## Branch Instructions

Standard branch instructions, \( B<\text{suffix}> \) and \( BL<\text{suffix}> \), change the PC based on the PCR. The next instruction's address is found by adding a 24-bit signed 2's complement immediate value multiplied by 4 to the PC+8, giving a range of +/- 32 Mbytes. Larger branches use the \( BX<\text{suffix}> \) instruction, where the next instruction's address is from a register.

### B type:

<table>
<thead>
<tr>
<th>Cond</th>
<th>101</th>
<th>L</th>
<th>Imm24</th>
</tr>
</thead>
</table>

### R type:

| Cond | 000 | 1 | 0010 1111 1111 1111 0001 | Rn |

0000 - EQ  - equals
0001 - NE  - not equals
0010 - CS  - carry set
0011 - CC  - carry clear
0100 - MI  - negative
0101 - PL  - positive or zero
0110 - VS  - overflow
0111 - VC  - no overflow
1000 - HI  - higher (unsigned)
1001 - LS  - lower or same (unsigned)
1010 - GE  - greater or equal (signed)
1011 - LT  - less than (signed)
1100 - GT  - greater than (signed)
1101 - LE  - less than or equal (signed)
1110 - "" - always

If the condition is true, the PC is changed by either adding the immediate value to PC+8, or setting it to the contents of Rn.

BTW, BX is encoded as a TEQ instruction with its S field set to "0".
Branch Examples

Bne  else

If some previous CMP instruction had a non-zero result (i.e. making the 'Z' bit 0 in the PSR), then this instruction will cause the PC to be loaded with the address having the label "else".

BLeq  func

If some previous CMP instruction set the 'Z' bit in the PSR, then this instruction will cause the PC to be loaded with the address having the label "func", and the address of the following instruction will be saved in R14.

BX    LR

"Always" loads the PC with the contents of R14. This is called an "unconditional" branch.

NOTE: ARM assemblers are "case-insensitive" with regard to instruction mnemonics. I am mixing upper and lower case here to emphasize the <suffix> component of the instruction.

loop: B loop

An infinite loop.

BTW: This instruction is encoded as: 0xEAFFFFFE

```
1110 101 0 1111 1111 1111 1111 1111 1110
```

"always"  B-type format  Don't save the return address  -2, which implies PC + 8 - 2\*4 = PC
A simple Program

; Assembly code for; sum = 0;
; for (i = 0; i <= 10; i++)
; sum = sum + i;

main:    mov   R1,#0 ; R1 is i
         mov   R0,#0 ; R0 is sum
loop:    add   R0,R0,R1 ; sum = sum + i
         add   R1,R1,#1 ; i++
         cmp   R1,#10 ; i <= 10
         ble   loop
halt:    b      halt

You will notice that the miniARM simulator, works like an actual processor... meaning that the first two words in memory, must be preloaded with two addresses, the first is an initial value for R13 (SP), and the second is an initial value for R15 (PC)
**Load and Stores in Action**

An example of how loads and stores are used to access arrays.

**Java/C:**

```java
int x[10];
int sum = 0;
for (int i = 0; i < 10; i++)
    sum += x[i];
```

**Assembly:**

```assembly
main:  mov     r0,#x    ; base of x
         mov     r1,#sum
         ldr     r2,[r1]
         mov     r3,#0    ; r3 is i
for:    ldr     r4,[r0,r3,lsl #2]
        add     r2,r2,r4
        add     r3,r3,#1
        cmp     r3,#10
        blt     for
        str     r2,[r1]
halt:   b       halt
```

*miniARM*

In addition to instructions and labels, assemblers also allow for certain "directives", like "word" and "space" that initialize memory, allocate space, and set the address where instructions should be loaded.
Next Time

We’ll write more Assembly programs

Still some loose ends

- Multiplication? Division? Floating point?
ASSEMBLING THE LAST FEW BITS

- Multiplication
- Division
- Block transfers
- Calling procedures
- Usage conventions

Need to get back in stride... Expect some schedule changes to accommodate Florence.

Friday’s class meeting will be part Lecture, part Lab.

Problem Set #1 is due before midnight (9/19)
Some "odd" instructions

The ARM multiply instruction was kind of an afterthought. It is "shoe-horned-in" using unused R-type encodings.

R type:  

<table>
<thead>
<tr>
<th>4</th>
<th>3</th>
<th>4</th>
<th>4</th>
<th>1</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>000</td>
<td>000 A</td>
<td>S</td>
<td>Rd</td>
<td>Rn</td>
<td>Rs</td>
<td>1001</td>
<td>Rm</td>
</tr>
</tbody>
</table>

You may recall that R-type instructions with included shifts always required bit 4 to be '0'. If bit 4 is a '1', several new instructions emerge.

All operands of multiply instructions are assumed to be 2's-complement integers.

if $A = 0$

MUL Rd,Rm,Rs ; Rd = Rm*Rs

if $A = 1$

MLA Rd,Rm,Rs,Rn ; Rd = Rm*Rs+Rn

Also, notice that for some odd reason, they swapped the meaning of the Rd and Rn fields.
DIVISION, NOT ONE

ARMv7 does not provide a DIVIDE instruction. Reasons?

1. Divisions often require multiple cycles
2. Integer divisions provide two results, a quotient and a remainder
3. Divisions by known constants can be implemented via multiplication and shifts
4. In floating point $1/y$ is easy to compute, so the product $x/y = x*(1/y)$ is often the implementation of choice
5. Usually implemented as a function.
Another "odd" instruction

ARM also provides an instruction that swaps the contents of registers with a memory location.

```
R type: 1110 000 10B0 0 Rn  Rd  0000 1001 Rm
```

Swap is used to implement synchronization primitives that are used by multiple processors and threads. The instruction is "atomic."

```
SWP Rd,Rm,[Rn] ; Rd <-- Memory[Rn] ; Memory[Rn] <-- Rm
```

The "B" bit when '0' swaps a word, and when '1', it swaps a byte.

Rd and Rn are back in their usual places.
Block Transfers

Arm provides a useful instruction for storing multiple registers into memory sequentially. It shares some commonality with the LDR and STR instructions.

<table>
<thead>
<tr>
<th>B type:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>P</td>
</tr>
<tr>
<td>U</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>L</td>
</tr>
<tr>
<td>Rn</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Register Vector</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L</th>
<th>P</th>
<th>U</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>LDMFD Rn!, {list of regs} ; save regs to increasing addresses</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SRMFD Rn!, {list of regs} ; load regs from decreasing addresses</td>
</tr>
</tbody>
</table>

Examples:

SRMFD SP!, {R4, R5, R6, LP}

...  

LRMFD SP!, {R4, R5, R6, PC}
Conditional Execution

Recall how branch instructions could be executed conditionally, based on the status flags set from some previous instruction. Also recall that, while condition flags are generally set using CMP or TST instructions, many instructions can be used to set status flags. Actually, there is full symmetry. Most instructions, in addition to branches can also be executed conditionally.

R type: Cond 000 Opcode S Rn Rd Shift | L A 0 Rm
I type: Cond 001 Opcode S Rn Rd Rotate Imm8
D type: Cond 010 1 U 0 0 L Rn Rd Imm12
X type: Cond 011 1 U 0 0 L Rn Rd Shift | L A 0 Rm
B type: Cond 101 L Imm24

0000 - EQ - equals
0001 - NE - not equals
0010 - CS - carry set
0011 - CC - carry clear
0100 - MI - negative
0101 - PL - positive or zero
0110 - VS - overflow
0111 - VC - no overflow
1000 - HI - higher (unsigned)
1001 - LS - lower or same (unsigned)
1010 - GE - greater or equal (signed)
1011 - LT - less than (signed)
1100 - GT - greater than (signed)
1101 - LE - less than or equal (signed)
1110 - "" - always
Example of Conditional Execution

```assembly
CMP    R3, R4 ; if (i >= j)
BLT    else ;
SUB    R0, R3, R4 ; x = i - j;
B      endif ; else
else: SUB    R0, R4, R3 ; x = j - i;
endif:

CMP    R3, R4 ; x = (i >= j) ? i - j : j - i;
SUBGE  R0, R3, R4 ;
SUBLT  R0, R4, R3 ;
```

This code is not only shorter, but it is much faster. Generally, taken branches are slower than ALU instructions on ARM.
Supporting procedure Calls

Functions and procedures are essential components of code reuse. They also allow code to be organized into modules. A key component of procedures is that they clean up behind themselves.

Basics of procedure calling:

1. Put parameters where the called procedure can find them
2. Transfer control to the procedure
3. Acquire the needed storage for procedure variables
4. Perform the expected calculation
5. Put the result where the caller can find them
6. Return control to the point just after where it was called
Register usage conventions

By convention, the ARM registers are assigned to specific uses and names. These are supported by the assembler, and higher-level languages. We’ll use these names increasingly. Why have such conventions?

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0-R3</td>
<td>First 4 function arguments. Return values are placed in R0 and R1.</td>
</tr>
<tr>
<td>R4-R10</td>
<td>Saved registers. Must save before using and restore before returning.</td>
</tr>
<tr>
<td>R11</td>
<td>FP - Frame pointer (to access a procedure’s local variables)</td>
</tr>
<tr>
<td>R12</td>
<td>IP - Temp register used by assembler</td>
</tr>
<tr>
<td>R13</td>
<td>SP - Stack pointer Points to next available word</td>
</tr>
<tr>
<td>R14</td>
<td>LR - Link Register (return address)</td>
</tr>
<tr>
<td>R15</td>
<td>PC - program counter</td>
</tr>
</tbody>
</table>
Basics of Calling

int gcd(a, b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}

int x = 35;
int y = 55;
int z;

z = gcd(x, y);

Here the assembly language version is actually shorter than the C/Java version.

main:   ldr    r0, x
        ldr    r1, y
        b    GCD
        str   r0, z
        halt

halt:   b    halt

x:      .word   35
y:      .word   55
z:      .word   0

GCD:    cmp    r0, r1
        bxeq   lr
        subgt  r0, r0, r1
        sublt  r1, r1, r0
        b    GCD

Doesn’t that require division? I thought ARM7 doesn’t have a division instruction? Thanks, Euclid!
That was a little too EASY

```assembly
main:  ldr r0,x
       bl fact
       str r0,y
halt:  b halt

x:     .word 5
y:     .word 0

fact:  cmp r0,#1
       bxle lr
       mov r4,r0
       sub r0,r0,#1
       bl fact
       mul r0,r0,r4
       bx lr

int fact(x) {
    if (x <= 1)
        return x;
    else
        return x*fact(x-1);
}

int x = 5;
int y;
y = fact(x);
```

This time, things are really messed up.
The recursive call to fact() overwrites the value of x that was saved in R4.

To make a bad thing worse, the LR is also overwritten.

I knew there was a reason that I avoid recursion.
Next Time

- Stacks
- Contracts
- Writing serious assembly code