

EECS 370 - Lecture 4

ARM

byte be like




Announcements

- Project 1
 - P1a due Thursday **1/29**
 - P1s due Thursday **2/5**
 - P1m due Thursday **2/5**
- HW1
 - Posted, due Monday **2/2**
- Lab
 - **Lab quiz due Thursday 1/22**
 - Lab meets this upcoming Friday/Monday/Tuesday
- Midterm:
 - Thursday March 12th, 7-9pm
 - *Exam conflict form is due by January 23rd via the [Exam Conflict form](#) ([ed post #17](#))*



Lecture 3 covered what you need.



Can start now, each problem is labelled by what lecture you need.

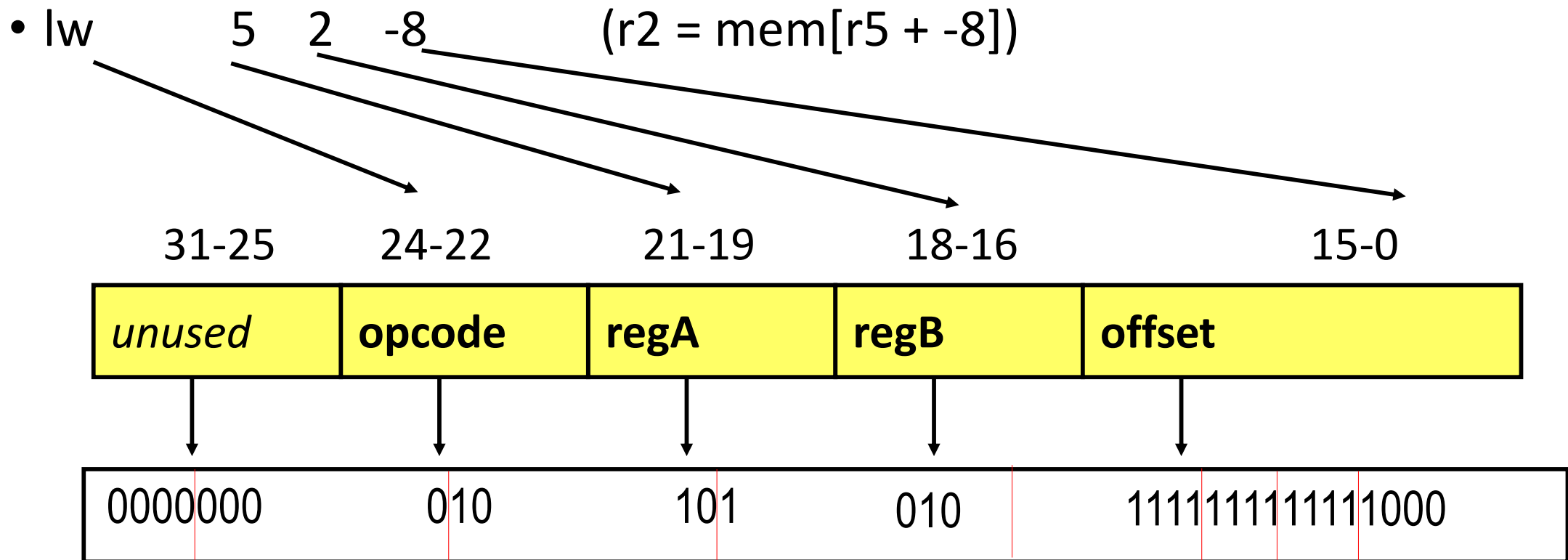
Resources

- Many resources on 370 website
 - <https://eecs370.github.io/#resources>
 - LEGv8 references

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	ADDS	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 of 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 of Xt from Xn + #simm9; zero extend upper 48b	
load byte	LDURB	Xt,	[Xn, #simm9]	$X2 = M[X6, \#18]$	byte load to least 8b of 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 $0 \leq 12$ 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 $0 \leq 12$ 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 $0 \leq 12$ 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	



Example Encoding - lw



Convert to Hex → 0x00AAFF8

Convert to Dec → 11206648

Note that we "bit-extend"
1 for negative numbers

Another way to think about the assembler

- Each line of assembly code corresponds to a number
 - “add 0 0 0” is just 0.
 - “lw 5 2 -8” is 11206648
- We only write in assembly because it’s easier to read and write

.fill

```
int GLOBAL = 7;
```

How can we
hardcode this 7 in
memory?

- I also might want a number, to be, well, a number.
 - Maybe I want the number 7 as a data element I can use.
- .fill tells the assembler to put a number instead of an instruction
- The syntax is just “.fill 7”.
- Question:
 - What do “.fill 7” and “add 0 0 7” have in common?

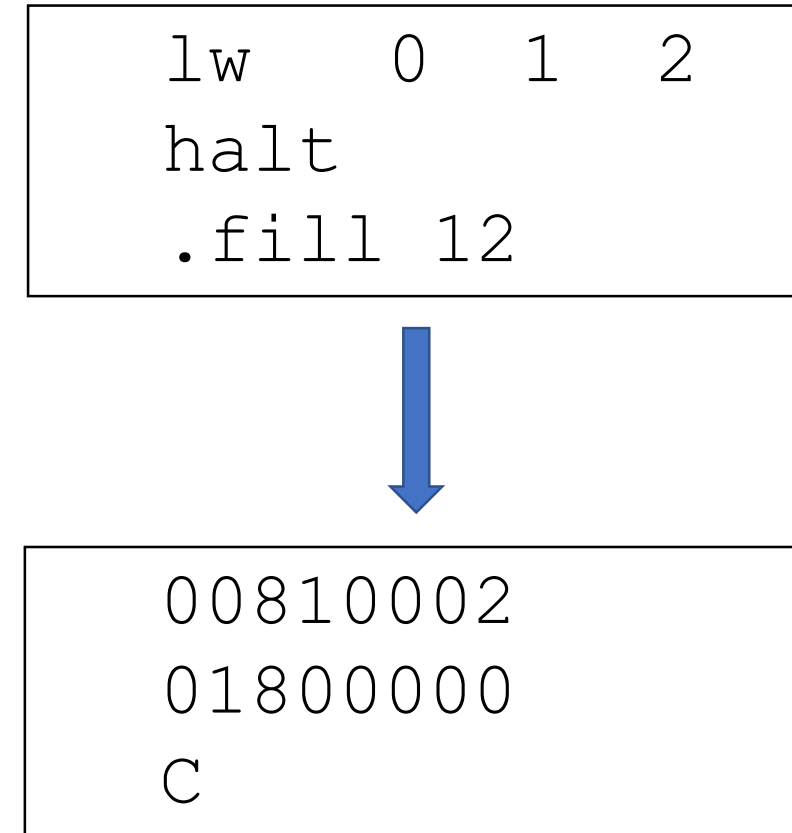
.fill



- .fill is NOT an instruction
- It does not have a corresponding opcode
- It should be used to initialize data in your program
 - If your PC ever points to it, something has probably gone wrong
- But if the PC **DOES** point to it, it will treat it as whatever type of instruction encodes to that number

.fill with lw / sw

- We most often use .fill along with lw or sw
- Remember: every line in an assembly program corresponds to an address in memory
 - When an instruction is to be executed, that address is sent to memory
- ".fill 12" is address 2, meaning mem[2]=12
- "lw 0 1 2" loads the contents of mem[2] into register 1



Labels in LC2K

- The code on the right is awkward
 - Need to count lines to see what it's doing
- **Labels** make code easier to read/write
- Label **definition**: listed to the **left** of the opcode
 - Can be defined on any line (only once)
- Label **use**: replaces offset value in lw/sw/beq instructions (any number)
- For lw/sw, assembler will replace label use with the line number of definition
 - In this example, data is on line 2

```
lw    0    1    2
halt
.fill 71
```

```
lw    0    1    data
halt
data .fill 71
```

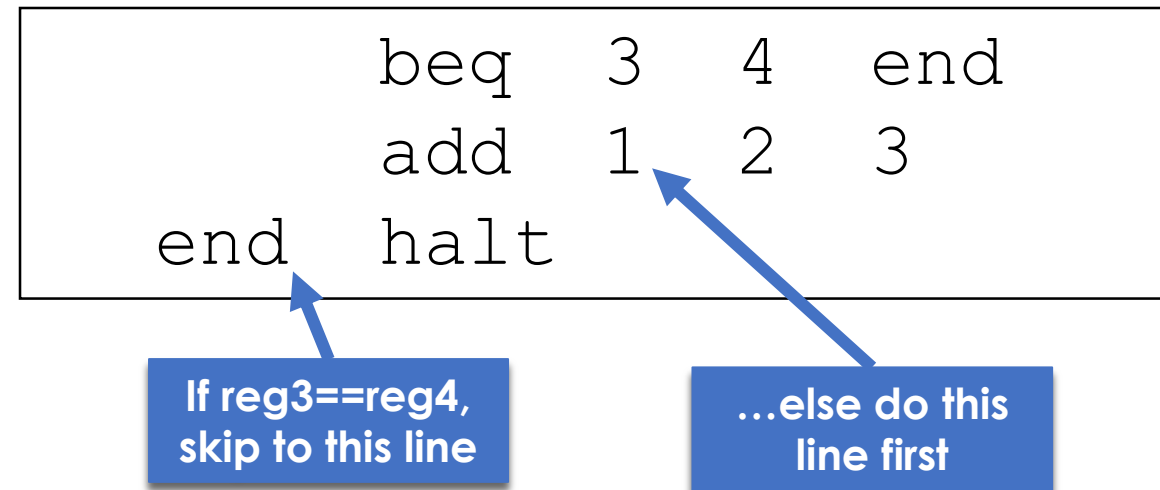
Definition of data label

Use of data label

Labels in LC2K - beq

- Labels with beq indicate **where** we should branch
- Assembler's job is a little trickier
 - Doesn't just replace it with line number
 - Remember: target address is $PC+1+offset$

beq	3	4	1
add	1	2	3
halt			



Exercise

```
// this is the assembly for:  
while(x != y) {  
    x *= 2;  
}
```

- What are the values of the labels here?

loop	beq	3	4	end
	add	3	3	3
	beq	0	0	loop
end	halt			

What are the labels replaced with?

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



Agenda

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

ARMv8 ISA

- LC2K is intended to be an extremely bare-bones ISA
 - "Bare minimum"
 - Easy to design hardware for, really annoying to program on (as you'll see in P1m)
 - Invented for our projects, not used anywhere in practice
- ARM (specifically v8) is a much more powerful ISA
 - Used heavily in practice (most smartphones, some laptops & supercomputers)
 - Subset (LEG) is focus of hw and lecture



ISA Types

Reduced Instruction Set Computing (RISC)

- Fewer, simpler instructions
- Encoding of instructions are usually the same size
- Simpler hardware
- Program is larger, more tedious to write by hand
- E.g. LC2K, RISC-V, ARM (kinda)
- More popular now

Complex Instruction Set Computing (CISC)

- More, complex instructions
- Encoding of instructions are different sizes
- More complex hardware
- Short, expressive programs, easier to write by hand
- E.g. x86
- Less popular now

ARM vs LC2K at a Glance

	LC2K	LEG
# registers	8	32
Zero-valued register	0	X31 (i.e. XZR)
Register width	32 bits	64 bits
Memory size	2^{18} bytes	2^{64} bytes
# instructions	8	40-ish
Addressability	Word	Byte

We'll discuss what this means in a bit



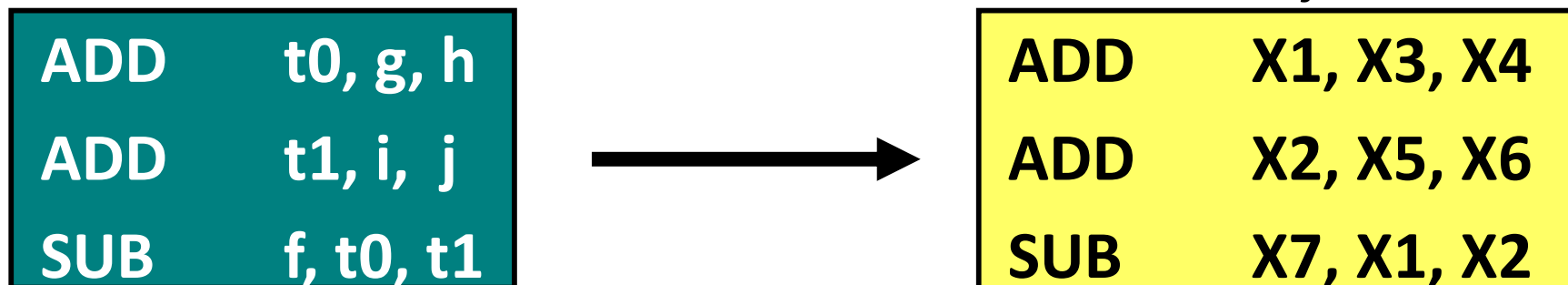
ARM Instruction Set—LEGv8 subset

- The main types of instructions fall into the familiar classes we saw with LC2K:
 1. Arithmetic
 - Add, subtract, (multiply not in LEGv8)
 2. Data transfer
 - Loads and stores—LDUR (load unscaled register), STUR, etc.
 3. Logical
 - AND, ORR, EOR, etc.
 - Logical shifts, LSL, LSR
 4. Conditional branch
 - CBZ, CBNZ, B.cond
 5. Unconditional branch (jumps)
 - B, BR, BL

LEGv8 Arithmetic Instructions

- Format: three operand fields
 - Dest. register usually the **first one** – check instruction format
 - ADD X3, X4, X7 // $X3 = X4 + X7$
 - **LC2K generally has the destination on the right!!!!**
 - E.g. add 1 2 3 // $r1 + r2 \rightarrow r3$
- C-code example: $f = (g + h) - (i + j)$

X1 \rightarrow t0
 X2 \rightarrow t1
 X3 \rightarrow g
 X4 \rightarrow h
 X5 \rightarrow i
 X6 \rightarrow j



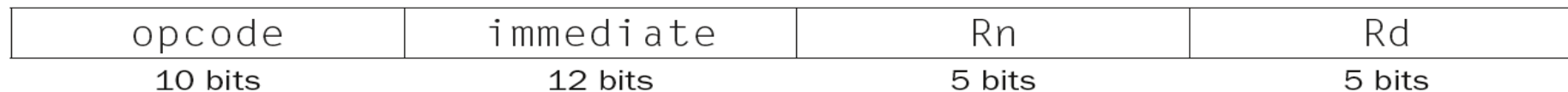
LEGv8 R-instruction Encoding

- Register-to-register operations
- Consider ADD X3, X4, X7
 - $R[Rd] = R[Rn] + R[Rm]$
 - $Rd = X3, Rn = X4, Rm = X7$
- Rm = second register operand
- ~~shamt = shift amount~~
 - not used in LEG for ADD/SUB and set to 0
- Rn = first register operand
- Rd = destination register
- ADD opcode is 10001011000, what are the other fields?

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

I-instruction Encoding

- Format: second source operand can be a register or **i**mmediate—a constant in the instruction itself
- e.g., `ADD X3, X4, #10` //although we write "ADD", this is "ADDI"
- Format: 12 bits for immediate constants 0-4095



- Don't need negative constants because we have SUBI
- C-code example: `f = g + 10`
- C-code example: `f = g - 10`

```
ADDI    X7, X5, #10
```

```
SUBI    X7, X5, #10
```

LEGv8 Logical Instructions

- Logical operations are bit-wise
- For example assume
 - $X9 = 11111111\ 11111111\ 11111111\ 00000000\ 00000000\ 00000000\ 00001101\ 11000000$
 - $X13 = 00000000\ 00000000\ 11011010\ 00000000\ 00000000\ 00000000\ 00111100\ 00000000$
 - AND $X2, X13, X9$ would result in
 - $X2 = 00000000\ 00000000\ 11011010\ 00000000\ 00000000\ 00000000\ 00001100\ 00000000$
- AND and OR correspond to C operators `&` and `|`
- For immediate fields the 12 bit constant is padded with zeros to the left—zero extended

Category	InstructionExample		Meaning	Comments
Logical	and	AND $X1, X2, X3$	$X1 = X2 \& X3$	Three reg. operands; bit-by-bit AND
	inclusive or	ORR $X1, X2, X3$	$X1 = X2 X3$	Three reg. operands; bit-by-bit OR
	exclusive or	EOR $X1, X2, X3$	$X1 = X2 \wedge X3$	Three reg. operands; bit-by-bit XOR
	and immediate	ANDI $X1, X2, 20$	$X1 = X2 \& 20$	Bit-by-bit AND reg. with constant
	inclusive or immediate	ORRI $X1, X2, 20$	$X1 = X2 20$	Bit-by-bit OR reg. with constant
	exclusive or immediate	EORI $X1, X2, 20$	$X1 = X2 \wedge 20$	Bit-by-bit XOR reg. with constant
	logical shift left	LSL $X1, X2, 10$	$X1 = X2 \ll 10$	Shift left by constant
	logical shift right	LSR $X1, X2, 10$	$X1 = X2 \gg 10$	Shift right by constant

LEGv8 Shift Logical Instructions

- LSR X6, X23, #2

X23 = 11111111 11111111 11111111 00000000 00000000 00000000 11011010 00000010
 X6 = 00111111 11111111 11111111 11000000 00000000 00000000 00110110 10000000

- C equivalent

- X6 = X23 >> 2;

- LSL X6, X23, #2

- What register gets modified?
- What does it contain after executing the LSL instruction?

- In shift operations Rm is always 0—shamt is 6 bit unsigned

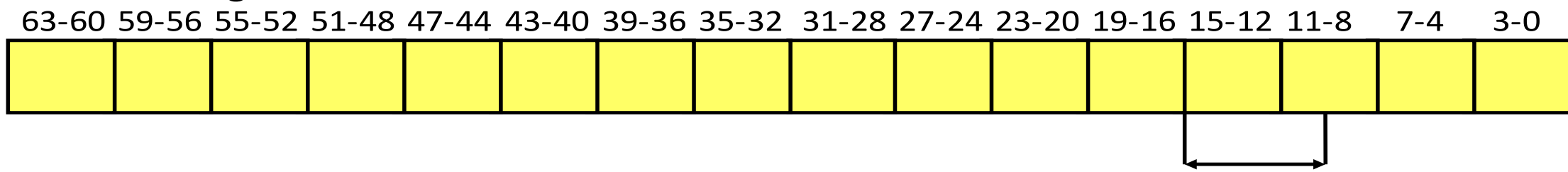
opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

Pseudo Instructions

- Instructions that use a shorthand “mnemonic” that expands to pre-existing assembly instruction
- Example:
 - `MOV X12, X2` // the contents of X2 copied to X12—X2 unchanged
- This gets expanded to:
 - `ORR X12, XZR, X2`
- What alternatives could we use instead of ORR?

Class Problem #1

- Show the C and LEGv8 assembly for extracting the value in bits 15:10 from a 64-bit integer variable



Assume the variable is in X1

Want these bits

```
x = x >> 10
```

```
x = x & 0x3F
```

```
LSR    X1, X1, #10
```

```
ANDI   X1, X1, #0x3F
```


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
 - Exact size 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	InstructionExample		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
	move wide with zero	MOVZ X1,20, LSL 0	$X1 = 20 \text{ or } 20 * 2^{16} \text{ or } 20 * 2^{32} \text{ or } 20 * 2^{48}$	Loads 16-bit constant, rest zeros
	move wide with keep	MOVK X1,20, LSL 0	$X1 = 20 \text{ or } 20 * 2^{16} \text{ or } 20 * 2^{32} \text{ or } 20 * 2^{48}$	Loads 16-bit constant, rest unchanged

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	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]

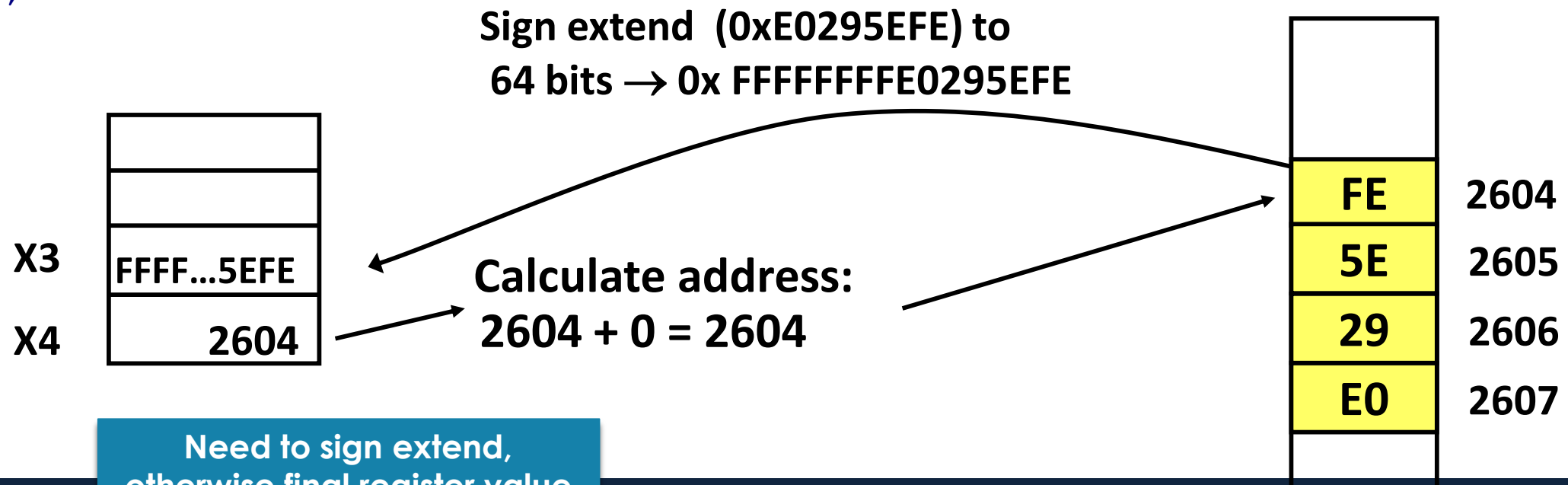


Load Instruction in Action – other example

```
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?

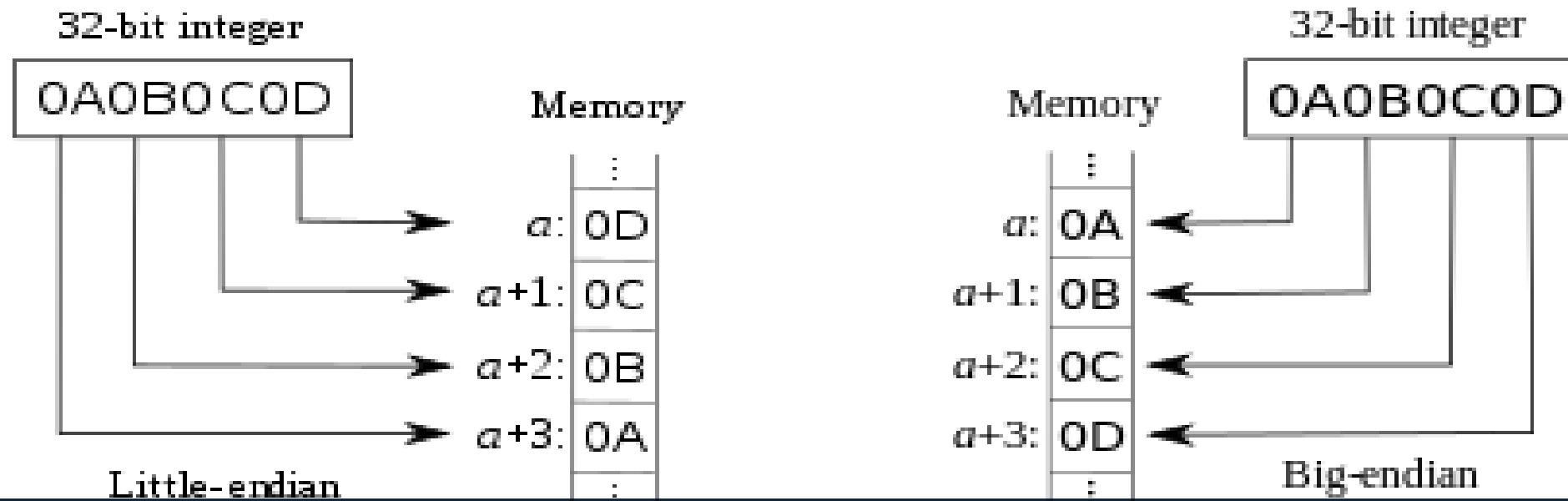
FE	2604
5E	2605
29	2606
E0	2607

- ... or like this?

E0	2604
29	2605
58	2606
FE	2607

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

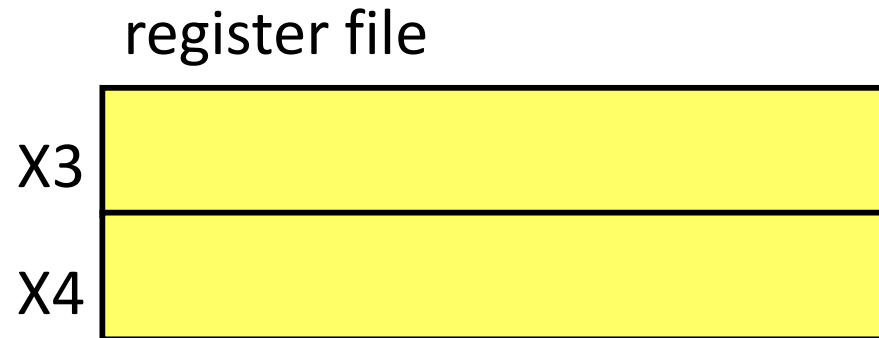
Agenda

- 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]
```



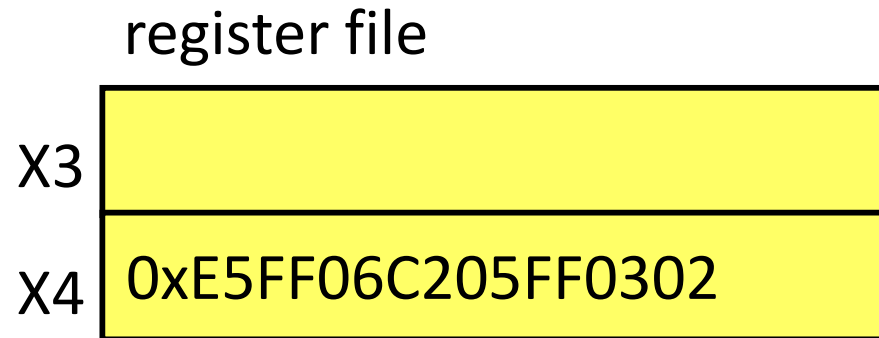
Memory
(each location is 1 byte)

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)

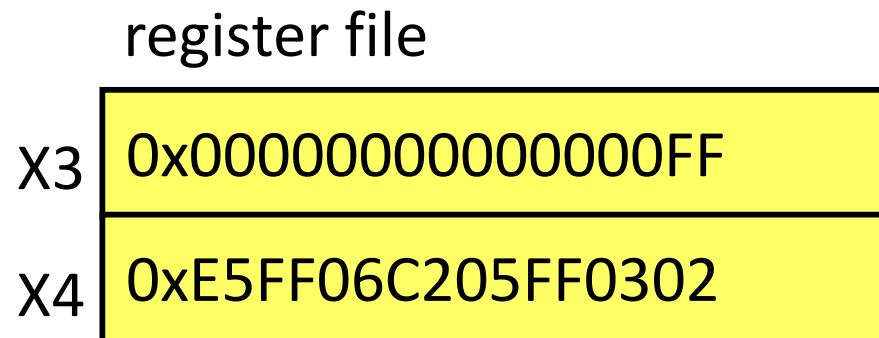
little endian

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

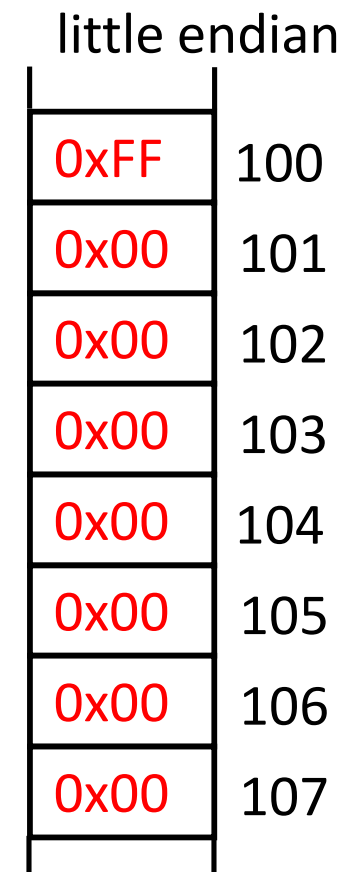
Example Code Sequence

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]
```



Memory
(each location is 1 byte)



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)

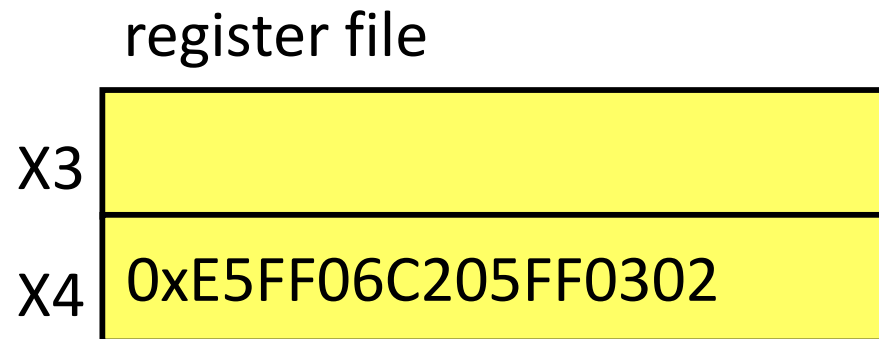
little endian

	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

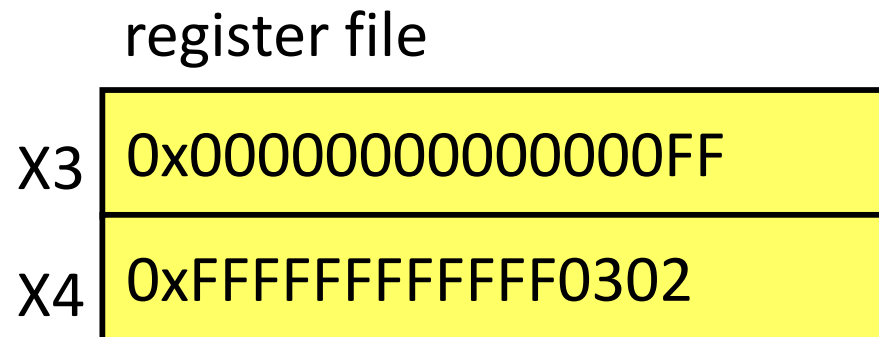
Memory
(each location is 1 byte)

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]
```



Memory
(each location is 1 byte)

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

Next Time

- More examples on doing stuff in ARM assembly
 - Like if/else, while loops, etc