Basic ARM Instructions

- Instructions include various "fields" that encode combinations of **Opcodes** and **arguments**
- special fields enable extended functions (more in a minute)
- several 4-bit OPERAND fields, for specifying the sources and destination of the operation, usually one of the 16 registers
- Embedded constants ("immediate" values) of various sizes,

The "basic" data-processing instruction formats:

	4	3	4	1	4	4	8	4
R type:	1110	00**0**	Opcode	0	Rn	Rd	00000000	Rm

	4	3	4	1	4	4	4	8
I type:	1110	00**1**	Opcode	0	Rn	Rd	Shift	Imm

R-type Data Processing

Instructions that process three-register arguments:

	4	3	4	1	4	4	8	4
R type:	1110	000	Opcode	S	Rn	Rd	00000000	Rm

0000 - AND
0001 - EOR
0010 - SUB
0011 - RSB
0100 - ADD
0101 - ADC
0110 - SBC
0111 - RSC
1000 - TST
1001 - TEQ
1010 - CMP
1011 - CMN
1100 - ORR
1101 - MOV
1110 - BIC
1111 - MVN

Simple R-type instructions follow the following template:

OP Rd, Rn, Rm

Later on we'll introduce more complex variants of these "simple" R-type instructions.

ADD R0, R1, R3

Is encoded as:

1110 0000 1000 0001 0000 0000 0000 0011

0xE0810003

I-type Data Processing

Instructions that process two registers and a constant:

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

0000 - AND
0001 - EOR
0010 - SUB
0011 - RSB
0100 - ADD
0101 - ADC
0110 - SBC
0111 - RSC
1000 - TST
1001 - TEQ
1010 - CMP
1011 - CMN
1100 - ORR
1101 - MOV
1110 - BIC
1111 - MVN

Simple I-type instructions follow the following template:

OP Rd, Rn, #constant

In the I-type instructions the second register operand is replaced by a constant that is encoded in the instruction

RSB R7, R10, #49

Is encoded as:

1110 0010 0110 1010 0111 0000 0011 0001

0xE26A7031

I-TYPE CONSTANTS

ARM7 provides only 8-bits for specifying an immediate constant value. Given that ARM7 is a 32-bit architecture, this may appear to be a severe limitation. However, by allowing for a shift (actually a "***right rotation***") to be applied to the constant.

```
imm32 = (imm8 >> (2 * shift)) | (imm8 << (32 - (2 * shift)))
```

Example: 1920 is encoded as:

Shift	Imm8
1101	00011110

= (30 >> (2*13)) | (30 << (32 - (2*13)))

= 0 | 30 * 64

= 1920

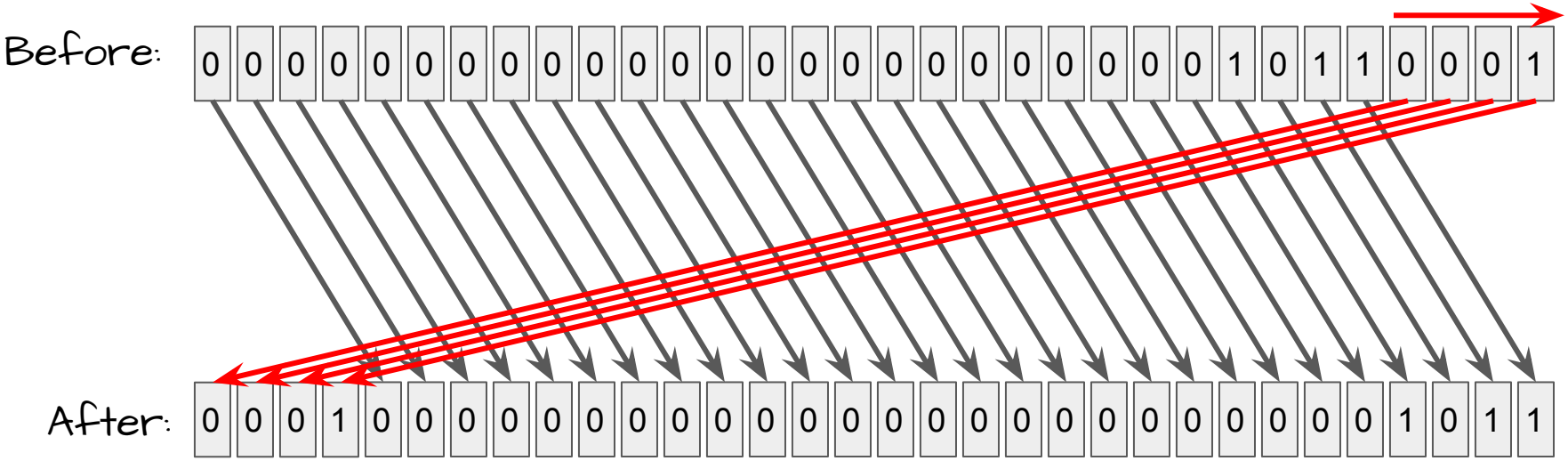
How would 256 be encoded?

Shift	Imm8
1100	00000001

Illustrating a right rotation

The "**shift**" field of the I-type instruction is a misnomer. It is acually a "**rotate-right**". What's a rotate right?

0xB1 (209) rotated 4 postions to the right



0x10000009 (268,435,465)

ARM IMMEDIATE CONSTANTS

Recall that immediate constants are encoded in two parts:

4	8
Shift	imm

Some constants can be encoded in multiple ways.

Thus fewer than 4096 32-bit numbers can be represented.

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

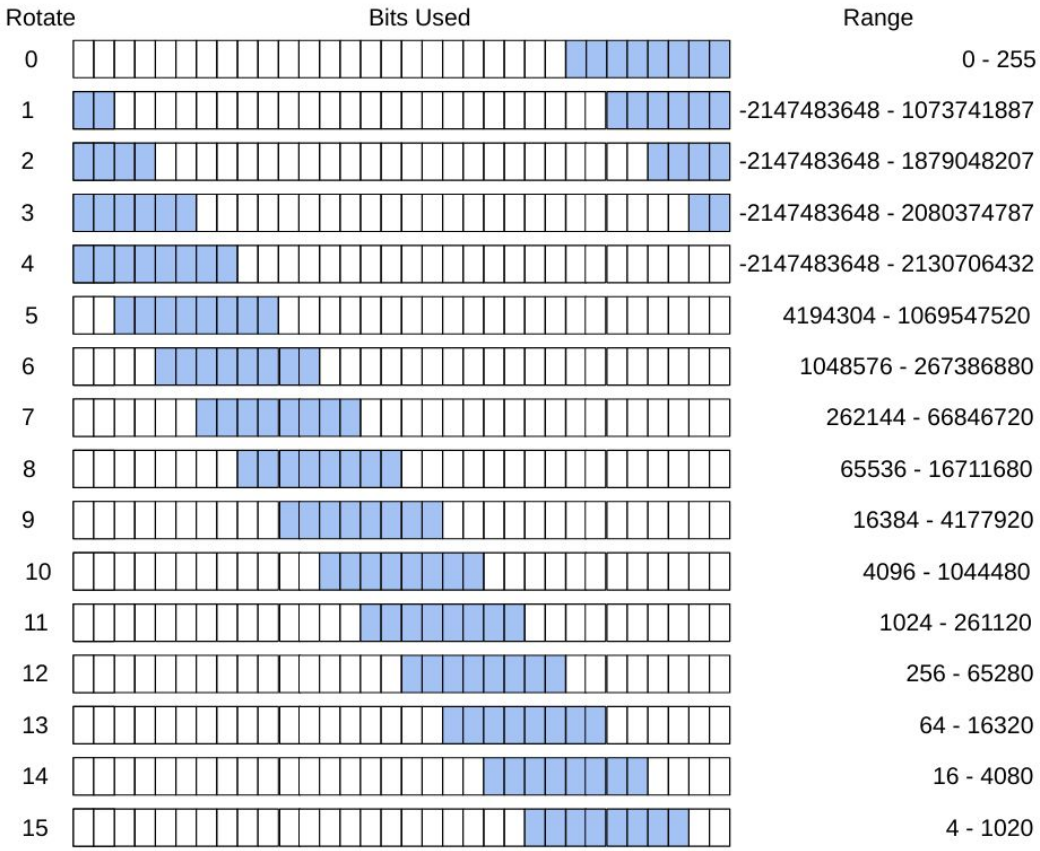
How might you encode 256?

1100	00000001

1101	00000100

1110	00010000

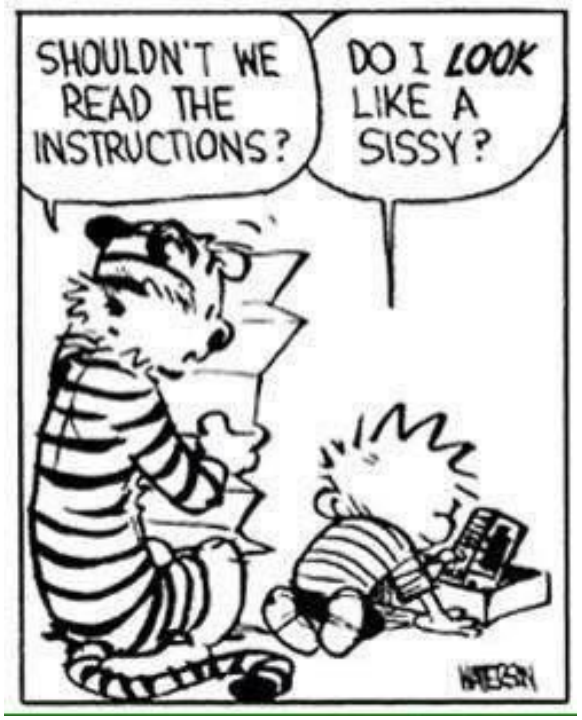
1111	01000000



Read the Instructions

.. when all else fails

- What do instructions do?
- How are instructions decoded?
- Uniformity and Symmetry
- Cramming stuff in
- CPU state
  - Condition codes
  - Program Status Register (PSR)



A closer look at the opcodes

The **Opcode** field is common to both of the basic instruction types

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

ARM data processing instructions can be broken into four basic groups:

- Arithmetic (6)
- Logic (4)
- Comparison (4)
- Register transfer (2)

0000 - **AND**
0001 - **EOR**
0010 - **SUB**
0011 - **RSB**
0100 - **ADD**
0101 - **ADC**
0110 - **SBC**
0111 - **RSC**
1000 - **TST**
1001 - **TEQ**
1010 - **CMP**
1011 - **CMN**
1100 - **ORR**
1101 - **MOV**
1110 - **BIC**
1111 - **MVN**

We haven't discussed the "S" field yet. If set, it tells the processor to retain some "state" after the instruction has executed. This "state" is in the form of 5-flags.

Many instructions (all we've seen thus far) have a special variant that sets the state flags. In these variants the opcode has an "S" appended.

Arithmetic Instructions

ADD R3, R2, R12

R3 ← R2 + R12
Registers can contain either 32-bit unsigned values or 32-bit 2's-complement signed values.

SUB R0, R4, R6

R0 ← R4 - R6
Once more, either 32-bit unsigned values or 32-bit 2's-complement signed values.

RSB R0, R4, R2

R0 ← - R4 + R2
The operands of the subtraction are in reversed order. It is called "Reverse Subtract". Why? The I-type version makes more sense.

ADC R1, R5, R8

R1 ← R5 + R8 + C
Where "C" is the Carry-out from some earlier instruction (usually an ADDS or ADCS) as saved in the Program Status Register (PSR)

SBC R2, R5, R7

R2 ← R5 - R7 - 1 + C
Where "C" is the Carry-out from some earlier instruction (usually a SUBS or SUBCS) as saved in the PSR

RSC R1, R5, R3

R1 ← - R5 + R3 - 1 + C
"Reverse Subtract" with a Carry. Usually a carry generated from a previous RSBS or RSCS instruction.

A byte-sized example:

					1=1-1+C			1
411 =	00000001	10011011			00000001		10011011
-42 =	00000000	00101010		+	11111111	+	11010101
				1	00000001	C=1	01110001

= 256 + 113 = 369

Logic Instructions

Logical operations on words operate "bitwise", that is they are applied to corresponding bits of both source operands.

```
AND R0,R1,R2
ORR R0,R1,R2
EOR R0,R1,R2
BIC R0,R1,R2
```

Commonly called "exclusive-or"

Called "Bit-clear" R0 ← R1 & ~(R2)

R1:	0000 0000 0000 0000 1111 1111 0000 0000
R2:	0000 0000 0000 0000 1111 0000 1111 0000
R0:	0000 0000 0000 0000 1111 0000 0000 0000
R0:	0000 0000 0000 0000 1111 1111 1111 0000
R0:	0000 0000 0000 0000 0000 1111 1111 0000
R0:	0000 0000 0000 0000 0000 1111 0000 0000

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 opeartion 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	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

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

```
CMP R0,R1
```

PSR flags set for the result R2 - R3

```
CMN R2,R3
```

PSR flags set for the result R2 + R3

```
TST R4,#8
```

PSR flags set for the result R4 & 8

```
TEQ R5,#102
```

PSR flags set for the result R5 ^ 1024

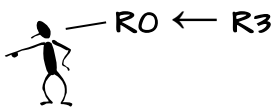
Register Transfer

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".

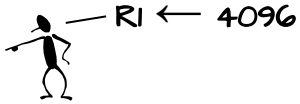
	4	3	4	1	4	4	8	4
R type:	1110	000	Opcode	S	0000	Rd	00000000	Rm
I type:	1110	001	Opcode	S	0000	Rd	Rotate	Imm8

1101 - MOV
1111 - MVN

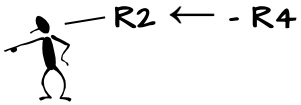
```
MOV R0,R3
```



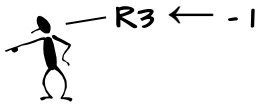

```
MOV R1,#4096
```




```
MVN R2,R4
```

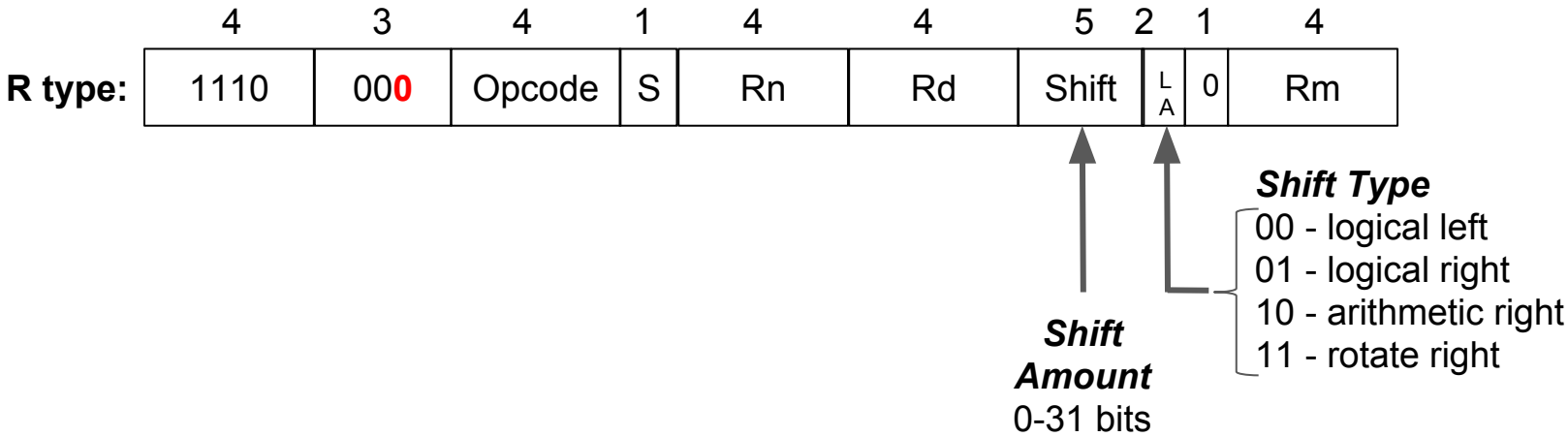



```
MVN R3,#1
```



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:	0000 0000 0000 0000 0000 0000 0000 0111	= 7
R1 after:	0000 0000 0000 0000 0000 0011 1000 0000	= 7 * 2^7 = 896

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
R1 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:	1111 1111 1111 1111 1111 1100 0000 0000	= -1024
R1 after:	1111 1111 1111 1111 1111 1111 0000 0000	= -1024 / 2^2 = -256

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
```

R0 before:	0000 0000 0000 0000 0000 0000 0000 0111	= 7
R1 after:	1100 0000 0000 0000 0000 0000 0000 0001	= -1,073,741,823

Why no rotate left shift?

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

Next time

Instructions still missing

- Access to memory
- Branches and Calls
- Control
- Multiplication?
- Division?
- Floating point?