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

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

if (expr)
{
    statementT1;
    ...
    statementTN;
}
else
{
    statementF1;
    ...
    statementFN;
}
C
while (expr)
{  statement1;
   ...
   statementN;
}

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

Flattened C

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

for (expr1; expr2; expr3)
{  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

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

Flattened 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
  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`
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:
```

Missing anything?
Control Flow with Signed Integers

Unconditional branch

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

Compare

\[
\text{cmp Xm, Xn} \quad \text{Compare Xm to Xn} \\
\text{cmp Wm, Wn} \quad \text{Compare Wm to Wn}
\]

• Set condition flags in PSTATE register

Conditional branches after comparing signed integers

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

• 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

```
b label  b label    Branch to label
```

Compare

```
cmp Xm, Xn  cmp Xm, Xn  Compare Xm to Xn
  cmp Wm, Wn  cmp Wm, Wn  Compare Wm to Wn
```

• Set condition flags in PSTATE register

Conditional branches after comparing **unsigned** integers

```
beq label  beq label   Branch to label if equal
  bne label  bne label   Branch to label if not equal
  blt label  blo label   Branch to label if lower
  ble label  bls label   Branch to label if lower or same
  bgt label  bhi label   Branch to label if higher
  bge label  bhs label   Branch to label if higher or same
```

• 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

```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
...  
    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

```
cbz Xn, label  Branch to label if Xn is zero
cbz Wn, label  Branch to label if Wn is zero
cbnz Xn, label  Branch to label if Xn is nonzero
cbnz Wn, label  Branch to label if Wn is nonzero
```
Agenda

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

Arrays

Structures
To do array lookup, need to compute address of \( a[i] \equiv *(a+i) \)

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

**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]
...
```

**Registers**

- `x0`
- `x1` 2
- `w2`

**Memory**

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

- x0: 1000
- x1: 2
- w2: 42

Memory

- a: [0: 1000, 1: 1004, 2: 1008, ...]
- 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

| x0 | 1000 |
| x1 |  8   |
| w2 |

Memory

```
  0 | 1000 |
  1 | 1004 |
  2 |  42  |
  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>1008</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Memory

```
a
  0  |  1  |  2  | ...  |
  1000| 1004| 1008| ...  |
  1396| 1400| 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

- x0: 1008
- x1: 8
- w2: 42

Memory

```

<table>
<thead>
<tr>
<th></th>
<th>Memory</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>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 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 | 1   | 2   | 99  |
----|----|-----|-----|-----|
    | 1000 | 1004 | 1008 |
    |   a  |   i  |   n  |
    |   42 | 1400 | 1404 |
```
Arrays: Brute Force

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]
...
```

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

| 0 | 1000 |
| 1 | 1004 |
| 2 | 1008 |
| ... |
| 99 | 1396 |
| i | 1400 |
| n | 42 |
| 42 | 1404 |
Arrays: Register Offset Addressing

C

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

Brute-Force

```asm
.section "bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
.section "text"
...
mov x1, 2
...
adrx0, a
lslx1, x1, 2
addx0, x0, x1
ldrw2, [x0]
adrx0, n
strw2, [x0]
...
```

Register Offset

```asm
.section "bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
.section "text"
...
mov x1, 2
...
adrx0, a
ldrw2, [x0, x1, lsl 2]
adrx0, n
strw2, [x0]
...
```

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.

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

C

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

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

Three-byte pad here

4, not 1
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
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
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
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:
  
  \[(src1 > 0 && src2 > 0 && sum < 0) ||
  (src1 < 0 && src2 < 0 && sum >= 0)\]
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:  
  
  \[(src1 > 0 && src2 < 0 && sum < 0) || (src1 < 0 && src2 > 0 && sum >= 0)\]
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
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 &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
Conditional Branches: Signed

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:**

- Certainly correct result
- $\Rightarrow V=0, \ N=0, \ V^N==0 \Rightarrow$ don't branch

<table>
<thead>
<tr>
<th>Case</th>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>largeposnum – smallposnum (not less than)</td>
<td>$\Rightarrow V=0, \ N=0, \ V^N==0 \Rightarrow$ don't branch</td>
</tr>
<tr>
<td>(2)</td>
<td>smallposnum – largeposnum (less than)</td>
<td>$\Rightarrow V=0, \ N=1, \ V^N==1 \Rightarrow$ branch</td>
</tr>
<tr>
<td>(3)</td>
<td>largenegnum – smallnegnum (less than)</td>
<td>$\Rightarrow V=0, \ N=1 \Rightarrow (V^N)==1 \Rightarrow$ branch</td>
</tr>
<tr>
<td>(4)</td>
<td>smallnegnum – largenegnum (not less than)</td>
<td>$\Rightarrow V=0, \ N=0 \Rightarrow (V^N)==0 \Rightarrow$ don't branch</td>
</tr>
</tbody>
</table>
Conditional Branches: Signed

(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