L3 (CHAPTER 6)

Programming in Assembly
Part 2: Data Manipulation

Acknowledgement: some slides taken from UC Berkeley CS61C courseware

Instruction Set Characteristics

- Fixed vs. variable length.
- Addressing modes.
- Number of operands.
- Types of operands.

RISC vs. CISC

- Complex instruction set computer (CISC):
  - Variable instruction lengths
  - Many addressing modes;
  - Many operations.
- Reduced instruction set computer (RISC):
  - Fixed instruction length (32-bit for ARM, 16-bit for Thumb)
  - Few addressing modes;
  - Few operations.

Programming Model

- Programming model: registers visible to the programmer.
- Some registers are not visible
  - Invisible: Instruction Register (IR) that holds the current instruction being executed
  - Visible: Program Counter (PC) that holds the next instruction to be fetched from memory

Multiple Implementations

- One instruction set (say ARM) may have different implementations by different companies (Freescale, ST Microelectronics, and many others)
  - varying clock speeds;
  - different bus widths;
  - different cache sizes;
  - etc.

ARM Assembly Syntax

- ARM instructions are written as an operation code (or simply, “opcode”) followed by zero or more operands that may be constants, registers, or memory references
- Instructions that transfer data between registers and memory specify two operands (; is followed by comments):

```
LDR R0,[R1] ;Load register R0 from memory address R1
STR R0,[R1] ;Store register R1 into memory address R1
```

- Arithmetic instructions perform a calculation between two source operands and specify a third destination operand.

```
ADD R0,R1,#5 ;Replace R0 by sum of R1 and an immediate operand (constant 5)
ADD R2,R3,R4 ;Replace R2 by sum of R3 and R4
```

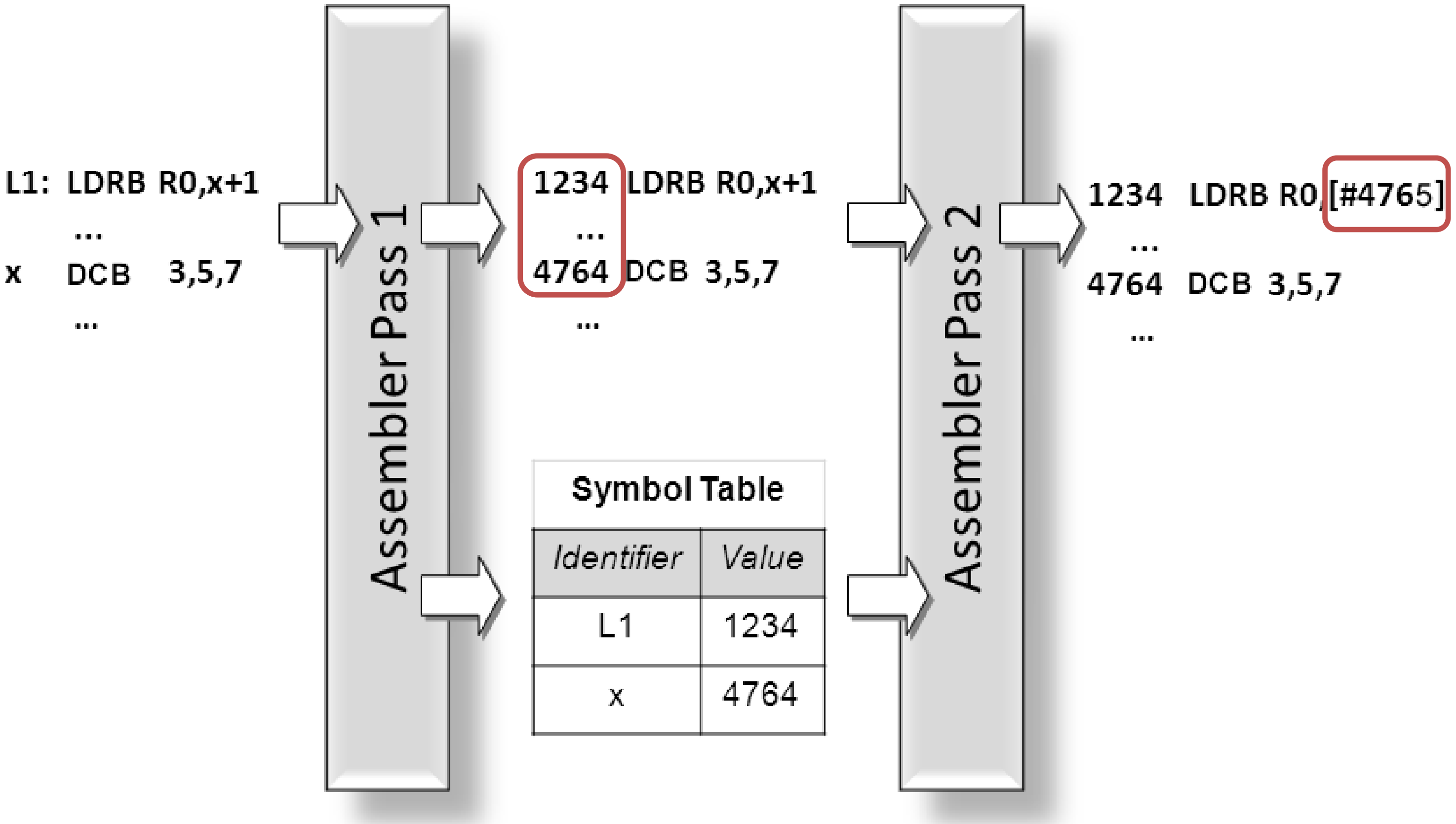
opcode	operand$_1$	operand$_2$

A typical instruction with 2 operands

LDR	R0	[R1]

Instruction LDR R0,[R1]

Two-Pass Assembly



```
L1:  LDRB R0,x+1
     ...
x    DCB   3,5,7
     ...
```

```
1234 LDRB R0,x+1
 ...
4764 DCB 3,5,7
 ...
```

Symbol Table

Identifier	*Value*
L1	1234
x	4764

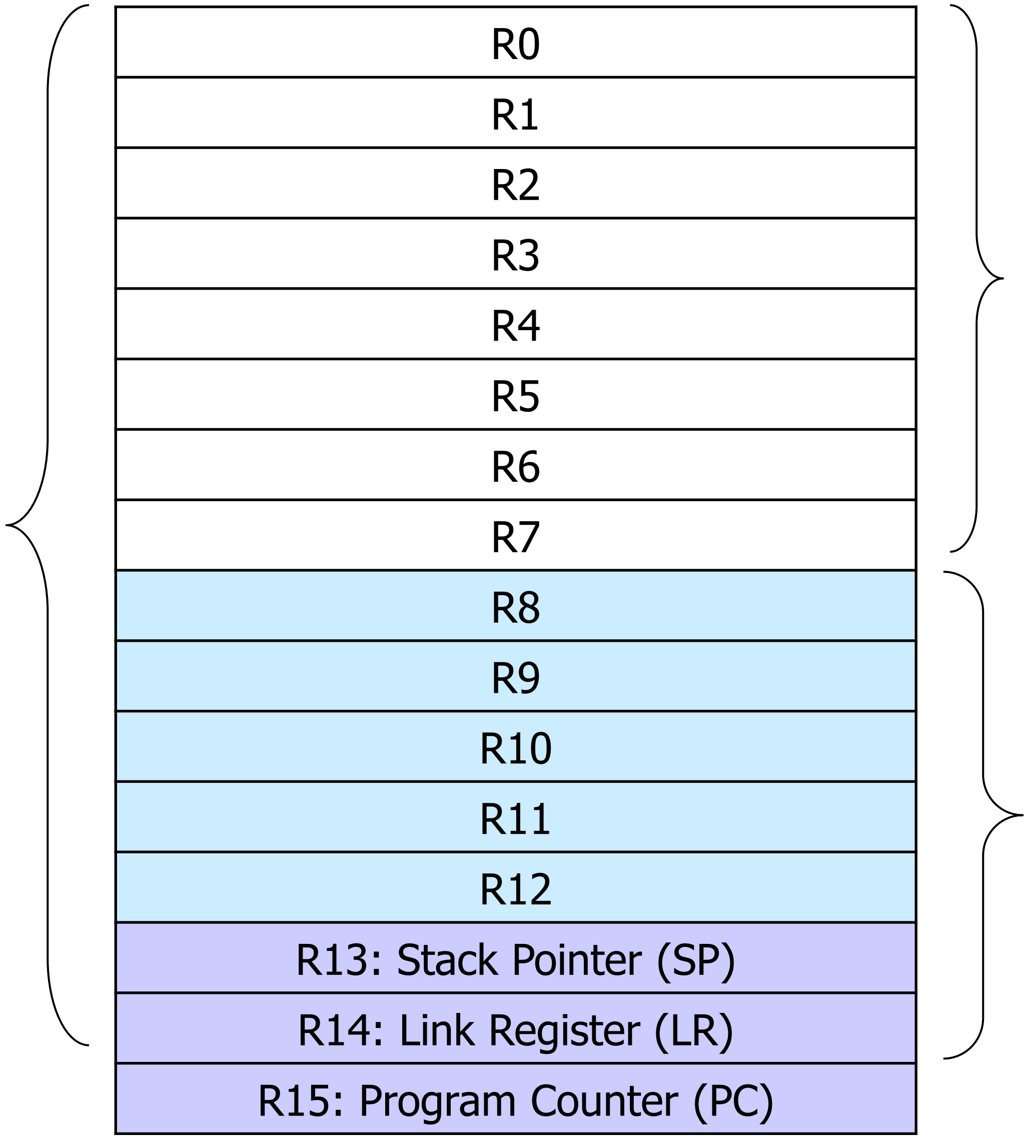
```
1234  LDRB R0,[#4765]
 ...
4764  DCB 3,5,7
 ...
```

- The Assembly makes two passes over the source code of the program.
  - During the first pass, it builds a symbol table that contains information about programmer-defined identifiers, such as the address of an instruction represented by a label attached to it, or the address of a variable represented by its identifier.
  - During the second pass, it uses this information to assemble the representation of the individual instructions.

ARM Data Types

- 1 word = 4 bytes = 32 bits
- ARM address is 32 bits long.
- Address refers to byte.
  - Address 4 starts at byte 4.
- Can be configured at power-up as either little- or big-endian.

ARM Registers



ARM Mode (32-bit instr):

15 general purpose registers

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13: Stack Pointer (SP)
R14: Link Register (LR)
R15: Program Counter (PC)

Thumb Mode (16-bit instr):

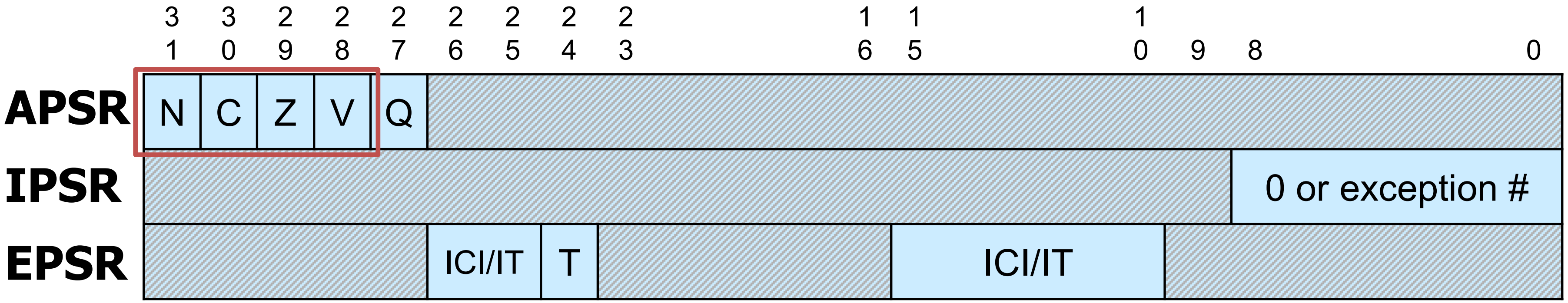
8 general purpose registers

7 “high” registers

R8-R12 only accessible with MOV, ADD, or CMP

Status Registers (xPSR)

- A 32-bit PSR (Program Status Register) stores a collection of 1-bit status flags and other information, divided into three bit fields:
  - APSR (Application Program Status Register), IPSR (Interrupt Program Status Register), and EPSR (Execution Program Status Register).
  - PSR = APSR | IPSR | EPSR (“|” stands for bitwise OR)



  - CPSR (Current Program Status Register) holds PSR of the current instruction being executed

Important Bit Flags in CPSR

Bits	Name	Description
31	**N**	**Negative (bit 31 of result is 1)**
30	**C**	**Unsigned Carry**
29	**Z**	**Zero or Equal**
28	**V**	**Signed Overflow**

Most important for application programming

- Every arithmetic, logical, or shifting operation sets CPSR bits:
  - N – Negative
    - is set if the result of a data processing instruction was negative.
  - Z – Zero
    - is set if the result was zero.
  - C – Carry
    - is set if true result > 2^n-1 for unsigned addition, or true result < 0 for unsigned subtraction (n=32 for ARM instruction set).
  - V – Overflow
    - is set if true result > 2^{n-1}-1 or true result < -2^{n-1} for signed addition or subtraction (n=32 for ARM instruction set).

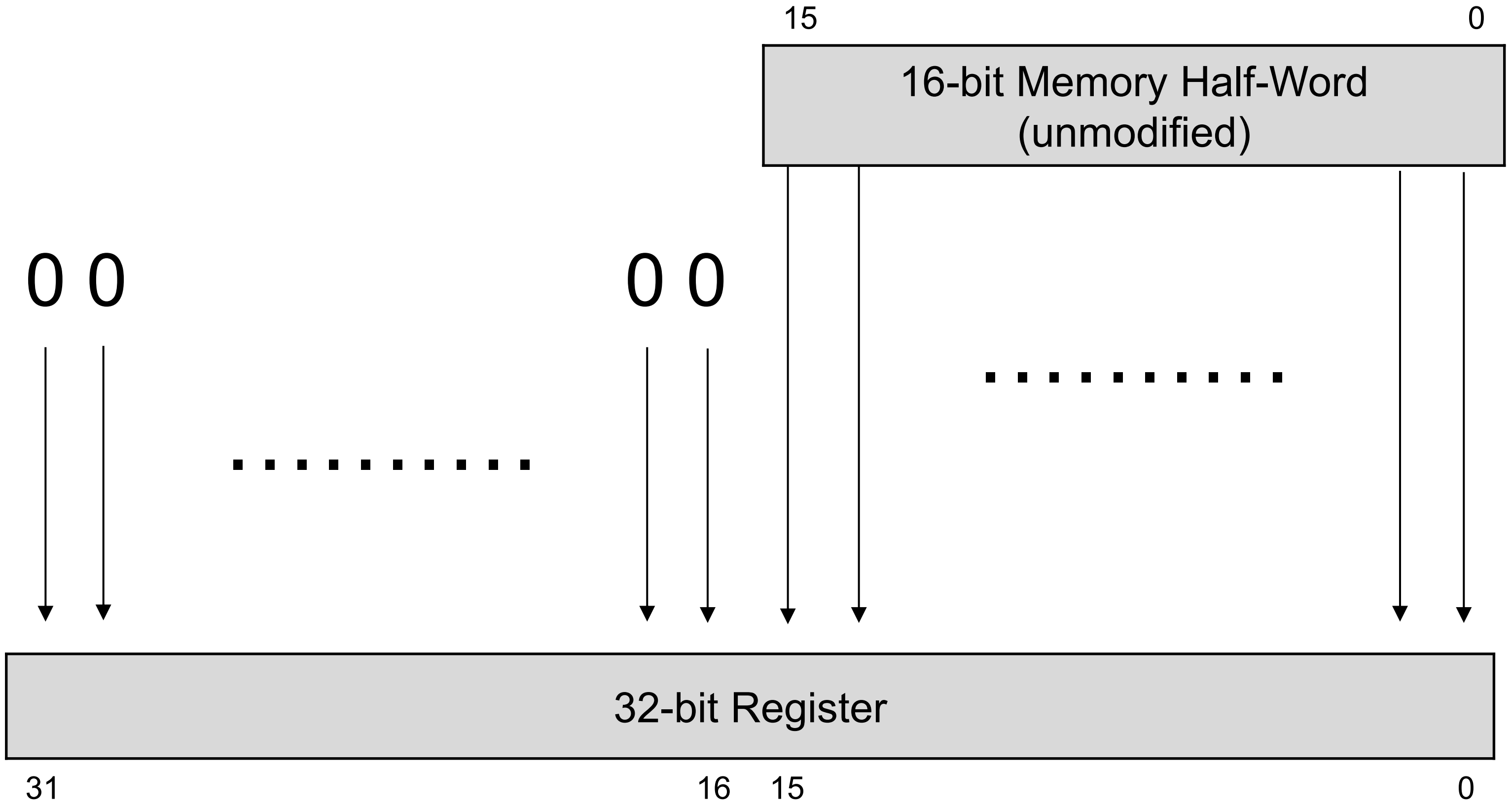
Loading Constants: MOV and MVN

- MOV r_d,*constant*
  - MOV instruction copies *constant* into register r_d;
  - Cannot support the full range of 32-bit values since only a small number of bits of the instruction are used to hold the constant. As a result, constants are limited in the range of 0 to 255 (8 bits).
  - Example: MOV R1,#100 *;R1 is assigned decimal number 100*
- MVN r_d,*constant*
  - MVN (MOV Negated) instruction copies *~constant (inverse of constant)* into register r_d;
  - Effectively doubles the # of constants
  - Assembly converts MOV w/neg. const to MVN.
  - Example: MVN R1,#100 *;R1 is assigned ~100 (decimal number −101 based on two's complement encoding)*
- MOV R0, R1 ; set R0 to R1
- MON R0, R1 ; set R0 to negated R1

Loading Constants: LDR

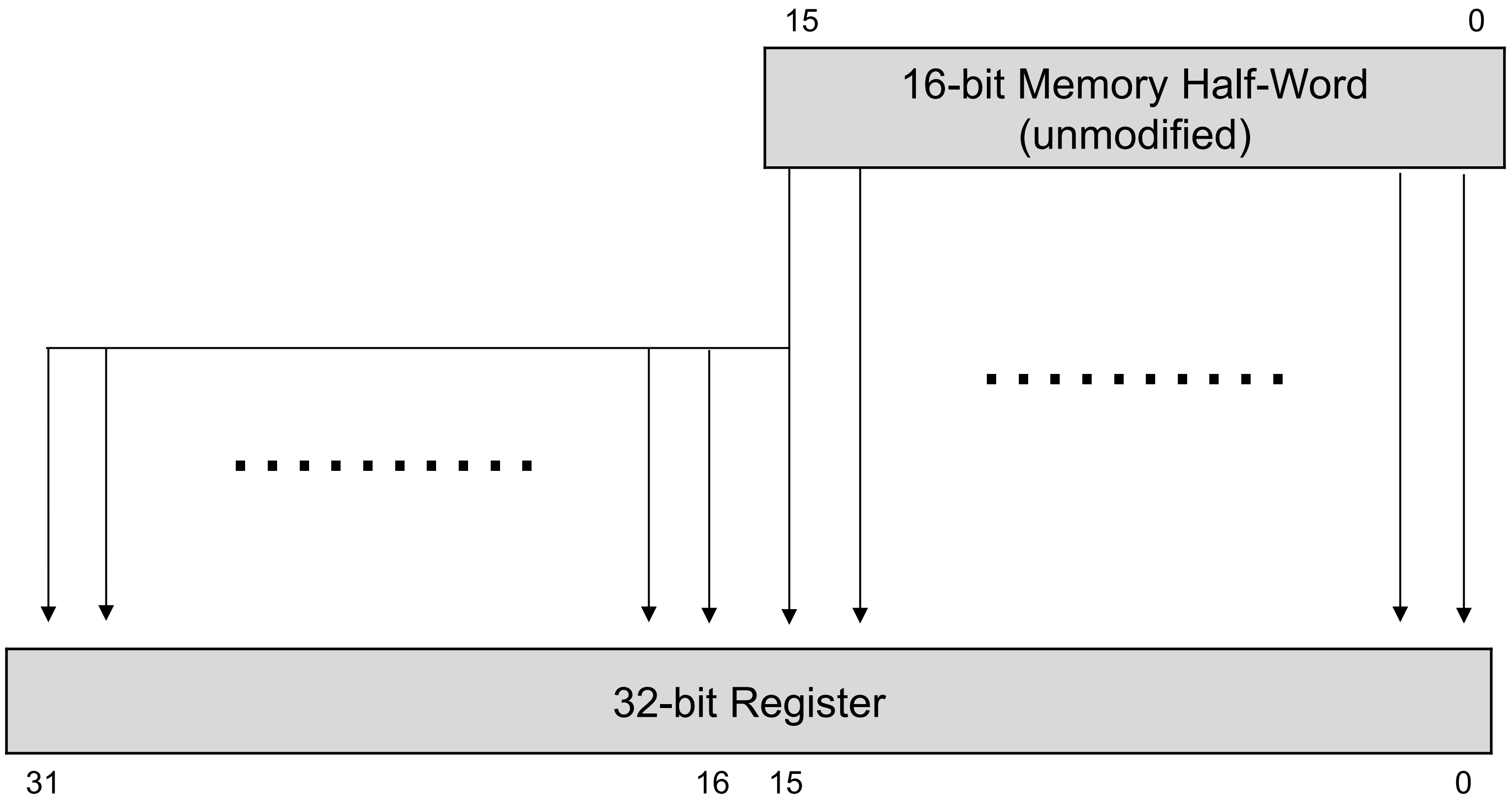
- LDR r_d,=*constant*
  - A special “pseudo-operation” that will work for any constant up to 32 bits wide.
  - You simply write what appears to be a regular ARM instruction (except that an equal sign is substituted for the pound sign) and let the Assembly sort out the most efficient way to achieve your objective:
  - Converted to MOV or MVN if possible
  - Else converts to LDR r_d,[pc,#imm]
- *Examples:*
  - LDR R1,=10 *;Assembly replaces this by MOV R1,#10.*
  - LDR R1,=−15 *;Assembly replaces this by MVN R1,#14.*
  - LDR R1,=−127435 *;Assembly replaces this by a memory reference instruction that loads the constant −127435 from a separate memory location.*

LDRH (Load Halfword)



When loading 8- or 16-bit data into a 32-bit register, the operand itself is always right justified within the register and its most significant bits filled according to whether the value is signed or unsigned. Unsigned operands less than 32 bits wide must fill the extra bit positions with zeroes (called “zero-filling”).

LDRSH (Load Signed Halfword)

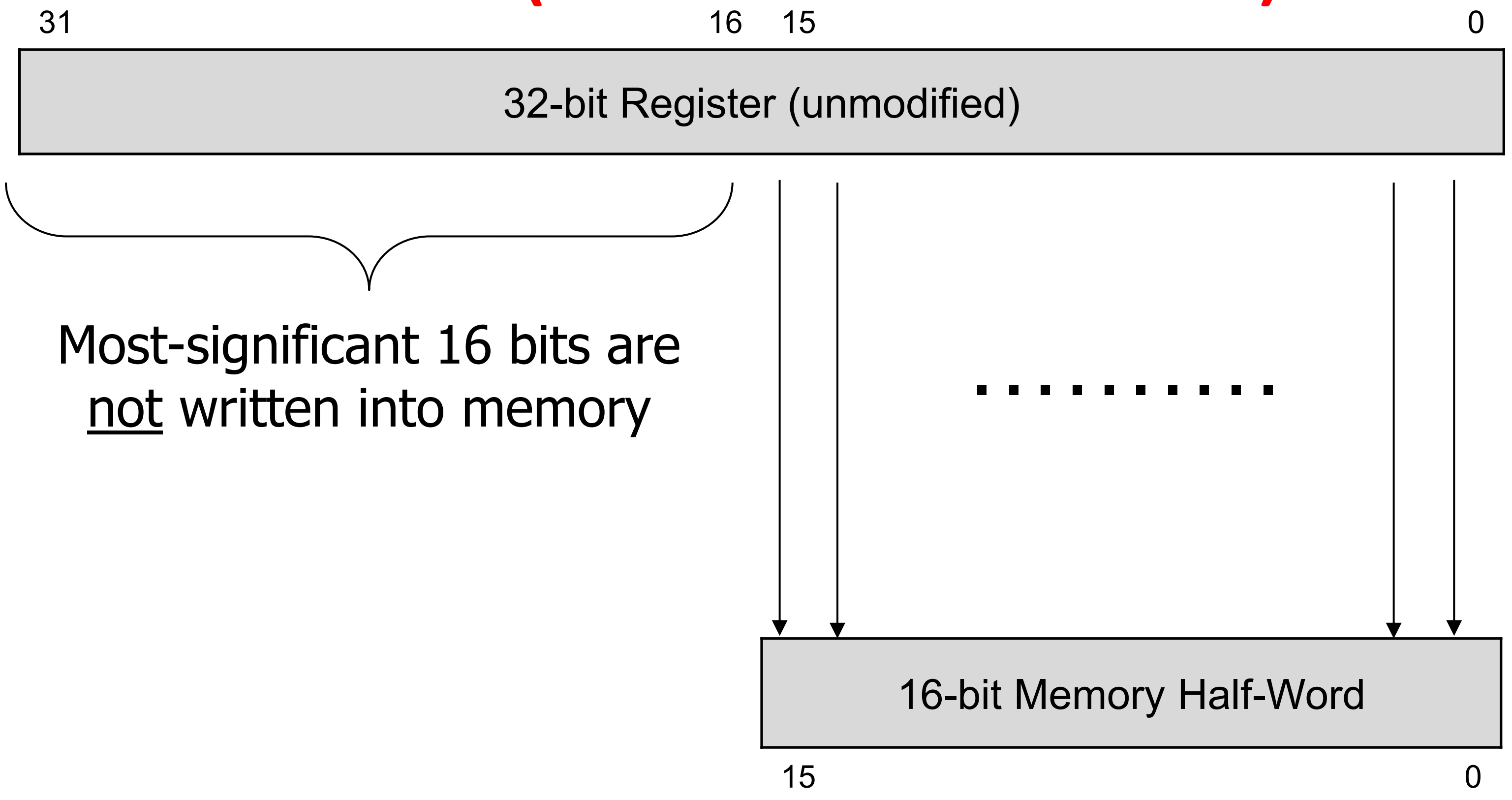


Signed operands less than 32 bits wide must fill the extra bit positions with copies of their sign bit

Load (from memory) Instructions

Load/Store Memory	***Operation***	***Notes***
LDR r_d,<mem>	$r_d \leftarrow \text{mem}_{32}$[address]	
LDRB r_d,<mem>	$r_d \leftarrow \text{mem}_{8}$[address]	Zero fills
LDRH r_d,<mem>	$r_d \leftarrow \text{mem}_{16}$[address]	Zero fills
LDRSB r_d,<mem>	$r_d \leftarrow \text{mem}_{8}$[address]	Sign extends
LDRSH r_d,<mem>	$r_d \leftarrow \text{mem}_{16}$[address]	Sign extends
LDRD r_t,r_{t2},<mem>	r_{t2}. $r_t \leftarrow \text{mem}_{64}$[address]	

STRH (Store Halfword)

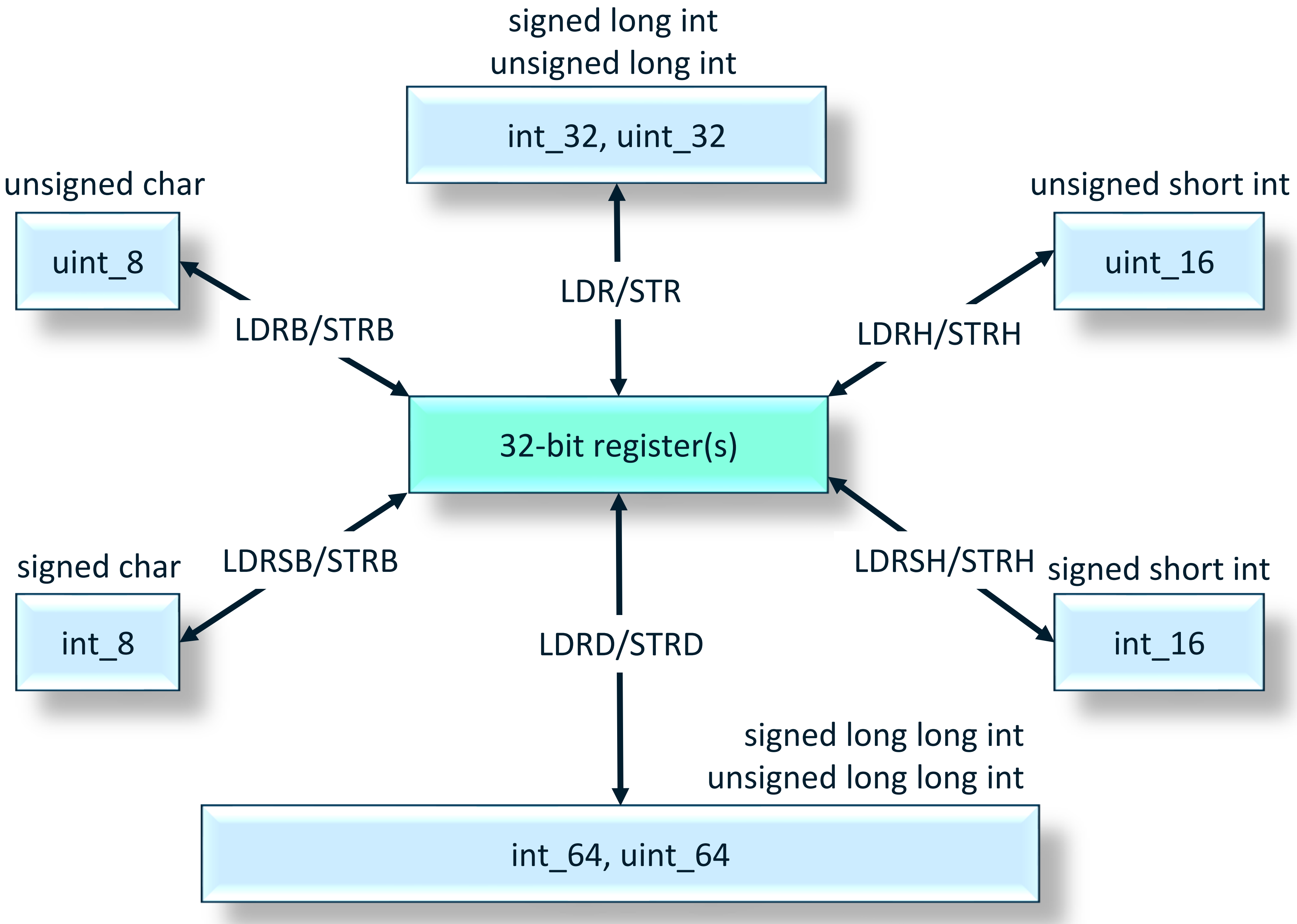

31
16
15
0
32-bit Register (unmodified)
Most-significant 16 bits are not written into memory
....
16-bit Memory Half-Word
15
0

Writing a half-word result to address 104: The two memory bytes that should be written are at addresses 104 and 105. However, the 32-bit memory data bus is actually 4 bytes wide, corresponding to addresses 104, 105, 106, and 107. Each byte of the memory data bus has its own write enable; during the memory write cycle, the write-enable signals for addresses 106 and 107 are disabled so that only the bytes at addresses 104 and 105 are modified.

Store (to memory) Instructions

Load/Store Memory	***Operation***	***Memory Byte Addresses Actually Written***			
STR r_d,<mem>	$r_d \rightarrow \text{mem}_{32}[\text{addr}]$	addr+3	addr+2	addr+1	addr
STRB r_d,<mem>	$r_d \rightarrow \text{mem}_{8}[\text{addr}]$				addr
STRH r_d,<mem>	$r_d \rightarrow \text{mem}_{16}[\text{addr}]$			addr+1	addr
STRD r_t, r_{t2},<mem>	$r_{t2}.r_t \rightarrow \text{mem}_{64}[\text{addr}]$				

Summary of LDR/STR Commands



Addressing Modes

- Offset addressing (most common):

```
LDR  R1,[R0]   ;Load R1 from memory address R0
LDR  R1,[R0,#4]  ;Load R1 from memory address R0+4 (R0 unchanged)
```

- Pre-index with update

```
LDR  R1,[R0,#4]!  ; Update R0 = R0+4, then load from memory address R0
```

- Post-Index

```
LDR  R1,[R0],#4  ;Load from memory address R0, then update R0 = R0+4
```

Addressing Format	Example	Equivalent
Offset addressing	LDR r1, [r0, #4]	r1 ← memory[r0 + 4], r0 is unchanged

Pre-index with update	LDR r1, [r0, #4]!	r1 ← memory[r0 + 4]
		r0 ← r0 + 4

Post-Index	LDR r1, [r0], #4	r1 ← memory[r0]
		r0 ← r0 + 4

Offset Addressing

Syntax	***Memory Address***	***Example***
[r_n]	r_n	[R5]
[<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]

Quiz: How can you put a 32-bit memory address into a 32-bit instruction?

Answer: The memory address is stored in a register r_n. For ARM instruction, only 4 bits are need to encode ID of register r_n (since there are a total of 15 general-purpose registers).

LDR	R0	[r_n]

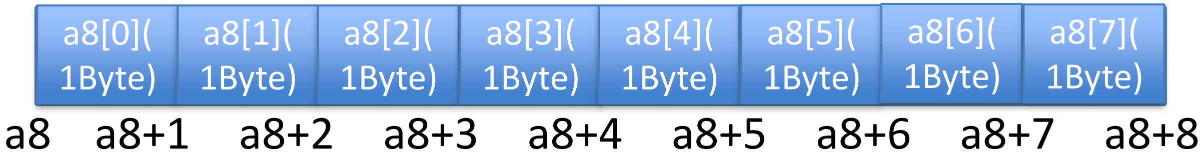
A typical instruction with 2 operands

Using Offset Addressing

```
C:              Assembly:
int *p ;        LDR     R0,=0
...             LDR     R1,p
*p = 0 ;        STR     R0,[R1] % Store 0 into memory address R1
```

```
C:              Assembly:
int *p ;        LDR     R0,=0
…               LDR     R1,p
*(p + 1) = 0 ;  STR     R0,[R1,#4] % Store 0 into memory address R1+4，
                                   % since a long is 4 bytes, so adding 1 to pointer p
                                   % increments the memory address by 4
```

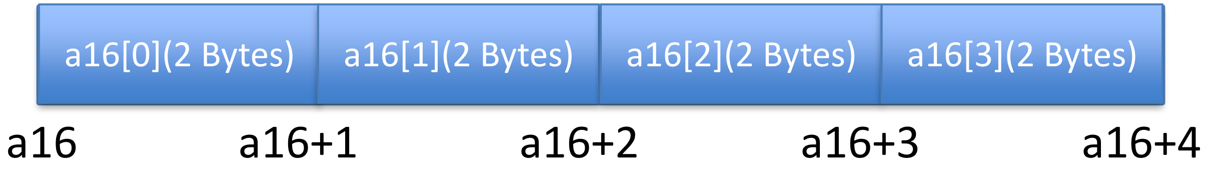
Offset Addressing for Arrays



a8[0](1Byte)	a8[1](1Byte)	a8[2](1Byte)	a8[3](1Byte)	a8[4](1Byte)	a8[5](1Byte)	a8[6](1Byte)	a8[7](1Byte)

a8 a8+1 a8+2 a8+3 a8+4 a8+5 a8+6 a8+7 a8+8

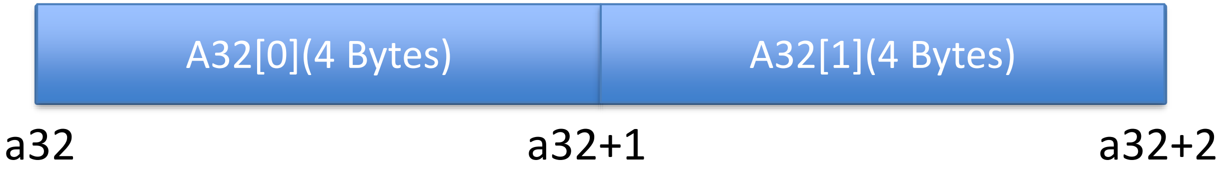
```
C:                  Assembly:
char a8[100];       LDR     R0,=0
int k;              ADR     R1,a8
…                   LDR     R2,k
a8[k] = 0 ;         STRB    R0,[R1,R2] % Store 0 into memory address a8+k*1, since a
                                        % char is 1 Byte
```



a16[0](2 Bytes)	a16[1](2 Bytes)	a16[2](2 Bytes)	a16[3](2 Bytes)

a16 a16+1 a16+2 a16+3 a16+4

```
short a16[100];     LDR     R0,=0
…                   ADR     R1,a16
a16[5] = 0 ;        STRH    R0,[R1,#10] % Store 0 into memory address a16+5*2, since
                                         % a short is 2 Bytes
```



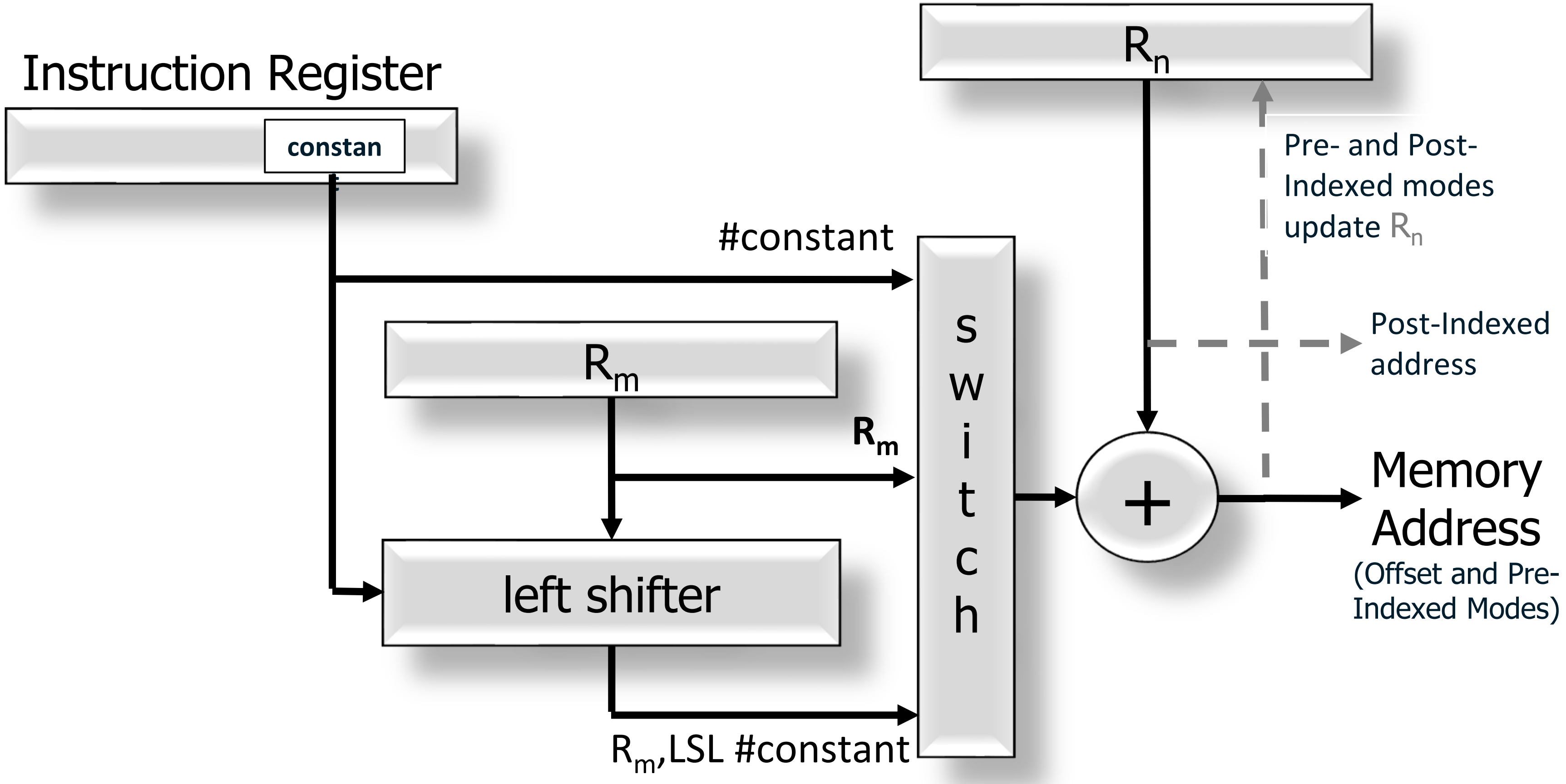
A32[0](4 Bytes)	A32[1](4 Bytes)

a32 a32+1 a32+2

```
int a32[100];       LDR     R0,=0
                    ADR     R1,a32
…                   LDR     R2,k
a32[k] = 0 ;        STRR0,[R1,R2,LSL #2] % Store 0 into memory address a32+k*4,
                                          % since an int is 4 Bytes
```

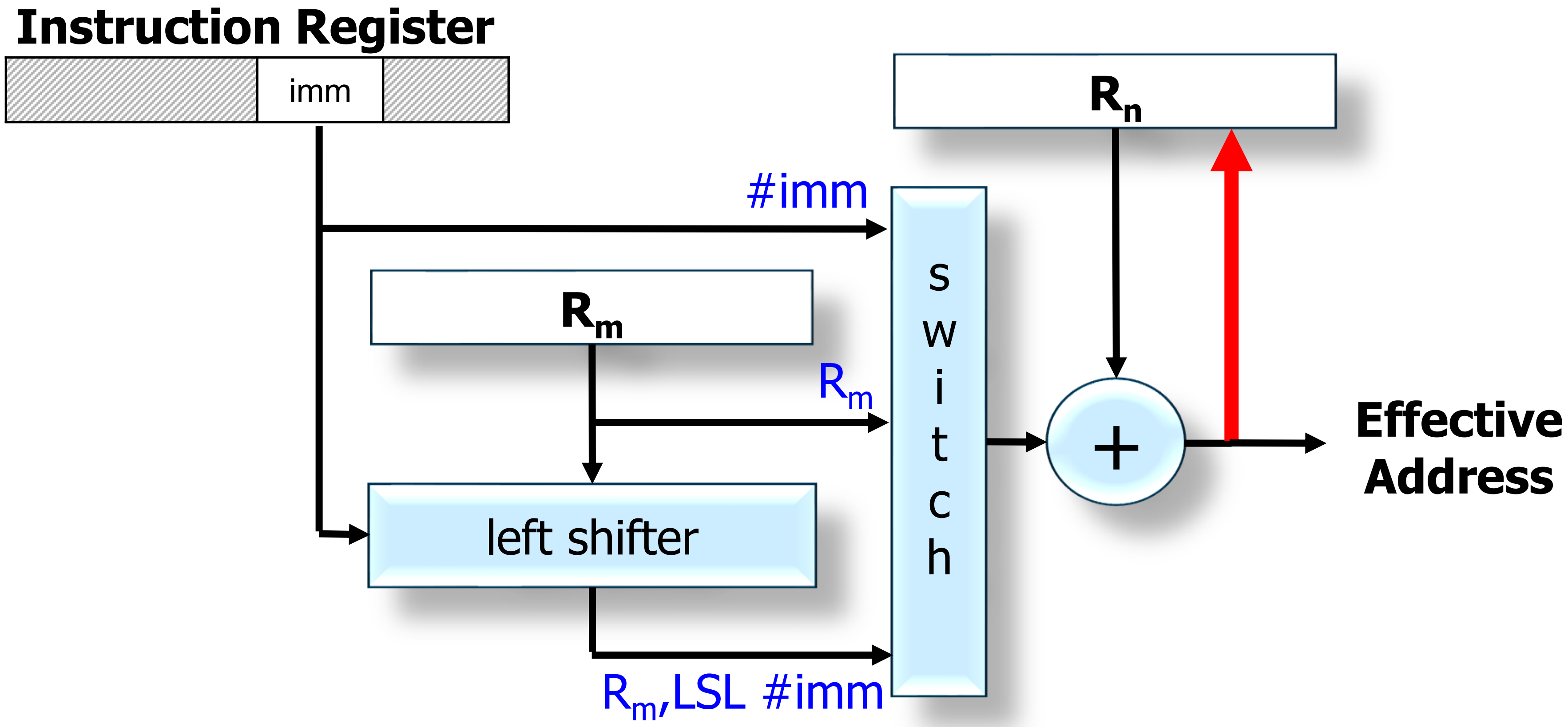
For an array x[N], x[n] is the element stored at memory address x+n*(sizeof(x)).

Address Calculation

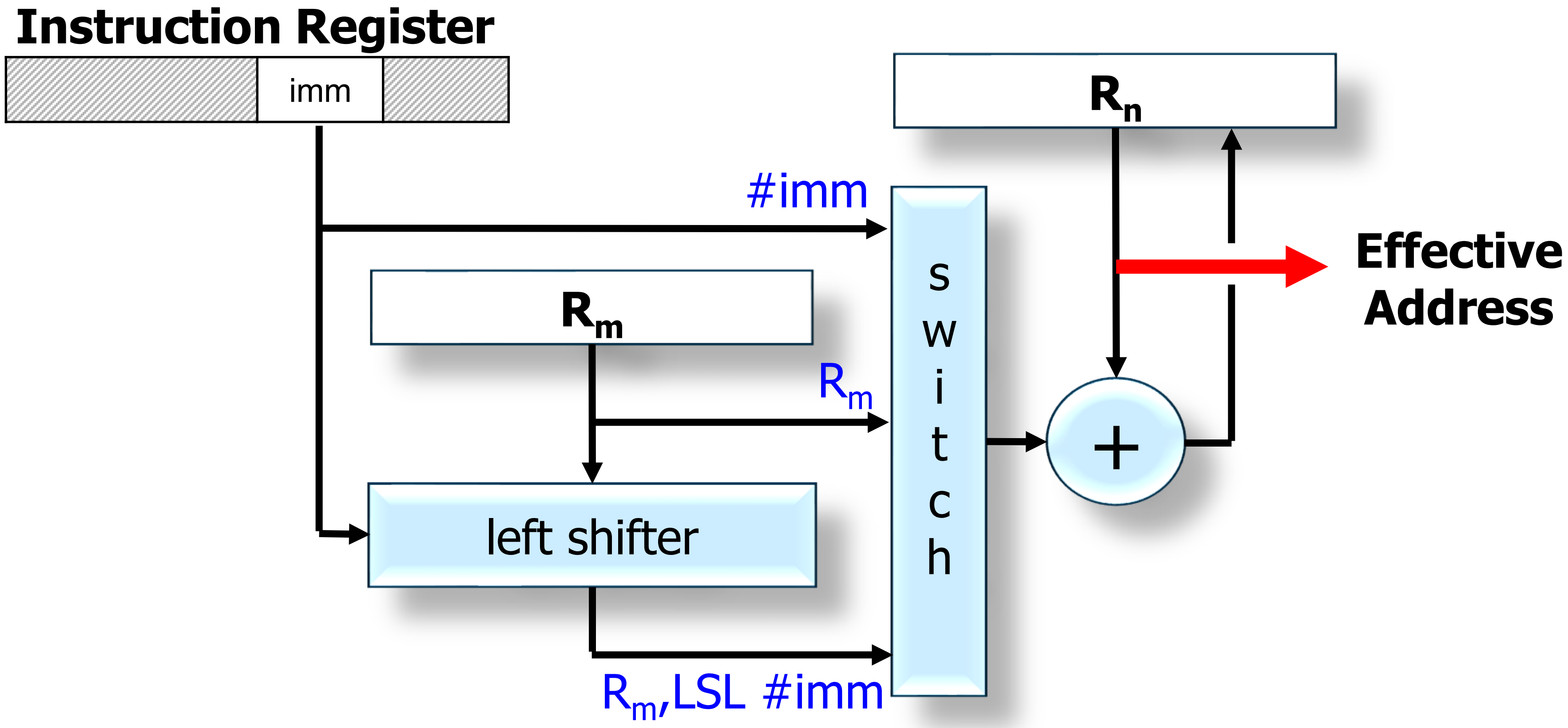


It illustrates how offset addressing can use registers, a shifter, and a small constant to generate the address of an instruction operand.

Pre-Indexed Addressing



Post-Indexed Addressing



ARM ADR Pseudo-op

- ADR pseudo-op generates instruction required to calculate address:

  ADR R1,x ;get memory address of variable x and put it in register R1

Example 1: Assignment

- C:

```
//assume x, a, b are 32-bit integer variables
x = a - b;
```

- Assembly:

```
ADR R4,a        ; get address for a
LDR R0,[R4] ; get value of a
ADR R4,b        ; get address for b, reusing R4
LDR R1,[R4] ; get value of b
SUB R0,R0,R1  ; subtract R1 from R0, and store result in R0
ADR R4,x        ; get address for x
STR R0,[R4] ; store value of x into memory
```

Example 2: Assignment

- C:

```
//assume x, a, b, c are 32-bit integer variables
x = (a + b) - c;
```

- Assembly:

```
ADR R4,a        ; get address for a
LDR R0,[R4] ; get value of a
ADR R4,b        ; get address for b, reusing R4
LDR R1,[R4] ; get value of b
ADD R3,R0,R1    ; compute a+b with R3=R0+R1
ADR R4,c        ; get address for c
LDR R2,[R4] ; get value of c
SUB R3,R3,R2    ; compute x with R3 -= R2
ADR R4,x        ; get address for x
STR R3,[R4] ; store value of x into memory
```

Can use R0 to replace R3 in this code, to reduce number of registers used, as in Example 1

Example 3: Assignment

- C:

```
//assume y, a, b, c are 32-bit integer variables
y = a*(b+c);
```

- Assembly:

```
ADR R4,b ; get address for b
LDR R0,[R4] ; get value of b
ADR R4,c ; get address for c
LDR R1,[R4] ; get value of c
ADD R2,R0,R1 ; compute partial result b+c
ADR R4,a ; get address for a
LDR R0,[R4] ; get value of a
MUL R2,R2,R0 ; compute final value for y=a*(b+c)
ADR R4,y ; get address for y
STR R2,[R4] ; store value of y into memory
```

Example 4: Assignment

- C:

```
//assume z, a, b are 32-bit integer variables
z = (a << 2) |  (b & 15);
```

- Assembly:

```
ADR R4,a ; get address for a
LDR R0,[R4] ; get value of a
MOV R0,R0,LSL 2 ; perform shift a<<2
ADR R4,b ; get address for b
LDR R1,[R4] ; get value of b
AND R1,R1,#15 ; perform AND
ORR R1,R0,R1 ; perform OR
ADR R4,z ; get address for z
STR R1,[R4] ; store value for z
```

Bitwise Instructions

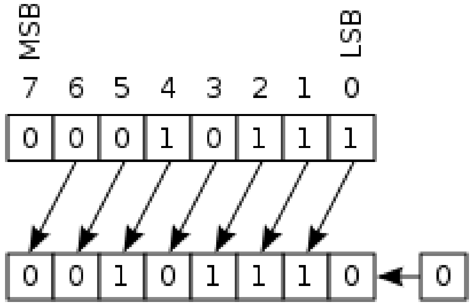
Bitwise Instructions	***Operation***	***{S}***	***<op>***	***Notes***
AND R_d,R_n,<op>	$R_d \leftarrow R_n$ & <op>	NZC	imm. const. -or- reg{,<shift>}	
ORR R_d,R_n,<op>	$R_d \leftarrow R_n \mid$ <op>	NZC		
EOR R_d,R_n,<op>	$R_d \leftarrow R_n$ ^ <op>	NZC		Exclusive OR
BIC R_d,R_n,<op>	$R_d \leftarrow R_n$ & ~<op>	NZC		Bit Clear
ORN R_d,R_n,<op>	$R_d \leftarrow R_n \mid$ ~<op>	NZC		OR Not
MVN R_d,R_n	$R_d \leftarrow$ ~R_n	NZC		Move Not

Bitfield Instructions

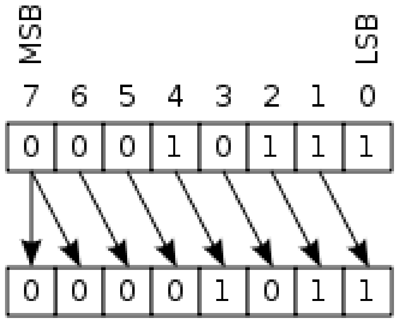
Bitfield Instructions	***Operation***	***{S}***	***Notes***
BFC R_d,#lsb,#width	R_d<bits> ← 0	n/a	
BFI R_d,R_n,#lsb,#width	R_d<bits> ← R_n<lsb's>	n/a	
SBFX R_d,R_n,#lsb,#width	R_d ← R_n<bits>	n/a	Sign extends
UBFX R_d,R_n,#lsb,#width	R_d ← R_n<bits>	n/a	Zero extends

Shift Instructions

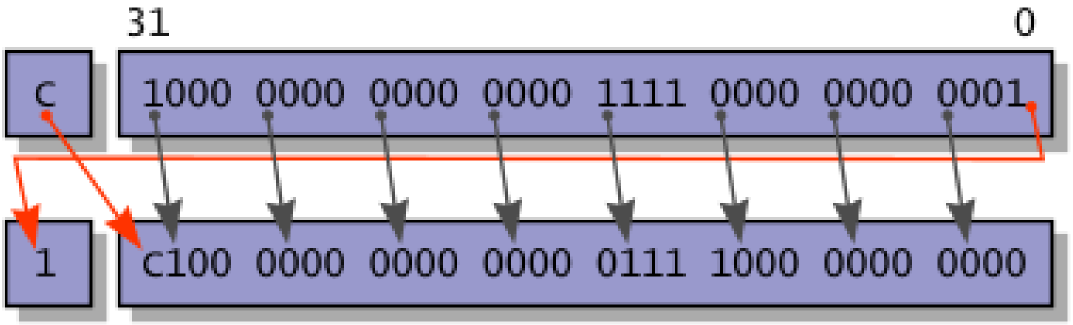
<shift>	*Meaning*	*Notes*
LSL #n	Logical shift left by n bits	Zero fills; 0 ≤ n ≤ 31
LSR #n	Logical shift right by n bits	Zero fills; 1 ≤ n ≤ 32
ASR #n	Arithmetic shift right by n bits	Sign extends; 1 ≤ n ≤ 32
ROR #n	Rotate right by n bits	1 ≤ n ≤ 32
RRX	Rotate right w/C by 1 bit	including C bit from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



LSL #1



LSR #1



RRX

Any of these may be applied to the 2nd operand register in Move / Add / Subtract, Compare, and Bitwise Groups.