Example: if statement

- C:

```
if (a > b) { x = 5; y = c + d; } else x = c - d;
```

- Assembler:

```
; compute and test condition
  LDR R4,=A ; get address for a
  LDR R0,[R4] ; get value of a
  LDR R4,=B ; get address for b
  LDR R1,[R4] ; get value for b
  CMP R0,R1 ;
  BLE fblock ;
```

If statement, cont'd.

```
; true block
   MOV R0,#5 ; generate value for x
   LDR R4,=X ; get address for x
   STR R0,[R4] ; store x
   LDR R4,=C ; get address for c
   LDR R0,[R4] ; get value of c
   LDR R4,=D ; get address for d
   LDR R1,[R4] ; get value of d
   ADD R0,R0,R1 ; compute y
   LDR R4,=Y ; get address for y
   STR R0,[R4] ; store y
   B after ; branch around false block
```

If statement, cont'd.

```
; false block
fblock LDR R4,=C ; get address for c
   LDR R0,[R4] ; get value of c
   LDR R4,=D ; get address for d
   LDR R1,[R4] ; get value for d
   SUB R0,R0,R1 ; compute c-d
   LDR R4,=X ; get address for x
   STR R0,[R4] ; store value of x
after ...
```

Example

- C:

```
for (i=0, f=0; i<N; i++)
  f = f + c[i]*x[i];
```

- Assembler

```
; loop initiation code
  MOV R0,#0 ; use r0 for I
  MOV R8,#0 ; use separate index for arrays
  LDR R2,=N ; get address for N
  LDR R1,[R2] ; get value of N
  MOV R2,#0 ; use r2 for f
```

Example, cont'.d

```
     LDR R3,=C ; load r3 with base of c
     LDR R5,=X ; load r5 with base of x
; loop body
loop   LDR R4,[R3,R8] ; get c[i]
       LDR r6,[R5,R8] ; get x[i]
       MUL R4,R4,R6 ; compute c[i]*x[i]
       ADD R2,R2,R4 ; add into running sum
       ADD R8,R8,#4 ; add one word offset to array index
       ADD R0,R0,#1 ; add 1 to i
       CMP R0,R1 ; exit?
       BLT loop ; if i < N, continue
```

Set Bit Example

The **or** operation to set bits 1 and 0 of a register.

The other six bits remain constant.

Friendly software modifies just the bits that need to be.

```
X |= 0x03;
```

Assembly:

```
LDR   R0,=X
LDR   R1,[R0]        ; read previous value
ORR   R1,R1,#0x03    ; set bits 0 and 1
STR   R1,[R0]        ; update
```

c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0	value of R1
0	0	0	0	0	0	1	1	`0x03` constant
c_7	c_6	c_5	c_4	c_3	c_2	1	1	result of the `ORR`

Toggle Bit example

The **exclusive or** operation can also be used to toggle bits.

```
X ^= 0x80;
```

Assembly:

```
LDR   R0,=X
LDR   R1,[R0]       ; read port D
EOR   R1,R1,#0x80   ; toggle bit 7
STR   R1,[R0]       ; update
```

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	value of R1
1	0	0	0	0	0	0	0	**0x80** constant
$\sim b_7$	b_6	b_5	b_4	b_3	b_2	b_1	b_0	result of the **EOR**

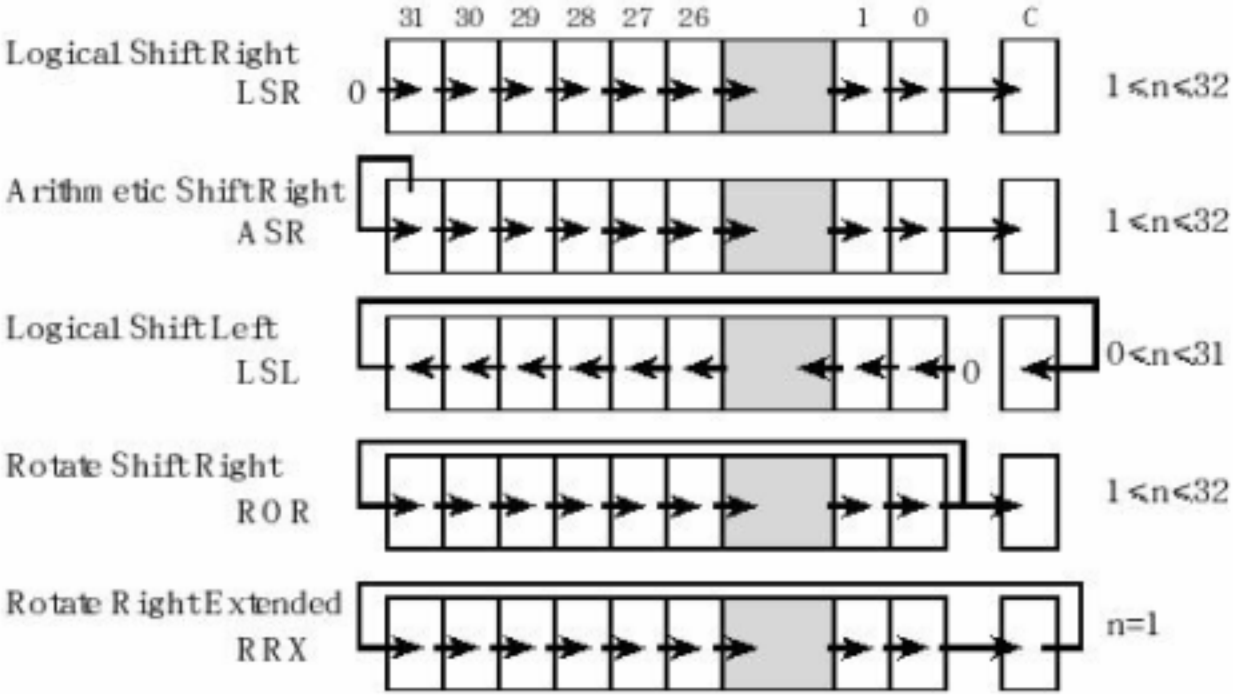


Figure 3.14. Shift operations.

```
LSR{S}{cond} Rd, Rm, Rs  ; logical shift right Rd=Rm>>Rs  (unsigned)
LSR{S}{cond} Rd, Rm, #n  ; logical shift right Rd=Rm>>n   (unsigned)
ASR{S}{cond} Rd, Rm, Rs  ; arithmetic shift right Rd=Rm>>Rs (signed)
ASR{S}{cond} Rd, Rm, #n  ; arithmetic shift right Rd=Rm>>n (signed)
LSL{S}{cond} Rd, Rm, Rs  ; shift left Rd=Rm<<Rs (signed, unsigned)
LSL{S}{cond} Rd, Rm, #n  ; shift left Rd=Rm<<n  (signed, unsigned)
ROR{S}{cond} Rd, Rm, Rs  ; rotate right
ROR{S}{cond} Rd, Rm, #n  ; rotate right
RXX{S}{cond} Rd, Rm      ; rotate right 1 bit with extension
```

Shift Example

`High` and **`Low`** are unsigned 4-bit components, which will be combined into a single unsigned 8-bit **`Result`**.

```
Result = (High<<4)|Low;
```

Assembly:

```
LDR   R0,=High
LDR   R1,[R0]         ; read value of High
LSL   R1,R1,#4        ; shift into position
LDR   R0,=Low
LDR   R2,[R0]         ; read value of Low
ORR   R1,R1,R2        ; combine the two parts
LDR   R0,=Result
STR   R1,[R0]         ; save the answer
```

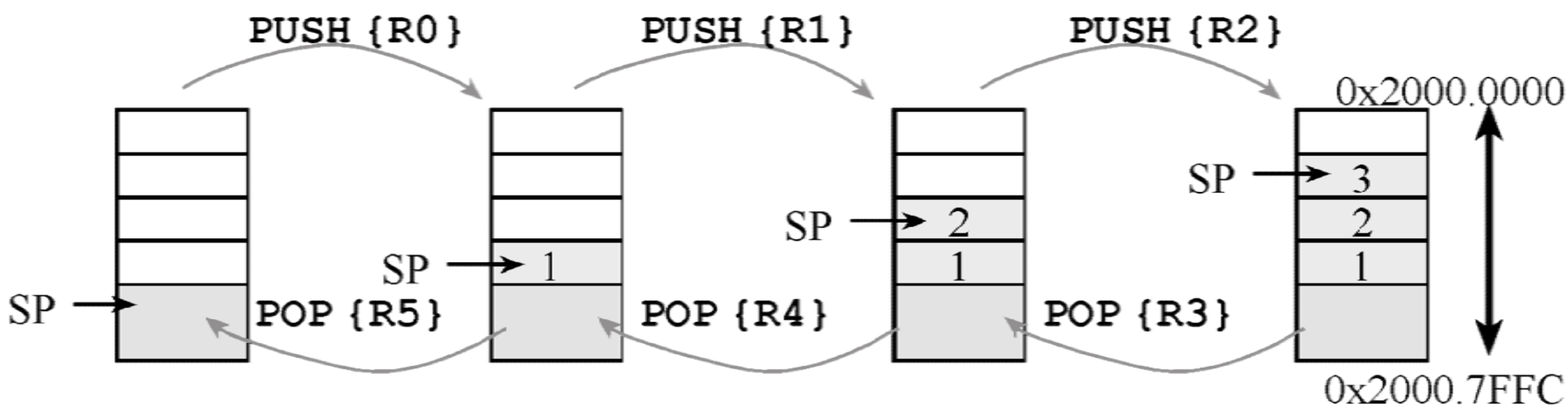
0	0	0	0	h_3	h_2	h_1	h_0	value of **`High`** in **`R1`**
h_3	h_2	h_1	h_0	0	0	0	0	after last **`LSL`**
0	0	0	0	l_3	l_2	l_1	l_0	value of **`Low`** in **`R2`**
h_3	h_2	h_1	h_0	l_3	l_2	l_1	l_0	result of the **`ORR`** instruction

The Stack

- Stack is last-in-first-out (LIFO) storage
  - 32-bit data
- Stack pointer, SP or R13, points to top element of stack
- Stack pointer *decremented* as data placed on stack
- **`PUSH`** and **`POP`** instructions used to load and retrieve data

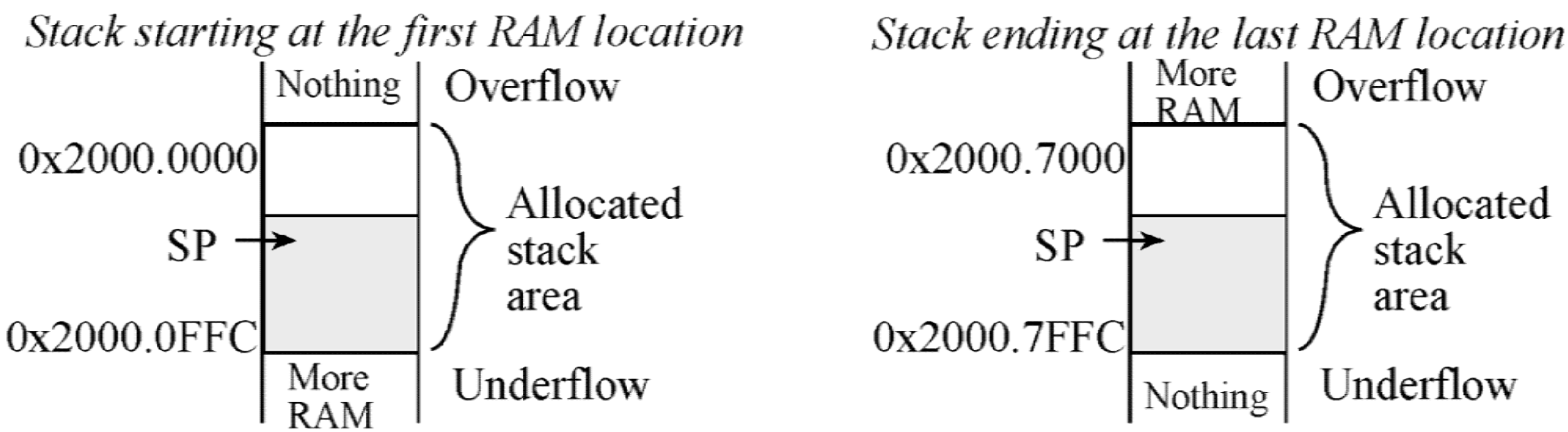
The Stack

- Stack is last-in-first-out (LIFO) storage
  - 32-bit data
- Stack pointer, SP or R13, points to top element of stack
- Stack pointer *decremented* as data placed on stack (*incremented* when data is removed)
- **`PUSH`** and **`POP`** instructions used to load and retrieve data



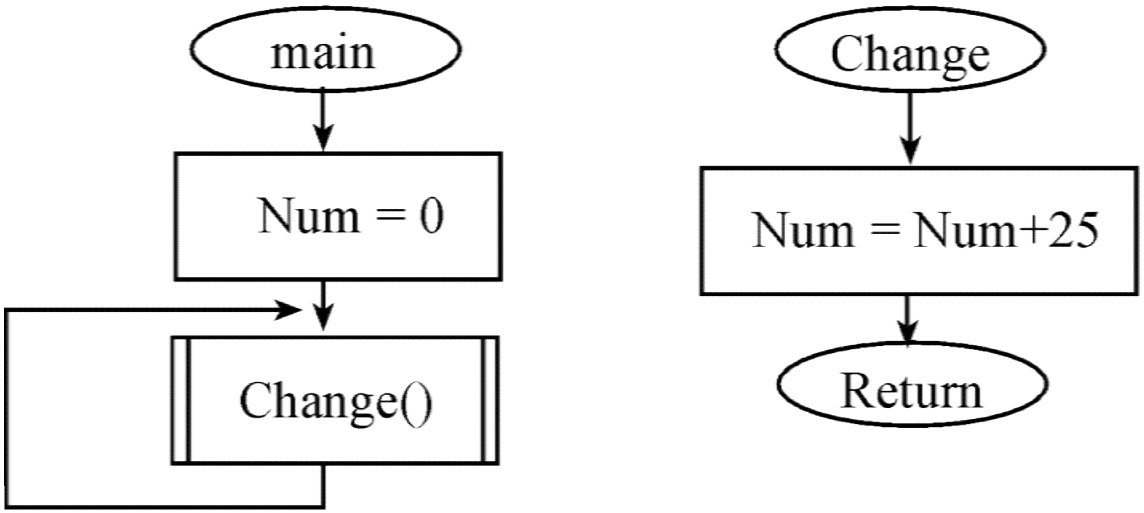
Stack Usage

- Stack memory allocation



- Rules for stack use
  - Stack should always be balanced, i.e. functions should have an equal number of pushes and pops
  - Stack accesses (push or pop) should not be performed outside the allocated area

Functions




```
Change LDR   R1,=Num    ; 5) R1 = &Num
       LDR   R0,[R1]    ; 6) R0 = Num
       ADD   R0,R0,#25  ; 7) R0 = Num+25
       STR   R0,[R1]    ; 8) Num = Num+25
       BX    LR         ; 9) return
main   LDR   R1,=Num    ; 1) R1 = &Num
       MOV   R0,#0      ; 2) R0 = 0
       STR   R0,[R1]    ; 3) Num = 0
loop   BL    Change     ; 4) function call
       B     loop       ; 10) repeat
```

```
unsigned long Num;
void Change(void){
  Num = Num+25;
}
void main(void){
  Num = 0;
  while(1){
    Change();
  }
}
```