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

EECS 370 - Lecture 4

ARM

byte be like

Live Poll + Q&A: slido.com #eecs370 *Poll and Q&A Link*

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
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

ARM ISA

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 -> r3

LC2K: reg1 reg2 dest
LEG: dest reg1 reg2

- 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?

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

I-instruction Encoding

ARM ISA

- 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

opcode	immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

- 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	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 ^ 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

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

Pseudo Instructions

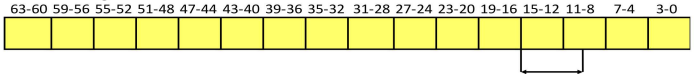
ARM ISA

- 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



Want these bits

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 ✓

LSR X2 X1 #10
ANDI X2 X2 #63
0x3F

Agenda

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

Memory Diagram:

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

- 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

D-Instruction fields

ARM ISA

- 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

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

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

    0xF6 → 0x0000..00F6
  - Option B: sign extend

    0xF6 → 0xFFFF..FFF6
- We'll need different instructions for different options

Load Instruction Sizes

le durb

ARM ISA

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)

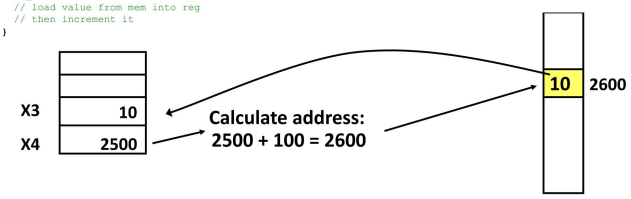
Load Instruction in Action

ARM ISA

```
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]



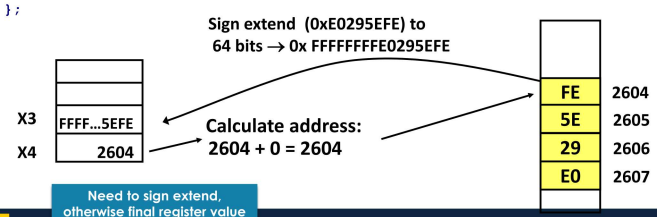
Load Instruction in Action – other example

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
};
```

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?

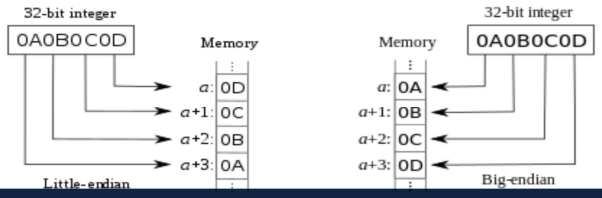
FE	2604
5E	2605
29	2606
E0	2607

- … 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

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	0xFFFF_FFFF_F654_3210

Agenda

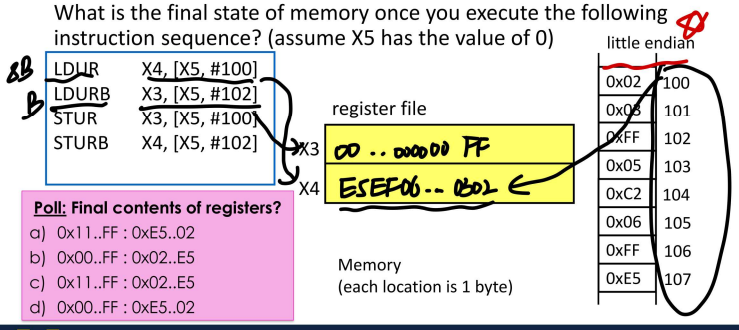
- ARM overview and basic instructions
- Memory instructions
  - Handling multiple data widths
- **Sample Problems**

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]
```



Poll: Final contents of registers?

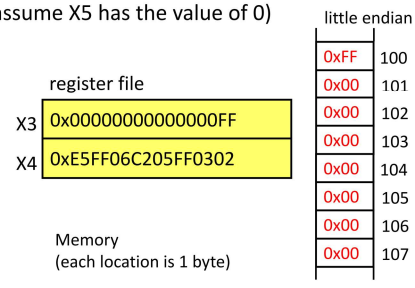
a) 0x11..FF : 0xE5..02
b) 0x00..FF : 0x02..E5
c) 0x11..FF : 0x02..E5
d) 0x00..FF : 0xE5..02

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]
```

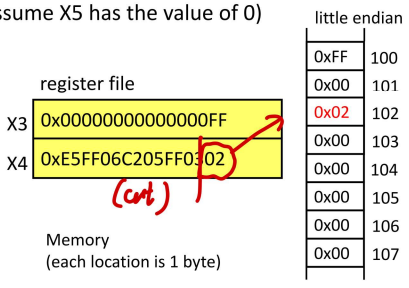


Example Code Sequence

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]
```

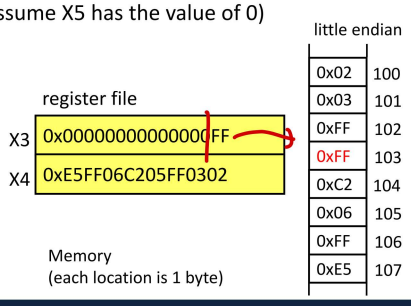


Example Code Sequence (2)

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]
```



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]
```