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 contains nested statements
Solution
- Flatten the C code to eliminate all nesting

Flattened C Code

```c
if (expr)
    { statement1;
      ...
      statementN;
    }
else
    { statementF1;
      ...
      statementFN;
    }
```

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

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

loop1:
    if (! expr1 goto endloop1;
        statement1;
        ...
        statementN;
        goto loop1;
        endloop1;

expr1;
loop1:
    if (! expr2 goto endloop1;
        ...
        statementN;
        ...
        expr2;
        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;
endif;
```

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:
```

Notes:

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

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

while Example

C

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

Flattened C

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

Notes:

- `b` instruction (unconditional branch)
while Example

Flattened C

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

n = 0;
goto loop1;
endloop1;

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

for Example

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:

Assembly

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

Control Flow with Signed Integers

Unconditional branch

label:
... cmp X, 0
bge label
...
...
...

Compare

cmp X, X
...
...
...

• Set condition flags in PSTATE register

Conditional branches after comparing signed integers

bne label
...
...
...

• Examine condition flags in PSTATE register

iClicker Question

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

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
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

Control Flow with Unsigned Integers

Unconditional branch
- Label: Branch to label

Compare
- cmp Xn, Xn: Compare Xn to Xn
- cmp Wn, Wn: Compare Wn to Wn
- Set condition flags in PSTATE register

Conditional branches after comparing unsigned integers
- ble label: Branch to label if equal
- bne label: Branch to label if not equal
- bhi label: Branch to label if higher
- bh label: Branch to label if lower or same

- Examine condition flags in PSTATE register

while Example

C
- unsigned int fact;
- ...
- fact = 1;
- while (n > 1) { fact *= n; n--;

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

Alternative Control Flow: CBZ, CBNZ

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

bcb Xn, label: Branch to label if Xn is zero
bhs Xn, label: Branch to label if Xn is higher or same
bhs Xn, label: Branch to label if Xn is higher
bhs Xn, label: Branch to label if Xn is higher or same
bhs Xn, label: Branch to label if Xn is higher or same
bhs Xn, label: Branch to label if Xn is higher...
Agenda

1. Flattened C
2. Control flow with signed integers
3. Control flow with unsigned integers
4. Arrays
5. Structures

Arrays: Brute Force

To do array lookup, need to compute address of 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]
...
```

C

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

i = 2;
...

n = a[i]  
```

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]
str w2, [x0]
...
```

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]
str w2, [x0]
...
```

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]
str w2, [x0]
...
```

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]
str w2, [x0]
...
```
Arrays: Brute Force

**Assembly**
```
.section .text
    .skip 400
    .skip 4
    .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]
```

**Memory**
```
<table>
<thead>
<tr>
<th>x0</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>1004</td>
</tr>
<tr>
<td>w2</td>
<td>1400</td>
</tr>
<tr>
<td></td>
<td>1396</td>
</tr>
<tr>
<td>n</td>
<td>1404</td>
</tr>
</tbody>
</table>
```

Agenda

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

C

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

Assembly

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

Structures: Offset Addressing

C

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

Brute-Force

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

Offset

```assembly
.adr x0, myStruct
mov w1, 18
str w1, [x0]
...;
mov w1, 19
add x0, x0, 4
str w1, [x0]
```

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

Intermediate aspects of AARCH64 assembly language...

Flattened C code

Control transfer with signed integers

Control transfer with unsigned integers

Arrays
- Addressing modes
- Structures
  - Padding

Summary

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]
Appendix

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

Example:
- **adds dest, src1, src2**
  - Compute sum \( \text{src1} + \text{src2} \)
  - Assign sum to \( \text{dest} \)
  - **N**: set to 1 iff sum < 0
  - **Z**: set to 1 iff sum == 0
  - **C**: set to 1 iff unsigned overflow: \( \text{src1} + \text{src2} < 0 \) or \( \text{src1} + \text{src2} > 0 \) and \( \text{sum} < 0 \)
  - **V**: set to 1 iff signed overflow:
    - \( \text{src1} > 0 \) and \( \text{src2} < 0 \) and \( \text{sum} < 0 \)
    - \( \text{src1} < 0 \) and \( \text{src2} > 0 \) and \( \text{sum} >= 0 \)

Condition Flags

Example: **cmp src1, src2**
- Recall that this is a shorthand for ```sub src1, src2```  
  - Compute sum \( \text{src1} + (-\text{src2}) \)
  - 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:
    - \( \text{src1} > 0 \) and \( \text{src2} < 0 \) and \( \text{sum} < 0 \)
    - \( \text{src1} < 0 \) and \( \text{src2} > 0 \) and \( \text{sum} >= 0 \)

Using Condition Flags

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 Z</td>
<td></td>
</tr>
<tr>
<td>bne label ~Z</td>
<td></td>
</tr>
<tr>
<td>bgt label C</td>
<td></td>
</tr>
<tr>
<td>blt label (~C) Z</td>
<td></td>
</tr>
<tr>
<td>bhi label C &amp; (~Z)</td>
<td></td>
</tr>
</tbody>
</table>

Note:
- If you can understand why ble branches iff ~C
- ... then the others follow

Why does bfe 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 Z</td>
<td></td>
</tr>
<tr>
<td>bne label ~Z</td>
<td></td>
</tr>
<tr>
<td>bge label ~V^N</td>
<td></td>
</tr>
<tr>
<td>ble label (V^N)~Z</td>
<td></td>
</tr>
<tr>
<td>bgt label (V^N)</td>
<td></td>
</tr>
<tr>
<td>bhi label C &amp; (~Z)</td>
<td></td>
</tr>
</tbody>
</table>

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

Why does blt branch ifl 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

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