1 memory location = 8 bits

Memory size = $2^n$ bytes

→ where n is no. of lines on address bus

| Type | Bytes | Bits | Range | |
|---|---|---|---|---|
| signed char | 1 | 8 | -128 -> +127 | |
| unsigned char | 1 | 8 | 0 -> +255 | |
| short int | 2 | 16 | -32,768 -> +32,767 | (+/-32KB) |
| unsigned short int | 2 | 16 | 0 -> +65,535 | ( 64KB) |
| unsigned int | 4 | 32 | 0 -> +4,294,967,295 | ( 4GB) |
| int | 4 | 32 | -2,147,483,648 -> +2,147,483,647 | (+/-2GB) |
| long int | 4 | 32 | -2,147,483,648 -> +2,147,483,647 | (+/-2GB) |
| long long int | 8 | 64 | -(2^32) -> (2^32)-1 | |
| float | 4 | 32 | | |
| double | 8 | 64 | | |
| long double | 12 | 96 | | |

Big endian : MSB on top

Little endian : LSB on top

Boolean data types → stored using a whole byte (inefficient, booleans tends to only need 1 bit)

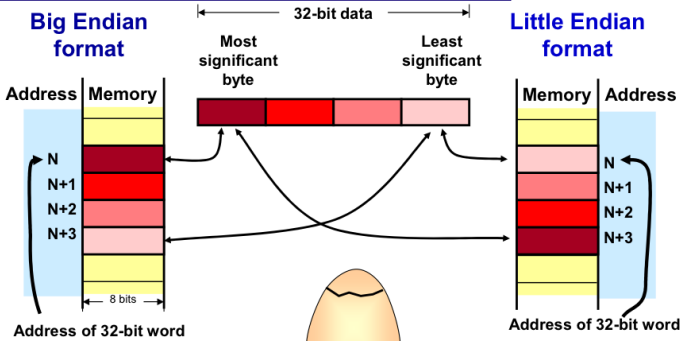
## Number Representation

| Type | Bytes | Bits | Range | |
|---|---|---|---|---|
| signed char | 1 | 8 | -128 -> +127 | |
| unsigned char | 1 | 8 | 0 -> +255 | |
| short int | 2 | 16 | -32,768 -> +32,767 | (+/-32KB) |
| unsigned short int | 2 | 16 | 0 -> +65,535 | ( 64KB) |
| unsigned int | 4 | 32 | 0 -> +4,294,967,295 | ( 4GB) |
| int | 4 | 32 | -2,147,483,648 -> +2,147,483,647 | (+/-2GB) |
| long int | 4 | 32 | -2,147,483,648 -> +2,147,483,647 | (+/-2GB) |
| long long int | 8 | 64 | -(2^63) -> (2^63)-1 | |
| float | 4 | 32 | | |
| double | 8 | 64 | | |
| long double | 12 | 96 | | |

applies to both signed & unsigned

unsigned magnitude range : 0 - 255
2's complement range : -128 to 127

## Data Organisation in Memory




## Array Representation

```
main(){
  char c[2]        // Instiating char array C with 2 index
  c[0] = 'A'       // Assigning 'A' to 1st index
  c[1] = 'B'       // Assigning 'B' to 2nd index
  Short int i[3]   // Instantiating short int array i with 3 index
   i[0] = 5;       // Assigning 5 to 1st index
   i[1] = 6;       // Assigning 6 to 2nd index
   i[2] = 7;       // Assigning 7 to 3rd index
```

| Address | Memory | |
|---|---|---|
| 0x100 | c[0] | → char takes 1 byte |
| 0x101 | c[1] | |
| 0x102 | 0x00 | i[0] |
| 0x103 | 0x05 | |
| 0x104 | 0x00 | i[1] |
| 0x105 | 0x06 | |
| 0x106 | 0x00 | i[2] |
| 0x107 | 0x07 | |

8 bits

Short int takes 2 bytes

NESTED ARRAY:

main() { int k[3][2]; }

- The offset from BA for element k[a][b]

= size of (datatype) * ((2 * a) + b)

32 bits

eg. 0x0114 k[2][1] $BA_k$ + 20

→ 4 * ((2*2)+1) = 20

## String Representation (C string)

```
main()
{
  char S[4] = "123";  // a string constant
}
```

| Address | Content in memory |
|---|---|
| 0x0100 | 0x31 |
| 0x0101 | 0x32 |
| 0x0102 | 0x33 |
| 0x0103 | 0x00 |

String must always be terminated by a null character.

For Pascal, it stores the length of the string at start of the string.

## Pointers Representation

```
main()
{
  char c;        //char variable c
  char *ptr;     // char pointer ptr
  c = 'A'        // assign value 'A' to c
  ptr = &c;      // ptr gets address of c
```

Char c has address 0x0100

It is assigned char 'A'

When ptr = &c, it points the address 0x0100

| Variable | Size (bytes) |
|---|---|
| c | 1 |
| ptr | 2 |

16 bits

## Dereferencing a Pointer

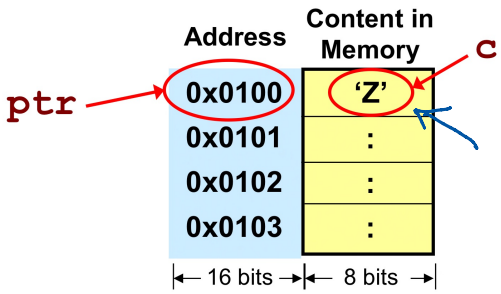
```
Main()
{
  char c;
  char *pointer;
  c = 'A'
  ptr = &c;
  *ptr = '2'
```

ptr gets address of c

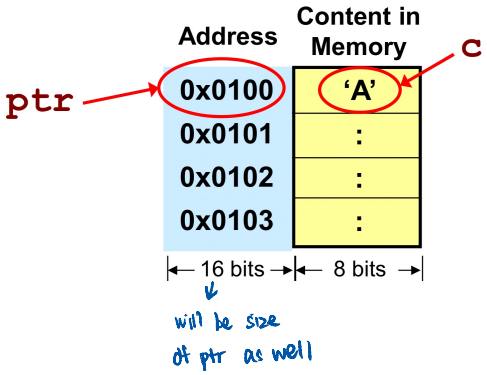
dereferencing; operator on ptr to give variable c value '2'

with the dereferencing variable *, it allows it to copy a value to the address pointed by the pointer *ptr.

- value of pointer is an address
- size is fixed
- size of pointer depends on the processor's address range




A gets updated to Z




will be size of ptr as well

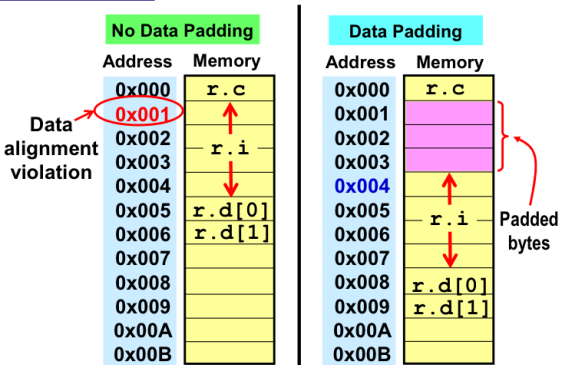
## Data Alignment

| Data Type | Size (Byte) | Example of allowable start addresses due to alignment |
|---|---|---|
| char | 1 | 0x..0000, 0x..0001, 0x..0002 |
| short | 2 | 0x..0000, 0x..0002, 0x..0004 |
| int | 4 | 0x..0000, 0x..0004, 0x..0008 |
| float | 4 | 0x..0000, 0x..0004, 0x..0008 |
| double | 8 | 0x..0000, 0x..0008, 0x..0010 |
| pointer | 8 | 0x..0000, 0x..0008, 0x..0010 |

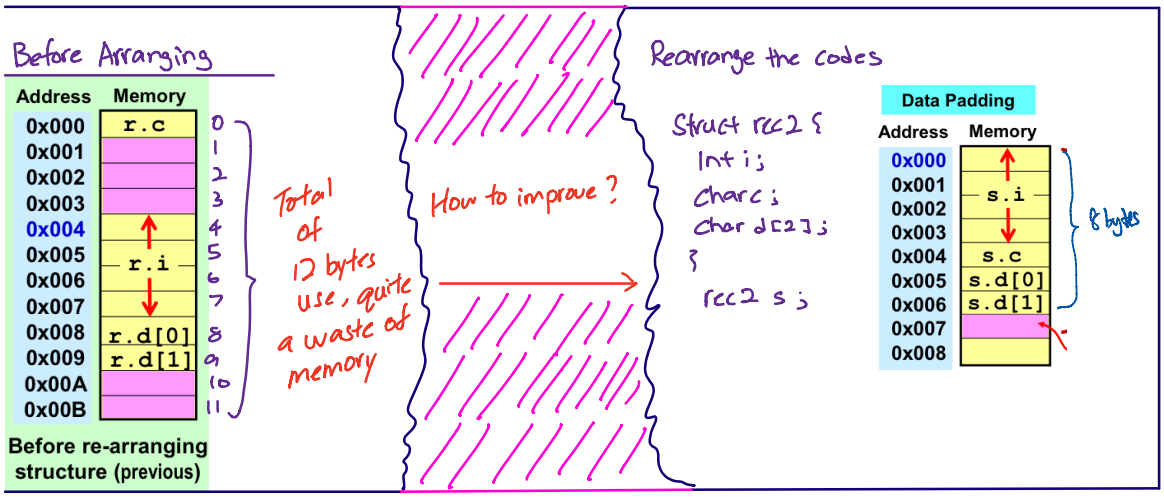
This describes the addresses that they can be at due to data alignment. It must be followed strictly. (all values are in hex)

## Data Alignment with Structures

```
Struct rec1 {
  Char c;
  int i;
  char d[2];
}
 rec1  r
   :
 rec1 + [10]
```




Data padding is required in assigning of Struct rec1 in memory so that it does not violate the data alignment table as shown above.



| CPSR bit(s) | Name | Description |
|---|---|---|
| 31 | N | Can set if last operation produced a **negative** result |
| 30 | Z | Can set if last operation produced a **zero** result |
| 29 | C | Can set if last operation produced a **carry out** in the most significant bit |
| 28 | V | Can set if the last operation produced an **overflow** for a **signed** arithmetic operation |

N – Negative; Z – Zero; C – Carry ; V – Overflow

# Basic Execution Cycle

## The MOV Instruction

MOV R1, R0 → Copy operation

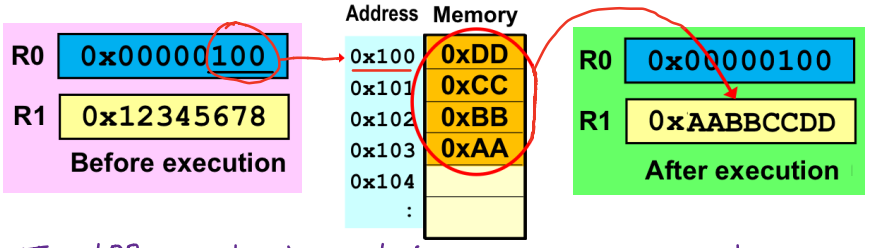
Destination ↑ Source

The MOV operator copies register content to a register. (R0 value remains the same)

## The LDR Instruction

LDR R1, [R0]

Destination ↑ Source

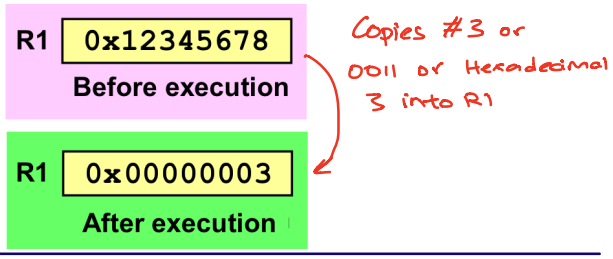



The LDR operator is used to copy memory content to a register.

## Immediate Addressing

MOV R1, #3 ← Immediate value

Destination ↑ Source



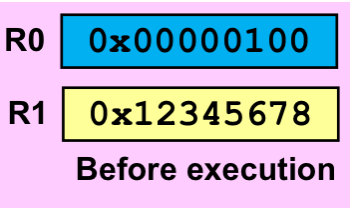

Copies #3 or 0011 or Hexadecimal 3 into R1

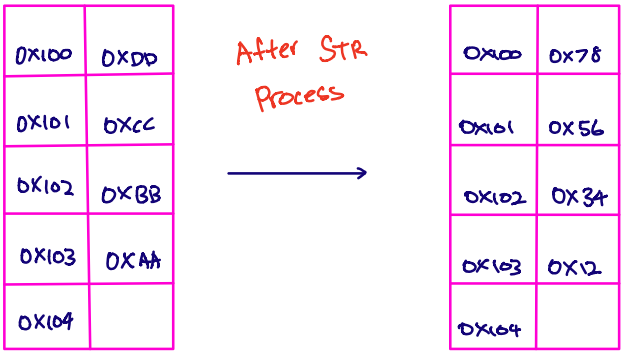
## The STR Instruction

STR R1 [R0]

Source: Register

Destination [Memory] Address in Register



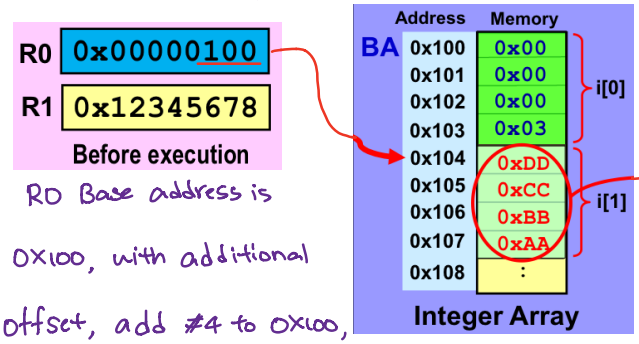




The STR operator is used to copy register content to memory.

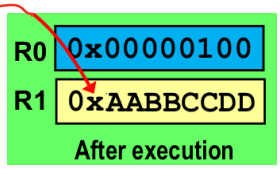
## Register Indirect with Offset

LDR R1, [R0, #4]

Destination (Register) — Source (memory) — Additional offset




R0 Base address is 0x100, with additional offset, add #4 to 0x100, and base Address becomes 0x104.

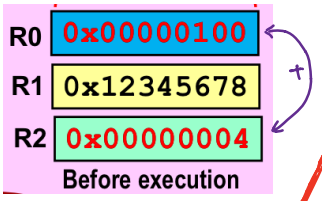



Register 1 will get 0x104 - 0x107 data.

## Register Indirect with Index Register

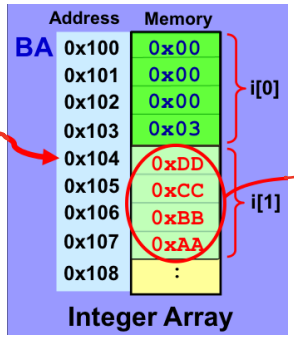
LDR R1, [R0, R2] = R0+R2

Destination (Register) — Source (memory)

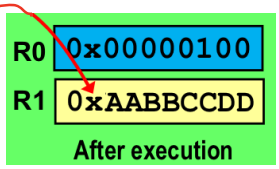



R0+R2 = 0x104

Base Address




Instead of using an offset, we use an index register, and it works the same way.




Register 1 will get 0x104 - 0x107 data.

## Acessing Array elements

```
MOV R2, #0x100      // Instantiate the base address into R2
MOV R1, #7          // Instantiate the value 7 into R1
STR R1, [R2, #0]    // Store the value of 7 into R2 base Address # 0x100
STR R1, [R2, #16]   // Store the value of 7 into R2 base Address # 0x 116
```

## Clearing Array elements

```
MOV R2, #0x100      // Instantiate the base address into R2
MOV R0, #0          // Instantiate the value 0 into R0
MOV R1, #0          // Instantiate the value 0 into R1
STR R0, [R2, R1]    // Copies the value of 0 from R0, with register index
ADD R1, R1, #4      //   #0x100 (R2) and #0 (R1), #0x100 + #0 = #0x100,
                    //   So the value will be copied from R0 to #0x100 - #0x103
                    // The next code increments the value/index of R1 by 4,
                    //   and it loops till it is finished
```

## Offset with Auto-indexing

```
LDR R1, [R0, #4]!
```

Destination (Register) ↑ Source (memory) ↑

## Index Register with Auto-Index

```
LDR R1, [R0, R2]!
```

Destination (Register) ↑ Source (memory) ↑

Both of this codes work the same way as the two above, the only difference is the addition of `!`. The purpose of this `!` would be to set the register to its new address value based on its last accessed address for the operation.

For e.g., for the code on the left, the address accessed was 0x104, whereas the og address of R0 is 0x100, after the operation, `!` updates R0 to #0x104.

## Clearing all Array elements with auto-index

```
MOV R2, #0x100      // Instantiate the base address to R2
Mov R1, #0          // Instantiate R1 with the value 0
STR R1, [R2]        // Stores the value of 0 from R1 to index R2
STR R1, [R2, #4]!   // Stores the value of 0 using the current effective address of
                    //   auto-index register R2 plus offset 4. Then put this EA
                    //   into R2.
```

## Pre-index

| | |
|---|---|
| `LDR R1, [R0, #4]!` ; R0 = R0+4<br>; R1 = mem[R0]<br>(Offset with auto-indexing (pre-index)) | `LDR R1, [R0, R2]!` ; R0 = R0+R2<br>; R1 = mem[R0]<br>(Index with auto-indexing (pre-index)) |

In Pre-index, the indirect register is auto-index before being used to compute the effective address.

## Post Index

| | |
|---|---|
| `LDR R1, [R0], #4` ; R1 = mem[R0]<br>R0 = R0+4<br>(Offset with auto-indexing (post-index)) | `LDR R1, [R0], R2` ; R1 = mem[R0]<br>; R0 = R0 ~~+4~~ + R2<br>(Index with Auto-indexing (post-index)) |

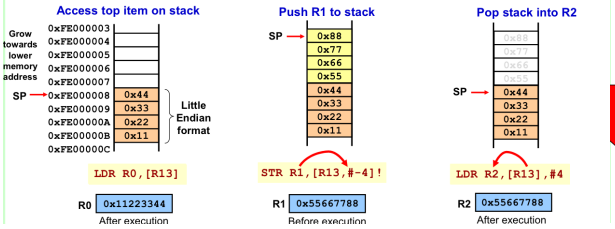
In post-index, the indirect-register is used to compute the effective address after it is auto-indexed.

## ARM Stack Implementation (FD)

The are 4 possible stack implementations supported by the ARM instruction set.

**F**ull **D**escending, **F**ull **A**scending, **E**mpty **D**escending and **E**mpty **A**scending

Example of **Full Descending** (FD) stack implementation:

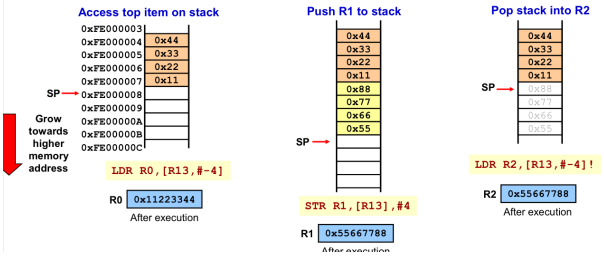



## Empty Ascending Implementation (EA)

EA is an alternative stack implementation.

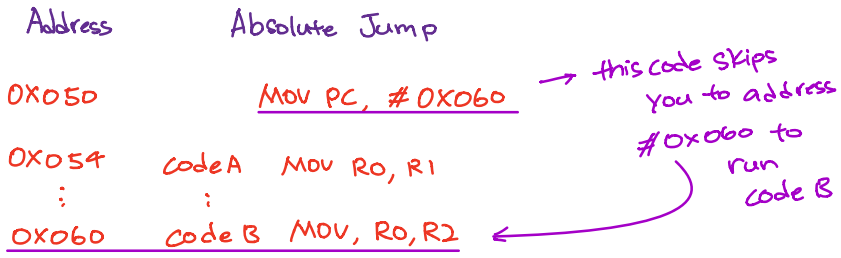
**Empty** means **SP** points to an available **unoccupied** stack space.

**Ascending** means stack grows toward **higher** memory address.



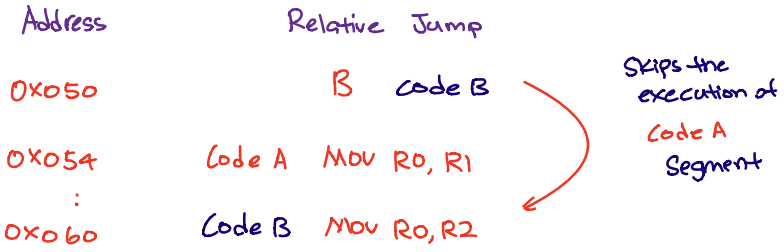
## Absolute Jump

A new address can be loaded into the PC to alter the sequential order of program execution. An absolute jump to a new code position is done by loading the address to jump to into the PC.




## Relative Jump

An offset can be added to the PC alter the sequential order of program execution. A relative jump is done using the branch instruction with an appropriate signed offset.



## Register Data Transfer

MOV commands

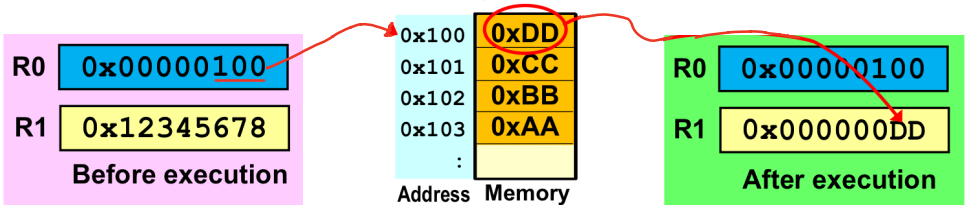
```
MOV  R1, R0    ; Copy R0 to R1
MOVS R0, #0    ; move 0 into R0 and set Z flag
MVN  R1, R0    ; R1 = NOT (R0)
MVN  R0, #0    ; move 32-bit value of -1 into R0
```

The command mvn acts as a not operator, it complements bits.

## Memory Data Transfer

### LDR commands

```
LDR R1, [R0]   ; Copy 32-bit value pointed by R0 into R1 → Copying of 4 bytes
LDRB R1, [R0]  ; Copy 8-bit value pointed by R0 into R2 → Copying of 1 bytes
```




### STR Commands

```
STR  R1, [R0] ;  Copy R1 (4 bytes) starting at address pointed by R0
STRB R2, [R0, #1]! ; Copy byte in R2 to only one address at [R0+1];
                     then R0 = R0+1 ;
```

Program Example : Copying a block of memory

Block copy 5 words starting at address 0x100 to 0x200

```
MOV R0, #0x100 ;    // R0 = 0x100 (Base Addr)
MOV R1, #0x200 ;    // R1 = 0x200 (Destination Addr)
LDR R2, [R0] ;      // Copying of 4 bytes of data in R0 into R2
STR R2, [R1] ;      // Copying of 4 bytes of data in R2 into R1
ADD R0, R0, #4      // This increments R0 by 4 → R0 = R0 + #4
ADD R1, R1, #4      // This increments R1 by 4 → R1 = R1 + #4
Loop 5 times
```

---

## Optimized Version

```
MOV  R0, #0x100 ;   // R0 = 0x100 (Base Addr)
MOV  R1, #0x200 ;   // R1 = 0x200 (Destination Addr)
LDR  R2, [R0], #4 ; // R2 get 4 bytes of data from R0, then R0 increments its
STR  R2, [R1], #4 ;    index by 4.
Loops 5 times       // R1 get 4 bytes of data from R2, then R1 increment its
                       index by 4.
```

---

## Further Optimization

```
MOV R0, #0x100      // R0 = #0x100 (Base Addr)
LDR R2, [R0], #4    // R2 get 4 bytes of data from R0, then R0 increments its
STR R2, [R0, #0xFC]    index by 4.
Loop 5 times        // R0 + #0xFC gets 4 bytes of data from R2
```

# Arithmetic Instructions

## ADD Instruction

ADDS R2, R0, #8

Destination ↑ + (Addition)

R2 = R0 + #8

## SUB Instruction

SUBS R2, R0, #4

Destination ↑ − (Subtract)

R2 = R0 − #4

## RSB Instruction

RSBS R2, R0, #4

Destination ↑ − R subtract

R2 = #4 − R0

| CPSR bit(s) | Name | Description NZVC CPSR |
|---|---|---|
| 31 | N | Can set if last operation produced a **negative** result |
| 30 | Z | Can set if last operation produced a **zero** result |
| 29 | C | Can set if last operation produced a **carry out** in the most significant bit |
| 28 | V | Can set if the last operation produced an **overflow** for a **signed** arithmetic operation |

All the "S" suffixes has been added to the all these instructions to influence the flags.

**N** – Negative; **Z** – Zero; **C** – Carry ; **V** – Overflow

How ADD and SUB affects the flags

CZ1106 CE1106

## Condition Code Flags and ADD

- **ADD** can affect all the **N**, **Z**, **V**, **C** flags.

| | | Signed Number | Unsigned Number |
|---|---|---|---|
| (+ve) | 0000 0001 | (1) | (1) |
| (+ve) | +0111 1111 | (127) | (127) |
| (-ve) | 1000 0000 | (-128) | (128) |

N=1, V=1 ← 2's complement oVerflow

| | | Signed Number | Unsigned Number |
|---|---|---|---|
| (+ve) | 0000 0001 | (1) | (1) |
| (-ve) | +1111 1111 | (-1) | (255) |
| (+ve) | 0000 0000 | (0) | (0) |

Z=1, C=1 ← unsigned overflow

- The **V** flag when set, indicates an **overflow** when adding **signed** numbers.
- Overflow is detected when both **signed numbers** added have the **same sign** but the result has the **opposite sign**.
- The **C** flag when set, indicates an **overflow** when adding **unsigned** numbers.

Left Diagram:

Change of sign in addition

= Negative ∴ N=1, V=1

Right Diagram:

Adding 1 + (-1) = 0

∴ Z=1, C=1 ← unsigned overflow

CZ1106 CE1106

## Condition Code Flags and SUB

- **SUB** can affect all the **N**, **Z**, **V**, **C** flags.
- In the ARM's **SUB** instruction, the **C** flag clears (**C**=0) if the subtraction produce a **borrow** and **C** sets (**C**=1)otherwise.
- A borrow occurs in subtraction when the **unsigned value** of the Minuend is **less** than the unsigned value of the Subtrahend.

Borrow

A ← Minuend

− B ← Subtrahend

(A-B)

Minuend Subtrahend
**unsigned (A) < unsigned (B)** **C=0**

Minuend Subtrahend
**unsigned (A) ≥ unsigned (B)** **C=1**

- The **V** flag is set (**V**=1) when the result is out of the signed 32-bit range. $-2^{31} < (A-B) \geq +2^{31}$ **V=1**
- An **unsigned underflow** is indicated by (**C**=0).

**Learn More:** Google "ARM subtraction carry flag"

## Convert Negative to Positive

| | |
|---|---|
| MOV R0, #0X100 | R0 = 0X100 (Base addr) |
| LDR R1, [R0] | Copies R0 data into R1 |
| MVN R1, R1 | MVN complements R1 |
| ADD R1, R1, #1 | R1 = R1 + #1 (To do 2's complement) |
| STR R1, [R0], #4 | Copies R1 data into R0 and then increments index by 4. |

## Optimized version

| | |
|---|---|
| MOV R0, #0X100 | R0 = 0X100 (Base addr) |
| MOV R2, #0 | Copies the IA value 0 into R2 |
| LDR R1, [R0] | Copies R0 data into R1 |
| SUB R1, R2, R1 | R1 = 0 - (-R1) = 0+R1 (This turns -ve to +ve) |
| STR R1, [R0], #4 | Copies R1 data into R0, then increments index by 4 |

## Further Optimisation

| | |
|---|---|
| MOV R0, #0X100 | R0 = 0X100 (Base addr) |
| LDR R1, [R0] | Copies R0 data into R1 |
| RSB R1, R1, #0 | R1 = #0 - (-R1) = 0+R1 (Reverse Subtraction) |
| STR R1, [R0], #4 | Copies R1 data into R0, then increments index by 4. |

## Carry-based Arithmetic Instructions

```
ADC R2, R0, R1 ;  R2 = R0 + R1 + C   (ADD with Carry)
SBC R2, R0, R1 ;  R2 = R0 - R1 + NOT(C)  (SUB with carry)
RSC R2, R0, R1 ;  R2 = R1 - R0 + NOT(C)  (RSB with carry)
```

## Logical Instructions

### AND, OR and EOR Applications

AND - clear specific bits in destination operand

ORR - set specific bits in destination operand

EOR - complement specific bits in dest operand.

**AND truth table**

| A | B | Z = A.B |
|---|---|---|
| 0 | 0 * | 0 |
| 0 | 1 | 0 |
| 1 | 0 * | 0 |
| 1 | 1 | 1 |

**OR truth table**

| A | B | Z = A+B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 * | 1 |
| 1 | 0 | 1 |
| 1 | 1 * | 1 |

**EX-OR truth table**

| A | B | Z = A⊕B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 * | 1 |
| 1 | 0 | 1 |
| 1 | 1 * | 0 |

Instruction examples:

Set R0 as 0101 0101

AND R1, R0, # 1111 0000 → Result

```
  0101 0101
& 1111 0000
-----------
  0101 0101
```

→ R1 = 0101 0101

ORR R0, R0, # 1111 0000 → Result

```
  0101 0101
| 1111 0000
-----------
  1111 0101
```

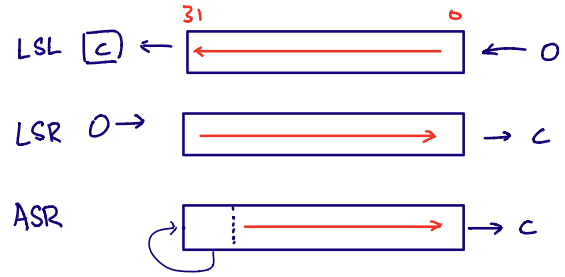
→ R0 = 1111 0101

EOR R2, R0, # 1111 0000 → Result

```
  0101 0101
^ 1111 0000
-----------
  1010 0101
```

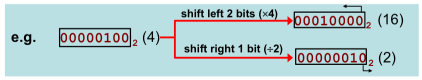
→ R2 = 1010 0101

### Shift and Rotate instructions




**Doing Arithmetic with Shift**

- Shift performs multiply (shift left) or divide (shift right) by a factor of $2^N$, where $N$ is the no. of bits shifted.

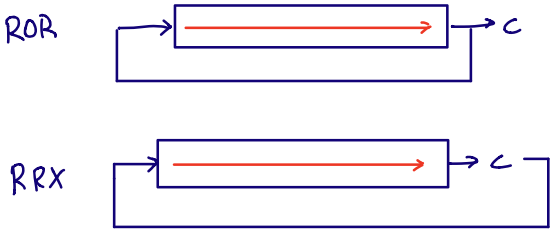



- In signed or unsigned **multiply**, binary "0" is shifted into the LSB of the register from the right using Logical Shift Left (LSL).
- In **unsigned divide**, binary "0" is shifted into the MSB of the register from the left using Logical Shift Right (LSR).
- In **signed divide**, the sign bit is shifted into the MSB from the left using Arithmetic Shift Right (ASR).


- The "S" suffix is used on the data processing operator to influence the C flag.

Taking 4x4, since $2^2 = 4$, N=2, bits shifted left=2

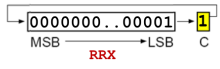
Taking 4/2, since $2^1 = 2$, N=1, bits shifted right=2




**Rotate Operations**

- Rotate is also called **cyclical shift**, as no bits in the register is lost during the shifting operation.
- In basic rotate right (ROR), the bit shifted out of register is returned in at the leftmost end and is also placed into the **C**-flag.


- In rotate right extended (RRX), the **C**-flag is shifted into the register at the leftmost end, while the bit shifted out replaces the current **C**-flag.



## Shift and Rotate Mnemonics

```
MOV R0, R0, LSL #1 ;        Shift R0 left by 1 bit
ADD R2,R1, R0, LSR #2 ;     Shift R0 2 bits to the right then ADD
ADDS R2, R1, R0, LSL R4 ;   Shift R0 by R4 bits to the left, then ADD. Influences Flags
ORRS R0, R0, R0, ROR #1 ;   OR R0 with a 1 bit-right rotated version of itself. Set
                            C flag if bit rotated out is 1.
```

## Count Number of 1's

```
MOV R2, #0          ; Clear 1- counter for binary 1s / Instantiate counter to be 0
MOV R3, #32         ; Set Loop counter to 32 times
MOV R0, R0, ROR #1  ; Rotate R0 by 1 bit, then R0 copies the edited R0
AND R1, R0, #1      ; Clear all bits except for the LSB
ADD R2, R2, R1      ; R2 = R2+R1
SUB R3, R3, #1      ; R3 = R3 - #1   (decrements the loop counter)
```

## Optimized Version

```
MOV R2, #0          ; clear one counter for binary 1s
MOVS R0, R0, LSR #1 ; Shift R0 1 bit to the right, OG R0 = changed R0, moves carry bit into C Flag
ADC R2, R2, #0      ; Add the C flag into the counter
```

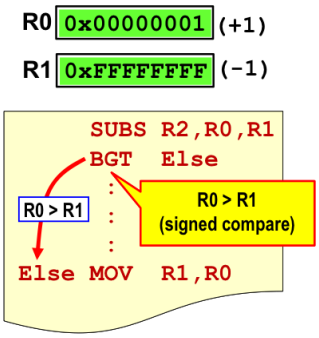
## Different Bcc Conditions

CZ1106
CE1106

### Different Bcc Conditions

- There are 15 possible conditional tests for **Bcc**.

| Suffix | Flags | Meaning | |
|---|---|---|---|
| EQ | Z = 1 | Equal | |
| NE | Z = 0 | Not equal | |
| CS or HS | C = 1 | Higher or same, **unsigned** | Unsigned comparison |
| CC or LO | C = 0 | Lower, **unsigned** | |
| MI | N = 1 | Negative | |
| PL | N = 0 | Positive or zero | |
| VS | V = 1 | Overflow | |
| VC | V = 0 | No overflow | |
| HI | C = 1 and Z = 0 | Higher, **unsigned** | Unsigned comparison |
| LS | C = 0 or Z = 1 | Lower or same, **unsigned** | |
| GE | N = V | Greater than or equal, **signed** | Signed comparison |
| LT | N != V | Less than, **signed** | |
| GT | Z = 0 and N = V | Greater than, **signed** | |
| LE | Z = 1 and N != V | Less than or equal, **signed** | |
| AL | Can have any value | Always. This is the default when no suffix is specified. | |






## Program Example : Count Number of 1s

(Making use of Bcc conditions and flags to loop a program)

| | |
|---|---|
| MOV R2, #0 | Clear 1-counter for binary 1s |
| MOV R3, #32 | Set the loop counter to 32 |
| MOV R0, R0, ROR #1 | Rotate R0 by 1 bit |
| AND R1, R0, #1 | Clear all bits except for LSB, place it in R1 |
| ADD R2, R2, R1 | R2 = R2 + R1 |
| SUBS R3, R3, #1 | R3 = R3 - 1 (Decrements index by 1), as well as setting |
| BNE Loop | flags with suffix "S" |

↳ If Z != 0, continue looping, otherwise exit

## Comparing Signed values

```
SUBS R1, R1, R2 ;   R1 = R1 - R2
BGE             ;   Jumps to the next line of code if True
```

R1 ≥ R2

For testing signed values, use GT, LT, GE, LE

## Comparing Unsigned values

```
SUBS R1, R1, R2 ;   R1 = R1 - R2
BHS             ;   Jumps to R1 ≥ R2 if R1 higher or equal to R2
```

R1 ≥ R2

For testing unsigned values, use HI, LO, HS, LS

### CMP Command

CMP does the same thing as subs, however, registers remains unchanged but cc-flags are set.

```
CMP R1, R1, R2  ;  test (R1 - 4), where R1 is a signed no
BGE             ;  branch to R1 ≥ 4 if result is positive (i.e R1 ≥ 4)
```

R1 ≥ R2

## Other Conditional Test Instructions

```
CMN R0, R1 ;   set (N, Z, C, V) based on R0+R1    Compare Negative
TST R0, R1 ;   Set (N, Z, C) based on R0 AND R1   Test Bits
TEQ R0, R1 ;   Set (N, Z, C) based on R0 EOR R1   Test Equivalence
```

Program: Find the largest Number

```
        MOV  R0, #0x100     R0 = 0x100 (Base addr)
        MOV  R1, #9         Load 9 into R1 as counter Register
        LDR  R3, [R0]       Copy data from R0 index into R3
(LOOP)  ADD  R0, R0, #4     Increments R0 address by 4
        LDR  R2, [R0]       Copy data from R0 index into R2
        CMP  R2, R3         Compare R2 and R3, where R2 - R3
        BLS  Skip           if R2 - R3 != Negative or 0 != true, runs mov
        MOV  R3, R2         else, skip to code SUBS, decrement loop counter by 1
Skip    SUBS R1, R1, #1     Jumps back to loot till 0 flag is set true.
        BNE  Loop
```

CZ1106 CE1106

## Conditional Execution

- ARM instructions can be conditionally executed based on the CC flags.

ARM code example

```
;C code
if (R0 == 1)
 R1 = 3;
else
 R1 = 5;
```

```
        CMP   R0,#1          ; set CC based on r0 –1
        BNE   ELSE           ; if (R0 == 1)
        MOV   R1,#3          ; then { R1 := 3}
        B     SKIP           ; skip over else code seg
ELSE    MOV   R1,#5          ; else  { R1 := 5}
SKIP      .....              ;
```

- The conditional execution feature allows us to make the execution of each instruction dependent on the current status of the **N**, **Z**, **V**, **C** flags.

```
        CMP    R0,#1         ; if (r0 == 1)
Executes if Z = 1
        MOVEQ  R1,#3         ; then { r1 := 3}
Executes if Z = 0
        MOVNE  R1,#5         ; else  { r1 := 5}
SKIP    .....                ;
```

## All commands

MOV R1, R0 — copy operation — (Normal MOV)
Destination: R1 ↑ Source: R0

MOVS R0, R1 — copy operation — (Normal MOV, but set flags)
Destination: R0 ↑ Source: R1

MOV R1, #3 — copy operation — (Normal MOV, but uses Immediate Addressing)
Dest: R1 ↑ source: #3

LDR R1, [R0] — copy operation — (Copies R0 index and its length of data into R1)
Dest: R1 ↑ source: R0

LDR R1, [R0, #4] — copy operation — (Copies R0 index plus offset 4, its lengths of data into R1)
Dest: R1 ↑ source: R0 ↑ offset: #4

LDR R1, [R0, R2] — copy operation — (Copies R0 index plus offset from R2, its lengths of data into R1)
Dest: R1 ↑ source: R0 ↑ offset: R2

LDR R1, [R0, #4]! — (Auto-indexed R0 to increment by 4, then copies the data from that address to R1)
Dest: R1 ↑ source: R0 ↑ offset: #4 ↑ Auto index

LDR R1, [R0, R2]! — (Auto-indexed R0 to increment by R2, then copies the data from that addr to R1)
Dest: R1 ↑ source: R0 ↑ offset: R2 ↑ Auto index

Pre index

LDR R1, [R0], #4 — (Copies the data first from R0 to R1, then auto-indexes R0 by 4)
Dest: R1 ↑ source: R0 ↑ offset + auto index: #4

LDR R1, [R0], R2 — (Copies the data first from R0 to R1, then auto-indexes R0 by R2)
Dest: R1 ↑ source: R0 ↑ offset + auto-index: R2

Post index

MVN R1, R0 — (complements R0, then inserts into R1)
Dest: R1 ↑ Source: R0

LDRB R2, [R0] — (Copies 1 Byte / 8 bits value at addr R0 to R2)
Dest: R2 ↑ Source: R0

STR R1, [R0] — copy operation — (Copies R1 data to R0 index and its length)
source: R1 ↑ Dest: R0

STRB R2, [R0, #1]! — (Copies 1 Byte from R2 to R0+1 Address, then auto-indexes R0)
source: R2 ↑ Dest: R0 ↑ offset: #1 ↑ Auto index

STR R1, [R0], #4 — (Copies 32 bit from R1 to R0, then auto indexes R0 to new addr)
source: R1 ↑ Dest: R0 ↑ offset + auto index: #4

`ADD R2, R0, R1` (R2 = R0+R1)
Dest, +

`ADD R2, R0, #4` (R2 = R0 + #4)
Dest, +

`ADDS R2, R0, #8` (R2 = R0+#8 Set Flags too)
Dest, +

`SUB R2, R0, #4` (R2 = R0 - #4)
Dest, -

`SUBS R2, R0, #8` (R2 = R0 - #8 Set Flags too)
Dest, -

`RSBS R2, R0, #4` (R2 = #4 - R0 Set Flags too)
Dest, -

`ADC R2, R0, R1` (R2 = R0+R1+C)
Dest

`SBC R2, R0, R1` (R2 = R0 - R1 + NOT(C))
Dest

`RSC R2, R0, R1` (R2 = R1 - R0 + NOT(C))
Dest

Set R0 as 0101 0101

`AND R1, R0, # 1111 0000` → Result
0101 0101
1111 0000
0101 0101
→ R1 = 0101 0101

`ORR R0, R0, #1111 0000` → Result
0101 0101
1111 0000
1111 0101
→ R0 = 1111 0101

`EOR R2, R0, # 11110000` → Result
0101 0101
1111 0000
1010 0101
→ R2 = 1010 0101

## Shift and Rotate Mnemonics

`MOV R0, R0, LSL #1 ;` Shift R0 left by 1 bit

`ADD R2, R1, R0, LSR #2 ;` Shift R0 2 bits to the right then ADD

`ADDS R2, R1, R0, LSL R4 ;` Shift R0 by R4 bits to the left, then ADD. Influences Flags

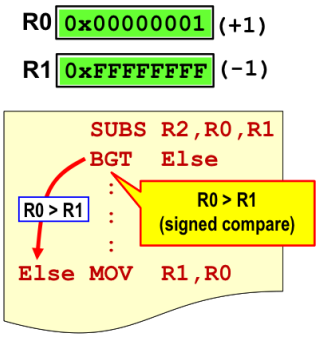
`ORRS R0, R0, R0, ROR #1 ;` OR R0 with a 1 bit-right rotated version of itself. Set C flag if bit rotated out is 1.

CZ1106
CE1106

## Different Bcc Conditions

- There are 15 possible conditional tests for **Bcc**.

| Suffix | Flags | Meaning | |
|---|---|---|---|
| EQ | Z = 1 | Equal | |
| NE | Z = 0 | Not equal | |
| CS or HS | C = 1 | Higher or same, **unsigned** | Unsigned comparison |
| CC or LO | C = 0 | Lower, **unsigned** | Unsigned comparison |
| MI | N = 1 | Negative | |
| PL | N = 0 | Positive or zero | |
| VS | V = 1 | Overflow | |
| VC | V = 0 | No overflow | |
| HI | C = 1 and Z = 0 | Higher, **unsigned** | Unsigned comparison |
| LS | C = 0 or Z = 1 | Lower or same, **unsigned** | Unsigned comparison |
| GE | N = V | Greater than or equal, **signed** | Signed comparison |
| LT | N != V | Less than, **signed** | Signed comparison |
| GT | Z = 0 and N = V | Greater than, **signed** | Signed comparison |
| LE | Z = 1 and N != V | Less than or equal, **signed** | Signed comparison |
| AL | Can have any value | Always. This is the default when no suffix is specified. | |






### CMP Command

CMP does the same thing as Subs, however, registers remains unchanged but cc-flags are set.

```
CMP R1, R1, R2    ; test (R1 - 4), where R1 is a signed no
BGE               ; branch to R1 ≥ 4 if result is positive (i.e R1 ≥ 4)
```

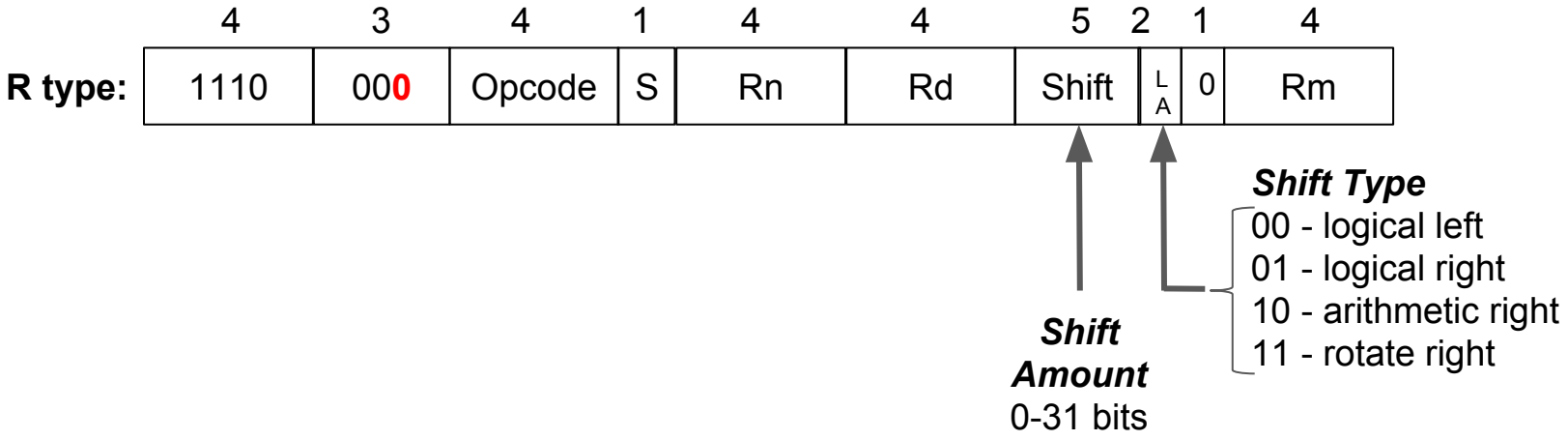
R1 ≥ R2

### Other Conditional Test Instructions

| | | |
|---|---|---|
| CMN R0, R1 ; | set (N, Z, C, V) based on R0+R1 | Compare Negative |
| TST R0, R1 ; | Set (N, Z, C) based on R0 AND R1 | Test Bits |
| TEQ R0, R1 ; | Set (N, Z, C) based on R0 EOR R1 | Test Equivalence |

# 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

LSL - Left shift ; multiply by $2^n$

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

**R0 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

LSR: Right shift ; divide by $2^s$ ; UNSIGNED

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

ASR : right shift ; divide by $2^n$ ; signed

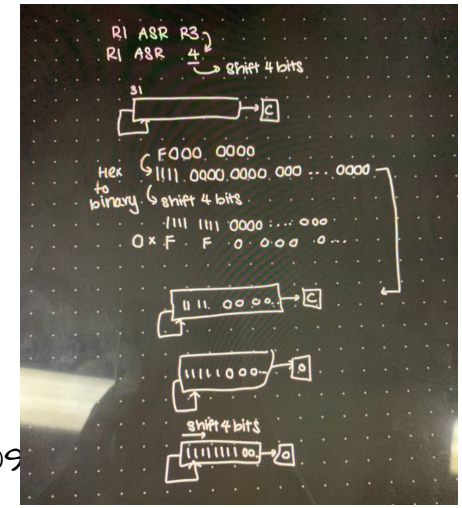
# Arithmetic Right Shifts

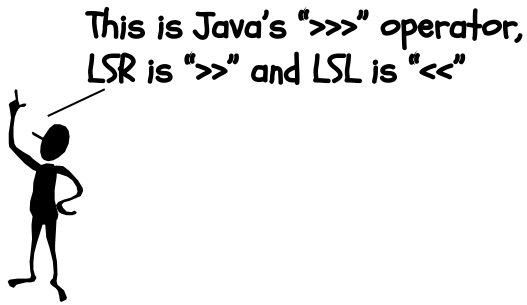
Arithmetic right shifts behave like ***dividing*** the contents of a register by $2^s$ where s is the shift amount, ***if*** you assume the contents of the register are ***signed***.

```
MOV R0,R0,ASR 2
```

**R0 before:** 1111 1111 1111 1111 1111 1100 0000 0000 = -1024

**R0 after:** 1111 1111 1111 1111 1111 1111 0000 0000 = -1024 / $2^2$ = -256

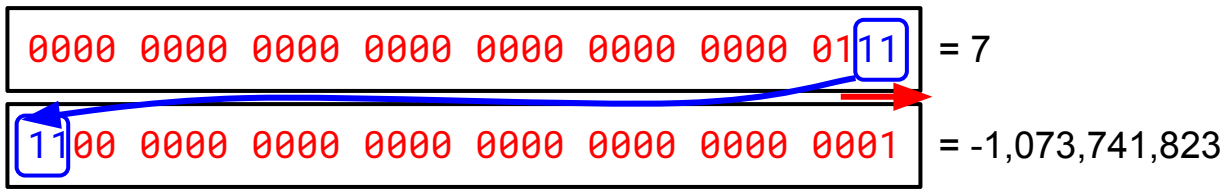


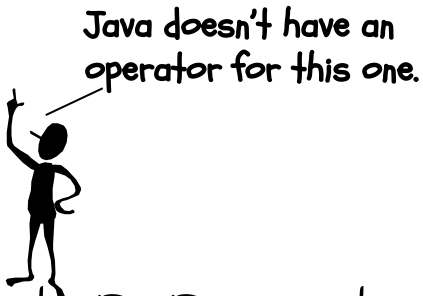
# Rotate Right Shifts

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
```




Why no rotate left shift?




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