EECS 370

Lab 3:
ARM Assembly

Upcoming Assignments

- Lab 3 due Wednesday 9/18
- Project 1s & 1m due Thursday 9/19
- Homework 1 due Monday 9/23

LC2K Instructions Compared to ARM

cheat sheet!

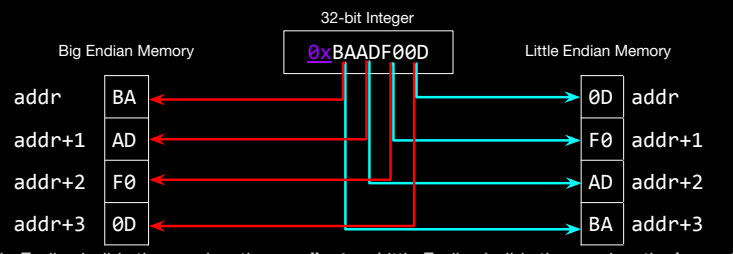
LC2K	ARM	Differences
add (R-type)	ADD	Also ADDI for constants, and SUB/SUBI for subtraction. (LC2K Subtraction: nor the 2nd with itself, add 1, add result to 1st one). MOVZ #0 zeros out reg. We use LSL, LSR for shifts. (Pseudo-instructions: MUL)
nor (R-type)	Load #-1 then EOR/SUB	Much easier to use ORR/ORRI, ADD/ADDI, EOR/EORI. (Note: a NOT instruction exists in pseudocode for ARM overall, but not LEGv8)
lw (I-type)	LDURSW	This is for 32 bits. Also LDUR, LDURH, LDURB.
sw (I-type)	STURW	This is for 32 bits. Also STUR, STURH, STURB.
beq (I-type)	CMP then B.EQ	CBZ branch if zero (beq 0 regB offset), B for unconditional (beq 0 0 offset)
jalr (J-type)	BR (branch reg)	Using BL with #0 right before BR stores return address
halt (O-type)	End of file	
noop (O-type)	NOP	Technically not in LEGv8, but present in ARM overall

Data is Stored in Memory in Chunks

Each chunk is (typically) the size of a single byte

Think of each chunk like a wooden letter block:
To interpret the word, we can rearrange the blocks, but can't change the letters.

Endianness in Byte-Addressable Systems



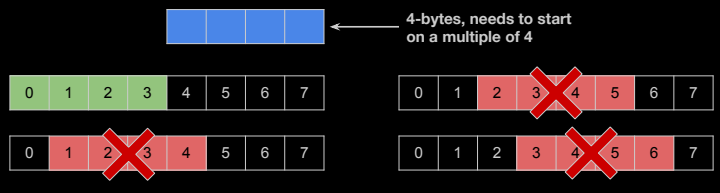
Big Endian builds the word so the **smallest** address is the MSB.
This can be more natural for humans to think about.

Little Endian builds the word so the **largest** address is the MSB.
Faster for a computer to read, as int ops can start after reading the first byte.

Memory Alignment

In memory, data is aligned to the size of its type

This makes it more efficient to access within memory
and is important for caching (we'll see why later in the course)



Typical Size of Data Types

cheat sheet!

Data Type		Size / Alignment
`char`		1 Byte
`short`		2 Bytes
`int, float`		4 Bytes
`double`		8 Bytes
`long`	*(32-bit Architecture)*	4 Bytes
`long`	*(64-bit Architecture)*	8 Bytes
pointer	*(32-bit Architecture)*	4 Bytes
pointer	*(64-bit Architecture)*	8 Bytes

Aligning Data within a `struct`

Since a `struct` can contain multiple different data types,
we must align the `struct` using the ***largest primitive type*** within it

Furthermore, we may need to pad the `struct` to keep everything aligned,
even padding the end in case we have an array of `structs`

Example: Addressing Data in Memory

Determine the start and end addresses for the following variables.
(Assume a 64-bit system and that the data starts on address 200_{10})

```
struct {
    short a;
    int b;
    char c;
    int* d;
    struct {
        int e;
        char f [10];
    } g;
} example;
```

```
a:      200 - 201
b:      204 - 207
c:      208 - 208
d:      216 - 223

e:      224 - 227
f:      228 - 237
g:      224 - 239
total:  200 - 239   →  40 bytes
```


Visual Walkthrough

```
struct {
    short a;
    int b;
    char c;
    int* d;
    struct {
        int e;
        char f [10];
    } g;
} example;
```

Address
200
201
202
203
204
205
206
207

Visual Walkthrough

```
struct {
    short a;
    int b;
    char c;
    int* d;
    struct {
        int e;
        char f [10];
    } g;
} example;
```

Address
200
201
202
203
204
205
206
207

Visual Walkthrough

```
struct {
    short a;
    int b;
    char c;
    int* d;
    struct {
        int e;
        char f [10];
    } g;
} example;
```

Address
200
201
202
203
204
205
206
207

Visual Walkthrough

```
struct {
    short a;
    int b;
    char c;
    int* d;
    struct {
        int e;
        char f [10];
    } g;
} example;
```

Address
208
209
210
211
212
213
214
215

Visual Walkthrough

```
struct {
    short a;
    int b;
    char c;
    int* d;
    struct {
        int e;
        char f [10];
    } g;
} example;
```

Address
216
217
218
219
220
221
222
223

Visual Walkthrough

```
struct {
    short a;
    int b;
    char c;
    int* d;
    struct {
        int e;
        char f [10];
    } g; //Align with e
} example;
```

Address
224
225
226
227
228
229
230
231

int e must start where struct g starts

Visual Walkthrough

```
struct {
    short a;
    int b;
    char c;
    int* d;
    struct {
        int e;
        char f [10];
    } g; //Align with e
} example;
```

Address
224
225
226
227
228
229
230
231

Visual Walkthrough

```
struct {
    short a;
    int b;
    char c;
    int* d;
    struct {
        int e;
        char f [10];
    } g; //Align with e
} example;
```

Address
224
225
226
227
228
229
230
231

Visual Walkthrough

```
struct {
    short a;
    int b;
    char c;
    int* d;
    struct {
        int e;
        char f [10];
    } g; //Align with e
} example;//Align with d
```

Address
232
233
234
235
236
237
238
239

Visual Walkthrough

```
struct {
    short a;
    int b;
    char c;
    int* d;
    struct {
        int e;
        char f [10];
    } g; //Align with e
} example;//Align with d
```

Address
232
233
234
235
236
237
238
239

struct g ends on multiple of 4

Visual Walkthrough

```
struct {
    short a;
    int b;
    char c;
    int* d;
    struct {
        int e;
        char f [10];
    } g; //Align with e
} example;//Align with d
```

Address
232
233
234
235
236
237
238
239

struct example ends on multiple of 8

Grid View

```
struct {
    short a;
    int b;
    char c;
    int* d;
    struct {
        int e;
        char f [10];
    } g;
} example;
```

offset	+0	+1	+2	+3	+4	+5	+6	+7
base	a	a			b	b	b	b
+8	c							
+16	d	d	d	d	d	d	d	d
+24	e	e	e	e	f	f	f	f
+32	f	f	f	f	f	f	g	g

Struct Memory Optimization (281 Tip)

- We can rearrange the variables in a struct to use as little padding as possible for BOTH 32-bit and 64-bit systems.
- Greedy yet optimal algorithm: start with the largest size primitives, then go down.
  - This also works by starting with the smallest, and working up.
  - And there might be more solutions for a given struct.
- *Count structs as multiples of their largest member, as with alignment.*

ARM's LEGv8 ABI

Application Binary Interface (ABI) defines convention of how registers should be used

Ex: LEGv8 Register X0-X7 are used for arguments/results

See here for full LEGv8 ABI

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
X0 – X7	0-7	Arguments / Results	No
X8	8	Indirect result location register	No
X9 – X15	9-15	Temporaries	No
X16 (IP0)	16	May be used by linker as a scratch register; other times used as temporary register	No
X17 (IP1)	17	May be used by linker as a scratch register; other times used as temporary register	No
X18	18	Platform register for platform independent code; otherwise a temporary register	No
X19-X27	19-27	Saved	Yes
X28 (SP)	28	Stack Pointer	Yes
X29 (FP)	29	Frame Pointer	Yes
X30 (LR)	30	Return Address	Yes
XZR	31	The Constant Value 0	N.A.

EECS 370

Lab 3:
ARM Assembly