Instruction sets

Overview

- Logic operations
- Shift and rotate operations
- Compare instructions
- Program Flow instructions
- Program control instructions

Logic Operations

- Logical AND(ANDS)
- Logical OR(ORRS)
- Logical Exclusive OR(EORS)
- Logical Bitwise clear(BICS)
- Logical Bitwise NOT(MVNS)
- Test Bitwise AND(TST)

Logic Operations

- Logical AND:

  ANDS <Rd>,<Rd>,<Rm> $\rightarrow$APSR N, Z updates

- Logical OR:

  ORRS <Rd>,<Rd>,<Rm> $\rightarrow$APSR N, Z updates

- Logical Exclusive OR:

  EORS <Rd>,<Rd>,<Rm> $\rightarrow$APSR N, Z updates

<Instruction> <Rdest>,<Rn1>,<Rn2>

Logic Operations

- Logical Bitwise clear:

  BICS <Rd>,<rd>,<Rm> → Rd=AND(Rd,NOT(Rm)), APSR N, Z updates

- Logical Bitwise NOT:

  MVNS <Rd>,<Rm> →Rd=NOT(Rm), APSR N, Z updates

- Test Bitwise AND:

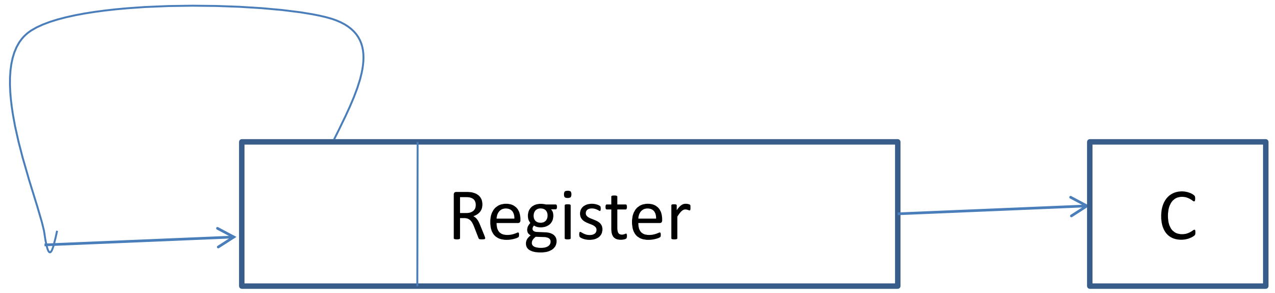
  TST <Rn>,<Rm> →AND(Rn,Rm), APSR N, Z updates, but AND result is not stored

Shift and Rotate operations

- Arithmetic Shift Right Shift(ASRS)
- Logical Shift Left Shift(LSLS)
- Logical Shift Right Shift(LSRS)
- ROtate Right(ROR)

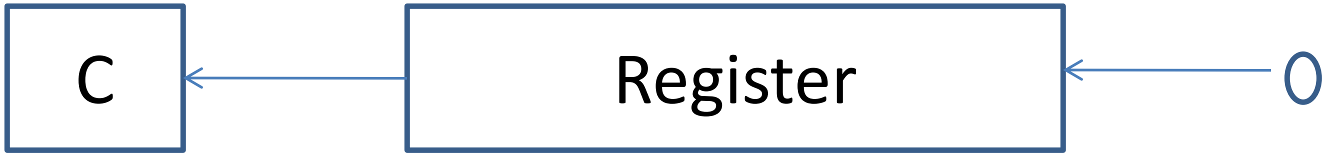
Shift and Rotate operations

- Arithmetic Shift Right(ASRS):
  - **ASRS <Rd>,<Rd>,<Rm>**
  - **ASRS <Rd>,<Rm>,#immed5**
  - Rd=Rd>>Rm, last bit out is copied to APSR C
  - APSR N and Z updates



Shift and Rotate operations

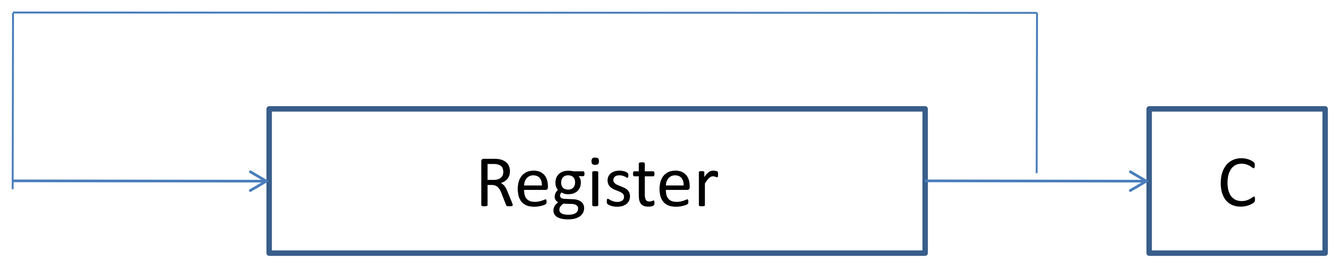
- Logical Shift Left(LSLS):
  - **LSLS <Rd>,<Rd>,<Rm>**
  - **LSLS <Rd>,<Rm>,#immed5**
  - Rd=Rd<<Rm, last bit out is copied to APSR C
  - APSR N and Z updates



R2: 00001111 << 2
Result R1: 00111100

Shift and Rotate operations

- Rotate Right(ROR):
  - **RORS <Rd>,<Rd>,<Rm>**
  - Rd=Rotate by Rm bits , last bit out is copied to APSR C
  - APSR N and Z updates



Compare Instructions

- Compare(CMP):
  - **CMP <Rn>,<Rm>**
  - Calculates Rn-Rm, but result is not stored
- Compare(CMP):
  - **CMP <Rn>,#immed8**
- Compare Negative(CMN):
  - **Rn-NEG(Rm)**

Program Flow Instructions

- Branch(B-unconditional)
- Branch(B<cond>-Conditional)
- Branch and Link(BL)
- Branch and Exchange(BX)
- Branch and Link with Exchange(BLX)

Extend and Reverse Ordering Operations

- **Instruction REV (Byte-Reverse Word)**
- Function Byte Order Reverse
  - **Syntax REV <Rd>, <Rm>**
  - **Note Rd ={Rm[7:0] , Rm[15:8], Rm[23:16], Rm[31:24]}**
- Rd and Rm are low registers.

0x12345678
0x78563412

Rm = 0x12345678
Rev16 Rm = 0x3412678

- **REV16 (Byte-Reverse Packed Half Word)**
- Function Byte Order Reverse within half word
    - **Syntax REV16 <Rd>, <Rm>**
    - **Note Rd = {Rm[23:16], Rm[31:24], Rm[7:0] , Rm[15:8]}**
- Rd and Rm are low registers.

Rm = 0xFF82
Sign-extended Result Rd: 0xFFFFFF82

- **REVSH (Byte-Reverse Signed Half Word)**
- Function Byte order reverse within lower half word, then sign extend result
    - **Syntax REVSH <Rd>, <Rm>**
    - **Note Rd =SignExtend({Rm[7:0] , Rm[15:8]})**
- Rd and Rm are low registers.

- **SXTB (Signed Extended Byte)**
- Function SignExtend lowest byte in a word of data
    - **Syntax SXTB <Rd>, <Rm>**
    - **Note Rd =SignExtend(Rm[7:0])**
- Rd and Rm are low registers.

Rm = 0xFF82

Sign-extended Result Rd: 0xFFFFFF82

- **SXTH (Signed Extended Half Word)**
- Function SignExtend lower half word in a word of data
    - **Syntax SXTH <Rd>, <Rm>**
    - **Note Rd = SignExtend(Rm[15:0])**
- Rd and Rm are low registers.

Rm = 0xFF82

Sign-extended Result Rd: 0xFFFFFF82

- **UXTB (Unsigned Extended Byte)**
- Function Extend lowest byte in a word of data
    - **Syntax UXTB <Rd>, <Rm>**
    - **Note Rd = ZeroExtend(Rm[7:0])**
- Rd and Rm are low registers.

Rm = 0xFF82
Zero-extended Result Rd: 0x00000082

- **UXTH (Unsign Extended Half Word)**
- Function Extend lower half word in a word of data
    - **Syntax UXTH <Rd>, <Rm>**
    - **Note Rd =ZeroExtend(Rm[15:0])**
- Rd and Rm are low registers.

Rm = 0xFF82
Zero-extended Result Rd: 0x0000FF82

- For example, if R0 is 0x55AA8765,
- What is the result of R1 after executing following instruction?
- SXTB R1, R0
- SXTH R1, R0
- UXTB R1, R0
- UXTH R1, R0

- SXTB R1, R0 ; R1 = 0x00000065
- SXTH R1, R0 ; R1 = 0xFFFF8765
- UXTB R1, R0 ; R1 =0x00000065
- UXTH R1, R0 ; R1 = 0x00008765

Program Flow Instructions

- Branch(B-unconditional):
  - Branch to an address
  - **B <label>**
  - Example: **stop B stop**

Program Flow Instructions

- Branch(B<cond>-Conditional):
  - Depending of APSR, Branch to an address
  - **B<cond> <label>**
  - Example:
    - CMP R0, #0x1 ; compare R0 with 0x1
    - **BEQ process1 ; Branch to process1 if R0 equal to 1** Bequal
    - There 14 available <cond> suffixes

BEQ – equality check and sets Z=1
BNE – Not equal to check and sets Z=0
BMI – Minus/Negative check and sets N=1
BPL – Plus/Positive check and sets N=0

BGE – Greater than Equal to
BLE – Lesser than or Equal to
BHI – Higher than
BLS – Lesser than or Same
BGT – Signed Greater Than
BLT – Signed Lesser Than

Program Flow Instructions

- Branch(B<cond>-Conditional):

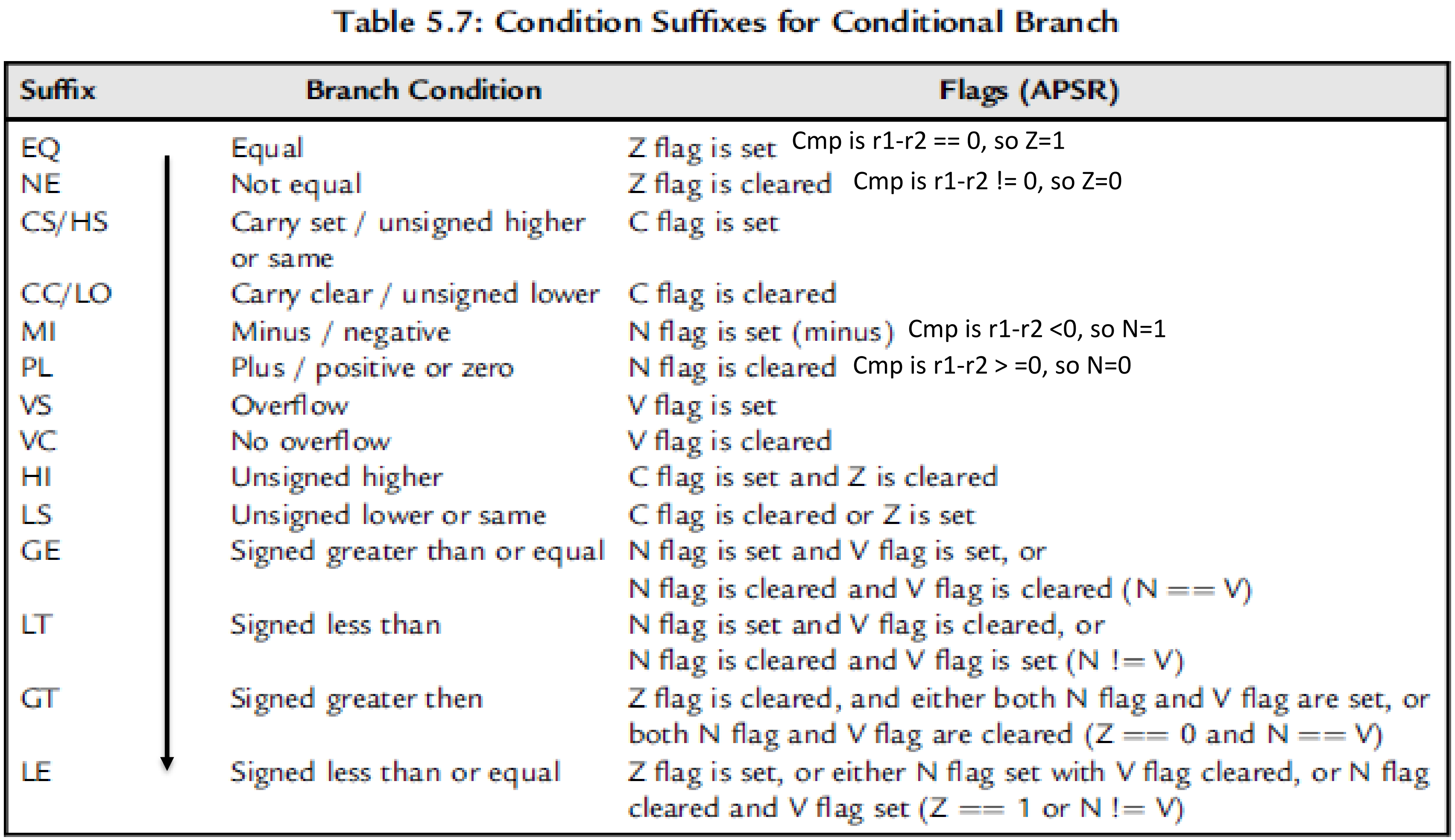


Table 5.7: Condition Suffixes for Conditional Branch

Suffix	Branch Condition	Flags (APSR)
EQ	Equal	Z flag is set Cmp is r1-r2 == 0, so Z=1
NE	Not equal	Z flag is cleared Cmp is r1-r2 != 0, so Z=0
CS/HS	Carry set / unsigned higher or same	C flag is set
CC/LO	Carry clear / unsigned lower	C flag is cleared
MI	Minus / negative	N flag is set (minus) Cmp is r1-r2 <0, so N=1
PL	Plus / positive or zero	N flag is cleared Cmp is r1-r2 > =0, so N=0
VS	Overflow	V flag is set
VC	No overflow	V flag is cleared
HI	Unsigned higher	C flag is set and Z is cleared
LS	Unsigned lower or same	C flag is cleared or Z is set
GE	Signed greater than or equal	N flag is set and V flag is set, or N flag is cleared and V flag is cleared (N == V)
LT	Signed less than	N flag is set and V flag is cleared, or N flag is cleared and V flag is set (N != V)
GT	Signed greater then	Z flag is cleared, and either both N flag and V flag are set, or both N flag and V flag are cleared (Z == 0 and N == V)
LE	Signed less than or equal	Z flag is set, or either N flag set with V flag cleared, or N flag cleared and V flag set (Z == 1 or N != V)

```
main:
    ; Call a subroutine named "subroutine"
    BL subroutine

    ; Rest of the main program subroutine:
    ; Subroutine code here

    ; Return from subroutine
    BX LR
```

Program Flow Instructions

- **Branch and Link(BL):**
  - Branch to an address and store return address to LR(link register).
  - **BL <label>**
  - Used for calling a subroutine or function
- **Branch and Exchange(BX):**
  - Branch to an address specified by link register and change the processor state depending on bit[0] of the register.
  - **BX <LR/Rm>**
- **Branch and Link with Exchange(BLX):**
  - Branch to an address specified by register and store return address to LR and change the processor state depending on bit[0] of the register.
  - **BLX <Rm>**

Memory Barrier Instructions

- Memory barrier instructions are often needed when the memory system is complex. In some cases, if the memory barrier instruction is not used, race conditions could occur and cause system failures.
- There are three memory barrier instructions that support on the Cortex-M0 processor:
- DMB - Data Memory Barrier (prev mem used before new)
- DSB - Data Synchronization Barrier (prev memory used before next instns)
- ISB – Instruction Sync Barrier (prev instns are executed before next)

Memory Barrier Instructions

- **DMB**
- Function :Data Memory Barrier
- Syntax :DMB
- Note :Ensures that all memory accesses are completed before new
- memory access is committed
- **DSB**
- Function : Data Synchronization Barrier
- Syntax : DSB
- Note : Ensures that all memory accesses are completed before the next
- instruction is executed
- **ISB**
- Function :Instruction Synchronization Barrier
- Syntax : ISB
- Note : Flushes the pipeline and ensures that all previous instructions are
- completed before executing new instructions

•The processor then switches to the exception handling routine labeled **SVC_Handler**.
•After handling the exception, the **BX LR** instruction returns from the exception and resumes execution of the main program.

Exception-Related Instructions

When the SVC #3 instruction is executed, it triggers an SVC exception with the parameter 3.
The processor then switches to the exception handling routine labeled SVC_Handler.
After handling the exception, the BX LR instruction returns from the exception and resumes execution of the main program.

- Function Supervisor call
- Syntax : SVC #<immed8>
- SVC <immed8> ;Trigger the SVC exception immed8
- SVC #3 ; SVC instruction, with parameter, equals 3.
- Alternative syntax without the "#" is also allowed. For example,
- SVC 3 ; this is the same as SVC #3.

- **CPS** (Change Processor State)
- Function : Change processor state: enable or disable interrupt
- Syntax :CPSIE I ; Interrupt Enable (Clearing PRIMASK)
- CPSID I ; Interrupt Disable (Setting PRIMASK)
- Note :PRIMASK only block external interrupts, SVC, PendSV, SysTick. But
- It does not block NMI and the hard fault handler

Sleep Mode Feature-Related Instructions

- The Cortex-M0 processor can enter sleep mode by executing the Wait-for-Interrupt (WFI) and Wait-for-Event (WFE) instructions
- **WFI – Turn On Sleep Mode**
- Function :Wait for Interrupt
- Syntax :WFI
- Note : Stops program execution until an interrupt arrives or until theprocessor enters a debug state.
- **WFE** is just like WFI, except that it can also be awakened by events.

- **SEV**
- Function: Send event to all processors in multiprocessing environment (including itself)
- Syntax :SEV
- Note: Set local event register and send out an event pulse to other microprocessor

Instruction Usage Examples

- **If-Then-Else**
- **Loop**

Instruction Usage Examples

- **If-Then-Else**

C code	Assembly language code

if (counter > 10) then
counter = 0
else
counter = counter + 1
	CMP R0, #10 ; compare to 10
	BLE incr_counter ; if less or equal, then branch
	MOVS R0, #0 ; counter = 0
	B counter_done ; branch to counter_done
	incr_counter
	ADDS R0, R0, #1 ; counter = counter+1
	counter_done

Instruction Usage Examples

- **Loop**

C code	Assembly language code

Total = 0;
for (i=0;i<5;i=i+1)
Total = Total + i;
	MOVS R0, #0 ; Total = 0
	MOVS R1, #0 ; i = 0
	loop
	    ADDS R0, R0, R1 ; Total = Total + i
	    ADDS R1, R1, #1 ; i=i + 1
	    CMP R1, #5 ; compare i to 5
	    BLT loop ; if less than then branch to loop

Different types of branch Instructions

Branch Type	Example
Normal branch	B label
Conditional branch	BEQ label
Branch and link	BL label
Branch and exchange state	BX LR
Branch and link with exchange state	BLX R4

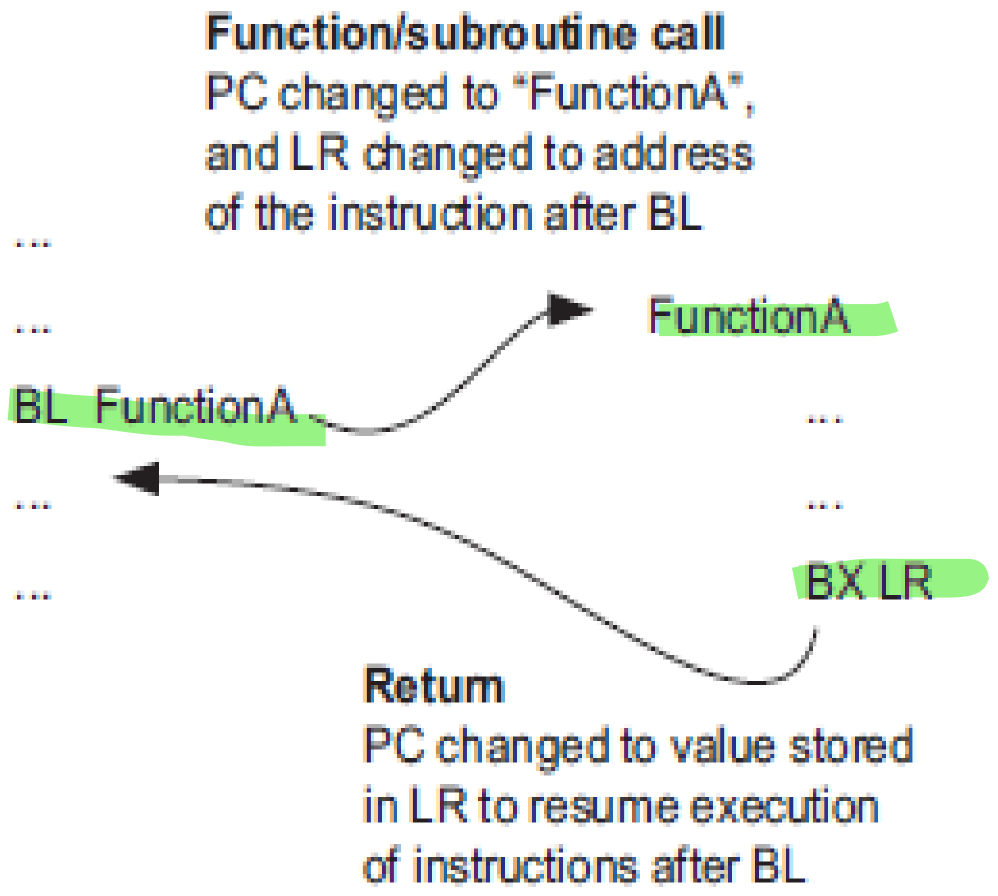
Conditional Branch instructions for value comparisons

Required Branch condition	Unsigned data	Signed data
If(R0==R1) then branch	BEQ label	BEQ label
If(R0!=R1) then branch	BNE label	BNE label
If(R0>R1) then branch	BHI label	BGT label
If(R0>=R1) then branch	BCS label/BHI label	BGE label
If(R0<R1) then branch	BCC label/BCO label	BLT label
If(R0<=R1) then branch	BLS label	BLE label

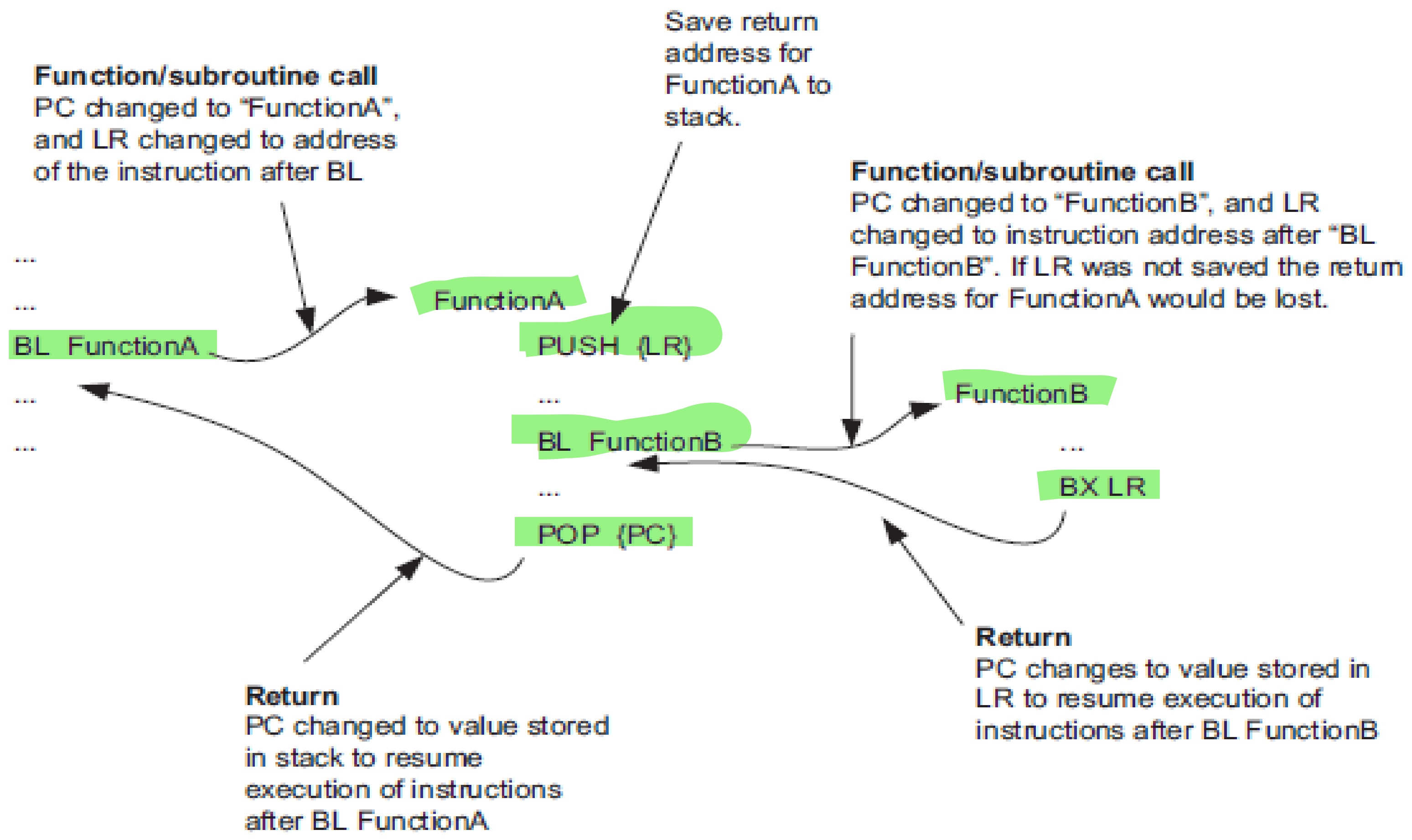
Conditional Branch instructions for overflow detection

Required Branch condition	Unsigned data	Signed data
If(overfolw(R0+R1)) then branch	BCS label	BVS label
If(no_overfolw(R0+R1)) then branch	BCC label	BVC label
If(overfolw(R0-R1)) then branch	BCC label	BVS label
If(no_overfolw(R0-R1)) then branch	BCS label	BCS label

Function Calls and Function Returns


Function/subroutine call
PC changed to "FunctionA", and LR changed to address of the instruction after BL
...
...
BL FunctionA
...
...
FunctionA
...
...
BX LR
Return
PC changed to value stored in LR to resume execution of instructions after BL

Nested function call


Function/subroutine call
PC changed to "FunctionA", and LR changed to address of the instruction after BL
Save return address for FunctionA to stack.
Function/subroutine call
PC changed to "FunctionB", and LR changed to instruction address after "BL FunctionB". If LR was not saved the return address for FunctionA would be lost.
...
...
BL FunctionA
...
...
FunctionA
PUSH {LR}
...
BL FunctionB
...
POP {PC}
FunctionB
...
BX LR
Return
PC changed to value stored in stack to resume execution of instructions after BL FunctionA
Return
PC changes to value stored in LR to resume execution of instructions after BL FunctionB

Branch Table

```
CMP R0, #3        ; Compare input to maximum valid choice
BHI default_case ; Branch to default case if higher than 3
MOVS R2, #4 ; Multiply branch table offset by 4
MULS R0, R2, R0 ; (size of each entry)
LDR R1,=BranchTable ; Get base address of branch table
LDR R2,[R1,R0] ; Get the actual branch destination
BX R2 ; Branches to the address stored in register R2, which is the computed branch destination
ALIGN 4 ; Alignment control. The table has
; to be word aligned to prevent unaligned read
BranchTable ; table of each destination addresses
DCD Dest0
DCD Dest1
DCD Dest2
DCD Dest3
default_case
. ; Instructions for default case
Dest0
. ; Instructions for case ‘0’
Dest1
. ; Instructions for case ‘1’
Dest2
. ; Instructions for case ‘2’
Dest3
. ; Instructions for case ‘3’
```

DCD will store the addresses of these destinations in consecutive memory locations starting from the address labeled BranchTable

CMP R0, #3: Compares the value in register R0 to the immediate value 3.

BHI default_case: Branches to default_case if the value in R0 is higher than 3. The BHI instruction means "Branch if Higher," which is true if the result of the previous comparison was greater.

MOVS R2, #4: Moves the immediate value 4 into register R2. This is likely to be used for multiplying the branch table offset by 4, assuming each entry in the table is 4 bytes.

MULS R0, R2, R0: Multiplies the value in R0 by the value in R2 and stores the result back in R0. This effectively scales the index for the branch table by 4, as mentioned in the comment.

LDR R1, =BranchTable: Loads the address of the branch table BranchTable into register R1.

LDR R2, [R1, R0]: Loads a word from memory at the address given by the sum of the value in R1 and the value in R0, and places it into register R2. This loads the actual branch destination address from the branch table.

BX R2: Branches to the address stored in register R2, which is the computed branch destination.

ALIGN 4: This instruction ensures that the following data (the branch table) is word-aligned, which means it starts at an address divisible by 4. This is necessary for efficient memory access.

Example of Using Memory Access Instruction

```
LDR r0,=0x00000000 ; Source address
LDR r1,=0x20000000 ; Destination address
LDR r2, =100 ; number of bytes to copy
copy_loop
LDRB r3, [r0] ; read 1 byte to r3
ADDS r0, r0, #1 ; increment source pointer to next byte
STRB r3, [r1] ; write 1 byte to r1
ADDS r1, r1, #1 ; increment destination pointer to nxtbyte
SUBS r2, r2, #1 ; decrement loop counter
BNE copy_loop ;
```

Implements a simple byte-by-byte(R0(0-100) to R1(0-100) copy operation, often referred to as "copy-pasting" or "copying."

```
; Load the source address into register r0
LDR r0, =0x00000000  ; r0 = Source address

; Load the destination address into register r1
LDR r1, =0x20000000  ; r1 = Destination address

; Load the number of bytes to copy into register r2
LDR r2, =100         ; r2 = Number of bytes to copy

; Start of the copy loop
copy_loop:

    ; Load a byte from the memory location pointed to by r0 and store it in r3
    LDRB r3, [r0]       ; r3 = Value at source address

    ; Increment the source pointer (r0) to point to the next byte
    ADDS r0, r0, #1     ; Increment source pointer

    ; Store the byte from r3 into the memory location pointed to by r1
    STRB r3, [r1]       ; Write value to destination address

    ; Increment the destination pointer (r1) to point to the next byte
    ADDS r1, r1, #1     ; Increment destination pointer

    ; Decrement the loop counter (r2)
    SUBS r2, r2, #1     ; Decrement loop counter

    ; Check if loop counter is not zero
    BNE copy_loop       ; If not zero, branch back to copy_loop

; End of the copy loop
```

Memory Access

```
            LDR r0,=0x00000000 ; Source address
            LDR r1,=0x20000000 ; Destination address
            LDR r2,=128 ; number of bytes to copy, also
copy_loop ; acts as loop counter
            LDMIA r0!,{r4-r7} ; Read 4 words(16 bytes) and increment r0
            STMIA r1!,{r4-r7} ; Store 4 words and increment r1
            LDMIA r0!,{r4-r7} ; Read 4 words and increment r0
            STMIA r1!,{r4-r7} ; Store 4 words and increment r1
            LDMIA r0!,{r4-r7} ; Read 4 words and increment r0
            STMIA r1!,{r4-r7} ; Store 4 words and increment r1
            LDMIA r0!,{r4-r7} ; Read 4 words and increment r0
            STMIA r1!,{r4-r7} ; Store 4 words and increment r1
            SUBS r2, r2, #64 ; Each time 64 bytes are copied
BNE copy_loop ; loop until all data copied
```

```
; Load the source address into register r0
LDR r0, =0x00000000   ; r0 = Source address

; Load the destination address into register r1
LDR r1, =0x20000000   ; r1 = Destination address

; Load the number of bytes to copy into register r2
LDR r2, =128          ; r2 = Number of bytes to copy, acts as loop counter

; Start of the copy loop
copy_loop:

  ; Load Multiple Increment After (LDMIA) from source address (r0) into registers r4-r7
  LDMIA r0!, {r4-r7}  ; Read 4 words and increment r0

  ; Store Multiple Increment After (STMIA) from registers r4-r7 to destination address (r1)
  STMIA r1!, {r4-r7}  ; Store 4 words and increment r1

  ; Repeat the LDMIA and STMIA operations three more times to process 16 bytes in total

  LDMIA r0!, {r4-r7}  ; Read 4 words and increment r0
  STMIA r1!, {r4-r7}  ; Store 4 words and increment r1

  LDMIA r0!, {r4-r7}  ; Read 4 words and increment r0
  STMIA r1!, {r4-r7}  ; Store 4 words and increment r1

  LDMIA r0!, {r4-r7}  ; Read 4 words and increment r0
  STMIA r1!, {r4-r7}  ; Store 4 words and increment r1

  ; Subtract 64 from the loop counter (r2)
  SUBS r2, r2, #64    ; Each time 64 bytes (16bytes*4) are copied

  ; Branch back to copy_loop if the loop counter is not zero
  BNE copy_loop       ; loop until all data copied
```

Usage of Push and pop

```
 PRESERVE8 ; Indicate the code here preserve
; 8 byte stack alignment
                    THUMB     ; Indicate THUMB code is used
                AREA    |.text|, CODE, READONLY

             EXPORT __main
; Start of CODE area
__main
              LDR r3,=0x20000100
   LDR r0,=0x20000050
   LDMIA r3!,{r1,r2} ; Read 4 words and increment r3

   mov SP,r0 ;Move stack pointer to r0
   PUSH {r1,r2} ;Push contents of r1, r2 to stack
  POP {r4,r5} ; Pop them to r4, r5
stop         B   stop
              END               ; End of file
```

```
PRESERVE8        ; Preserve 8-byte stack alignment

; Indicate the use of THUMB code
THUMB

; Define a code area named ".text" that is read-only
AREA |.text|, CODE, READONLY

; Export the symbol "__main"
EXPORT __main

; Start of the main code area
__main:
   ; Load the address 0x20000100 into register r3
   LDR r3, =0x20000100

   ; Load the address 0x20000050 into register r0
   LDR r0, =0x20000050

   ; Load multiple words (4 words) from memory pointed to by r3 into registers r1 and r2, and increment r3
   LDMIA r3!, {r1,r2}

   ; Move the value in r0 (0x20000050) to the stack pointer (SP)
   MOV SP, r0

   ; Push the values in registers r1 and r2 onto the stack
   PUSH {r1,r2}

   ; Pop the values from the stack into registers r4 and r5
   POP {r4,r5}

Stop B stop
   ; Infinite loop to halt the program
END
```

Endian Conversion

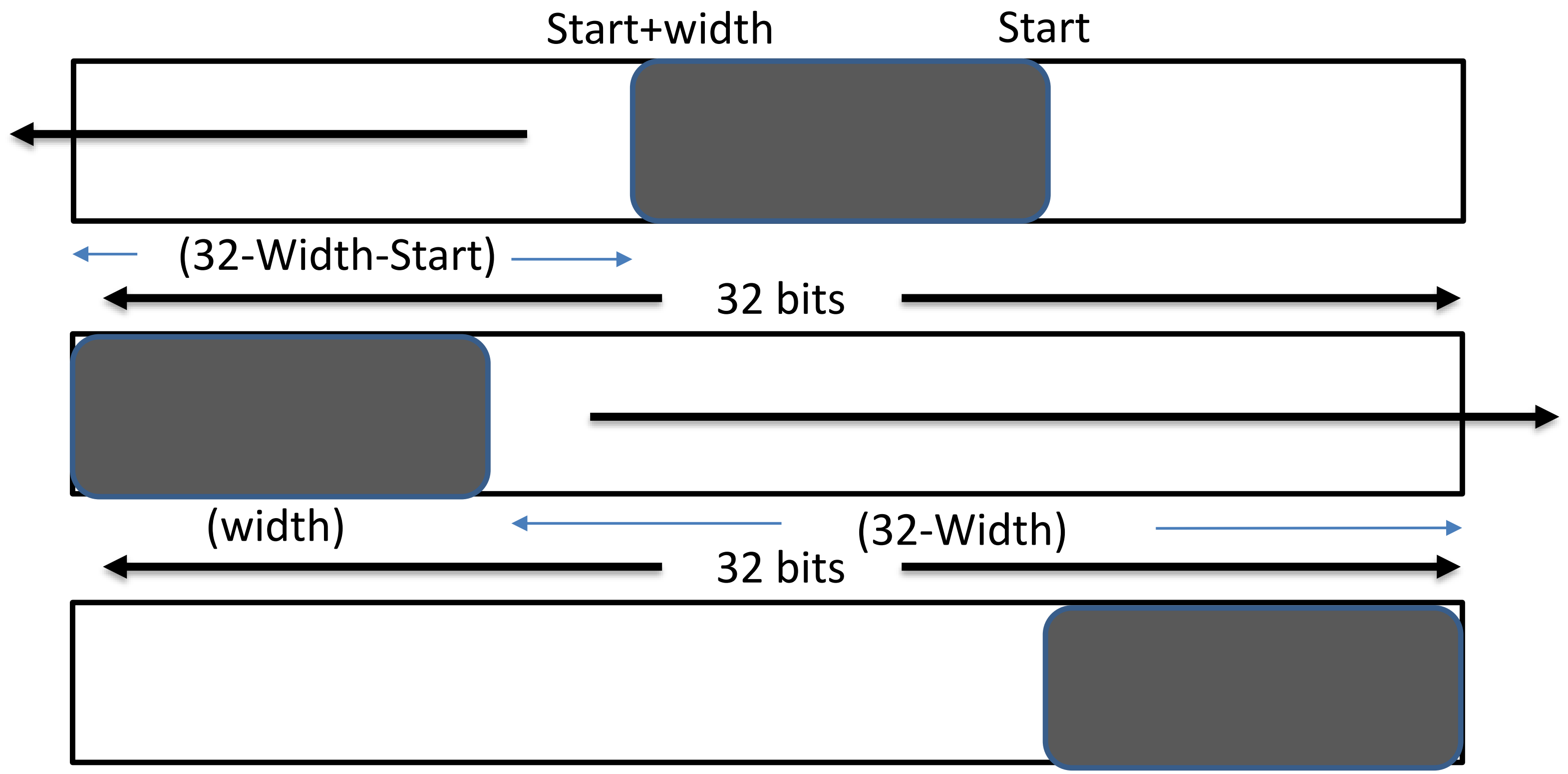
- **Conversion and Operation Instruction**
- Convert a little endian 32-bit value to big endian, or vice versa- REV (REV R1, 0x11223344 ; R1 = 0x44332211)
- Convert a little endian 16-bit unsigned value to big endian, or vice versa- REV16 (REV16 R1, 0x2233 ; R1 = 0x3322)
- Convert a little endian 16-bit signed value to big endian, or vice versa – REVSH (REVSH R1, 0xF0F1 ; (MSB) of 0xF1F0 is 1, R1 = 0xFFF1)

64-Bit Substraction

- LDR r0, =0x00000001 ; X_Low(X = 0x00000001 00000001)
- LDR r1,=0x00000001 ; X_High
- LDR r2, =0x00000003 ; Y_Low(Y = 0x00000000 00000003)
- LDR r3,=0x00000000 ; Y_High
- SUBS r0,r0,r2 ; lower 32-bit, r0 = X_Low - Y_Low
- SBCS r1,r1,r3 ; upper 32-bit, r1 = X_High - Y_High - borrow ; Subtract the carry flag from the upper 32 bits of X (r1) if the previous subtraction resulted in borrow

To extract an 8-bit-width bit field from bit 4 to bit 11

- LSLS R0, R0, #(32-Width-Start) ; Remove un-needed top bits
- LSRS R0, R0, #(32-Width) ; Align required bits to bit 0



Extract those width long bits

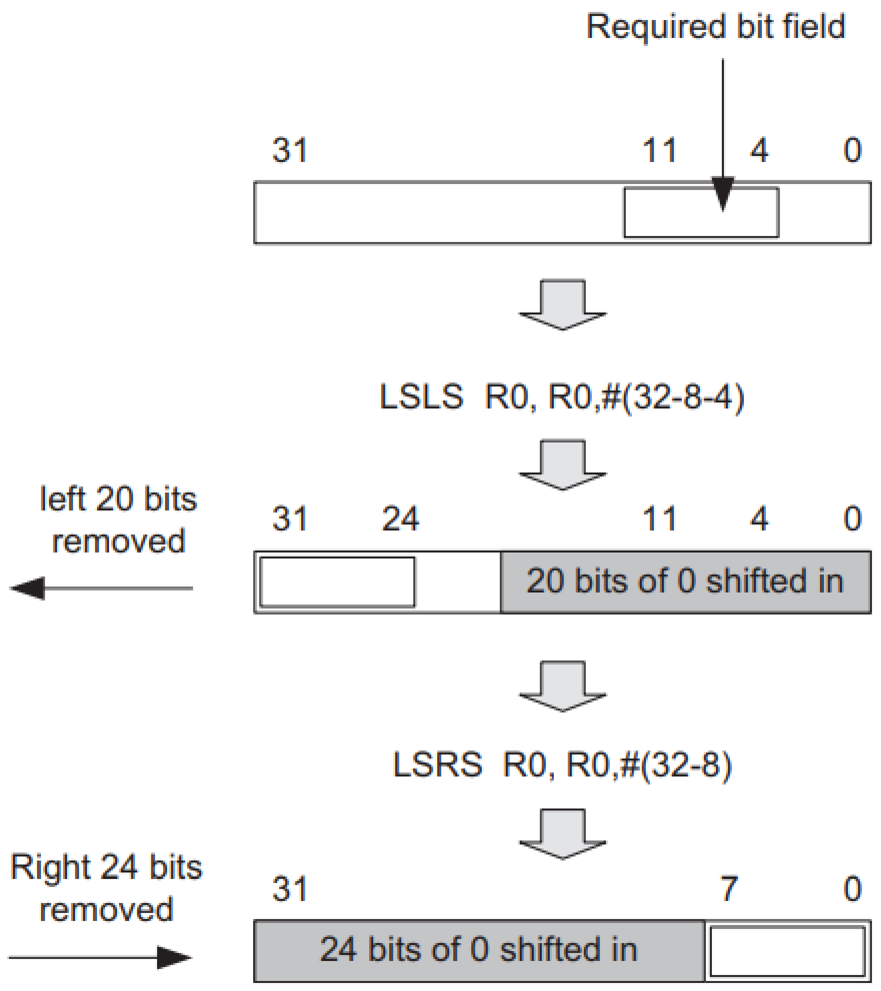


Figure 6.9:
Bit field extract operation.

To clear an 8-bit-width bit field from bit 4 to bit 11

- RORS R0, R0, #S ; Shift unneeded bit to bit 0
- LSRS R0, R0, #W ; Align required bits to bit 0
- RORS R0, R0, #(32-W-S) ; store value to original position

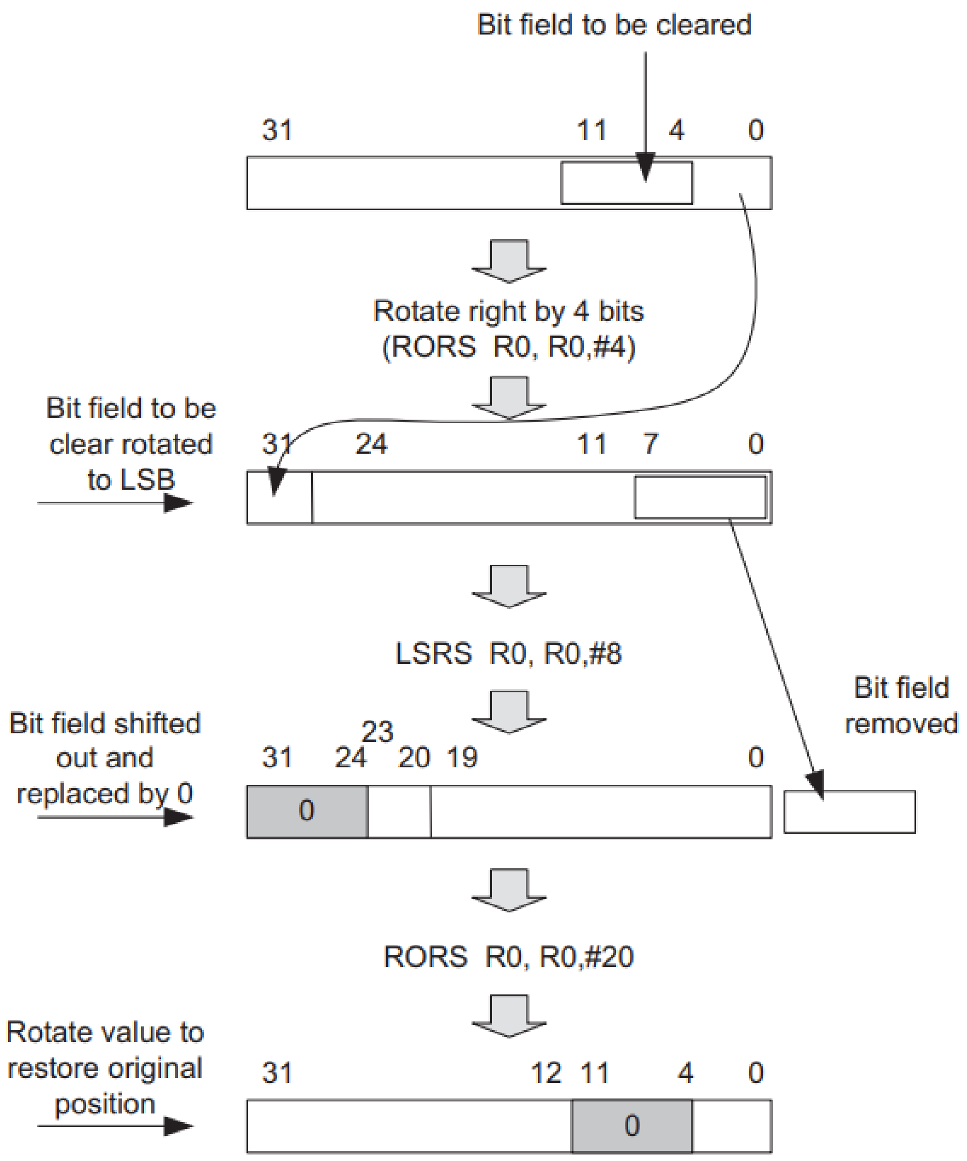


Figure 6.10:
Bit field clear operation.

SWAP

```
PRESERVE8           ; Preserve 8-byte stack alignment
THUMB               ; Indicate THUMB code is used
AREA |.text|, CODE, READONLY

EXPORT main         ; Export the symbol "main"

main:
    LDR R0, =0xF631024C    ; Load immediate value 0xF631024C into R0
    LDR R1, =0x17539ABD    ; Load immediate value 0x17539ABD into R1

    EORS R0, R0, R1        ; Perform R0 XOR R1 and store the result in R0
    EORS R1, R0, R1        ; Perform R1 XOR R0 and store the result in R1
    EORS R0, R0, R1        ; Perform R0 XOR R1 and store the result in R0

stop:
    B stop              ; Infinite loop to halt program execution

END                 ; End of the code
```

TST(Test)

```
PRESERVE8 ; Indicate the code here preserve
          ; 8 byte stack alignment
            THUMB     ; Indicate THUMB code is used
          AREA    |.text|, CODE, READONLY

        EXPORT __main
          ; Start of CODE area
__main
                    LDR r0,=0xF0000000;
                    LDR r2,=0xF0000000;
                    TST r0,r2;
                     MRS r3,XPSR;updating only N and Z flags not C
                    LDR r0,=0x70000000;
                    LDR r2,=0x70000000;
                    TST r0,r2;
                     MRS r4,XPSR;updating only N and Z flags not C
stop B stop
  END
```

```
PRESERVE8           ; Preserve 8-byte stack alignment
THUMB               ; Indicate THUMB code is used
AREA |.text|, CODE, READONLY

EXPORT __main       ; Export the symbol "__main"

__main:
   LDR r0, =0xF0000000  ; Load immediate value 0xF0000000 into r0
   LDR r2, =0xF0000000  ; Load immediate value 0xF0000000 into r2

   TST r0, r2          ; Perform bitwise AND operation between r0 and r2, and update flags
                  ; The result is discarded, only the flags (N, Z, C, and V) are updated

   MRS r3, XPSR        ; Move the contents of the Program Status Register (XPSR) into r3
                  ; This instruction retrieves the values of the flags after the TST operation
                  ; However, only the N (Negative) and Z (Zero) flags are updated in this case, not the C (Carry) flag

   LDR r0, =0x70000000  ; Load immediate value 0x70000000 into r0
   LDR r2, =0x70000000  ; Load immediate value 0x70000000 into r2

   TST r0, r2          ; Perform bitwise AND operation between r0 and r2, and update flags
                  ; The result is discarded, only the flags (N, Z, C, and V) are updated

   MRS r4, XPSR        ; Move the contents of the Program Status Register (XPSR) into r4
                  ; This instruction retrieves the values of the flags after the TST operation
                  ; However, only the N (Negative) and Z (Zero) flags are updated in this case, not the C (Carry) flag

stop:
   B stop             ; Infinite loop to halt program execution

END                 ; End of the code
```

Use of BIC
Bitwise AND NOT (complement)

```
PRESERVE8 ; Indicate the code here preserve
          ; 8 byte stack alignment
             THUMB    ; Indicate THUMB code is used
           AREA   |.text|, CODE, READONLY

         EXPORT __main
          ; Start of CODE area
__main

          LDR r1,=0x00000080;
          LDR r2,=0x80000000;
          BICS r2,r2,r1
stop B stop
   END
```

Use of ASRS

ASRS (Arithmetic Shift Right with immediate)

```
PRESERVE8 ; Indicate the code here preserve
          ; 8 byte stack alignment
             THUMB    ; Indicate THUMB code is used
          AREA   |.text|, CODE, READONLY

         EXPORT __main
          ; Start of CODE area
__main

          LDR r2,=0x00000080;
          ASRS r0,r2,#04
          LDR r2,=0x80000000;
          ASRS r0,r2,#04
stop B stop
   END
```

Use of CMN

CMN (Compare Negative)

```
PRESERVE8 ; Indicate the code here preserve
          ; 8 byte stack alignment
            THUMB     ; Indicate THUMB code is used
          AREA    |.text|, CODE, READONLY

        EXPORT __main
          ; Start of CODE area
__main

          LDR r1,=0x00000080;
          LDR r2,=0x80000000;
          CMN  r2,r1
       MRS r3,XPSR;
stop B stop
   END
```