Assembling the last few bits

- Multiplication
- Division
- Block transfers
- Calling procedures
- Usage conventions
Some "Odd" Instructions

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

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.

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

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

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>
<th>4</th>
<th>3</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>4</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 100</td>
<td></td>
<td>P</td>
<td>U</td>
<td>0</td>
<td>1</td>
<td>L</td>
<td>Rn</td>
<td></td>
<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.

<table>
<thead>
<tr>
<th>R type:</th>
<th>Cond</th>
<th>000</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Shift</th>
<th>LA</th>
<th>0</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>I type:</td>
<td>Cond</td>
<td>001</td>
<td>Opcode</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>Rotate</td>
<td>Imm8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D type:</td>
<td>Cond</td>
<td>010</td>
<td>1</td>
<td>U</td>
<td>0</td>
<td>0</td>
<td>L</td>
<td>Rn</td>
<td>Rd</td>
<td>Imm12</td>
</tr>
<tr>
<td>X type:</td>
<td>Cond</td>
<td>011</td>
<td>1</td>
<td>U</td>
<td>0</td>
<td>0</td>
<td>L</td>
<td>Rn</td>
<td>Rd</td>
<td>Shift</td>
</tr>
<tr>
<td>B type:</td>
<td>Cond</td>
<td>101</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Imm24</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 0000 - EQ - equals
- 0001 - NE - not equals
- 0010 - CS - carry set
- 0011 - CC - carry clear (signed)
- 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
- 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>LP - Link Pointer (return address)</td>
</tr>
<tr>
<td>R15</td>
<td>PC - program counter</td>
</tr>
</tbody>
</table>
Basics of Calling

```
LDR R0, x
LDR R1, y
BL GCD
STR R0, z
```

```
halt: B  halt
```

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

```
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.
That was a little too EASY

LDR R0, x
BL fact
STR R0, y
halt: B halt

x: .word 5
y: .word 0

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 R1.
To make a bad thing worse, the LP is also overwritten.
I knew there was a reason that I avoid recursion.
Next Time

- Stacks
- Contracts
- Writing serious code