

# Computer organization and architecture

## Lesson 5

### Branch instructions

A program usually executes in sequence, with the **program counter (PC)** incrementing by 4 after each instruction to point to the next instruction.

**Branch instructions** change the program counter.

Branches are also called **jumps** in some architectures.

ARM includes two types of branches:

B – a simple branch

BL – branch and link

BL is used for function calls

Branches can be unconditional or conditional.

## Unconditional branching

```
ADD R1, R2, #17      ; R1 = R2 + 17
B TARGET             ; branch to TARGET
ORR R1, R1, R3        ; not executed
AND R3, R1, #0xFF     ; not executed
TARGET                ; this is called the label
SUB R1, R1, #78       ; R1 = R1 - 78
```

## Conditional branching

```
MOV R0, #4           ; R0 = 4
ADD R1, R0, R0        ; R1 = R0 + R0 = 8
CMP R0, R1
BEQ THERE             ; branch not taken (Z != 1)
ORR R1, R1, #1        ; R1 = R1 OR 1 = 9
THERE
ADD R1, R1, #78       ; R1 = R1 + 78 = 87
```

## if Statements

### High-level

```
if (a == b)
    f = i + 1;

f = f - i;
```

### ARM assembly

```
CMP R0, R1 ; a == b ?
BNE L1     ; if !=, skip if block
ADD R2, R3, #1 ; if block

L1
SUB R2, R2, R3 ; f = f - i
```

### Another ARM assembly code

```
CMP R0, R1 ; a == b ?
ADDEQ R2, R3, #1 ; f = i + 1 on equality (Z = 1)
SUB R2, R2, R3 ; f = f - i
```

The last code is faster because it involves one fewer instruction.

Branches sometimes introduce extra delay, whereas conditional execution is always fast.

## if/else Statements

### High-level

```
if (a == b)
    f = i + 1;
else
    f = f - i;
```

### ARM assembly

```
CMP R0, R1 ; a == b?
BNE L1 ; if !=, skip if block
ADD R2, R3, #1 ; if block
B L2 ; skip else block
L1
SUB R2, R2, R3 ; else block
L2
```

### Another ARM assembly code

```
CMP R0, R1 ; a == b ?
ADDEQ R2, R3, #1 ; f = i + 1 on equality (Z = 1)
SUBNE R2, R2, R3 ; f = f - i on not equal (Z = 0)
```

# switch/case Statements

## High-level

```
switch (button) {  
    case 1: amt = 20;  
    break;  
  
    case 2: amt = 50;  
    break;  
  
    case 3: amt = 100;  
    break;  
  
    default: amt = 0;  
}
```

## ARM assembly

```
CMP R0, #1  
MOVEQ R1, #20  
BEQ DONE  
  
CMP R0, #2  
MOVEQ R1, #50  
BEQ DONE  
  
CMP R0, #3  
MOVEQ R1, #100  
BEQ DONE  
  
MOV R1, #0  
DONE
```

## while Loops


Determine the value of  $x$   
such that  $2^x = 128$

High-level

```
int pow = 1;
int x = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

## ARM assembly

```
MOV R0, #1           ; pow = 1
MOV R1, #0           ; x = 0
WHILE 
    CMP R0, #128      ; pow != 128 ?
    BEQ DONE          ; if pow == 128, exit loop
    LSL R0, R0, #1     ; pow = pow * 2
    ADD R1, R1, #1     ; x = x + 1
    B WHILE            ; repeat loop
DONE
```

## for Loops

Add the numbers  
from 0 to 9

High-level



```
int i;  
int sum = 0;  
  
for (i = 0; i < 10; i = i + 1) {  
    sum = sum + i;  
}
```

## ARM assembly

MOV R1, #0	; sum = 0
MOV R0, #0	; i = 0
FOR	
CMP R0, #10	; i < 10 ?
BGE DONE	; if (i >= 10) exit loop
ADD R1, R1, R0	; sum = sum + i loop body
ADD R0, R0, #1	; i = i + 1 loop operation
B FOR	; repeat loop
DONE	



# Example: Hailstone sequence

Starting with any positive whole number  $n$  form a sequence in the following way:

If  $n$  is even, divide it by 2 to give  $n' = n/2$

If  $n$  is odd, multiply it by 3 and add 1 to give  $n' = 3n + 1$

Take  $n'$  as the new starting number and repeat the process.

For example,  $n = 5$  gives the sequence

5, 16, 8, 4, 2, 1, 4, 2, 1, ...

Hailstone sequences eventually end in the endless cycle

4, 2, 1, 4, 2, 1, ...

ARM lacks any instructions related to division.

However,  $n / 2$  can be done with a right shift,  $n \gg 1$

Multiplying  $n$  by 3 is computed with a left shift and addition,  $n + (n \ll 1)$

Testing whether whether  $n$  is odd can be done by testing whether  $n$ 's LSB is set.

```
again  MOV    R0, #5                ; R0 is current number
      ANDS   R1, R0, #1            ; test whether R0 is odd
      BEQ    even
      ADD    R0, R0, R0, LSL #1    ; if odd, set R0=R0+(R0<<1)+1
      ADD    R0, R0, #1
      B      again                ; repeat (because R0>1)
even   MOV    R0, R0, ASR #1        ; if even, set R0 = R0 >> 1
      CMP    R0, #1
      BNE    again                ; repeat if R0 != 1
```

# A pipeline diagram for the ARM7TDMI

Cycle		1	2	3	4	5		
Address	Operation							
0x8000	BL	Fetch	Decode	Execute	Linkret	Adjust		
0x8004	X		Fetch	Decode				
0x8008	XX			Fetch				
0x8FEC	ADD				Fetch	Decode	Execute	
0x8FF0	SUB					Fetch	Decode	Execute
0x8FF4	MOV						Fetch	

The three-stage pipeline can fetch one instruction from memory, decode another instruction, and execute a third instruction, all in the same clock cycle.

An ADD, SUB, and MOV instruction presents no problems, since there is nothing present that would cause an instruction to force the processor to wait for it to complete.

# A pipeline diagram for the ARM7TDMI

Cycle			1	2	3	4	5
<u>Address</u>	<u>Operation</u>						
0x8000	BL	Fetch	Decode	Execute	Linkret	Adjust	
0x8004	X	_____	Fetch	Decode			
0x8008	XX	_____	_____	Fetch			
0x8FEC	ADD	_____	_____	_____	Fetch	Decode	Execute
0x8FF0	SUB	_____	_____	_____	_____	Fetch	Decode
0x8FF4	MOV	_____	_____	_____	_____	_____	Fetch

A branch instruction will cause the entire pipeline to be flushed.

A branch instruction effectively tells the machine to start fetching new instructions from a different address in memory.

# A pipeline diagram for the ARM7TDMI

Cycle		1	2	3	4	5		
Address	Operation							
0x8000	BL	Fetch	Decode	Execute	Linkret	Adjust		
0x8004	X		Fetch	Decode				
0x8008	XX			Fetch				
0x8FEC	ADD				Fetch	Decode	Execute	
0x8FF0	SUB					Fetch	Decode	Execute
0x8FF4	MOV						Fetch	

in cycle 1, the branch (BL) has entered the Execute stage of the pipeline, and two instructions have already been fetched (one from address 0x8004 and one from 0x8008).

Since the branch says to begin fetching new instructions from address 0x8FEC, those unused instructions must be thrown away.

In a three-stage pipeline, the effects are not nearly as deleterious, but consider what would happen in a very deep pipeline, say 24 stages.

A branch that is not handled correctly could force the processor to abandon significant amounts of work.

In many algorithms, especially signal processing algorithms, speed is the most important consideration in its implementation.

If any delays can be removed from the code, even at the expense of memory, then sometimes they are.

Instructions that are between the start of a loop and the branch back to the beginning can be repeated many times, a process known as **unrolling a loop**.

For example, if you had one instruction that was inside of a for loop, that is,

```
MOV    r1, #10                ; j = 10
Loop
  MLA   r3, r2, r4, r5        ; r3 = r2*r4 + r5
  SUBS  r1, r1, #1            ; j = j - 1
  BNE   Loop                  ; if j = 0, finish
```

you could do away with the for loop entirely by simply repeating the MLA instruction 10 times.

The routine may be significantly faster but it will occupy more memory because of the repeated instructions.

## Exercise 5.1

Design an algorithm for counting the number of 1's in a 32-bit number.

Implement your algorithm using ARM assembly code.

## Exercise 5.2

Write ARM assembly code to reverse the bits in a register.

$$d_{31} d_{30} d_{29} \dots d_1 d_0 \rightarrow d_0 d_1 d_2 \dots d_{30} d_{31}$$

Use as few instructions as possible.

Check:

0xAB3C71FF should be transformed into 0xFF8E3CD5



## Exercise 5.3

Each number in the Fibonacci series is the sum of the previous two numbers.

$n$	1	2	3	4	5	6	7	8	9
$fib(n)$	1	1	2	3	5	8	13	21	34

Implement this series using ARM assembly code.

Put  $n$  in R2, and  $fib(n)$  in R3