

EECS 370 - Lecture 5

ARM to Assembly

byte be l i k e



Announcements

- HW 2
 - Posted on website, due Sunday night
- P1
 - 3 parts, first part due Wednesday
 - See walkthrough on website!
- Get exam conflicts and SSD accommodations sent to us by **Friday**
 - Forms listed on the website

Resources

- Many resources on 370 website
 - <https://eecs370.github.io/#resources>
 - ARmv8 references
- Async discussion recordings

EECS 370: GREEN CARD FOR LEGv8						
Arithmetic Operations	Assembly code			Semantics	Comments	
add	ADD	Xd,	Xn,	Xm	$X5 = X2 + X7$	register-to-register
add & set flags	ADDSD	Xd,	Xn,	Xm	$X5 = X2 + X7$	flags NZVC
add immediate	ADDI	Xd,	Xn,	#uimm12	$X5 = X2 + \#19$	$0 \leq 12$ bit unsigned ≤ 4095
add immediate & set flags	ADDIS	Xd,	Xn,	#uimm12	$X5 = X2 + \#19$	flags NZVC
subtract	SUB	Xd,	Xn,	Xm	$X5 = X2 - X7$	register-to-register
subtract & set flags	SUBS	Xd,	Xn,	Xm	$X5 = X2 - X7$	flags NZVC
subtract immediate	SUBI	Xd,	Xn,	#uimm12	$X5 = X2 - \#20$	$0 \leq 12$ bit unsigned ≤ 4095
subtract immediate & set flags	SUBIS	Xd,	Xn,	Xm	$X5 = X2 - \#20$	flags NZVC
Data Transfer Operations	Assembly code			Semantics	Comments	
load register	LDUR	Xt,	[Xn, #simm9]	$X2 = M[X6, \#18]$	double word load into Xt from Xn + #simm9	
load signed word	LDURSW	Xt,	[Xn, #simm9]	$X2 = M[X6, \#18]$	word load to lower 32b Xt from Xn + #simm9; sign extend upper 32b	
load half	LDURH	Xt,	[Xn, #simm9]	$X2 = M[X6, \#18]$	$\frac{1}{2}$ word load to lower 16b Xt from Xn + #simm9; zero extend upper 48b	
load byte	LDURB	Xt,	[Xn, #simm9]	$X2 = M[X6, \#18]$	byte load to least 8b Xt from Xn + #simm9 zero extend upper 56b	
store register	STUR	Xt,	[Xn, #simm9]	$M[X5, \#12] = X4$	double word store from Xt to Xn + #simm9	
store word	STURW	Xt,	[Xn, #simm9]	$M[X5, \#12] = X4$	word store from lower 32b of Xt to Xn + #simm9	
store half word	STURH	Xt,	[Xn, #simm9]	$M[X5, \#12] = X4$	$\frac{1}{2}$ word load from lower 16b of Xt to Xn + #simm9	
store byte	STURB	Xt,	[Xn, #simm9]	$M[X5, \#12] = X4$	byte load from least 8b of Xt to Xn + #simm9	
	offset	#simm9 = -256 to +255			-256 \leq 9 bits signed immediate \leq +255	
move wide with zero	MOVZ	Xd,	#uimm16,	LSL N	$X9 = 0..0N0..0$	zeros out Xd then place a 16b (#uimm) into the first (N = 0)/second (N = 16)/third (N = 32)/fourth (N = 48) 16b slot of Xd
move wide with keep	MOVK	Xd,	#uimm16,	LSL N	$X9 = x..xNx..x$	place a 16b (#uimm) into the first (N = 0)/second (N = 16)/third (N = 32)/fourth (N = 48) 16b slot of Xd, without changing the other values (x's)
register aliases		$X28 = SP; X29 = FP; X30 = LR; X31 = XZR$				
Logical Operations	Assembly code			Semantics	Using C operations of & ^ < > >	
and	AND	Xd,	Xn,	Xm	$X5 = X2 \& X7$	bit-wise AND
and immediate	ANDI	Xd,	Xn,	#uimm12	$X5 = X2 \& \#19$	bit-wise AND with 0 ≤ 12 bit unsigned ≤ 4095
inclusive or	ORR	Xd,	Xn,	Xm	$X5 = X2 X7$	bit-wise OR
inclusive or immediate	ORRI	Xd,	Xn,	#uimm12	$X5 = X2 \#11$	bit-wise OR with 0 ≤ 12 bit unsigned ≤ 4095
exclusive or	EOR	Xd,	Xn,	Xm	$X5 = X2 \wedge X7$	bit-wise EOR
exclusive or immediate	EORI	Xd,	Xn,	#uimm12	$X5 = X2 \wedge \#57$	bit-wise EOR with 0 ≤ 12 bit unsigned ≤ 4095
logical shift left	LSL	Xd,	Xn,	#uimm6	$X1 = X2 << \#10$	shift left by a constant ≤ 63
logical shift right	LSR	Xd,	Xn,	#uimm6	$X5 = X3 >> \#20$	shift right by a constant ≤ 63
Unconditional branches	Assembly code			Semantics	Also known as Jumps	
branch	B	#simm26	goto PC + #1200		PC relative branch PC + 26b offset; $-2^{25} \leq \#simm26 \leq 2^{25}-1$; 4b instruction	
branch to register	BR	Xt	target in Xt		Xt contains a full 64b address	
branch with link	BL	#simm26	$X30 = PC + 4; PC = \#11000$		PC relative branch to PC + 26b offset; 16 million instructions; X30 = LR contains return from subroutine address	

Instruction Set Architecture (ISA) Design Lectures

- Lecture 2: ISA - storage types, binary and addressing modes
- Lecture 3 : LC2K
- Lecture 4 : ARM
- **Lecture 5 : Converting C to assembly – basic blocks**
- Lecture 6 : Converting C to assembly – functions
- Lecture 7 : Translation software; libraries, memory layout





le durb

Load Instruction Sizes

How much data is retrieved from memory at the given address?

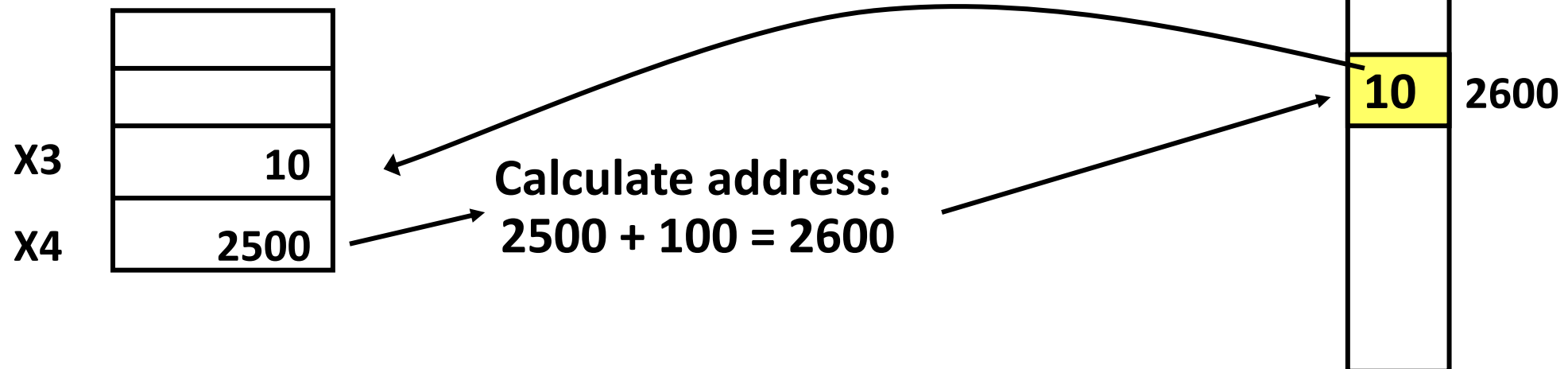
Desired amount of data to transfer?	Operation	Unused bits in register?	Example
64-bits (double word or whole register)	LDUR (Load unscaled to register)	N/A	0x FEDC_BA98_7654_3210
16-bits (half-word) into lower bits of reg	LDURH	Set to zero	0x0000_0000_0000_ 3210
8-bits (byte) into lower bits of reg	LDURB	Set to zero	0x0000_0000_0000_00 10
32-bits (word) into lower bits of reg	LDURSW (load signed word)	Sign extend (0 or 1 based on most significant bit of transferred word)	0x0000_0000_ 7654_3210 or 0xFFFF_FFFF_ F654_3210 (depends on bit 31)

Load Instruction in Action

```
struct {  
    int arr[25];  
    unsigned char c;  
} my_struct;
```

```
int func() {  
    my_struct.c++;  
    // load value from mem into reg  
    // then increment it  
}
```

LDURB X3, [X4, #100]

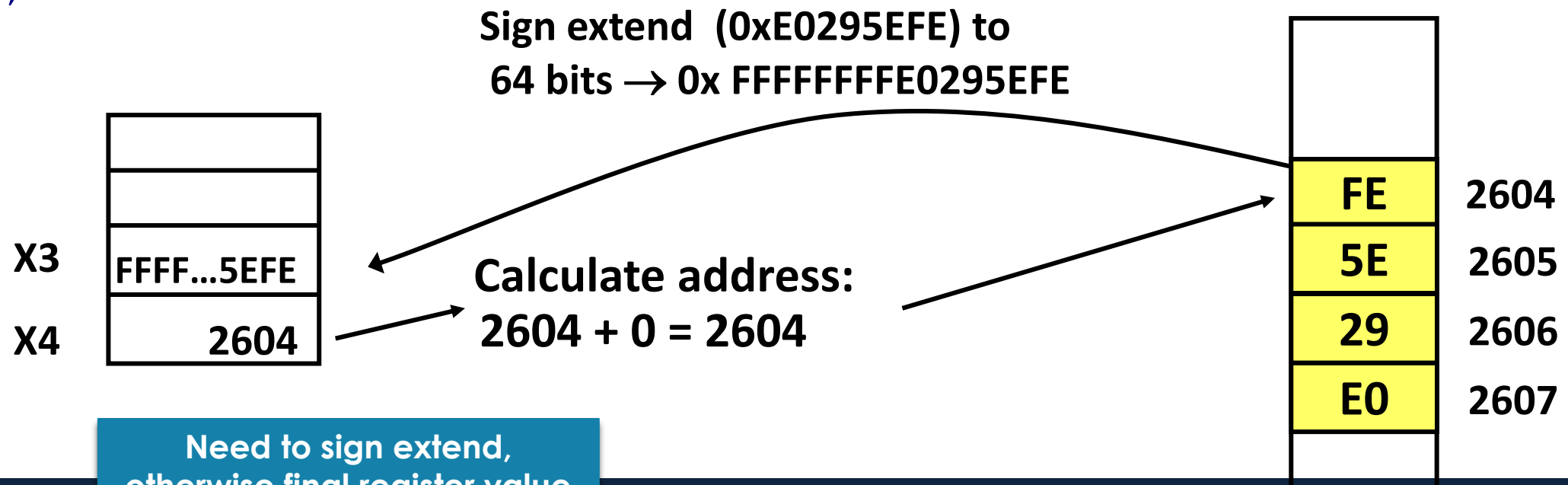


Load Instruction in Action – other example

```
int my_big_number = -534159618; // 0xE0295EFE in 2's complement
```

```
int inc_number() {
    my_big_number++;
    // load value from mem into reg
    // then increment it
};
```

LDURSW X3, [X4, #0]



Need to sign extend,
otherwise final register value
will be positive!!!

But wait...

```
int my_big_number = -534159618; // 0xE0295EFE in 2's complement
```

- If I want to store this number in memory... should it be stored like this?

FE	2604
5E	2605
29	2606
E0	2607

- ... or like this?

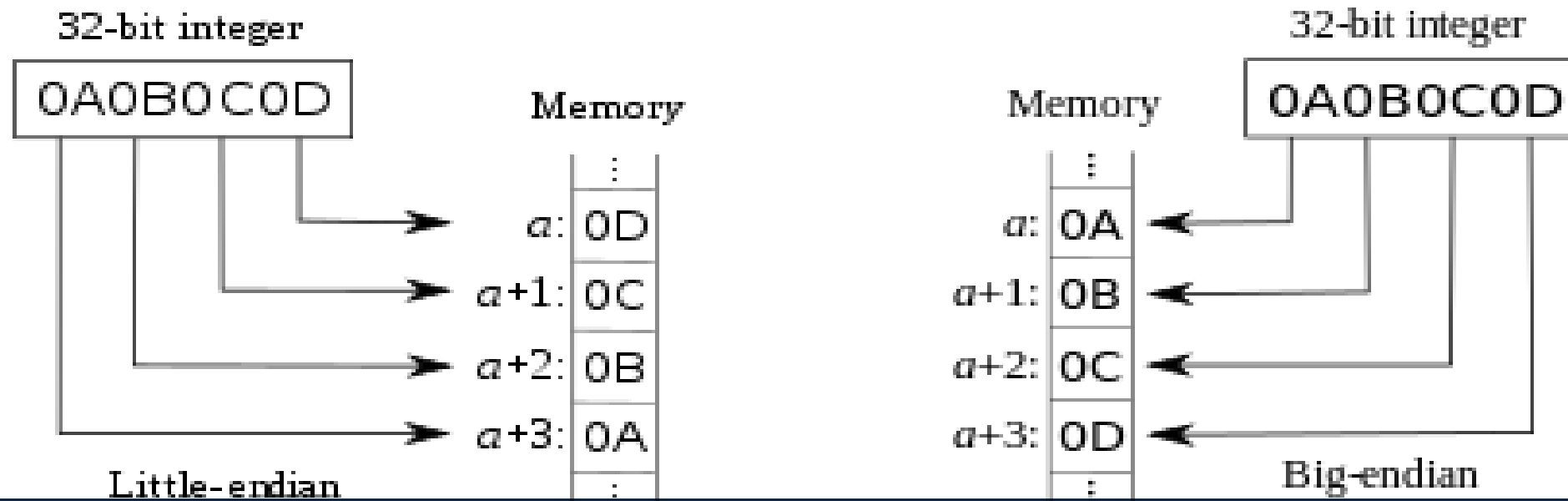
E0	2604
29	2605
58	2606
FE	2607

Poll: Which do you prefer?

- a) Big Endian
- b) Little Endian

Big Endian vs. Little Endian

- Endian-ness: ordering of bytes within a word
 - Little – Bigger address holds more significant bits
 - Big – Opposite, smaller address hold more significant bits
 - The Internet is big endian, x86 is little endian, LEG and ARMv8 can switch
 - But in general assume little endian. (Figures from Wikipedia)



Store Instructions

- Store instructions are simpler—there is no sign/zero extension to consider (do you see why?)

Desired amount of data to transfer?	Operation	Example
64-bits (double word or whole register)	STUR (Store unscaled register)	0xFEDC_BA98_7654_3210
16-bits (half-word) from lower bits of reg	STURH	0x0000_0000_0000_3210
8-bits (byte) from lower bits of reg	STURB	0x0000_0000_0000_0010
32-bits (word) from lower bits of reg	STURW	0xFFFF_FFFF_F654_3210

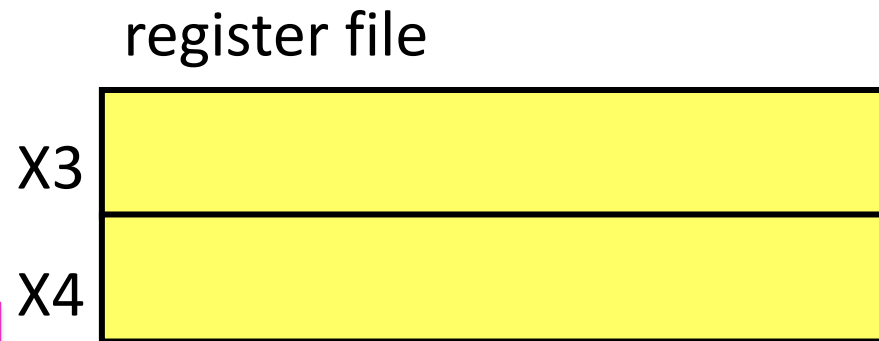
Example Code Sequence

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR    X4, [X5, #100]
LDURB   X3, [X5, #102]
STUR     X3, [X5, #100]
STURB    X4, [X5, #102]
```

Poll: Final contents of registers?

- a) 0x11..FF : 0xE5..02
- b) 0x00..FF : 0x02..E5
- c) 0x11..FF : 0x02..E5
- d) 0x00..FF : 0xE5..02



Memory
(each location is 1 byte)

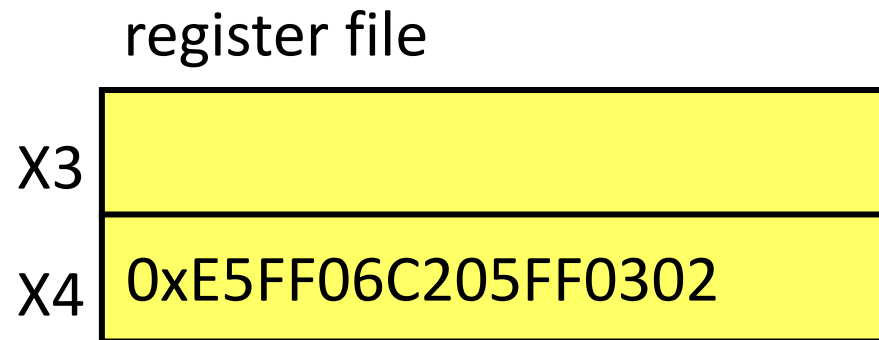
little endian

0x02	100
0x03	101
0xFF	102
0x05	103
0xC2	104
0x06	105
0xFF	106
0xE5	107

Example Code Sequence

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR    X4, [X5, #100]
LDURB   X3, [X5, #102]
STUR    X3, [X5, #100]
STURB   X4, [X5, #102]
```



Memory
(each location is 1 byte)

little endian	
0x02	100
0x03	101
0xFF	102
0x05	103
0xC2	104
0x06	105
0xFF	106
0xE5	107

Example Code Sequence

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR    X4, [X5, #100]
LDURB   X3, [X5, #102]
STUR     X3, [X5, #100]
STURB    X4, [X5, #102]
```

register file	
X3	0x0000000000000000FF
X4	0xE5FF06C205FF0302

Memory
(each location is 1 byte)

little endian	
0x02	100
0x03	101
0xFF	102
0x05	103
0xC2	104
0x06	105
0xFF	106
0xE5	107

Example Code Sequence

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR    X4, [X5, #100]
LDURB   X3, [X5, #102]
STUR    X3, [X5, #100]
STURB   X4, [X5, #102]
```

register file

X3	0x0000000000000000FF
X4	0xE5FF06C205FF0302

Memory
(each location is 1 byte)

little endian

0xFF	100
0x00	101
0x00	102
0x00	103
0x00	104
0x00	105
0x00	106
0x00	107

Example Code Sequence

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR    X4, [X5, #100]
LDURB   X3, [X5, #102]
STUR     X3, [X5, #100]
STURB   X4, [X5, #102]
```

register file	
X3	0x0000000000000000FF
X4	0xE5FF06C205FF0302

Memory
(each location is 1 byte)

little endian	
0xFF	100
0x00	101
0x02	102
0x00	103
0x00	104
0x00	105
0x00	106
0x00	107

Example Code Sequence (2)

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR    X4, [X5, #100]
LDURB   X3, [X5, #102]
STURB   X3, [X5, #103]
LDURSW  X4, [X5, #100]
```

We shown the registers as blank. What do they actually contain before we run the snippet of code?

register file



Memory
(each location is 1 byte)

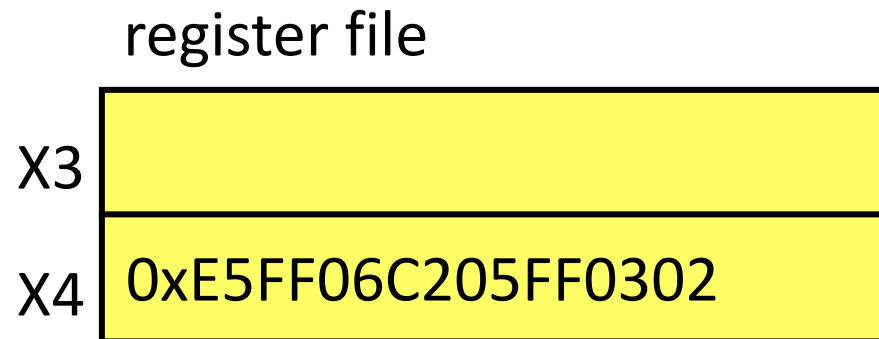
little endian

	100
0x02	101
0x03	102
0xFF	103
0x05	104
0xC2	105
0x06	106
0xFF	107
0xE5	

Example Code Sequence (2)

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR      X4, [X5, #100]  
LDURB     X3, [X5, #102]  
STURB     X3, [X5, #103]  
LDURSW    X4, [X5, #100]
```



Memory
(each location is 1 byte)

little endian	
0x02	100
0x03	101
0xFF	102
0x05	103
0xC2	104
0x06	105
0xFF	106
0xE5	107

Example Code Sequence (2)

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR    X4, [X5, #100]
LDURB   X3, [X5, #102]
STURB   X3, [X5, #103]
LDURSW  X4, [X5, #100]
```

register file	
X3	0x0000000000000000FF
X4	0xE5FF06C205FF0302

Memory
(each location is 1 byte)

little endian	
0x02	100
0x03	101
0xFF	102
0x05	103
0xC2	104
0x06	105
0xFF	106
0xE5	107

Example Code Sequence (2)

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR    X4, [X5, #100]
LDURB   X3, [X5, #102]
STURB   X3, [X5, #103]
LDURSW  X4, [X5, #100]
```

register file

X3	0x0000000000000000FF
X4	0xE5FF06C205FF0302

Memory
(each location is 1 byte)

little endian

0x02	100
0x03	101
0xFF	102
0xFF	103
0xC2	104
0x06	105
0xFF	106
0xE5	107

Example Code Sequence (2)

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR    X4, [X5, #100]
LDURB   X3, [X5, #102]
STURB   X3, [X5, #103]
LDURSW  X4, [X5, #100]
```

register file	
X3	0x0000000000000000FF
X4	0xFFFFFFFFFFFFFFFF0302

Memory
(each location is 1 byte)

little endian	
0x02	100
0x03	101
0xFF	102
0xFF	103
0xC2	104
0x06	105
0xFF	106
0xE5	107

Converting C to assembly – example 2

- Write ARM assembly code for the following C expression:
 - (assume an int is 4 bytes and that struct elements are stored contiguously)

```
struct { int a; unsigned char b, c; } y;  
y.a = y.b + y.c;
```

- Assume that a pointer to **y** is in X1.

Converting C to assembly – example 2

- Write ARM assembly code for the following C expression:
 - (assume an int is 4 bytes and that struct elements are stored contiguously)

```
struct { int a; unsigned char b, c; } y;  
y.a = y.b + y.c;
```

- Assume that a pointer to **y** is in X1.

```
LDURB X2, [X1, #4] // load y.b  
LDURB X3, [X1, #5] // load y.c  
ADD X4, X2, X3 // calculate y.b+y.c  
STURW X4, [X1, #0] // store y.a
```

Calculating Load/Store Addresses for Variables

Datatype	size (bytes)
short	2
char	1
int	4
double	8

```
short  a[100];  
char   b;  
int    c;  
double d;  
short  e;  
struct {  
    char f;  
    int  g[1];  
    char h;  
} i;
```

- *Problem:* Assume data memory starts at address 100, calculate the total amount of memory needed

$$a = 2 \text{ bytes} * 100 = 200$$

$$b = 1 \text{ byte}$$

$$c = 4 \text{ bytes}$$

$$d = 8 \text{ bytes}$$

$$e = 2 \text{ bytes}$$

$$i = 1 + 4 + 1 = 6 \text{ bytes}$$

total = 221, right or wrong?

Memory layout of variables

- Compilers don't like variables placed in memory arbitrarily
- As we'll see later in the course, memory is divided into fixed sized **chunks**
 - When we load from a particular chunk, we really read the whole chunk
 - Usually an integer number of words (32 bits)
- If we read a single char (1 byte), it doesn't matter where it's placed

0x1000	0x1001	0x1002	0x1003
'a'	'b'	'c'	'd'

ldurb [x0, 0x1002]

- Reads [0x1000-0x1003], then throws away all but 0x1002, **fine**

Memory layout of variables

- BUT, if we read a 32-bit integer word, and that word starts at 0x1002:

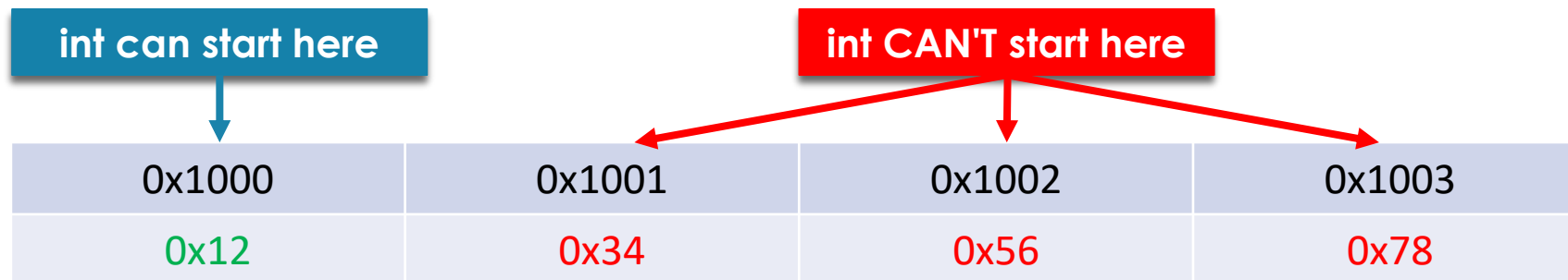
0x1000	0x1001	0x1002	0x1003
0xFF	0xFF	0x12	0x34
0x1004	0x1005	0x1006	0x1007
0x56	0x78	0xFF	0xFF

- First we need to read [0x1000-0x1003], throw away 0x1000 and 0x1001, **then** read [0x1004-0x1007]
- Need to read from memory twice! Slow! Complicated! **Bad!**

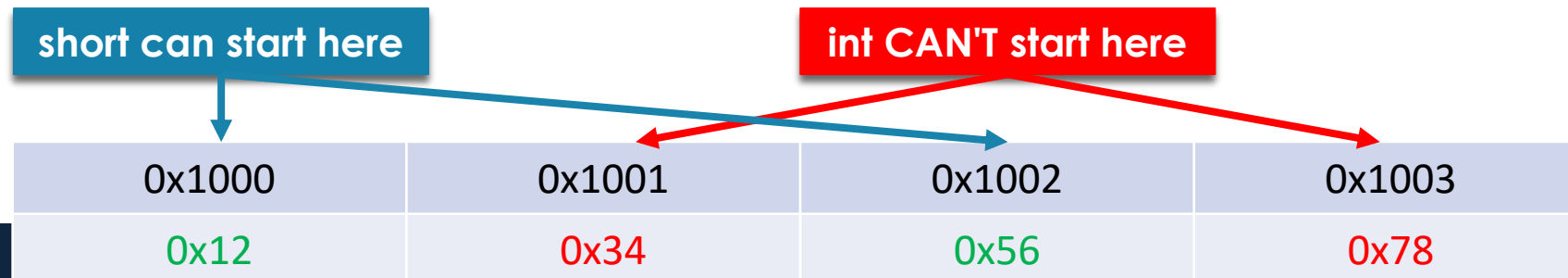
Solution: Memory Alignment

Poll: Where can chars start?

- Most modern ISAs require that data be aligned
 - An N-byte variable must start at an address A, such that $(A\%N) == 0$
- For example, starting address of an **int** must be divisible by 4



- Starting address of a **short** must be divisible by 2



Golden Rule of Alignment

```
char  c;  
short s;  
int   i;
```

- Every (primitive) object starts at an address divisible by its size
- "Padding" is placed in between objects if needed

0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007
[c]	[padding]	[s]		[i]			


- But what about non-primitive data types?
 - Arrays? Treat as independent objects
 - Structs? Trickier...

Problem with Structs

- If we align each element of a struct according to the Golden Rule, we can still run into issues
 - E.g.: An array of structs

```
char c;  
  
struct {  
    char c;  
    int i;  
} s[2];
```

Amount of padding
is different across
different instances



1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	100A	100B	100C	100D	100E	100F
c	s[0].c	[pad]	[pad]	s[0].i				s[1].c	[pad]	[pad]	[pad]	s[1].i			

- Why is this bad?
- It makes "for" loops very difficult to write!
 - Offsets need to be different on each iteration

Structure Alignment

- Solution: in addition to laying out each field according to Golden Rule...
 - Identify largest (primitive) field
 - Starting address of overall struct is aligned based on the largest field
 - Padded in the back so total size is a multiple of the largest primitive

```
char c;  
  
struct {  
    char c;  
    int i;  
} s[2];
```

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	100A	100B	100C	100D	100E	100F
c	[pad]	[pad]	[pad]	s[0].c	[pad]	[pad]	[pad]	s[0].i				s[1].c	[pad]	[pad]	[pad]

Guaranteed to lay
out each instance
identically

Structure Example

```
struct {  
    char w;  
    int x[3]  
    char y;  
    short z;  
}
```

Poll: What boundary should this struct be aligned to?

- a) 1 byte
- b) 4 bytes
- c) 12 bytes
- d) 2 bytes
- e) 19 bytes

- Assume struct starts at location 1000,
 - char w → 1000
 - x[0] → 1004-1007, x[1] → 1008 – 1011, x[2] → 1012 – 1015
 - char y → 1016
 - short z → 1018 – 1019

Total size = 20 bytes!

Calculating Load/Store Addresses for Variables

Datatype	size (bytes)
short	2
char	1
int	4
double	8

```
short a[100];  
char b;  
int c;  
double d;  
short e;  
struct {  
    char f;  
    int g[1];  
    char h;  
} i;
```

- *Problem:* Assume data memory starts at address 100, calculate the total amount of memory needed

a = 200 bytes (100-299)

b = 1 byte (300-300)

c = 4 bytes (304-307)

d = 8 bytes (312-319)

e = 2 bytes (320-321)

struct: largest field is 4 bytes, start at 324

f = 1 byte (324-324)

g = 4 bytes (328-331)

h = 1 byte (332-332)

i = 12 bytes (324-335)

236 bytes total!! (compared to 221, originally)

Class Problem

- How much memory is required for the following data, assuming that the data starts at address 200 and is a 32 bit address space?

```
int a;  
struct {double b, char c, int d} e;  
char* f;  
short g[20];
```

Poll: How much memory?

- a) $x < 40$ bytes
- b) $40 < x < 50$ bytes
- c) $50 < x < 60$ bytes
- d) $60 < x$ bytes

Data Layout – Why?

- Does gcc (or another compiler) reorder variables in memory to avoid padding?
- No, C99 forbids this
 - Memory is laid out in order of declaration for structs
- The programmer (i.e., you) are expected to manage data layout of variables for your program and structs.
- Two optimal strategies:
 - Order fields in struct by datatype size, smallest first
 - Or by largest first

ARM/LEGv8 Sequencing Instructions

- Sequencing instructions change the flow of instructions that are executed
 - This is achieved by modifying the program counter (PC)
- Unconditional branches are the most straightforward they ALWAYS change the PC and thus “jump” to another instruction out of the usual sequence
- Conditional branches

If (condition_test) goto target_address

condition_test examines the four flags from the processor status word (SPSR)

target_address is a 19 bit signed word displacement on current PC

LEGv8 Conditional Instructions

- Two varieties of conditional branches
 1. One type compares a register to see if it is equal to zero.
 2. Another type checks the condition codes set in the status register.

Conditional branch	compare and branch on equal 0	CBZ X1, 25	if (X1 == 0) go to PC + 100	Equal 0 test; PC-relative branch
	compare and branch on not equal 0	CBNZ X1, 25	if (X1 != 0) go to PC + 100	Not equal 0 test; PC-relative branch
	branch conditionally	B.cond 25	if (condition true) go to PC + 100	Test condition codes; if true, branch

- Let's look at the first type: CBZ and CBNZ
 - CBZ: Conditional Branch if Zero
 - CBNZ: Conditional Branch if Not Zero

LEGv8 Conditional Instructions

- CBZ/CBNZ: test a register against zero and branch to a PC relative address
 - The relative address is a 19 bit signed integer—the number of instructions. Recall instructions are 32 bits of 4 bytes

Conditional branch	compare and branch on equal 0	CBZ X1, 25	if (X1 == 0) go to PC + 100	Equal 0 test; PC-relative branch
	compare and branch on not equal 0	CBNZ X1, 25	if (X1 != 0) go to PC + 100	Not equal 0 test; PC-relative branch
	branch conditionally	B.cond 25	if (condition true) go to PC + 100	Test condition codes; if true, branch

- Example: CBNZ X3, Again
 - If X3 doesn't equal 0, then branch to label "Again"
 - "Again" is an offset from the PC of the current instruction (CBNZ)
 - Why does "25" in the above table result in PC + 100?

LEGv8 Conditional Instructions

- Example: What would the offset or displacement be if there were two instructions between ADDI and CBNZ?

Again: ADDI X3, X3, #-1

CBNZ X3, Again

Poll: What's the offset?

- a) -16
- b) -12
- c) -4
- d) -3
- e) 0

Next Time

- More C-to-Assembly
 - Function calls
- Lingering questions / feedback? I'll include an anonymous form at the end of every lecture: <https://bit.ly/3oXr4Ah>

