Assembling the last few bits

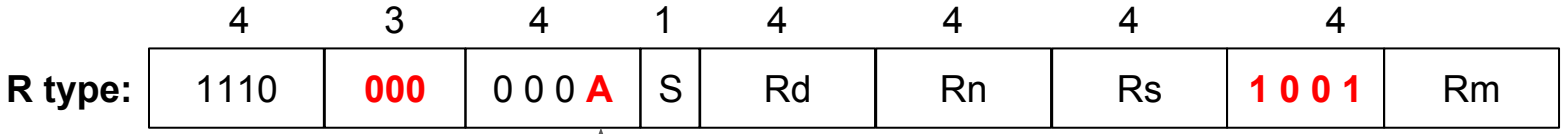
- Multiplication
- Division
- Block transfers
- Calling procedures
- Usage conventions

Grades for Labs 1 and 2 should be posted.

Problem Set #1 due midnight Wed (9/20)

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.

Also, notice that for some odd reason, they swapped the meaning of the Rd and Rn fields

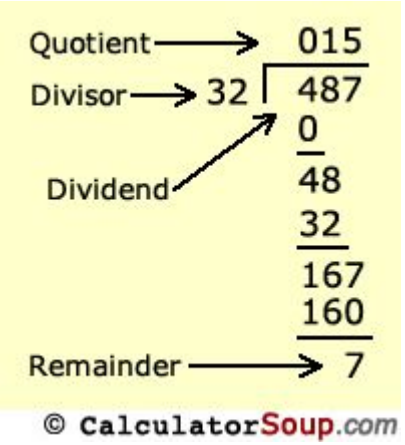
All operands of multiply instructions are assumed to be 2's-complement integers.

```
if A == 0
    MUL Rd,Rm,Rs      ; Rd = Rm*Rs

if A == 1
    MLA Rd,Rm,Rs,Rn   ; Rd = Rm*Rs+Rn
```

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.

	4	3	4	1	4	4	4	4	
R type:	1110	**000**	1 0 **B** 0	0	Rn	Rd	0 0 0 0	**1 0 0 1**	Rm

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

Rd and Rn are back in their usual places

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
```

Block Transfers

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

	4	3	1	1	1	1	1	4	16
B type:	1110	**100**	P	U	0	1	L	Rn	Register Vector

L	P	U	Instruction
1	0	1	`LDMFD  Rn!,{list of regs}` ; save regs to increasing addresses
0	1	0	`SRMFD  Rn!,{list of regs}` ; load regs from decreasing addresses

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***.

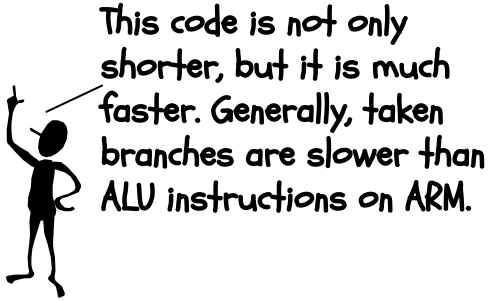
R type:	Cond	000	Opcode				S	Rn	Rd	Shift		L A	0	Rm
I type:	Cond	001	Opcode				S	Rn	Rd	Rotate				Imm8
D type:	Cond	010	1	U	0	0	L	Rn	Rd	Imm12				
X type:	Cond	011	1	U	0	0	L	Rn	Rd	Shift		L A	0	Rm
B type:	Cond	101	L	Imm24										

0000 - **EQ - equals**
0001 - **NE - not equals**
0010 - **CS - carry set**
0011 - **CC - carry clear**
(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**

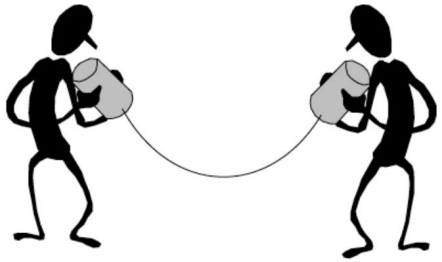
Example of Conditional Execution

```
        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   ;
```



Supporting Procedure Calls

Functions and procedures are essential components of code reuse. The also allow code to be organized into modules. A key component of 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?

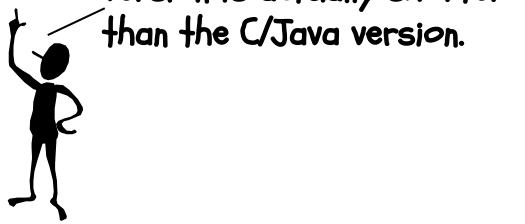
Register	Use
R0-R3	First 4 function arguments. Return values are placed in R0 and R1.
R4-R10	Saved registers. Must save before using and restore before returning.
R11	FP - Frame pointer (to access a procedure's local variables)
R12	IP - Temp register used by assembler
R13	SP - Stack pointer Points to next available word
R14	LP - Link Pointer (return address)
R15	PC - program counter

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
```

```
GCD:    CMP     R0,R1
        BXEQ    LP
        SUBGT   R0,R0,R1
        SUBLT   R1,R1,R0
        B       GCD
```

```
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);
```



That was a little too easy

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

x:    .word 5
y:    .word 0
```

```
fact: CMP   R0,#1
      BXLE  LP
      MOV   R1,R0
      SUB   R0,R0,#1
      BL    fact
      MUL   R0,R0,R1
      BX    LP
```

```
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 Rl.

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