arm

Armv7-A/R ISA Overview (Part 1)

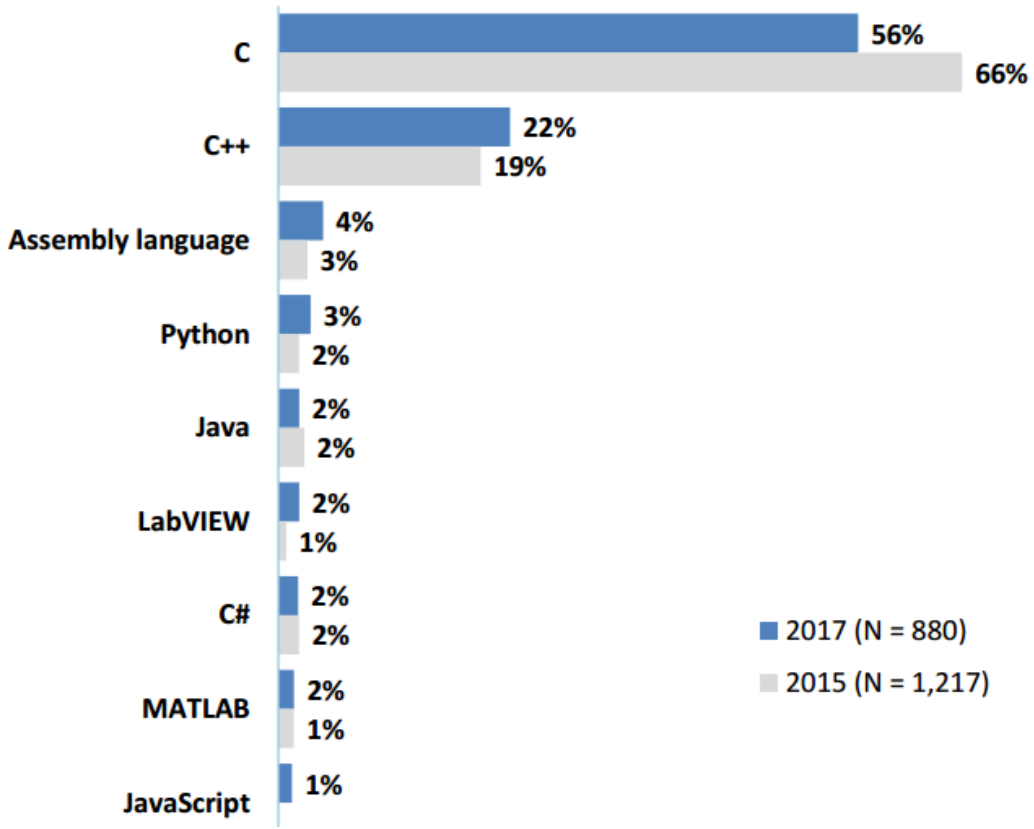
Learning Outcomes

At the end of this module, you will be able to:

- Identify the instruction types and operations for data transfer including load and store.
- Determine the results of basic data processing instructions when executed, including arithmetic, comparison, and logical instructions.
- Explain the operations of bit manipulation and move instruction types.

Why Do You Need to Know Assembler?

- Most design efforts for many systems are focused on high-level programming.
  - C/C++ is the most used language for development of embedded systems by engineers.
  - Knowledge of the instruction set is not required.



Source: UBM Tech Embedded Market Study 2017
Survey responds to the question: “My current embedded project is programmed mostly in:”

arm

Why Do You Need to Know Assembler?

Embedded systems require initialization code and interrupt routines.

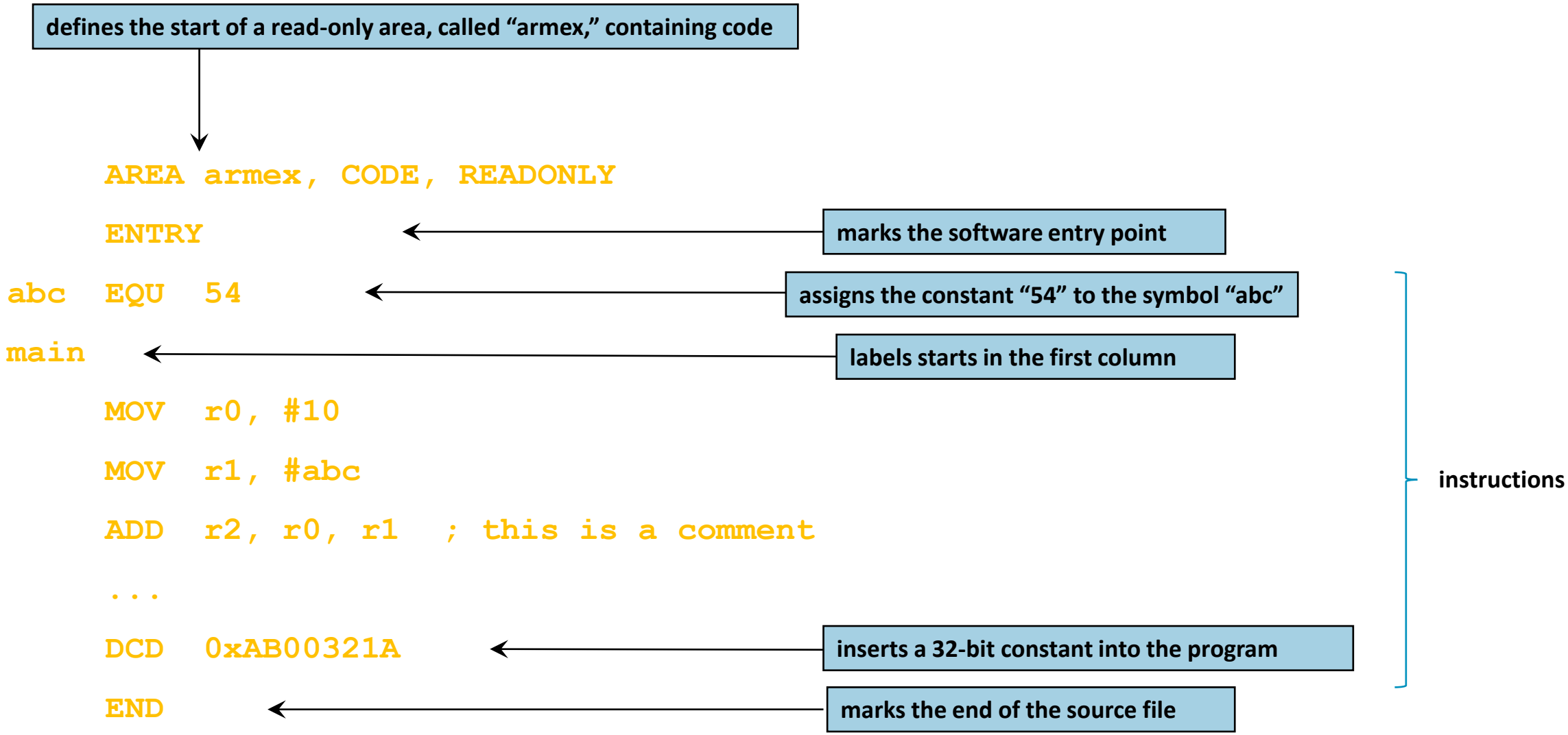
Driver writing requires knowledge of the lowest level of program abstraction.

All systems require debugging, sometimes at the instruction level.

Performance gains can be made by writing assembler routines.

Also, some features of the Arm architecture are not expressible in high-level languages.

Arm Assembler File Syntax



Single/Double Register Data Transfer

- Used to move data between one or two registers and memory

LDRD	STRD	Doubleword
LDR	STR	Word
LDRB	STRB	Byte
LDRH	STRH	Halfword
LDRSB		Signed byte load
LDRSH		Signed halfword load

- Syntax

```
LDR{<type>}{<cond>} Rd, <address>
STR{<type>}{<cond>} Rd, <address>
```

- Example
-

```
LDRB r0, [r1]              ; load bottom byte of r0 from the
                           ; byte of memory at address in r1
```


arm

Addressing Memory

- The address accessed by LDR/STR is specified by a base register with an optional offset:
  - Base register only (no offset)

```
LDR r0, [r1]
```

  - Base register plus constant

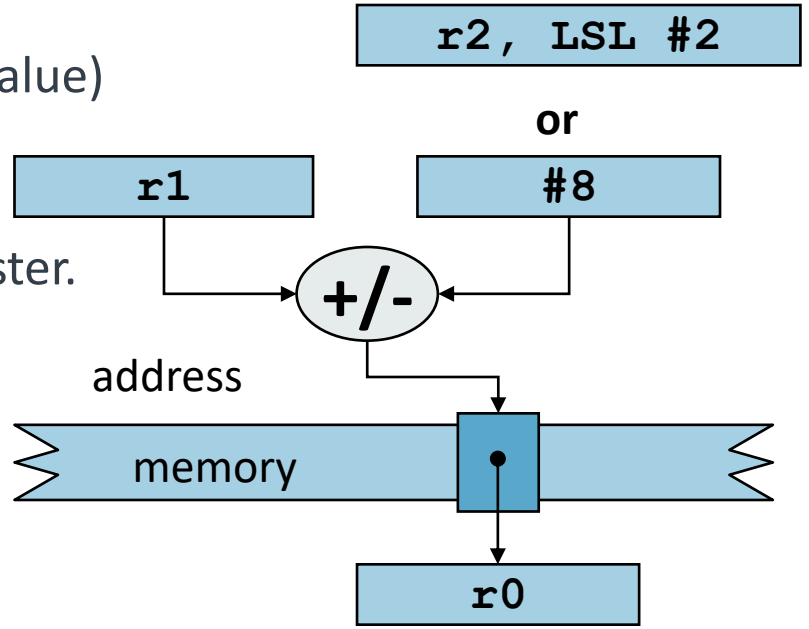
```
LDR r0, [r1, #8]
```

  - Base register, plus register (optionally shifted by an immediate value)

```
LDR r0, [r1, r2]
LDR r0, [r1, r2, LSL #2]
```

  - The offset can be either added or subtracted from the base register.

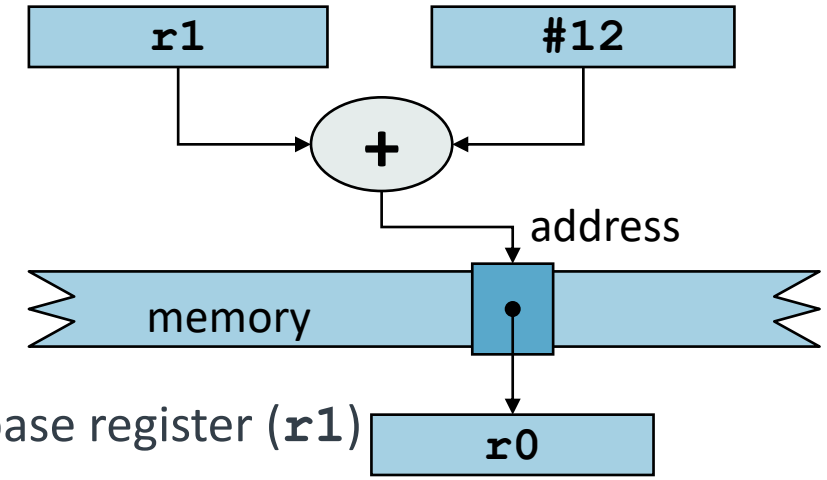
```
LDR r0, [r1, #-8]
LDR r0, [r1, -r2]
LDR r0, [r1, -r2, LSL #2]
```



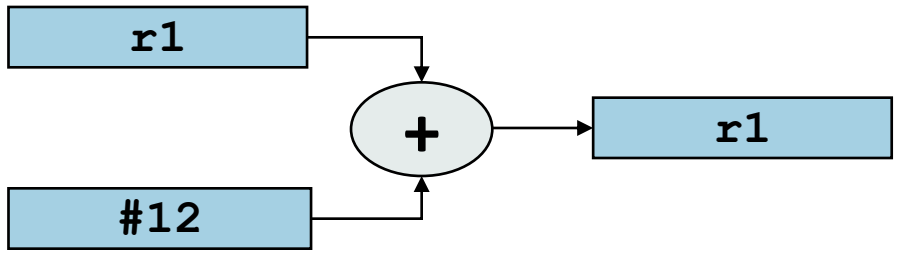
Pre- and Post-Indexed Addressing

Pre-indexed (add offset before memory access)

```
LDR r0, [r1, #12]!
```

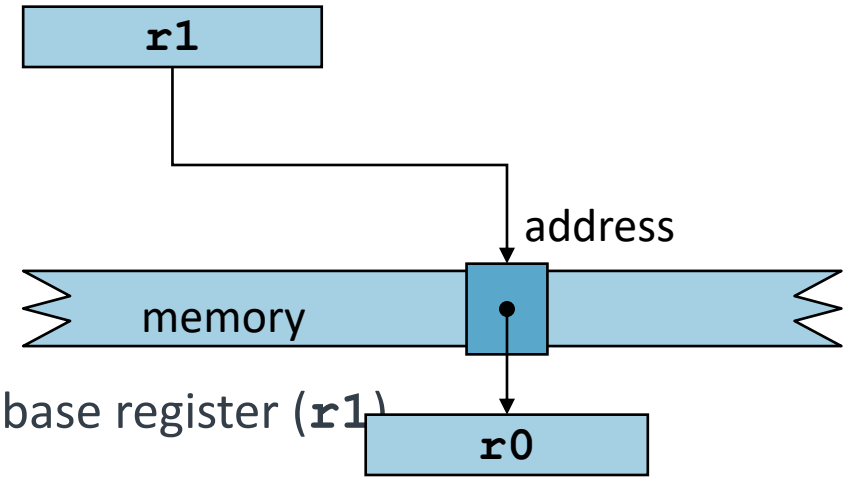


Update base register (**r1**)

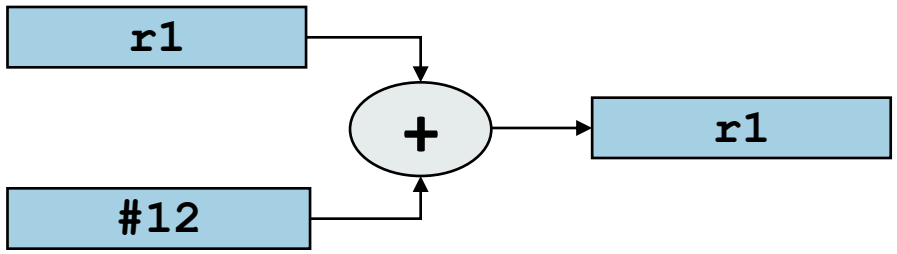


Post-indexed (add offset after memory access)

```
LDR r0, [r1], #12
```



Update base register (**r1**)



arm

Multiple Register Data Transfer

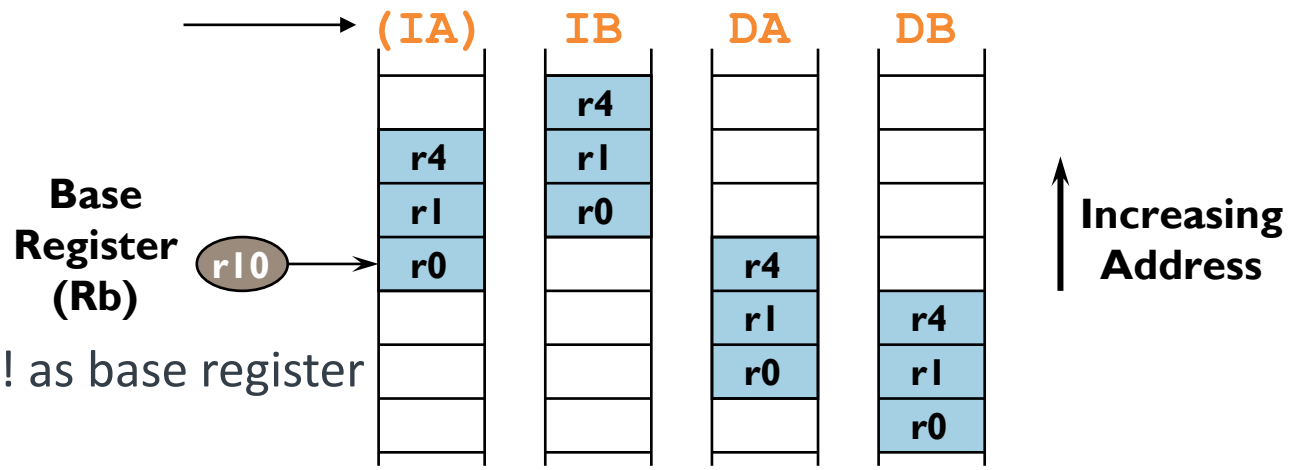
- These instructions move data between multiple registers and memory.
- Syntax:
  - `<LDM|STM>{<addressing_mode>}{<cond>} Rb{!}, <register list>`

Four addressing modes:

  - Increment after/before
  - Decrement after/before

Also

  - `PUSH/POP`, equivalent to `STMDB/LDMIA` with SP! as base register

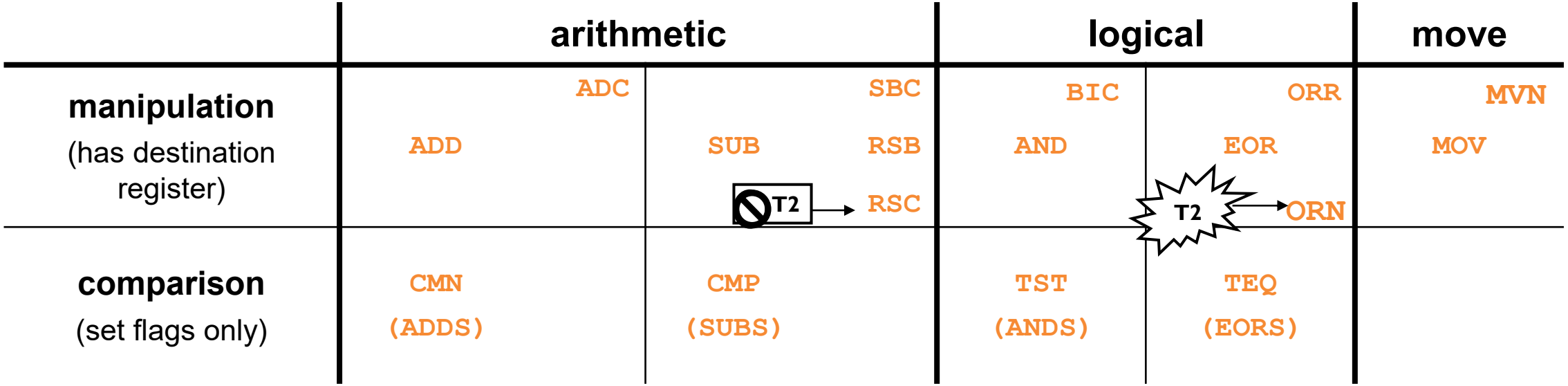


Example

```
LDM    r10, {r0,r1,r4}    ; load registers, using r10 base
PUSH   {r4-r6,pc}         ; store registers, using SP base
```

Data Processing Instructions

- These instructions operate on the contents of registers (they do not affect memory).



- Syntax:
- **<Operation>{S}{<cond>} {Rd,} Rn, Operand2**
- Examples:

```
ADD r0, r1, r2     ; r0 = r1 + r2
TEQ r0, r1 ; if r0 = r1, Z flag will be set
MOV r0, r1 ; copy r1 to r0
```

Shift/Rotate Operations

LSL - Logical Shift Left

Multiplication by a power of 2

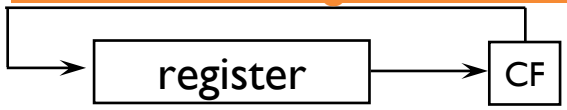
ASR - Arithmetic Shift Right

Division by a power of 2,
preserving the sign bit

LSR - Logical Shift Right

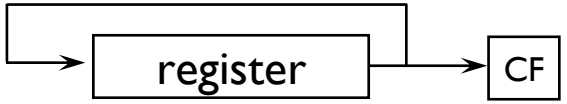
Division by a power of 2

RRX - Rotate Right Extended



Single bit rotate with wrap around
from CF to MSB

ROR - Rotate Right



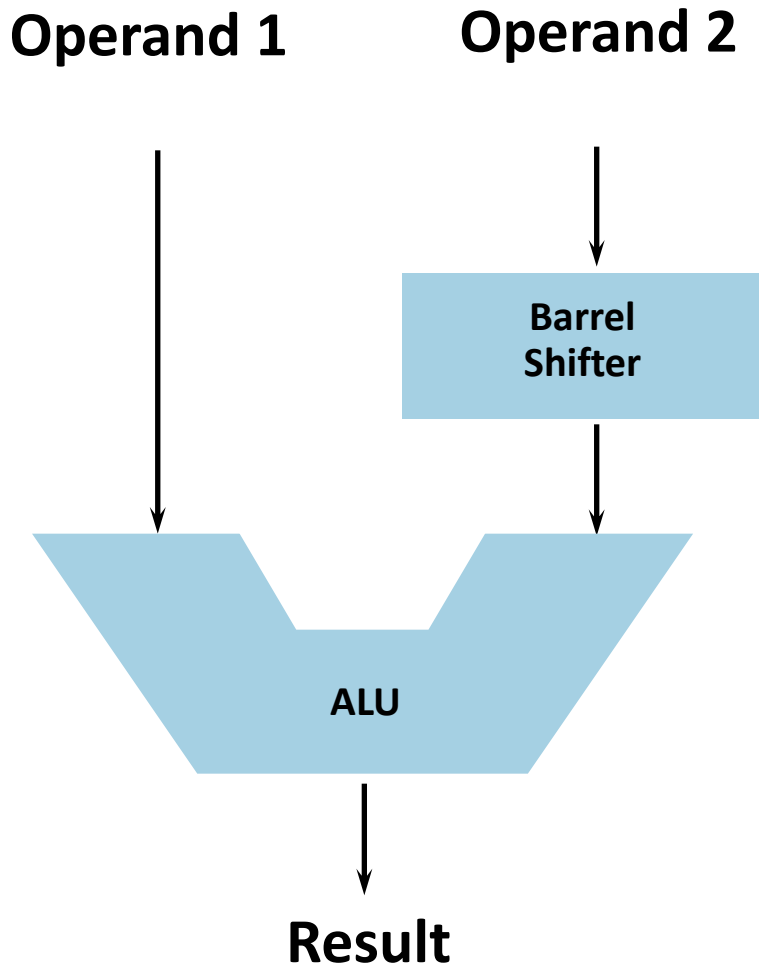
Bit rotate with wrap around
from LSB to MSB

- These are also available as part of the flexible second operand.

arm

The Flexible Second Operand

- For many instructions, the second operand is flexible.
- Register, with optional shift
  - Shift value can be either
    - 5-bit unsigned integer
    - Specified in bottom byte of another register (Arm only)
  - Can be used for multiplication by constant, e.g.,
    - `ADD r0,r5,r5, LSL #1   ; r0 = r5 * 3`
- The flexible second operand can also be an immediate value.
  - 8-bit number, with a range of 0-255
    - Arm: Rotated right by an even number of places
    - Thumb: Shifted left by any amount
  - Thumb also allows constants of this form.
    - **`0x00XY00XY`**, **`0xXY00XY00`**, and **`0xXYXYXYXY`**



arm

Instructions for Loading Constants

- The assembler provides some instructions for loading values into registers.

- Absolute constants
- `LDR Rn, =<constant>`
  `LDR Rn, =label`
  - Pseudo instruction
  - Assembler will use optimal sequence to generate constants into specified registers (one of `MOV`, `MVN`, or an `LDR` from a literal pool).
  - Can load to the PC, causing a branch
  - Use for absolute addressing and references outside the current section (resulting in position-dependent code)
  - Constant determined at assembly or link time

Instructions for Loading Constants

- The assembler provides some instructions for loading values into registers.

- PC- or register-relative constants
- `ADR Rn, label`
  - Add or subtract an immediate value to or from the PC to generate the address of the label into the specified register, using one instruction.
  - `ADRL` pseudo instruction uses two instructions, giving a better range.
  - Can be used to generate addresses for position-independent code (but only if in same code section)
  - Constant determined at run time

LDR= Examples

- The following examples show how the **LDR=** pseudo-instruction makes code more readable, portable, and flexible.

Code

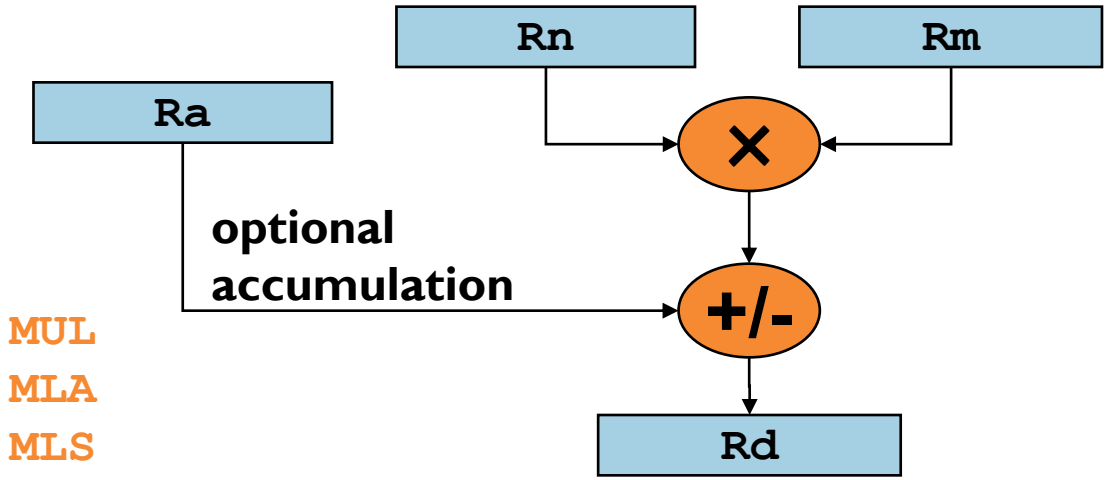
```
LDR r0, =0x2543

LDR r0, =0xFFFF43FF

LDR r0, =0xFFFFF5
```

Disassembly

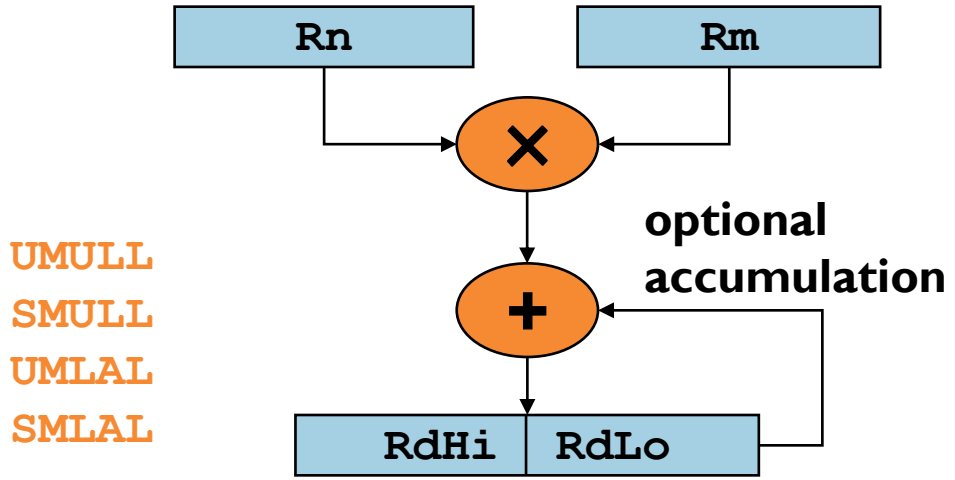
```
MOV r0, #0x2543

MVN r0, #0xBC00

LDR r0, [pc, #xx]
...
DCD 0xFFFFF5
```

Multiply/Divide

- **32-bit multiplication**



- **64-bit multiplication**



  - Examples:
    - `MLA r0, r1, r2, r3 ; r0 = r3 + (r1 * r2)`
    - `[U|S]MULL r4, r5, r2, r3 ; r5:r4 = r2 * r3`

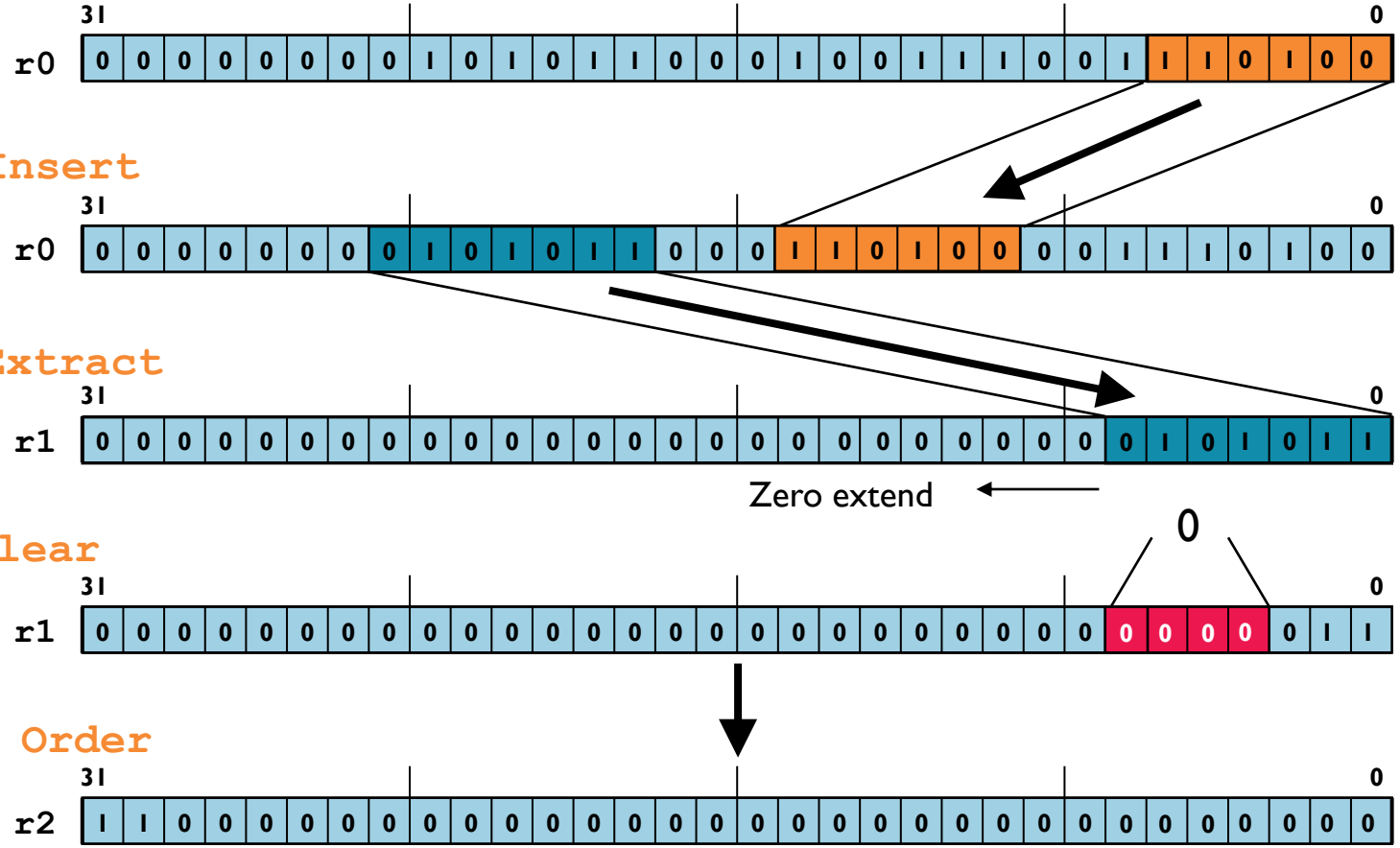
- **Division:**

Optional in 7-A

  - `SDIV r0, r1, r2 ; signed: r0 = r1 / r2`
  - `UDIV r0, r1, r2 ; unsigned: r0 = r1 / r2`

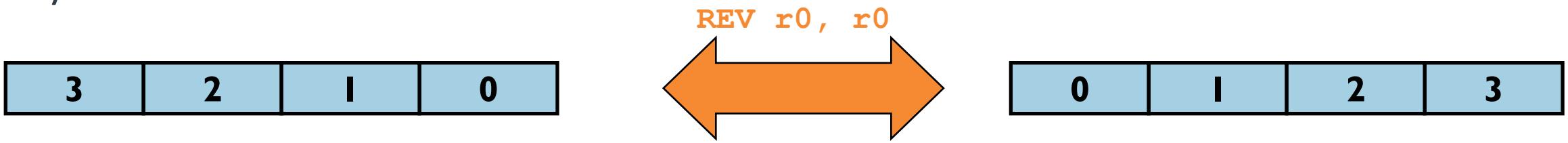
Bit Manipulation Instructions

```
BFI r0, r0, #9, #6    ; Bit Field Insert

UBFX r1, r0, #18, #7  ; Bit Field Extract

BFC r1, #3, #4        ; Bit Field Clear

RBIT r2, r1           ; Reverse Bit Order
```



Byte Reversal

- Byte reversal instructions



- `REV{cond} Rd, Rm` Reverses the bytes in a word
- `REV16{cond} Rd, Rm` Reverses the bytes in each halfword
- `REVSH{cond} Rd, Rm` Reverses the bottom two bytes, and sign extends the result to 32 bits

Pre-V6

```
EOR     r1, r0, r0, ROR #16
BIC     r1, r1, #0xFF0000
MOV     r0, r0, ROR #8
EOR     r0, r0, r1, LSR #8
```

V6 and later

```
REV     r0, r0
```


arm