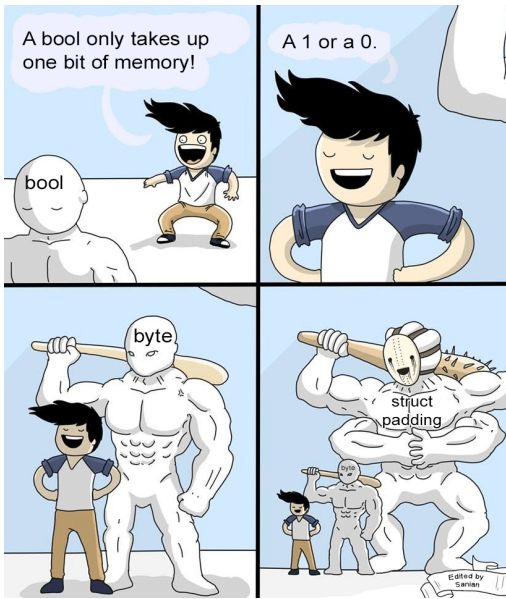


EECS 370 - Lecture 5

C to Assembly



Announcements

- P1a due Thursday
- Pre-lab 3 quiz due Thursday on Gradescope
- Reminder lab assignments must be completed in your assigned lab
 - Group assignments sent out later this week
 - Fill out admin form if you are sick / have a conflict
- HW 1 will be posted in the next day, due in two weeks

Agenda

- ARM overview and basic instructions
- **Memory instructions**
 - Handling multiple data widths
- Sample Problems

Word vs Byte Addressing

- A **word** is a collection of bytes
 - The “natural unit” of data that’s processed - depends on architecture
 - in LC2K and ARM, 4 bytes
 - **Double word** is 8 bytes
- LC2K is **word addressable**
 - Each **address** refers to a particular **word** in memory
 - Wanna move forward one int? Increment address by **one**
 - Wanna move forward one char? Uhhh...
- ARM (and most modern ISAs) is **byte addressable**
 - Each **address** refers to a particular **byte** in memory
 - Wanna move forward one int? Increment address by **four**
 - Wanna move forward one char? Increment address by **one**



LEGv8 Memory Instructions

- Like LC2K, employs base + displacement addressing mode
 - Base is a register
 - Displacement is 9-bit immediate ± 256 bytes—sign extended to 64 bits
- Unlike LC2K (which always transfers 4 bytes), we have several options in LEGv8

Category	Instruction Example	Meaning	Comments
	load register	LDUR X1, [X2,40]	$X1 = \text{Memory}[X2 + 40]$ Doubleword from memory to register
	store register	STUR X1, [X2,40]	$\text{Memory}[X2 + 40] = X1$ Doubleword from register to memory
	load signed word	LDURSW X1, [X2,40]	$X1 = \text{Memory}[X2 + 40]$ Word from memory to register
	store word	STURW X1, [X2,40]	$\text{Memory}[X2 + 40] = X1$ Word from register to memory
	load half	LDURH X1, [X2,40]	$X1 = \text{Memory}[X2 + 40]$ Halfword memory to register
	store half	STURH X1, [X2,40]	$\text{Memory}[X2 + 40] = X1$ Halfword register to memory
	load byte	LDURB X1, [X2,40]	$X1 = \text{Memory}[X2 + 40]$ Byte from memory to register
	store byte	STURB X1, [X2,40]	$\text{Memory}[X2 + 40] = X1$ Byte from register to memory

D-Instruction fields

- **D**ata transfer
- opcode and op2 define data transfer operation
- address is the ± 256 bytes displacement
- Rn is the base register
- Rt is the destination (loads) or source (stores)
- More complicated modes are available in full ARMv8

Look over formatting
on your own

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

Agenda

- ARM overview and basic instructions
- Memory instructions
 - **Handling multiple data widths**
- Sample Problems

LEGv8 Memory Instructions

- Registers are 64 bits wide
- But sometimes we want to deal with non-64-bit entities
 - E.g. ints (32 bits), chars (8 bits)
- When we load smaller elements from memory, what do we set the other bits to?

- Option A: set to zero



- Option B: sign extend



- We'll need different instructions for different options



Load Instruction Sizes

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

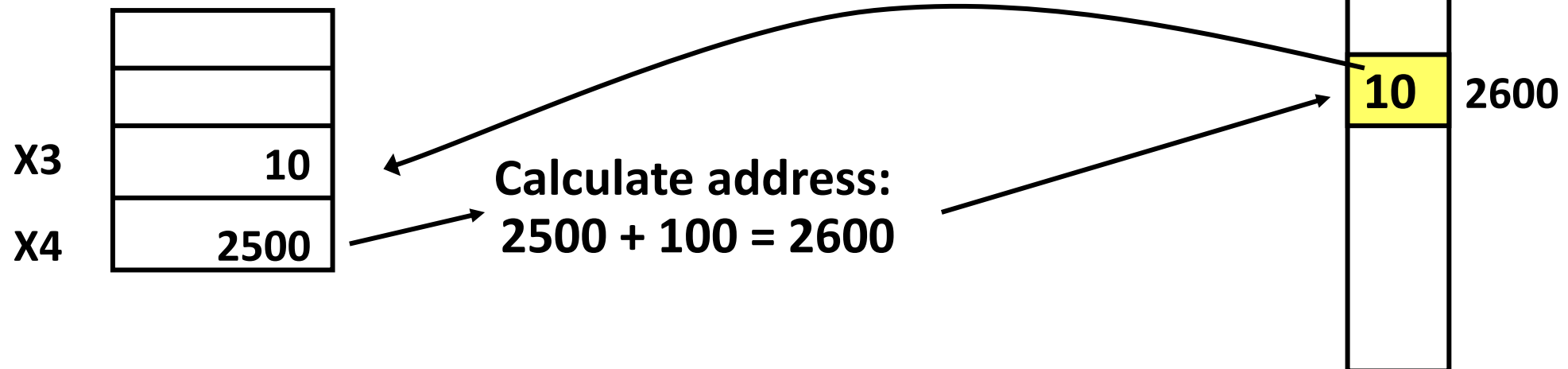
Desired amount of data to transfer?	Operation	Unused bits in register?	Example
64-bits (double word or whole register)	LDUR (Load unscaled to register)	N/A	0xFEDC_BA98_7654_3210
16-bits (half-word) into lower bits of reg	LDURH	Set to zero	0x0000_0000_0000_3210
8-bits (byte) into lower bits of reg	LDURB	Set to zero	0x0000_0000_0000_0010
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]  
ADD    X3, X3, #1  
STURB  X3, [X4, #100]
```

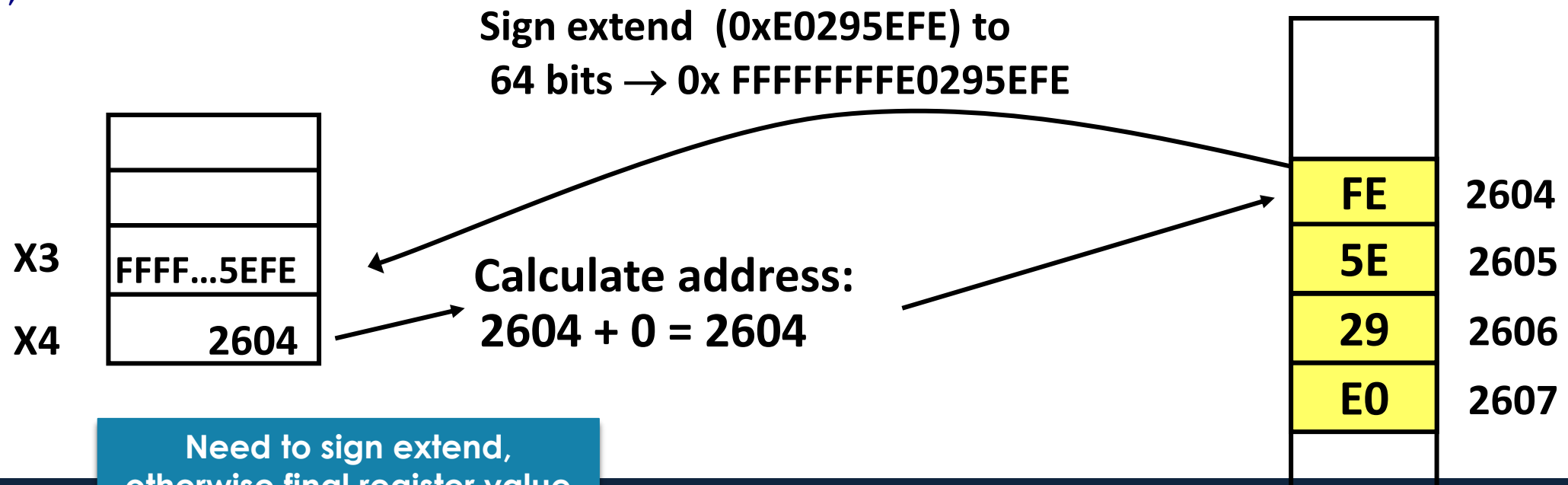


Load Instruction in Action – other example

```
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]
ADD     X3, X3, #1
STURW   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?

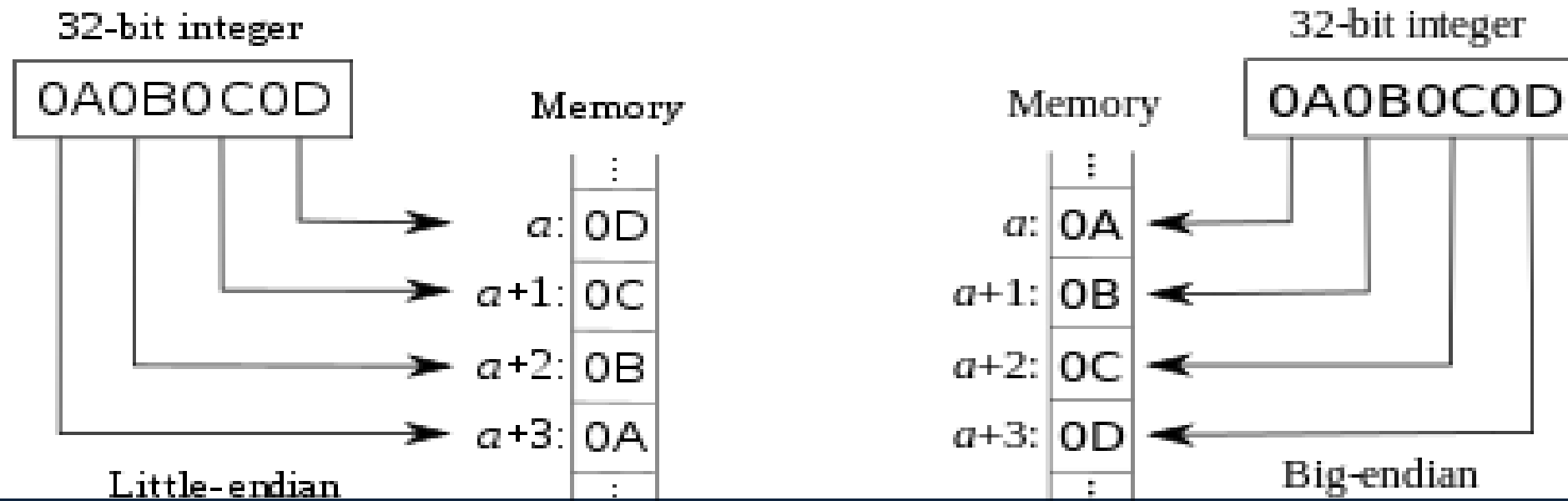
FE	2604
5E	2605
29	2606
E0	2607

- ... or like this?

E0	2604
29	2605
58	2606
FE	2607

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

Agenda

- ARM overview and basic instructions
- Memory instructions
 - Handling multiple data widths
- **Sample Problems**

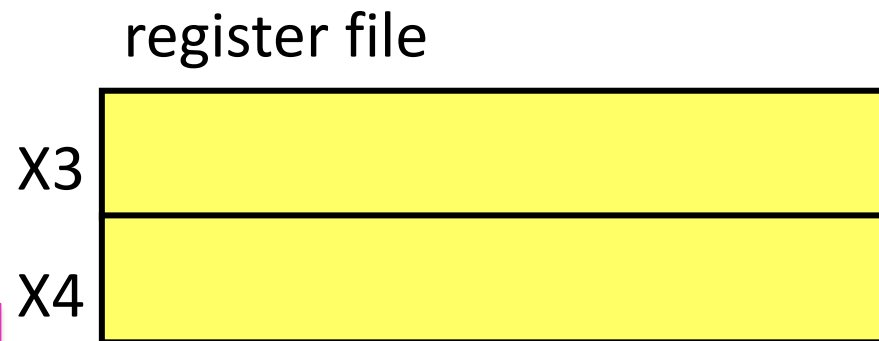
Example Code Sequence

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)

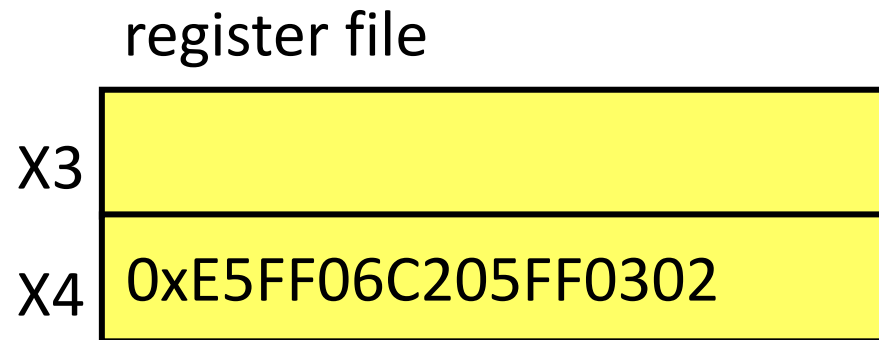
little endian

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)

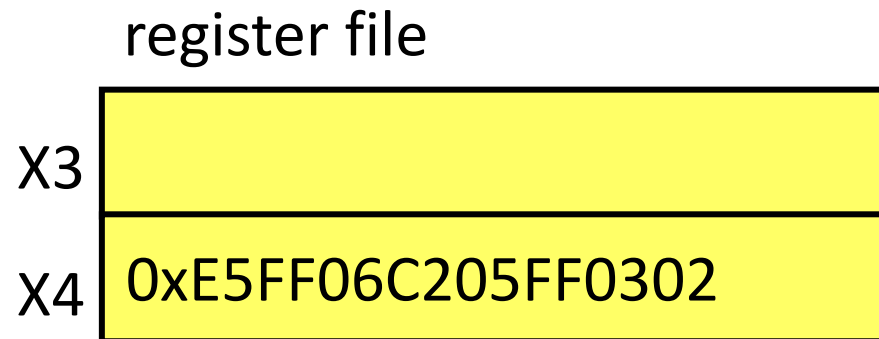
little endian

	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

Agenda

- **Memory alignment**
 - Aligning Structs
- Control flow instructions
 - C-code examples
- Extra Problems

Calculating Load/Store Addresses for Variables

Datatype	size (bytes)
char	1
short	2
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 and no reordering, calculate the total amount of memory needed

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

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

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

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

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

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

$$\text{total} = 221, \text{ right or wrong?}$$

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:

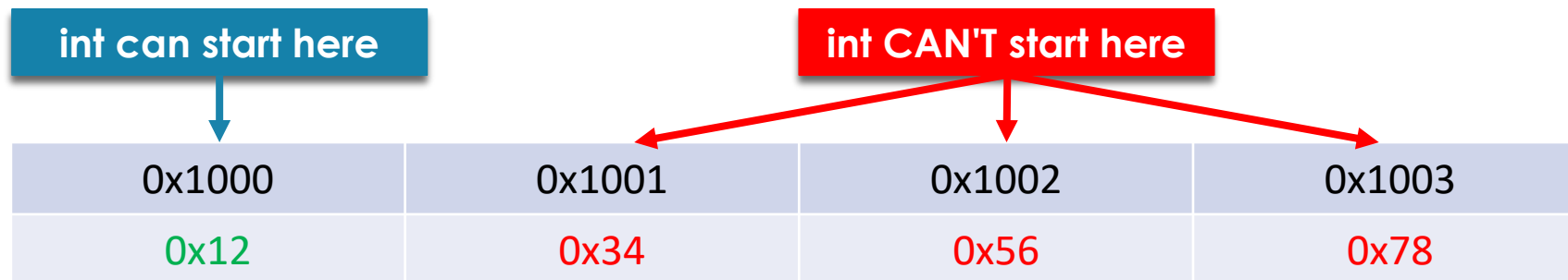
0x1000	0x1001	0x1002	0x1003
0xFF	0xFF	0x12	0x34
0x1004	0x1005	0x1006	0x1007
0x56	0x78	0xFF	0xFF

- 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!**

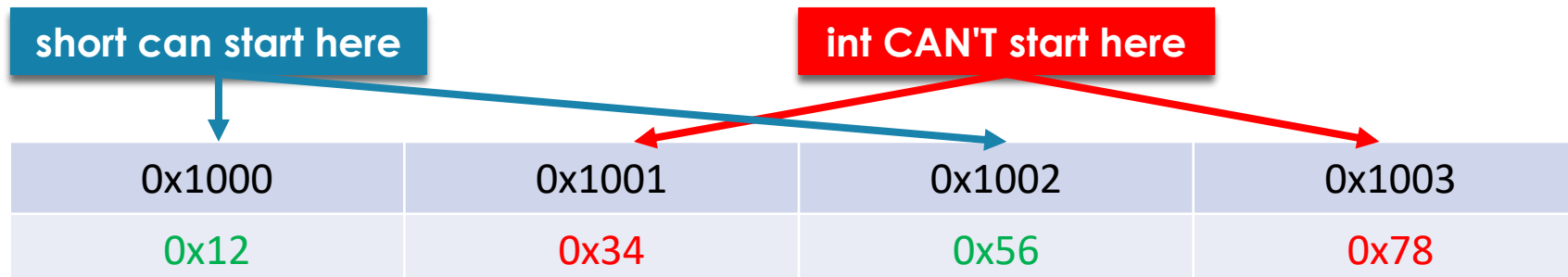
Solution: Memory Alignment

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 a 32 bit **int** must be divisible by 4



- Starting address of a 16 bit **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...

Agenda


- Memory alignment
 - **Aligning Structs**
- Control flow instructions
 - C-code examples
- Extra Problems

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 (potentially nested) **primitive** component
 - Starting address of overall struct is aligned based on the largest component
 - Padded in the back so total size is a multiple of the largest component

```
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)
char	1
short	2
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 and no reordering, 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)

Data Layout – Why?

- Does gcc (or another compiler) reorder variables in memory to avoid padding?
- Only outside structs
- C99 forbids reordering elements inside a struct
- 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

Next Time

- More C-to-Assembly
 - Function calls
- Lingering questions / feedback? Post it to Slido