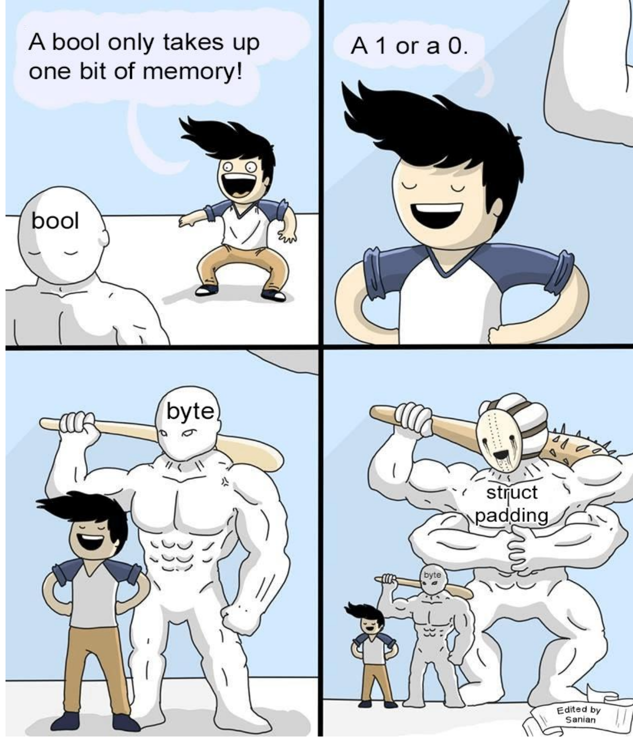


# EECS 370 - Lecture 5

## C to Assembly

## Announcements

- Project 1
  - P1a due Thursday **1/29**
  - P1s due Thursday **2/5**
  - P1m due Thursday **2/5**
- HW1
  - Posted, due Monday **2/2**
- Lab
  - **Lab quiz due TODAY 1/22, 11:55pm**
  - Lab meets this upcoming Friday/Monday/Tuesday
- Midterm:
  - Thursday March 12th, 7-9pm
  - *Exam conflict form is due by January 23rd via the Exam Conflict form* (ed post #17)

**Lecture 3 covered what you need.**

**Each problem is labelled by what lecture you need.**

# Agenda—last lecture

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

ARM ISA

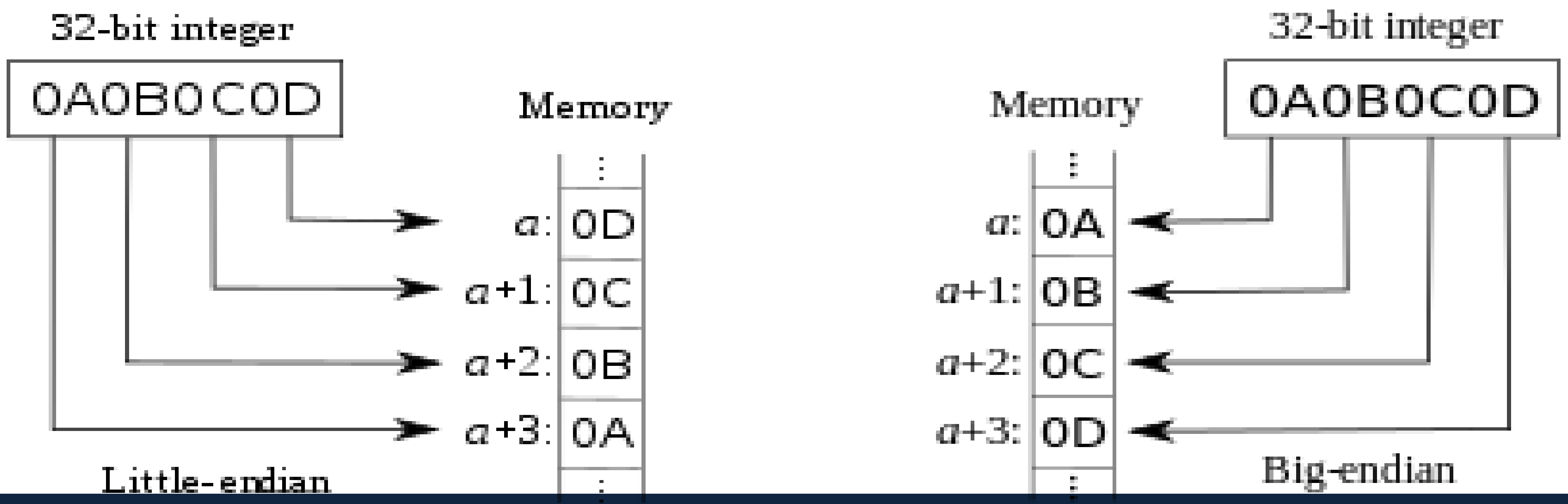
# Load Instruction Sizes

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 or 0xFFFF_FFFF _**F**654_3210 (depends on bit 31) |

# 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)



# Agenda—last lecture

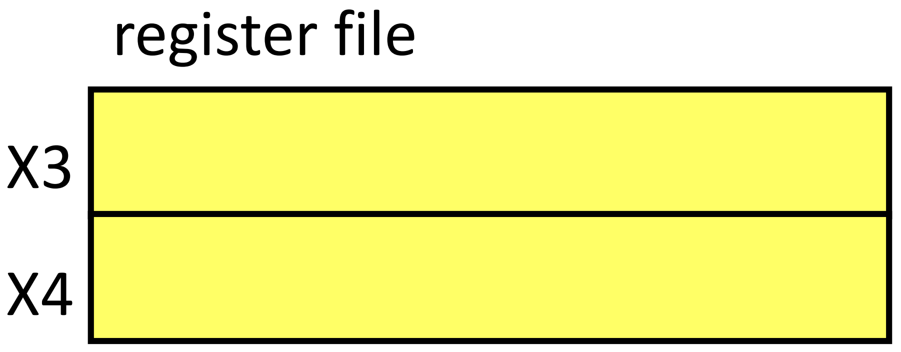
- ARM overview and basic instructions
- Memory instructions
  - Handling multiple data widths
- **Sample Problems**

ARM ISA

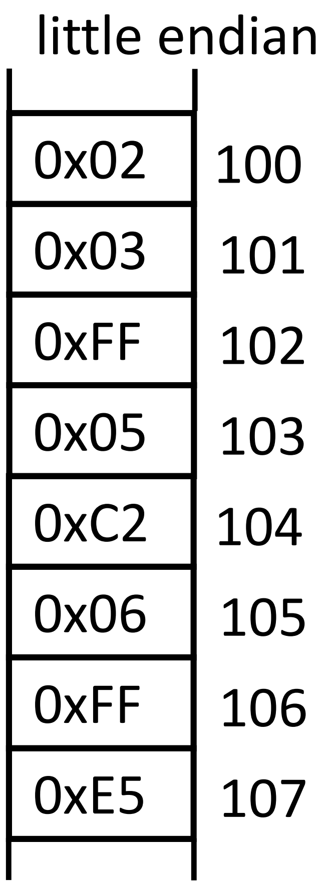
# Example Code Sequence

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]
```



Memory
(each location is 1 byte)

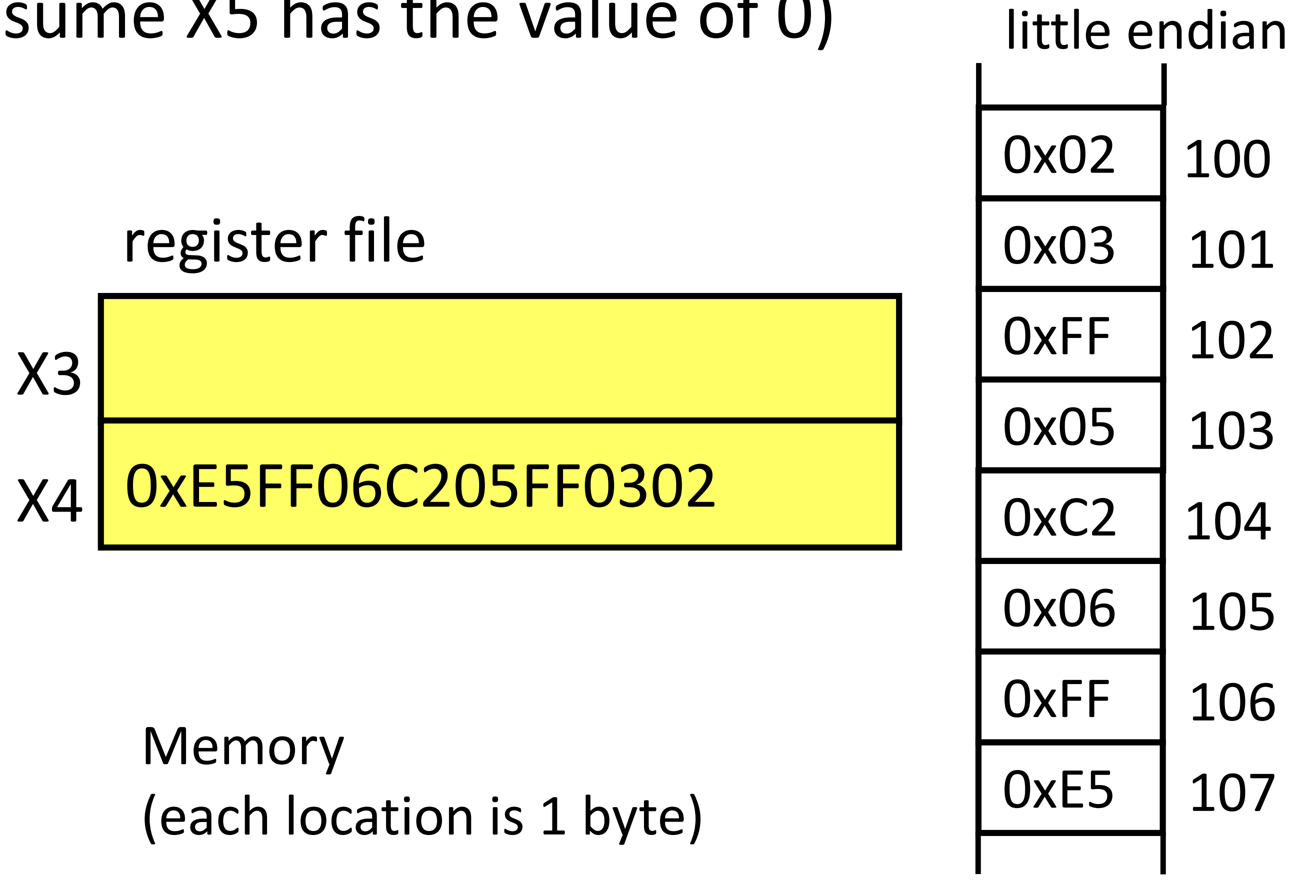


ARM ISA

## Example Code Sequence

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]
```

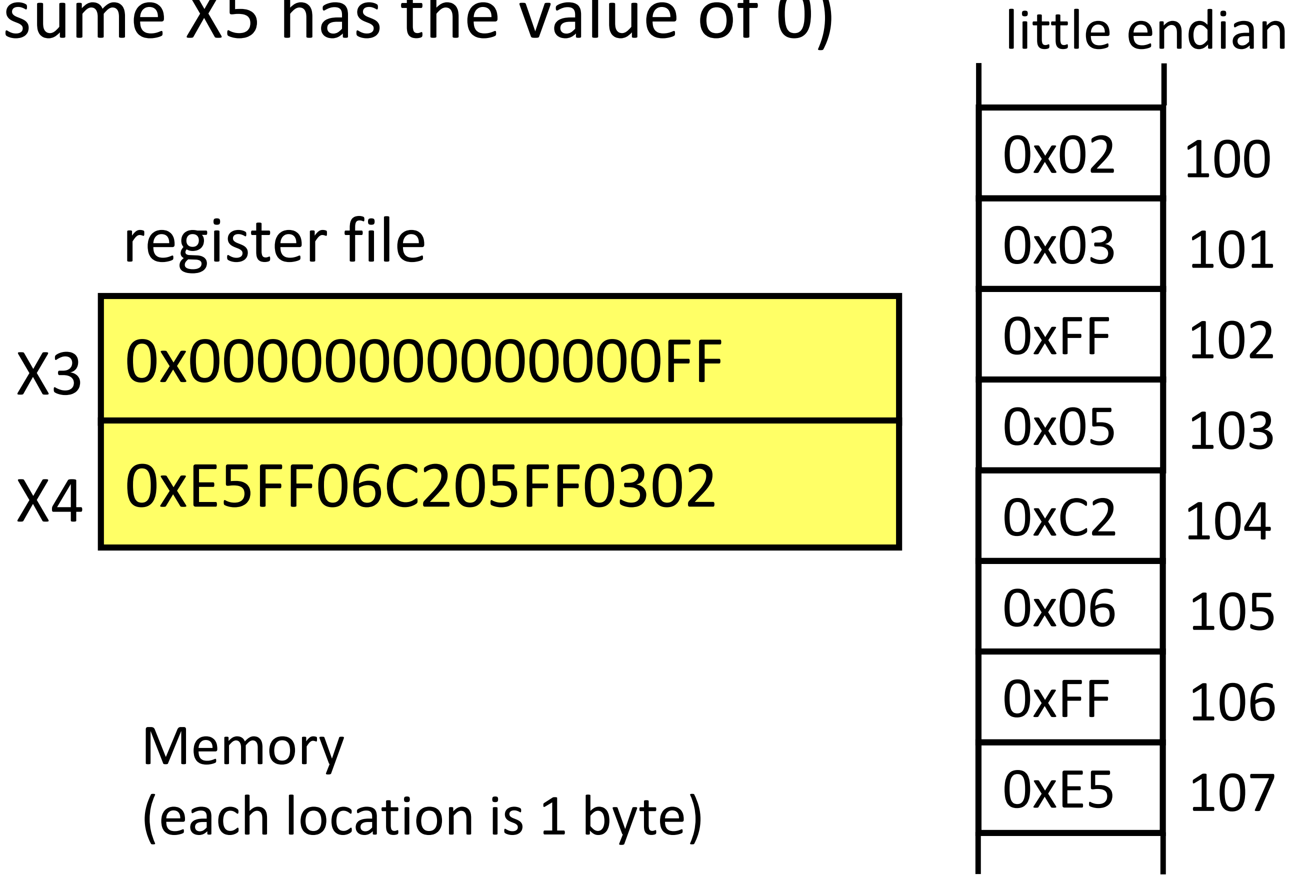


# Example Code Sequence

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]
```



ARM ISA

# Example Code Sequence

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]
```

register file

| | |
|---|---|
| X3 | 0x00000000000000FF |
| X4 | 0xE5FF06C205FF0302 |

Memory
(each location is 1 byte)

little endian

| | |
|---|---|
| 0xFF | 100 |
| 0x00 | 101 |
| 0x00 | 102 |
| 0x00 | 103 |
| 0x00 | 104 |
| 0x00 | 105 |
| 0x00 | 106 |
| 0x00 | 107 |

ARM ISA

# Example Code Sequence

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]
```

register file

| | |
|---|---|
| X3 | 0x00000000000000FF |
| X4 | 0xE5FF06C205FF0302 |

little endian

| | |
|---|---|
| 0xFF | 100 |
| 0x00 | 101 |
| 0x02 | 102 |
| 0x00 | 103 |
| 0x00 | 104 |
| 0x00 | 105 |
| 0x00 | 106 |
| 0x00 | 107 |

Memory
(each location is 1 byte)

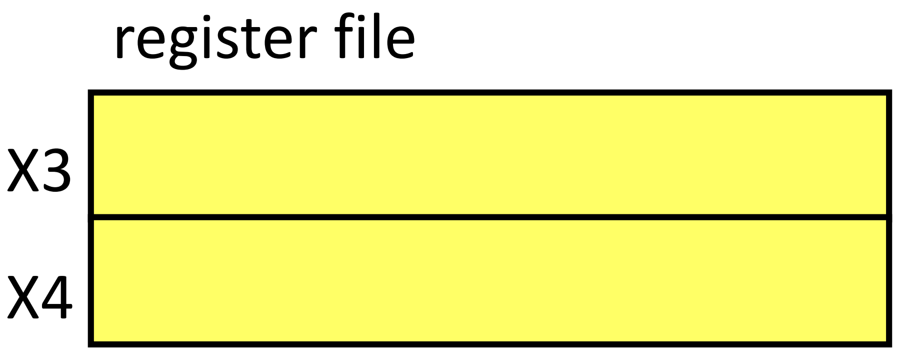
ARM ISA

# Example Code Sequence (2)

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]
```

*We shown the registers as blank. What do they actually contain before we run the snippet of code?*



Memory
(each location is 1 byte)

little endian

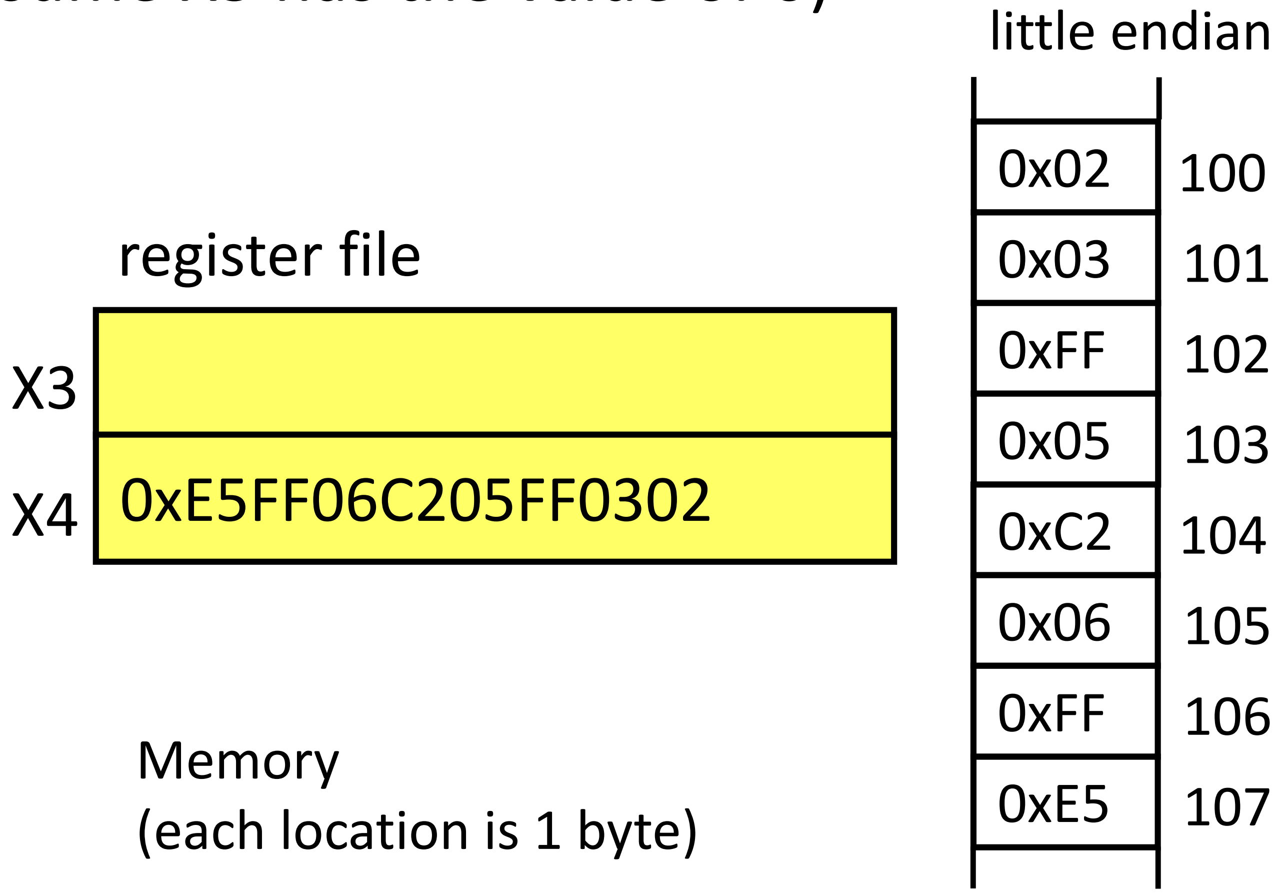
| | |
|---|---|
| 0x02 | 100 |
| 0x03 | 101 |
| 0xFF | 102 |
| 0x05 | 103 |
| 0xC2 | 104 |
| 0x06 | 105 |
| 0xFF | 106 |
| 0xE5 | 107 |

ARM ISA

## Example Code Sequence (2)

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]
```



| | |
|---|---|
| 0x02 | 100 |
| 0x03 | 101 |
| 0xFF | 102 |
| 0x05 | 103 |
| 0xC2 | 104 |
| 0x06 | 105 |
| 0xFF | 106 |
| 0xE5 | 107 |

ARM ISA

# Example Code Sequence (2)

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]
```

register file

| | |
|---|---|
| X3 | 0x00000000000000FF |
| X4 | 0xE5FF06C205FF0302 |

Memory
(each location is 1 byte)

little endian

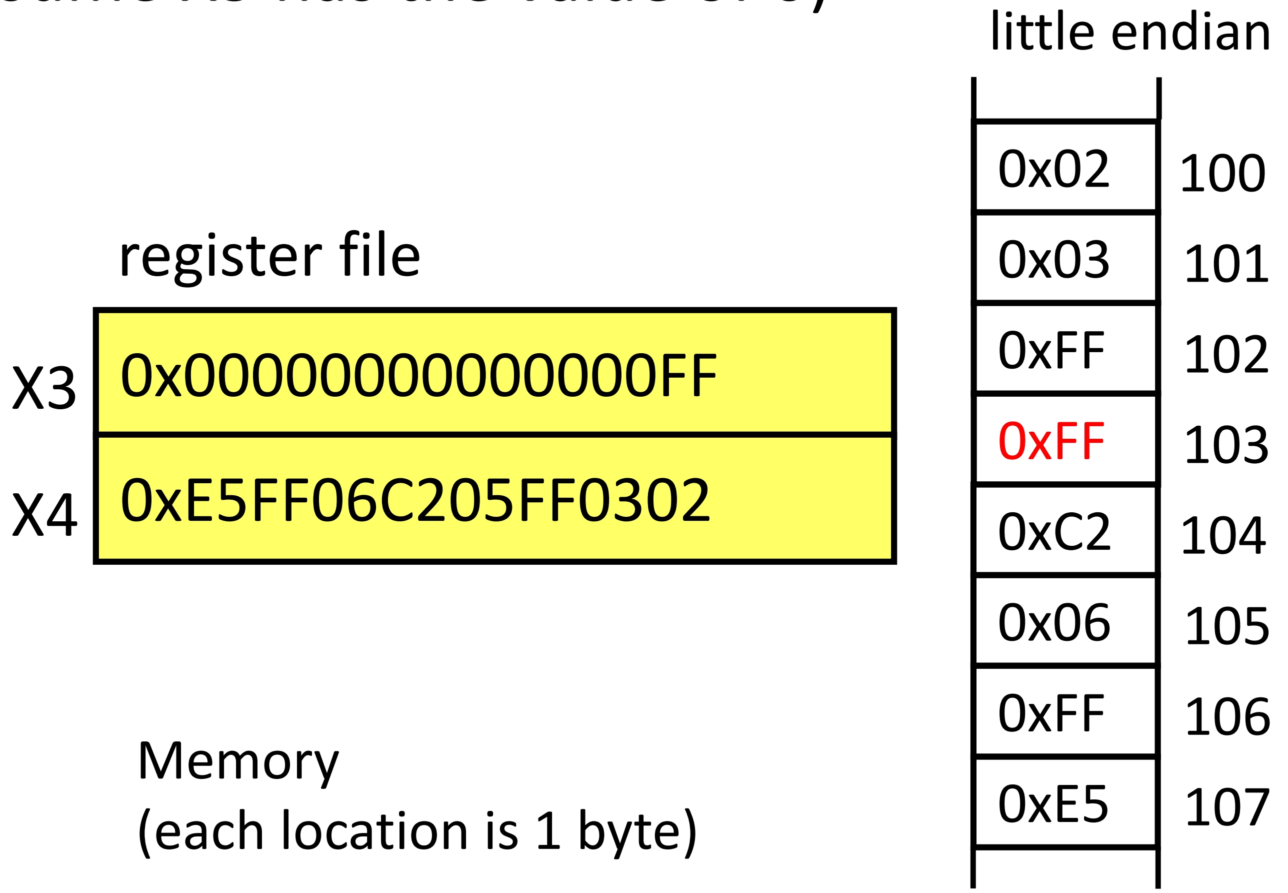
| | |
|---|---|
| 0x02 | 100 |
| 0x03 | 101 |
| 0xFF | 102 |
| 0x05 | 103 |
| 0xC2 | 104 |
| 0x06 | 105 |
| 0xFF | 106 |
| 0xE5 | 107 |

ARM ISA

# Example Code Sequence (2)

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]
```



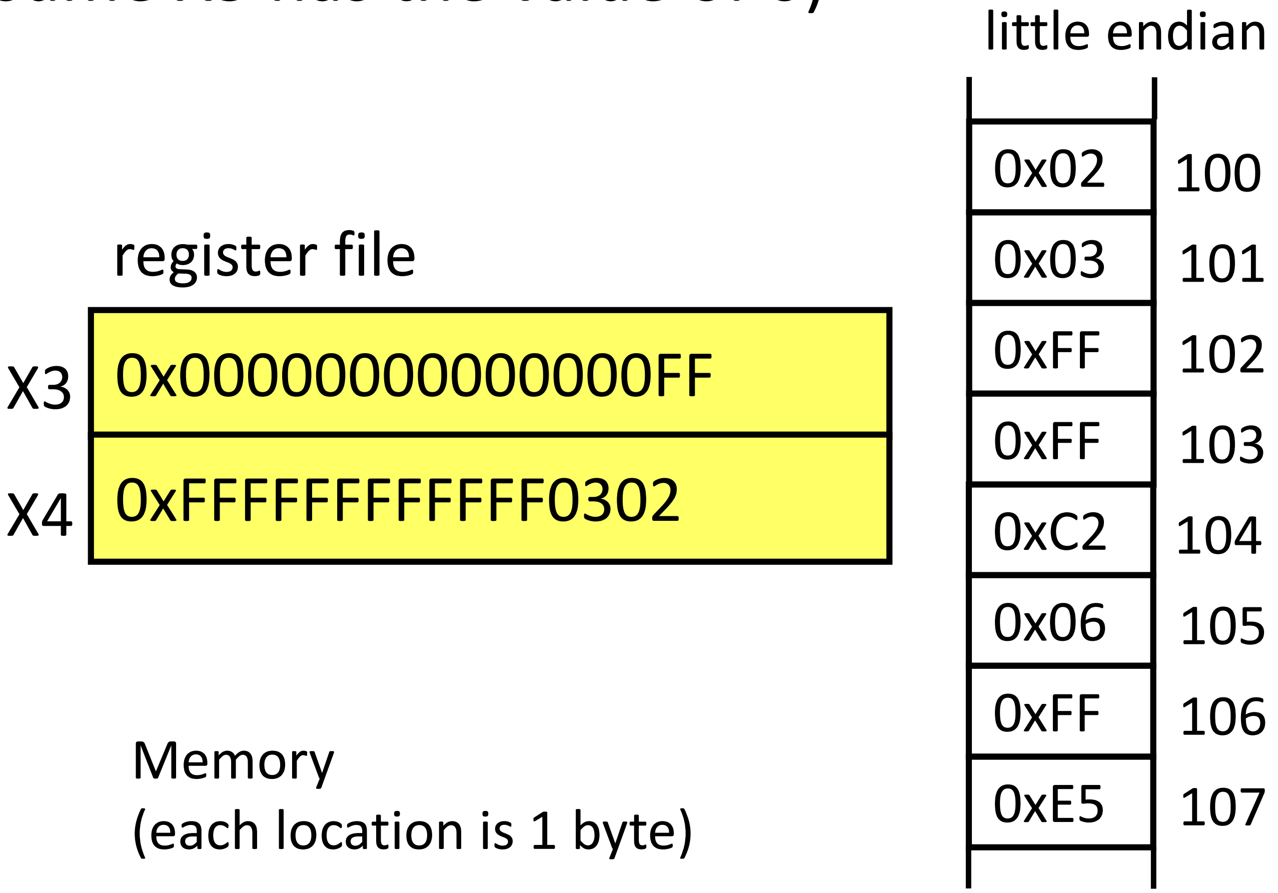
Memory
(each location is 1 byte)

# 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]
```



Memory
(each location is 1 byte)

## 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

- **Memory alignment**
    - Aligning Structs
- Control flow instructions
    - C-code examples
- Extra Problems

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

# Calculating Load/Store Addresses for Variables

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

- *Problem*: Assume data memory starts at address 100 and no reordering, calculate the total amount of memory needed

  a = 2 bytes * 100 = 200

  b = 1 byte

  c = 4 bytes

  d = 8 bytes

  e = 2 bytes

  i = 1 + 4 + 1 = 6 bytes

  total = 221, right or wrong?

## Memory layout of variables

- Compilers don’t like to place variables in memory arbitrarily
- As we'll see later in the course, memory is divided into fixed-sized **chunks**
  - When we load from a particular chunk, we really read the whole chunk
  - Usually an integer number of words (32 bits)
- If we read a single char (1 byte), it doesn't matter where it's placed

| 0x1000 | 0x1001 | 0x1002 | 0x1003 |
|---|---|---|---|
| 'a' | 'b' | 'c' | 'd' |

```
ldurb [x0, 0x1002]
```

- Reads [0x1000-0x1003], then throws away all but 0x1002, fine

# Memory layout of variables

- BUT, if we read a 32-bit integer word, and that word starts at 0x1002:

| 0x1000 | 0x1001 | 0x1002 | 0x1003 |
|---|---|---|---|
| 0xFF | 0xFF | 0x12 | 0x34 |

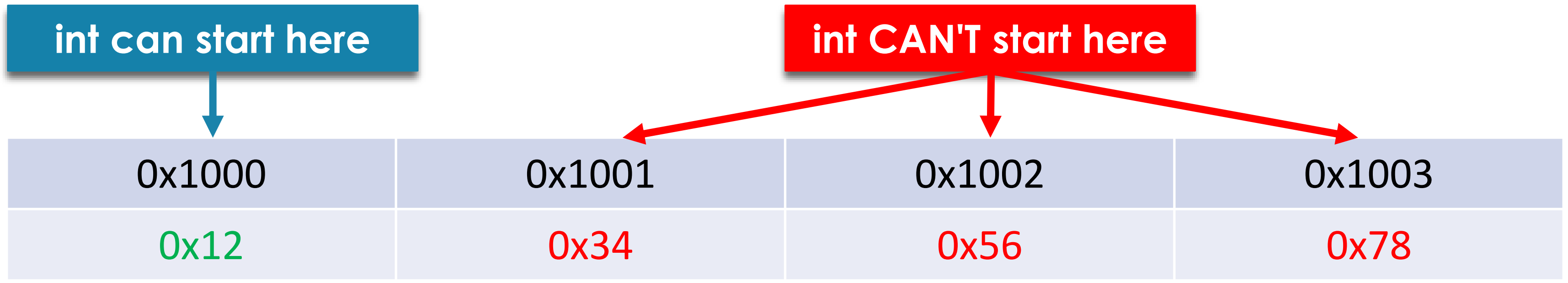
| 0x1004 | 0x1005 | 0x1006 | 0x1007 |
|---|---|---|---|
| 0x56 | 0x78 | 0xFF | 0xFF |

- First we need to read [0x1000-0x1003], throw away 0x1000 and 0x1001, **then** read [0x1004-0x1007]
- Need to read from memory twice! Slow! Complicated! **Bad!**

# Solution: Memory Alignment

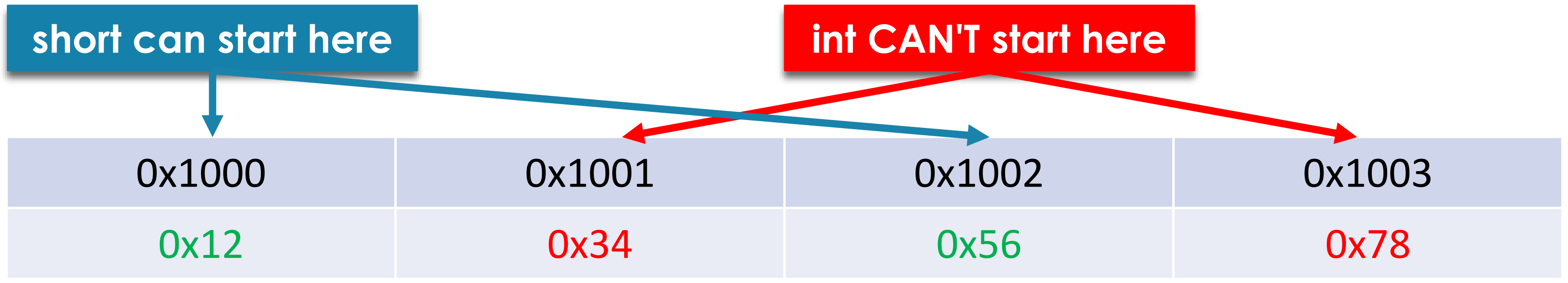
**Where can chars start?**

- Most modern ISAs require that data be aligned
  - An N-byte variable must start at an address A, such that (A%N) == 0
- For example, starting address of a 32-bit **int** must be divisible by 4



| 0x1000 | 0x1001 | 0x1002 | 0x1003 |
|---|---|---|---|
| 0x12 | 0x34 | 0x56 | 0x78 |

- Starting address of a 16-bit **short** must be divisible by 2



| 0x1000 | 0x1001 | 0x1002 | 0x1003 |
|---|---|---|---|
| 0x12 | 0x34 | 0x56 | 0x78 |

# Golden Rule of Alignment

```
char  c;
short s;
int   i;
```

- Every (primitive) object starts at an address divisible by its size
- "Padding" is placed in between objects if needed

| 0x1000 | 0x1001 | 0x1002 | 0x1003 | 0x1004 | 0x1005 | 0x1006 | 0x1007 |
|---|---|---|---|---|---|---|---|
| [c] | [padding] | [s] | | [i] | | | |

- But what about non-primitive data types?
  - Arrays? Treat as independent objects
  - Structs? Trickier…

# Agenda

- Memory alignment
  - **Aligning Structs**
- Control flow instructions
  - C-code examples
- Extra Problems

# Problem with Structs

```
char c;

struct {
  char c;
  int i;
} s[2];
```

- If we align each element of a struct according to the Golden Rule, we can still run into issues
  - E.g.: An array of structs

Amount of padding is different across different instances

| 1000 | 1001 | 1002 | 1003 | 1004 | 1005 | 1006 | 1007 | 1008 | 1009 | 100A | 100B | 100C | 100D | 100E | 100F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| c | s[0].c | [pad] | [pad] | s[0].i | | | | s[1].c | [pad] | [pad] | [pad] | s[1].i | | | |

- Why is this bad?
- It makes "for" loops very difficult to write!
  - Offsets need to be different on each iteration

# Structure Alignment

- Solution: in addition to laying out each field according to Golden Rule…
  - Identify largest (primitive) field
    - Starting address of overall struct is aligned based on the largest field
    - Padded in the back so total size is a multiple of the largest primitive

```
char c;

struct {
  char c;
  int i;
} s[2];
```

| 1000 | 1001 | 1002 | 1003 | 1004 | 1005 | 1006 | 1007 | 1008 | 1009 | 100A | 100B | 100C | 100D | 100E | 100F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| c | [pad] | [pad] | [pad] | s[0].c | [pad] | [pad] | [pad] | s[0].i | | | | s[1].c | [pad] | [pad] | [pad] |

Guaranteed to lay out each instance identically

# Structure Example

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

**What boundary should this struct be aligned to?**

a) 1 byte
b) 4 bytes
c) 12 bytes
d) 2 bytes
e) 19 bytes

- Assume struct starts at location 1000,
  - char w → 1000
  - x[0] → 1004-1007, x[1] → 1008 – 1011, x[2] → 1012 – 1015
  - char y → 1016
  - short z → 1018 – 1019

Total size = 20 bytes!

# Calculating Load/Store Addresses for Variables

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

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

- *Problem*: Assume data memory starts at address 100 and no reordering, calculate the total amount of memory needed

  a = 200 bytes (100-299)
  b = 1 byte (300-300)
  c = 4 bytes (304-307)
  d = 8 bytes (312-319)
  e = 2 bytes (320-321)
  struct: largest field is 4 bytes, start at 324
  f = 1 byte (324-324)
  g = 4 bytes (328-331)
  h = 1 byte (332-332)
  i = 12 bytes (324-335)
  236 bytes total!! (compared to 221, originally)

## Data Layout – Why?

- Does gcc (or another compiler) reorder variables in memory to avoid padding?
- Only outside structs
  - C99 forbids reordering elements inside a struct!
- The programmer (i.e., you) are expected to manage data layout of variables for your program and structs.
- Two optimal strategies:
  - Order fields in struct by datatype size, smallest first
  - Or by largest first

## Agenda

- Memory alignment
  - Aligning Structs
- **Control flow instructions**
  - C-code examples
- Extra Problems

# ARM/LEGv8 Sequencing Instructions

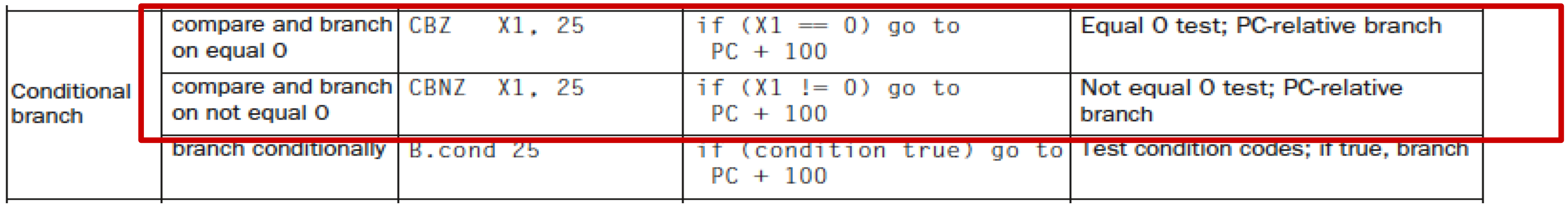
- Sequencing instructions change the flow of instructions that are executed
  - This is achieved by modifying the program counter (PC)
- Unconditional branches are the most straightforward
  they ALWAYS change the PC and thus "jump" to another instruction out of the usual sequence
- Conditional branches

If (condition_test) goto target_address

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

# LEGv8 Conditional Instructions

- 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.

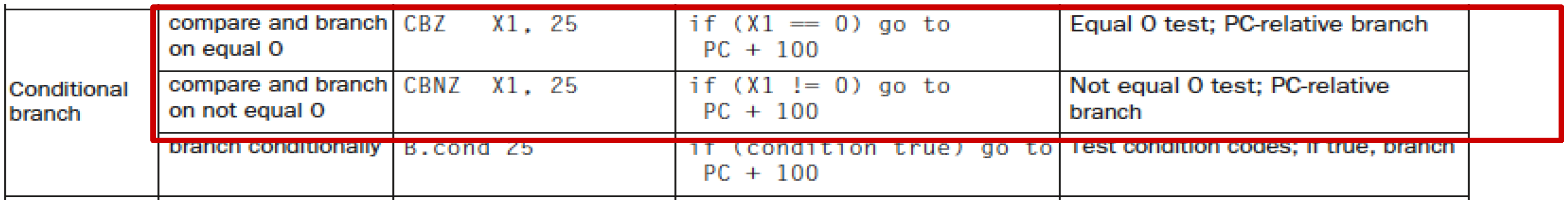


| | | | | |
|---|---|---|---|---|
| Conditional branch | compare and branch on equal 0 | `CBZ X1, 25` | `if (X1 == 0) go to PC + 100` | Equal 0 test; PC-relative branch |
| | compare and branch on not equal 0 | `CBNZ X1, 25` | `if (X1 != 0) go to PC + 100` | Not equal 0 test; PC-relative branch |
| |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

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

# LEGv8 Conditional Instructions

- 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 |

- Example: CBNZ X3, Again
  - If X3 doesn't equal 0, then branch to label "Again"
  - "Again" is an offset from the PC of the current instruction (CBNZ)
  - Why does "25" in the above table result in PC + 100?

# LEGv8 Conditional Instructions

- Example: What would the offset or displacement be if there were two instructions between ADDI and CBNZ?

```
Again:      ADDI    X3, X3, #-1
            --------------
            --------------
            CBNZ    X3, Again
```

**What is the offset?**

a) -16
b) -12
c) -4
d) -3
e) 0

# LEGv8 Conditional Instructions

- Example: What would the offset or displacement be if there were two instructions between ADDI and CBNZ?

```
Again:    ADDI   X3, X3, #-1
          --------------
          --------------
          CBNZ   X3, Again
```

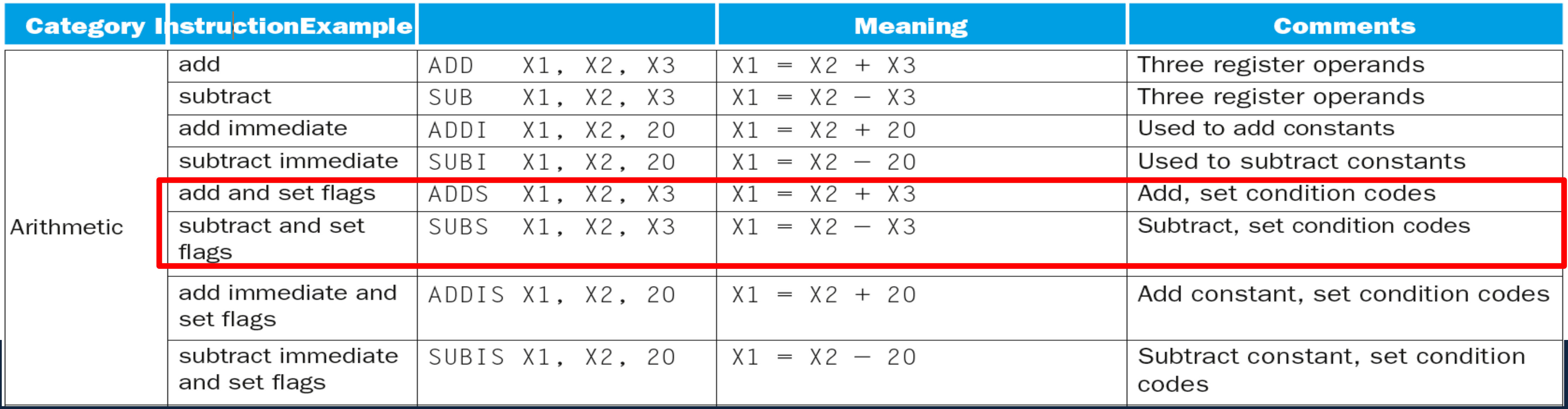
- Answer = -3
  - The offset field is 19 bits signed so the bit pattern would be 111 1111 1111 1111 1101
  - Two 00's are appended to the above 19 bits and then the result would be sign-extended (with one's) to 64 bits and added to the value of PC at CBNZ
  - Why the two 00's?

## LEGv8 Conditional Instructions

- Motivation:
  - Some types of branches makes sense to check if a certain value is zero or not
    - while(a)
  - But not all:
    - if(a > b)
    - if(a == b)
  - Using an extra **program status register** to check for various conditions allows for a greater breadth of branching behavior

# LEGv8 Conditional Instructions Using FLAGS

- FLAGS: NZVC record the results of (arithmetic) operations Negative, Zero, oVerflow, Carry—not present in LC2K
- We explicitly set them using the “set” modification to ADD/SUB etc.
- Example: ADDS causes the 4 flag bits to be set according as the outcome is negative, zero, overflows, or generates a carry

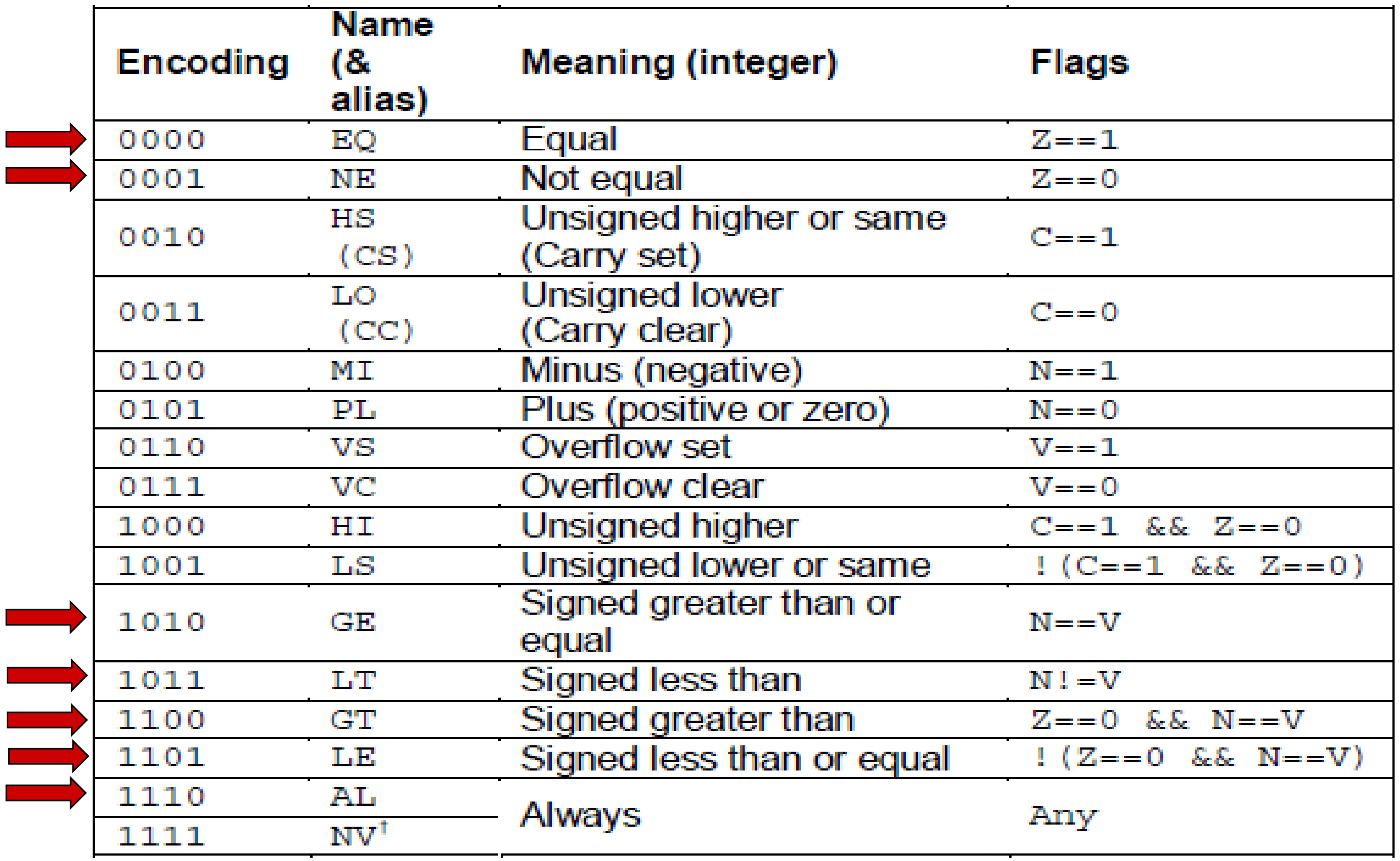


| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add | ADD X1, X2, X3 | X1 = X2 + X3 | Three register operands |
| | subtract | SUB X1, X2, X3 | X1 = X2 – X3 | Three register operands |
| | add immediate | ADDI X1, X2, 20 | X1 = X2 + 20 | Used to add constants |
| | 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 |

## ARM Condition Codes Determine Direction of Branch

- 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)~~
- Don’t worry about the C and V for this class

# ARM Condition Codes Determine Direction of Branch--continued



|  | 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† |  |  |

Need to know the 7 with the red arrows

```
CMP   X1, X2
B.LE  Label1
```

For this example, we branch if X1 is <= to X2

## Conditional Branches: How to use

- CMP instruction lets you compare two registers.
  - Could also use SUBS etc.
    - That could save you an instruction.
- B.cond lets you branch based on that comparison.

- Example:

```
CMP   X1, X2
B.GT  Label1
```

  - Branches to Label1 if X1 is greater than X2.

## Agenda

- Memory alignment
  - Aligning Structs
- Control flow instructions
  - **C-code examples**
- Extra Problems

## Branch—Example

- Convert the following C code into LEGv8 assembly (assume x is in X1, y in X2):

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

# Branch—Example

- Convert the following C code into LEGv8 assembly (assume x is in X1, y in X2):

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

Using Labels

```
    CMP  X1, X2
    B.NE L1
    ADD  X1, X1, #1
    B    L2
L1: ADD  X2, X2, #1
L2: …
```

Note that conditions in assembly are often the inverse of the "if" condition. Why?

Without Labels

```
CMP    X1, X2
B.NE   3
ADD    X1, X1, #1
B      2
ADD    X2, X2, #1
```

Assemblers must deal with labels and assign displacements

## Loop—Example

// assume all variables are long long integers (64 bits or 8 bytes)
// i is in X1, start of a is at address 100, sum is in X2

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

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

# of branch instructions
= 3*10 + 1= 31

a.k.a. while-do template

## Agenda

- Memory alignment
  - Aligning Structs
- Control flow instructions
  - C-code examples
- **Extra Problems**

# Extra Example: Do-while Loop

// assume all variables are long long integers (64 bits or 8 bytes)
// i is in X1, start of a is at address 100, sum is in X2

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

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

# of branch instructions
= 2*10 = 20

a.k.a. do-while template

# Extra Example: Do-while Loop

// assume all variables are long long integers (64 bits or 8 bytes)
// i is in X1, start of a is at address 100, sum is in X2

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

# of branch instructions
= 2*10 = 20

a.k.a. do-while template

## Extra Problem – For Your Reference

- Write the ARM assembly code to implement the following C code:

```
// assume ptr is in X1
// struct {int val; struct node *next;} node;
// struct node *ptr;

if ((ptr != NULL) && (ptr->val > 0))
  ptr->val++;
```

## Extra Problem

- Write the ARM assembly code to implement the following C code:

```
// assume ptr is in X1
// struct {int val; struct node *next;} node;
// struct node *ptr;

if ((ptr != NULL) && (ptr->val > 0))
  ptr->val++;
```

```
        cmp r1, #0
        beq Endif
        ldursw  r2, [r1, #0]
        cmp r2, #0
        b.le Endif
        add r2, r2, #1
        str r2, [r1, #0]
Endif : ….
```

## Extra Class Problem

- How much memory is required for the following data, assuming that the data starts at address 200 and is a 32 bit address space?

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

**How much memory?**

a) x < 40 bytes
b) 40 < x < 50 bytes
c) 50 < x < 60 bytes
d) 60 < x bytes

## Next Time

- More C-to-Assembly
    - Function calls