Note Title 27-08-2011

Aug. 27

Advanced ARM Assembly

C code

```
if (x==1)
        y = a+b;
```

$x - r_1$ $y - r_2$
$a - r_3$ $b - r_4$

ARM assembly

```
CMP  r1, #1
BNE  .L1
ADD  r2, r3, r4

.L1  ---
```

(Let us try to compress this)

Conditional Instructions

Optimal Sequence $\left\{ \begin{array}{l} \text{CMP} \quad r_1, \#1 \\ \text{ADDEQ} \quad r_2, r_3, r_4 \end{array} \right.$

ADDEQ → It adds only if the result of the last comparison is equal

Otherwise, it skips.

General Structure of a conditional instruction (Data Processing Instructions)

$<$operation$>$ $<$Condition$>$

ADD	EQ	(Equality)	HI (Unsigned Higher)
SUB	NE	($\neq$)	
mov	LE	($<=$)	HS (Signed)
⋮	LT	($<$)	
	GE	($>=$)	‖ (ALWAYS)
	GT	($>$)	RSVD

MOVLE (Move if the last comparison has the following condition to be true Less then Equal)

LSL GT

What does a compare instruction do?

Puts the result of the compare in a special register: CPSR

Current Program Status Register

$$\overbrace{N \qquad Z \qquad C \qquad O}^{\text{condition Flags}}$$

Negative Zero Carry overflow

LE :	N || Z	EQ	Z	LT	( )
GT :	(!N) && (!Z)	NE	(!Z)	GE	( )

SUB → Normally does not set the condition Flags.

SUB$\textcircled{S}$ → Sets the condition Flags.

ADDS $\rightarrow$ Set the condition Flags.

C code.

```
r1 = r2 - r3
if (r1 < 0)
        r4 = r5 + r6
```

ARM ASSEMBLY CODE

```
SUB     r1, r2, r3
ADDLT   r4, r5, r6
```

Addressing Modes

Data Trasfer Insts (LDR/STR)

1) LDR r_1, [r_2] (Register Indirect)

Address (A) $= r_2$

2) LDR r_1, [r_2, #4] (Base Offset)

$A = r_2 + 4$

3) LDR r_1, [r_2, r_3] (Register Offset)

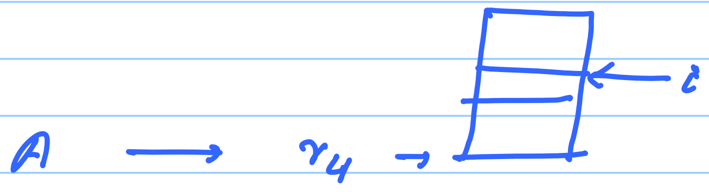
$A = r_2 + r_3$

4) LDR r_1, [r_2, r_3, LSL #2] (Shifted register Offset)

$A = r_2 + r_3 << 2$

Pre-Indexed

```
1) sum=0;
2) for (i=0; i<n; i++)
         sum += A[i];
3)
```



Address = r_4 + $r_2 << 2$
(A)
A + i × 4

Can I combine the ticked instructions.

sum → r_1 A → r_4
i → r_2
n → r_3

```
     MOV  r1, #0          (sum=0)
     MOV  r2, #0          (i = 0)

.L1  CMP  r2, r3          ( i < n)
     BGE  exit

✓    LDR  r5 , [r4, r2, LSL
                  # 2]
                    (Load A[i])

     ADD  r1, r5, r1
                  (sum += A[i])

✓    ADD  r2, r2, # 1

     B .  .L1

exit :               (Example 1)
```

Pre - indexed access.   Address = $r_4 + r_2 << 2$

(Normal) LDR r_5, [r_4, r_2, LSL #2]

(Pre- Indexed)
LDR r_5, [r_4, r_2, LSL #2] !

r_4 += r_2 << 2

Then :
Address = r_4

Normal Load
Value of base register
r_4 does not change

Pre Indexed Access
~~Post~~
Value of base register r_4
does change.

Post Indexed access.

[++i; Pre
i++; post]

LDR r_5, $[r_4]$, r_2, LSL#2

1) Address $= r_4$

2) Performs the load

3) Then :

$r_4 \mathrel{+}= r_2 << 2$

Two Operations here:

(a) Perform the load

&

(b) Change the base Address

Pre

(b) $\longrightarrow$ (a)

Post

(a) $\longrightarrow$ (b)

To summarize: We can replace both the ticked instructions with one post-indexed access in Example 1.

We can also add conditionals to branch instructions

BEQ (Branch-if-equal)

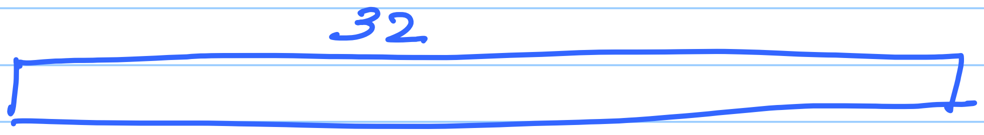
BLNE (Branch & link if not equal)

Instruction Format

Encoding to create a sequence of 0s and 1s from an assembly instruction, which the computer can understand.

ARm : Regular Format

Each Assembly Instruction $\longrightarrow$ 32 bits.
(4) bytes.



Data Processing Instructions.

OpCode. → What kind of operation this is
(4 bits) (ADD, SUB, LSL, MOV...)

Format → Data Processing OR DATA TRANSFER OR CONTROL
(2 bits)

16 kinds of conditions (EQ, LE, NE, LT, GE, GT, HI, HS, ALWAYS, RSVD....)
(4 bits)

S → (ADD → ADDS) (Set the condition Flags)
(1)

ADD $r_3, r_2, \boxed{r_1}$ ← reg.

OR

ADD $r_3, r_2, \boxed{\#4}$ ← immediate

$\underset{(1)}{I}$ → (I = = 1, The last operand is an immediate)

(I == 0, The last operand is a register)

🚩 Uptil now: 12 bits are gone : 20 bits are left

Out of the 20 bits 16 registers in ARM (4 bits)

destination register: 4 bits.

src1 reg: 4 bits.

🚩 Uptil now: 8 out of 20 bits are gone
12 bits are left

$$\begin{cases} \text{(Reg)} & \text{ADD } r_3, r_2, r_1 \\ \text{(Imm)} & \text{ADD } r_3, r_2, \#4 \end{cases}$$

If src2 is a register.
(Use 4 out of the 12 bits)

If src 2 is an immediate
(Use 12 bits to represent the immediate)

🚩 🚩 🚩

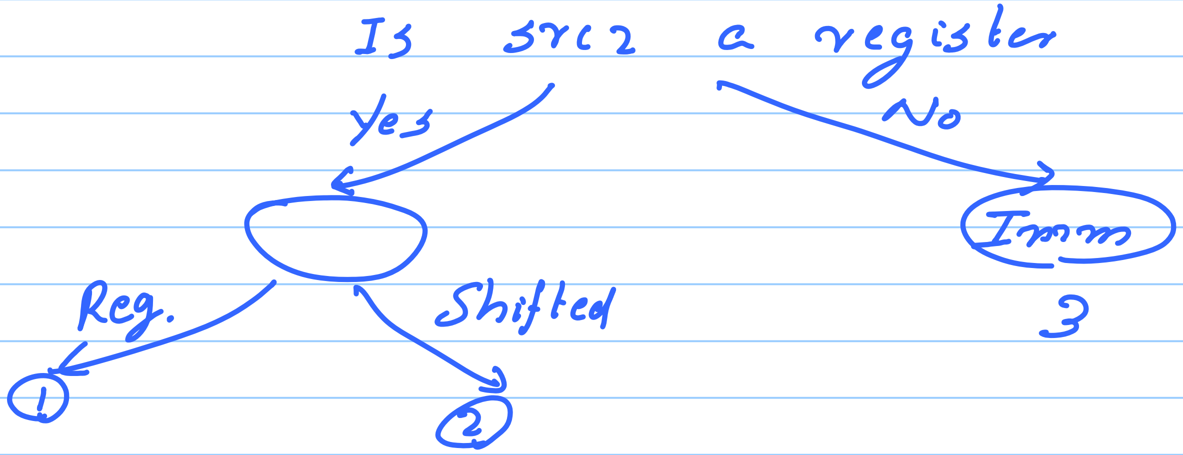
We do not have any more bits left.

Another Format:

ADD r_1, r_2, r_3, LSL #2 (Shifted)

$r_1 = r_2 + r_3 << 2$

Last 12 bits



1) 4 bits to represent a register.
We have 12 bits.

Shifted

$$2) \quad \text{ADD} \quad r_3, \quad r_1, r_2, \quad \overbrace{\left[\begin{array}{c} \text{LSL} \\ \text{LSR} \\ \text{ASR} \\ \text{ROR} \end{array}\right] \boxed{\#2}}^{12 \text{ bits.}}$$

(0 - 31)

src 2 register — 4 bits

Is it in shifted format - 1 bit

Type of the shift (LSL or LSR or ASR or ROR)

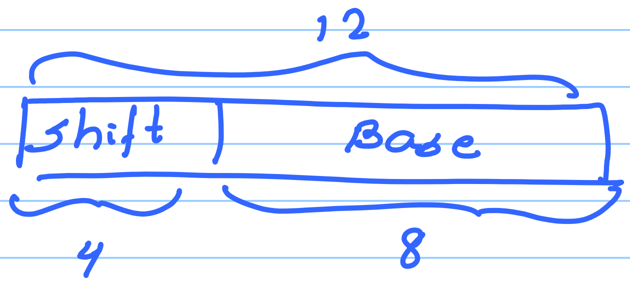
- 2 bits

Value of the shift -5 bits (0 — 31)

3) Immediate

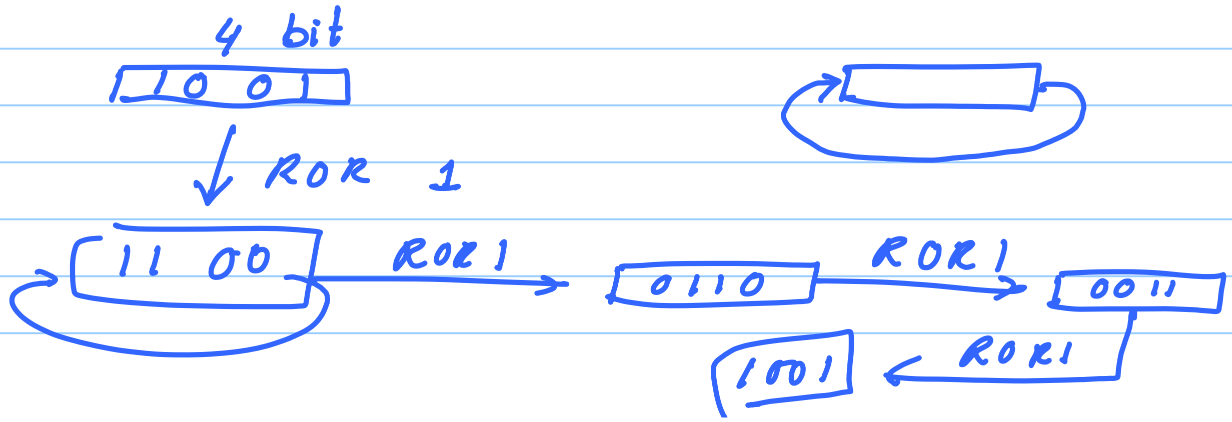
12 bits

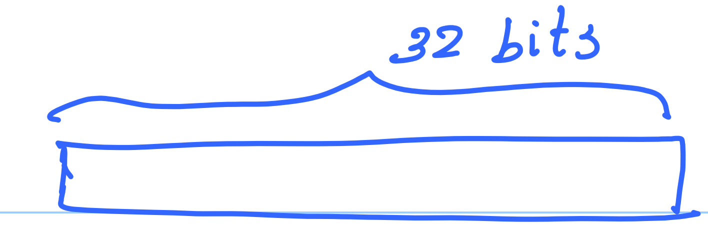
1½ bytes

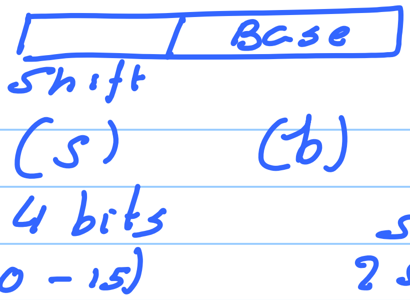


1 - char
2 - short
4 - int / float
8 - long / double.

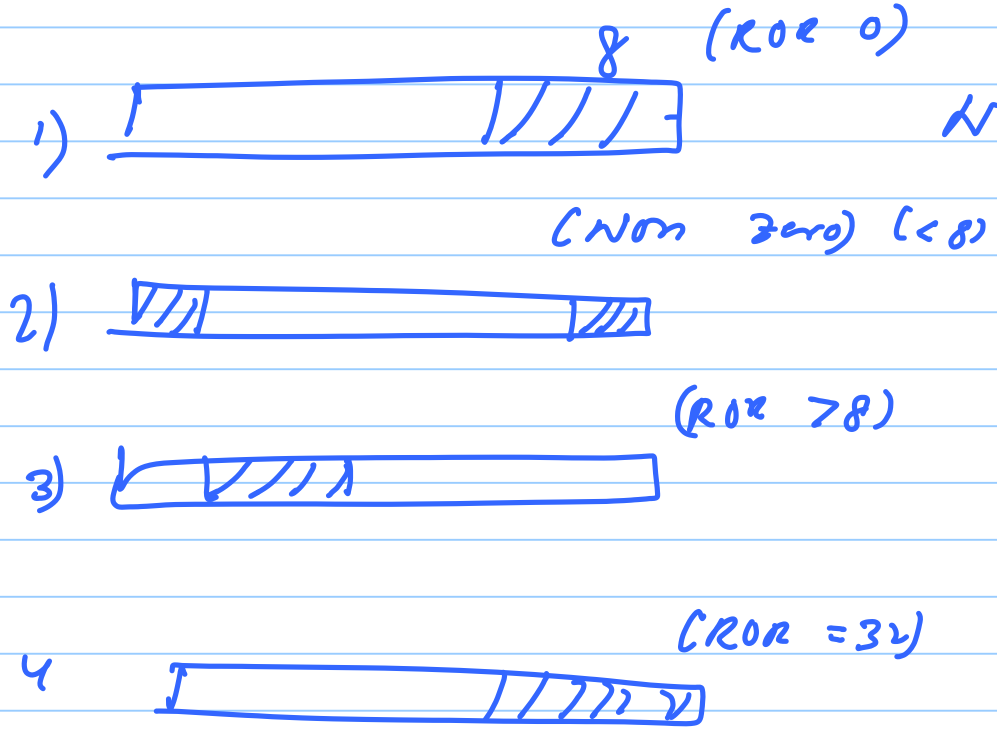
Right Rotate (ROR):







s (0 - 15)
2s (0 - 30)



$N = b$ ROR $(2s)$

why do we multiply s by 2?

(1) This is the common case

(2) This extends our range from (0 - 15) to (0 - 30)

Hardware:

$$12 \text{ bits} \xrightarrow{\hspace{2em}} 32 \text{ bits}$$

(decoding)

Assembler/ Compiler

$$32 \text{ bits} \xrightarrow{\hspace{2em}} 12 \text{ bits}$$

(encoding)

Suppose you give an immediate value that cannot be encoded, to the assembler.

$\longrightarrow$ Throw an error.

Eg.

mov r1, #0x F0 00 00 0E

(Valid → Legal Encoding)

$\underbrace{(\boxed{2 \mid EF}}_{12}$ 2S → 4

0x $\underline{00}$ $\underline{00}$ $\underline{00}$ EF

ROR 4

mov r1, 0x F0 0A 00 0E

(Invalid)