4. Instruction Set Architecture – ARM - from C to Assembly

EECS 370 – Introduction to Computer Organization – Winter 2023

EECS Department
University of Michigan in Ann Arbor, USA

Announcements—Reminders

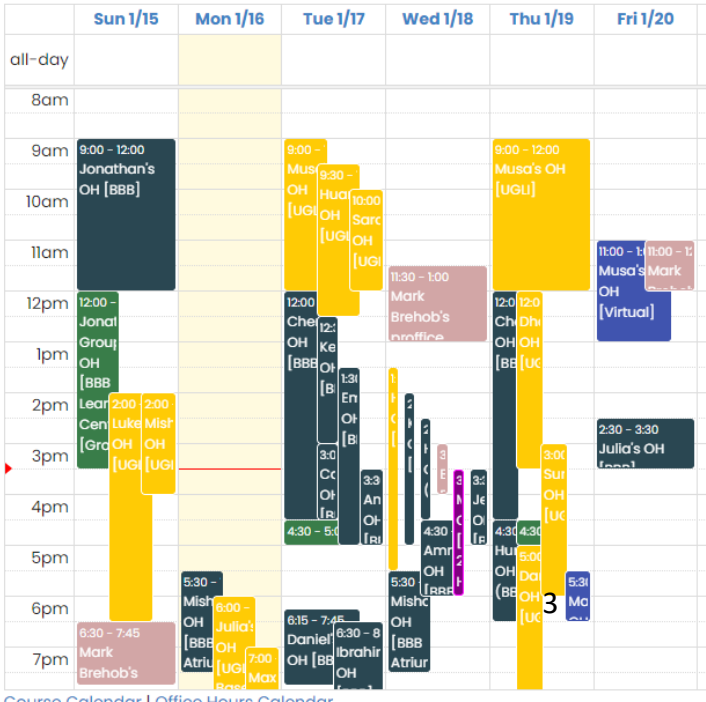
- Project 1.a due Thursday 1/26
- Project 1.s and 1.m due Thursday 2/2

- Homework 1 due Monday 1/23

- All due dates on calendar on web page

Office hours

- Most of the office hours use an on-line queue.
  - We have three queues: One for BBB, one for the UGLI, one for on-line
- We also have group office hours. These don't have queues and are focused on questions about lecture material, homework, and project specifications and high-level project issues.
  - See the website.



Instruction Set Architecture (ISA) Design Lectures

- Lecture 2: Storage types
- Lecture 3 : Addressing modes and LC2K
- **Lecture 4 : ARM Assembly**
- Lecture 5 : More Assembly and Function Calls
- Lecture 6 : Caller/Callee and Linker

Left-over questions:

"*I'm not getting labels & offsets for lw/sw/beq*" (LC2K)

```
y = 10;
while(x != y) {
  x *= 2;
}
```

```
        lw    0   4   ten
loop    beq   3   4   end
        add   3   3   3
        beq   0   0   loop
end     halt
ten     .fill 10
```

```
lw    0   4   5
beq   3   4   2
add   3   3   3
beq   0   0   -3
halt
.fill 10
```

label in lw/sw literally gets replaced w/ line #

label in beq gets replaced with offset, so offset+PC+1 = line #

Label	Line #
loop	1
end	4
ten	5

ARMv8 ISA
64-bit ISA

Chevrolet Corvette Z06

LS7
2006 7.0L V-8 (LS7)

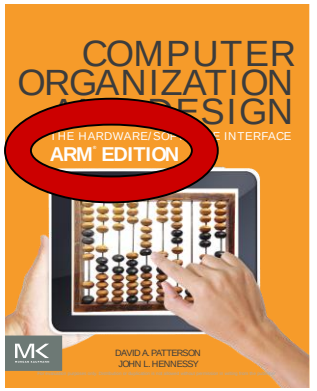
ARMv8 ISA

- 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 used in lecture and homework where LC2K cannot be.

ARMs and LEGs

ARM ISA

- ARMv8 is the 64 bit version—all registers are 64 bits wide
- Addresses are calculated to be 64 bits too
- BUT: Instructions are 32 bits
- We use a (small) subset of the v8 ISA used in P+H
- It is referred to as the LEGv8 in keeping with the body part theme!



ARM vs LC2K at a Glance

	LC2K	LEG
# registers	8	32
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

ARM ISA

- 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 a 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

ARM ISA

- Format: three operand fields
  - Dest. register usually the first one – check instruction format
  - ADD X3, X4, X7 – Think ADD X3 = X4, X7
  - LC2K generally has the destination on the right!!!!
  - E.g. add 1 2 3 // r3 = r1 + r2

- C-code example: f = (g + h) – (i + j)

X1 →t0
X2 →t1
X3→g
X4→h
X5→i
X6→j

```
ADD   t0, g, h
ADD   t1, i, j
SUB   f, t0, t1
```

```
ADD   X1, X3, X4
ADD   X2, X5, X6
SUB   X7, X1, X2
```

LEGv8 R-instruction Fields

ARM ISA

- 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?

LEGv8 Arithmetic Operations

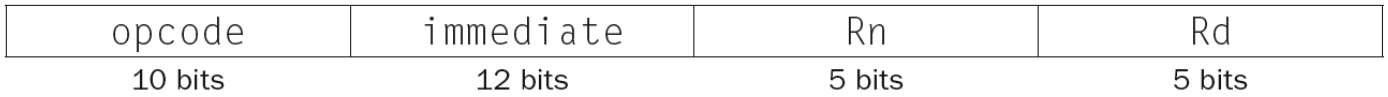
ARM ISA

- Machine State—more on this concept as our understanding evolves
  1. Registers: 32 64 bit registers. X31 aliased to XZR which is always 0 Can't use as a destination
  2. PC—Program counter
  3. FLAGS: NZVC record the results of (arithmetic) operations Negative, Zero, oVerflow, Carry—not present in LC2K

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	ADD X1, X2, X3	X1 = X2 + X3	Three register operands
	subtract	SUB X1, X2, X3	X1 = X2 – X3	Three register operands
	add immediate	ADDI X1, X2, 20	X1 = X2 + 20	Used to add constants
	subtract immediate	SUBI X1, X2, 20	X1 = X2 – 20	Used to subtract constants
	add and set flags	ADDS X1, X2, X3	X1 = X2 + X3	Add, set condition codes
	subtract and set flags	SUBS X1, X2, X3	X1 = X2 – X3	Subtract, set condition codes
	add immediate and set flags	ADDIS X1, X2, 20	X1 = X2 + 20	Add constant, set condition codes
	subtract immediate and set flags	SUBIS X1, X2, 20	X1 = X2 – 20	Subtract constant, set condition codes

I-instruction fields

- Format: second source operand can be a register or immediate—a constant in the instruction itself
- e.g., ADDI X3, X4, #10
- Format: 12 bits for immediate constants 0-4095



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

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

- C-code example: f = g - 10

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

LEGv8 Logical Instructions

ARM ISA

- Logical operations are bit-wise
- For example assume
  - X9 = 11111111 11111111 11111111 00000000 00000000 00000000 00001101 11000000
  - X14 = 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	Instruction	Example	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 ^ 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 ^ 20	Bit-by-bit XOR reg. with constant
	logical shift left	LSL X1, X2, 10	X1 = X2 << 10	Shift left by constant
	logical shift right	LSR X1, X2, 10	X1 = X2 >> 10	Shift right by constant

LEGv8 Shift Logical Instructions

ARM ISA

- 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;

- CLASS Question: 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

Pseudo Instructions

- Instructions that use a shorthand "mnemonic" that expands to an assembly instruction

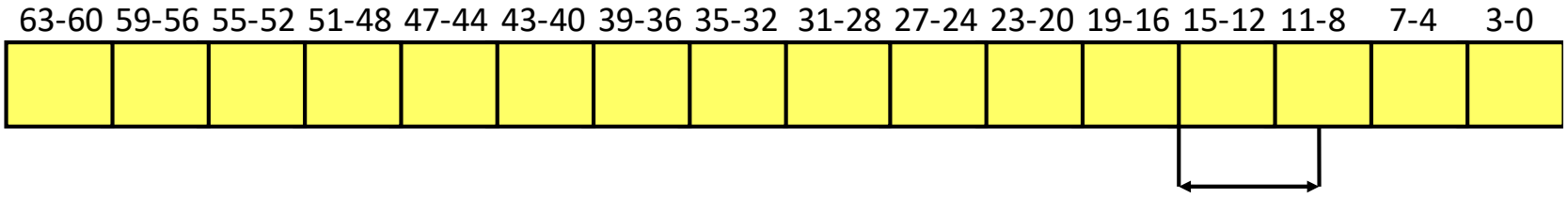
- Example:
  - MOV X12, X2 // the contents of X2 copied to X12—X2 unchanged

- This gets expanded to:
  - ORR X12, XZR, X2

- What alternatives could we use instead of ORR?

Class Problem

- 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

```
x = x >> 10
x = x & 0x3F
```

```
LSR  X1, X1, #10
ANDI X1, X1, #0x3F
```

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

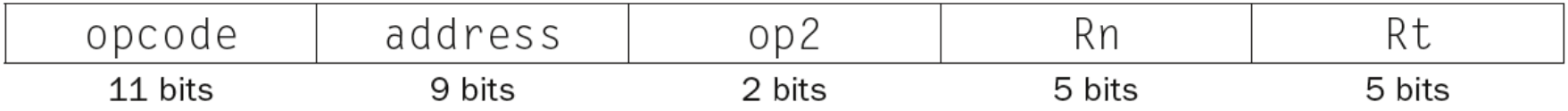
ARM ISA

- Employs base + displacement addressing mode
  - Base is a register
  - Displacement is 9-bit immediate ±256 bytes—sign extended to 64 bits

Category	Instruction	Example	Meaning	Comments
	load register	LDUR X1, [X2,40]	X1 = Memory[X2 + 40]	Doubleword from memory to register
	store register	STUR X1, [X2,40]	Memory[X2 + 40] = X1	Doubleword from register to memory
	load signed word	LDURSW X1,[X2,40]	X1 = Memory[X2 + 40]	Word from memory to register
	store word	STURW X1, [X2,40]	Memory[X2 + 40] = X1	Word from register to memory
	load half	LDURH X1, [X2,40]	X1 = Memory[X2 + 40]	Halfword memory to register
	store half	STURH X1, [X2,40]	Memory[X2 + 40] = X1	Halfword register to memory
	load byte	LDURB X1, [X2,40]	X1 = Memory[X2 + 40]	Byte from memory to register
	store byte	STURB X1, [X2,40]	Memory[X2 + 40] = X1	Byte from register to memory
	move wide with zero	MOVZ X1,20, LSL 0	X1 = 20 or 20 * 2^{16} or 20 * 2^{32} or 20 * 2^{48}	Loads 16-bit constant, rest zeros
	move wide with keep	MOVK X1,20, LSL 0	X1 = 20 or 20 * 2^{16} or 20 * 2^{32} or 20 * 2^{48}	Loads 16-bit constant, rest unchanged

D-Instruction fields

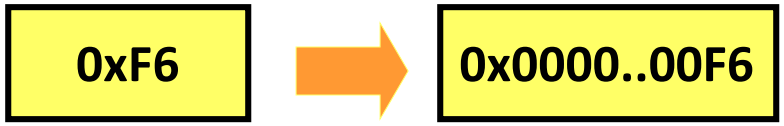
- Data transfer
- opcode and op2 define data transfer operation
- address is the ±256 bytes displacement
- Rn is the base register
- Rt is the destination (loads) or source (stores)
- More complicated modes are available in full ARMv8



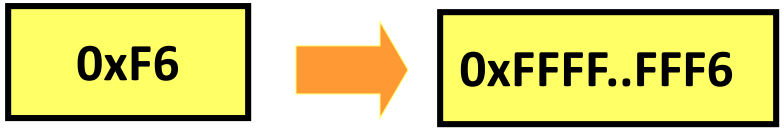
LEGv8 Memory Instructions

ARM ISA

- 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

ARM ISA

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

- LDUR X3, [X4, #100]
  - Load (unscaled) to register—retrieve a double word (64 bits) from address (X4+100)
- LDURH X3, [X4, #100]
  - Load halfword (16 bits) from address (X4+100) to the low 16 bits of X3—top 48 bits of X3 are set zero
- LDURB X3, [X4, #100]
  - Load byte (8 bits) from address (X4+100) and put in the low 8 bits of X3—zero extend the destination register X3 (top 56 bits)
- What about loading words?
- LDURSW X3, [X4, #100]
  - retrieve a word (32 bits) from address (X4+100) and put in lower half of X3—top 32 bits of X3 are sign extended

LEGv8 Data Transfer Instructions--Stores

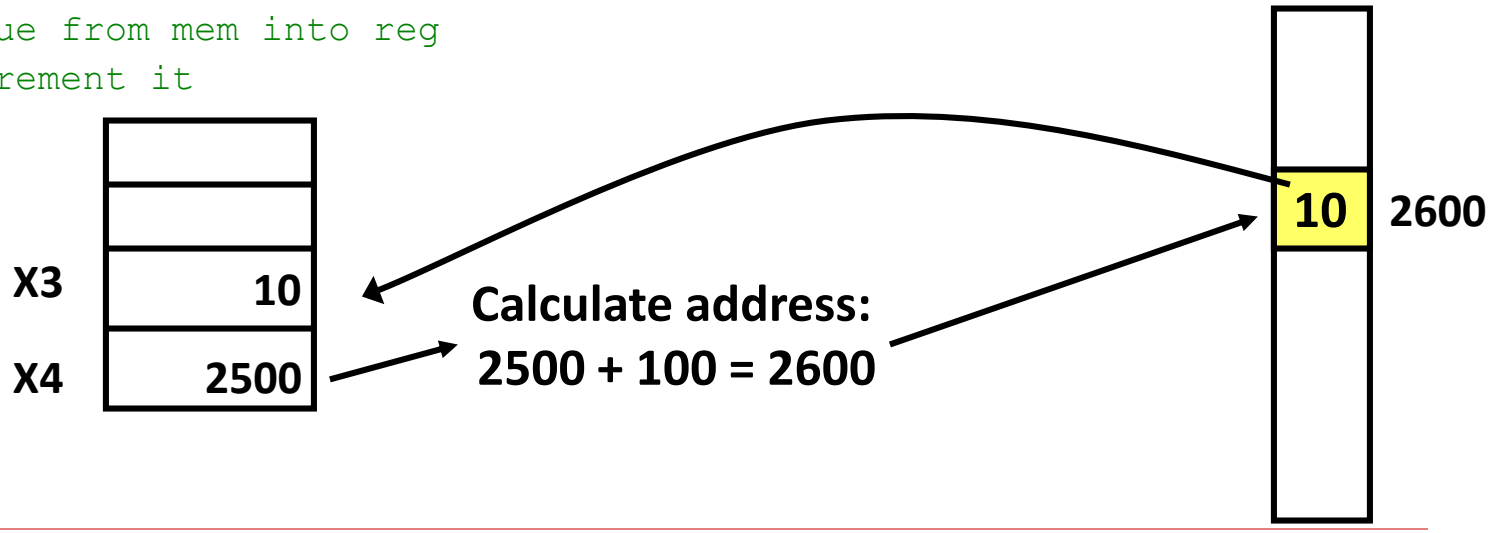
ARM ISA

- Store instructions are simpler—there is no sign/zero extension to consider
- STUR X3, [X4, #100]
  - Store (unscaled) register—write the double word (64 bits) in register X3 to the 8 bytes at address (X4+100)
- STURW X3, [X4, #100]
  - Store word—write the word (32 bits) in the low 4 bytes of register X3 to the 4 bytes at address (X4+100)
- STURH X3, [X4, #100]
  - Store half word—write the half word (16 bits) in the low 2 bytes of register X3 to the 2 bytes at address (X4+100)
- STURB X3, [X4, #100]
  - Store byte—write the least significant byte (8 bits) in register X3 to the byte at address (X4+100)

Load Instruction in Action

- LDURB X3, [X4, #100] // load byte
  Retrieves 8-bit value from memory location (X4+100) and puts the result into X3. The other 56 most significant bits are zero extended

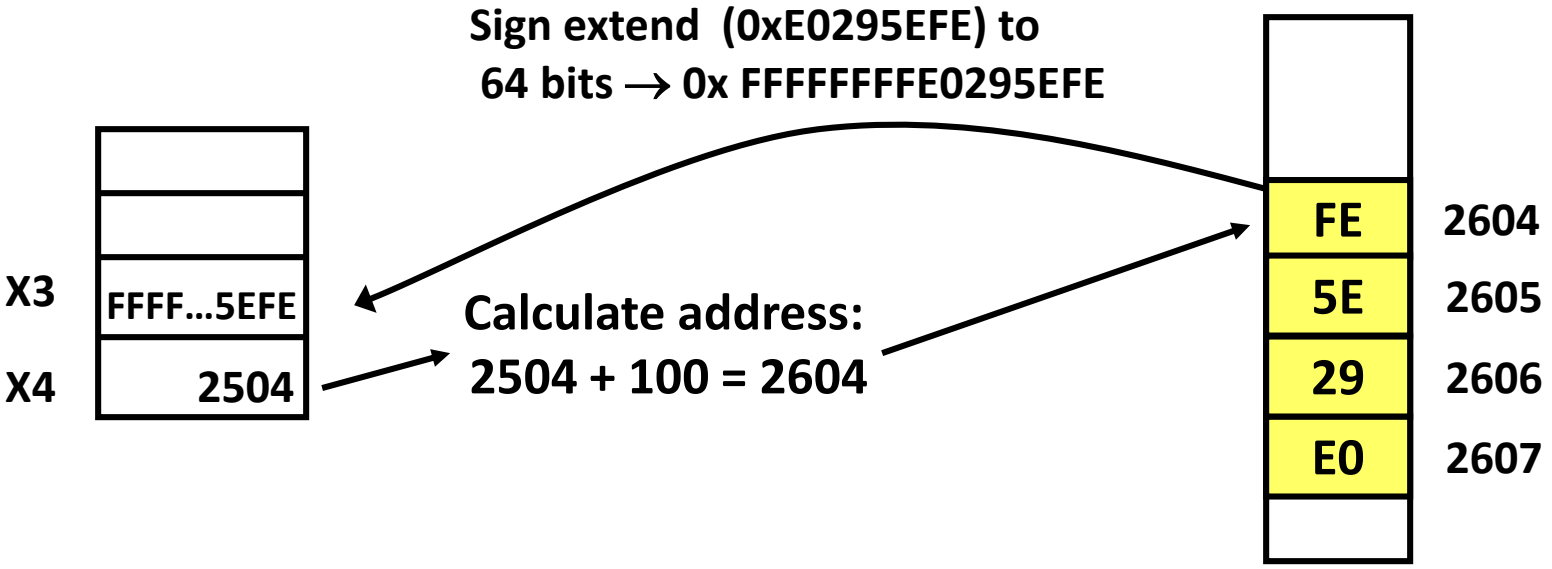
```
struct {
  int arr[25];
  unsigned char c;
} my_struct;

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



Load Instruction in Action – other example

ARM ISA

- LDURSW X3, [X4, #100] // load signed word
  Retrieves 4 byte value from memory location (X4+100) and puts the result into X3 (sign extended)



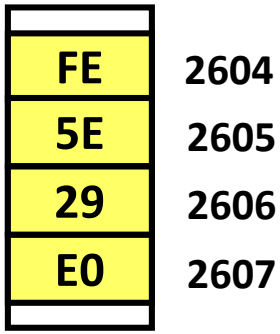
- We have mixed decimal and hex to keep the numbers easy to read
- Recall that E = 1110 and thus is treated as a negative 2's complement number

But wait…

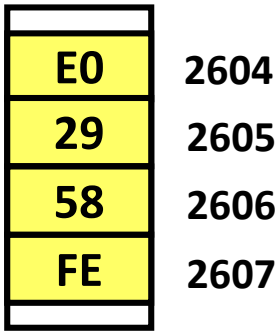
```
int my_big_number = -534159618; // 0xE0295EFE in 2's complement
```

- If I want to store this number in memory…

should it be stored like this?

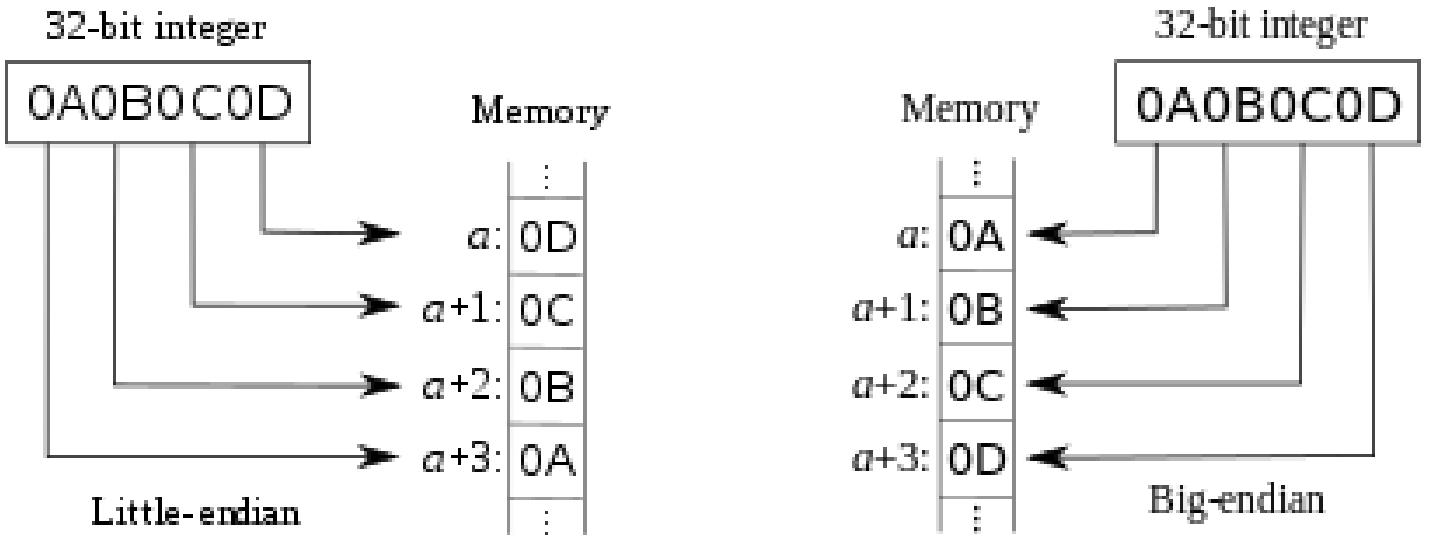


… or like this?



Big Endian vs. Little Endian

- Endian-ness: ordering of bytes within a word
  - Little - increasing numeric significance with increasing memory addresses
  - Big – The opposite, most significant byte first
  - 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

ARM ISA

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

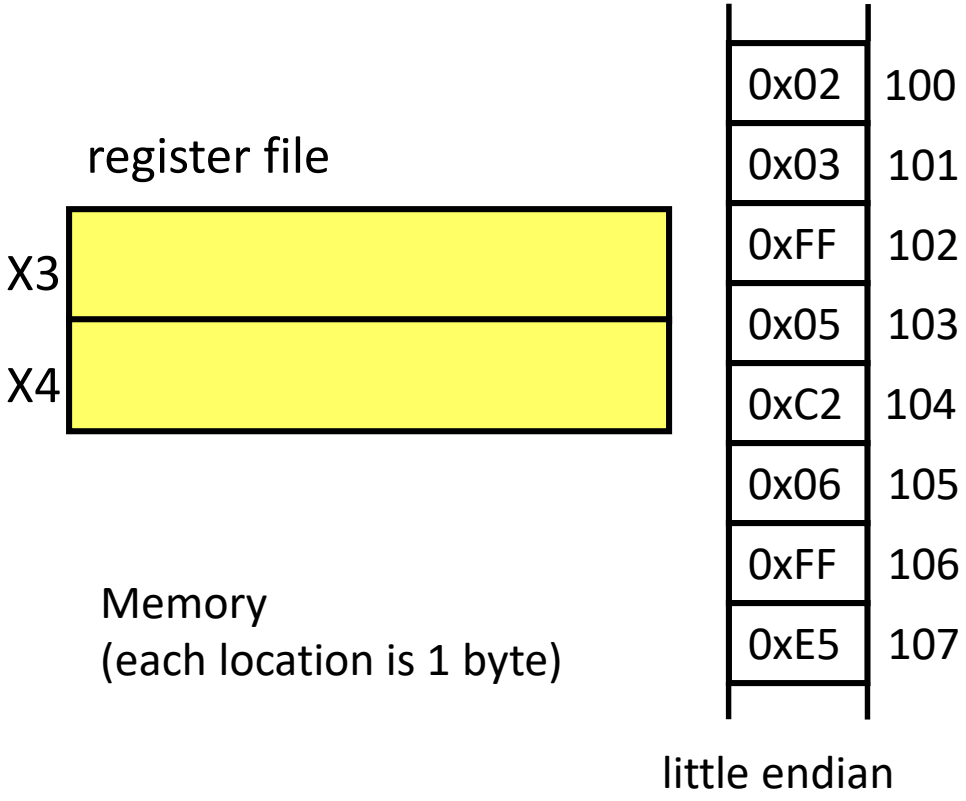
Desired amount of data to transfer?	Operation	Example
64-bits (double word or whole register)	STUR (Store unscaled register)	0xFEDC_BA98_7654_3210
16-bits (half-word) from lower bits of reg	STURH	0x0000_0000_0000_3210
8-bits (byte) from lower bits of reg	STURB	0x0000_0000_0000_0010
32-bits (word) from lower bits of reg	STURW	0x1111_1111_F654_3210

Example Code Sequence

ARM ISA

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]
```

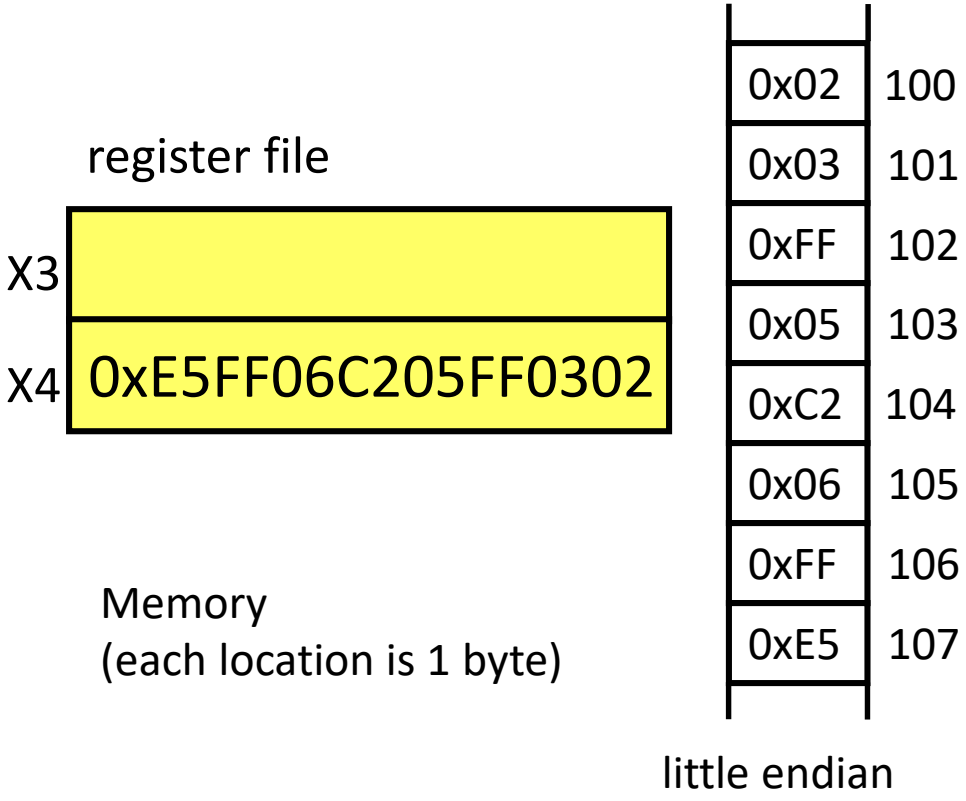


Example Code Sequence

ARM ISA

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]
```

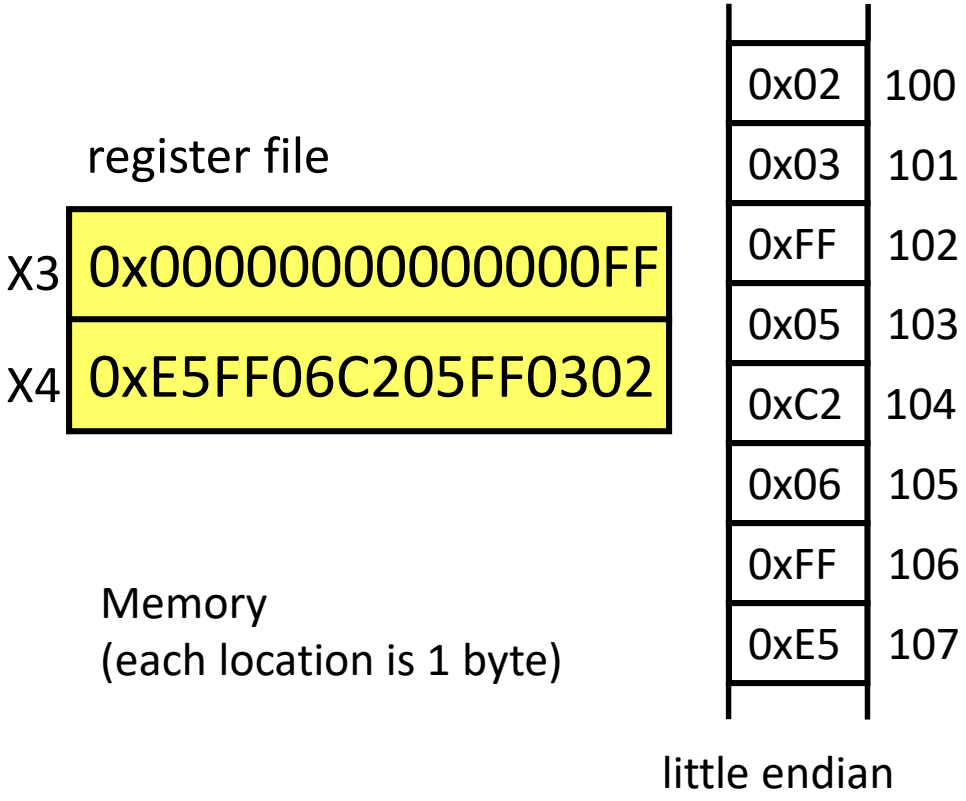


Example Code Sequence

ARM ISA

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]
```

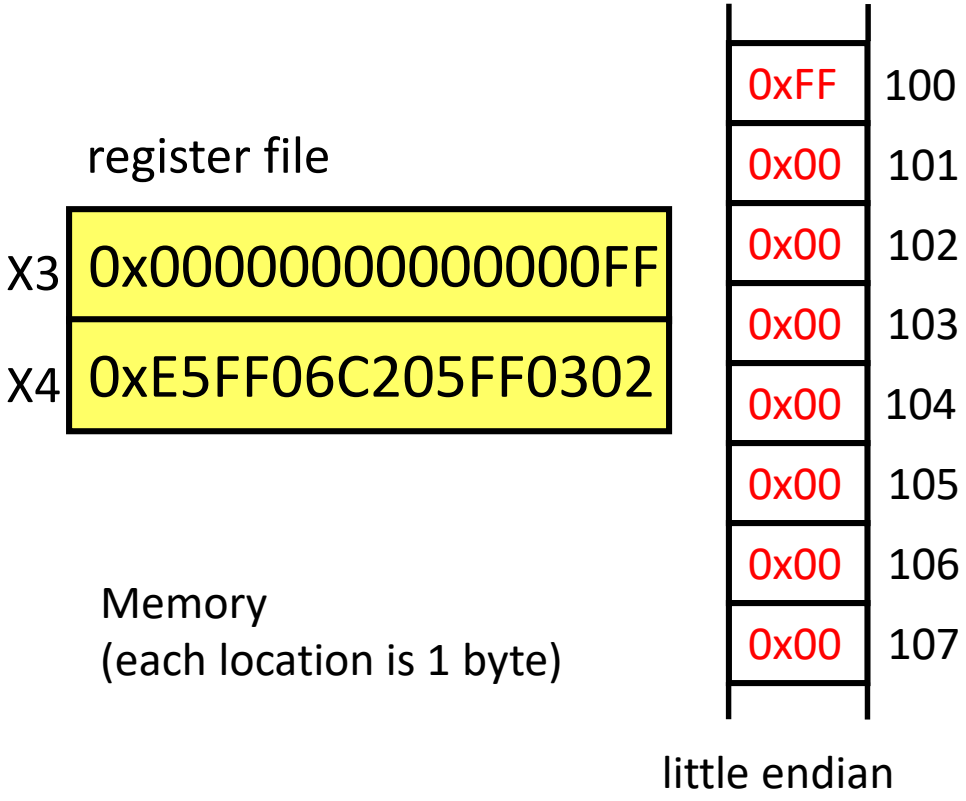


Example Code Sequence

ARM ISA

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]
```

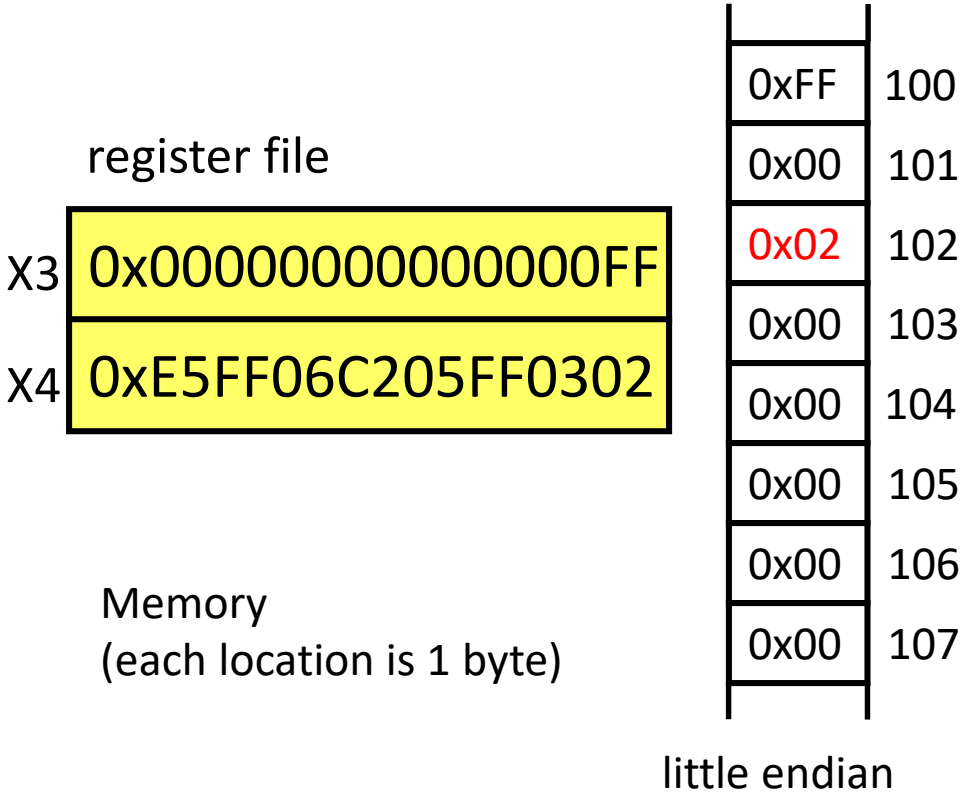


Example Code Sequence

ARM ISA

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]
```

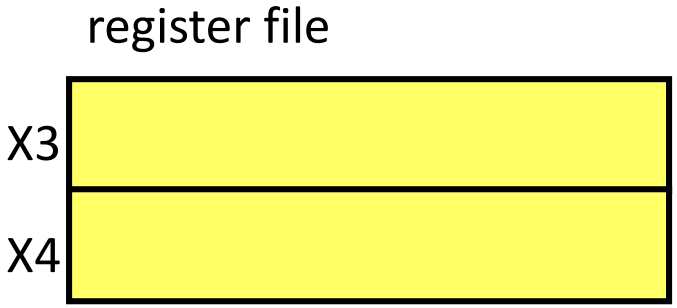


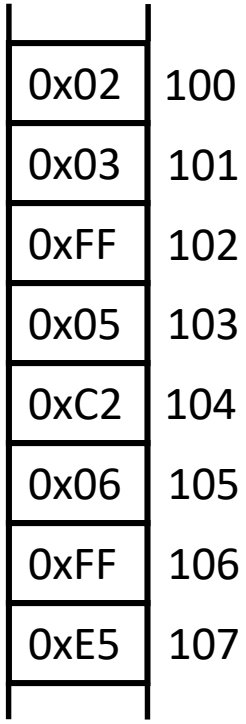
Example Code Sequence (2)

ARM ISA

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

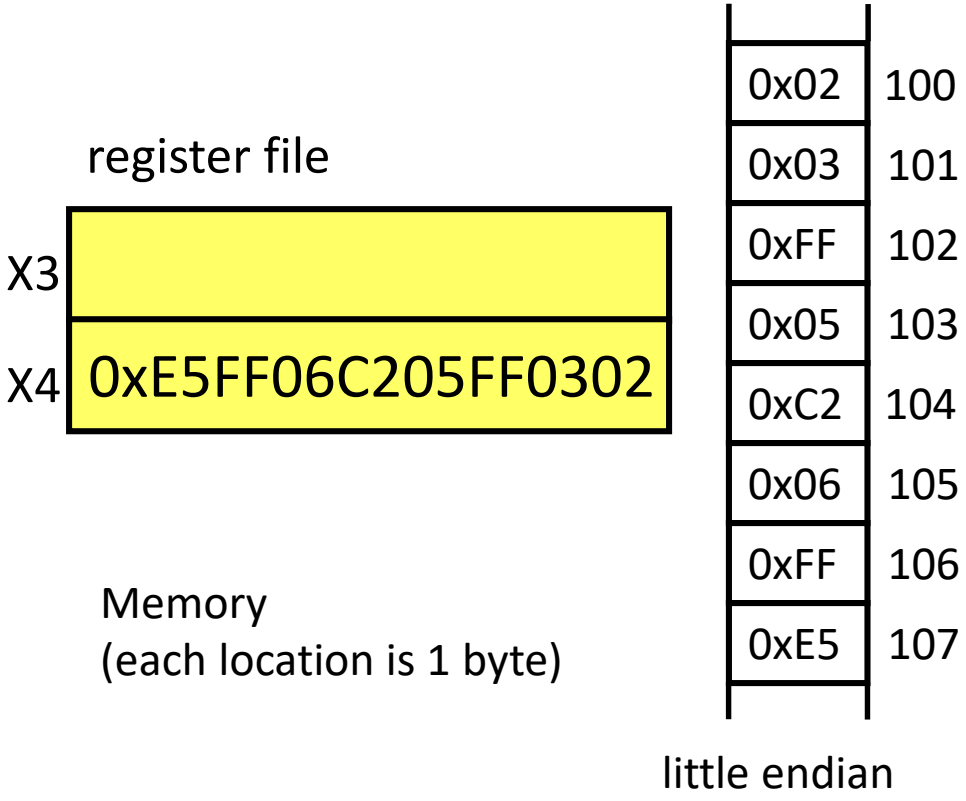
Memory
(each location is 1 byte)

little endian

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]
```

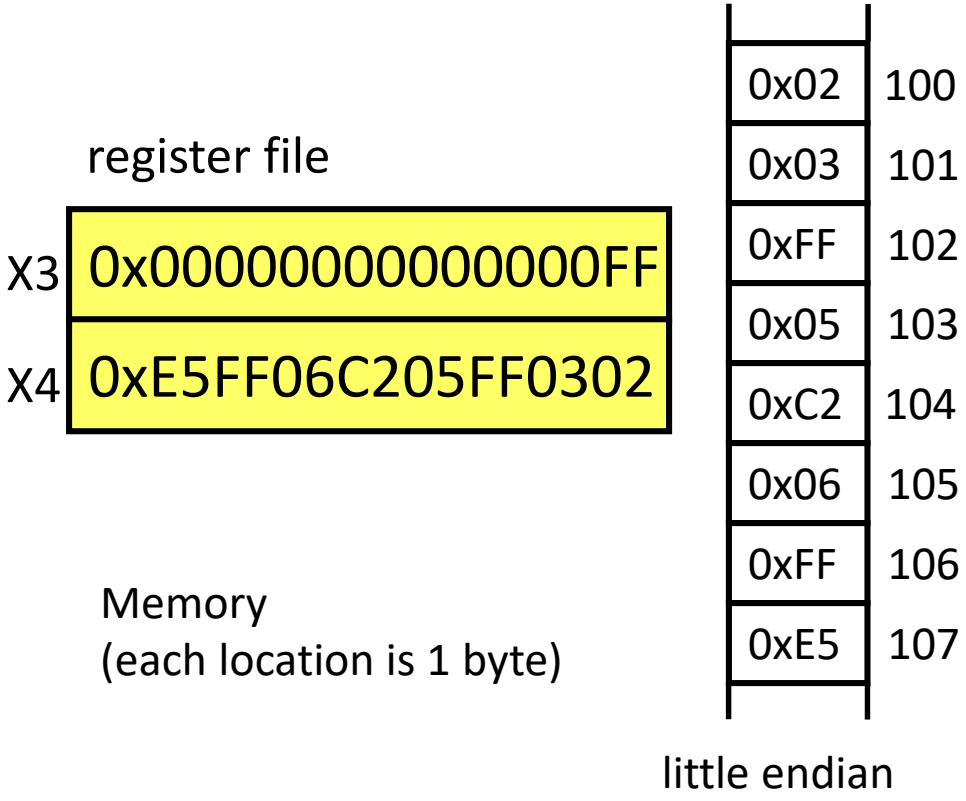


Example Code Sequence (2)

ARM ISA

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]
```

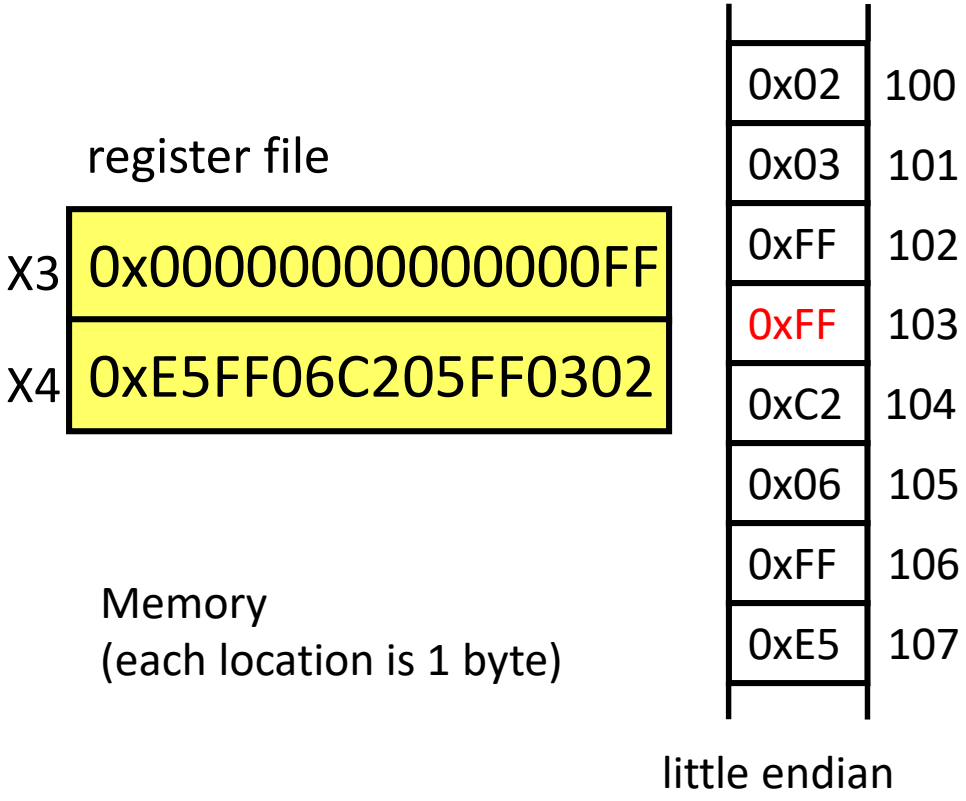


Example Code Sequence (2)

ARM ISA

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

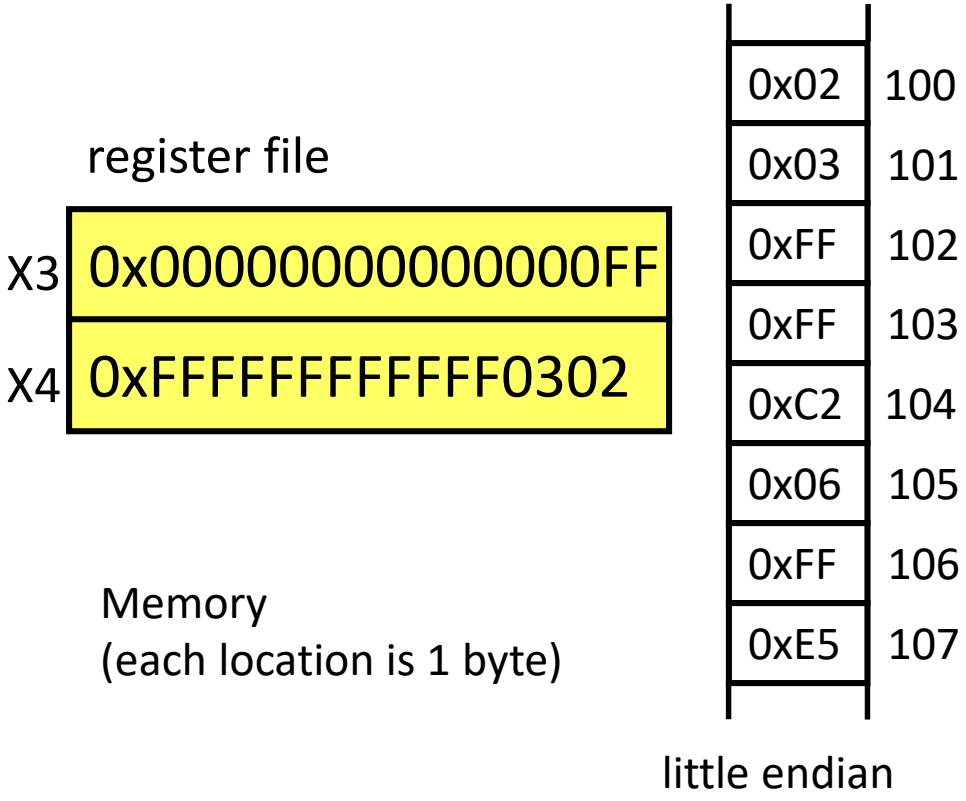
```
LDUR    X4, [X5, #100]
LDURB   X3, [X5, #102]
STURB   X3, [X5, #103]
LDURSW  X4, [X5, #100]
```



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]
```



Converting C to assembly – example 1

DATA LAYOUT

Write ARM assembly code for the following C expression:

C: a = b + names[i];

Assume that **a** is in X1, **b** is in X2, **i** is in X3, and the address of the array **names** is in X4. The array holds 64-bit integers

```
LSL     X5, X3, #3      // calculate array offset  i*8 (why #3 if we want 8?)
ADD     X6,  X4,  X5    // calculate address of names[i]
LDUR    X4,  [X6, #0]   // load names[i]
ADD     X1,  X2,  X4    // calculate b + names[i]
```

CLASS QUESTION: What if names was an array of **`int`** (4 bytes) type?

Converting C to assembly – example 2

DATA LAYOUT

Write ARM assembly code for the following C expression:

(assume an int is 4 bytes)

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
```

How do you determine the offsets for the struct sub-fields?