L5_1 Assembly_Data-Layout

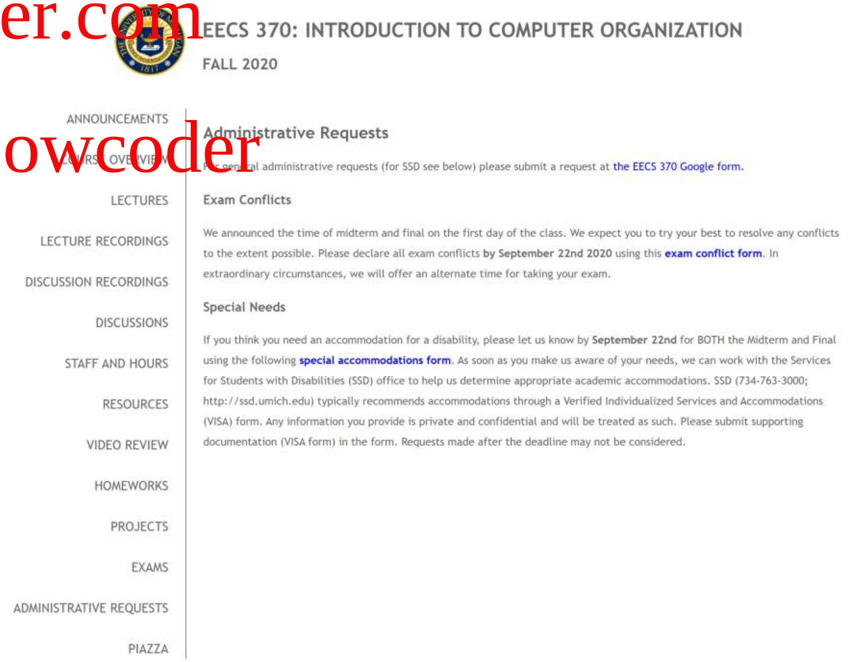
EECS 370 – Introduction to Computer Organization – Fall 2020

Learning Objectives

- Understand mapping of C-code data structures (`struct`) to data layout in memory (e.g., stack)

Reminders

- P1a due Thursday!!!
- Midterm exam 10/20
  - Complete request for alternate or submit requests for accommodations by 9/22
- Drop by office hours
  - PrOffice hours on the calendar



EECS 370: INTRODUCTION TO COMPUTER ORGANIZATION

FALL 2020

Administrative Requests

For general administrative requests (for SSD see below) please submit a request at the EECS 370 Google form.

Exam Conflicts

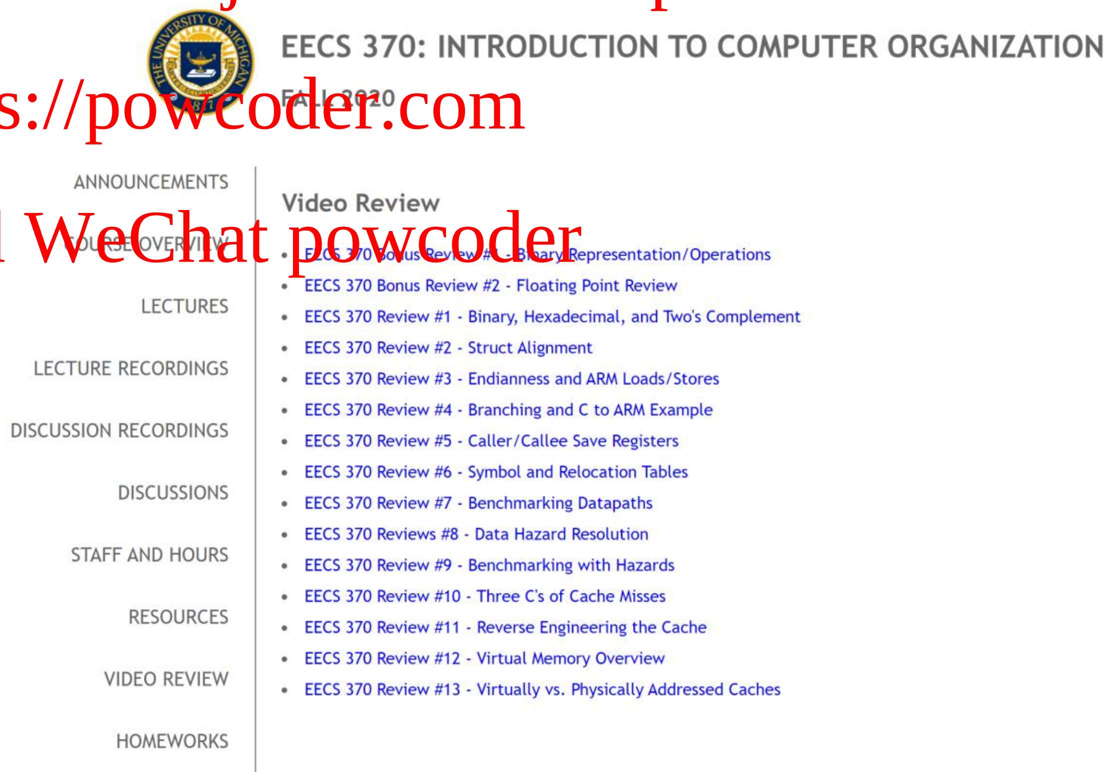
We announced the time of midterm and final on the first day of the class. We expect you to try your best to resolve any conflicts to the extent possible. Please declare all exam conflicts **by September 22nd 2020** using this **exam conflict form**. In extraordinary circumstances, we will offer an alternate time for taking your exam.

Special Needs

If you think you need an accommodation for a disability, please let us know by **September 22nd** for BOTH the Midterm and Final using the following **special accommodations form**. As soon as you make us aware of your needs, we can work with the Services for Students with Disabilities (SSD) office to help us determine appropriate academic accommodations. SSD (734-763-3000; http://ssd.umich.edu) typically recommends accommodations through a Verified Individualized Services and Accommodations (VISA) form. Any information you provide is private and confidential and will be treated as such. Please submit supporting documentation (VISA form) in the form. Requests made after the deadline may not be considered.

Resources

- Video reviews of many topics!
  - https://www.eecs.umich.edu/courses/eecs370/eecs370.f20/video_reviews/



EECS 370: INTRODUCTION TO COMPUTER ORGANIZATION

FALL 2020

Video Review

- EECS 370 Bonus Review #1 - Binary Representation/Operations
- EECS 370 Bonus Review #2 - Floating Point Review
- EECS 370 Review #1 - Binary, Hexadecimal, and Two's Complement
- EECS 370 Review #2 - Struct Alignment
- EECS 370 Review #3 - Endianness and ARM Loads/Stores
- EECS 370 Review #4 - Branching and C to ARM Example
- EECS 370 Review #5 - Caller/Callee Save Registers
- EECS 370 Review #6 - Symbol and Relocation Tables
- EECS 370 Review #7 - Benchmarking Datapaths
- EECS 370 Reviews #8 - Data Hazard Resolution
- EECS 370 Review #9 - Benchmarking with Hazards
- EECS 370 Review #10 - Three C's of Cache Misses
- EECS 370 Review #11 - Reverse Engineering the Cache
- EECS 370 Review #12 - Virtual Memory Overview
- EECS 370 Review #13 - Virtually vs. Physically Addressed Caches

Converting C to assembly – Example #2

ARM ISA

REVIEW!

Write ARM assembly code for the following C expression (assume an `int` is 4 bytes, `unsigned char` is 1 byte)

Register to variable mapping

X1→*pointer to y*

C-code instructions

```
struct { int a; unsigned char b, c; } y;
y.a  = y.b + y.c;
```

ARM LEGv8

```
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
```

See supplemental video for detailed explanation

How do you determine offsets for struct sub-fields? *THIS lecture will detail*

Calculating Load/Store Addresses

Data Layout

Problem: Calculate the total amount of memory needed for the struct instance x
Assume data memory starts at address `1000`

Datatype	size (bytes)
`short`	2
`char`	1
`int`	4
`double`	8

C-code

```
short a[100];
char b;
int c;
double d;
short e;
struct {
  char f;
  int g[1];
  char h;
} x;
```

Calculating Load/Store Addresses

Data Layout

Problem: Calculate the total amount of memory needed for the struct instance x
Assume data memory starts at address 1000

Datatype	size (bytes)
short	2
char	1
int	4
double	8

C-code

```
short a[100];
char b;
int c;
double d;
short e;
struct {
  char f;
  int g[1];
  char h;
} x;
```

a = 2 bytes * 100 = 200 B 1000 → 1199

b = 1 B : 1200 → 1200

c = 4 B : 1201 → 1204

d = 8 B : 1205 → 1212

e = 2 B : 1213 → 1214

struct x : f : 1 byte ⇒ 1215 → 1215

g[1] : 4 bytes 1216 → 1219

h : 1 B 1220 → 1220

NO!

1000 → 1220 = ~~221 B~~ ???

Calculating Load/Store Addresses

Data Layout

Problem: Calculate the total amount of memory needed for the struct instance x
Assume data memory starts at address 1000

Datatype	size (bytes)
short	2
char	1
int	4
double	8

C-code

```
short a[100];
char b;
int c;
double d;
short e;
struct {
  char f;
  int g[1];
  char h;
} x;
```

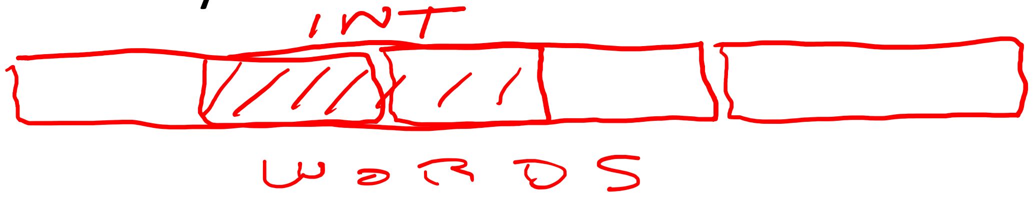
```
Solution???
a = 2 bytes * 100 = 200
b = 1 byte
c = 4 bytes
d = 8 bytes
e = 2 bytes
x = 1 + 4 + 1 = 6 bytes

Total: 221 bytes???
```

Correct or incorrect?

Memory layout of variables

Data Layout

- Most modern ISAs require that data be aligned.
- What do we mean by alignment in this context?
  - An N-byte variable must start at an address A, such that `(A % N) == 0`
- “Golden” rule – Address of a variable is aligned based on the size of the variable
  - **char** is byte aligned (any address is fine)
  - **short** is half-word (H) aligned (LSBit of address must be `0`)
  - **int** is word aligned (W) (2 LSBit’s of addr must be `0`)
- This greatly simplifies hardware needed for loads and stores
  - Otherwise, multiple memory accesses need to be used to access one piece of data



Structure Alignment

Data Layout

- Each field is laid out in the order it is declared using the Golden Rule for alignment

- Identify largest (primitive) field
  - Starting address of overall struct is aligned based on the largest field
  - Size of overall struct is a multiple of the largest field
  - Reason for this is so we can have an array of structs
    - Guarantees that each instance of struct is aligned the same way

Structure Alignment - Example

Data Layout

Problem: What boundary should this `struct` be aligned to?
What is the total size of the `struct`?

Datatype	size (bytes)
`short`	2
`char`	1
`int`	4
`double`	8

C-code

```
struct {
  char w;
  int x[3]
  char y;
  short z;
} s;
```

Structure Alignment - Example

Data Layout

Problem: What boundary should this `struct` be aligned to?
What is the total size of the `struct`?

Datatype	size (bytes)
short	2
char	1
int	4
double	8

C-code

```
struct {
   char w;
   int x[3]
   char y;
   short z;
} s;
```

start address 1000

w : 1 B 1000 → 1000

PADDING 3B 1001 → 1003

x[3] : ~~1001~~ → 1004 → 1015

y : 1 B 1016 → 1016

~~z : 2 B 1017~~

Padding : 1017 → 1017 1 B

z : 2 B 1018 → 1019

S 1000 → 1019 : 20 B

Structure Alignment - Example

Data Layout

Problem: What boundary should this `struct` be aligned to?
What is the total size of the `struct`?

Datatype	size (bytes)
`short`	2
`char`	1
`int`	4
`double`	8

C-code

```
struct {
  char w;
  int x[3]
  char y;
  short z;
} s;
```

Largest field is 4 bytes (`int`), therefore:

- `struct` size will be multiple of 4
- `struct` starting address is word aligned, since a word is 4 bytes

Assume struct starts at address `1000`, what is the data layout of the struct?

Structure Alignment - Example

Data Layout

Problem: What boundary should this `struct` be aligned to?
What is the total size of the `struct`?

Datatype	size (bytes)
`short`	2
`char`	1
`int`	4
`double`	8

C-code

```
struct {
  char w;
  int x[3]
  char y;
  short z;
} s;
```

Largest field is 4 bytes (`int`), therefore:

- `struct` size will be multiple of 4
- `struct` starting address is word aligned, since a word is 4 bytes

Assume struct starts at address `1000`, what is the data layout of the struct?

```
char w -> 1000
// padding 1001-1003
x[0] -> 1004-1007
x[1] -> 1008-1011
x[2] -> 1012-1015
char y -> 1016
// padding 1017
short z -> 1018-1019
```

start: `1000`

end: `1019`
Total size = `20` bytes

Why padding?
"Golden" rule – Address of a variable is aligned based on the size of the variable

Calculating Load/Store Addresses – 2nd Try

Data Layout

Problem: Calculate the total amount of memory needed for the declarations. Assume data memory starts at address `100`

C-code

```
short a[100];
char b;
int c;
double d;
short e;
struct {
  char f;
  int g[1];
  char h;
} x;
```

Datatype	size (bytes)
short	2
char	1
int	4
double	8

An N-byte variable must start at an address A, such that `(A % N) == 0`

Calculating Load/Store Addresses – 2nd Try

Data Layout

Problem: Calculate the total amount of memory needed for the declarations. Assume data memory starts at address 100

Datatype	size (bytes)
short	2
char	1
int	4
double	8

C-code

```
short a[100];
char b;
int c;
double d;
short e;
struct {
   char f;
   int g[1];
   char h;
} x;
```

a : 2B × 100 = 200B 100 → 299

b : 1B 300 → 300

Padding 301 → 303

c : 4B 304 → 307

Padding 308 → 311

d : 8B 312 → 319

e : 2B 320 → 321

Padding : 2B 322 → 323

x { f : 1B 324 → 324

Padding : 3B 325 → 327

g : 4B 328 → 331

h : 1B 332 → 332

Padding : 3B 333 → 335

12B

100 → 335 : 236B

An N-byte variable must start at an address A, such that (A % N) == 0

Calculating Load/Store Addresses – 2nd Try

Data Layout

Problem: Calculate the total amount of memory needed for the declarations.
Assume data memory starts at address 100

Datatype	size (bytes)
short	2
char	1
int	4
double	8

C-code

```
short a[100];
char b;
int c;
double d;
short e;
struct {
  char f;
  int g[1];
  char h;
} x;
```

C-code	Bytes	Start	End	Notes
short a[100];	200	100	299	
char b;	1	300	300	
	3	301	303	padding
int c;	4	304	307	
	4	308	311	padding
double d;	8	312	319	
short e;	2	320	321	
	2	322	323	padding
struct {	12	324	335	largest field: 4 bytes
char f;	1	324	324	
	3	325	327	padding
int g[1];	4	328	331	
char h;	1	332	332	
	3	333	335	padding
} x;	12	324	335	

An N-byte variable must start at an address A, such that
`(A % N) == 0`

Total size: 236 bytes

Pause

The next example is a review of Lecture 5 worksheet 1.

Pause, complete the worksheet, then proceed.

Example 2

Worksheet #1

Problem: Calculate the total amount of memory needed for the declarations.
Assume data memory starts at address **200**

C-code

```
int a;
struct {
  double b;
  char c;
  int d;
} x;
char *f;
short g[20];
```

Datatype	size (bytes)
short	2
char	1
int	4
double	8
address	4

Example 2

Worksheet #1

Problem: Calculate the total amount of memory needed for the declarations. Assume data memory starts at address **200**

C-code

```
int a;
struct {
  double b;
  char c;
  int d;
} x;
char *f;
short g[20];
```

C-code	Bytes	Start	End	Notes
int a;	4	200	203	
	4	204	207	padding
struct {	16	208	223	largest field: 8 bytes
double b;	8	208	215	
char c;	1	216	216	
	3	217	219	padding
int d;	4	220	223	
} x;	16	208	223	
char *f;	8	224	231	
short g[20];	40	232	271	
TOTAL:	72	200	271	

Datatype	size (bytes)
short	2
char	1
int	4
double	8
address	8

Data Layout – Why?

- Does gcc (or another compiler) reorder variables in memory to avoid padding?
- No, a compiler will not optimize data layout to remove padding.
- C99 standard prohibits this
  - Memory is laid out in order of declaration for structs.
- gcc has implemented an option for this, then later removed it
- The programmer (i.e., you) are expected to manage data layout of variables for your program and structs.
- For a start: order fields in struct by datatype size, smallest first

Logistics

- There are 3 videos for lecture 5
  - L5_1 – Assembly_Data-Layout
  - L5_2 – Assembly_Flow-Control
  - L5_3 – C-to-Assembly_Examples
- There are two worksheets for lecture 5
  1. **Data Layout –** you can do this now
  2. C to Assembly

L5_2 Assembly _ Flow-Control

EECS 370 – Introduction to Computer Organization – Fall 2020

Learning Objectives

- Recognize the set of branching instructions for ARM ISA and be able to describe the operations and operands for instructions
  - LEGv8 subset

- Understand mapping of complex C-code branching instructions into corresponding assembly code instructions

ARM/LEGv8 Sequencing Instructions

ARM Branching

PC + 4 B

- Sequencing instructions change the flow of instructions that are executed
  - This is achieved by modifying the program counter (PC)
- Unconditional branches are the most straightforward they ALWAYS change the PC and thus "jump" to another instruction out of the usual sequence
- Conditional branches

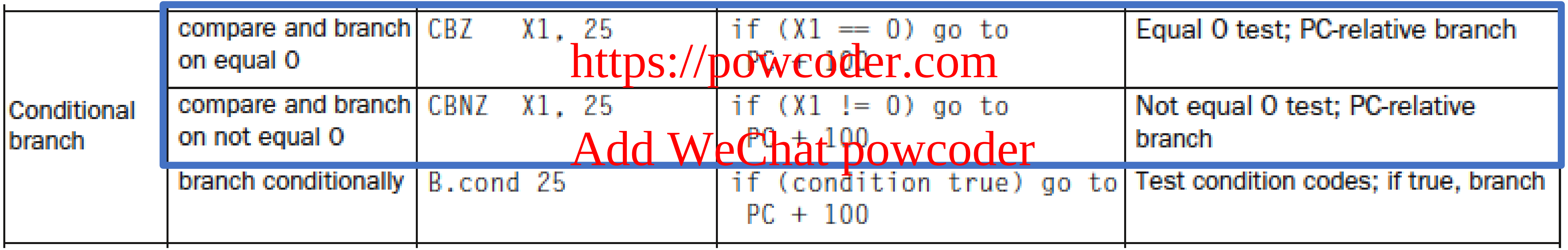
```
if (condition_test) goto target_address
```

- *condition_test* examines the four flags from the processor status word (SPSR)
- *target_address* is the 19-bit signed word displacement from current PC

LEGv8 Conditional Instructions

ARM Branching

- Two varieties of conditional branches
  1. One type compares a register to see if it is equal to zero.
  2. Another type checks the condition codes set in the status register.



Conditional branch	compare and branch on equal 0	`CBZ X1, 25`	`if (X1 == 0) go to PC + 100`	Equal 0 test; PC-relative branch
	compare and branch on not equal 0	`CBNZ X1, 25`	`if (X1 != 0) go to PC + 100`	Not equal 0 test; PC-relative branch
	branch conditionally	`B.cond 25`	`if (condition true) go to PC + 100`	Test condition codes; if true, branch

- Let us look at the first type: CBZ and CBNZ
  - CBZ – Conditional Branch if Zero
  - CBNZ – Conditional Branch if Not Zero

LEGv8 Compare and Branch Instructions

ARM Branching

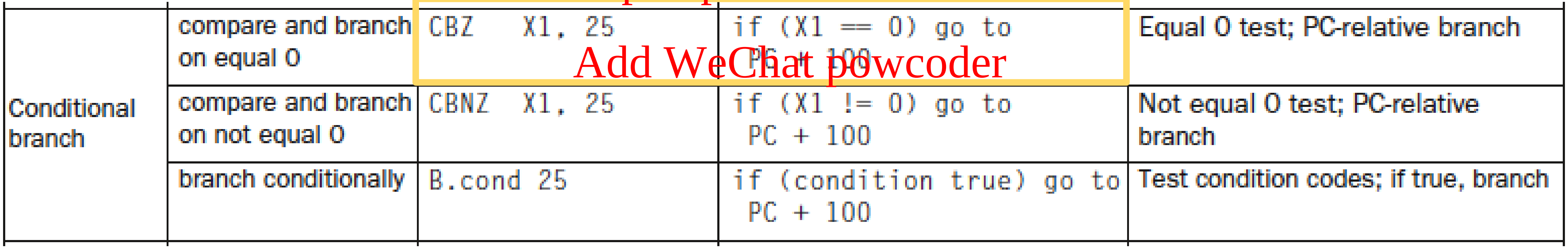
- CBZ/CBNZ: test a register against zero and branch to a PC relative address
  - The relative address is a 19-bit signed integer—the number of instructions. Recall instructions are 32 bits of 4 bytes

ARM LEGv8	Description
`CBNZ X3, foo`	• if X3 does not equal `0`, then branch to label `foo`
`CBNZ X3, 25`	• 25 is an offset from the PC of the current instruction (CBNZ)

LEGv8 Compare and Branch Instructions

ARM Branching

- CBZ/CBNZ: test a register against zero and branch to a PC relative address
  - The relative address is a 19-bit signed integer—the number of instructions. Recall instructions are 32 bits of 4 bytes



Conditional branch	compare and branch on equal 0	`CBZ X1, 25`	`if (X1 == 0) go to PC + 100`	Equal 0 test; PC-relative branch
	compare and branch on not equal 0	`CBNZ X1, 25`	`if (X1 != 0) go to PC + 100`	Not equal 0 test; PC-relative branch
	branch conditionally	`B.cond 25`	`if (condition true) go to PC + 100`	Test condition codes; if true, branch

Why does 25 in the table result in `PC + 100`?

Offset is # of instructions (words)

Conditional Branch Offset Example

ARM Branching

Problem: Calculate the numerical offset for the CBNZ instruction.

ARM LEGv8

```
loop:    ADDI X3, X3, #1
         SUBI X4, X4, #1
         ADD  X5, X3, X4
         CBNZ X5, loop
```

#-3

Conditional Branch Offset Example

ARM Branching

Problem: Calculate the numerical offset for the CBNZ instruction.

ARM LEGv8

```
loop:    ADDI X3, X3, #1
         SUBI X4, X4, #1
         ADD  X5, X3, X4
         CBNZ X5, loop
```

Answer: -3

Offset field: 19-bit, signed

111 1111 1111 1111 1101

The assembler will calculate the offset

If any instructions are added or removed while writing code, using a label saves from recalculating the offset

Conditional Branch Offset Example

ARM Branching

How is the branch target address calculated?	ARM LEGv8
	`CBNZ X5, #-3`

Conditional Branch Offset Example

ARM Branching

How is the branch target address calculated?	ARM LEGv8
0x0A...	CBNZ X5, #-3

64-bit address

signed 19-bit

$$\underbrace{11 \ldots 111\ 1111\ 1111\ 1111\ 1111\ 1101\ \boxed{00}}_{64\text{ bits}} \ (-3)$$

LSL → *4

21 bits

64 bits

+

PC (address of CBNZ X5, #-3)

PC = NEXT ADDRESS

Conditional Branch Offset Example

ARM Branching

How is the branch target address calculated?	ARM LEGv8
	`CBNZ X5, #-3`

1. Offset field: 19 bits (-3 decimal)

```
111 1111 1111 1111 1101
```

2. Append two zeros

```
1 1111 1111 1111 1111 0100
```

3. Sign extend to 64 bits

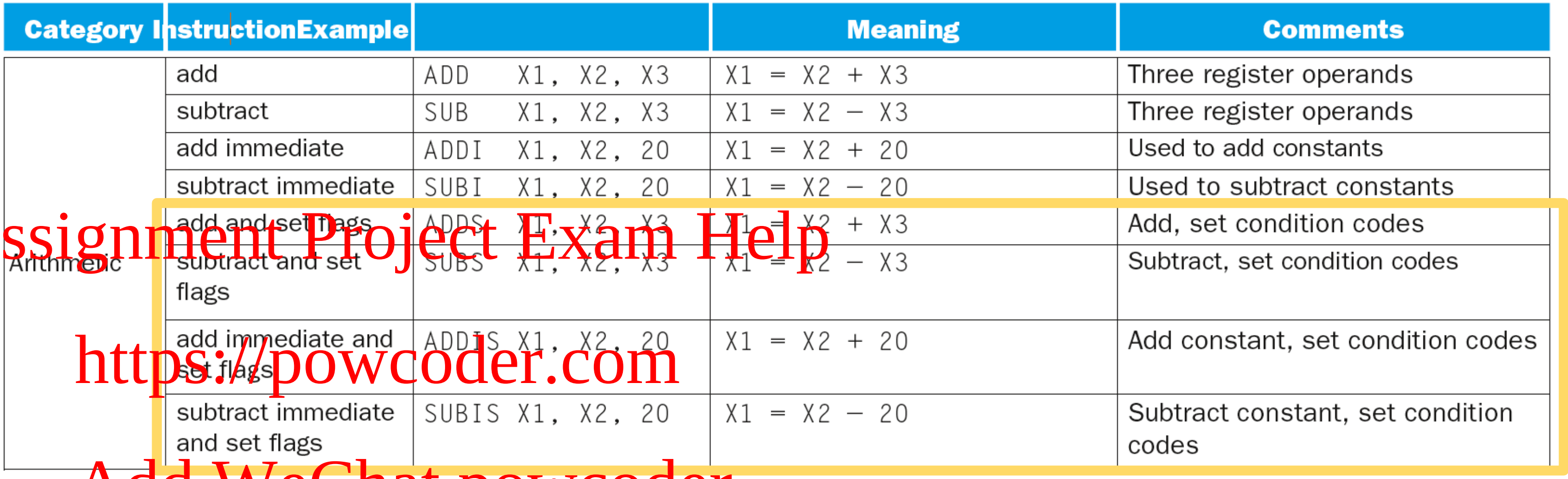
```
1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 0100
```

4. Add offset to PC of CBNZ instruction

LEGv8 Conditional Instructions Using FLAGS

ARM Branching

- FLAGS: **NZVC** record the results of (arithmetic) operations **N**egative, **Z**ero, o**V**erflow, **C**arry—not present in LC-2K
- We explicitly set them using the “Set” modification to ADD/SUB etc.



Category	InstructionExample		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

FLAGS: BINARY, SET OR UNSET

ARM LEGv8	Description
ADDS X1, X2, X3	Causes the 4 flag bits to be set accordingly as the outcome is negative, zero, overflows, or generates a carry bit

LEGv8 Condition Codes

ARM Branching

- In LEGv8 only ADDS / SUBS / ADDIS / SUBIS / CMP /CMPI set the condition codes FLAGs or condition codes in PSR—the program status register

- Four primary condition codes evaluated:
  - N – set if the result is negative (i.e., bit 63 is non-zero)
  - Z – set if the result is zero (i.e., all 64 bits are zero)
  - C – set if last addition/subtraction had a carry/borrow out of bit 63
  - V – set if the last addition/subtraction produced an overflow (e.g., two negative numbers added together produce a positive result)

- Do not worry about the C and V bits *per se*. They are important but are tricky to understand.
  - Instead we will just be using branches based on these results for signed numbers which is a lot easier to deal with.

Conditional Branches

- CMP instruction lets you compare two registers, set NZVC flags
  - Could also use ADDS etc.

- `B.`*`condition`* lets you branch based on the flags set by CMP (ADDS, etc.)

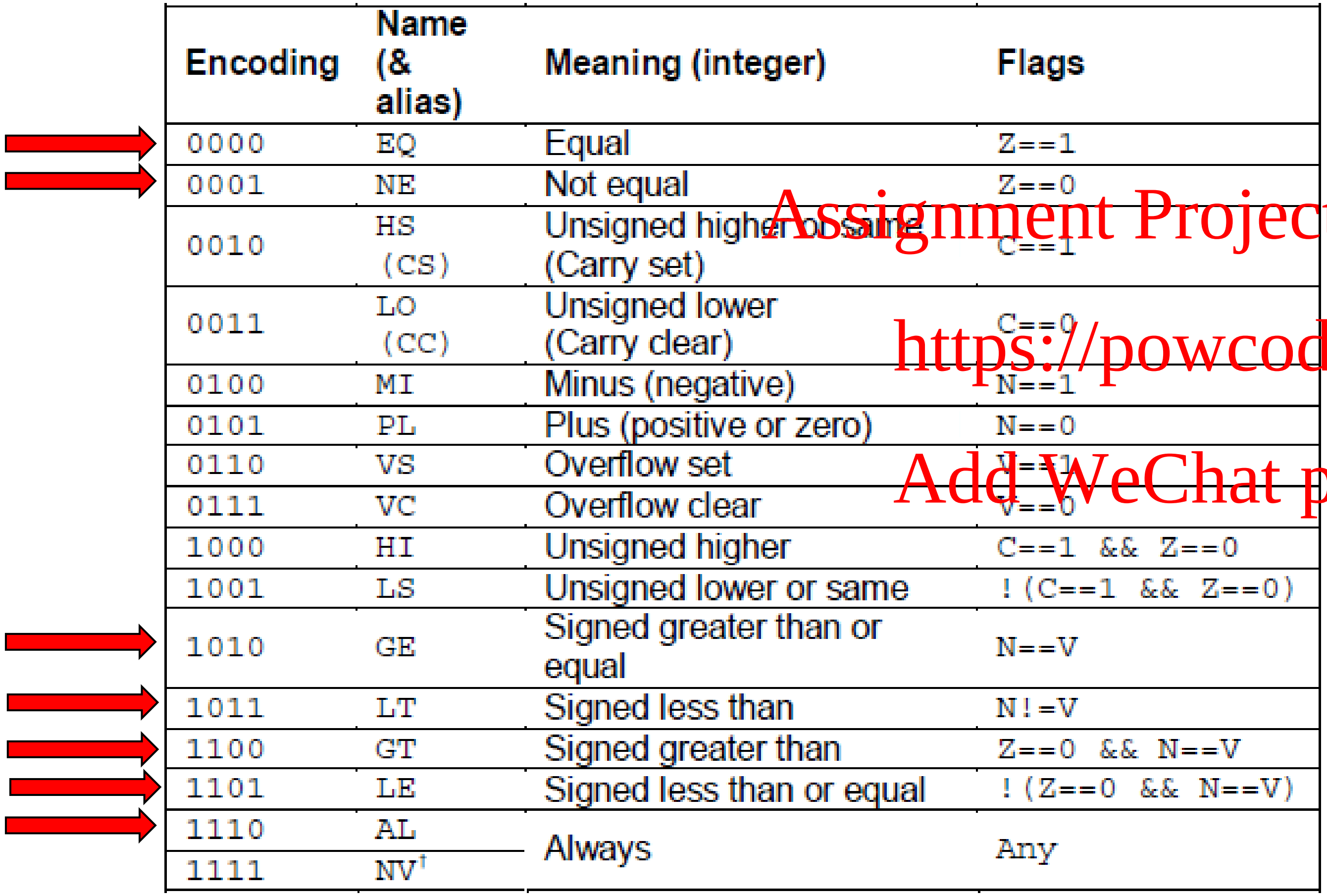
ARM LEGv8	Description

`CMP X1, X2`
`B.GT label1`
	Branches to `label1` if value in register X1 greater than value in register X2

- What is the set of conditions for `B.`*`condition`* ?

LEGv8 Conditional Instructions

ARM Branching



	Encoding	Name (& alias)	Meaning (integer)	Flags
➡	0000	EQ	Equal	Z==1
➡	0001	NE	Not equal	Z==0
	0010	HS (CS)	Unsigned higher or same (Carry set)	C==1
	0011	LO (CC)	Unsigned lower (Carry clear)	C==0
	0100	MI	Minus (negative)	N==1
	0101	PL	Plus (positive or zero)	N==0
	0110	VS	Overflow set	V==1
	0111	VC	Overflow clear	V==0
	1000	HI	Unsigned higher	C==1 && Z==0
	1001	LS	Unsigned lower or same	!(C==1 && Z==0)
➡	1010	GE	Signed greater than or equal	N==V
➡	1011	LT	Signed less than	N!=V
➡	1100	GT	Signed greater than	Z==0 && N==V
➡	1101	LE	Signed less than or equal	!(Z==0 && N==V)
➡	1110	AL	Always	Any
	1111	NV†		

ARM LEGv8

```
CMP X1, X2
B.GT label1
```

```
CMP X3, X4
B.EQ label2
```

```
CMP X5, X6
B.LE label3
```

You need to know the 7 with red arrows

Branching Far Away

- The underlying philosophy of ISA design and microachitecture in general is to **make the common case fast**
- In the case of branches, you are commonly going to branch to other instructions nearby.
  - In ARMv8, the encoding for the displacement of conditional branches is 19 bits.
  - Having a displacement of 19 bits is usually enough

- BUT what if we need jump to a target (Label) that we cannot get to with a 19-bit displacement from the current PC?

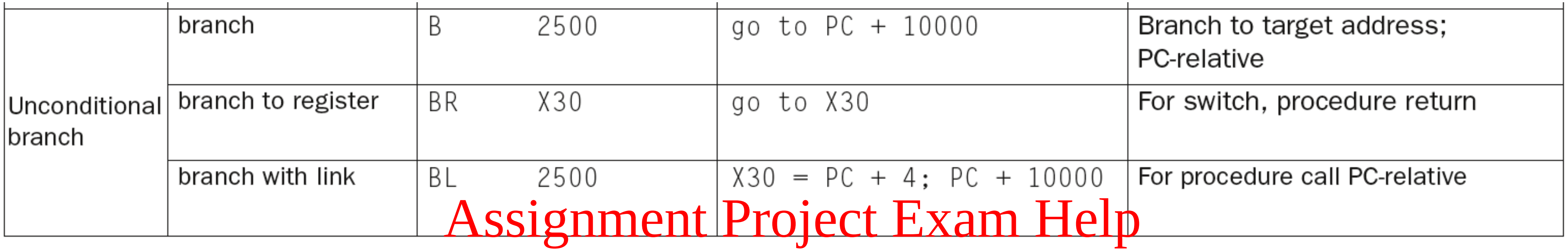
```
CBZ    X15, FarLabel
```

- The assembler is smart enough to replace that with

```
    CBNZ X15, L1
    B    FarLabel
L1:
```

- The simple branch instruction (B) has a 26-bit offset which spans about 64 million instructions!

Unconditional Branching Instructions



Unconditional branch	branch	B 2500	go to PC + 10000	Branch to target address; PC-relative
	branch to register	BR X30	go to X30	For switch, procedure return
	branch with link	BL 2500	X30 = PC + 4; PC + 10000	For procedure call PC-relative

- There are three types of unconditional branches in the LEGv8 ISA.
  - The first **(B)** is the PC relative branch with the 26-bit offset from the last slide.
  - The second **(BR)** jumps to the address contained in a register (X30 above)
  - The third **(BL)** is like our PC relative branch but it does something else.
    - It sets X30 (always) to be the current PC+4 before it branches.
    - Why?
    - Function calls – return to next instruction

Logistics

- There are 3 videos for lecture 5
  - L5_1 – Assembly_Data-Layout
  - L5_2 – Assembly_Flow-Control
  - L5_3 – C-to-Assembly_Examples
- There are two worksheets for lecture 5
  1. Data Layout
  2. C to Assembly – wait until after next video

L5_3 C-to-Assembly_Examples

EECS 370 – Introduction to Computer Organization – Fall 2020

Learning Objectives

- Translate C-code *statements* to ARM assembly code
  - Break down complex C-code branching instructions into a series of assembly operations
  - Map conditions in C to comparison and branch instructions in assembly

Branching - Example

ARM Branching

Problem: Convert the C code to LEGv8 assembly: 1: using labels, 2. without labels

Register to variable mapping

X1→x
X2→y

C-code instructions

```
int x, y;
// update x,y
if (x == y)
   x++;
else
   y++;
```

Branching - Example

ARM Branching

Problem: Convert the C code to LEGv8 assembly: 1: using labels, 2. without labels

Register to variable mapping
X1→x X2→y

C-code instructions

```
int x, y;
// update x,y
if (x == y)
   x++;
else
   y++;
```

B.NE

lbl1

lbl2

```
      CMP    X1, X2
      B.NE   lbl1
      ADDI   X1, X1, #1
      B      lbl2
lbl1: ADDI   X2, X2, #1
lbl2:
```

Branching - Example

ARM Branching

Problem: Convert the C code to LEGv8 assembly: 1: using labels, 2. without labels

Register to variable mapping

X1→x
X2→y

C-code instructions

```
int x, y;
// update x,y
if (x == y)
   x++;
else
   y++;
```

ARM LEGv8 – w/labels

```
      CMP  X1, X2
      B.NE lbl1
      ADDI X1, X1, #1
      B    lbl2
lbl1: ADDI X2, X2, #1
lbl2:
```

Branching - Example

ARM Branching

Problem: Convert the C code to LEGv8 assembly: 1: using labels, 2. without labels

Register to variable mapping

X1→x
X2→y

C-code instructions

```
int x, y;
// update x,y
if (x == y)
   x++;
else
   y++;
```

ARM LEGv8 – w/o labels

```
      CMP  X1, X2
      B.NE 3
      ADDI X1, X1, #1
      B    2
lbl1: ADDI X2, X2, #1
lbl2:
```

Loop - Example

ARM Branching

Problem: Convert the C code to LEGv8 assembly (assume no registers initialized)

Register to variable mapping
X1→i
X2→sum
X4→#10
X5→a[i]
X6→i*8

a is array of long long integers (64 bits, 8 bytes)
Start of a at address `100`,
`sum` starts at address 96

C-code instructions

```
sum = 0;
for (i=0 ; i < 10 ; i++) {
  if (a[i] >= 0) {
    sum += a[i];
  }
}
```

Loop - Example

ARM Branching

Problem: Convert the C code to LEGv8 assembly (assume no registers initialized)

Register to variable mapping
X1→i
X2→sum
X4→#10
X5→a[i]
X6→i*8

a is array of long long integers (64 bits, 8 bytes) Start of a at address 100, sum starts at address 96

C-code instructions

```
sum = 0;
for (i=0 ; i < 10 ; i++) {
   if (a[i] >= 0) {
      sum += a[i];
   }
}
```

loop:

B.LT

endif:

endLoop:

```
       MOV    X2, XZR
       MOV    X1, XZR
       MOVZ   X4, #10
loop:  CMP    X1, X4
       BGE    endloop
       LSL    X6, X1, #3
       LDUR   X5, [X6, #100]
       CMPI   X5, #0
       B.LT   endif
       ADD    X2, X2, X5
       STURW  X2, [XZR, #96]
endif: ADDI   X1, X1, #1
       B      loop
endloop:
```

Loop - Example

ARM Branching

Problem: Convert the C code to LEGv8 assembly (assume no registers initialized)

Register to variable mapping

```
X1→i
X2→sum
X4→#10
X5→a[i]
X6→i*8
```

a is array of long long integers (64 bits, 8 bytes)

Start of a at address 100,

sum starts at address 96

C-code instructions

```
sum = 0;
for (i=0 ; i < 10 ; i++) {
   if (a[i] >= 0) {
      sum += a[i];
   }
}
```

ARM LEGv8

```
          MOV     X1, XZR
          MOV     X2, XZR
          MOVZ    X4, #10
Loop1:    CMP     X1, X4
          B.GE    endLoop
          LSL     X6, X1, #3
          LDUR    X5, [X6, #100]
          CMPI    X5, #0
          B.LT    endif
          ADD     X2, X2, X5
          STURW   X2, [XZR, #96]
endif:    ADDI    X1, X1, #1
          B       Loop1
endLoop:
```

Loop - Example

ARM Branching

Alternate Solution: do-while

Register to variable mapping

X1→i
X2→sum
X4→#10
X5→a[i]
X6→i*8

a is array of long long integers (64 bits, 8 bytes)
Start of a at address 100,
sum starts at address 96

C-code instructions

```
sum = 0;
for (i=0 ; i < 10 ; i++) {
  if (a[i] >= 0) {
    sum += a[i];
  }
}
```

ARM LEGv8

```
       MOV    X1, XZR
       MOV    X2, XZR
       MOVZ   X4, #10
Loop1: LSL    X6, X1, #3
       LDUR   X5, [X6, #100]
       CMPI   X5, #0
       B.LT   endif
       ADD    X2, X2, X5
       STUR  X2, [XZR, #96]
endIf: ADDI   X1, X1, #1
       CMP    X1, X4
       B.LT   Loop1
endLoop:
```

Logistics

- There are 3 videos for lecture 5
  - L5_1 – Assembly_Data-Layout
  - L5_2 – Assembly_Flow-Control
  - L5_3 – C-to-Assembly_Examples
- There are two worksheets for lecture 5
  1. Data Layout
  2. **C to Assembly** – can do this now