `power` in .data and rest in .rodata
D. `power` in .bss and rest in .data
E. `power` in .data and rest in .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:
```
Control Flow with Signed Integers

Unconditional branch

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

Compare

\begin{align*}
\text{cmp } X_m, X_n & \quad \text{Compare } X_m \text{ to } X_n \\
\text{cmp } W_m, W_n & \quad \text{Compare } W_m \text{ to } W_n
\end{align*}

- Set condition flags in PSTATE register

Conditional branches after comparing signed integers

\begin{align*}
\text{beq } \text{label} & \quad \text{Branch to label if equal} \\
\text{bne } \text{label} & \quad \text{Branch to label if not equal} \\
\text{blt } \text{label} & \quad \text{Branch to label if less than} \\
\text{ble } \text{label} & \quad \text{Branch to label if less or equal} \\
\text{bgt } \text{label} & \quad \text{Branch to label if greater than} \\
\text{bge } \text{label} & \quad \text{Branch to label if greater or equal}
\end{align*}

- Examine condition flags in PSTATE 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
- Division (\texttt{sdiv} vs. \texttt{udiv})
- Control flow
Control Flow with Unsigned Integers

Unconditional branch

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

Compare

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

- Set condition flags in PSTATE register

Conditional branches after comparing unsigned integers

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

- Examine condition flags in PSTATE register
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:
```
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**

```assembly
... 
  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:
```

**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

\[
\begin{align*}
\text{cbz } Xn, \text{ label} & \quad \text{Branch to label if } Xn \text{ is zero} \\
\text{cbz } Wn, \text{ label} & \quad \text{Branch to label if } Wn \text{ is zero} \\
\text{cbnz } Xn, \text{ label} & \quad \text{Branch to label if } Xn \text{ is nonzero} \\
\text{cbnz } Wn, \text{ label} & \quad \text{Branch to label if } Wn \text{ is nonzero}
\end{align*}
\]
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 4
...n: .skip 4
...mov x1, 2
...adr x0, a
...ls1 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]. Let's take it one step at a time…
Arrays: Brute Force

Assembly

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

- x0
- x1: 2
- w2

Memory

<table>
<thead>
<tr>
<th>a</th>
<th>0</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<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

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

| x0 | 1000 |
| x1 | 2    |
| w2 |      |

Memory

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

Assembly

```
.section " .bss"
a: .skip 400
i: .skip 4
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>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>8</td>
</tr>
<tr>
<td>w2</td>
<td></td>
</tr>
</tbody>
</table>

Memory

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

### Assembly

```
.section " .bss"
a: .skip 400
i: .skip 4
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>1008</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>8</td>
</tr>
<tr>
<td>w2</td>
<td></td>
</tr>
</tbody>
</table>

### Memory

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>42</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

```
.section ".bss"
a: .skip 400
i: .skip 4
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>a</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>1004</td>
</tr>
<tr>
<td>99</td>
<td>1396</td>
</tr>
<tr>
<td>i</td>
<td>1400</td>
</tr>
<tr>
<td>n</td>
<td>1404</td>
</tr>
</tbody>
</table>
Arrays: Brute Force

Assembly

```
.section "".bss"
a: .skip 400
i: .skip 4
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>1404</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>a</th>
<th>0</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1004</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1008</td>
</tr>
<tr>
<td></td>
<td>99</td>
<td>1396</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

```
.section "\".bss\"
  a: .skip 400
  i: .skip 4
  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>1404</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>8</td>
</tr>
<tr>
<td>w2</td>
<td>42</td>
</tr>
</tbody>
</table>

Memory

```
    0     1     2     99  i  n
    1000  1004  1008  1396 1400 1404
```

33
Arrays: Register Offset Addressing

**C**

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

**Brute-Force**

```assembler
.section ".bss"
a: .skip 400
i: .skip 4
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]
...
```

**Register Offset**

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

This uses a different *addressing mode* for the load
Memory Addressing Modes

Address loaded:

- **ldr Wt, [Xn, offset]**
  - $X_n + \text{offset}$ \((-2^8 \leq \text{offset} < 2^{14})\)

- **ldr Wt, [Xn]**
  - $X_n$ \(\text{(shortcut for offset=0)}\)

- **ldr Wt, [Xn, Xm, LSL n]**
  - $X_n + (X_m \ll n)$ \((n = 3 \text{ for 64-bit, 2 for 32-bit})\)

- **ldr Wt, [Xn, Xm]**
  - $X_n + X_m$

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

- **ldr Xt, [Xn, offset]**
  - $X_n + \text{offset}$

etc.
Agenda

- Flattened C
- Control flow with signed integers
- Control flow with unsigned integers
- Arrays
  - Structures
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 ???
```
Q: Which addressing mode is most appropriate for the last store?

A. \texttt{str Wt, [Xn, offset]}
B. \texttt{str Wt, [Xn]}
C. \texttt{str Wt, [Xn, Xm LSL n]}
D. \texttt{str Wt, [Xn, Xm]}

![Diagram of x0 and RAM with values 18 and 19]
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]
```
Structures: Padding

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:
Compiler sometimes inserts padding after fields

Three-byte pad here

4, not 1
## Structures: Padding

### AARCH64 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 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

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: \texttt{adds dest, src1, src2}

- Compute sum (src1+src2)
- Assign sum to \texttt{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:
  \((src1 > 0 && src2 > 0 && sum < 0) || (src1 < 0 && src2 < 0 && sum >= 0)\)
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)`
Using Condition Flags

Question
• How do conditional branch instructions use the condition flags?

Answer
• (See following slides)
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>bls label</td>
<td>C</td>
</tr>
<tr>
<td>bhi label</td>
<td>~(C</td>
</tr>
</tbody>
</table>

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

Why does blo branch iff C? Informal explanation:

(1) largenum – smallnum (not below)
  • Correct result
  • ⇒ C=0 ⇒ don’t branch

(2) smallnum – largenum (below)
  • Incorrect result
  • ⇒ C=1 ⇒ 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
   - $\Rightarrow V=0, N=0, V^N==0 \Rightarrow$ don’t branch

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

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

4. smallnegnum – largenegnum (not less than)
   - Certainly correct result
   - $\Rightarrow V=0, N=0 \Rightarrow (V^N)\!==\!0 \Rightarrow$ don't branch
(5) posnum – negnum (not less than)
   • Suppose correct result
   • ⇒ V=0, N=0 ⇒ (V^N)==0 ⇒ don't branch

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

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

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