Flow Control With ARMv4 Branches

Instructor: Dr. Vinicius Prado da Fonseca (vpradodafons@online.mun.ca)

Flow Control

- A computer can performs different tasks depending conditions
- Some statements conditionally execute code depending on some test
  - if/else
  - switch/case
  - while and for loops
- One way to make decisions is to use conditional execution to ignore certain instructions

  ADDEQ R0, R0, R1

  - Works well for simple if statements, a small number of instructions are ignored
- Wasteful for if statements with many instructions in the body
- Insufficient to handle loops
- Flow control means change the PC (program counter)
- ARMv4:
  - +4 (next instruction)
  - +8 (false conditional execution)

Maximum (max.s)

- Find the maximum of r0, r1, r2, and r3
- Place the maximum in r0

```
mov    r0, #13  // a = 13
mov    r1, #3   // b = 3
mov    r2, #22  // c = 22
mov    r3, #10  // d = 10
cmp    r0, r1
movlt r0, r1         // if (r0 < r1) r0 = r1
cmp    r0, r2
movlt r0, r2         // if (r0 < r2) r0 = r2
cmp    r0, r3
movlt r0, r3         // if (r0 < r3) r0 = r3
mov   pc, lr         // return value in r0
```

1011	LT	Less Than	$N \oplus V$

Maximum (max.s)

- cmp r0, r1 sets the condition codes depending on the result of r0 - r1
- If r1 is greater than r0, the result will be less than zero (N=1)
- movlt r0, r1 only execute if the condition codes indicate that the result is less than 0 (N=0)

```
mov    r0, #13 // a = 13
mov    r1, #30 // b = 30
mov    r2, #22 // c = 22
mov    r3, #10 // d = 10
cmp    r0, r1
movlt r0, r1         // if (r0 < r1) r0 = r1
cmp    r0, r2
movlt r0, r2         // if (r0 < r2) r0 = r2
cmp    r0, r3
movlt r0, r3         // if (r0 < r3) r0 = r3
mov   pc, lr         // return value in r0
```

Count First Four Bits (count.s)

- tst r0, #1 sets the condition codes on the result of r0 & 1
  - If bit 0 is a 1, the result is not zero
  - if bit 0 is 0, the result is 0
- addne r0, r0, #1 execute if the condition codes indicate a nonzero result.

TST Rd, Rn, Rm	R[m] & R[n], set codes
TEQ Rd, Rn, Rm	R[m] ^ R[n], set codes
CMP Rd, Rn, Rm	R[m] - R[n], set codes
CMN Rd, Rn, Rm	R[n] + R[m], set codes

```
mov    r0, #6          // r0 = 6
mov    r1, r0
mov    r0, #0          // r0 = 0
tst    r1, #1          // check bit0
addne r0, r0, #1 // if ( r1 & 1 ) r0 += 1
tst    r1, #2 // check bit1
addne r0, r0, #1 // if ( r1 & 2 ) r0 += 1
tst    r1, #4          // check bit2
addne r0, r0, #1 // if ( r1 & 4 ) r0 += 1
tst    r1, #8 // check bit3
addne r0, r0, #1 // if ( r1 & 8 ) r0 += 1
mov  pc, lr            // return value in r0
```

0001	NE	Not Equal	$\overline{Z}$

Count First Four Bits (count.s)

- Count the number of ones in the first four bits of r0
- 0110 & 0001
  r0 = 0
- 0110 & 0010
  r0 = 1
- 0110 & 0100
  r0 = 2
- 0110 & 1000
  r0 = 2

```
mov    r0, #6          // r0 = 6
mov    r1, r0
mov    r0, #0          // r0 = 0
tst    r1, #1          // check bit0
addne r0, r0, #1 // if ( r1 & 1 ) r0 += 1
tst    r1, #2 // check bit1
addne r0, r0, #1 // if ( r1 & 2 ) r0 += 1
tst    r1, #4          // check bit2
addne r0, r0, #1 // if ( r1 & 4 ) r0 += 1
tst    r1, #8 // check bit3
addne r0, r0, #1 // if ( r1 & 8 ) r0 += 1
mov  pc, lr            // return value in r0
```

Branching

- ARM and most other architectures use branch instructions to skip over sections of code or repeat code
- A program usually executes in sequence
- The program counter (PC) incrementing by 4 after each instruction to point to the next instruction
  - Instructions are 4 bytes long
  - ARM is a byte-addressed architecture
- Branch instructions change the program counter

Branching

- The B (branch) and BL (branch and link) instructions change the PC to the target location.
  - B target // PC = target
  - BL target // LR = PC+4; PC = target
- The B and BL can also be controlled by the condition codes (e.g. lt, eq)
- Branches are used to implement flow control in a program
- The typical flow control statements are if, if-else, while, and do-while
- The BL is used to implement a subroutine call
- The lr (link register) saves the instruction after the branch and link to enable the subroutine to return

If Statement (mk_even.s)

- The condition of the if statement determines if the body of the if statement is executed
- A branch is used to skip the body of the if statement
- The branch is executed if the if statement condition is false.

```
if (r0 & 1) :
        r0 = r0 + 1
```

```
main:

    // make even if odd

    mov    r0, #13        // a = 13

    tst    r0, #1     // check bit 0

    beq    skip_if // skip if result is 0

    add    r0, r0, #1

skip_if:

    mov  pc, lr // return value in r0
```

0000	EQ	Equal	Z

If Statement (mk_even.s)

- The skip_if label is placed after the body of the if statement
- If beq is true, then the number is already even, and the branch will skip the add r0, r0, #1 instruction
- Can this program be rewritten without a branch instruction?

```
main:
    // make even if odd
    mov   r0, #13        // a = 13
    tst   r0, #1    // check bit 0
    beq   skip_if // skip if result is 0
    add   r0, r0, #1
skip_if:
    mov  pc, lr // return value in r0
```

If Statement

- Longer if block
- Consider the following python program that has more instructions in the body of the if statement.

```
if (r0 & 1):
    r0 = r0 + 1
    r1 = 8
    r2 = 400
```

```
main:
    // make even if odd
    mov   r0, #13 // a = 13
    tst   r0, #1  // check bit 0
    beq   skip_if // skip if result is 0
    add   r0, r0, #1      // make even
    mov   r1, #8          // r1 = 8 other stuff
    mov   r2, #400        // r2 = 400
skip_if:
    mov  pc, lr // return value in r0
```

If-else Statement (ifelse.s)

- The if-else statement provides two blocks of code

```
r0 = 6
r1 = 10
if r0 > r1:
    r1 = 5
    r2 = 7
else:
    r1 = 10 // double values
    r2 = 14
r0 = r1 + r2
```

```
main:
        mov   r0, #6
        mov   r1, #10
        cmp   r0, r1     // if test
        ble   skip_if
        mov   r1, #5     // if body
        mov   r2, #7
        b     skip_else
skip_if:
        mov   r1, #10    // else body
        mov   r2, #14
skip_else:
        add r0, r1, r2
        mov   pc, lr // return value in r0
```

While statement (while.s)

- A python while loop that sums the first 16 integers is:

```
sum = 0
i = 0
while i <= 16 :
    sum += i
    i += 1
```

```
main:
        mov   r0, #0
        mov   r1, #1
top:
        cmp   r1, #16        // while test
        bgt   skip_while // python test is opposite
        add   r0, r0, r1     // while body
        add   r1, r1, #1
        b     top          // branch to top
skip_while:
        mov   pc, lr       // return value in r0
```


MEMORIAL UNIVERSITY
www.mun.ca

Counting Bits With While

- A while loop can be used to count the number of 1 bits in a register
- The equivalent python code is:

```
r1 = 0x3e80  // test input, random number
      (on the board use 6)
r0 = 0
while r1 != 0 :
      r0 += r1 & 1
      r1 = r1 >> 1
```

do-bits.s

```
main:
      mov   r0, #0
      mov   r1, #0x3e80
top:
      cmp   r1, #0     // while test
      beq   done // done when r1 == 0
      and   r2, r1, #1      // extract bit0
      add   r0, r0, r2      // count bit
      lsr   r1, r1, #1      // shift right
      b     top            // branch to top
done:
      mov   pc, lr     // return value in r0
```

Counting Bits With While

- and r2, r1, #1:
  - Set r2 to 1 if bit 0 of r1 was a 1
  - Otherwise it will be 0
- This value is added to r0
  - If it was a 1 then the 1 will be counted
- The lsr r1, r1, #1:
  - will eventual shift all the bits through bit 0, where they are counted

```
main:
        mov   r0, #0
        mov   r1, #0x3e80
top:
        cmp   r1, #0     // while test
        beq   done // done when r1 == 0
        and   r2, r1, #1      // extract bit0
        add   r0, r0, r2      // count bit
        lsr   r1, r1, #1      // shift right
        b     top            // branch to top
done:
        mov   pc, lr     // return value in r0
```

Count Bits With Do-While

- In Java and C, a do-while will always execute the loop once.

```
r1 = 0x3e80   // test input

r0 = 0

while True:

    r0 += r1 & 1

    r1 = r1 >> 1

    if r1 == 0 : break
```

```
main:

      mov  r0, #0

      mov  r1, #0x3e80

top:

      and  r2, r1, #1    // extract bit0

      add  r0, r0, r2    // count bit

      lsr  r1, r1, #1    // shift right

      cmp  r1, #0        // while test

      bne  top      // done when r1 == 0

done:

      mov  pc, lr // return r0
```

Count Bits With Do-While

- Notice there is only one branch and its test is opposite the test in the while example
- One test check if the loop should continue, the other test checks of the loop should stop

```
main:
        mov   r0, #0
        mov   r1, #0x3e80
top:
        and   r2, r1, #1      // extract bit0
        add   r0, r0, r2      // count bit
        lsr   r1, r1, #1      // shift right
        cmp   r1, #0          // while test
        bne   top         // done when r1 == 0
done:
        mov   pc, lr     // return r0
```

Questions?