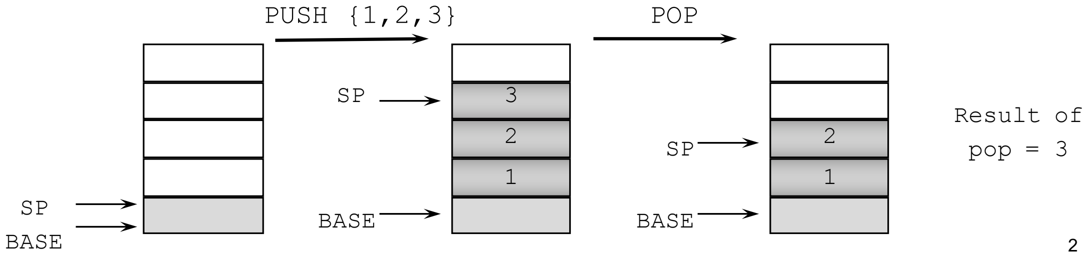
Memory Stack

Stacks

- A stack is an area of memory which grows as new data is "pushed" onto the "top" of it, and shrinks as data is "popped" off the top - LIFO Queue
- Two pointers define the current limits of the stack.
  - A base pointer
    - used to point to the "bottom" of the stack (the first location).
  - A stack pointer
    - used to point the current "top" of the stack.



Stack Operation

Multiple-register transfer instructions are particularly useful for stack operation.

Stack is used to save multiple-register content that may be affected when performing a subroutine call.

Using one `STM` instruction can save several registers onto the stack.

Example:
To save multiple-register content on a stack

```
        STMDB    sp!, {r4-r8}
```

This instruction ‘pushes’ the four-register content to the stack.

The corresponding instruction to ‘pop’ the content from the stack is

```
        LDMIA    sp!, {r4-r8}
```

Stack Implementations

The option of using ‘`DB`’ or ‘`DA`’ for push operation (& ‘`IA`’ or ‘`IB`’ for corresponding pop operation) depends on the way a stack is implemented.

- Full stack: address pointed by sp is already used
- Empty stack: address pointed by sp is still empty
- Descending: stack grows down in memory address
- Ascending: stack grows up in memory address

Discussion:
The earlier example uses `STMDB` and `LDMIA` to manipulate the stack content. What type of stack is this?

Stack-Specific Instructions

To avoid using the wrong type of `STM` and `LDM` instructions when dealing with stack operation

- special suffixes are used instead to match the different stack implementation

- `FD`: for Fully Descending stack
- `FA`: for Fully Ascending stack
- `ED`: for Empty Descending stack
- `EA`: for Empty Ascending stack

Example:
For the ARM processor that uses a fully descending stack, the pair of stack instructions is `STMFD` and `LDMFD`.

Stack Operation

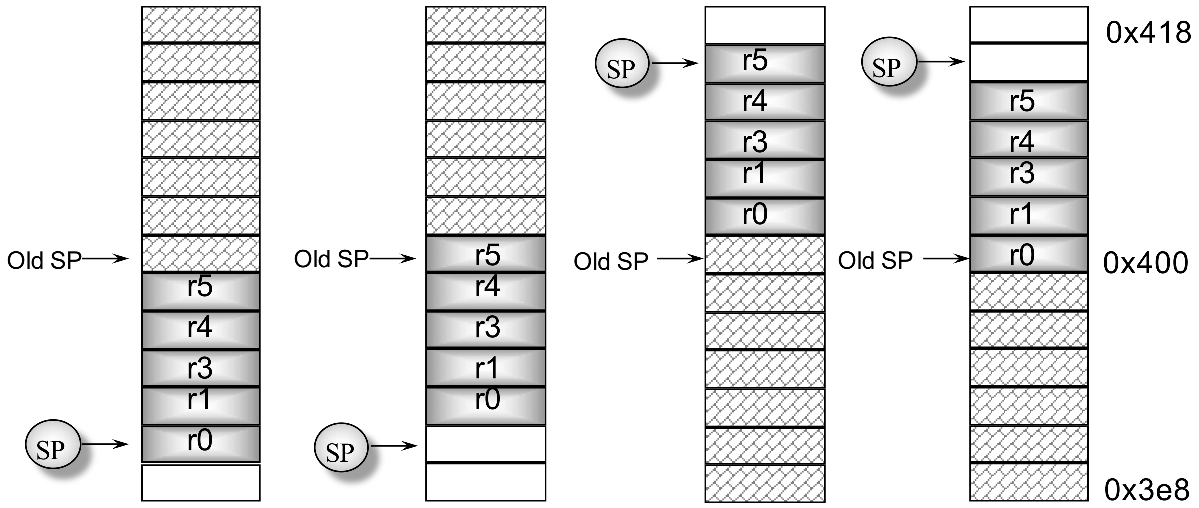
- Stack Usually Grows Down in Memory
  - last “pushed” value is at the lowest address
- ARM also supports ascending stacks
  - stack structure grows up through memory.
- The value of the stack pointer can either:
  - Point to the last occupied address (Full stack)
    - needs pre-decrementing (ie before the push)
  - Point to the next occupied address (Empty stack)
    - needs post-decrementing (ie after the push)

Stack Operation

- Stack Type to be used is given by the postfix to the instruction:
  - `STMFD` / `LDMFD` : Full Descending stack
  - `STMFA` / `LDMFA` : Full Ascending stack.
  - `STMED` / `LDMED` : Empty Descending stack
  - `STMEA` / `LDMEA` : Empty Ascending stack
- ARM Compiler will always use a Full descending stack

Stack Examples

`STMED sp!,{r0,r1,r3-r5}` `STMEA sp!,{r0,r1,r3-r5}`



`STMFD sp!{r0,r1,r3-r5}` `STMFA sp!,{r0,r1,r3-r5}`

Stacks and Subroutines

- Stacks can be Used to create temporary register workspace for subroutines.
- Any registers that are needed can be pushed onto the stack at the start of the subroutine and popped off again at the

```
STMFD sp!,{r0-r12, lr}        ; stack all registers
........                      ; and the return address
........
LDMFD sp!,{r0-r12, pc}        ; load all the registers
                              ; and return automatically
```

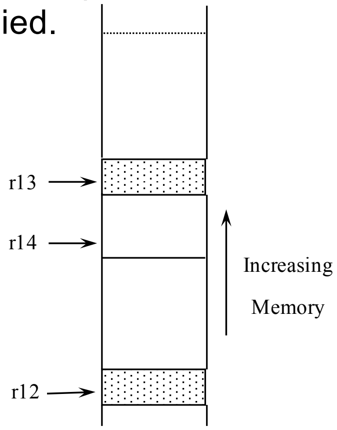
Functionality of Block Data Transfer

- When `LDM` / `STM` are not being used to implement stacks, it is clearer to specify exactly what functionality of the instruction is:
  - i.e. specify whether to increment / decrement the base pointer, before or after the memory access.
- In order to do this, `LDM` / `STM` support a further syntax in addition to the stack one:
  - `STMIA` / `LDMIA` : Increment After
  - `STMIB` / `LDMIB` : Increment Before
  - `STMDA` / `LDMDA` : Decrement After
  - `STMDB` / `LDMDB` : Decrement Before

Example: Block Copy

- Copy a block of memory, which is an exact multiple of 12 words long from the location pointed to by r12 to the location pointed to by r13. r14 points to the end of block to be copied.

```
; r12 points to the start of the source data
; r14 points to the end of the source data
; r13 points to the start of the destination data
loop    LDMIA   r12!, {r0-r11} ; load 48 bytes
        STMIA   r13!, {r0-r11} ; and store them
        CMP     r12, r14       ; check for the end
        BNE     loop           ; and loop until done
```



- This loop transfers 48 bytes in 31 cycles
- Over 50 Mbytes/sec at 33 MHz

Load and Stores - User Mode Privilege

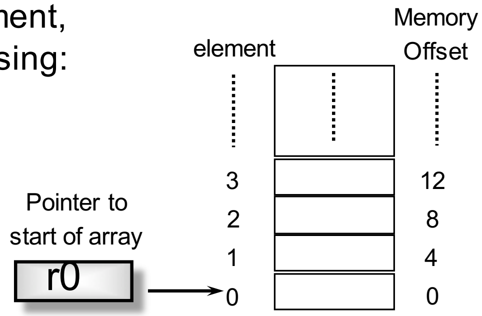
- When using post-indexed addressing, there is a further form of Load/Store Word/Byte:

```
<LDR|STR>{<cond>}{B}T Rd, <post_indexed_address>
```

- When used in a privileged mode, this does the load/store with user mode privilege.
  - Normally used by an exception handler that is emulating a memory access instruction that would normally execute in user mode.

Example Usage of Addressing Modes

- Imagine an array, the first element of which is pointed to by the contents of `r0`.
- If we want to access a particular element, then we can use pre-indexed addressing:
  - `r1` is element we want.
  - `LDR r2, [r0, r1, LSL #2]`



- If we want to step through every element of the array, for instance to produce sum of elements in the array, then we can use post-indexed addressing within a loop:
  - `r1` is address of current element (initially equal to r0).
  - `LDR r2, [r1], #4`

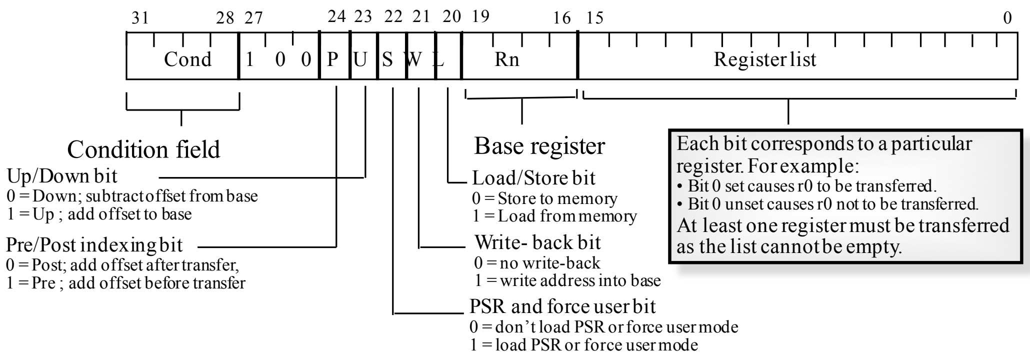
  Use a further register to store the address of final element, so that the loop can be correctly terminated.

Offsets for Halfword and Signed Halfword / Byte Access

- The Load and Store Halfword and Load Signed Byte or Halfword instructions can make use of pre- and post-indexed addressing in much the same way as the basic load and store instructions.
- However the actual offset formats are more constrained:
  - The immediate value is limited to 8 bits (rather than 12 bits) giving an offset of 0-255 bytes.
  - The register form ***cannot*** have a shift applied to it.

Block Data Transfer

- The Load and Store Multiple instructions (`LDM/STM`) allow between 1 and 16 registers to be transferred to or from memory.
- The transferred registers can be either:
  - Any subset of the current bank of registers (default).
  - Any subset of the user mode bank of registers when in a privileged mode (postfix instruction with a '^').



31 28	27	24	23	22	21	20	19 16	15 0
Cond	1 0 0	P	U	S	W	L	Rn	Register list

Condition field

Up/Down bit
0 = Down; subtract offset from base
1 = Up ; add offset to base

Pre/Post indexing bit
0 = Post; add offset after transfer,
1 = Pre ; add offset before transfer

Base register

Load/Store bit
0 = Store to memory
1 = Load from memory

Write- back bit
0 = no write-back
1 = write address into base

PSR and force user bit
0 = don't load PSR or force user mode
1 = load PSR or force user mode

Each bit corresponds to a particular register. For example:
- Bit 0 set causes r0 to be transferred.
- Bit 0 unset causes r0 not to be transferred.

At least one register must be transferred as the list cannot be empty.

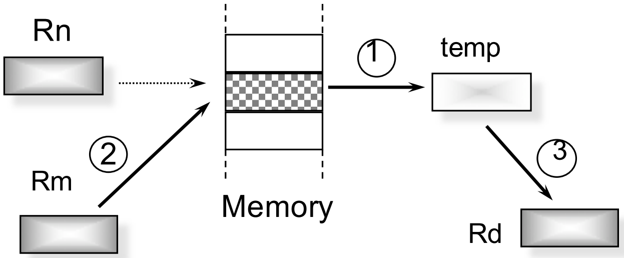
Block Data Transfer

- Base register used to determine where memory access should occur.
  - 4 different addressing modes allow increment and decrement inclusive or exclusive of the base register location.
  - Base register can be optionally updated following the transfer (by appending it with an '**!**'.
  - Lowest register number is always transferred to/from lowest memory location accessed.
- These instructions are very efficient for
  - Saving and restoring context
    - For this useful to view memory as a stack.
  - Moving large blocks of data around memory
    - For this useful to directly represent functionality of the instructions.

Swap and Swap Byte Instructions

- Atomic operation of a memory read followed by a memory write which moves byte or word quantities between registers and memory.
- Syntax:

```
SWP{<cond>}{B} Rd, Rm, [Rn]
```



- Thus to implement an actual swap of contents make Rd = Rm.
- The compiler cannot produce this instruction.

Data Movement Instructions

ARM can only manipulate data within registers.

- so data has to be loaded from memory into register(s) for data operation
- with results eventually stored back into memory

A lot of instructions in a running program involve data movement.

- important to have efficient addressing mode and data loading and storing instructions
- to optimize processor performance

MOV Format

General Form:

31 28	27	26	25	24 21	20	19 16	15 12	11 0
cond	0	0	1	opcode	S	Rn	Rd	shifter_operand

Form with Immediate Operand:

31 28	27	26	25	24 21	20	19 16	15 12	11 8	7 0
cond	0	0	1	opcode	S	Rn	Rd	rotate_imm	8_bit_immediate

8-bit Immediate Means [0,255] Only !!!!!

Condition Codes

All ARM Instructions utilize condition codes:

Cond	Instruction Bitmap	No	Cond Code	Executes if
0000xxxx	xxxxxxxx xxxxxxxx xxxxxxxx	0	EQ(Equal)	Z
0001xxxx	xxxxxxxx xxxxxxxx xxxxxxxx	1	NE(Not Equal)	~Z
0010xxxx	xxxxxxxx xxxxxxxx xxxxxxxx	2	CS(Carry Set)	C
0011xxxx	xxxxxxxx xxxxxxxx xxxxxxxx	3	CC(Carry Clear)	~C
0100xxxx	xxxxxxxx xxxxxxxx xxxxxxxx	4	MI(MInus)	N
0101xxxx	xxxxxxxx xxxxxxxx xxxxxxxx	5	PL(PLus)	~N
0110xxxx	xxxxxxxx xxxxxxxx xxxxxxxx	6	VS(oVerflow Set)	V
0111xxxx	xxxxxxxx xxxxxxxx xxxxxxxx	7	VC(oVerflow Clear)	~V
1000xxxx	xxxxxxxx xxxxxxxx xxxxxxxx	8	HI(HIgher)	C and ~Z
1001xxxx	xxxxxxxx xxxxxxxx xxxxxxxx	9	LS(Lower or Same)	~C and Z
1010xxxx	xxxxxxxx xxxxxxxx xxxxxxxx	A	GE(Greater or equal)	N = V
1011xxxx	xxxxxxxx xxxxxxxx xxxxxxxx	B	LT(Less Than)	N = ~V
1100xxxx	xxxxxxxx xxxxxxxx xxxxxxxx	C	GT(Greater Than)	(N= V)and~Z
1101xxxx	xxxxxxxx xxxxxxxx xxxxxxxx	D	LE(Less or equal)	(N =~V)orZ
1110xxxx	xxxxxxxx xxxxxxxx xxxxxxxx	E	AL(Always)	True
1111xxxx	xxxxxxxx xxxxxxxx xxxxxxxx	F	NV(Never)	False

Data Movement

- Operations are:
  - `MOV` Rd <- operand2
  - `MVN` Rd <- NOT operand2

- Syntax:
  - `<Operation>{<cond>}{S} Rd, Operand2`

- Examples:

```
– MOV     r0, r1       ;r0 <- r1
– MOVS    r2, #10      ;r0 <- #10
– MVNEQ   r1, #0       ;if(Z==0) r1 <-0xFFFFFFFF
```

Loading full 32 bit constants

- If Instructions are 32 bits long (16 for thumb), how are 32-bit immediate values encoded into an Instruction?
  - USES LITERAL POOLS

`MVNEQ r1, #0` *Instruction useful since allows* `0xFFFFFFFF` *to be used as 32-bit "immediate" in 32-bit Instruction*

Literal Pools

- Literals Pools – Constant Data Areas Embedded in the Code
  - At end of modules, between functions
- Although the `MOV/MVN` mechanism in combination with shifts will load a large range of constants into a register, sometimes this mechanism will not generate the required constant.
- Therefore, the assembler also provides a method which will load *ANY* 32 bit constant:

```
– LDR rd, =numeric constant
```

Literal Pools

- If the constant can be constructed using either a `MOV` or `MVN` then this will be the instruction actually generated by the assembler
- Otherwise, the assembler will produce an `LDR` instruction with a PC(r15)-relative address to read the constant from a literal pool
- Pseudo-instructions

```
– LDR r0,=0x42        ;generates MOV r0,#0x42
– LDR r0,=0x55555555  ;generate LDR r0,[pc, offset to lit pool]
```

- As this mechanism will always generate the best instruction for a given case, ***it is the recommended way of loading constants***.

Loading 32 bit constants

- To allow larger constants to be loaded, the assembler offers a pseudo-instruction:
  - `LDR rd, =const`
- This will either:
  - Produce a **`MOV`** or **`MVN`** instruction to generate the value (if possible).

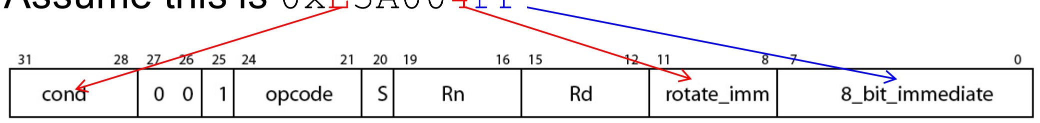
  or
  - Generate a **`LDR`** instruction with a PC-relative address to read the constant from a *literal pool* (Constant data area embedded in the code).
- For example

```
– LDR r0,=0xFF          =>     MOV r0,#0xFF
– LDR r0,=0x55555555    =>     LDR r0,[PC,#Imm12]
                               …
                               …
                               DCD 0x55555555
```

- This is the recommended way of loading constants into a register

MOV Format

Assume this is `0xE3A004FF`



31–28	27	26	25	24–21	20	19–16	15–12	11–8	7–0
cond	0	0	1	opcode	S	Rn	Rd	rotate_imm	8_bit_immediate

```
mov   r0,  #0xFF,  8    ;r0 <- 0xFF000000
```

Last 12 bits of machine code are `0x4FF`

Rotation Value is ALWAYS Multiplied by 2 in Machine Code: (4*2=8) So Can Only Rotate by EVEN Values, ODD Value causes Next Lower Even Rotate to Occur

8-bit Immediate is `0xFF` Can Only Use `0x01` thru `0xFF`

Value 8 Causes Immendiate to be Rotated to Right by 8 Bits

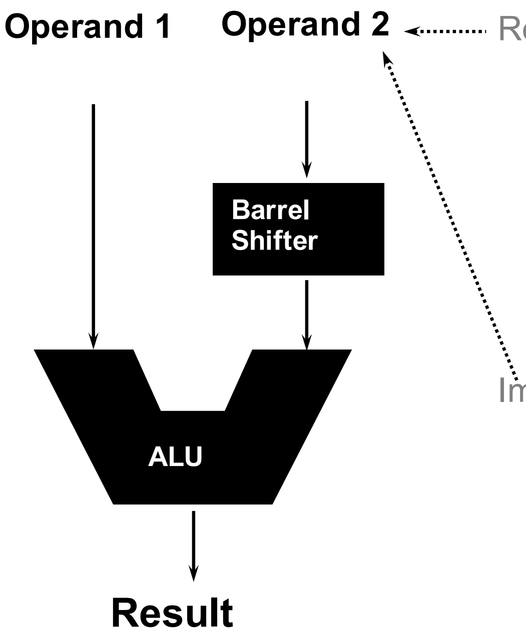
MOV with Rotate Right Operand

31 28	27 26	25	24 21	20	19 16	15 12	11 8	7 0
cond	0 0	1	opcode	S	Rn	Rd	rotate_imm	8_bit_immediate

```
mov  r0, #0xFF, <n>
```

n Value	Operation
30	;r0 <- 0x000003FC
28	;r0 <- 0x00000FF0
26	;r0 <- 0x00003FC0
24	;r0 <- 0x0000FF00
22	;r0 <- 0x0003FC00
	.....
8	;r0 <- 0xFF000000
6	;r0 <- 0xFC000003
4	;r0 <- 0xF000000F
2	;r0 <- 0xC000003F

Using a Barrel Shifter:The 2nd Operand



Register, optionally with shift operation

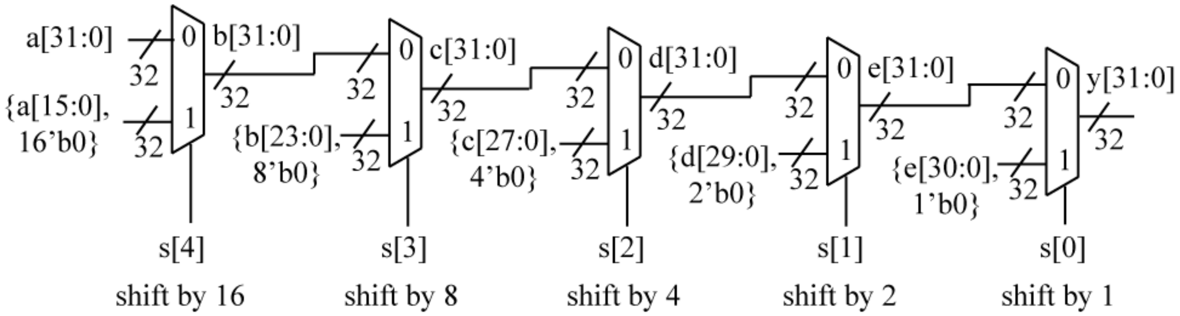
- Shift value can be either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.
- Used for multiplication by constant

Immediate value

- 8 bit number, with a range of 0-255.
  - Rotated right through number of positions
- Allows increased range of 32-bit constants to be loaded directly into registers

Barrel Shifter

(a) Multiplexer implementation of a 32-bit left-shift barrel shifter



(b) Verilog implementation

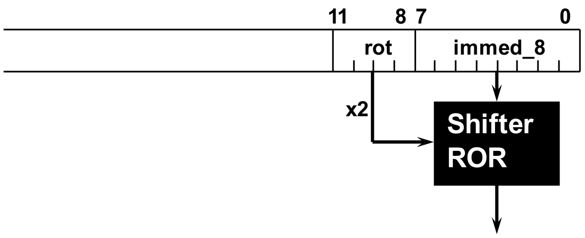
```
module bshift_32bit (s, a, y);
input [4:0] s;
input [31:0] a;
output [31:0] y;

assign y = a << s;

endmodule
```

Immediate constants

- No ARM instruction can contain a 32 bit immediate constant
  - All ARM instructions are fixed as 32 bits long
- The data processing instruction format has 12 bits available for operand2



ARM's scheme

has in-line ROR

- 4 bit rotate value (0-15) is multiplied by two to give range 0-30 (in steps of 2 in Machine Code)

Immediate constants

- Many Constants can be Generated by Using the In-Line Shifter
- Not ALL Constants can However
- For those that Can’t be Generated using the Shifter:
  - Literal Pools with `MOV` Translated to `LDR`
- Examples:

```
mov       r0, #0xff          ;r0 <- 255
mov       r0, #0xff, 30      ;r0 <- 1020
mov       r0, #0xff, 26      ;r0 <- 4096
add       r0, r2, #0xff000000      ;r0 <- r2 + 0xff000000
sub       r2, r3, #0x8000    ;r2 <- r3 – 0x8000
rsb       r8, r9, #0x8000    ;r8 <- 0x8000 – r9
                             ;   (reverse sub)
```

Example

- Want to generate constant 4080 using byte rotation
- $(4080)_{10}=(1111\ 1111\ 0000)_2$

```
mov       r0, #0xff, 28  ;r0 <- 4080
```

- Want to shift FF 4-bits to Left
- ARM Only has Right Rotator
- Rotate Right by 28 is Equal to Left Shift by 4:

  (32 – 4) = 28

MVN Instruction

- Moves One's Complement Value into Register
- One's Complement is bit-by-bit Complement
- Examples:

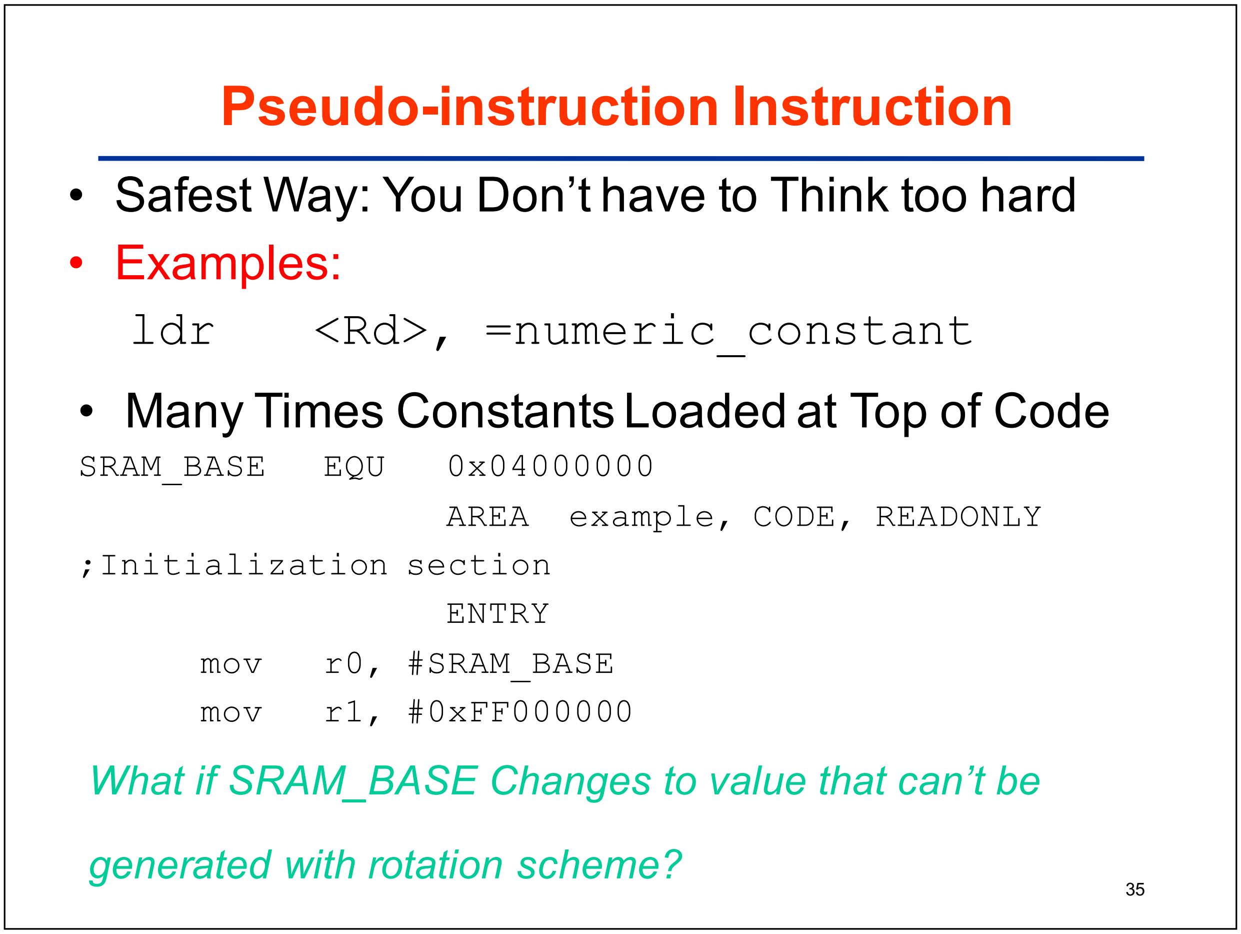
```
mvn    r0, #0       ;What is in r0?


mvn    r0, #0xFF, 8 ;What is in r0?
```

MVN Instruction

- Moves One's Complement Value into Register
- One's Complement is bit-by-bit Complement
- Examples:

```
mvn    r0, #0       ;What is in r0?
                ;r0 <- 0xFFFFFFFF
mvn    r0, #0xFF, 8 ;What is in r0?
                ;r0 <- 0x00FFFFFF
```



Pseudo-instruction Instruction

- Safest Way: You Don't have to Think too hard
- Examples:

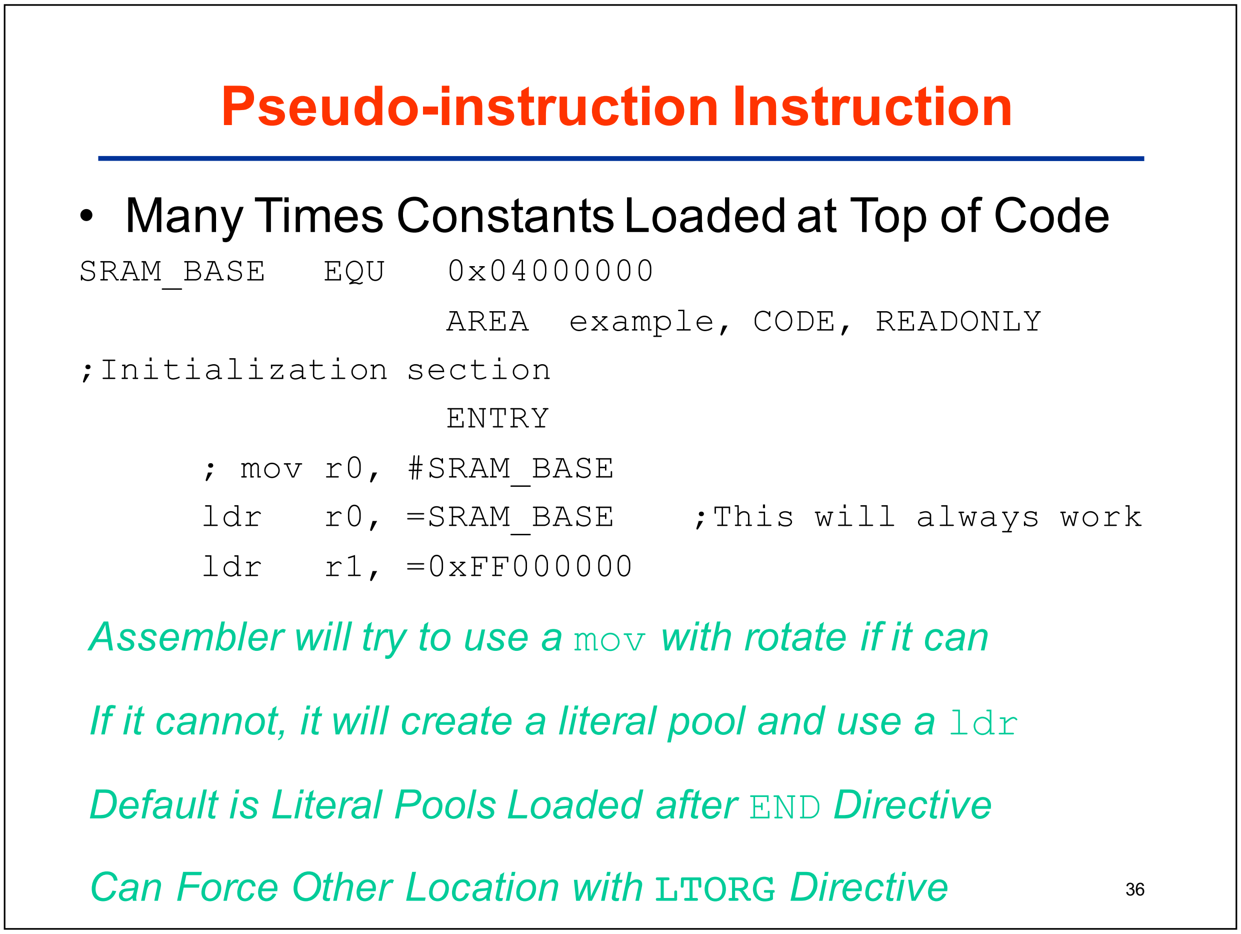
```
 ldr    <Rd>, =numeric_constant
```

- Many Times Constants Loaded at Top of Code

```
SRAM_BASE   EQU   0x04000000
                  AREA  example, CODE, READONLY
;Initialization section
                  ENTRY
      mov   r0, #SRAM_BASE
      mov   r1, #0xFF000000
```

What if SRAM_BASE Changes to value that can't be

generated with rotation scheme?



Pseudo-instruction Instruction

- Many Times Constants Loaded at Top of Code

```
SRAM_BASE   EQU   0x04000000
                  AREA  example, CODE, READONLY
;Initialization section
                  ENTRY
      ; mov r0, #SRAM_BASE
      ldr   r0, =SRAM_BASE    ;This will always work
      ldr   r1, =0xFF000000
```

Assembler will try to use a `mov` *with rotate if it can*

If it cannot, it will create a literal pool and use a `ldr`

Default is Literal Pools Loaded after `END` *Directive*

Can Force Other Location with `LTORG` *Directive*