4. Instruction Set Architecture – from C to assembly – Basic blocks

EECS 370 – Introduction to Computer Organization – Winter 2015

Robert Dick, Andrew Lukefahr, and Satish Narayanasamy

EECS Department
University of Michigan in Ann Arbor, USA

Announcements

- HW-1 due on 1/22

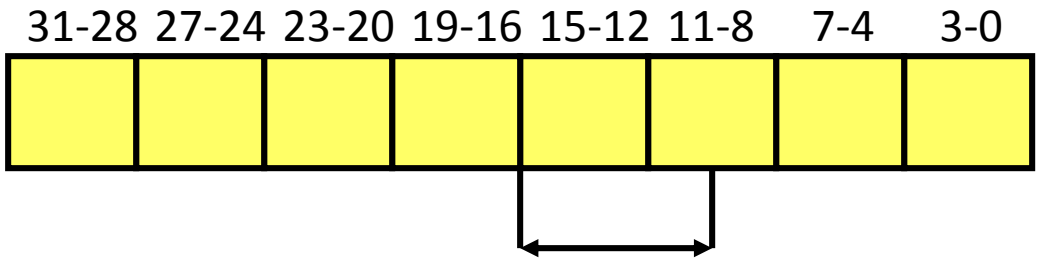
Recap

- LC-2K
  - Instructions in LC2K
  - Assembling
  - Addressing: only base+displacement
- ARM
  - Arithmetic, logic instructions
  - Many loads and stores in ARM
  - Addressing: base+displacement and base+register

- Slides from the previous lecture

Class Problem 2

- Show the C and ARM assembly for extracting the value in bits 15:10 from a 32-bit integer variable



Want these bits

Remember each hex digit is 4 bits

ARM Memory Instructions

ARM ISA

- Supports base + displacement and base + register modes only
  - Base is a register
  - Offset is a 12-bit immediate, either positive or negative

- Format: 2 registers (dest, base) and 12-bit immediate (offset)

  - Example:
    ldr r3, [r4, #1000] // load word
    Retrieves 32-bit value from memory location (r4 + 1000) and puts the result into r3

Load Instruction Sizes

ARM ISA

How much data is retrieved from memory at the given address?

- ldr r3, [r4, #1000]
  - retrieve a word (32 bits) from address (r4+1000)
- ldrh r3, [r4, #1000]
  - retrieve a halfword (16 bits) from address (r4+1000)
- ldrb r3, [r4, #1000]
  - retrieve a byte (8 bits) from address (r4+1000)

Sign/Zero Extension

ARM ISA

- Registers in ARM are 32 bits!
- So what happens when you load 8 or 16 bits?

  - Sign extend if the load is signed

  - Zero extend if the load is unsigned

ARM Memory Instructions - Summary

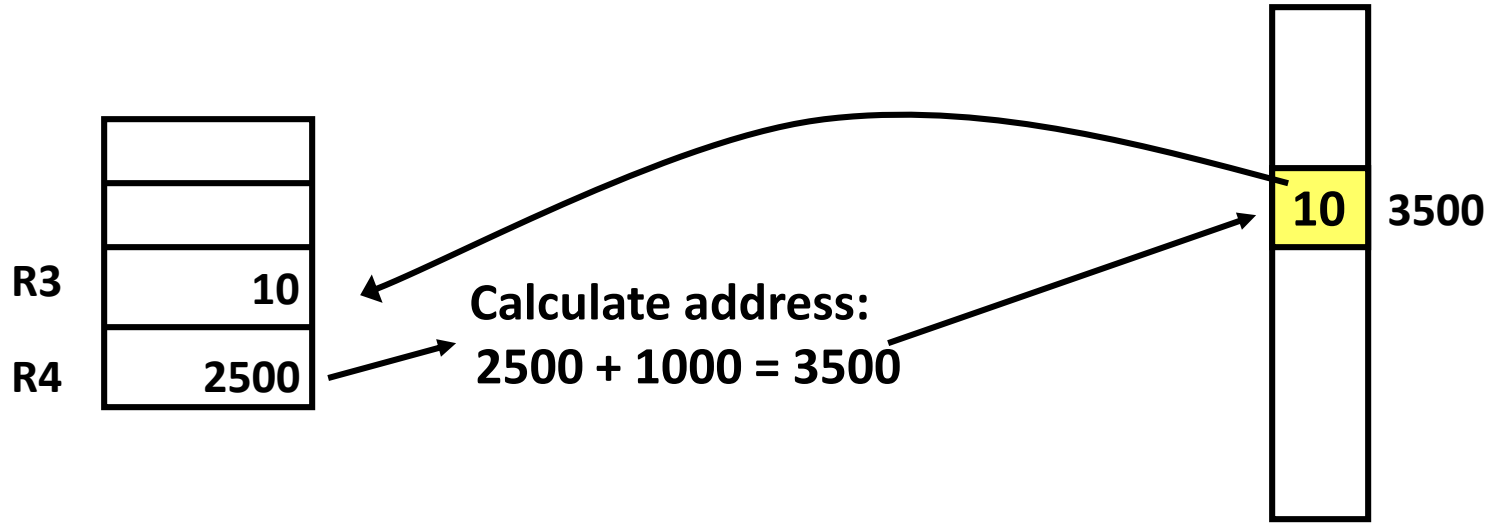
ARM ISA

- Load instructions
  - ldrsb \\ load byte signed (load 8 bits, sign extend)
  - ldrb \\ load byte unsigned (load 8 bits, zero extend)
  - ldrsh \\ load halfword (load 16 bits, sign extend)
  - ldrh \\ load halfword unsigned (load 16 bits, zero extend)
  - ldr \\ load word (load 32 bits, no extension)

- Store instructions (No sign/zero extension for stores)
  - strb r3, [r4, #1000] \\ store 8 LSBs of r3 to M[r4+1000]
  - strh r3, [r4, #1000] \\ store 16 LSBs of r3 to M[r4+1000]
  - str r3, [r4, #1000] \\ store all 32 bits of r3 to M[r4+1000]

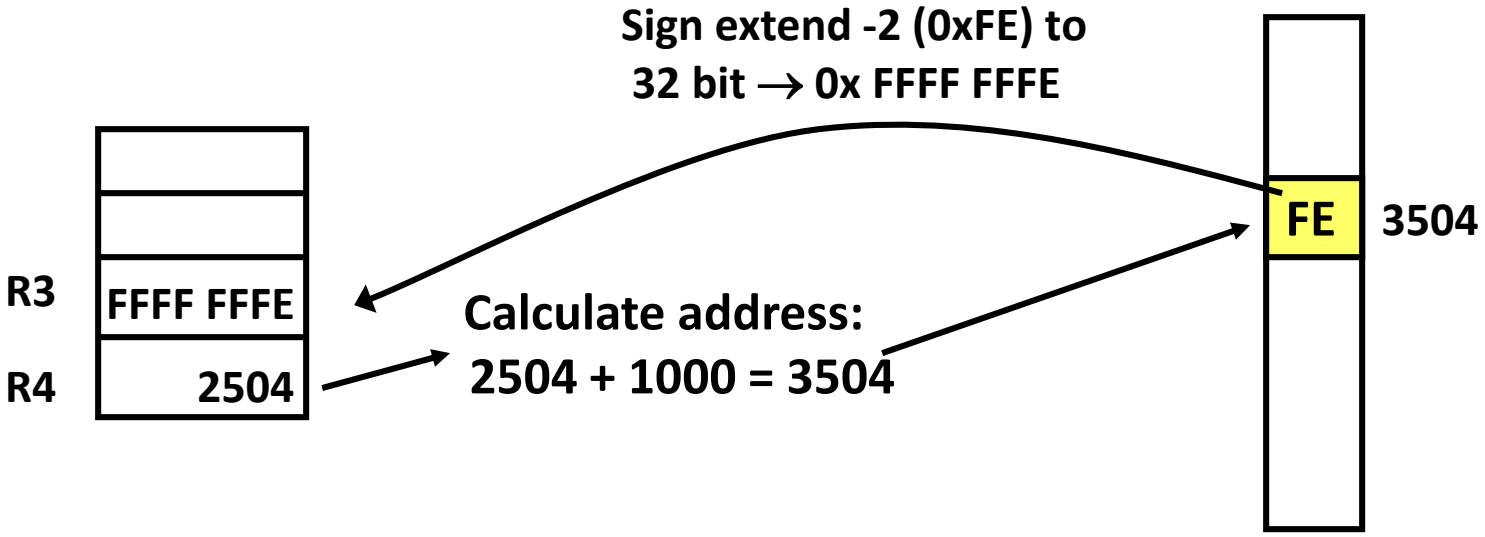
Load Instruction in Action

- ldrsb r3, [r4, #1000] // load signed byte
  Retrieves 8-bit value from memory location (r4+1000) and puts the result into r3 (sign extended)



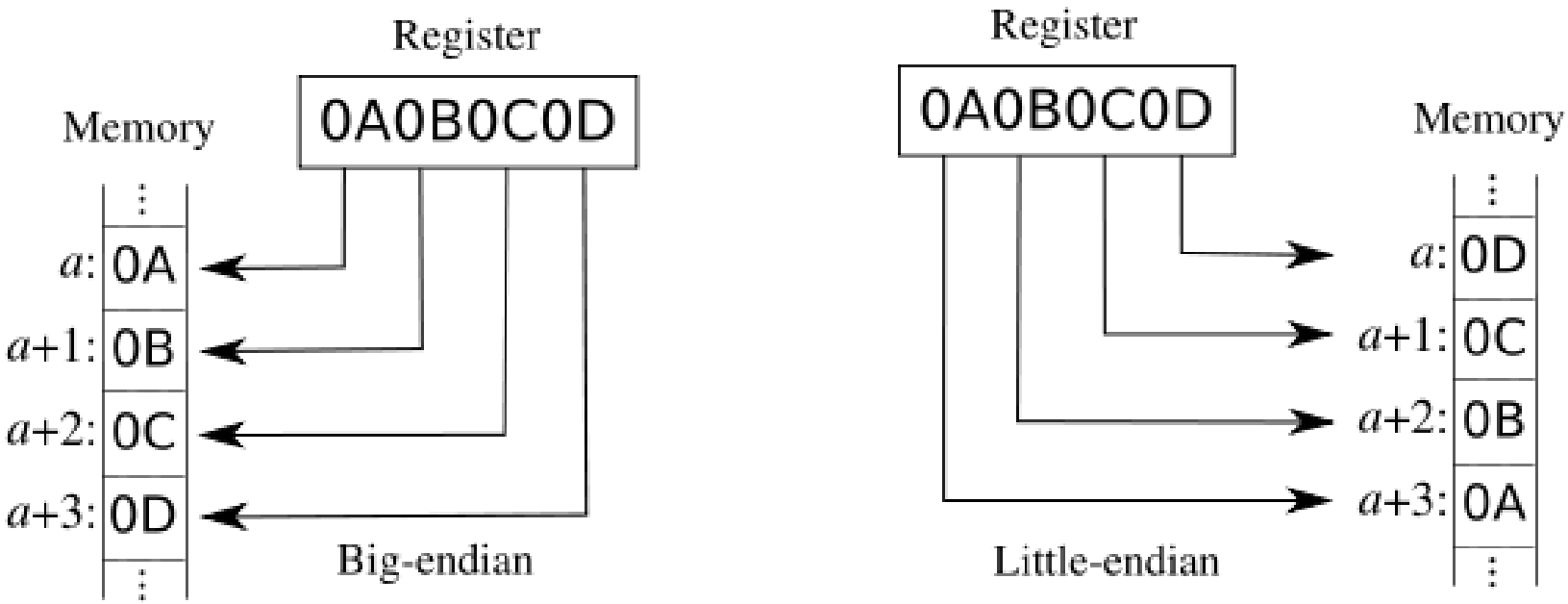
Load Instruction in Action – other example

- ldrsb r3, [r4, #1000] // load signed byte
  Retrieves 8-bit value from memory location (r4+1000) and puts the result into r3 (sign extended)



Big Endian vs. Little Endian

- Endian-ness: ordering of bytes within a word
  - Little - increasing numeric significance with increasing memory addresses
  - Big – The opposite, most significant byte first
  - The Internet is big endian, x86 and ARM are little endian

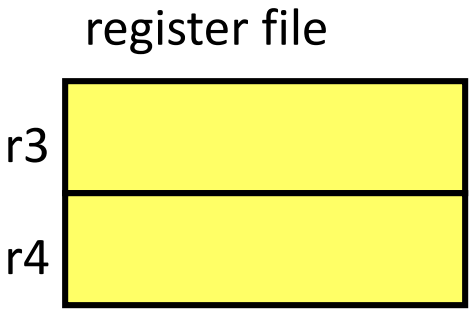


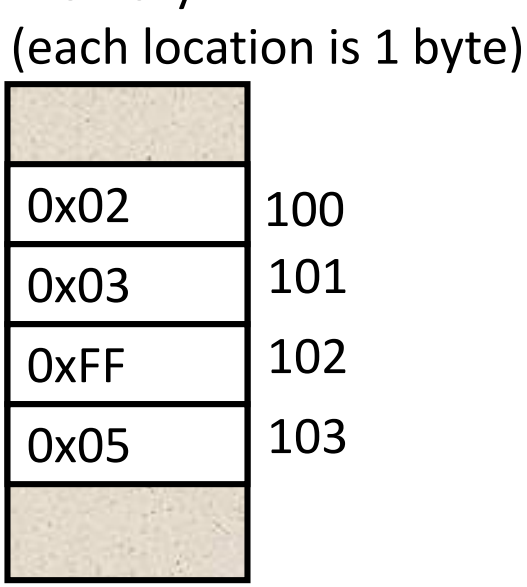
Example Code Sequence

ARM ISA

What is the final state of memory once you execute the following instruction sequence? (assume r0 has the value of 0)

```
ldr     r4, [r0, #100]
ldrsb   r3, [r0, #102]
str     r3, [r0, #100]
strb    r4, [r0, #102]
```



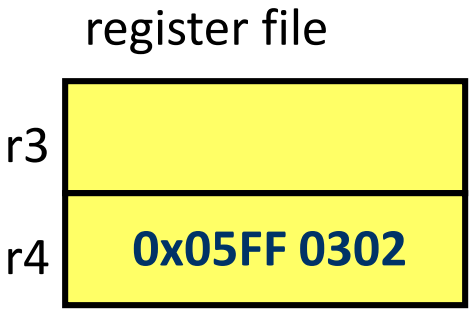


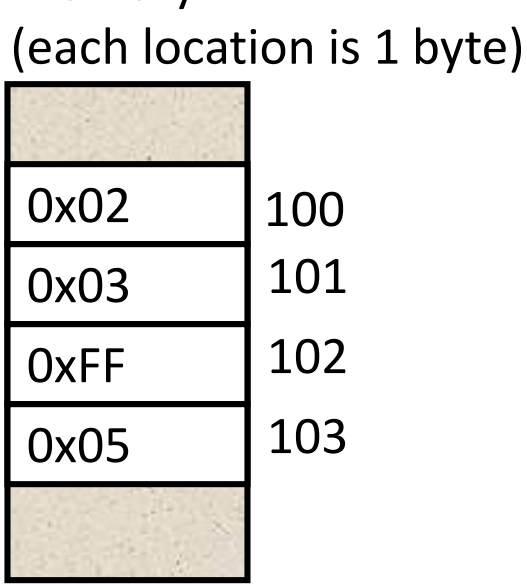
Example Code Sequence – insn 1

ARM ISA

What is the final state of memory once you execute the following instruction sequence?

```
ldr     r4, [r0, #100]
ldrsb   r3, [r0, #102]
str     r3, [r0, #100]
strb    r4, [r0, #102]
```





Example Code Sequence – insn 2

ARM ISA

What is the final state of memory once you execute the following instruction sequence?

```
ldr     r4, [r0, #100]
ldrsb   r3, [r0, #102]
str     r3, [r0, #100]
strb    r4, [r0, #102]
```

register file

r3	0xFFFF FFFF
r4	0x05FF 0302

Memory
(each location is 1 byte)

0x02	100
0x03	101
0xFF	102
0x05	103

Example Code Sequence – insn 3

ARM ISA

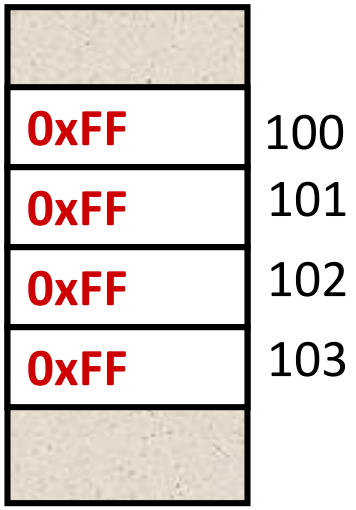
What is the final state of memory once you execute the following instruction sequence?

```
ldr     r4, [r0, #100]
ldrsb   r3, [r0, #102]
str     r3, [r0, #100]
strb    r4, [r0, #102]
```

register file

r3	0xFFFF FFFF
r4	0x05FF 0302

Memory
(each location is 1 byte)



Example Code Sequence – insn 4

ARM ISA

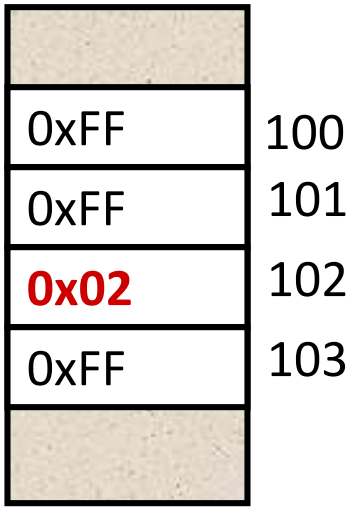
What is the final state of memory once you execute the following instruction sequence?

```
ldr     r4, [r0, #100]
ldrsb   r3, [r0, #102]
str     r3, [r0, #100]
strb    r4, [r0, #102]
```

register file

r3	0xFFFF FFFF
r4	**0x05FF 0302**

Memory
(each location is 1 byte)

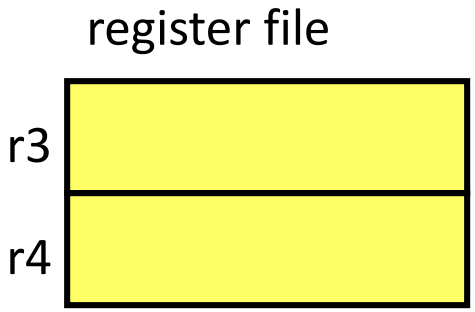


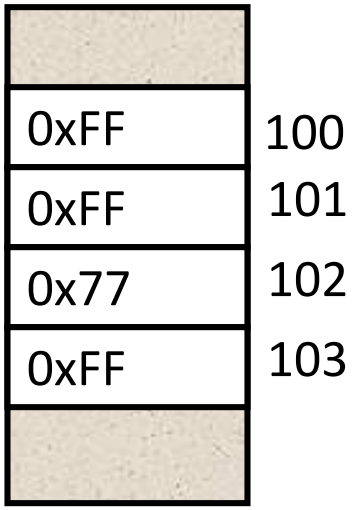
Class Problem 3

ARM ISA

What is the final state of memory once you execute the following instruction sequence? (assume r0 has the value of 0)

```
ldrh    r3, [r0, #100]
ldrb    r4, [r0, #102]
str     r3, [r0, #100]
strh    r4, [r0, #102]
```





Class Problem 3

What is the final state of memory once you execute the following instruction sequence? (assume r0 has the value of 0)

```
ldrh    r3, [r0, #100]
ldrb    r4, [r0, #102]
str     r3, [r0, #100]
strh    r4, [r0, #102]
```

register file

r3	0x0000 FFFF
r4	0x0000 0077

Memory
(each location is 1 byte)

0xFF	100
0xFF	101
0x77	102
0x00	103

- Slides from the previous lecture ends

Instruction Set Architecture (ISA) Design Lectures

- Lecture 2: Storage types and addressing modes
- Lecture 3 : LC-2K and ARM architecture
- **Lecture 4 : Converting C to assembly – basic blocks**
- Lecture 5 : Converting C to assembly – functions
- Lecture 6 : Translation software; libraries, memory layout

Converting C to Assembly

- Memory layout → memory addresses
- Branches
- Procedure calls
- Expression trees
- Register allocation

Converting C to assembly – example 1

DATA LAYOUT

Write ARM assembly code for the following C expression:

C: a = b + names[i];

Assume that **a** is in r1, **b** is in r2, **i** is in r3, and the array **names** starts at address 1000 and holds 32-bit integers

```
mov     r6,   #4    // set up integer size
mul     r5, r3, r6// calculate array offset  i*4
ldr     r4,  [r5, #1000]     // load names[i]
add     r1,  r2,  r4         // calculate b + names[i]
```

Converting C to assembly – example 2

DATA LAYOUT

Write ARM assembly code for the following C expression:

class { int a; unsigned char b, c; } y;

y.a = y.b + y.c;

Assume that a pointer to y is in r1.

```
ldrb    r2,  [r1, #4]   // load y.b
ldrb    r3,  [r1, #5]   // load y.c
add     r4,  r2,  r3    // calculate y.b+y.c
str     r4,  [r1, #0]   // store y.a
```

How do you determine the offsets for the class sub-fields?

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, 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

DATA LAYOUT

- For ARM, cannot always arbitrarily pack variables into memory
  - Need to worry about alignment
  - An N-byte variable must start at an address A, such that (A % N) == 0
  - Newer ARM processors will perform unaligned loads/stores, but they are VERY SLOW

- "Golden" rule – Address of a variable is aligned based on the size of the variable
  - **char** is byte aligned (any addr is fine)
  - **short** is half-word aligned (LSBit of addr must be 0)
  - **int** is word aligned (2 LSBit's of addr must be 0)

Structure/Class alignment

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

- Identify largest 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

Structure Example

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

The largest field is **int** (4 bytes), hence:

⇒ struct size is multiple of 4

⇒ struct's starting addr is word aligned

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!

Earlier Example – 2nd Try

Assume data memory starts at address 100

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

→ 200 bytes → 100-299
→ 1 byte → 300-300
→ 4 bytes → 304-307
→ 8 bytes → 312-319
→ 2 bytes → 320-321
→ largest field is 4 bytes → start at 324
→ 1 byte → 324-324
→ 4 bytes → 328 - 331
→ 1 byte → 332-332
→ struct size is 12 bytes, spanning 324 – 335

236 bytes total!!

Class Problem 1

- How much memory is required for the following data, assuming that the data starts at address 200?

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

ARM Sequencing Instructions

BRANCH

- Sequencing instructions change the flow of instructions that are executed
  - This is achieved by modifying the program counter (r15)
- Conditional branches
  - If (condition_test) goto target_address
    - *condition_test* examines flags from the processor status word (PSR)
    - *target_address* is a 24-bit word displacement on current PC+8

- **cmp r1, r2**
  beq label
- if (r1 == r2) then PC = label else PC = PC + 4

ARM Condition Codes Determine Direction of Branch

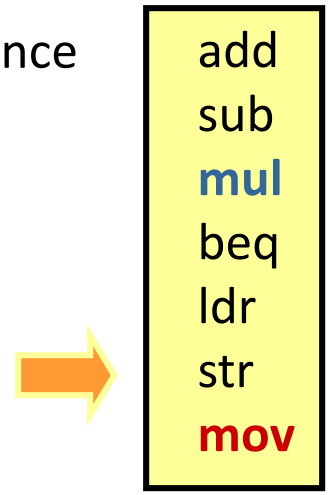
- Most arithmetic/logic instructions set condition codes in PSR
  - add, sub, cmp, and, eor, etc…
- Four primary condition codes evaluated:
  - N – set if the result is negative (i.e., bit 31 is non-zero)
  - Z – set if the result is zero (i.e., all 32 bits are zero)
  - C – set if last addition/subtraction had a carry/borrow out of bit 31
  - V – set if the last addition/subtraction produced an overflow (e.g., two negative numbers added together produce a positive result)
- Branch conditions:
  - eq – (Z == 1)
  - ne – (Z == 0)
  - ge – (N == V)
  - lt – (N != V)
  - gt – (Z == 0 || N == V)
  - le – (Z == 1 || N != V)
  - al – 1 (can use shorthand “b label”)

Setting the Branch Displacement Field

BRANCH

- Target address is a 24-bit signed aligned displacement on current PC+8
  - Target = PC + 8 + 4*24_bit_signed_displacement
    - beq 1 // branch **3** instructions ahead if flag Z == 1
    - beq -3 // branch **1** instruction back if flag Z == 1
    - beq -2 // Infinite loop if flag Z == 1

Example code sequence



Other Branching Instructions

BRANCH

- cmp r2, r3 // branch to 4*offset+PC+8 if r2 ≠ r3
  bne offset
- cmp r2, r3 // branch to 4*offset+PC+8 if r2 < r3
  blt offset
- cmp r2, r3 // branch to 4*offset+PC+8 if r2 ≥ r3
  bge offset

- b offset // jump to 4*offset+PC+8
  // unconditional jump, is taken regardless of PSR flags

- mov r15, r3 // jump to address in r3 -- when is this useful?

- bl offset // put PC+4 into register r14 (LR) and jump to 4*offset+PC+8

Branch - Example

BRANCH

Convert the following C code into ARM assembly (assume x is in r1, y in r2):

```
    int x, y;
    if (x == y)
        x++;
(L1) else
        y++;
(L2) ...
```

Using Labels

```
        cmp  r1, r2
        bne  L1
        add r1, r1, #1
        b L2
L1:  add r2, r2, #1
L2:  ...
```

Without Labels

```
cmp r1, r2
bne 1
add r1, r1, #1
b 0
add r2, r2, #1
```

Assemblers must deal with labels and assign displacements

Loop – Example

BRANCH

```
// assume all variables are integers
// i is in r1, start of a is at address 500, sum is in r2
for (i=0 ; i < 100 ; i++) {
      if (a[i] > 0) {
            sum += a[i];
      }
}
```

```
            mov     r1, #0
            mov     r4, #400
Loop1:      cmp     r1, r4
            bge     endLoop
            ldr     r5, [r1, #500]
            cmp     r5, #0
            ble     endIf
            add     r2, r2, r5
endIf:      add     r1, r1, #4
            b       Loop1
endLoop:
```

# of branch instructions
= 3*100 = 300

a.k.a. while-do template

Same Loop, Different Assembly

BRANCH

```
// assume all variables are integers
// i is in r1, start of a is at address 500, sum is in r2
for (i=0 ; i < 100 ; i++) {
      if (a[i] > 0) {
            sum += a[i];
      }
}
```

```
            mov     r1, #0
            mov     r4, #400
            cmp     r1, r4
            bge     endLoop
Loop1:      ldr     r5, [r1, #500]
            cmp     r5, #0
            ble     endIf
            add     r2, r2, r5
endIf:      add     r1, r1, #4
            cmp     r1, r4
            blt     Loop1
endLoop:
```

# of branch instructions
= 2*100 + 1 = 201

a.k.a. do-while template

Class Problem 2

BRANCH

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

```
// assume ptr is in r1
// struct {int val; struct node *next;} node;
// struct node *ptr;

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