Upcoming Assignments

- Homework 1 due Monday Sept 22
- Project 1s & 1m due Thursday Sept 18

This Lab Will Cover:

1. **ARM LEGv8 Assembly**
2. Memory and Struct Alignment

LC2K Instructions Compared to ARM

cheat sheet!

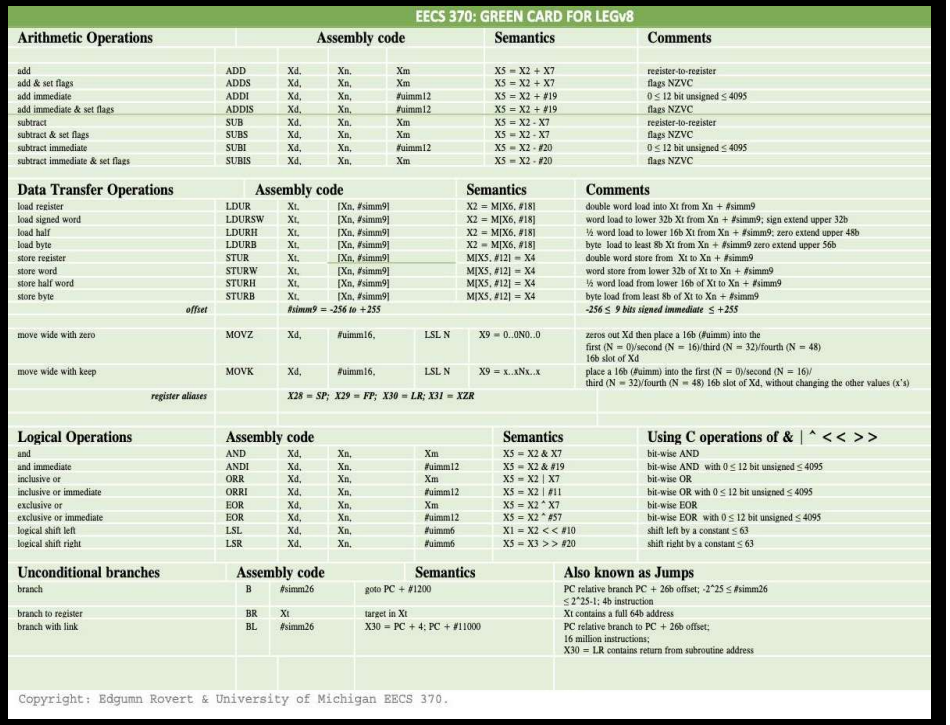
UNIVERSITY OF MICHIGAN

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, AND/ANDI, 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

ARM LEGv8 Reference Card

See the ARM LEGv8 reference card for a helpful overview of important LEGv8 instructions.

This can be found in the website under “References Material” in the “Course Resources” section.



EECS 370: GREEN CARD FOR LEGv8

Arithmetic Operations	Assembly code				Semantics	Comments
add	ADD	Xd,	Xn,	Xm	X5 = X2 + X7	reister-to-reister
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	reister-to-reister
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

ARM LEGv8 ABI

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

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

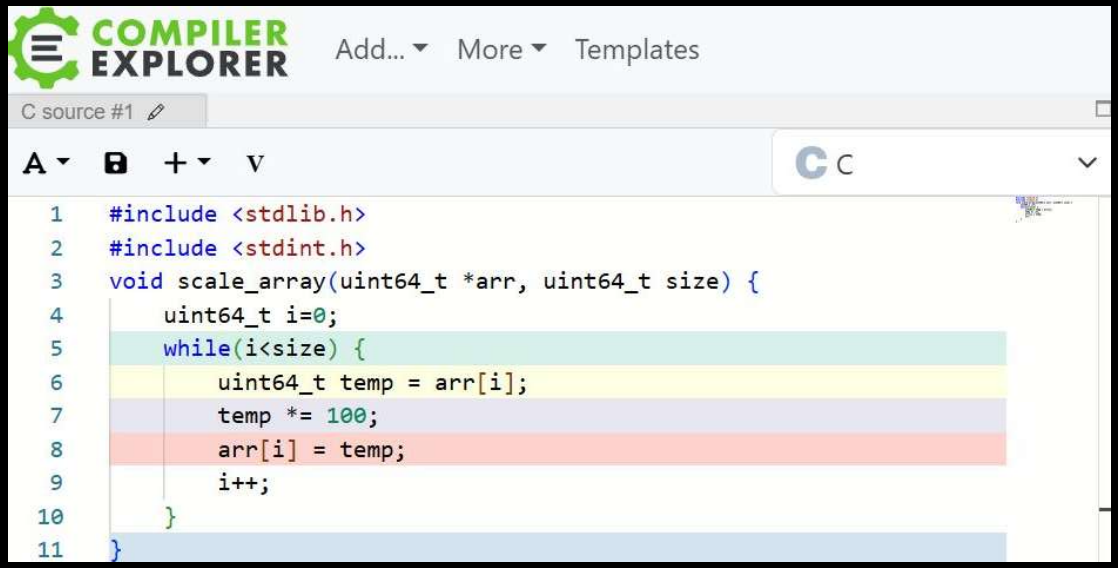
Snippet shown here

REGISTER NAME, NUMBER, USE, CALL CONVENTION

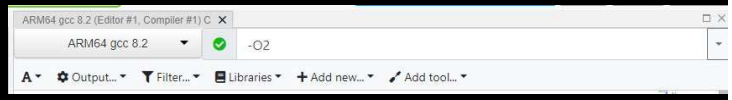
NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
X0 – X7	0-7	Arguments / Results	No
X8	8	Indirect result location register	No
X9 – X15	9-15	Temporaries	No
X16 (IP0)	16	May be used by linker as a scratch register; other times used as temporary register	No
X17 (IP1)	17	May be used by linker as a scratch register; 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.

Problem 1 & 2

- Introduce yourself to your group
- Go to godbolt.org
- Write code into source tab



- Select "carm64g820" or “ARM64 gcc 8.2" for the architecture and "-O2" for the compiler options



- Read the code and walk through how it multiplies without a multiply instruction

This Lab Will Cover:

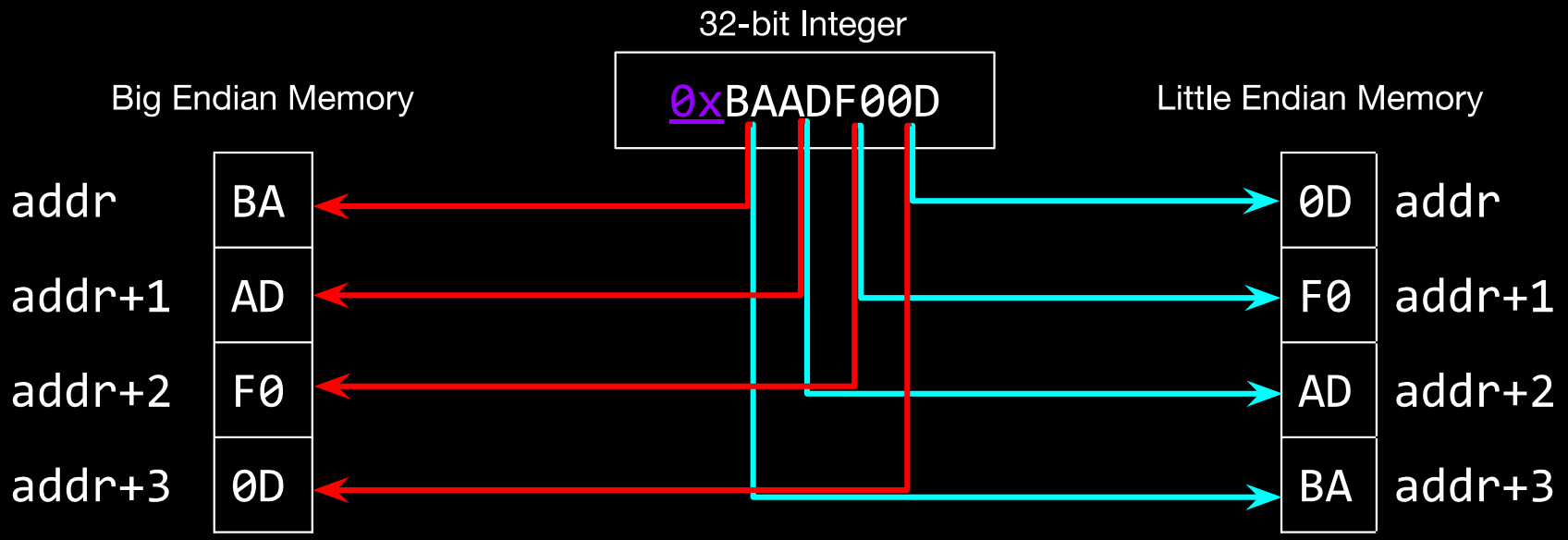
1. ARM LEGv8 Assembly
2. **Memory and Struct Alignment**

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.

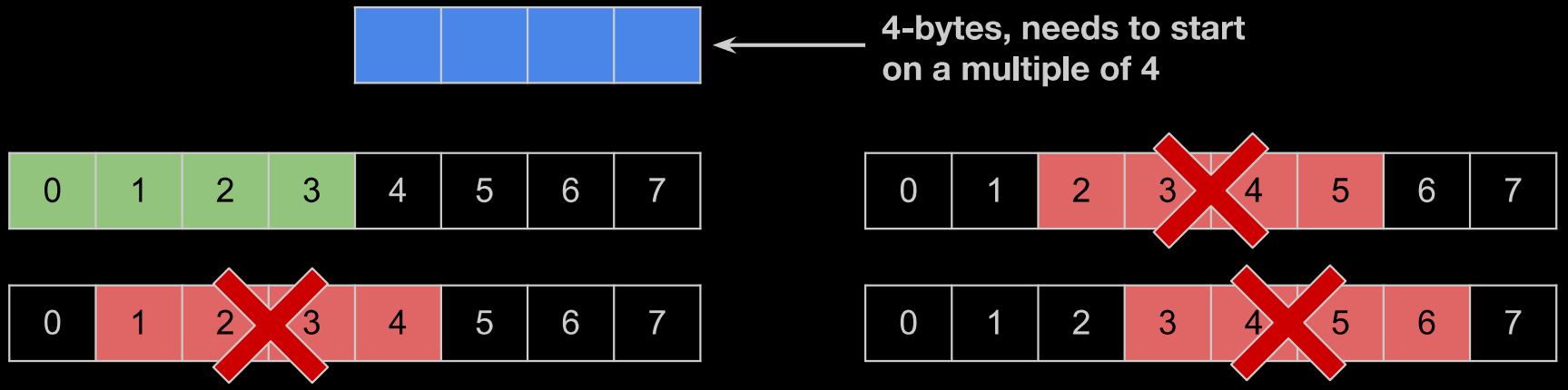
Little Endian builds the word so the **largest** address is the **MSB.**

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!

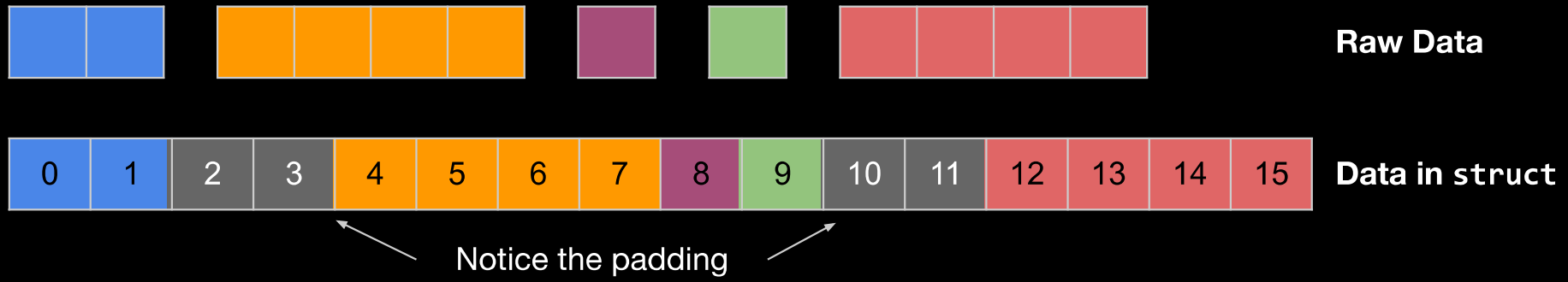
UNIVERSITY OF MICHIGAN

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
```

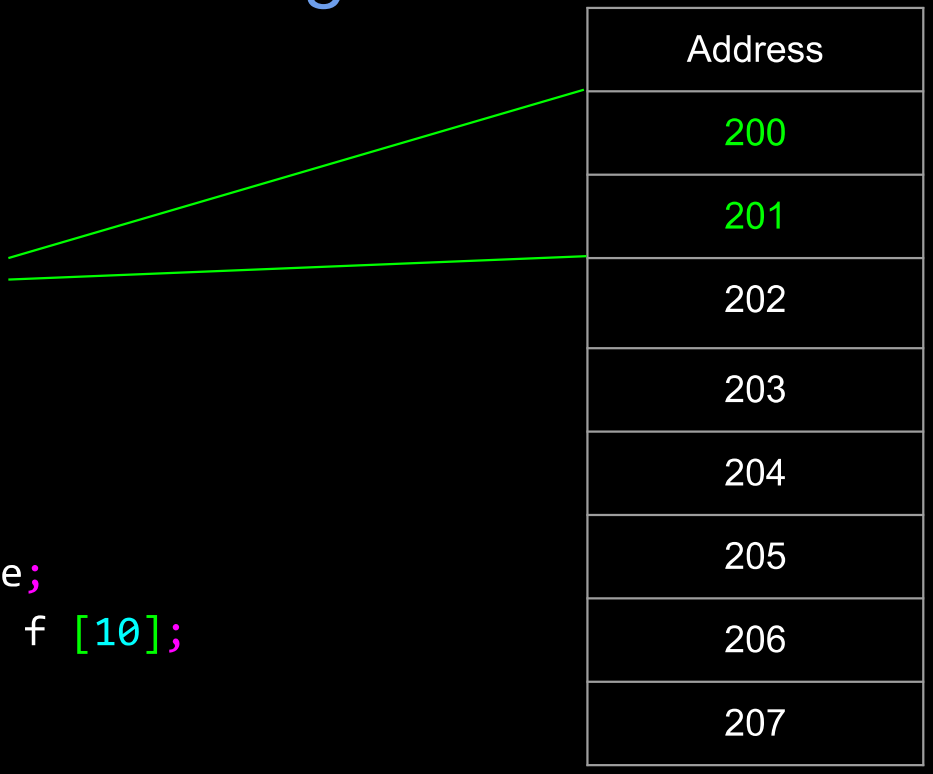
Visual Walkthrough

```
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

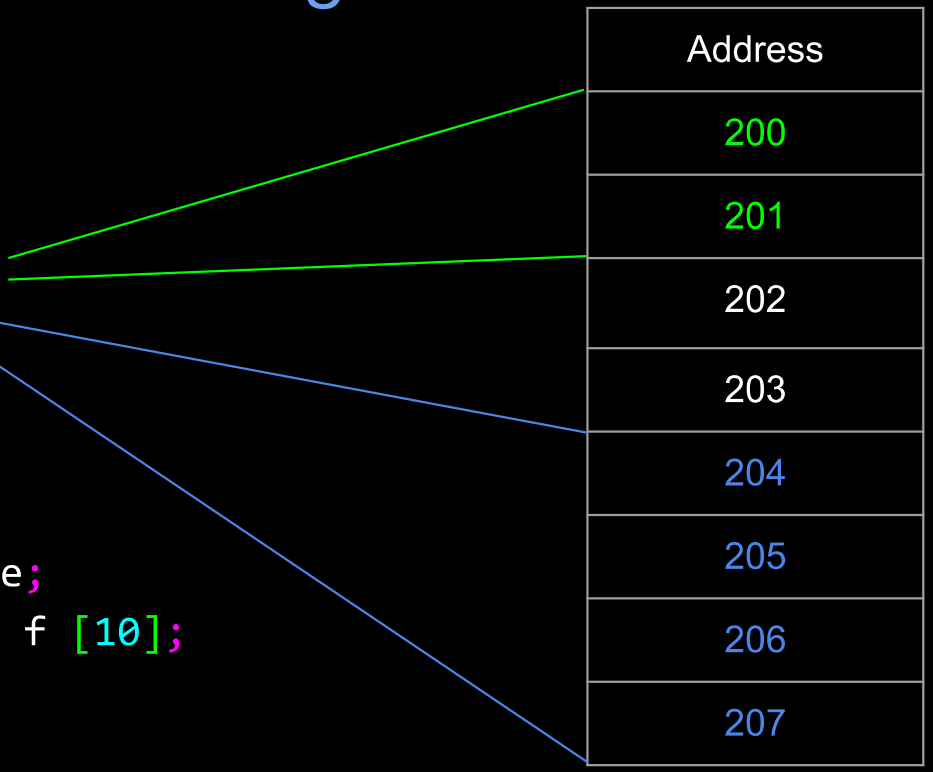
Visual Walkthrough

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



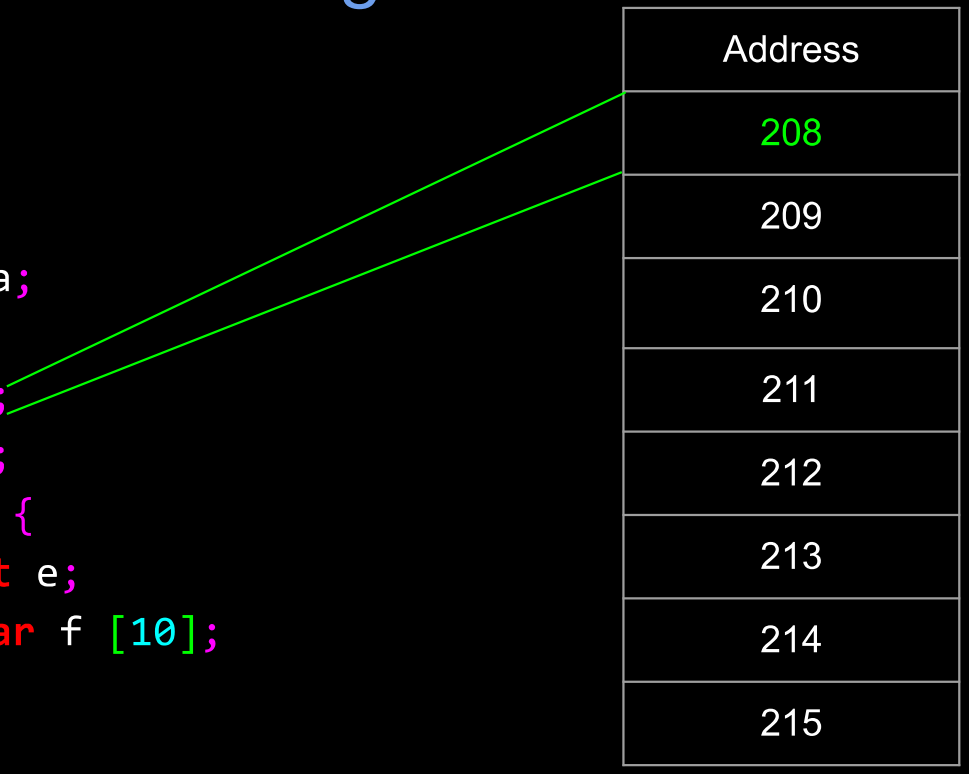
Visual Walkthrough

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



Visual Walkthrough

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



Visual Walkthrough

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

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

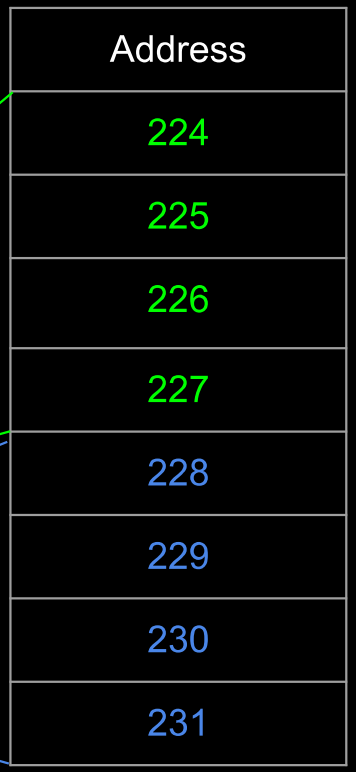
int e must start where struct g starts

Visual Walkthrough

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

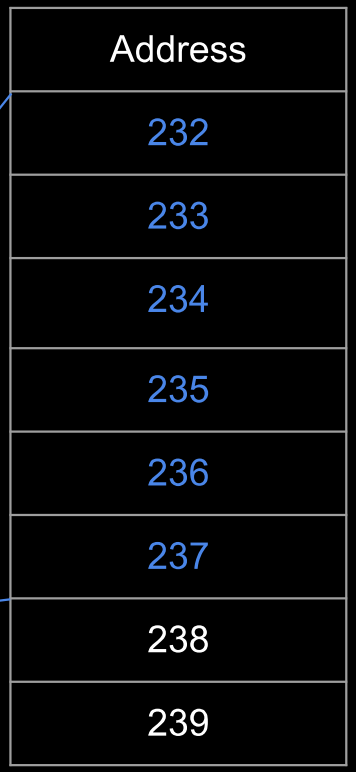
Visual Walkthrough

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



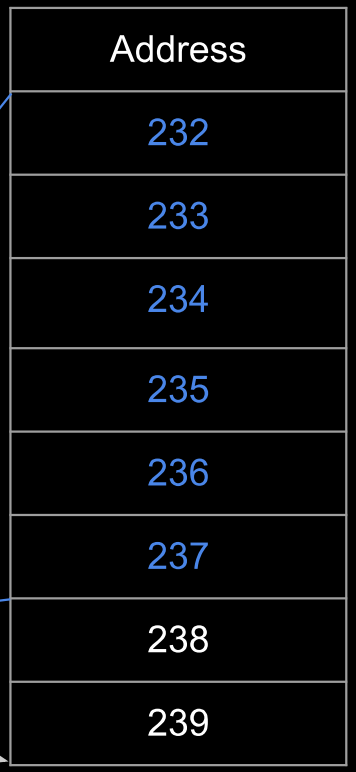
Visual Walkthrough

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



Visual Walkthrough

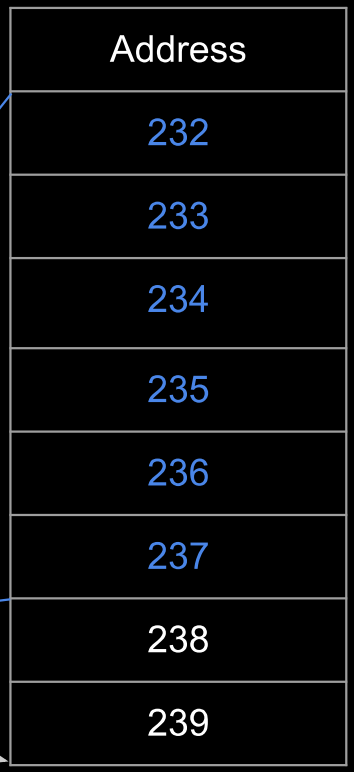
```
struct {
    short a;
    int b;
    char c;
    int* d;
    struct {
        int e;
        char f [10];
    } g; //Align with e
} example;//Align with d
```



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
```



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)

UNIVERSITY OF MICHIGAN

- 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.*

Problem 3

Complete the C to ARM translation

C

```
struct pencil_pack{
    char pack_id[20]; //20B
    int64_t num_penc; //8B

};

int32_t
check_pencil_arr(
          struct pencil_pack data[]) {
  int32_t good_count = 0;

  for(int32_t i = 0; i < 512; ++i){
     if(data[i].num_penc < 20){

          data[i].num_penc *= 4;

     } else {
        ++good_count;
     }
  }

  return good_count;
}
```

ARM

```
func:
     MOVI    X2,   #_____
     MOVI    X3,   #0x00
loop:
     CMPI    X3,   #512
     B.EQ    _____
     LSL     X4,   ____,   #_____
     ADD     X4,    X4,    _____
     ____    X5,   [X4,   #_____]
     CMPI    X5    #20

     B.LT    _____
     ADDI    _____,  ____,   #1
     B       ______
smaller:
     ____    X5,    X5,   #_____
     ____    ____    [____,#_____]
forend:
     ADDI    _____  _____ #_____
     ______    ______
end:
```

Extra Example

Putting It All Together: C → ARM Example

We will be writing a basic ARM LEGv8 program that accomplishes the following snippet of C code. Assume the starting address of the array `arr` is stored in register `X0` and index `i` is stored in register `X1`.

```c
struct point {
    int8_t x;   // 1B
    int8_t y;   // 1B
}
```

```c
void foo(point arr[], int i) {
    arr[i].x += 5;
}
```

Is this code correct?

Take some time to examine the assembly code.

```
struct point {
    int8_t x;   // 1B
    int8_t y;   // 1B
}

void foo(point arr[], int i) {
    arr[i].x += 5;
}
```

Starting address of `arr`: X0. Index `i`: X1.

```
foo:
    ADD     X2, X0,  X1
    LDURB   X3, [X2, #0]
    ADDI    X3, X3,  #5
    STURB   X3, [X2, #0]
end:
    ...
```

This won't work!!

Let's see why in the next slide!

Assume i = 2. What goes wrong?

```
struct point {
    int8_t x;  // 1B
    int8_t y;  // 1B
}
void foo(point arr[], int i) {
    arr[i].x += 5;
}
```

Starting address of arr: X0. Index i: X1.

```
foo:
    ADD     X3, X0,  X1
    LDURB   X4, [X3, #0]
    ADDI    X4, X4,  #5
    STURB   X4, [X3, #0]
```

[X0] +[X1] [X3]

+i = +2

We wanted to load arr[2].x!!

Brainstorm what went wrong!

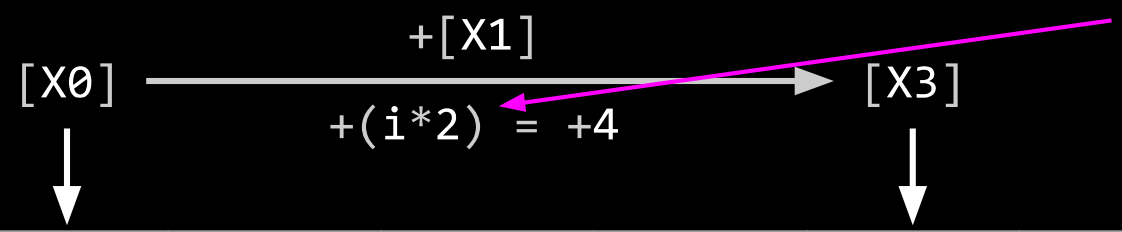
offset	+0	+1	+2	+3	+4	+5	+6	+7
base	arr[0].x	arr[0].y	arr[1].x	arr[1].y	arr[2].x	arr[2].y	arr[3].x	arr[3].y

What should we have done?

```
struct point {
    int8_t x;  // 1B
    int8_t y;  // 1B
}
void foo(point arr[], int i) {
    arr[i].x += 5;
}
```

`point` is 2 bytes and LEGv8 is ***byte-addressable***!

So, to traverse an array of `point`'s, we must traverse by 2 bytes at a time!



What are some ways to multiply by 2?

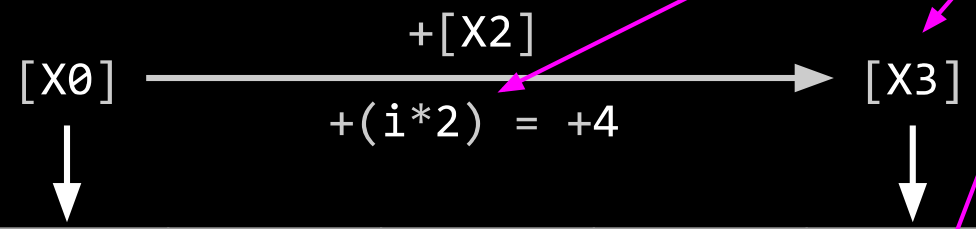
offset	+0	+1	+2	+3	+4	+5	+6	+7
base	arr[0].x	arr[0].y	arr[1].x	arr[1].y	arr[2].x	arr[2].y	arr[3].x	arr[3].y

Fixing our Code - Solution

```
struct point {
    int8_t x;  // 1B
    int8_t y;  // 1B
}
void foo(point arr[], int i) {
    arr[i].x += 5;
}
```

Starting address of `arr`: `X0`. Index `i`: `X1`.

```
foo:
    LSL     X2, X1,  #1
    ADD     X3, X0,  X2
    LDURB   X4, [X3, #0]
    ADDI    X4, X4,  #5
    STURB   X4, [X3, #0]
```



Final Note: Make sure to choose the right LD/ST size for the data you are handling!

offset	+0	+1	+2	+3	+4	+5	+6	+7
base	arr[0].x	arr[0].y	arr[1].x	arr[1].y	arr[2].x	arr[2].y	arr[3].x	arr[3].y

UNIVERSITY OF MICHIGAN

EECS 370

Lab 3:
ARM Assembly