CHAPTER 6

Programming in Assembly
Part 2: Data Manipulation

Loading Constants

MOV r_d,*constant*

- Works for 0 - 255 and “some” others

MVN r_d,*constant ;* r_d *<- ~constant*

- Effectively doubles the # of constants
- Assembler converts MOV w/neg. const to MVN

LDR r_d,=*constant*

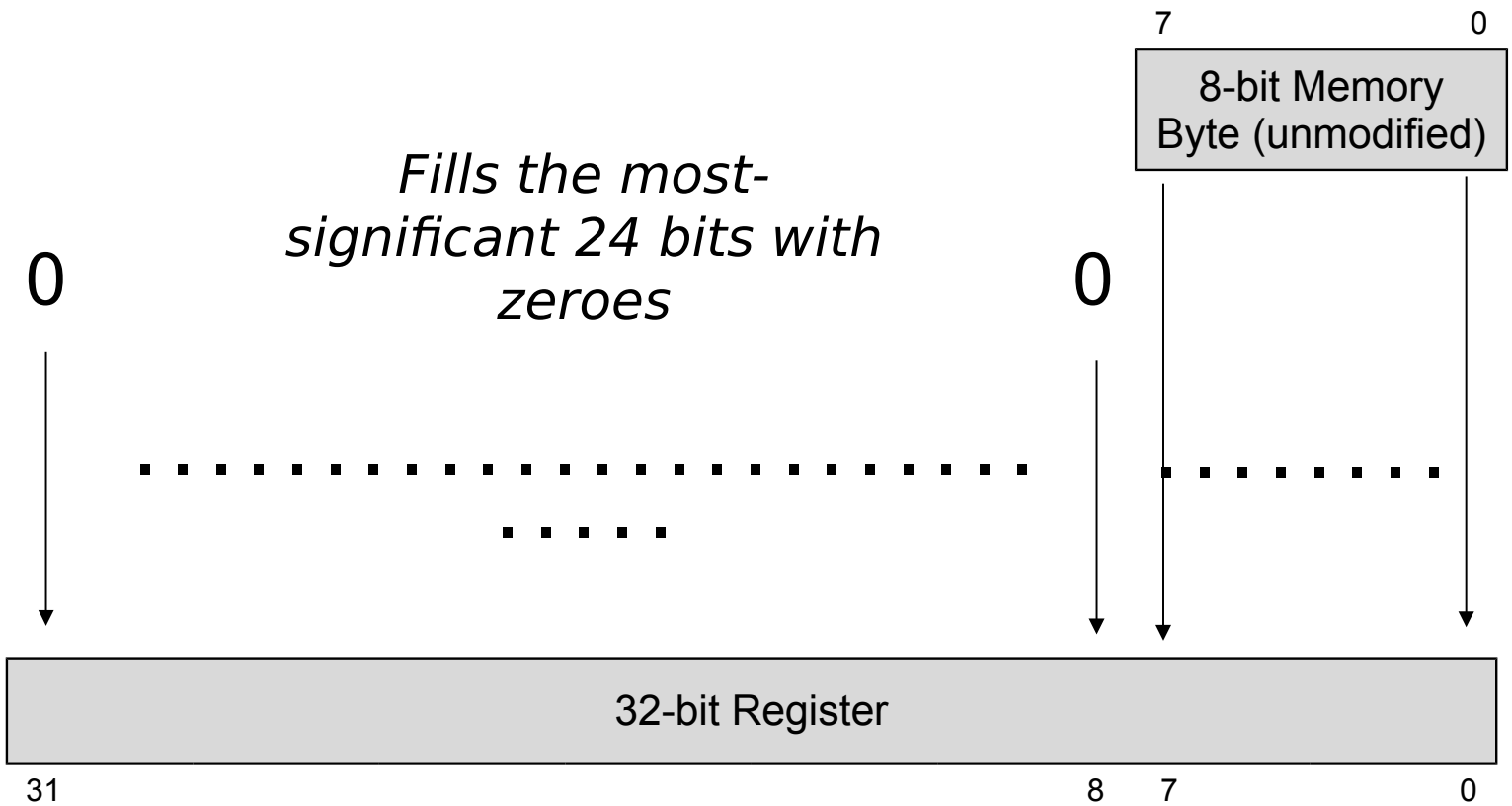
- An assembler pseudo-op, not an instruction
- Converted to MOV or MVN if possible
- Else converts to LDR r, [pc, #imm]

Load (from memory) Instructions

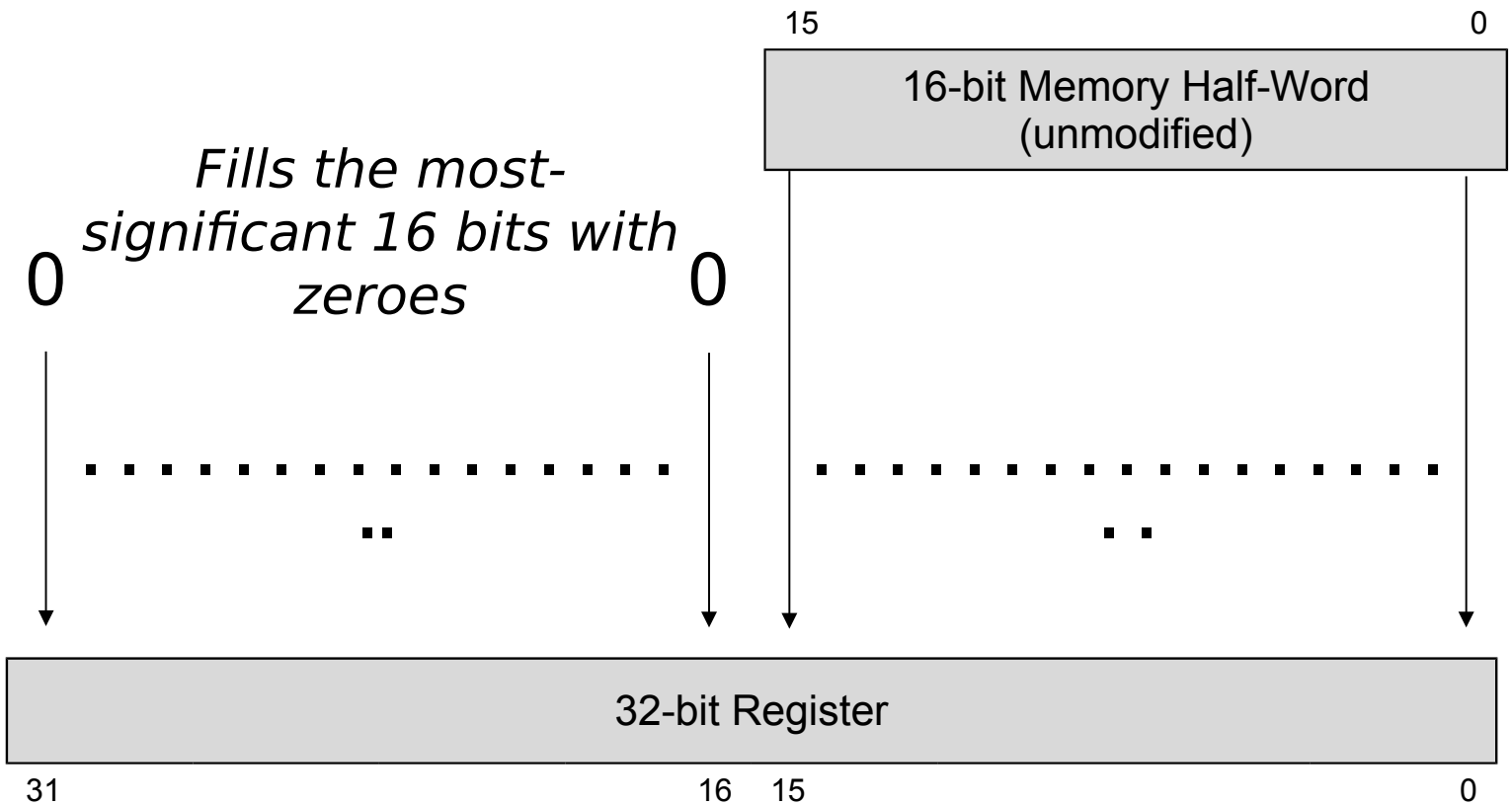
Load/Store Memory	*Operation*	*Notes*
LDR R_d,<mem>	R_d = mem_{32}[address]	
LDRB R_d,<mem>	R_d = mem_8[address]	Zero fills
LDRH R_d,<mem>	R_d = mem_{16}[address]	Zero fills
LDRSB R_d,<mem>	R_d = mem_8[address]	Sign extends
LDRSH R_d,<mem>	R_d = mem_{16}[address]	Sign extends
LDRD R_t,R_{t2},<mem>	R_{t2}.R_t = mem_{64}[address]	Addr. Offset must be imm.

These instructions never affect flags in xPSR!

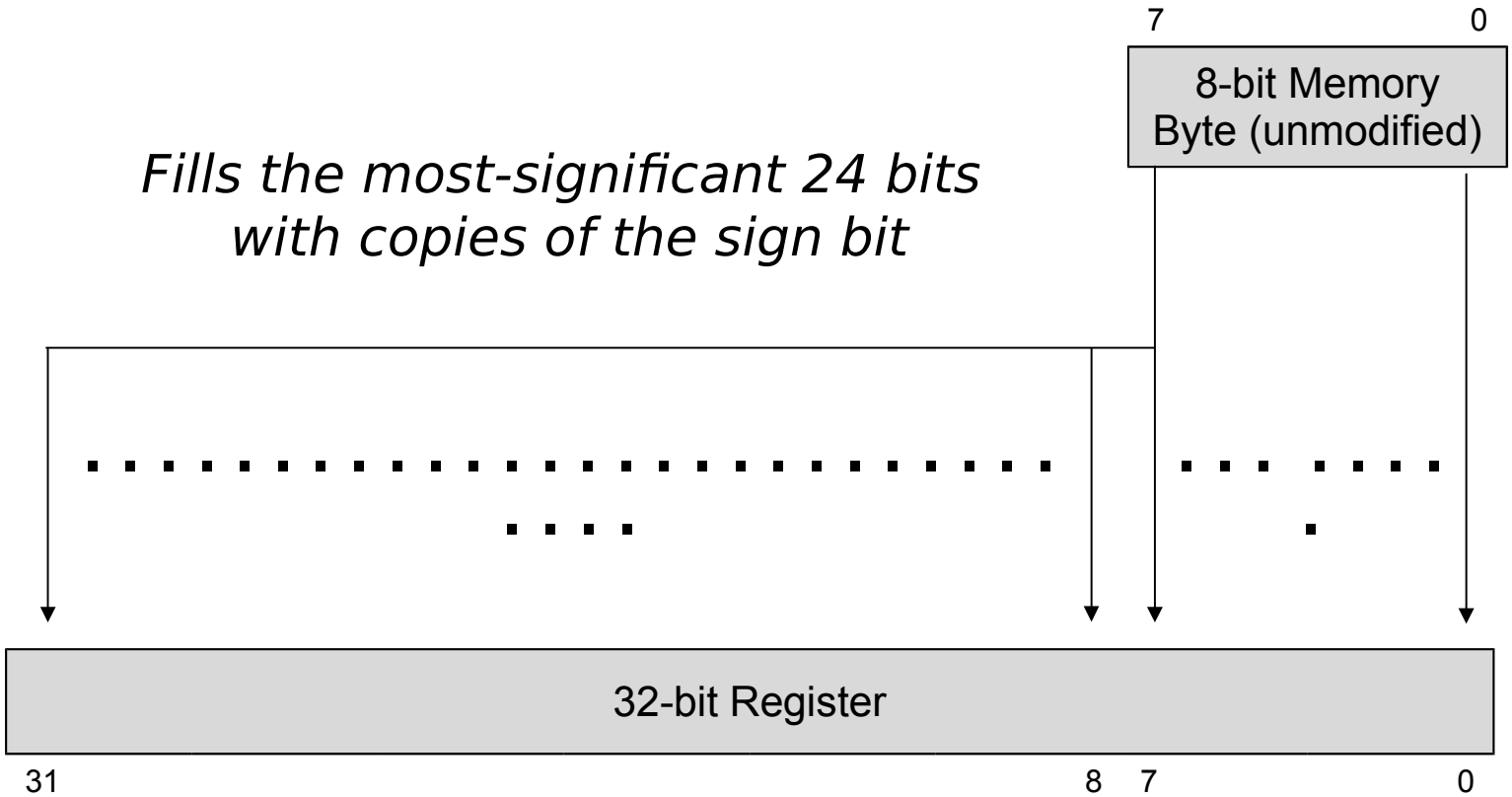
LDRB: Load Register with (unsigned) Byte



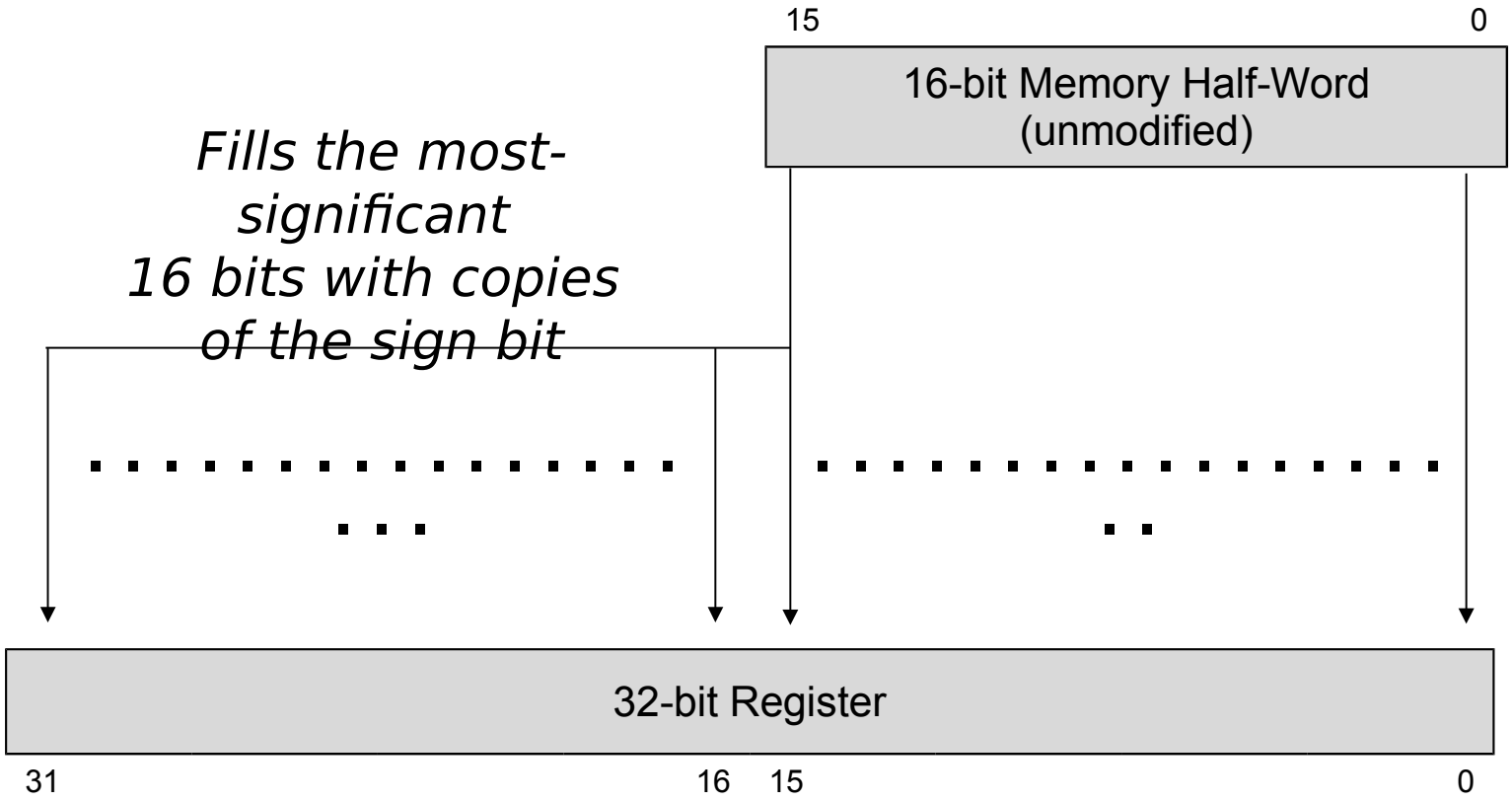
LDRH: Load Register with (unsigned) Half-Word



LDRSB: Load Register with Signed Byte



LDRSH: Load Register with Signed Half-Word

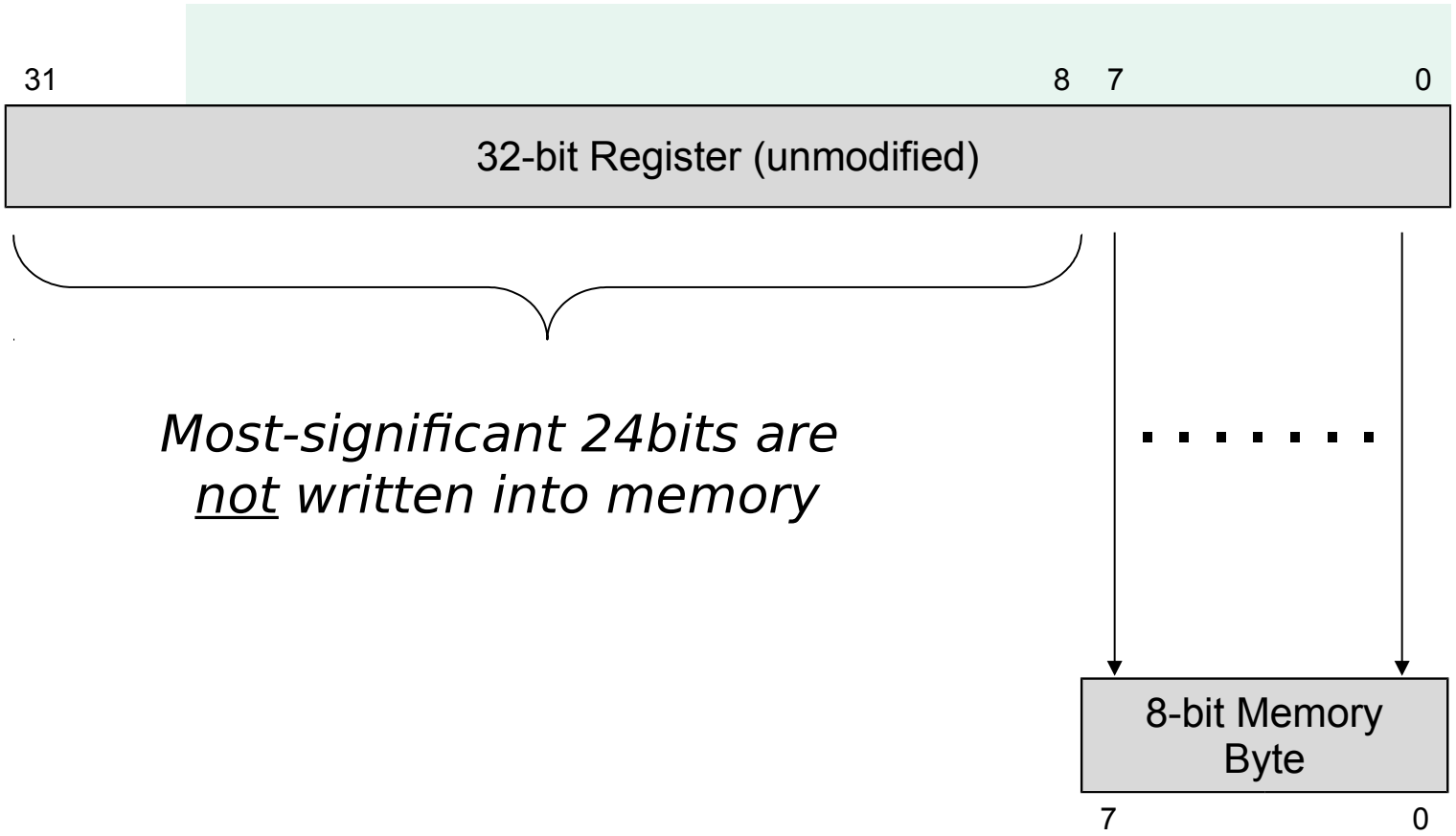


Store (to memory) Instructions

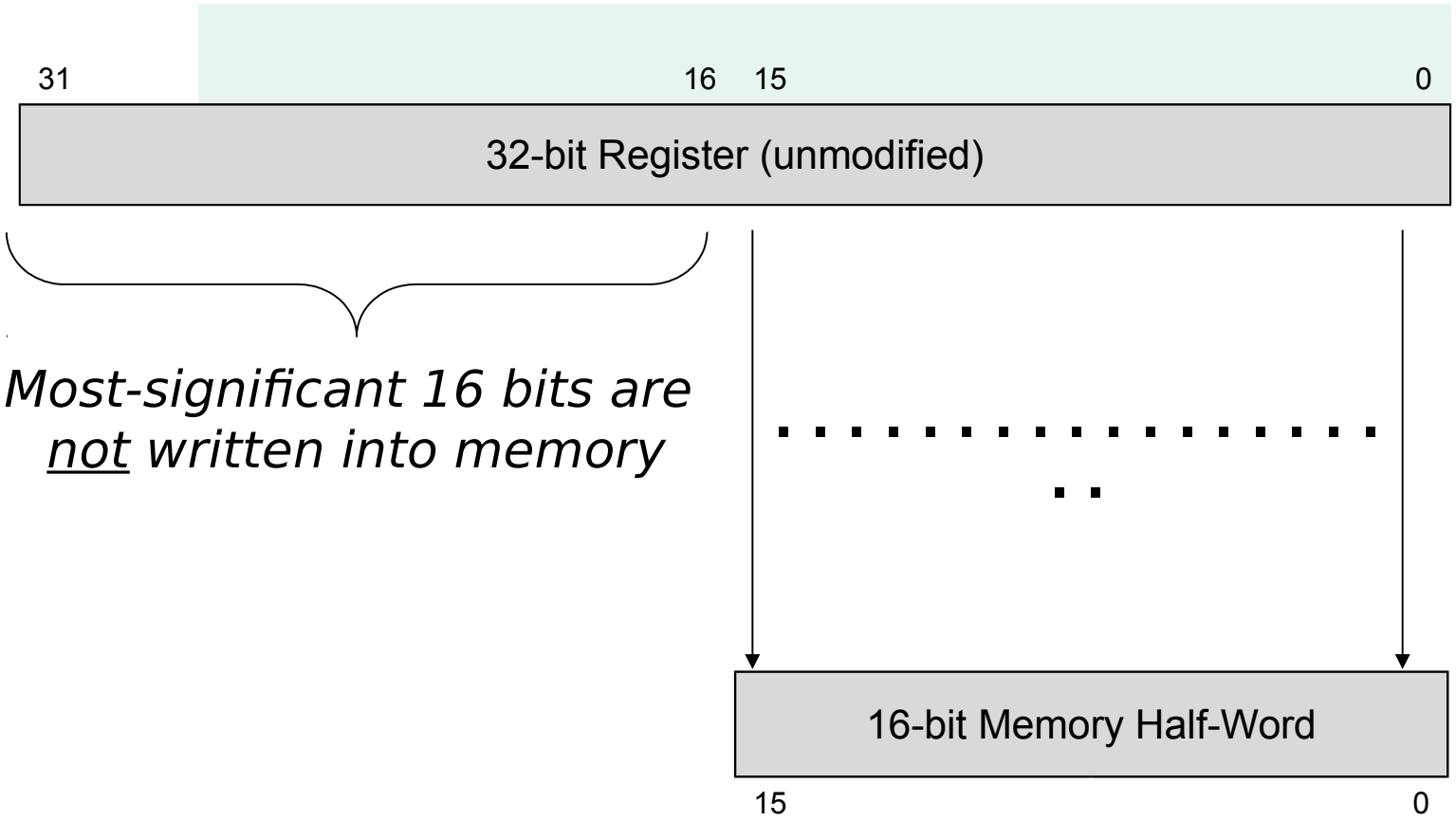
Load/Store Memory	*Operation*	*Notes*
STR R_d,<mem>	R_d ▭ mem_{32}[address]	
STRB R_d,<mem>	R_d ▭ mem_8[address]	
STRH R_d,<mem>	R_d ▭ mem_{16}[address]	
STRD R_t,R_{t2},<mem>	R_{t2}.R_t ▭ mem_{64}[address]	Addr. Offset must be imm.

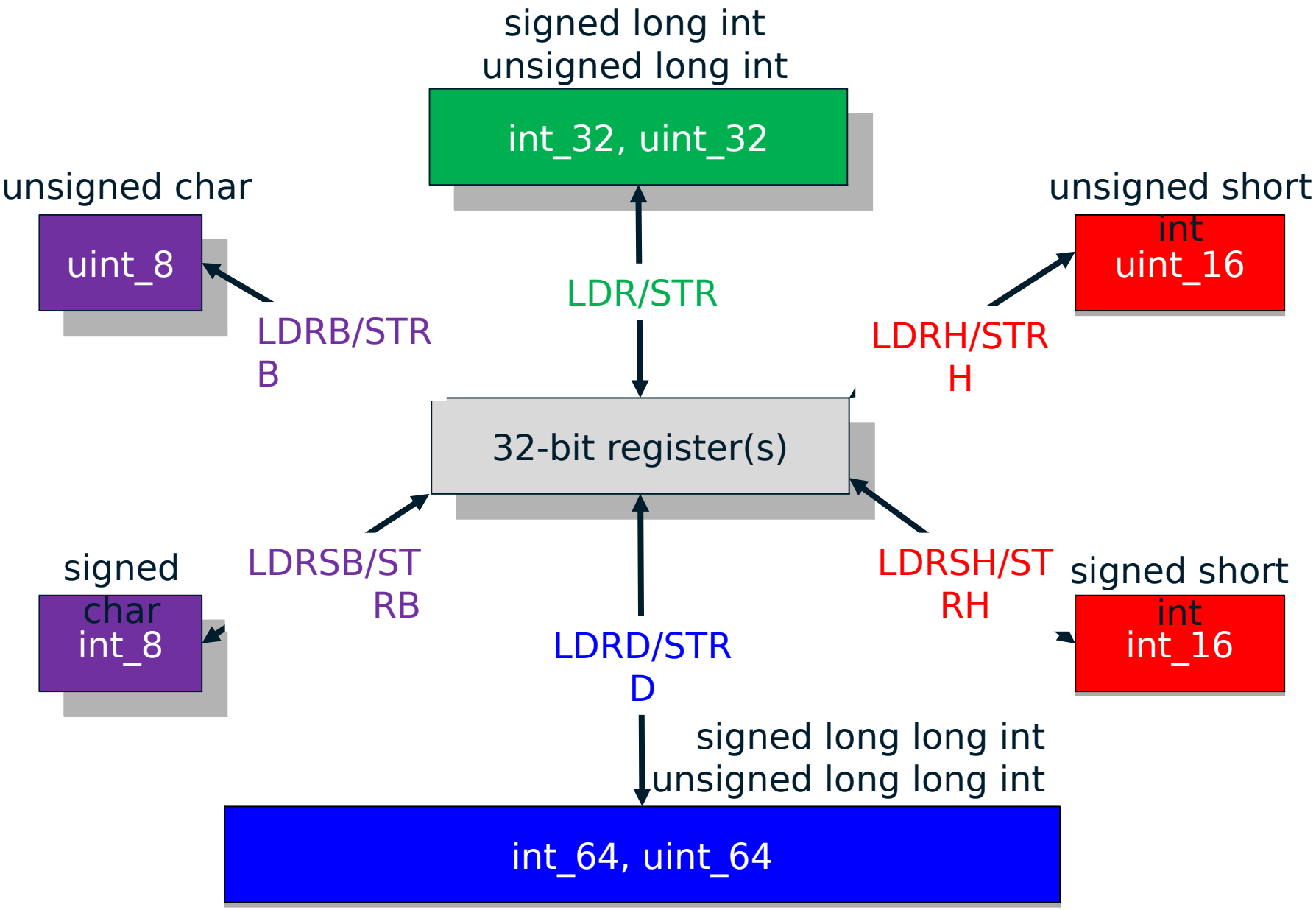
These instructions never affect flags in xPSR!

STRB: Store Register to Byte
(signed or unsigned)



STRH: Store Register to Half-Word
(signed or unsigned)




signed long int
unsigned long int
int_32, uint_32
unsigned char
uint_8
unsigned short int
uint_16
LDR/STR
LDRB/STRB
LDRH/STRH
32-bit register(s)
LDRSB/STRB
LDRSH/STRH
signed char
int_8
signed short int
int_16
LDRD/STRD
signed long long int
unsigned long long int
int_64, uint_64

variable = constant

Always load the constant into a 32-bit register;

Choose a STR instruction to match destination size.

```
int32_t  x32 ;      =    LDR    R0,=0 ; load constant zero
x32 = 0 ;      STR   R0,x32    ; store all 32 bits

int16_t  x16 ;      =    LDR    R0,=0 ; load constant zero
x16 = 0 ;      STRH  R0,x16    ; store lower 16 bits

int8_t  x8 ;      =    LDR    R0,=0 ; load constant zero
x8 = 0 ;       STRB  R0,x8 ; store lower 8 bits

int64_t  x64 ;  =    LDR    R0,=0 ; load constant zero
x64 = 0 ;      STRD  R0,R0,x64
```

variable = variable (same size)

Match both LDR & STR to destination size

```
int32_t  a32, b32 ;  =  LDR   R0,b32   ; load 32 bit word
a32 = b32 ;             STR   R0,a32   ; store all 32 bits

int16_t  a16, b16 ;  =  LDRH  R0,b16   ; only need 16 bits
a16 = b16 ;             STRH  R0,a16   ; store lower 16 bits

int8_t  a8, b8 ;  =  LDRB  R0,b8 ; only need 8 bits
a8 = b8 ;            STRB  R0,a8 ; store lower 8 bits

int64_t  a64, b64 ;  =  LDRD  R0,R1,b64
a64 = b64 ;             STRD  R0,R1,a64
```

Demotion: small variable = large variable

Match both LDR & STR to destination size (Minimizes potential memory alignment penalties)

```
int32_t  x32 ;                  int16_t  x16 ;     int8_t  x8 ;

x8 = (int8_t) x32 ;  =          LDRB   R0,x32    ; only need 8 bits
                                STRB   R0,x8 ; store lower 8 bits

x16 = (int16_t) x32 ;           =    LDRH  R0,x32    ; only need 16 bits
                                STRH   R0,x16    ; store lower 16 bits

x8 = (int8_t) x16 ;  =          LDRB   R0,x16    ; only need 8 bits
                                STRB   R0,x8 ; store lower 8 bits
```

Promotion: large unsigned = small unsigned

Match LDR to source and zero-fill; Match STR to destination

```
uint64_t  u64 ;
uint16_t  u16 ;

u32 = (uint32_t) u8 ;
bits

u32 = (uint32_t) u16 ;
bits

u16 = (uint16_t) u8 ;
bits

bits
```

```
uint32_t  u32 ;
uint8_t  u8 ;

=   LDRB  R0,u8   ; zero-fill to 32

STR    R0,u32 ; save all 32 bits

=   LDRH  R0,u16 ; zero-fill to 32

STR    R0,u32 ; save all 32 bits

=   LDRB  R0,u8   ; zero-fill to 32

STRH  R0,u16 ; save lower 16
```

Promotion: large signed = small signed

Match LDR to source and sign-extend; Match STR to destination

```
int64_t  u64 ;                int32_t  u32 ;
int16_t  u16 ;                int8_t  u8 ;

s32 = (int32_t) s8 ;          =   LDRSB   R0,s8   ; sign-ext.
8 = 32 bits
                                  STR   R0,s32 ; save all
32 bits

s32 = (int32_t) s16 ;         =   LDRSH   R0,s16 ; sign-ext.
16 = 32 bits
                                  STR   R0,s32 ; save all
32 bits

s16 = (int16_t) s8 ;          =   LDRSB   R0,s8   ; sign-ext.
8 = 32 bits
                                  STRH  R0,s16 ; save
```

Two Common Mistakes

```
y = x + 5 ;     LDR    R0,x+5
        STR    R0,y
```

This addition occurs during assembly – NOT when you run the program. Remember: The value of the symbol "x" in assembly is its address, which is a constant!

```
        LDR    R0,x
        ADD    R0,R0,#5
        STR    R0,y
```

```
x = y ;     LDR    R0,x
        LDR    R1,y
        MOV    R0,R1
```

This does NOT move x into R0 – it copies the contents of x into R0.

```
        LDR    R0,y
        STR    R0,x
```

Memory Access Modes

MOST IMPORTANT!

- **Offset Addressing Mode**
- **Pre-Indexed Addressing Mode**
- **Post-Indexed Addressing Mode**

Used only w/Load/Store Instructions.

Restrictions on LDRD & STRD

Offset Addressing

(The Most Important Memory Access Mode!)

Syntax	***Effective Address***	***Example***
<label>	PC + imm	result
[<R_n>,#imm]	R_n + imm	[r5,#100]
[<R_n>,<R_m>]	R_n + R_m	[r4,r5]
[<R_n>,<R_m>,LSL #<imm>]	R_n + (R_m << imm)	[r4,r5,LSL #3]

Special case of R_n + imm, where R_n = pc

Not available with LDRD or STRD

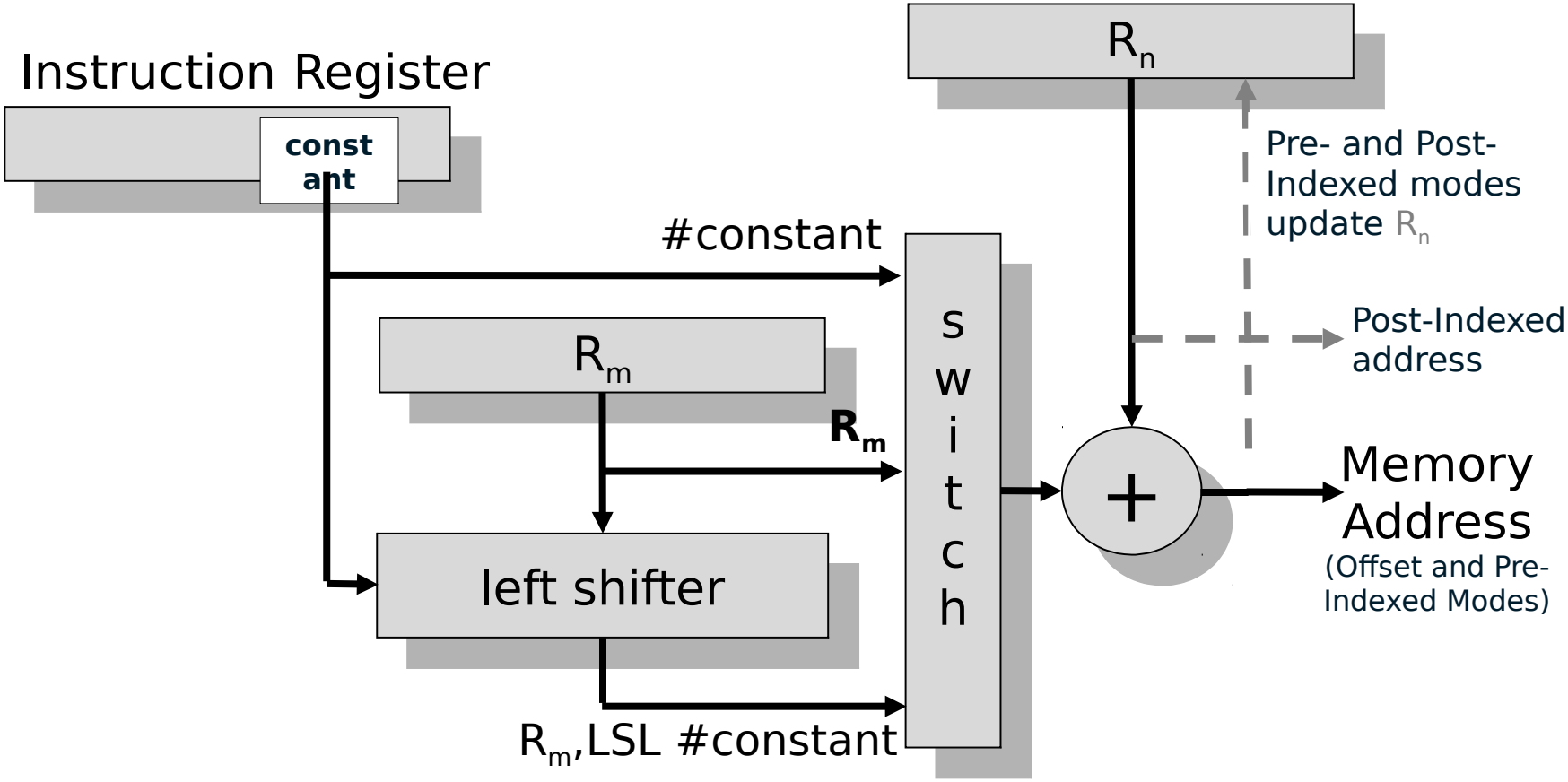
Common Mistake

```
int32_t x, y ;

y = x+5 ;    LDR   R0,x        ; syntax doesn't support the
             LDR   R1,[R0+5]   ; use of arithmetic operators
             STR   R1,y        ; inside the square brackets.
```

```
             LDR   R0,x        ; copy contents of x into R0
             ADD   R0,R0,#5    ; R0 = R0 + 5
             STR   R0,y        ; copy contents of R0 into y
```

Address Calculation



Using Offset Addressing

```
int32_t *p ;        ⊟  LDR   R0,=0
...                    LDR   R1,p
*p = 0 ;               STR   R0,[R1]

int32_t  *p ;       ⊟  LDR   R0,=0
...                    LDR   R1,p
*(p + 1) = 0 ;         STR   R0,[R1,#4]

int32_t  *p, k ;    ⊟  LDR   R0,=0
...                    LDR   R1,p
*(p + k) = 0 ;         LDR   R2,k
                       STR   R0,[R1,R2,LSL #2]
```

ADR versus LDR

```
LDR  R0,operand  ; LDR copies the contents of a memory
                 ; operand (i.e., a variable) into a register.
```

C:	Assembly:
int32_t result, answer ;	LDR R0,result
answer = result ;	STR R0,answer

```
ADR  R0,operand ; ADR copies the address of a memory
                ; operand (i.e., a constant) into a register.
```

C:	Assembly:
int8_t data, p ;	ADR R0,data
p = &data ;	STR R0,p

Using Offset Addressing (Cont'd)

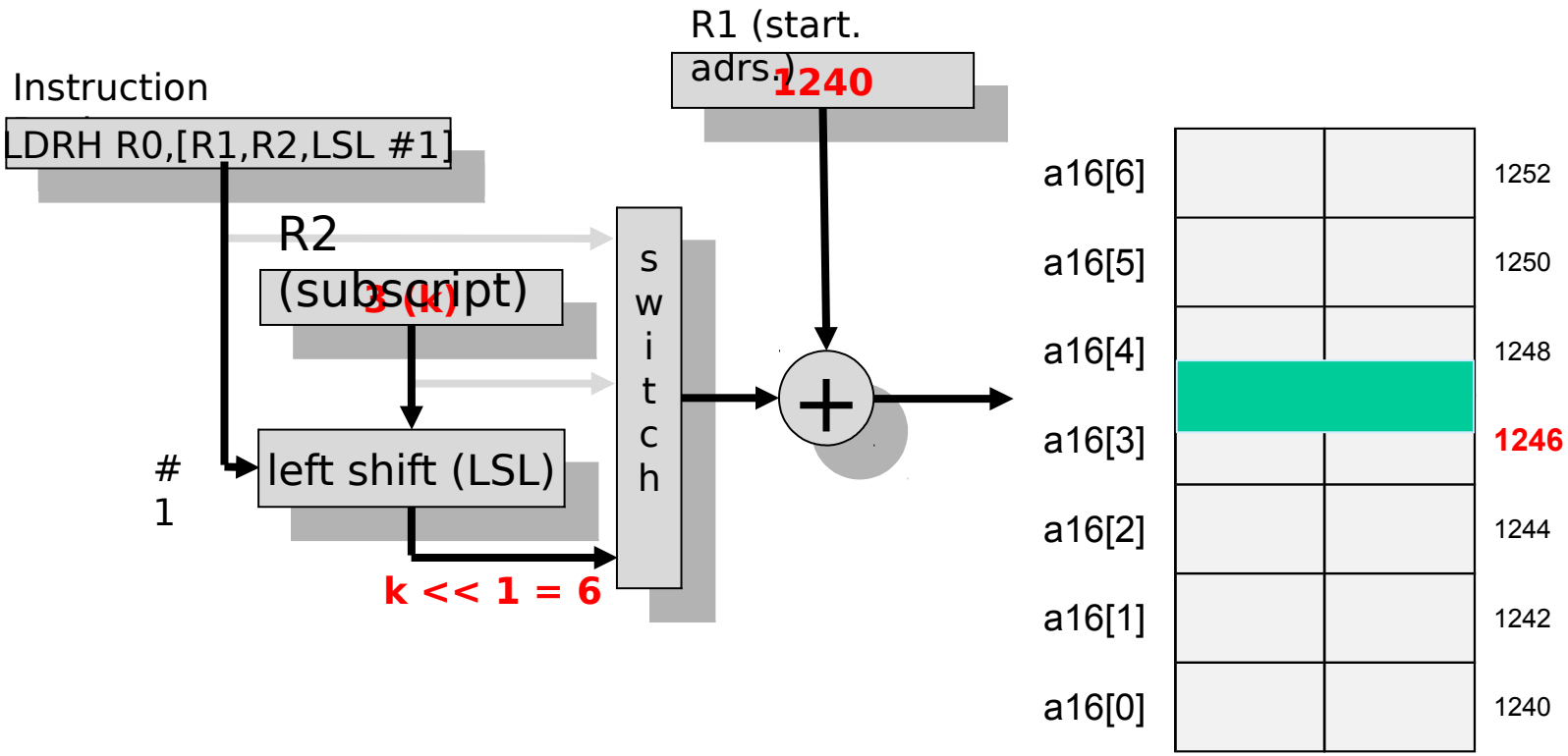
```
int8_t  a8[100] ;  ⊟  LDR   R0,=0
int32_t  k32 ;        ADR   R1,a8
…                  LDR   R2,k32
a8[k32] = 0 ;         STRB  R0,[R1,R2]

int16_t  a16[100] ;  ⊟  LDR   R0,=0
…                  ADR   R1,a16
a16[5] = 0 ;          STRH  R0,[R1,#10]

int32_t  a32[100] ;  ⊟  LDR   R0,=0
int32_t  k32 ;        ADR   R1,a32
…                  LDR   R2,k32
a32[k32] = 0 ;        STR   R0,[R1,R2,LSL #2]
```

Subscripting: x16 = a16[k]

```
ADR    R1,a16          ; R1 = start. adr. of array
LDR    R2,k            ; R2 = subscript (k=3)
LDRH   R0,[R1,R2,LSL #1] ; R0 = a16[k]
STRH   R0,x16          ; x16 = R0
```



Pre-Indexed Addressing

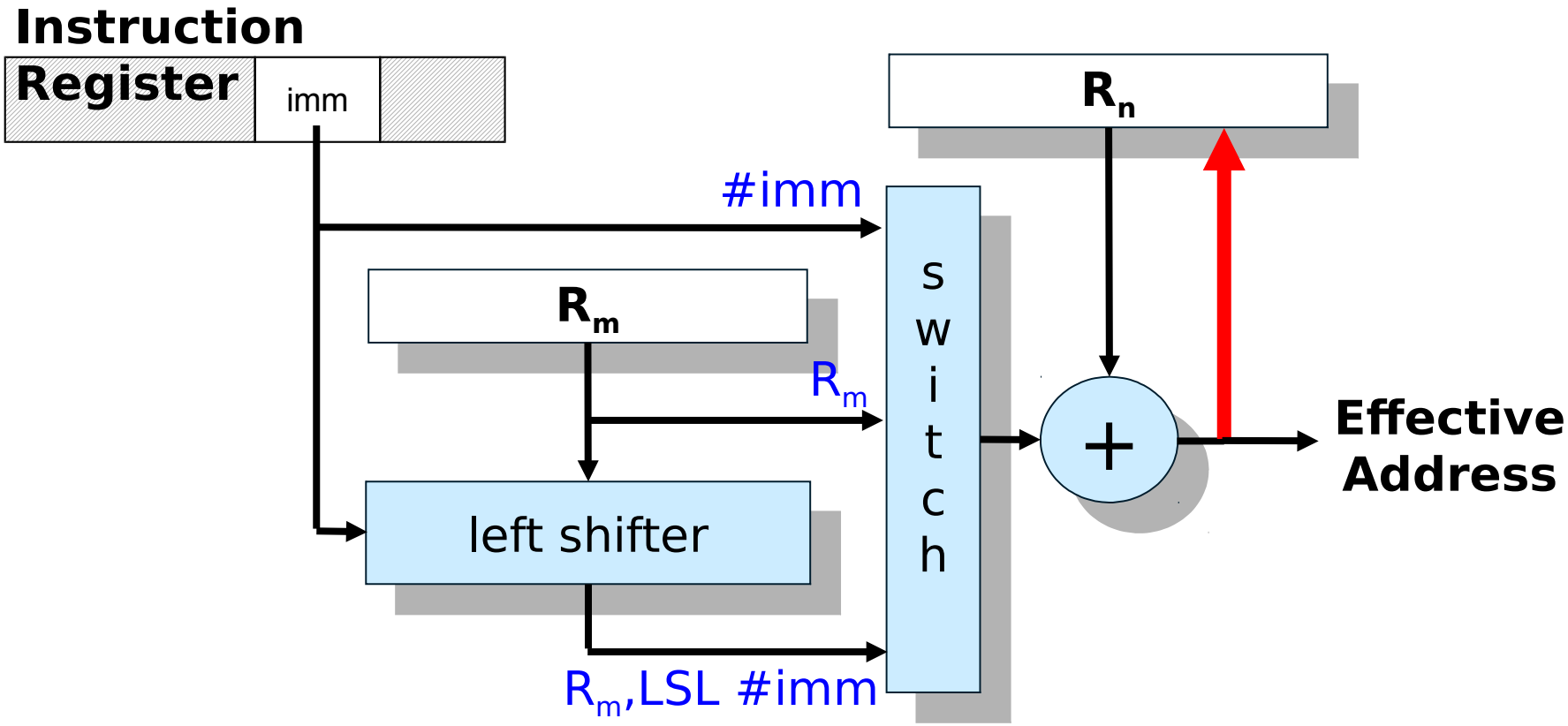
(Used Infrequently)

Syntax	***Effective Address***	***Example***
[<R_n>,#imm] !	R_n := R_n + imm EA = R_n	[r5,#100] !
[<R_n>,<R_m>] !	R_n := R_n + R_m EA = R_n	[r4,r5] !
[<R_n>,<R_m>,LSL #<imm>] !	R_n := R_n + (R_m << imm) EA = R_n	[r4,r5,LSL #3] !

Not available with LDRD or STRD

Pre-Indexed Addressing

(Used Infrequently)



Post-Indexed Addressing

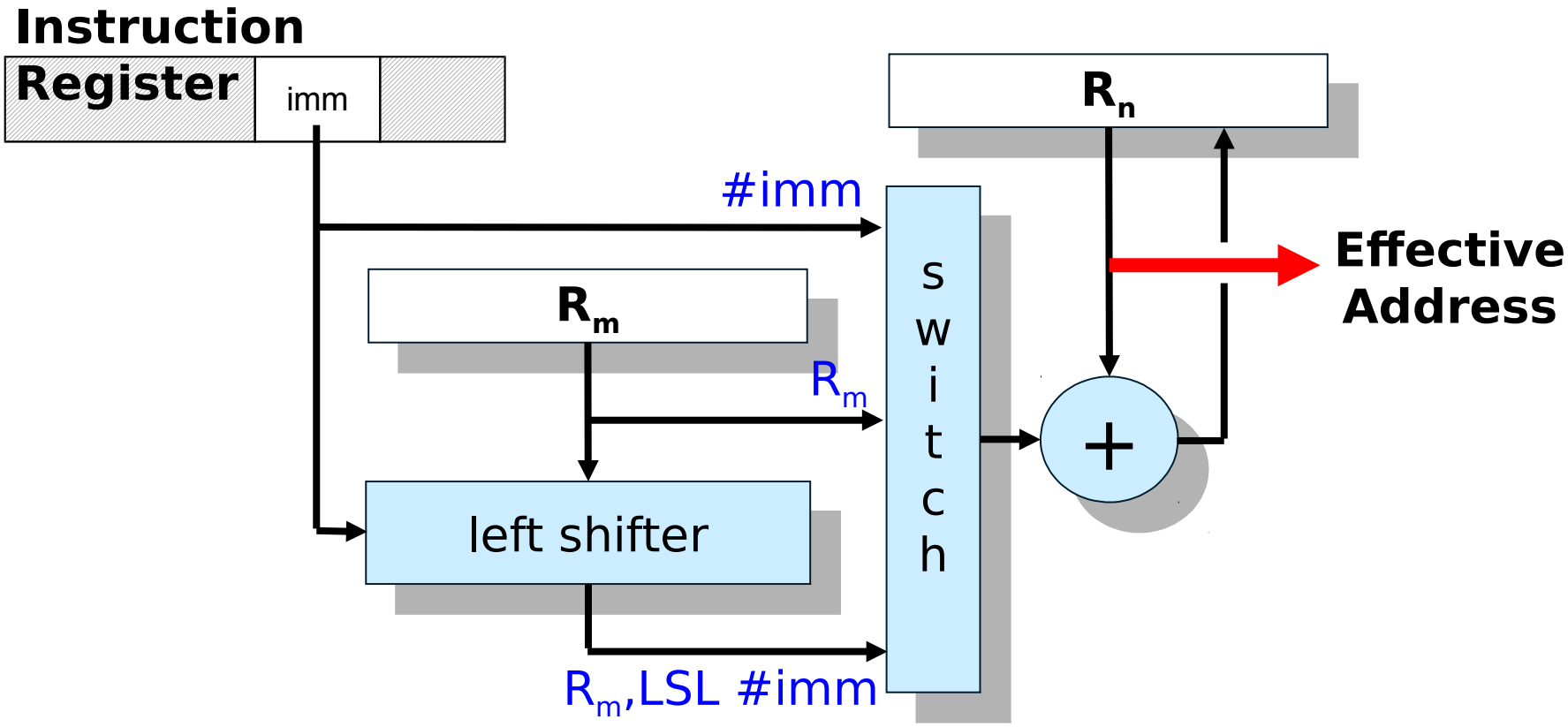
(Used Infrequently)

Syntax	***Effective Address***	***Example***
[<R_n >],#imm	EA = R_n R_n = R_n + imm	[r5],#100
[<R_n >],<R_m>	EA = R_n R_n = R_n + R_m	[r4],r5
[<R_n>],<R_m>,LSL #<imm>	EA = R_n R_n = R_n + (R_m << imm)	[r4],r5,LSL #3

Not available with LDRD or STRD

Post-Indexed Addressing

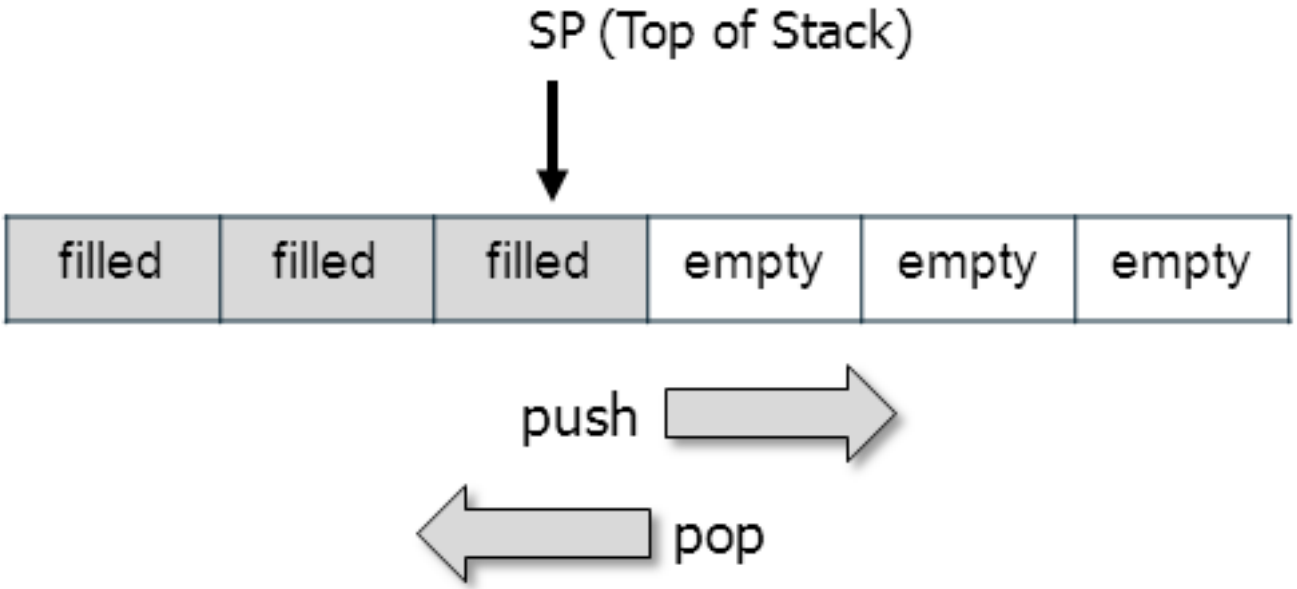
(Used Infrequently)



PUSH and POP Instructions

Multiple Load/Store	*Operation*	*{S}*	*Notes*
POP {register list}	*Repeat per reg:* reg = mem[SP] ; SP = SP + 4	n/a	**Register list: May not include SP; May include PC or LR, but not both**
PUSH {register list}	*Repeat per reg:* SP = SP – 4 ; reg = mem[SP]	n/a	**Register list may not include SP or PC.**
LDMIA R_n!,<reg. list>	regs = mem[R_n]; if !, then R_n += 4 × #regs	n/a	**Same as LDMFD; no R_n update w/out !**
STMIA R_n!,<reg. list>	regs = mem[R_n]; if !, then R_n += 4 × #regs	n/a	**Same as STMEA; no R_n update w/out !**
LDMDB R_n!,<reg. list>	regs = mem[R_n – 4 × #regs]; if !, then R_n –= 4 × #regs	n/a	**Same as LDMEA; no R_n update w/out !**
STMDB R_n!,<reg. list>	**regs = mem[R_n – 4 × #regs]; if !, then R_n –= 4 × #regs**	**n/a**	**Same as STMFD; no R_n update w/out !**

Stack Operations

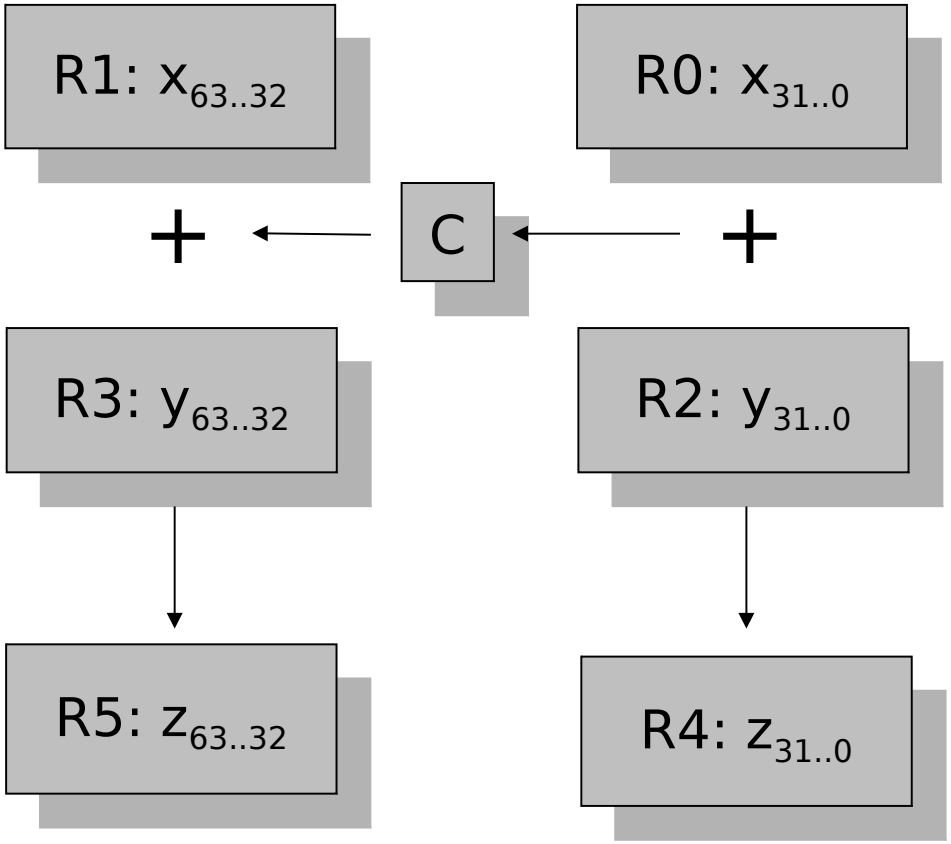


Stack grows down (towards lower memory addresses)

Move/Add/Subtract Instructions

Move / Add / Subtract		***Operation***	***{S}***	***<op>***	***Notes***
ADR	R_d,<label>	R_d = PC + imm	n/a	n/a	
MOV	R_d,<op>	R_d = <op>	NZC	imm. const. -or- reg{,<shift>}	
ADD	R_d,R_n,<op>	R_d = R_n + <op>	NZCV		
ADD	R_d,R_n,SP,<op>	R_d = R_n + SP + <op>	NZCV		
ADC	R_d,R_n,<op>	R_d = R_n + <op> + C	NZCV		
SUB	R_d,R_n,<op>	R_d = R_n – <op>	NZCV		
SUB	R_d,SP,<op>	R_d = SP – <op>	NZCV		
SBC	R_d,R_n,<op>	R_d = R_n – <op> + C – 1	NZCV		
RSB	R_d,R_n,<op>	R_d = <op> –R_n	NZCV		
NEG	R_d,R_n	R_d = –R_n	n/a		updates NZCV

Double-Precision Addition




```
LDRD  R0,R1,x
LDRD  R2,R3,y
ADDS  R4,R0,R2
ADC   R5,R1,R3
STRD  R4,R5,z
```

Multiply/Divide Instructions

Multiply / Divide		***Operation***	***{S}***	***Notes***
MUL	R_d, R_n, R_{dm}	$R_d = R_n \times R_{dm}$	NZ C	32-bit product; C = undefined
MLA	R_d, R_n, R_m, R_a	$R_d = R_a + (R_n \times R_m)$	n/a	32-bit product
MLS	R_d, R_n, R_m, R_a	$R_d = R_a - (R_n \times R_m)$	n/a	
UMULL	$R_{dlo}, R_{dhi}, R_n, R_m$	$R_{dhi}R_{dlo} = R_n \times R_m$	n/a	Unsigned 64-bit product
UMLAL	$R_{dlo}, R_{dhi}, R_n, R_m$	$R_{dhi}R_{dlo} = R_{dhi}R_{dlo} + R_n \times R_m$	n/a	
SMULL	$R_{dlo}, R_{dhi}, R_n, R_m$	$R_{dhi}R_{dlo} = R_n \times R_m$	n/a	Signed 64-bit product
SMLAL	$R_{dlo}, R_{dhi}, R_n, R_m$	$R_{dhi}R_{dlo} = R_{dhi}R_{dlo} + R_n \times R_m$	n/a	
UDIV	R_d, R_n, R_m	$R_d = R_n \div R_m$	n/a	Unsigned 32-bit quotient; no remainder
SDIV	R_d, R_n, R_m	$R_d = R_n \div R_m$	n/a	Signed 32-bit quotient; no remainder

Remainder (C = A % B)

```
LDR   R0,A        ; R0 = dividend (A)
LDR   R1,B        ; R1 = divisor (B)
UDIV  R2,R0,R1    ; R2 = quotient (A/B)
MLS   R0,R2,R1,R0 ; R0 = A – B×(A/B)
STR   R0,C        ; C = remainder
```

Bitwise Instructions

Bitwise Instructions	***Operation***	***{S}***	***<op>***	***Notes***
AND R_d,R_n,<op>	R_d = R_n & <op>	NZC	imm. const. -or- reg{,<shift>}	
ORR R_d,R_n,<op>	R_d = R_n | <op>	NZC		
EOR R_d,R_n,<op>	R_d = R_n ^ <op>	NZC		
BIC R_d,R_n,<op>	R_d = R_n & ~<op>	NZC		
ORN R_d,R_n,<op>	R_d = R_n | ~<op>	NZC		
MVN R_d,R_n	R_d = ~R_n	NZC		

Bitwise Instructions

```
uint32_t  a, b ;
a = b & ~(1 << 4) ;
```

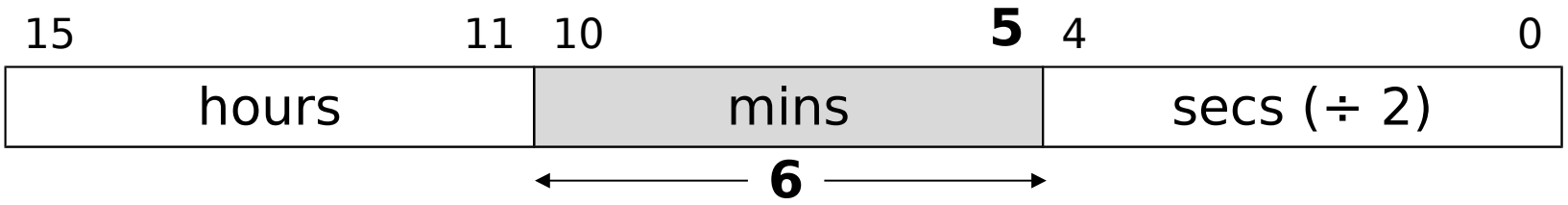
```
LDR     R0,b
AND     R0,R0,#0xFFFFFFEF
STR     R0,a
```

```
uint32_t  a ;
a &= ~(1 << 4) ;
```

```
LDR     R0,a
BIC R0,R0,#0x10
STR     R0,a
```

Bitfield Instructions

Bitfield Instructions	***Operation***	***{S}***	***Notes***
BFC R_d,#lsb,#width	R_d<bits> := 0	n/a	
BFI R_d,R_n,#lsb,#width	R_d<bits> := R_n<lsb's>	n/a	
SBFX R_d,R_n,#lsb,#width	R_d := R_n<bits>	n/a	Sign extends
UBFX R_d,R_n,#lsb,#width	R_d := R_n<bits>	n/a	Zero extends




```
struct { uint16_t  hours:5,  mins:6,  secs:5; } time ;
uint32_t  minutes ;
```

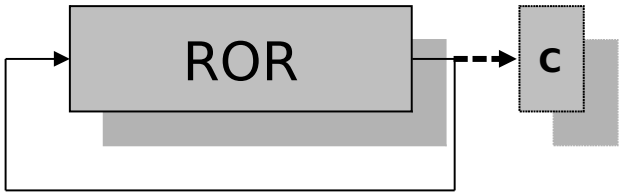
```
minutes = time.mins ;   ⇒   LDRH   R0,time
                            UBFX   R0,R0,#5,#6
                            STR    R0,minutes

time.mins = 55 ;        ⇒   LDRH   R0,time
                            LDR    R1,=55
                            BFI R0,R1,#5,#6
                            STRH   R0,time
```

Shift Codes:
Any of these may be applied to the register option of "<op>" in Move / Add / Subtract, Compare, and Bitwise Groups.

<shift>	***Meaning***	***Notes***
LSL #n	Logical shift left by n bits	Zero fills; 0 ≤ n ≤ 31
LSR #n	Logical shift right by n bits	Zero fills; 1 ≤ n ≤ 32
ASR #n	Arithmetic shift right by n bits	Sign extends; 1 ≤ n ≤ 32
ROR #n	Rotate right by n bits	1 ≤ n ≤ 32
RRX	Rotate right w/C by 1 bit	

Types of Shift Operations



RRX only shifts by 1 bit,
3rd operand omitted

Shift Instructions

Shifts	*Operation*	*{S}*	*<op>*	*Notes*
ASR R_d,R_n,<op>	R_d = R_n >> <op>	NZC	R_m –or- imm	**Sign extends**
LSL R_d,R_n,<op>	R_d = R_n << <op>	NZC	R_m –or- imm	**Zero fills**
LSR R_d,R_n,<op>	R_d = R_n >> <op>	NZC	R_m –or- imm	
ROR R_d,R_n,<op>	R_d = R_n >> <op>	NZC	R_m –or- imm	**right rotate**
RRX R_d,R_n	**R_d = R_n >> 1**	**NZC**	**n/a**	**right shift, fill w/C**

Double Precision Left Shift

		R1	R0
LDRD	R0,R1,data64	b_{63} … b_{32}	b_{31} … b_0
LSL	R1,R1,#1	b_{62} … b_{32} 0	b_{31} … b_0
ORR	R1,R1,R0,LSR #31	b_{62} … $b_{32}b_{31}$	b_{31} … b_0
LSL	R0,R0,#1	b_{62} … b_{31}	b_{30} … b_0 0
STRD	R0,R1,data64		