Assembly Language

Part 2
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) {
    statementT1;
    ...
    statementTN;
} else {
    statementF1;
    ...
    statementFN;
}

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

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

C

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

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

Flattened C

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

expr1;
loop1:
  if (! expr2) goto endloop1;
  ...
  statementN;
  expr3;
  goto loop1;
endloop1:
Agenda

Flattened C code

**Control flow with signed integers**

Control flow with unsigned integers

Arrays

Structures
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)
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:
```

Note:

**ble** instruction (conditional branch if less than or equal)

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:
We *could* store here, but not needed for this code.
for Example

C
int power = 1;
int base;
int exp;
int i;
...
for (i = 0; i < exp; i++)
    power *= base;

Flattened 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 (globals) `power, base, exp, i` go into?

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
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
...
```
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
```

Missing anything?
Control Flow with Signed Integers

Unconditional branch

```
label
```

Branch to label

Compare

```
cmp Xm, Xn
```

Compare Xm to Xn

```
cmp Wm, Wn
```

Compare Wm to Wn

- Set condition flags in PSTATE register

Conditional branches after comparing signed integers

```
beq label
```

Branch to label if equal

```
bne label
```

Branch to label if not equal

```
blt label
```

Branch to label if less than

```
ble label
```

Branch to label if less or equal

```
bgt label
```

Branch to label if greater than

```
bge label
```

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
```

Compare

```
cmp Xm, Xn
cmp Wm, Wn
```

- Set condition flags in PSTATE register

Conditional branches after comparing **unsigned** integers

```
beq label
bne label
blt label
ble label
bgt label
bge label
beq label
bne label
blo label
bhi label
bhs label
```

- Examine condition flags in PSTATE register
**while Example**

<table>
<thead>
<tr>
<th>Flattened C</th>
<th>Assembly: Signed → Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unsigned int n;</code></td>
<td></td>
</tr>
<tr>
<td><code>unsigned int fact;</code></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td><code>fact = 1;</code></td>
<td></td>
</tr>
<tr>
<td><code>loop1:</code></td>
<td></td>
</tr>
<tr>
<td><code>if (n &lt;= 1)</code></td>
<td></td>
</tr>
<tr>
<td><code>goto endloop1;</code></td>
<td></td>
</tr>
<tr>
<td><code>fact *= n;</code></td>
<td></td>
</tr>
<tr>
<td><code>n--;</code></td>
<td></td>
</tr>
<tr>
<td><code>goto loop1;</code></td>
<td></td>
</tr>
<tr>
<td><code>endloop1:</code></td>
<td></td>
</tr>
</tbody>
</table>

Note:

`bls` instruction (instead of `ble`)

22
Alternative Control Flow: CBZ, CBNZ

Special-case, all-in-one compare-and-branch instructions

• DO NOT examine condition flags in PSTATE register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cbz Xn, label</code></td>
<td>Branch to label if Xn is zero</td>
</tr>
<tr>
<td><code>cbz Wn, label</code></td>
<td>Branch to label if Wn is zero</td>
</tr>
<tr>
<td><code>cbnz Xn, label</code></td>
<td>Branch to label if Xn is nonzero</td>
</tr>
<tr>
<td><code>cbnz Wn, label</code></td>
<td>Branch to label if Wn is nonzero</td>
</tr>
</tbody>
</table>
Agenda

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

Arrays

Structures
Arrays: Brute Force (Setup)

C

```c
int a[100];
long i;
int n;
...
i = 99;
...
n = a[i]
...
```

Assembly

```assembly
    .section "bss"
    a: .skip 400
    i: .skip 8
    n: .skip 4
    ...
    .section "text"
    ...
    mov x1, 99
    ...
    adr x0, a
    lsl x1, x1, 2
    add x0, x0, x1
    ldr w2, [x0]
    adr x0, n
    str w2, [x0]
    ...
```

Memory

```
  a  
  1000 1004 1008 1396 1400 1408
  i  
  n  
```

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 (Initialize i)

Assembly

```assembly
[section "bss"
  a: .skip 400
  i: .skip 8
  n: .skip 4
...
[section "text"
  ...
  mov x1, 99
...
  adr x0, a
  lsl x1, x1, 2
  add x0, x0, x1
  ldr w2, [x0]
  adr x0, n
  str w2, [x0]
  ...
```

Registers

- x0
- x1
- w2

Memory

- x0: 1000
- x1: 1004
- w2: 1008
- i: 42
- n: 1396
- a: 1400
- i: 1408
Arrays: Brute Force (Get a's base address)

Assembly

```
.section "bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
.section "text"
...
    mov x1, 99
...
    adr x0, a
    lsl x1, x1, 2
    add x0, x0, x1
    ldr w2, [x0]
    adr x0, n
    str w2, [x0]
...
```

Registers

- x0: 1000
- x1: 99
- w2: (not shown)

Memory

- a: 1000
- i: 1396
- n: 1408
Arrays: Brute Force (Calculate byte-offset of $i$)

Assembly

```
.section ".bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
.section ".text"
...
  mov x1, 99
...
  adr x0, a
  lsl x1, x1, 2
  add x0, x0, x1
  ldr w2, [x0]
  adr x0, n
  str w2, [x0]
```

Registers

- $x0$: 1000
- $x1$: 396
- $w2$

Memory

- $a$: 1000, 1004, 1008
- $i$: 1396, 1400
- $n$: 1408
Arrays: Brute Force (Calculate address of a[i])

Assembly

```assembly
.section "bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
.section "text"
...
mov x1, 99
...
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>1396</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>396</td>
</tr>
<tr>
<td>w2</td>
<td></td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>a</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1004</td>
</tr>
<tr>
<td></td>
<td>1008</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>1396</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1400</td>
</tr>
<tr>
<td></td>
<td>1408</td>
</tr>
</tbody>
</table>
Arrays: Brute Force (Read value at \(a[i]\) into \(w2\))

### Assembly

```assembly
.section ".bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
.section ".text"
...
mov x1, 99
...
ad x0, a
lsl x1, x1, 2
add x0, x0, x1
ldr w2, [x0]
ad x0, n
str w2, [x0]
...
```

### Registers

- **\(x0\)**: 1396
- **\(x1\)**: 396
- **\(w2\)**: 42

### Memory

- **\(a\)**: 1000, 1004, 1008
- **\(i\)**: 1396, 1400, 1408
- **\(n\)**:
Arrays: Brute Force (Get n's address)

Assembly

```
section "\bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
section "\text"
...
mov x1, 99
...
adr x0, a
lsl x1, x1, 2
add x0, x0, x1
ldr w2, [x0]
adr x0, n
str w2, [x0]
...
```

Registers

- x0: 1408
- x1: 396
- w2: 42

Memory

- a:
  - 1000
  - 1004
  - 1008
- i:
  - 42
  - 1396
  - 1400
- n:
  - 1408
Arrays: Brute Force (Store value into n)

Assembly

```
.section ".bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
.section ".text"
...
mov x1, 99
...
adr x0, a
lsl 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, 1004, 1008
- i: 42, 1396, 1400
- n: 42, 1408
Arrays: Register Offset Addressing

C

```
int a[100];
long i;
int n;
...
i = 99;
...
n = a[i]
...
```

Brute-Force

```
.section ".bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
.section ".text"
...
mov x1, 99
...
adr x0, a
lsl x1, x1, 2
add x0, x0, x1
ldr w2, [x0]
adr x0, n
str w2, [x0]
...
```

Register Offset

```
.section ".bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
.section ".text"
...
mov x1, 99
...
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]`: \( Xn + \text{offset} \) (\(-2^8 \leq \text{offset} < 2^{14}\))
- `ldr Wt, [Xn]`: \( Xn \) (shortcut for \text{offset}=0)
- `ldr Wt, [Xn, Xm]`: \( Xn + Xm \)
- `ldr Wt, [Xn, Xm, LSL n]`: \( Xn + (Xm \ll n) \) (\( n = 3 \) for 64-bit elements, 2 for 32-bit elements, ...)

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

- `ldr Xt, [Xn, offset]`: \( Xn + \text{offset} \)

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 to store `myStruct.j`?

A. `str W1, [X0, offset]`
B. `str W1, [X0]`
C. `str W1, [X0, Xm, LSL 2]`
D. `str W1, [X0, Xm]`

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

```
.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
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:
As we've seen, the Compiler sometimes inserts padding after fields
So now that you're the "Compiler" ...
## 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
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: \texttt{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:
  \[(\text{src1} > 0 \&\& \text{src2} > 0 \&\& \text{sum} < 0) \text{ } || \text{ } (\text{src1} < 0 \&\& \text{src2} < 0 \&\& \text{sum} \geq 0)\]
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>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) \textit{does} cause carry
• ⇒ C=1 ⇒ don’t branch

(2) smallnum – largenum (below)

• smallnum + (two’s complement of largenum) \textit{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>$\sim Z$</td>
</tr>
<tr>
<td>blt label</td>
<td>$V \land N$</td>
</tr>
<tr>
<td>bge label</td>
<td>$(V \land N)$</td>
</tr>
<tr>
<td>ble label</td>
<td>$(V \land N) \lor Z$</td>
</tr>
<tr>
<td>bgt label</td>
<td>$(V \land N) \land Z$</td>
</tr>
</tbody>
</table>

**Note:**
- If you can understand why `blt` branches iff $V\land N$
- ... 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

<table>
<thead>
<tr>
<th>No.</th>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5)</td>
<td>posnum – negnum (not less than)</td>
<td>Suppose correct result: $V=0, N=0 \Rightarrow (V^N)\neq 0 \Rightarrow$ don't branch</td>
</tr>
<tr>
<td>(6)</td>
<td>posnum – negnum (not less than)</td>
<td>Suppose incorrect result: $V=1, N=1 \Rightarrow (V^N)\neq 0 \Rightarrow$ don't branch</td>
</tr>
<tr>
<td>(7)</td>
<td>negnum – posnum (less than)</td>
<td>Suppose correct result: $V=0, N=1 \Rightarrow (V^N)=1 \Rightarrow$ branch</td>
</tr>
<tr>
<td>(8)</td>
<td>negnum – posnum (less than)</td>
<td>Suppose incorrect result: $V=1, N=0 \Rightarrow (V^N)=1 \Rightarrow$ branch</td>
</tr>
</tbody>
</table>