



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

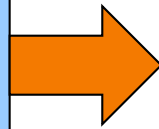
- **Flatten** the C code to eliminate all nesting

Flattened C Code



C

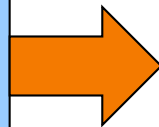
```
if (expr)
{
    statement1;
    ...
    statementN;
}
```



Flattened C

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

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



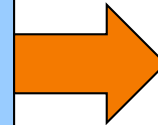
```
    if (! expr) goto else1;
        statementT1;
        ...
        statementTN;
        goto endif1;
else1:
    statementF1;
    ...
    statementFN;
endif1:
```

Flattened C Code



C

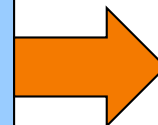
```
while (expr)  
{ statement1;  
  ...  
  statementN;  
}
```



Flattened C

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

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



```
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

```
int i;  
...  
if (i < 0)  
    i = -i;
```

Flattened C

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




if Example

Flattened 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:
```

Assembler shorthand for
`subs wzr, w1, 0`

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

```
int i;  
int j;  
int smaller;  
...  
if (i < j)  
    smaller = i;  
else  
    smaller = j;
```

Flattened 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

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

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

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

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:
```

Note:

ble instruction (conditional branch if less than or equal)

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:
```

iClicker Question

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

```
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

```
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
...
```


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

```
...
    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 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

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 (`sdiv` vs. `udiv`)
- Control flow

Control Flow with Unsigned Integers



Unconditional branch

<code>b label</code>	Branch to label
----------------------	-----------------

Compare

<code>cmp Xm, Xn</code>	Compare Xm to Xn
<code>cmp Wm, Wn</code>	Compare Wm to Wn

- Set condition flags in PSTATE register

Conditional branches after comparing **unsigned** integers

<code>beq label</code>	Branch to label if equal
<code>bne label</code>	Branch to label if not equal
<code>blo label</code>	Branch to label if lower
<code>bls label</code>	Branch to label if lower or same
<code>bhi label</code>	Branch to label if higher
<code>bhs label</code>	Branch to label if higher or same

- Examine condition flags in PSTATE register

while Example



C

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

Flattened 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

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

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

Arrays: Brute Force



C

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

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]`.
Let's take it one step at a time...

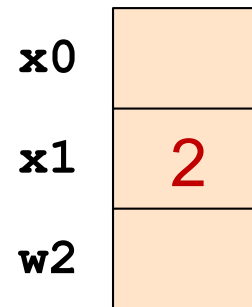
Arrays: Brute Force



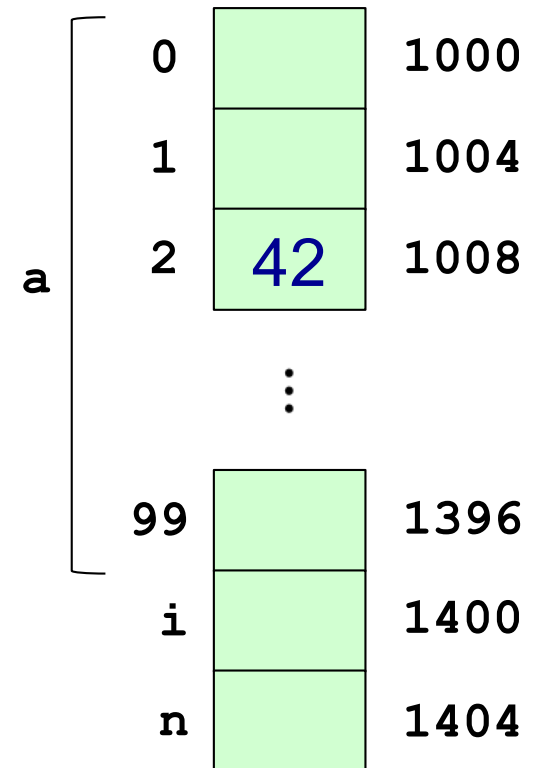
Assembly

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

Registers



Memory



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	

Memory

a	0		1000
	1		1004
	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
...
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

a	0		1000
	1		1004
	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
...
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	

Memory

a	0		1000
	1		1004
	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
...
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

a	0		1000
	1		1004
	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
...
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	0		1000
	1		1004
	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
...
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	0		1000
	1		1004
	2	42	1008
		⋮	
	99		1396
	i		1400
	n	42	1404

Arrays: Register Offset Addressing



C

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

Brute-Force

```
.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]
...
```

Register Offset

```
.section ".bss"
a: .skip 400
i: .skip 8
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]

$Xn + \text{offset}$ ($-2^8 \leq \text{offset} < 2^{14}$)

ldr Wt, [Xn]

Xn (shortcut for offset=0)

ldr Wt, [Xn, Xm, LSL n]

$Xn + (Xm \ll n)$ ($n = 3$ for 64-bit, 2 for 32-bit)

ldr Wt, [Xn, Xm]

$Xn + Xm$

All these addressing modes 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

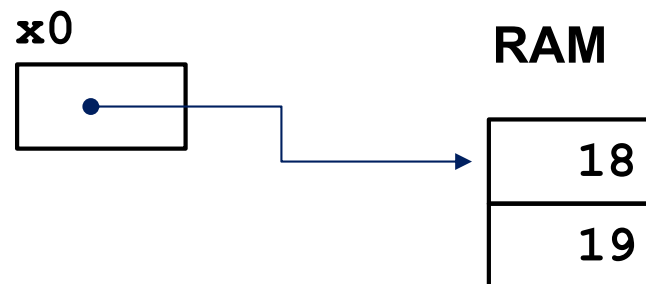


C

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

Assembly

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

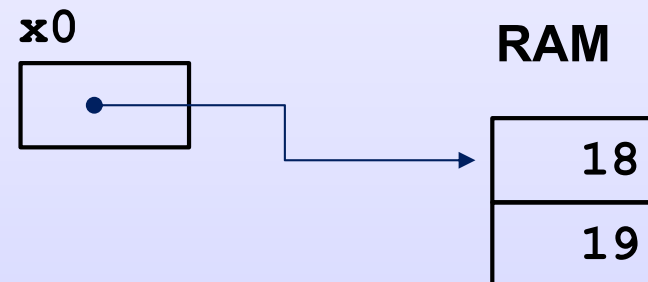


▶ iClicker Question

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]`

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



Structures: Offset Addressing



C

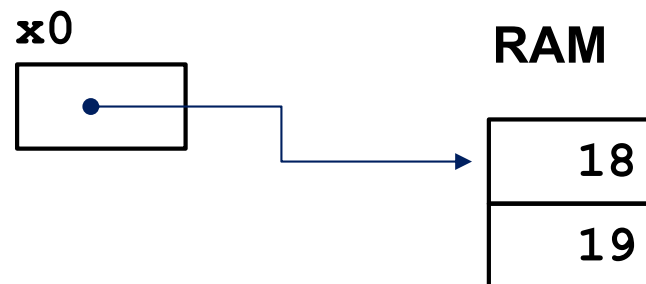
```
struct S
{ int i;
  int j;
};
...
struct S myStruct;
...
myStruct.i = 18;
...
myStruct.j = 19;
```

Brute-Force

```
.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

```
.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

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

Three-byte
pad here

Assembly

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

4, not 1

Beware:

Compiler sometimes inserts padding after fields

Structures: Padding



AARCH64 rules

Data type	Within a struct, must begin at address that is evenly divisible by:
(unsigned) char	1
(unsigned) short	2
(unsigned) int	4
(unsigned) long	8
float	4
double	8
long double	16
any pointer	8

- 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

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



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) ||`
`(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)

Conditional Branches: Unsigned



After comparing **unsigned** data

Branch instruction	Use of condition flags
beq label	Z
bne label	$\sim Z$
blo label	$\sim C$
bhs label	C
bls label	$(\sim C) \mid Z$
bhi label	$C \ \& \ (\sim Z)$

Note:

- If you can understand why **blo** branches iff $\sim C$
- ... 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
- $\Rightarrow C=1 \Rightarrow$ don't branch

(2) smallnum – largenum (below)

- smallnum + (two's complement of largenum) *does not* cause carry
- $\Rightarrow C=0 \Rightarrow$ branch

Conditional Branches: Signed



After comparing **signed** data

Branch instruction	Use of condition flags
beq label	Z
bne label	$\sim Z$
blt label	$V \wedge N$
bge label	$\sim(V \wedge N)$
ble label	$(V \wedge N) \mid Z$
bgt label	$\sim((V \wedge N) \mid Z)$

Note:

- If you can understand why **blt** branches iff $V \wedge N$
- ... then the others follow

Conditional Branches: Signed



Why does blt branch iff $V \wedge N$? Informal explanation:

(1) $\text{largeposnum} - \text{smallposnum}$ (not less than)

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

(2) $\text{smallposnum} - \text{largeposnum}$ (less than)

- Certainly correct result
- $\Rightarrow V=0, N=1, V \wedge N == 1 \Rightarrow$ branch

(3) $\text{largenegnum} - \text{smallnegnum}$ (less than)

- Certainly correct result
- $\Rightarrow V=0, N=1 \Rightarrow (V \wedge N) == 1 \Rightarrow$ branch

(4) $\text{smallnegnum} - \text{largenegnum}$ (not less than)

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

Conditional Branches: Signed



(5) posnum – negnum (not less than)

- Suppose correct result
- $\Rightarrow V=0, N=0 \Rightarrow (V \wedge N) == 0 \Rightarrow$ don't branch

(6) posnum – negnum (not less than)

- Suppose incorrect result
- $\Rightarrow V=1, N=1 \Rightarrow (V \wedge N) == 0 \Rightarrow$ don't branch

(7) negnum – posnum (less than)

- Suppose correct result
- $\Rightarrow V=0, N=1 \Rightarrow (V \wedge N) == 1 \Rightarrow$ branch

(8) negnum – posnum (less than)

- Suppose incorrect result
- $\Rightarrow V=1, N=0 \Rightarrow (V \wedge N) == 1 \Rightarrow$ branch