Assembly Language

Part 2

PRINCETON UNIVERSITY

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

What goes where?

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

```
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

E

none are string literals: not RODATA

all are file scope, process duration: not STACK

power is initialized: DATA

the rest are not: BSS

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

Flatly Iffy

C

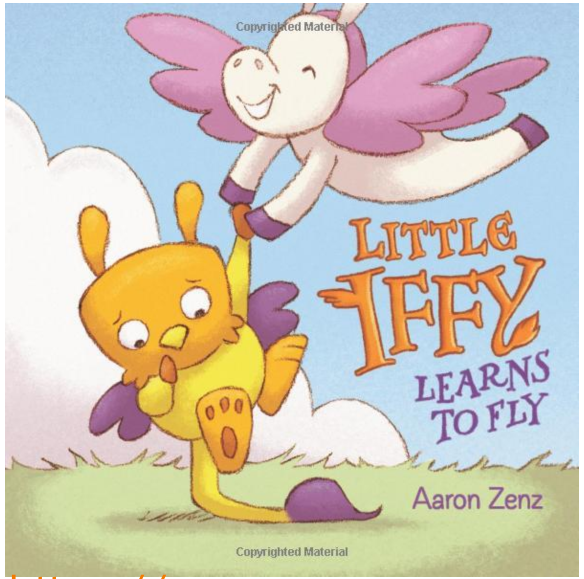
```
if (expr)
{  statement1;
   …
   statementN;
}
```

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

Flattened C

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

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



https://www.aaronzenz.com

Flatly Loopy

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:
// str w2 into fact
```

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

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:
```

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:
// str w1 into power
```

Missing anything?

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

- Examine condition flags in PSTATE register

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: Signed → Unsigned

```
...
  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

Arrays: Brute Force (Setup)

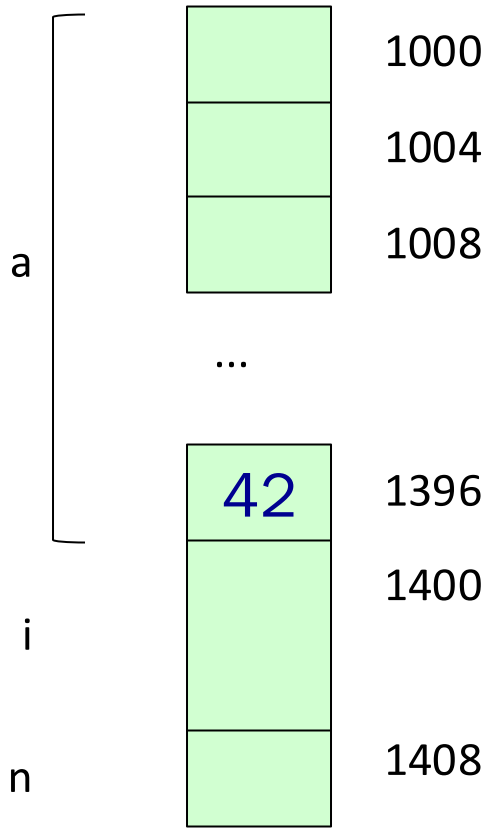
C

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

Memory

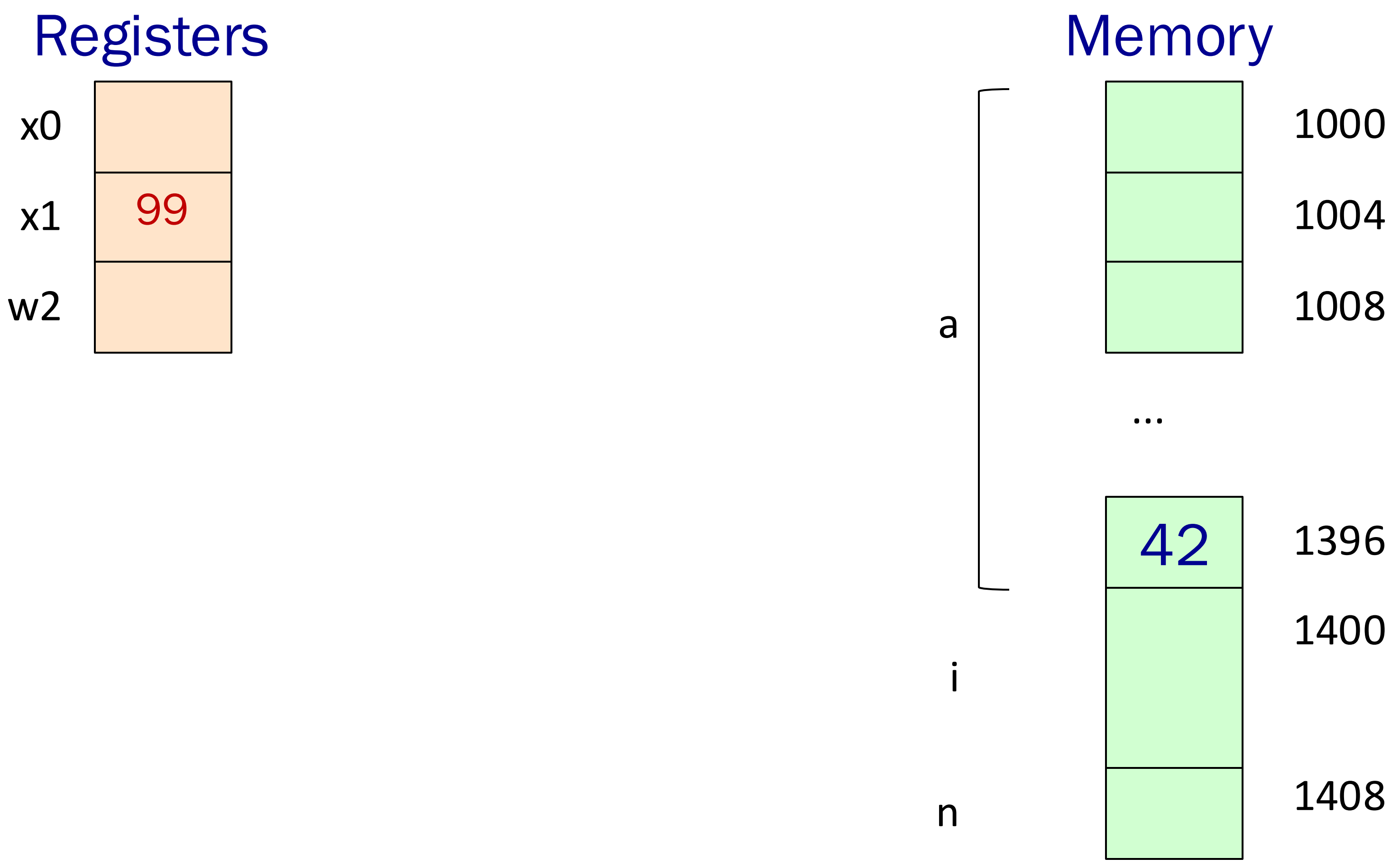


To do array lookup, need to compute address of a[i] ≡ *(a+i)
Let's take it one step at a time…

Arrays: Brute Force (Initialize i)

Assembly

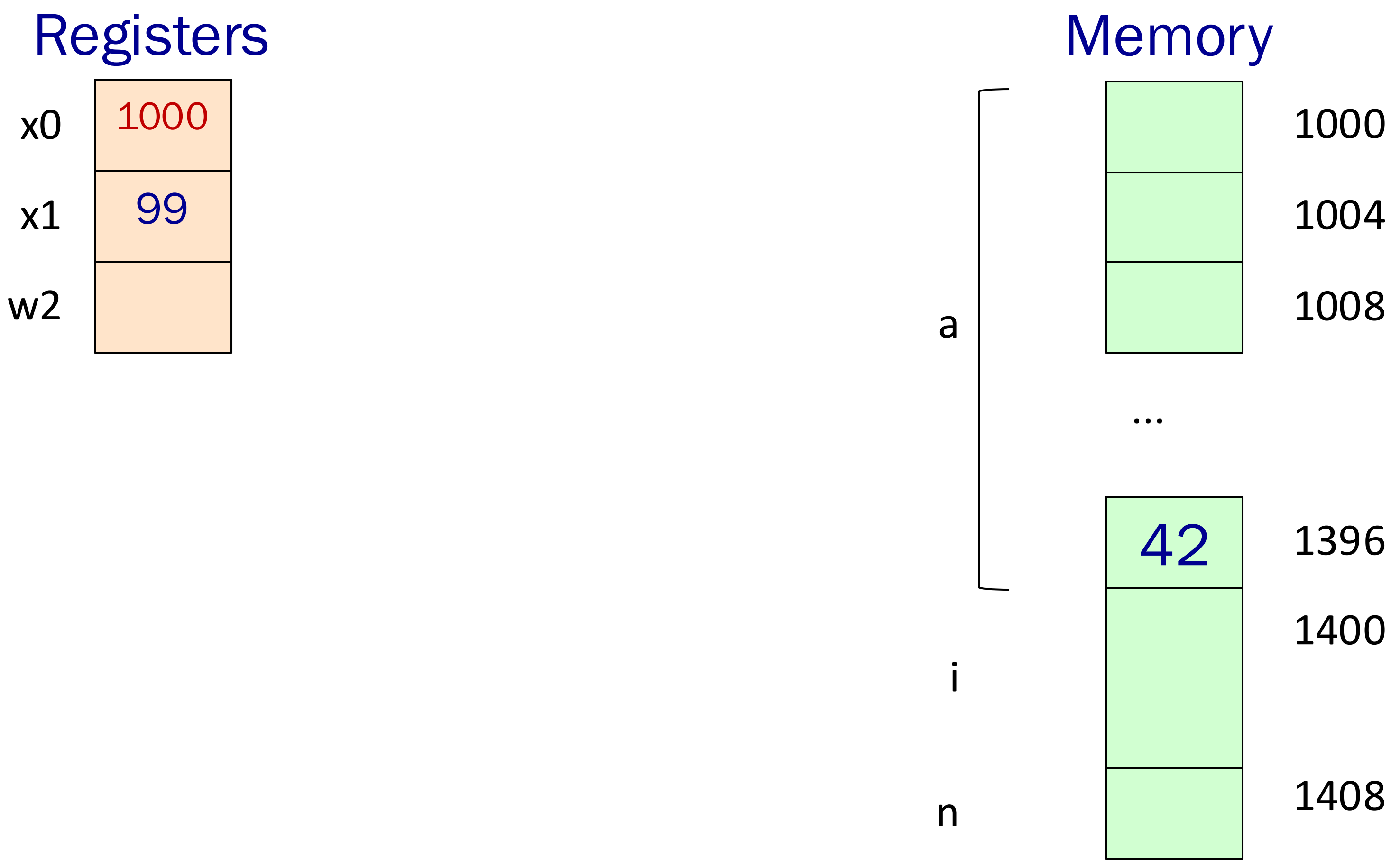
```
  .section ".bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
  .section ".text"
...
  mov x1, 99
...
  adr x0, a
  lsl x1, x1, 2
  add x0, x0, x1
  ldr w2, [x0]
  adr x0, n
  str w2, [x0]
...
```



Arrays: Brute Force (Get a's base address)

Assembly

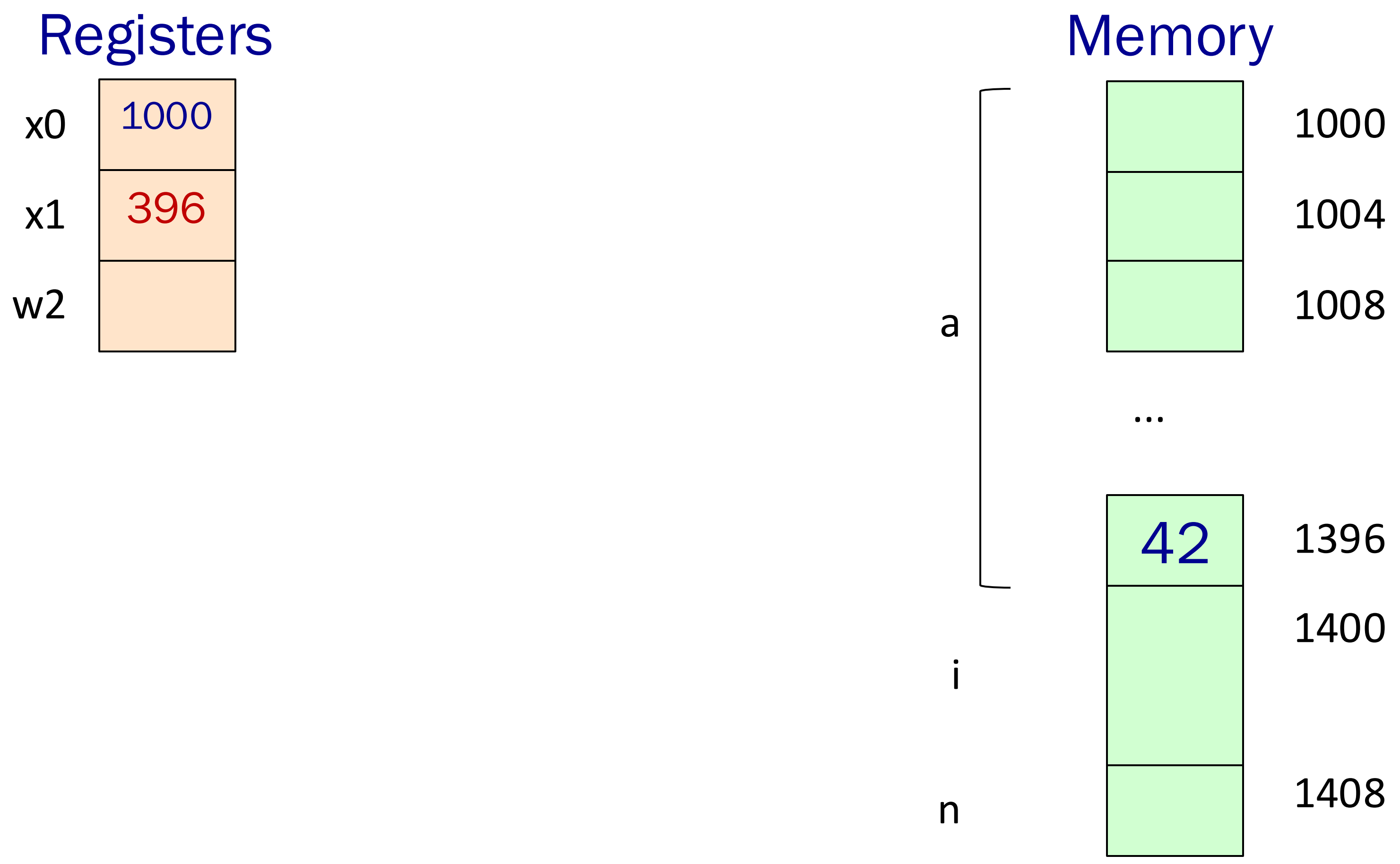
```
  .section ".bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
  .section ".text"
...
  mov x1, 99
...
  adr x0, a
  lsl x1, x1, 2
  add x0, x0, x1
  ldr w2, [x0]
  adr x0, n
  str w2, [x0]
...
```



Arrays: Brute Force (Calculate byte-offset of i)

Assembly

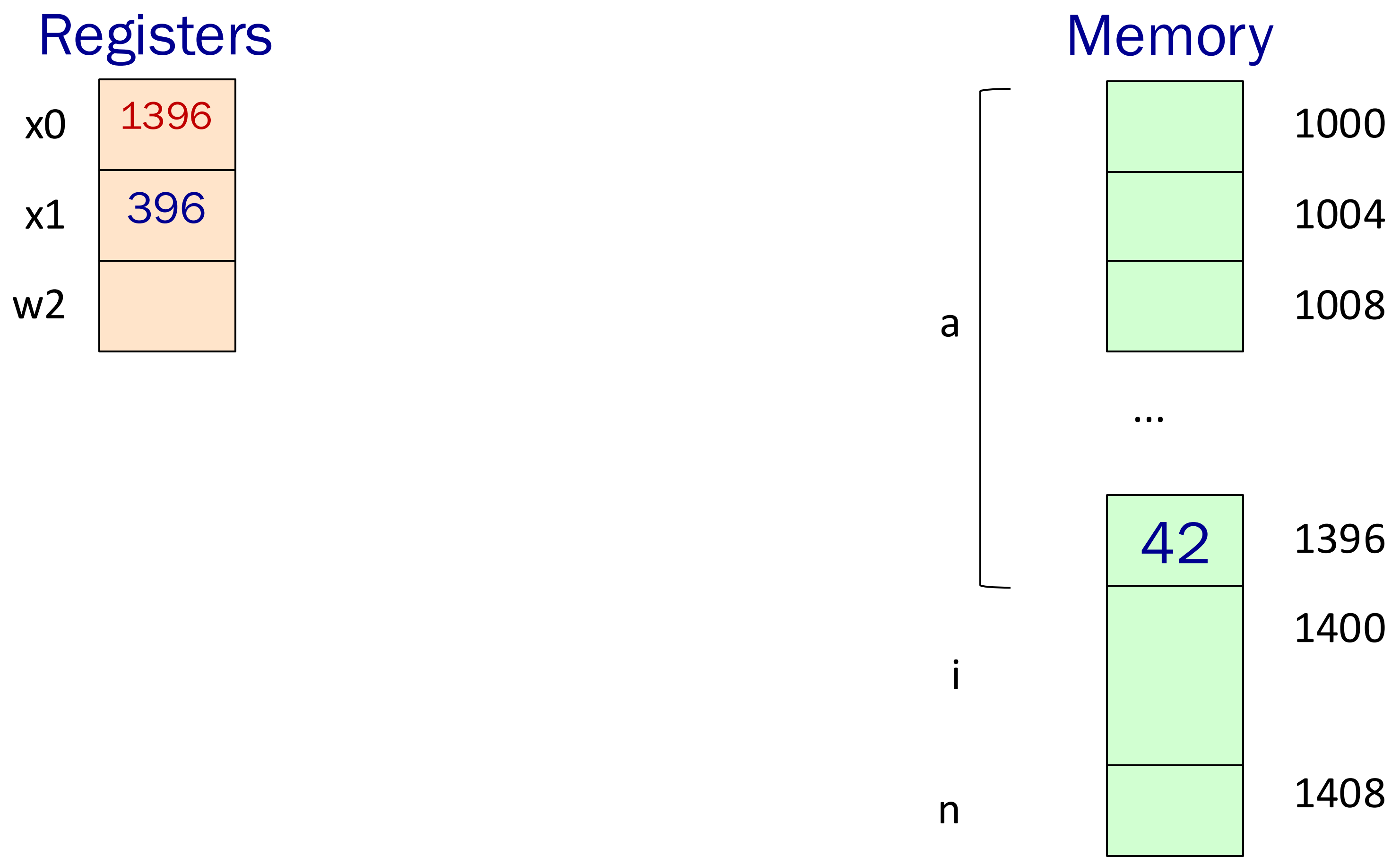
```
  .section ".bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
  .section ".text"
...
  mov x1, 99
...
  adr x0, a
  lsl x1, x1, 2
  add x0, x0, x1
  ldr w2, [x0]
  adr x0, n
  str w2, [x0]
...
```



Arrays: Brute Force (Calculate address of a[i])

Assembly

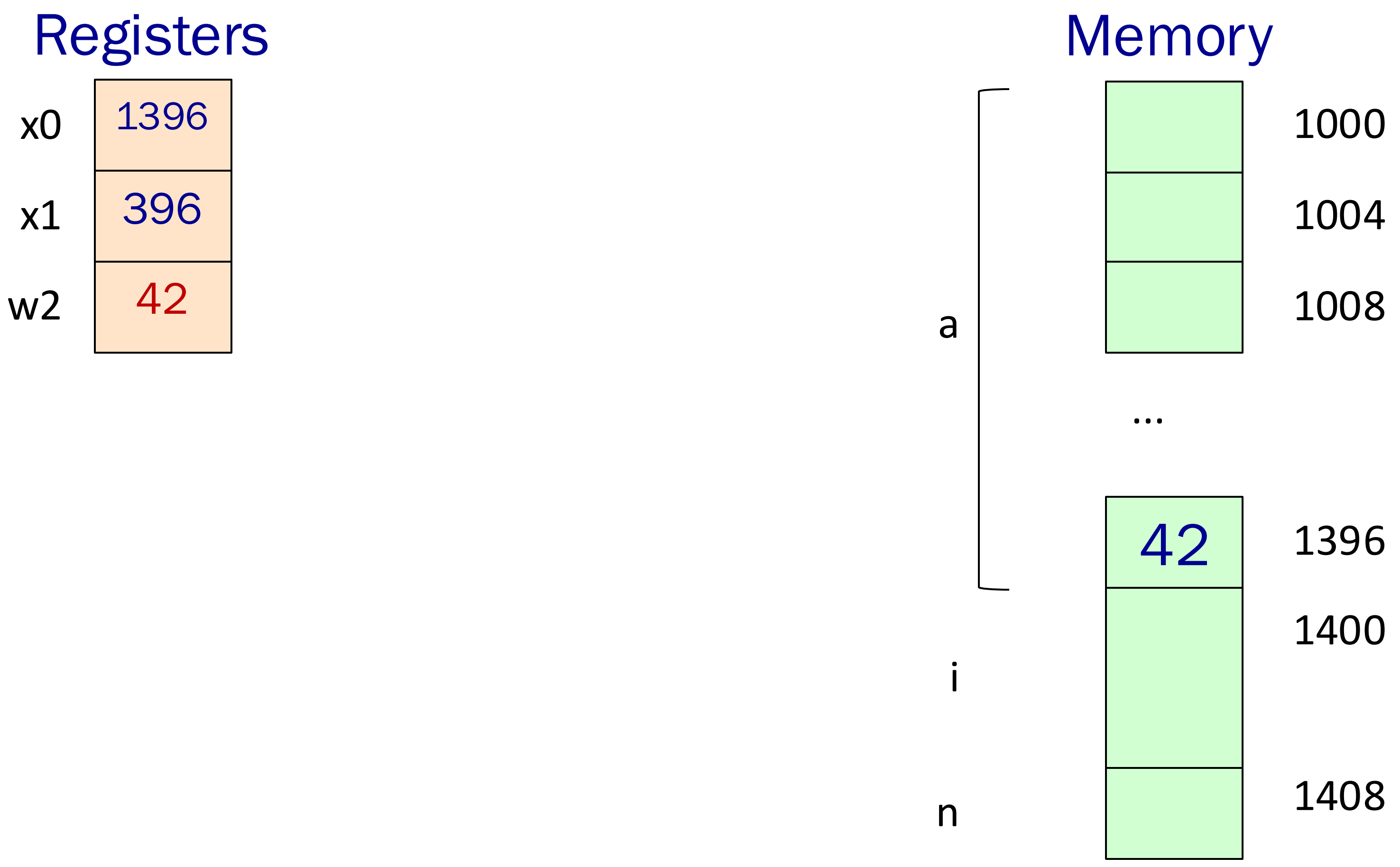
```
   .section ".bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
   .section ".text"
...
   mov x1, 99
...
   adr x0, a
   lsl x1, x1, 2
   add x0, x0, x1
   ldr w2, [x0]
   adr x0, n
   str w2, [x0]
...
```



Arrays: Brute Force (Load value at a[i] into w2)

Assembly

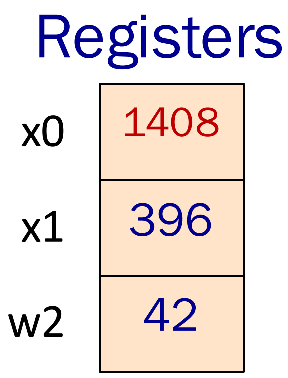
```
  .section ".bss"
a: .skip 400
i: .skip 8
n: .skip 4
...
  .section ".text"
...
  mov x1, 99
...
  adr x0, a
  lsl x1, x1, 2
  add x0, x0, x1
  ldr w2, [x0]
  adr x0, n
  str w2, [x0]
...
```

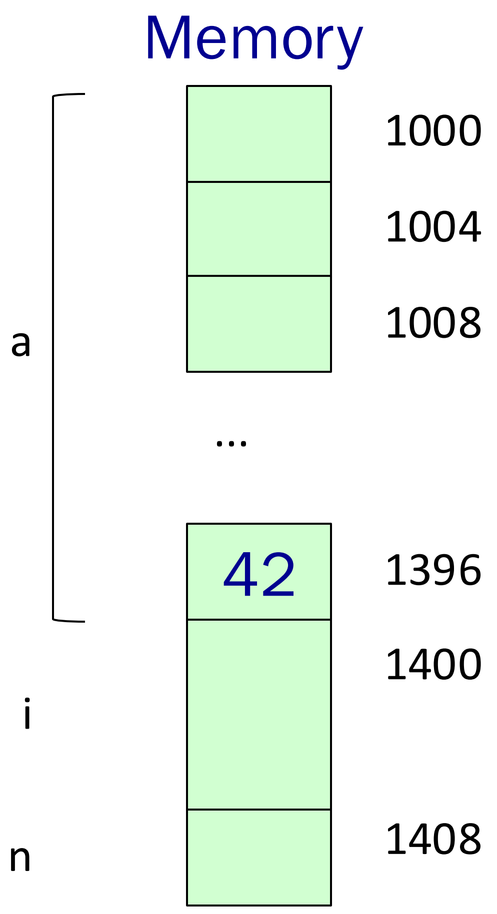


Arrays: Brute Force (Get n's address)

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]
...
```

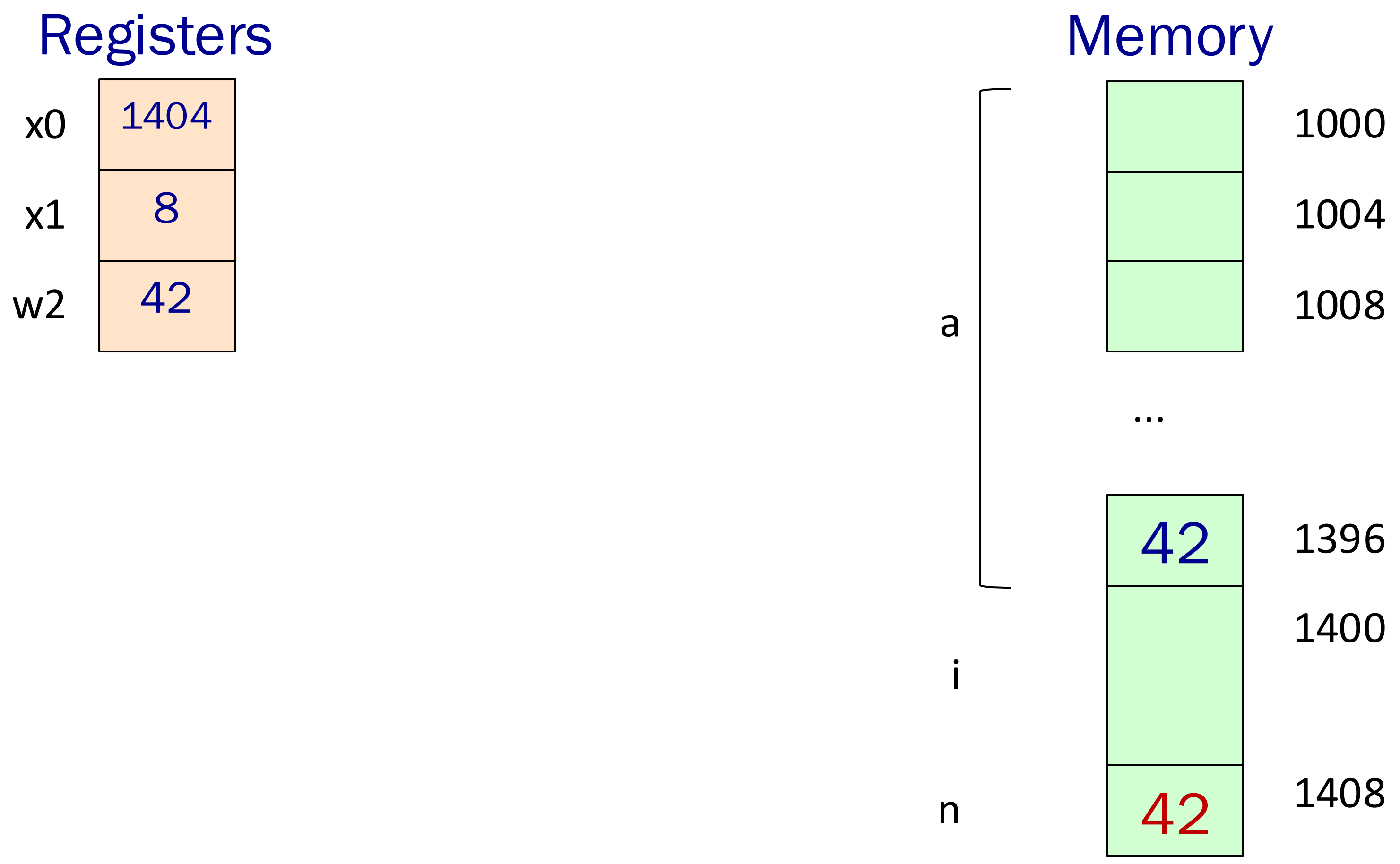




Arrays: Brute Force (Store value into n)

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]
...
```



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]
...
```

Scaled 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]
...
```

Doesn't change x0 or x1

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 = 2 for 32-bit elements, 1 for 16-bit elements using ldrh)

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

ldr Xt, [Xn, offset]	Xn+offset	

etc. (n = 3 for 64-bit elements in scaled register offset mode)

*All these addressing modes are also available for **stores** from either x or w sources.*

Agenda

Flattened C

Control flow with signed integers

Control flow with unsigned integers

Arrays

Structures

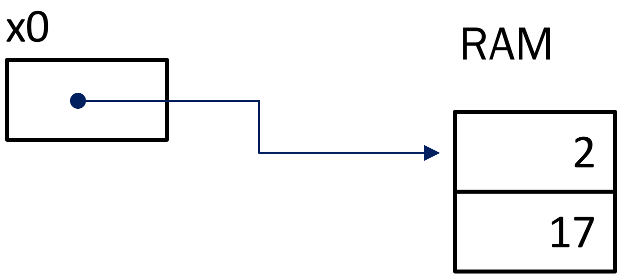
Structures: Brute Force

C

```
struct S
{ int i;
  int j;
};
...
struct S myStruct;
...
myStruct.i = 2;
...
myStruct.j = 17;
```

Assembly

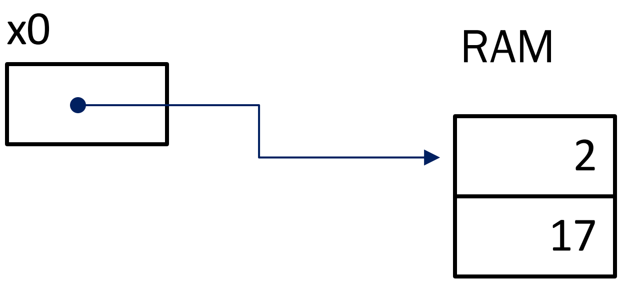
```
  .section ".bss"
myStruct: .skip 8
...
  .section ".text"
...
  adr x0, myStruct
...
  mov w1, 2
  str w1, [x0]
...
  mov w1, 17
  str ???
```



Which mode is à la mode?

Q: Which addressing mode is most appropriate to store myStruct.j?

```
  .section ".bss"
myStruct: .skip 8
...
  .section ".text"
...
  adr x0, myStruct
...
  mov w1, 2
  str w1, [x0]
...
  mov w1, 17
  str ???
```



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.

Structures: Offset Addressing

C

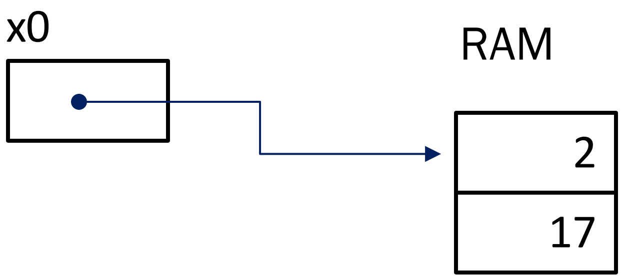
```
struct S
{ int i;
  int j;
};
…
struct S myStruct;
…
myStruct.i = 2;
…
myStruct.j = 17;
```

Brute-Force

```
   .section ".bss"
myStruct: .skip 8
…
   .section ".text"
…
   adr x0, myStruct
…
   mov w1, 2
   str w1, [x0]
…
   mov w1, 17
   add x0, x0, 4
   str w1, [x0]
```

Immediate Offset

```
   .section ".bss"
myStruct: .skip 8
…
   .section ".text"
…
   adr x0, myStruct
…
   mov w1, 2
   str w1, [x0]
…
   mov w1, 17
   str w1, [x0, 4]
```



Structures: Padding

C

```
struct S
{ char c;
  int i;
};
...
struct S myStruct;
...
myStruct.c = 'A';
...
myStruct.i = 217;
```

Three-byte pad here

Assembly

```
  .section ".bss"
myStruct: .skip 8
...
  .section ".text"
...
  adr x0, myStruct
...
  mov w1, 'A'
  strb w1, [x0]
...
  mov w1, 217
  str w1, [x0, 4]
```

Still 8, not 5

Still 4, not 1

Beware:

As we've seen, the Compiler sometimes inserts padding after fields

So now that you're the "Compiler" ...

Structures: Padding

AARCH64 rules:

Data type	Within a struct, field 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, so that first field of next element is aligned

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 an arithmetic instruction 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 || sum < 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, interpreting both as unsigned)
- 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

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	~Z
blo label	~C
bhs label	C
bls label	(~C) | Z
bhi label	C & (~Z)

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

Branch instruction	Use of condition flags
beq label	Z
bne label	~Z
blt label	V ^ N
bge label	~(V ^ N)
ble label	(V ^ N) | Z
bgt label	~((V ^ N) | Z)

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
- ⇒ 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

(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