**Poll:** Anything positive about your weekend you'd like to share?

# EECS 370 - Lecture 4

## ARM

# Announcements

- Hope you had a meaningful MLK Day
  - Take some time to reflect on what it means for you
- HW 2
  - Posted on website, due next Monday
- P1
  - 3 parts, first part due next Thursday
- Always check course calendar for details on staff OH

# 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 | ADDS | Xd, | Xn, | Xm | X5 = X2 + X7 | flags NZVC |
| add immediate | ADDI | Xd, | Xn, | #uimm12 | X5 = X2 + #19 | 0 ≤ 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 ≤ 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] | ½ 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 | ½ 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 ≤ 9 bits signed immediate ≤ +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 ^ X7 | bit-wise EOR |
| exclusive or immediate | EOR | Xd, | Xn, | #uimm12 | X5 = X2 ^ #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 ≤ #simm26 ≤ 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 |

# Left-over questions:

"***What happens when you execute a .fill instruction?***"

- Hopefully, that never happens!
- Remember – (encoded) instructions and data coexist in memory
- .fill directive gives us a way to place a specific value somewhere in memory
  - It's intended to be the target of a load, the PC should never point there
- If PC **does** end up pointing there, it will be executed as if it was the corresponding encoded instruction
  - Which in this case is…?

```
 lw    0   4 seven
 halt
seven .fill 7
```

```
 lw    0   4 seven
 halt
 seven add 0 0 7
```

# Left-over questions:

- Remember, you can ask questions in Slido or end-of-lecture form:

# 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

# Labels in LC2K

- Labels are used in lw/sw instructions or beq instruction
- For lw or sw instructions, the assembler should compute offsetField to be equal to the address of the label
  - i.e. offsetField = address of the label
- For beq instructions, the assembler should translate the label into the numeric offsetField needed to branch to that label
  - i.e. PC+1+ offsetField = address of the label

# Labels in LC2K

- Labels are a way of referring to a line number in an assembly program.

```
loop   beq   3   4   end
       noop
       beq   0   0   loop
end    halt
```

- Here **loop** is 0 and **end** is 3.

```
// 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
tom    noop
       beq   0   0   loop
end    halt
```

**<u>Poll:</u> 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

# 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

# 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

- 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

ARM ISA

- 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 -> r3

- 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 Encoding

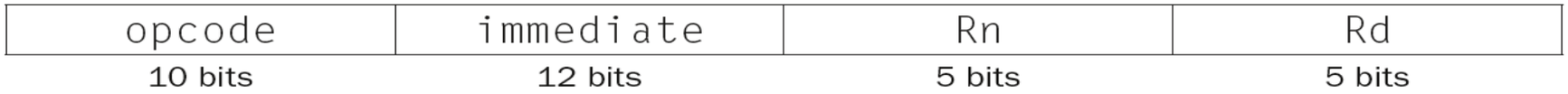
ARM ISA

- 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?

ARM ISA

# I-instruction Encoding

- Format:  second source operand can be a register or immediate—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

```
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
  - 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 | 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
```

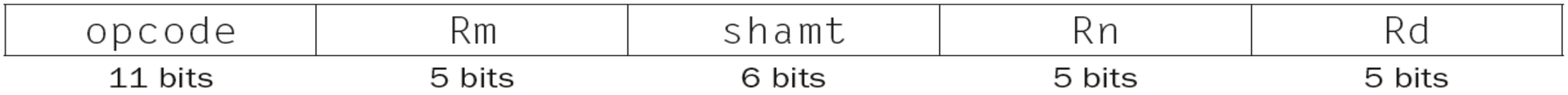
- C equivalent
  - X6 = X23 >> 2;

- LSL X6, X23, #2
  - What register gets modified?
  - What does it contain after executing the LSL instruction?

**Poll: Why is shifting so valuable?**

a) Makes multiplying easier
b) Allows quicker 2s-complement conversions
c) Allows for more complex branching behavior
d) It's always a good time to get shifty

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



ARM ISA

# 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

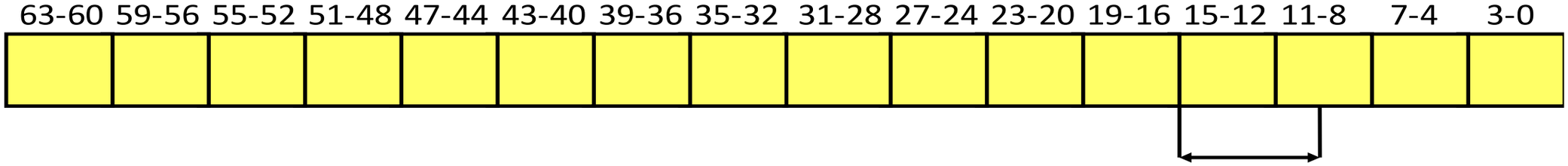
- This gets expanded to:
  - ORR   X12, XZR, X2

- What alternatives could we use instead of ORR?

# Class Problem #1

ARM ISA

- 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
```

**Poll: Which operations did you use (select all)?**

a) and
b) or
c) add
d) left shift
e) right shift

# Word vs Byte Addressing

**Memory Diagram:**

- 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

- 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

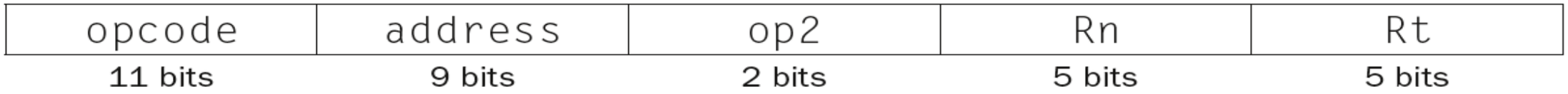
| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| | load register | LDUR X1, [X2,40] | X1 = Memory[X2 + 40] | Doubleword from memory to register |
| | store register | STUR X1, [X2,40] | Memory[X2 + 40] = X1 | Doubleword from register to memory |
| | load signed word | 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 |

ARM ISA

# 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

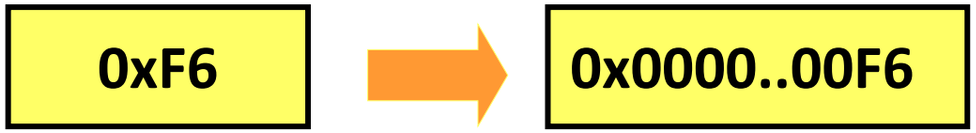
Look over formatting on your own



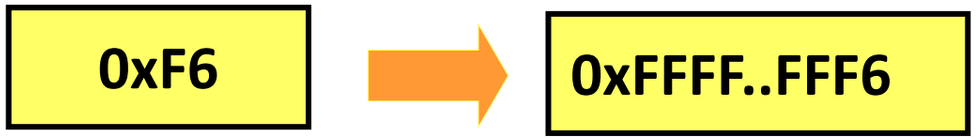
ARM ISA

# 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_**7**654_3210<br>or<br>0xFFFF_FFFF_**F**654_3210<br>(depends on bit 31) |

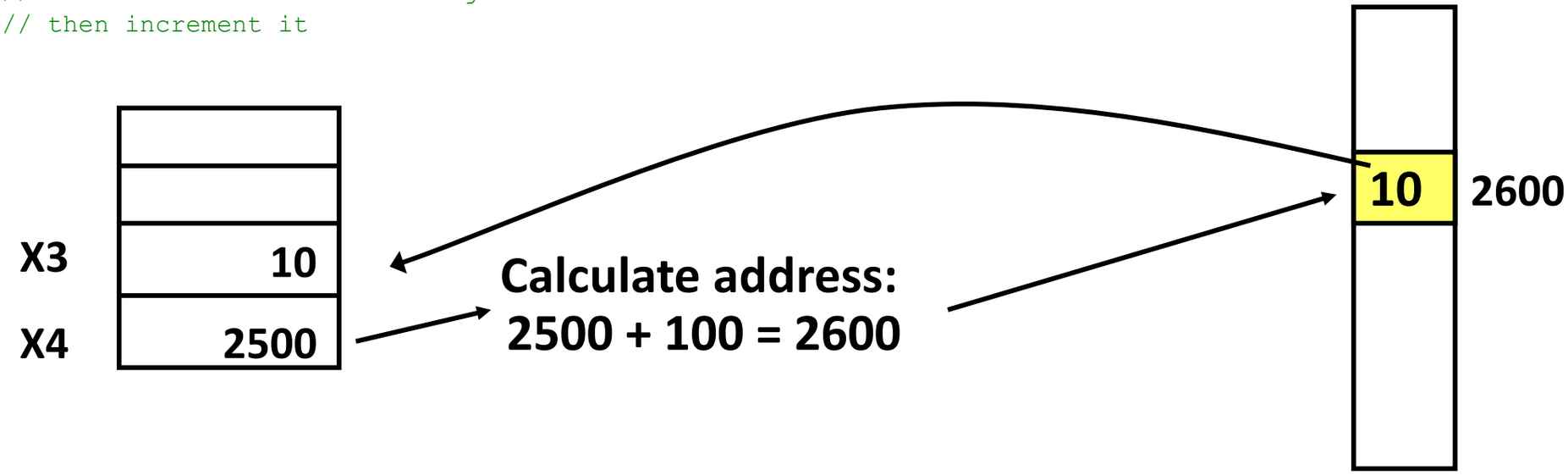
# Load Instruction in Action

ARM ISA

```
struct {
  int arr[25];
  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

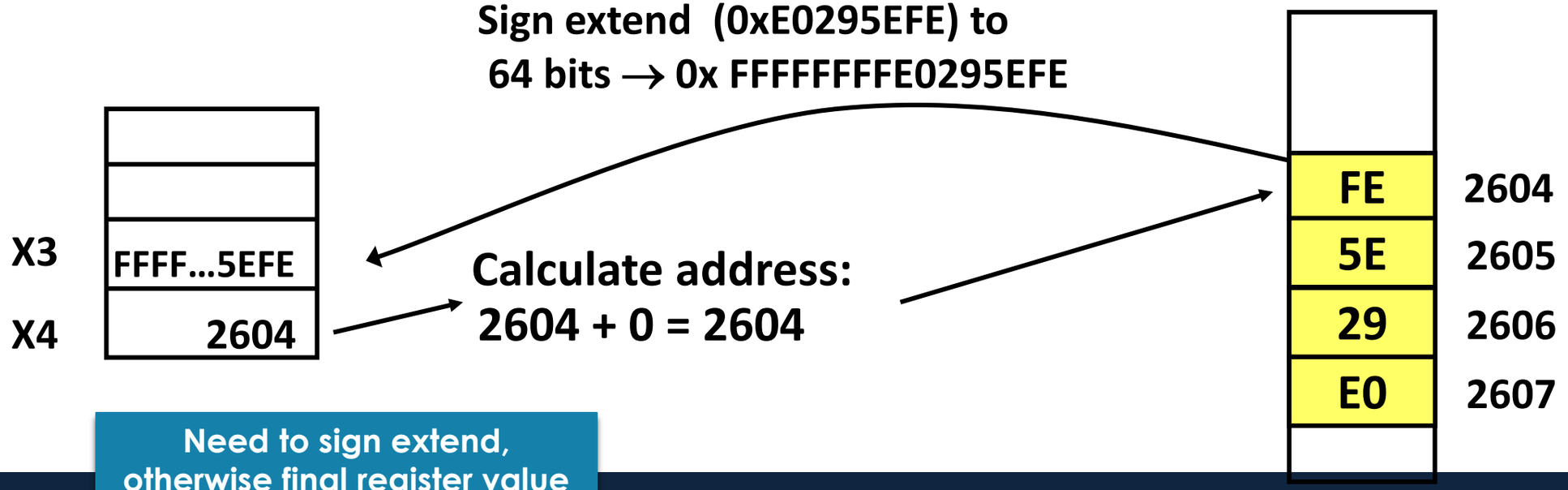
ARM ISA

```
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
};
```

TODO: add other instructions?

```
LDURSW  X3, [X4, #0]
```



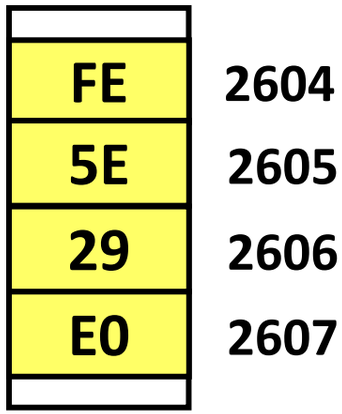

Need to sign extend, otherwise final register value will be positive!!!

# But wait…

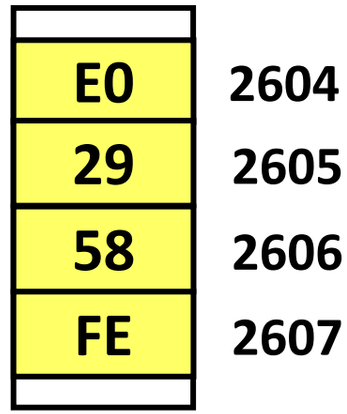
```
int my_big_number = -534159618; // 0xE0295EFE in 2's complement
```

- If I want to store this number in memory…

should it be stored like this?




… or like this?



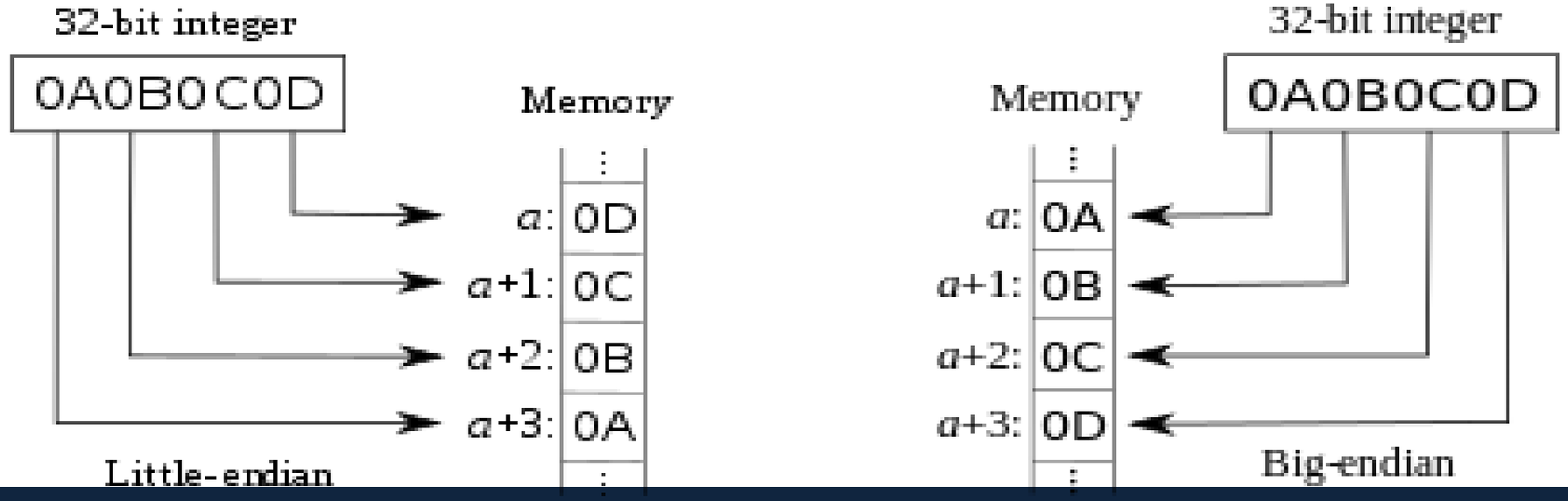
# Big Endian vs. Little Endian

**Poll: Which do you prefer?**

a) Big Endian

b) 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

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 |

# Next Time

- More examples on doing stuff in ARM assembly
  - Like if/else, while loops, etc
- Lingering questions / feedback? I'll include an anonymous form at the end of every lecture: https://bit.ly/3oXr4Ah