

STATUS FLAGS



Now it is time to discuss what *status flags* are available. These five status flags are kept in a special register called the Program Status Register (PSR). The PSR also contains other important bits that control the processor.

- **N** - set if the result of an operation is negative (Most Significant Bit (MSB) is a 1)
- **Z** - set if the result of an operation is "0"
- **C** - set if the result of an operation has a carry out of it's MSB
- **V** - set if a sum of two positive operands gives a negative result, or if the sum of two negative operands gives a positive result
- **Q** - a sticky version of overflow created by instructions that generate multiple results (more on this later on).

COMPARISON INSTRUCTIONS



These instructions modify the status flags, but leave the contents of the registers unchanged. They are used to test register contents, and they **must** have their "S" bit set to "1". They also don't modify their Rd, and by **convention**, Rd is set to "0000".

	4	3	4	^S 1	4	4	8	4
R type:	1110	000	Opcode	1	Rn	0000	00000000	Rm
I type:	1110	001	Opcode	1	Rn	0000	Rotate	Imm8

CMP R0, R1



PSR flags set for the result R0 - R1

1000 - TST
1001 - TEQ
1010 - CMP
1011 - CMN

CMN R2, R3



PSR flags set for the result R2 + R3

TST R4, #8



PSR flags set for the result R4 & 8

TEQ R5, #1024



PSR flags set for the result R5 ^ 1024

REGISTER TRANSFER




These instructions are used to transfer the contents of one register to another, or simply to initialize the contents of a register. They make use of only one operand, and, by convention, have their Rn field set to "0000".


MOV R0, R3  R0 ← R3

MOV R1, #4096  R1 ← 4096

MVN R2, R4  R2 ← - R4

MVNS R3, #1  R3 ← - 1, and set PSR (Z = 0, N = 1, V = 0, C = 0)

	4	3	4	1	4	4	8	4
R type:	1110	000	Opcode	S	0000	Rd	00000000	Rm
I type:	1110	001	Opcode	S	0000	Rd	Rotate	Imm8

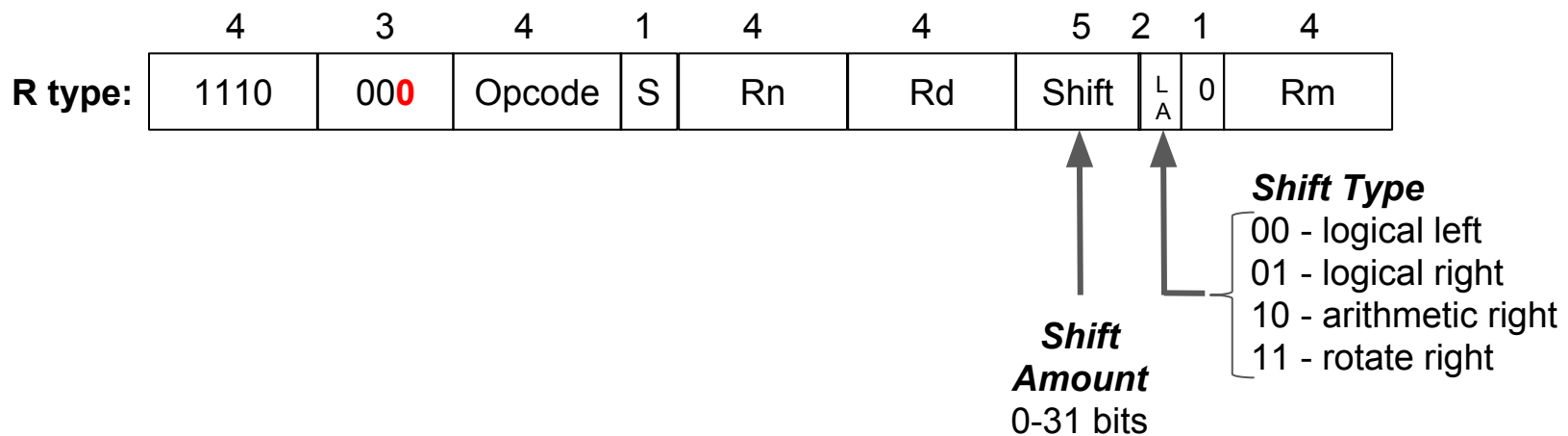


 1101 - MOV
 1111 - MVN

ARM SHIFT OPERATIONS



A novel feature of ARM is that *all* data-processing instructions can include an optional "shift", whereas most other architectures have separate shift instructions. This is actually very useful as we will see later on. The key to shifting is that 8-bit field between Rd and Rm.





LEFT SHIFTS

Left shifts effectively multiply the contents of a register by 2^s where s is the shift amount.

MOV R0, R0, LSL 7

R0 before:	<table border="1"><tr><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0111</td></tr></table>	0000	0000	0000	0000	0000	0000	0000	0111	= 7
0000	0000	0000	0000	0000	0000	0000	0111			
R0 after:	<table border="1"><tr><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0011</td><td>1000</td><td>0000</td></tr></table>	0000	0000	0000	0000	0000	0011	1000	0000	= $7 * 2^7 = 896$
0000	0000	0000	0000	0000	0011	1000	0000			

Shifts can also be applied to the second operand of any data processing instruction

ADD R1, R1, R0, LSL 7



RIGHT SHIFTS

Right Shifts behave like *dividing* the contents of a register by 2^s where s is the shift amount, *if* you assume the contents of the register are *unsigned*.

MOV R0, R0, LSR 2

R0 before:

0000	0000	0000	0000	0000	0100	0000	0000
------	------	------	------	------	------	------	------

= 1024

R0 after:

0000	0000	0000	0000	0000	0001	0000	0000
------	------	------	------	------	------	------	------

= $1024 / 2^2 = 256$

ARITHMETIC RIGHT SHIFTS



Arithmetic right shifts behave like *dividing* the contents of a register by 2^s where s is the shift amount, *if* you assume the contents of the register are *signed*.

MOV R0, R0, ASR 2

R0 before:	<table border="1"><tr><td>1111</td><td>1111</td><td>1111</td><td>1111</td><td>1111</td><td>1100</td><td>0000</td><td>0000</td></tr></table>	1111	1111	1111	1111	1111	1100	0000	0000	= -1024	
1111	1111	1111	1111	1111	1100	0000	0000				
R0 after:	<table border="1"><tr><td>11</td><td>11</td><td>1111</td><td>1111</td><td>1111</td><td>1111</td><td>1111</td><td>0000</td><td>0000</td></tr></table>	11	11	1111	1111	1111	1111	1111	0000	0000	= -1024 / 2^2 = -256
11	11	1111	1111	1111	1111	1111	0000	0000			

Note: A red arrow points from the 6th bit of the 'before' register to the 6th bit of the 'after' register, indicating a right shift by 2 positions.

This is Java's ">>>" operator,
LSR is ">>" and LSL is "<<"

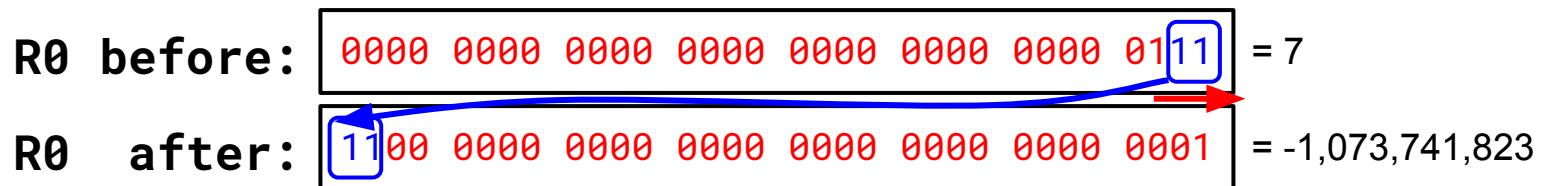


ROTATE RIGHT SHIFTS



Rotating shifts have no arithmetic analogy. However, they don't lose bits like both logical and arithmetic shifts. We saw rotate right shift used for the I-type "immediate" value earlier.

MOV R0, R0, ROR 2



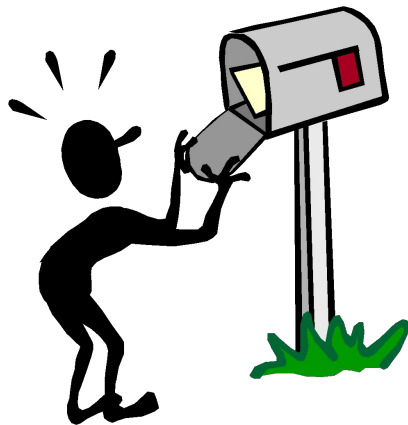
Java doesn't have an operator for this one.

Why no rotate left shift?

- Ran out of encodings?
- Almost anything Rotate lefts can do ROR can do as well!



ADDRESSING MODES AND BRANCHES



- More on Immediates
- Reading and Writing Memory
- Registers holding addresses
- Pointers
- Changing the PC
 - Loops
 - Labels
 - Calling Functions

WHY BUILT-IN CONSTANT OPERANDS?

(IMMEDIATES)



	4	3	4	1	4	4	4	8
I type:	1110	001	Opcode	S	Rn	Rd	Rotate	Imm8

- Alternatives? Why not? Do we have a choice?
 - put constants in memory (was common in older instruction sets)
- SMALL constants are used frequently (50% of operands)
 - In a C compiler (gcc) 52% of ALU operations involve a constant
 - In a circuit simulator (spice) 69% involve constants
 - e.g., $B = B + 1$; $C = W \& 0xff$; $A = B - 1$;

- ISA Design Principle:

Make the common case easy
Make the common case fast

How large of constants should we allow for? If they are too big, we won't have enough bits leftover for the instructions or operands.





ROTATIONS TO MAKE CONSTANTS

Recall that immediate constants are encoded in two parts:
Thus, only a subset of 4096 32-bit numbers can
be used directly as an
operand.

There are actually
only **3073** distinct
constants. There are
16, "0s" and 4 ways to
represent all powers 2.

From last time, how
might you encode 256?

1100	00000001	1101	00000100
------	----------	------	----------

1110	00010000	1111	01000000
------	----------	------	----------

09/13/2017

4	8
Rotate	imm

Rotate	Bits Used	Range
0		0 - 255
1		-2147483648 - 1073741887
2		-2147483648 - 1879048207
3		-2147483648 - 2080374787
4		-2147483648 - 2130706432
5		4194304 - 1069547520
6		1048576 - 267386880
7		262144 - 66846720
8		65536 - 16711680
9		16384 - 4177920
10		4096 - 1044480
11		1024 - 261120
12		256 - 65280
13		64 - 16320
14		16 - 4080
15		4 - 1020

MOVES AND ORS



We can load any 32-bit constant using a series of instructions, one-byte at a time.

```
MOV R0, #85 ; 0x55 in hex
ORR R0, R0, #21760 ; 0x5500 in hex
ORR R0, R0, #5570560 ; 0x550000 in hex
ORR R0, R0, #1426063360 ; 0x55000000 in hex
```

But there are often better, faster, ways to load constants, and the assembler can figure out how for you, even if it needs to generate multiple instructions.

```
MOV R0, =1431655765 ; 0x55555555 in hex
```

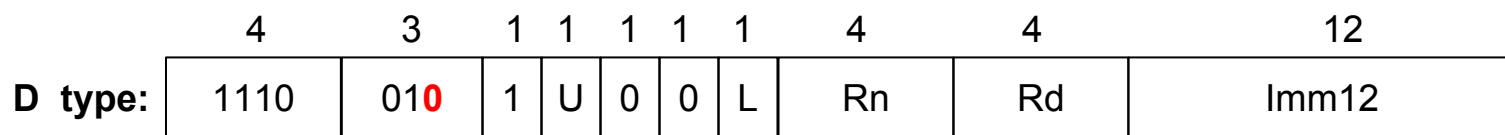
Note that an
equal sign is used
here rather than
a hashtag.



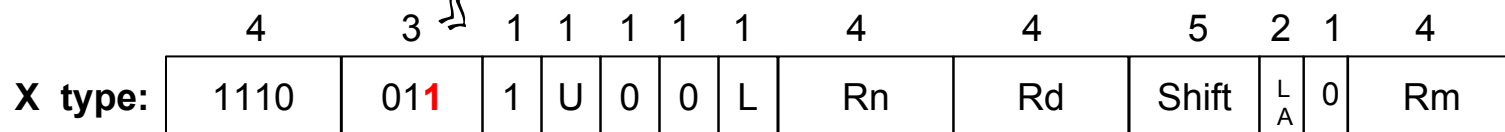
LOAD AND STORE INSTRUCTIONS



ARM is a "Load/Store architecture". That means that only a special class of instructions are used to reference data in memory. As a rule, data is loaded into registers first, then processed, and the results are written back using stores. Load and Store instructions have their own format:



Why does a "1" imply an immediate operand for ALU types, but "0" for Loads and Stores?



If U is "0" subtract offset from base, otherwise add them.



L is a "1" for a Load and "0" for a Store









The same "shift" options that we saw for the data processing instructions





LOAD AND STORE OPTIONS

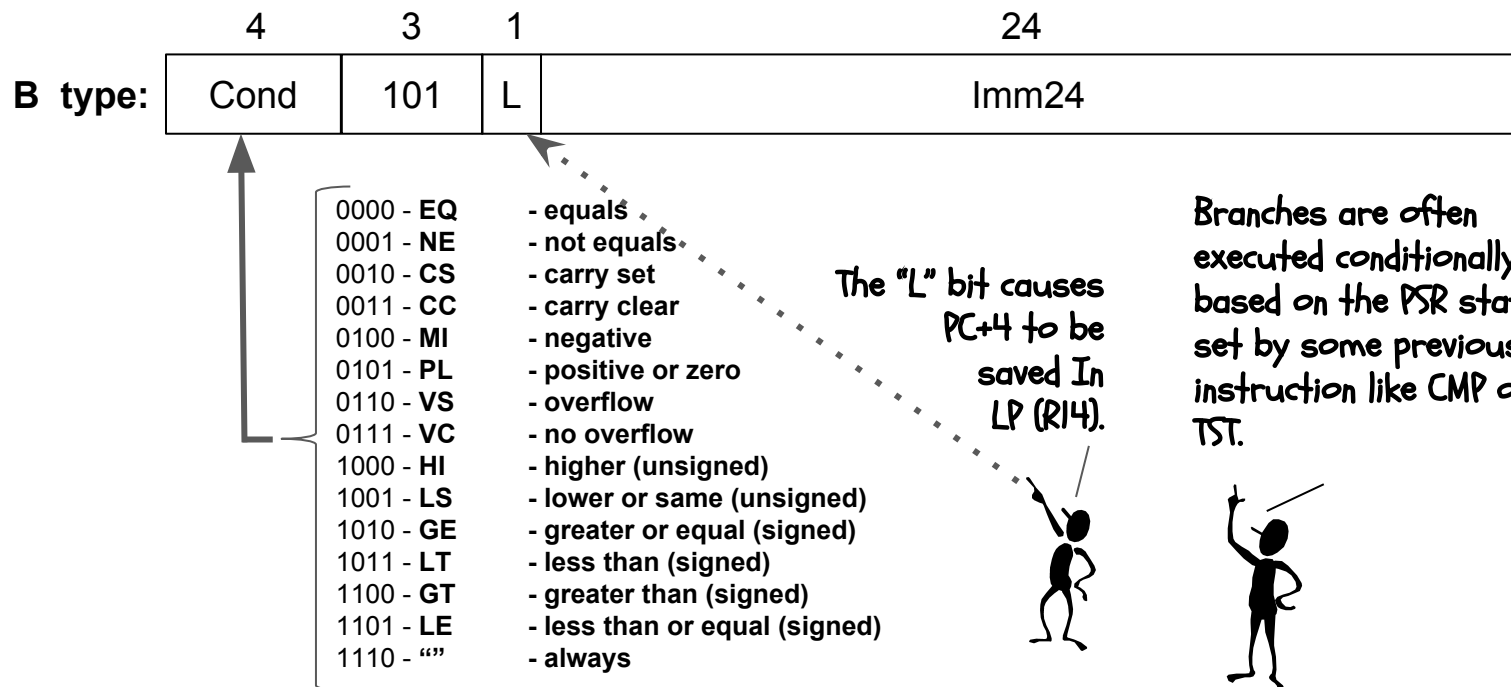
ARM's load and store instructions are versatile. They provide a wide range of addressing modes. Only a subset is shown here.

- LDR $R_d, [R_n, \#imm12]$  $R_d \leftarrow \text{Memory}[R_n + imm12]$
 R_d is loaded with the contents of memory at the address found by adding the contents of the base register to the supplied constant
- STR $R_0, [R_1, \#-4]$  $\text{Memory}[R_1 - 4] \leftarrow R_0$
 Offsets can be either added or subtracted, as indicated by a negative sign
- LDR $R_2, [R_3]$  If no offset is specified it is assumed to be zero
- STR $R_4, [R_5, R_6]$  The contents of a second register can be used as an offset rather than a constant (using the X-type format)
- LDR $R_4, [R_5, -R_6]$  Register offsets can be either added or subtracted, like constants
- STR $R_4, [R_5, R_4, LSL \ 2]$  Register offsets can also be optionally shifted, which is great for indexing arrays!

CHANGING THE PC



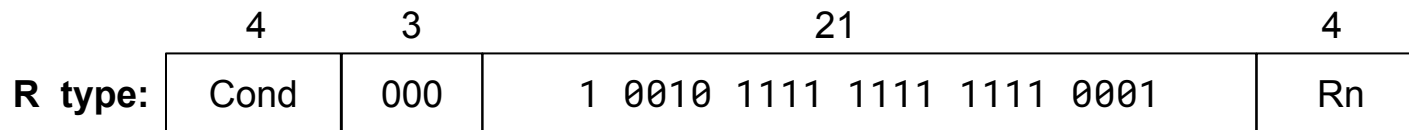
The Program Counter is special register (R15) that tracks the address of the next instruction to be fetched. There are special instructions for changing the PC.



BRANCH USING REGISTERS



The standard Branch instruction has a limited range, the 24-bit signed 2's complement immediate value is multiplied by 4 and added to the PC+8, giving a range of +/- 32 Mbytes. Larger branches make use of addresses previously loaded into a register using the **BX** instruction.



- 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 loaded with 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'.

BEQL 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



Loads the PC with the contents of R14.

loop: B loop



An infinite loop



A SIMPLE PROGRAM

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

LOAD AND STORES IN ACTION



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

Java/C:

```
int x[10];
int sum = 0;

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

Assembly:

```
.align 4
x:      .space 40
sum:    .word 0

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]
```

NEXT TIME



We'll write more Assembly programs

Still some loose ends

- Multiplication? Division? Floating point?

