Agenda

- **Introduction**

  Architecture

  Programmers Model

  Instruction Set

History of ARM

- **ARM (Acorn RISC Machine) started as a new, powerful, CPU design for the replacement of the 8-bit 6502 in Acorn Computers (**Cambridge, UK, 1985**)**
- **First models had only a 26-bit program counter, limiting the memory space to 64 MB (**not too much by today standards, but a lot at that time**).**
- **1990 spin-off: ARM renamed Advanced RISC Machines**
- **ARM now focuses on Embedded CPU cores**
  - IP licensing: Almost every silicon manufacturer sells some microcontroller with an ARM core. Some even compete with their own designs.
  - Processing power with low current consumption
    - Good MIPS/Watt figure
    - Ideal for portable devices
  - Compact memories: 16-bit opcodes (Thumb)
- **New cores with added features**
  - Harvard architecture (ARM9, ARM11, Cortex)
  - Floating point arithmetic
  - Vector computing (VFP, NEON)
  - Java language (Jazelle)

ARM

RISC Architecture vs CISC Architecture

RISC	CISC
It stands for Reduced Instruction Set Computer.	It stands for Complex Instruction Set Computer.
It is a microprocessor architecture that uses small instruction set of uniform length. These simple instructions are executed in one clock cycle.	This offers hundreds of instructions of different sizes to the users. This architecture has a set of special purpose circuits which help execute the instructions at a high speed.
These chips are relatively simple to design.	These chips are complex to design.
They are inexpensive.	They are relatively expensive.
Examples of RISC chips include SPARC, POWER PC.	Examples of CISC include Intel architecture, AMD.

RISC Architecture vs CISC Architecture

RISC	CISC
It has less number of instructions.	It has more number of instructions.
It has fixed-length encodings for instructions.	It has variable-length encodings of instructions.
Simple addressing formats are supported.	The instructions interact with memory using complex addressing modes.
It doesn't support arrays.	It has a large number of instructions. It supports arrays.
It doesn't use condition codes.	Condition codes are used.
Registers are used for procedure arguments and return addresses.	The stack is used for procedure arguments and return addresses.

ARM®

Facts

- **32-bit CPU**
- **3-operand instructions (typical): ADD Rd,Rn,Operand2**
- **RISC design…**
  - Few, simple, instructions
  - Load/store architecture (instructions operate on registers, not memory)
  - Large register set
  - Pipelined execution
- **… Although with some CISC touches…**
  - *Multiplication* and *Load/Store Multiple* are complex instructions (many cycles longer than regular, RISC, instructions)
- **… And some very specific details**
  - No stack. Link register instead
  - PC as a regular register
  - Conditional execution of all instructions
  - Flags altered or not by data processing instructions (selectable)
  - Concurrent shifts/rotations (at the same time of other processing)
  - …

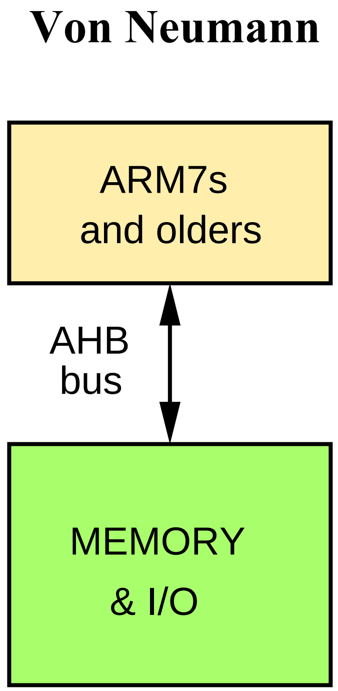
ARM

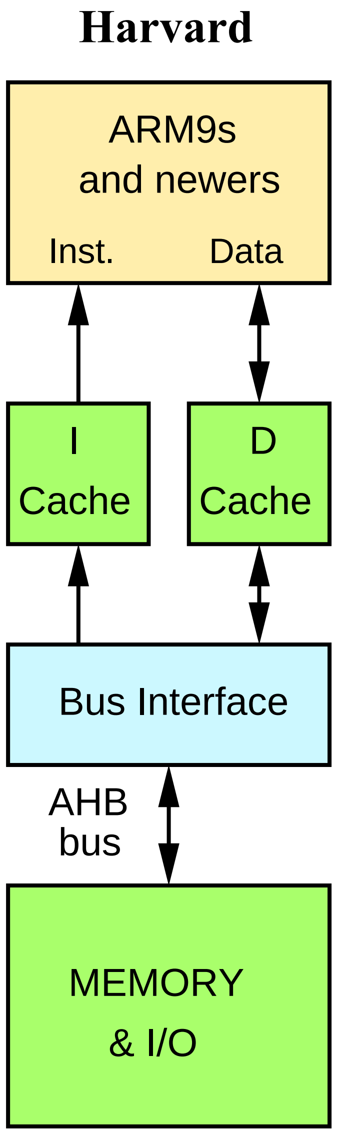
Agenda

- Introduction
- **Architecture**
- Programmers Model
- Instruction Set

ARM

Topologies

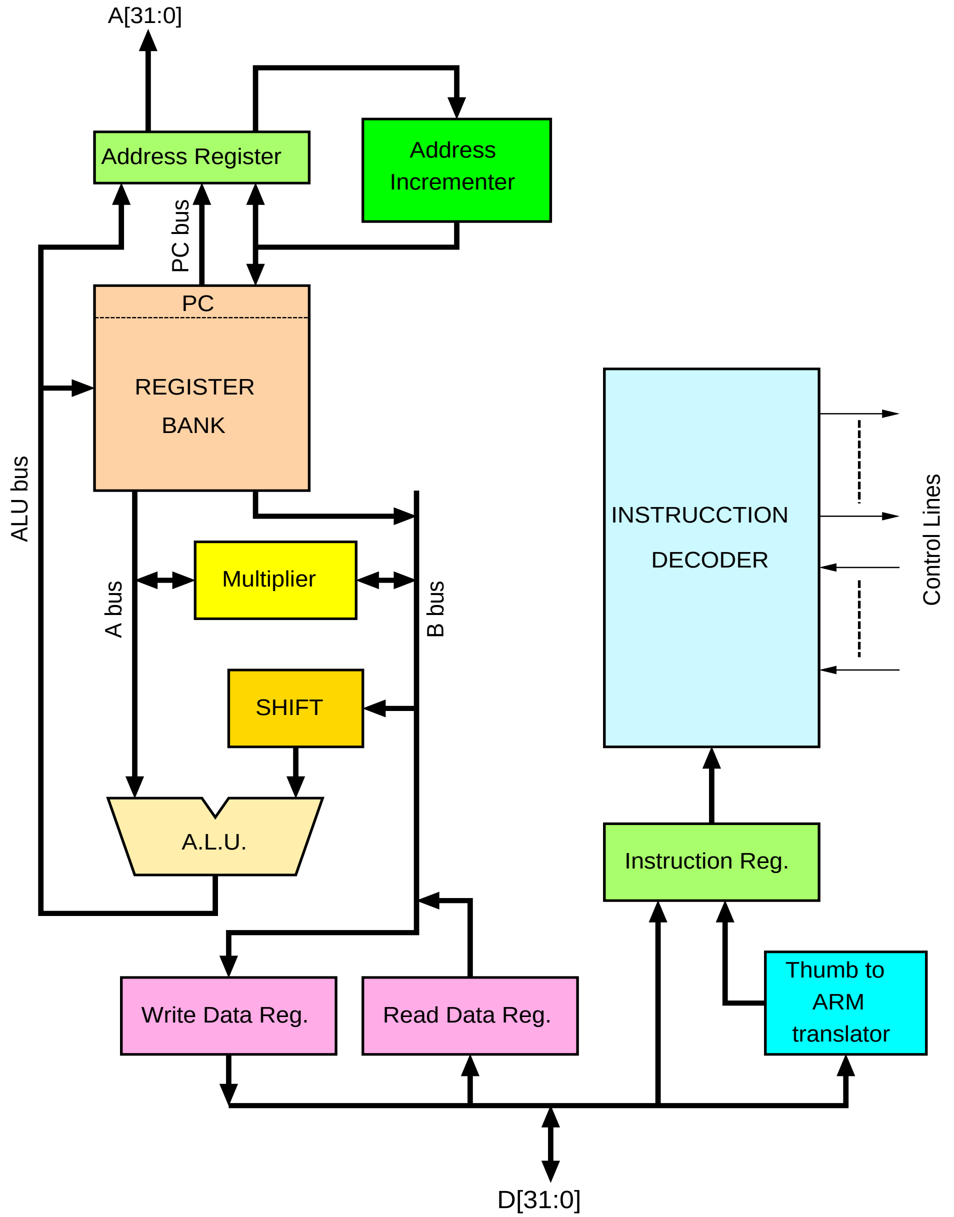




Memory-mapped I/O:

- No specific instructions for I/O (use Load/Store instr. instead)
- Peripheral’s registers at some memory addresses

ARM7TDMI (**Advanced RISC Machines 7 Thumb Debug Multiplier ICE)**
Block Diagram



Agenda

Introduction

Architecture

- **Programmers Model**

Instruction Set

ARM

Data Sizes and Instruction Sets

- **The ARM is a 32-bit architecture.**

- **When used in relation to the ARM:**
  - **Byte** means 8 bits
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)

- **Most ARM's implement two instruction sets**
  - 32-bit **ARM** Instruction Set
  - 16-bit **Thumb** Instruction Set

Processor Modes

- **The ARM has seven operating modes:**
  - **User** : unprivileged mode under which most tasks run
  - **FIQ(Fast Interrupt Request)** : entered when a high priority (fast) interrupt is raised
  - **IRQ (Interrupt Request)** : entered when a low priority (normal) interrupt is raised
  - **SVC** (SuperVisor Call): entered on reset and when a Software Interrupt instruction is executed
  - **Abort** : used to handle memory access violations
  - **Undef** : used to handle undefined instructions
  - **System** : privileged mode using the same registers as user mode

ARM

The Registers

- **ARM has 37 registers all of which are 32-bits long.**
  - 30 general purpose registers
  - 1 dedicated program counter
  - 1 dedicated current program status register
  - 5 dedicated saved program status registers

- **The current processor mode governs which of several banks is accessible. Each mode can access**
  - a particular set of r0-r12 registers
  - a particular r13 (the stack pointer, sp) and r14 (the link register, lr)
  - the program counter, r15 (pc)
  - the current program status register, cpsr

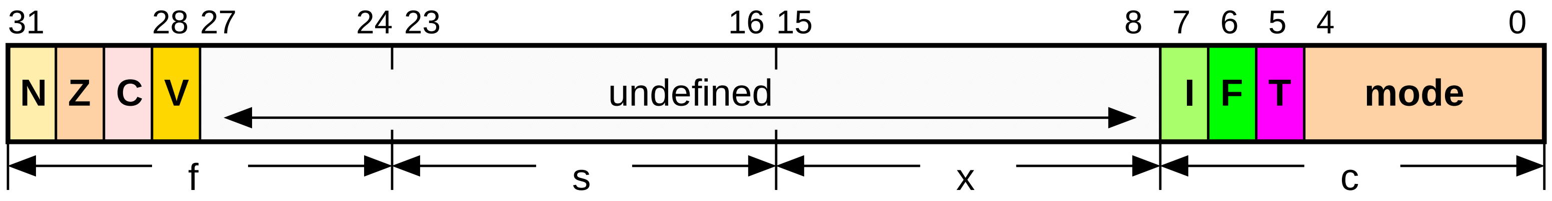
Privileged modes (except System) can also access
  - a particular spsr (saved program status register)

ARM®

Special Registers

- **Special function registers:**
  - **PC** (R15): Program Counter. Any instruction with PC as its destination register is a program branch
  - **LR** (R14): Link Register. Saves a copy of PC when executing the BL instruction (subroutine call) or when jumping to an exception or interrupt routine
    - It is copied back to PC on the return from those routines
  - **SP** (R13): Stack Pointer. There is **no stack** in the ARM architecture. Even so, R13 is usually reserved as a pointer for the program-managed stack
  - **CPSR** : Current Program Status Register. Holds the visible status register
  - **SPSR** : Saved Program Status Register. Holds a copy of the previous status register while executing exception or interrupt routines
    - It is copied back to CPSR on the return from the exception or interrupt
    - No SPSR available in User or System modes

Program Status Registers



- **Condition code flags**
  - N = **N**egative result from ALU
  - Z = **Z**ero result from ALU
  - C = ALU operation **C**arried out
  - V = ALU operation o**V**erflowed

- **Mode bits**

10000	**User**
10001	**FIQ**
10010	**IRQ**
10011	**Supervisor**
10111	**Abort**
11011	**Undefined**
11111	**System**

Interrupt Disable bits.

I = 1: Disables the IRQ.

F = 1: Disables the FIQ.

T Bit **(Arch. with Thumb mode only)**

T = 0: Processor in ARM state

T = 1: Processor in Thumb state

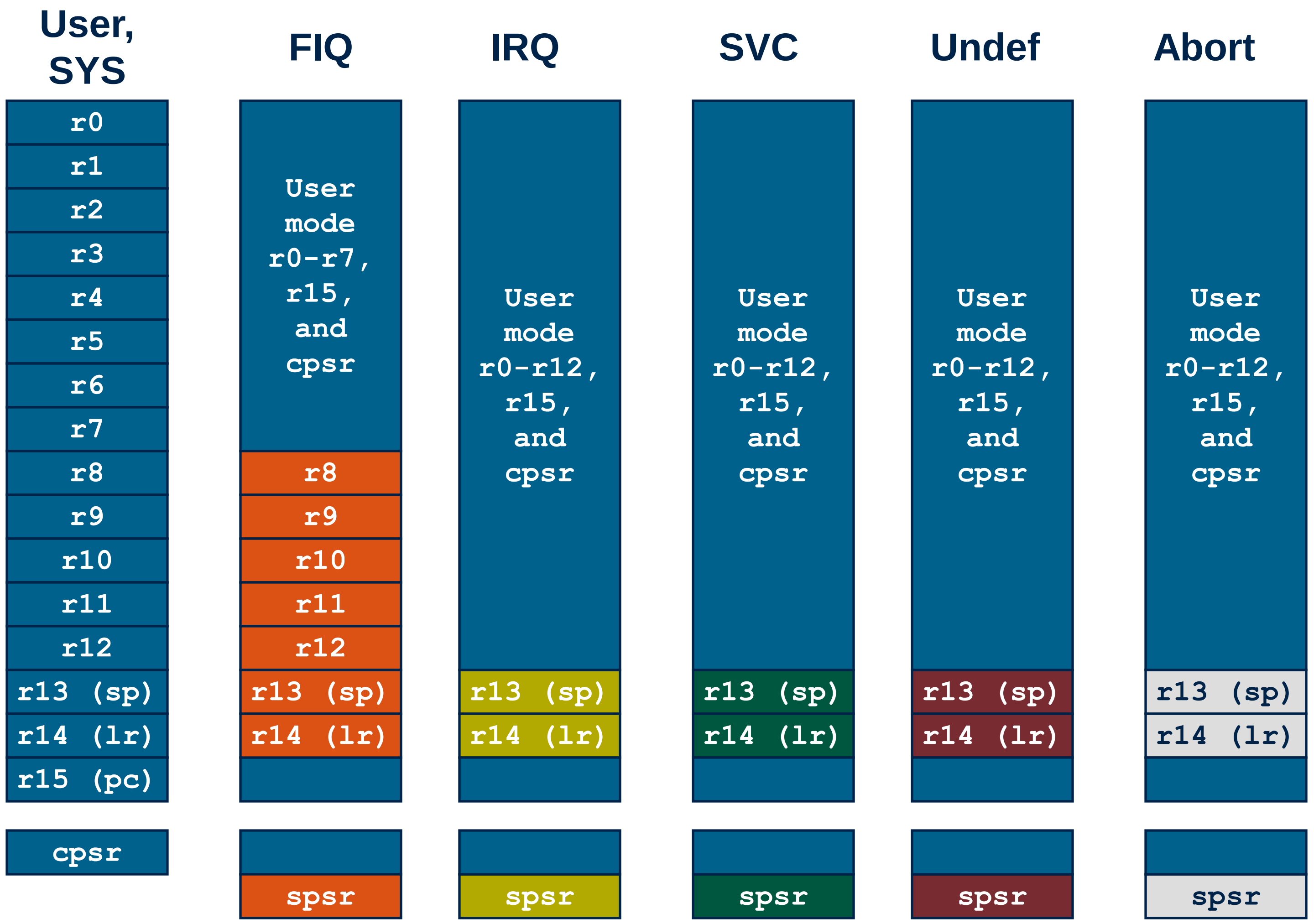
Never change T directly (use BX instead)

Changing T in CPSR will lead to unexpected behavior due to pipelining

Tip: Don’t change undefined bits.

This allows for code compatibility with newer ARM processors

Register Organization Summary



Note: System mode uses the User mode register set

ARM

Addressing Modes

- **The addressing modes are most conveniently classified with respect to the type of instruction.**
  - Load/Store Addressing
  - Data Processing Instruction Addressing
  - Branch Instruction Addressing
  - Load/Store Multiple Addressing

ARM

Load/Store Addressing

Memory is addressed by a register and an offset.

```
LDR  R0, [R1]      @ Load the word data from mem[R1]   -> R0
```

Three ways to specify offsets:

- **Immediate**

```
STRB  R0, [R1, #4]         ; Load the Byte data from mem[R1+4]   -> R0
```

- **Register**

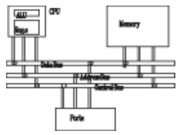
```
LDR  R0, [R1, R2]          ; Load the word data from mem[R1+R2] -> R0
```

- **Scaled register**

```
LDR  R0, [R1, R2, LSL #2]  ; Load the word data from mem[R1+4*R2]
                                                           to R0
```

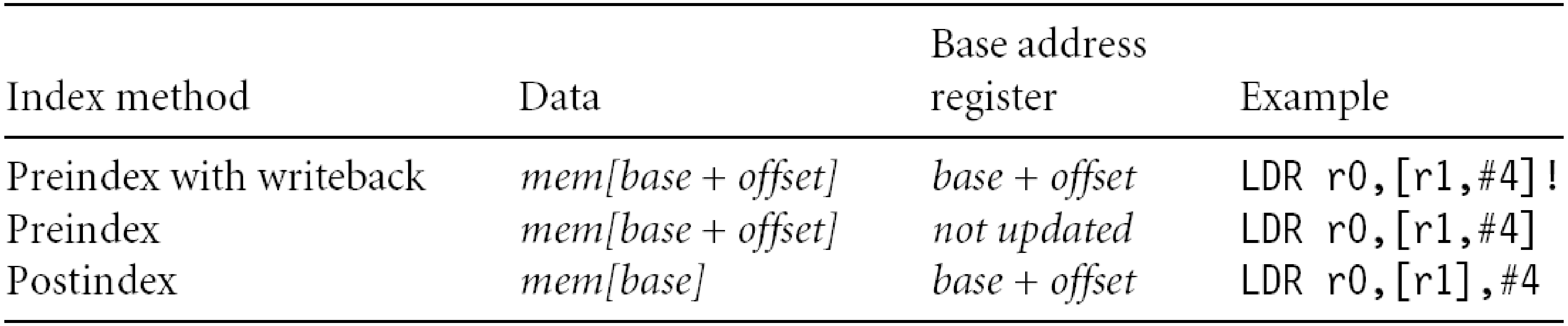
NB: ‘;’ is used to comment a line

Addressing modes



- Pre-index addressing
  - without a writeback (**STRB R0, [R1, #4]**)
  - with a writeback (**STRB R0, [R1, #4]!**)

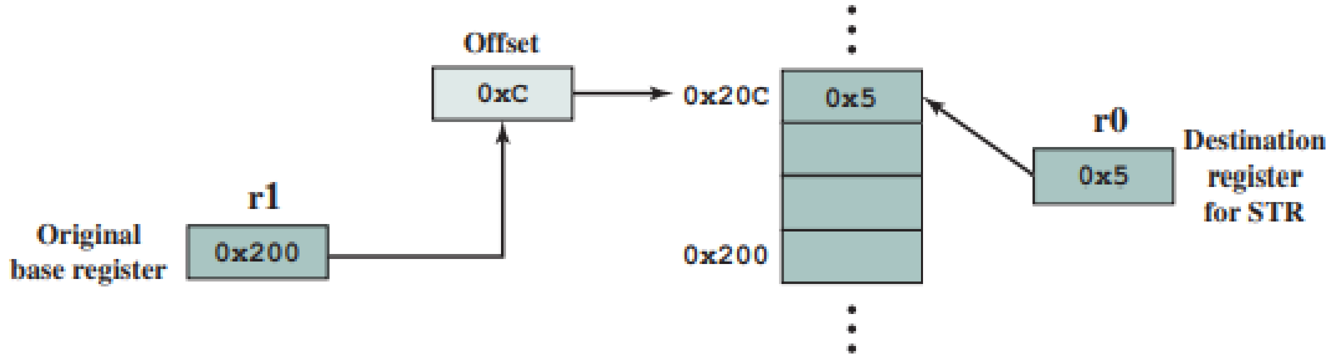
- Post-index addressing (**STRB R0, [R1], #4**)
  - calculation after accessing with a writeback



Index method	Data	Base address register	Example
Preindex with writeback	*mem[base + offset]*	*base + offset*	`LDR r0,[r1,#4]!`
Preindex	*mem[base + offset]*	*not updated*	`LDR r0,[r1,#4]`
Postindex	*mem[base]*	*base + offset*	`LDR r0,[r1],#4`

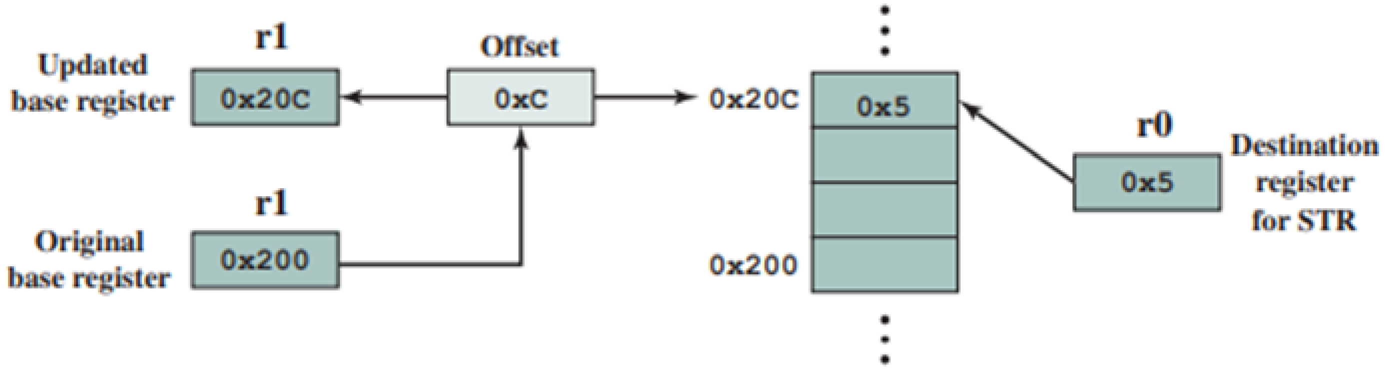
Addressing modes

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



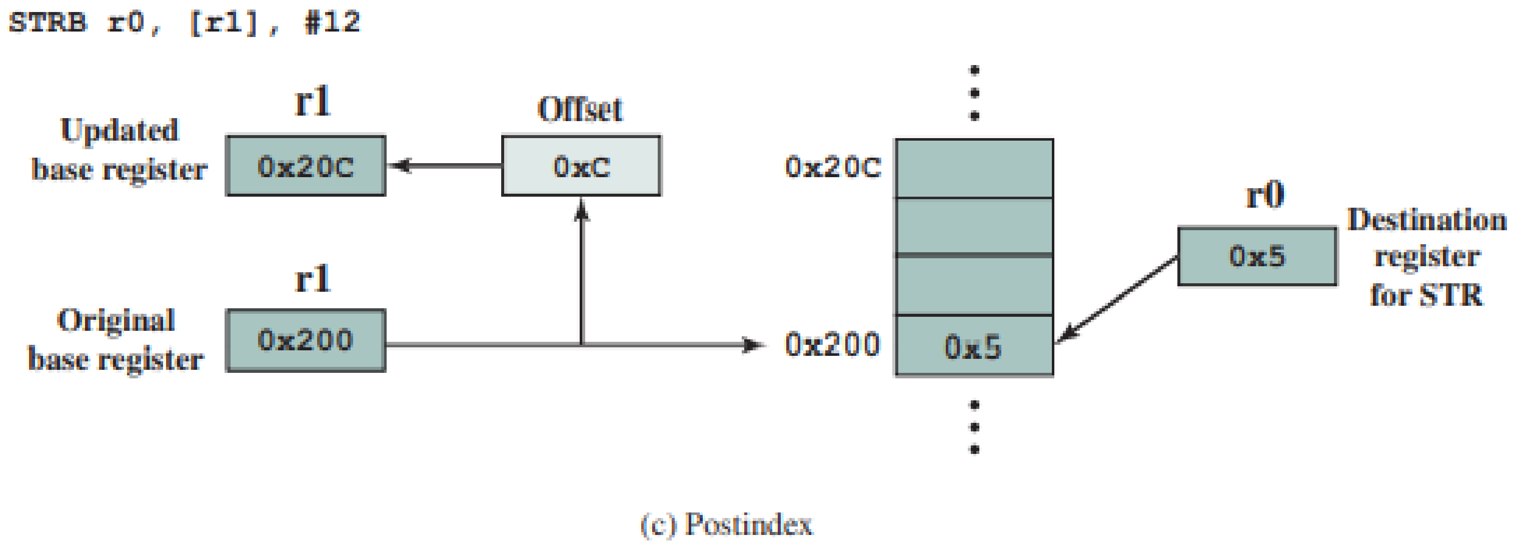
(a) Pre-indexing without writeback

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



(b) Pre-indexing with writeback

Addressing modes



(c) Post-indexing

Data Processing Addressing

Data processing instructions use either register to address or a mixture of register and immediate addressing.

ADD R0, R0, R0, LSL #2	$\longrightarrow$	**Scaled Register addressing**
ADD R0, R1, R2	$\longrightarrow$	**Register**
ADD R3, R3, #1	$\longrightarrow$	**Immediate**

Branch Instruction Addressing

The only form of addressing for branch instructions is immediate addressing.

The branch instruction contains a 24-bit value.

Load/Store Multiple Addressing

Load Multiple instructions load a subset (possibly all) of the general- purpose registers from memory.

Store Multiple instructions store a subset (possibly all) of the general-purpose registers to memory.

The list of registers for the load or store is specified in a 16-bit field in the instruction with each bit corresponding to one of the 16 registers.

Four addressing modes are used : increment after, increment before, decrement after, and decrement before.

ARM

Load/Store Multiple Addressing

Load/store a subset of the general-purpose registers from/to memory.

`LDM<suffix>`	`load multiple registers`
`STM<suffix>`	`store multiple registers`

suffix	meaning
IA	increase after
IB	increase before
DA	decrease after
DB	decrease before

ARM

Load/Store Multiple Addressing

```
LDMxx r10, {r0, r1, r4}
STMxx r10, {r0, r1, r4}
```

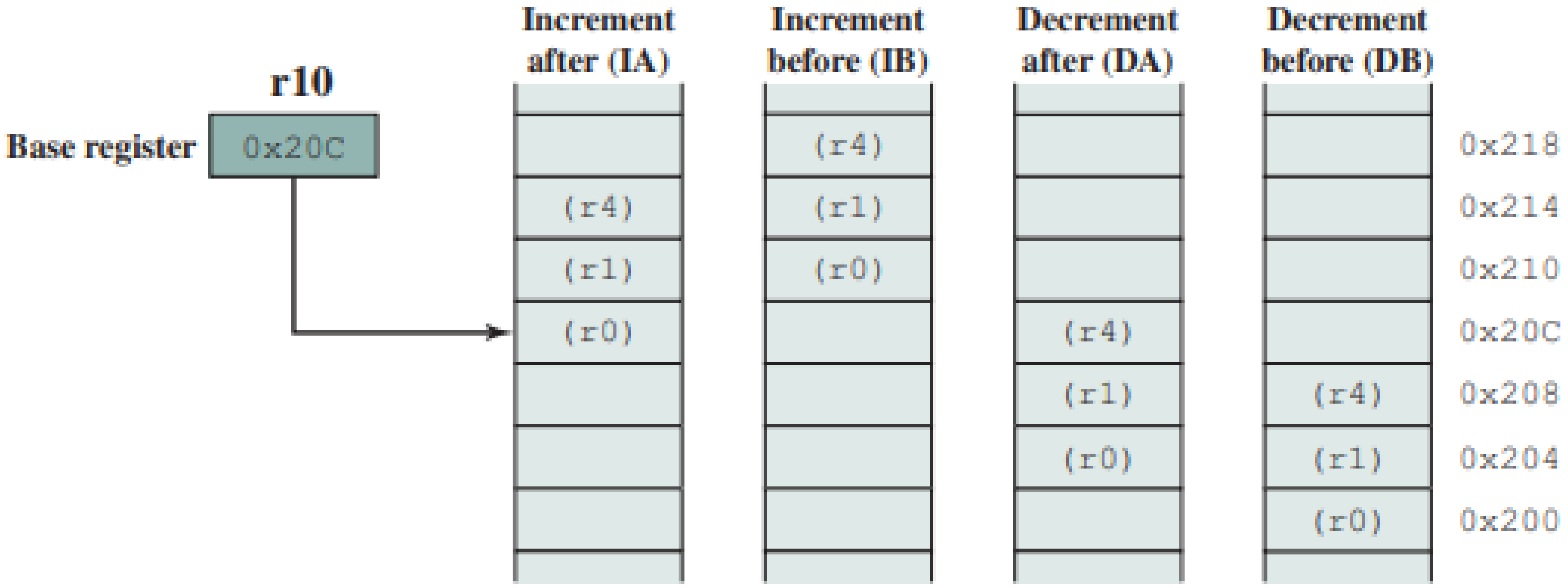


Figure 13.4 ARM Load/Store Multiple Addressing

ARM Assembly Language Programming

- Outline:
  - the ARM instruction set
  - writing simple programs
  - examples

☞ hands-on: writing simple ARM assembly programs

ARM

The ARM instruction set

- ARM instructions fall into three categories:
  - data processing instructions
    - operate on values in registers
  - data transfer instructions
    - move values between memory and registers
  - control flow instructions
    - change the program counter (PC)

The ARM instruction set

- ARM instructions fall into three categories:
  - **data processing instructions**
    - operate on values in registers
  - data transfer instructions
    - move values between memory and registers
  - control flow instructions
    - change the program counter (PC)

ARM®

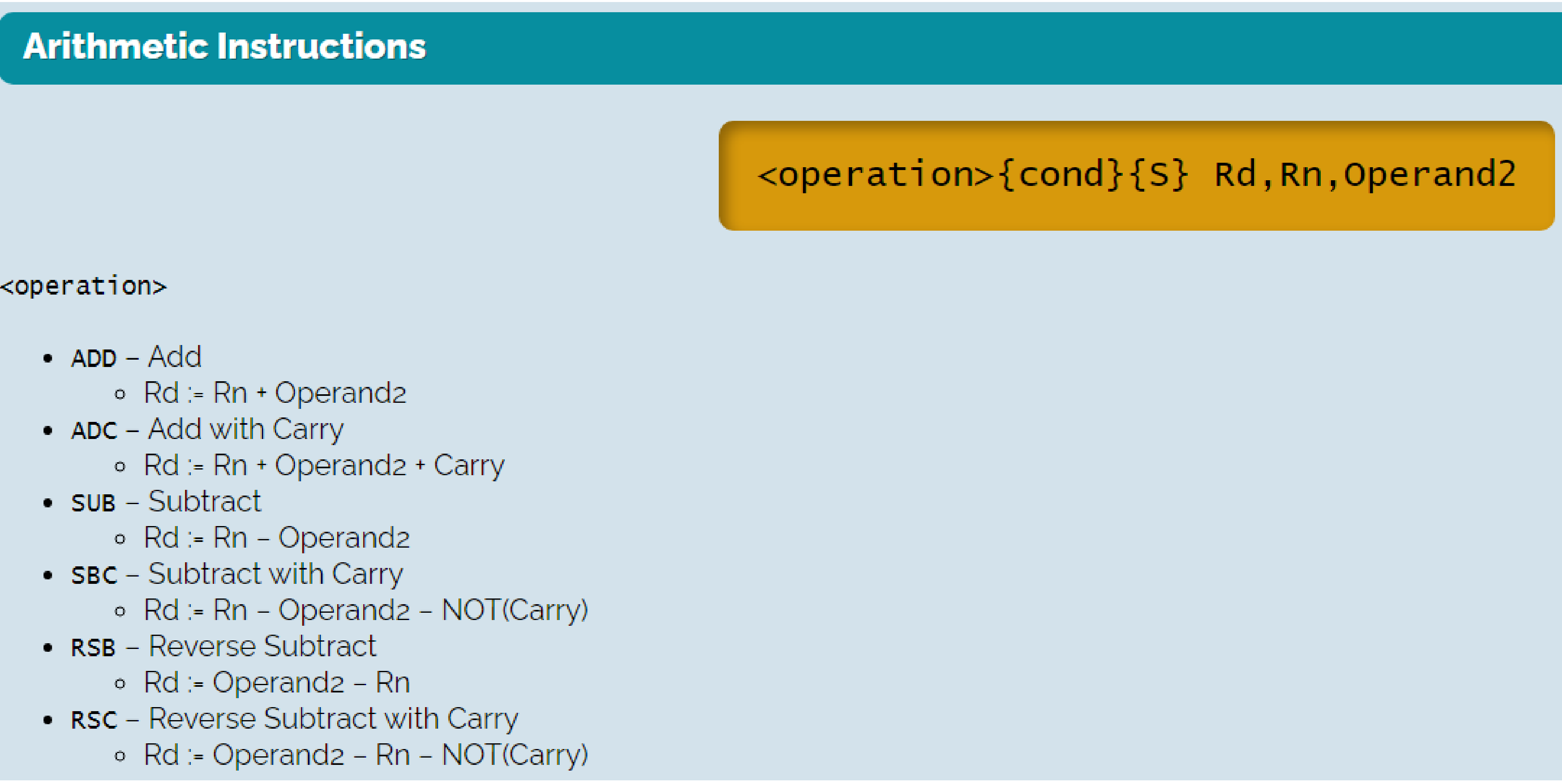
Data processing instructions

- All operands are 32-bits wide and either:
  - come from registers, or
  - are literals (‘immediate’ values) specified in the instruction
- The result, if any, is 32-bits wide and goes into a register
  - except long multiplies generate 64-bit results
- All operand and result registers are independently specified

ARM®

Data processing instructions

1. Arithmetic Instructions :



Arithmetic Instructions

```
<operation>{cond}{S} Rd,Rn,Operand2
```

`<operation>`

- `ADD` – Add
  - Rd := Rn + Operand2
- `ADC` – Add with Carry
  - Rd := Rn + Operand2 + Carry
- `SUB` – Subtract
  - Rd := Rn – Operand2
- `SBC` – Subtract with Carry
  - Rd := Rn – Operand2 – NOT(Carry)
- `RSB` – Reverse Subtract
  - Rd := Operand2 – Rn
- `RSC` – Reverse Subtract with Carry
  - Rd := Operand2 – Rn – NOT(Carry)

ARM®

Data processing instructions

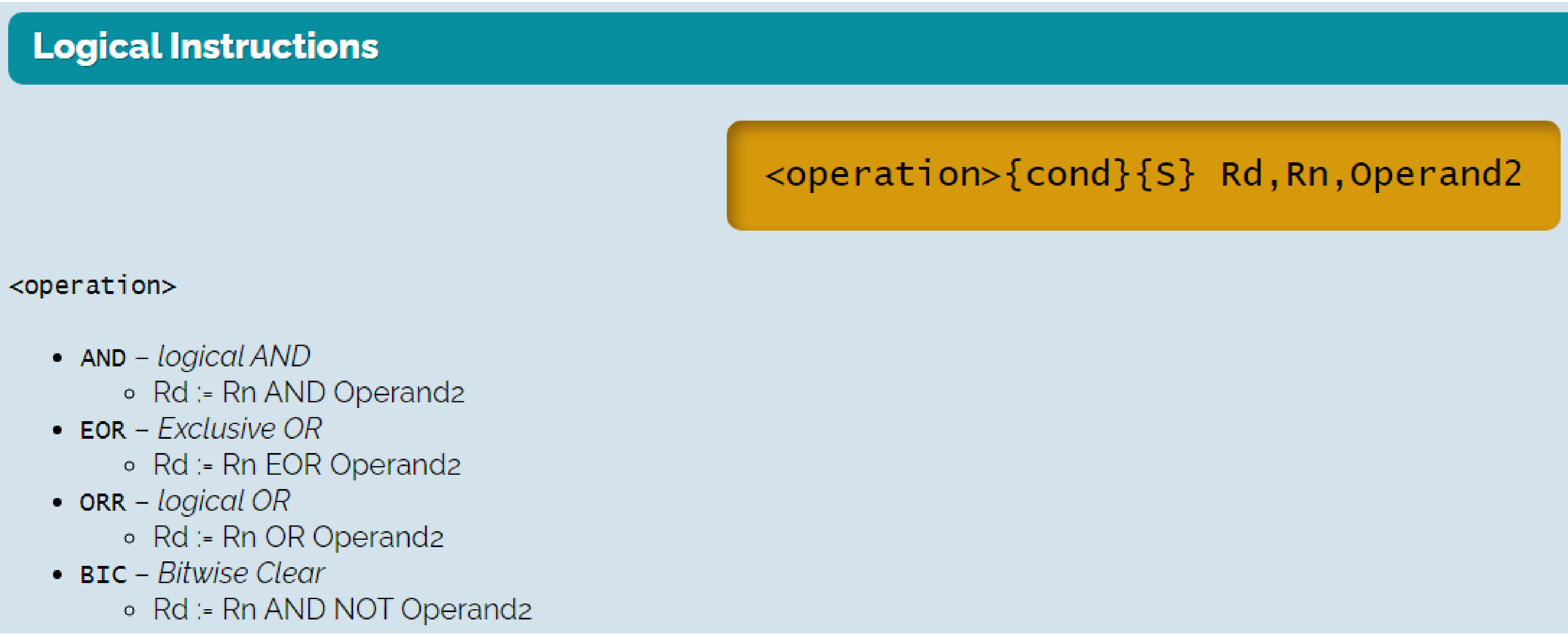
- Arithmetic operations:

```
ADD   r0, r1, r2       ; r0 := r1 + r2
ADC   r0, r1, r2       ; r0 := r1 + r2 + C
SUB   r0, r1, r2       ; r0 := r1 - r2
SBC   r0, r1, r2       ; r0 := r1 - r2 + C - 1
RSB   r0, r1, r2       ; r0 := r2 - r1
RSC   r0, r1, r2       ; r0 := r2 - r1 + C - 1
```

  - C is the C bit in the CPSR
  - the operation may be viewed as unsigned or 2’s complement signed

Data processing instructions

2. Logical Instructions :



Logical Instructions

```
<operation>{cond}{S} Rd,Rn,Operand2
```

`<operation>`

- `AND` – *logical AND*
  - Rd := Rn AND Operand2
- `EOR` – *Exclusive OR*
  - Rd := Rn EOR Operand2
- `ORR` – *logical OR*
  - Rd := Rn OR Operand2
- `BIC` – *Bitwise Clear*
  - Rd := Rn AND NOT Operand2

ARM

Data processing instructions

- Bit-wise logical operations:

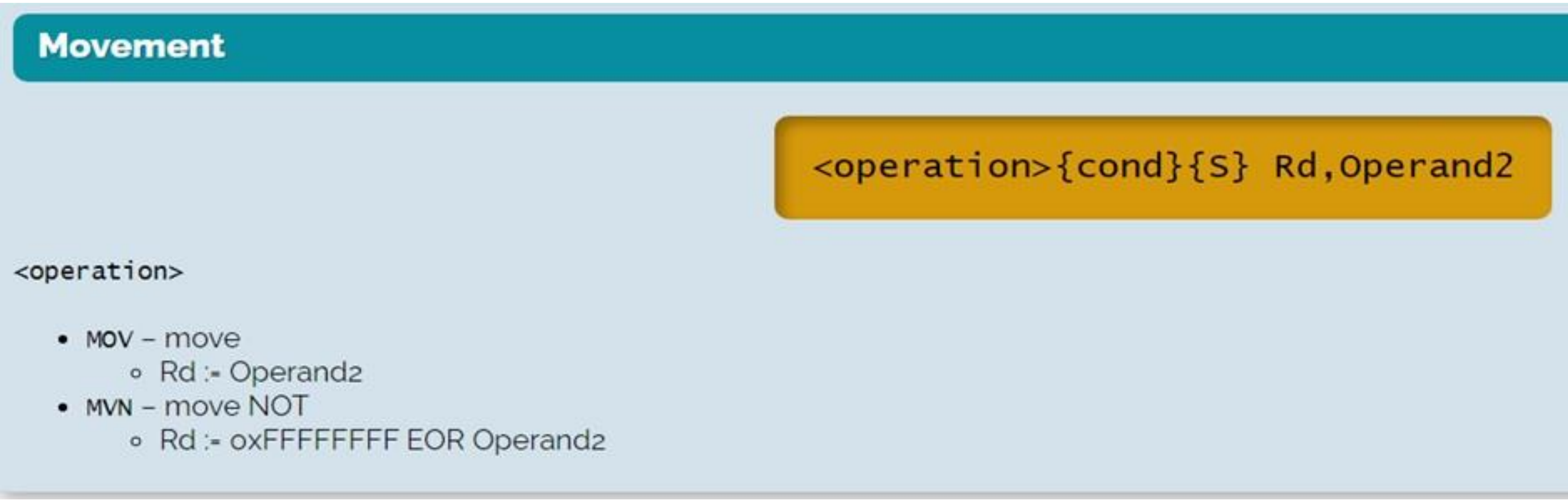
```
AND    r0, r1, r2      @ r0 := r1 and  r2

ORR    r0, r1, r2      @ r0 := r1 or  r2
EOR    r0, r1, r2      @ r0 := r1 xor  r2

BIC    r0, r1, r2      @ r0 := r1 and  not  r2
```

  - the specified Boolean logic operation is performed on each bit from 0 to 31
  - BIC stands for ‘bit clear’
    - each ‘1’ in r2 clears the corresponding bit in r1

ARM

Data processing instructions

3. Move Instructions :



Movement

```
<operation>{cond}{S} Rd,Operand2
```

`<operation>`

- `MOV` – move
  - Rd := Operand2
- `MVN` – move NOT
  - Rd := 0xFFFFFFFF EOR Operand2

Data processing instructions

- Register movement operations:

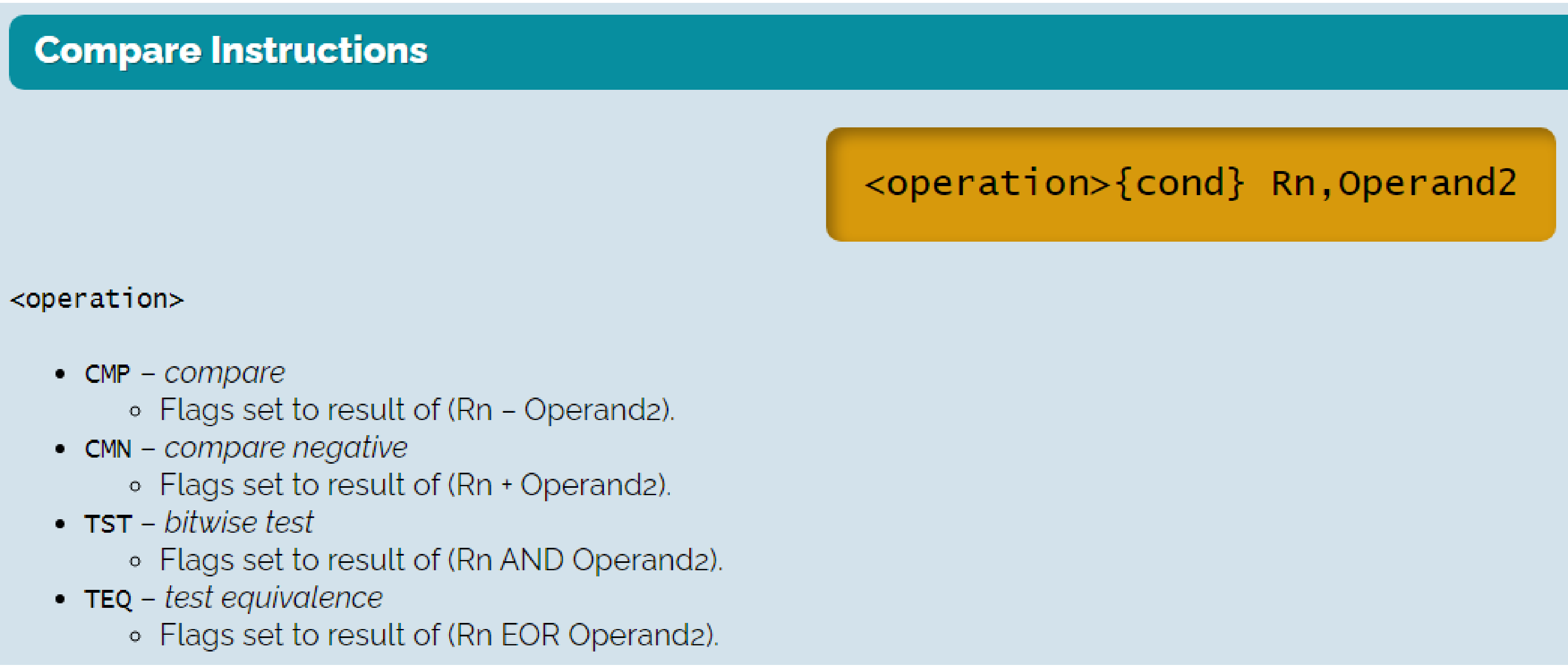
```
MOV   r0, r2          ; r0 := r2
MVN   r0, r2          ; r0 := not r2
```

  - MVN stands for ‘move negated’
  - there is no first operand (r1) specified as these are unary operations

ARM

Data processing instructions

4. **Compare Instructions :**



Compare Instructions

```
<operation>{cond} Rn,Operand2
```

`<operation>`

- `CMP` – *compare*
  - Flags set to result of (Rn – Operand2).
- `CMN` – *compare negative*
  - Flags set to result of (Rn + Operand2).
- `TST` – *bitwise test*
  - Flags set to result of (Rn AND Operand2).
- `TEQ` – *test equivalence*
  - Flags set to result of (Rn EOR Operand2).

ARM

Data processing instructions

- Comparison operations:

```
CMP   r1, r2          ; set cc on r1 - r2


CMN   r1, r2          ; set cc on r1 + r2


TST   r1, r2          ; set cc on r1 and  r2
TEQ   r1, r2          ; set cc on r1 xor  r2
```

- These instructions just affect **the condition codes (N, Z, C, V)** in the CPSR

ARM

Data processing instructions

- Immediate operands
  - The second source operand (r2) may be replaced by a constant:

```
ADD    r3, r3, #1      ; r3 := r3 + 1
AND    r8, r7, #&ff    ; r8 := r7
```

  - # indicates an immediate value
    - & indicates hexadecimal notation
    - C-style notation (#0xff) is also supported
  - Allowed immediate values are (in general): (0 □ 255) x 2^{2n}

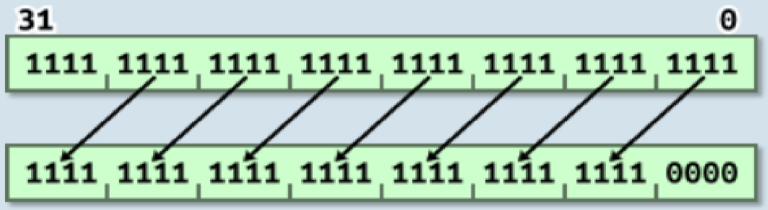
ARM

Data processing instructions

5. Shift and Rotate Instructions :

LSL – Logical Shift Left

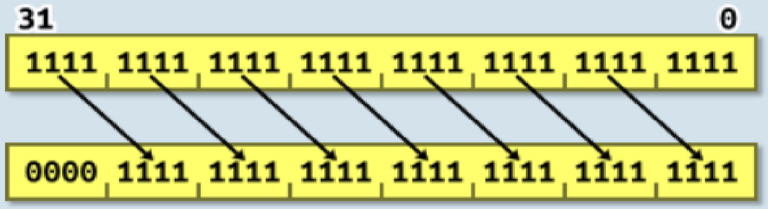
Example: Logical Shift Left by 4.



Equivalent to << in C.

LSR – Logical Shift Right

Example: Logical Shift Right by 4.



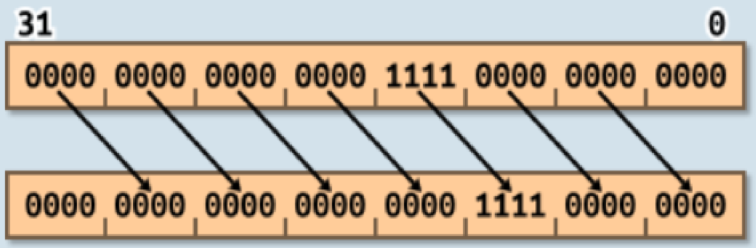
Equivalent to >> in C. i.e. unsigned division by a power of 2.

Data processing instructions

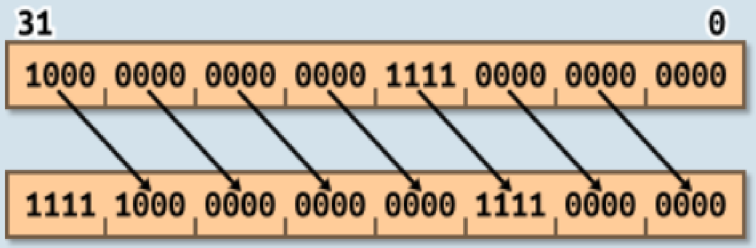
5. Shift and Rotate Instructions :

ASR – Arithmetic Shift Right

Example: Arithmetic Shift Right by 4, positive value.



Example: Arithmetic Shift Right by 4, negative value.



Equivalent to >> in C. i.e. signed division by a power of 2.

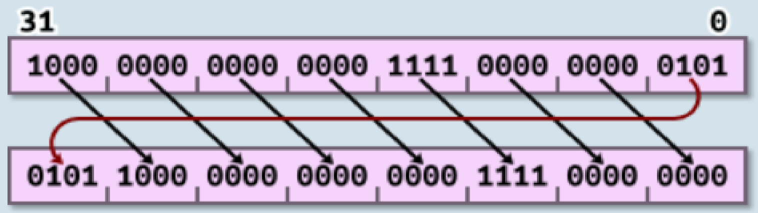
ARM

Data processing instructions

5. Shift and Rotate Instructions :

ROR – Rotate Right

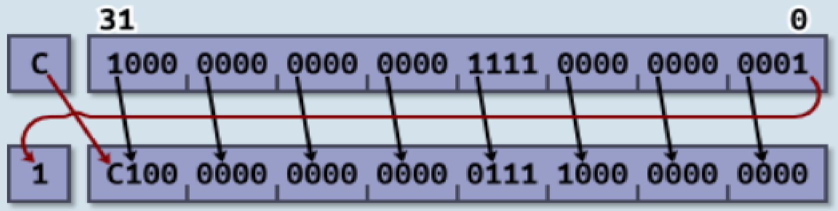
Example: Rotate Right by 4.



Bit rotate with wrap-around.

RRX – Rotate Right Extended

Example: Rotate Right Extended.



33-bit rotate with wrap-around through carry bit.

ARM

Data processing instructions

- Shifted register operands
  - the second source operand may be shifted
    - by a constant number of bits positions:

```
ADD    r3,    r2,    r1,    LSL #3    @   r3   :=    r2+r1<<3
```

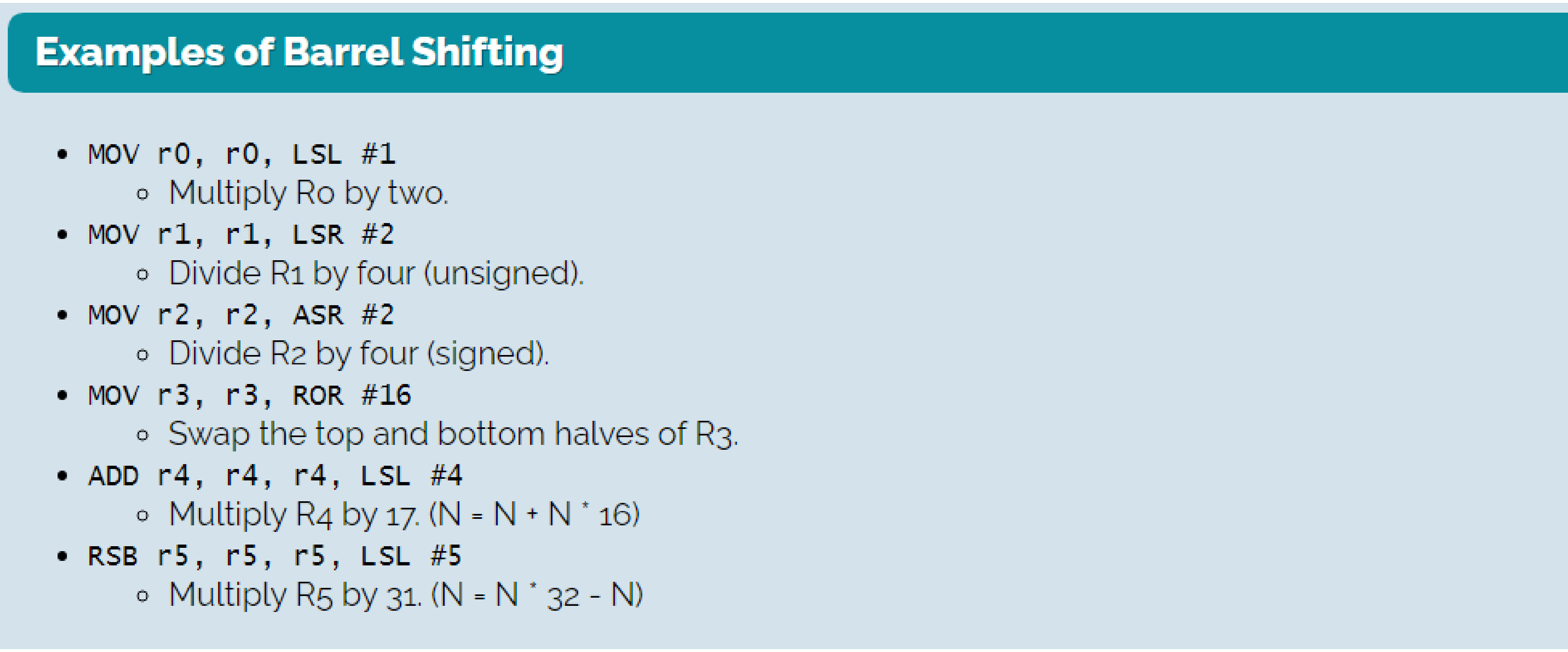
  - or by a register-specified number of bits:

```
ADD    r5,    r5,    r3,    LSL r2    @    r5   +=   r3<<r2
```

    - LSL, LSR mean ‘logical shift left’, ‘logical shift right’
    - ASL, ASR mean ‘arithmetic shift left’, ‘ …right’
    - ROR means ‘rotate right’
    - RRX means ‘rotate right extended’ by 1 bit

Data processing instructions

5. Shift and Rotate Instructions :



Examples of Barrel Shifting

- `MOV r0, r0, LSL #1`
  - Multiply R0 by two.
- `MOV r1, r1, LSR #2`
  - Divide R1 by four (unsigned).
- `MOV r2, r2, ASR #2`
  - Divide R2 by four (signed).
- `MOV r3, r3, ROR #16`
  - Swap the top and bottom halves of R3.
- `ADD r4, r4, r4, LSL #4`
  - Multiply R4 by 17. (N = N + N * 16)
- `RSB r5, r5, r5, LSL #5`
  - Multiply R5 by 31. (N = N * 32 - N)

ARM®

Data processing instructions

- Setting the condition codes
  - All data processing instructions **may** set the condition codes.
    - the comparison operations always do so
  - For example, here is code for a 64-bit add:

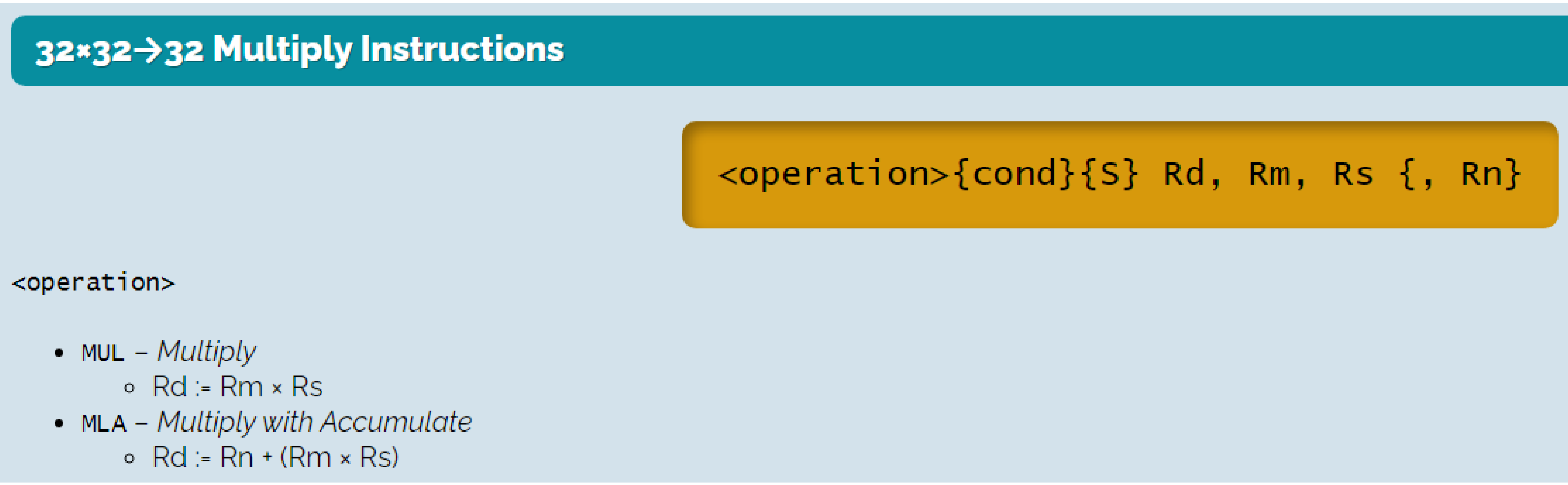
```
ADDS   r2, r2, r0        @ 32-bit carry-out -> C
ADC    r3, r3, r1        @ added into top 32 bits
```

    - S means ‘Set condition codes’
  - The C flag comes from: the adder in arithmetic operations
    the shifter in logical operations
  - the primary use of the condition codes is in control flow – see later

ARM

Data processing instructions

6. Multiplication Instructions :



32×32→32 Multiply Instructions

```
<operation>{cond}{S} Rd, Rm, Rs {, Rn}
```

`<operation>`

- `MUL` – *Multiply*
  - Rd := Rm × Rs
- `MLA` – *Multiply with Accumulate*
  - Rd := Rn + (Rm × Rs)

ARM

Data processing instructions

- Multiplication
  - ARM has special multiplication instructions

    MUL r4, r3, r2 ; r4 := $(r3 \times r2)_{[31:0]}$

    – only the bottom 32 bits are returned

    – immediate operands are not supported

    – multiplication by a constant is usually best done with a short series of adds and subtracts with shifts
  - There is also a multiply-accumulate form:

    MLA r4, r3, r2, r1 ; r4 := $(r3 \times r2+r1)_{[31:0]}$
  - 64-bit result forms are supported too

ARM

The ARM instruction set

- ARM instructions fall into three categories:
  - data processing instructions
    - operate on values in registers
  - **data transfer instructions**
    - move values between memory and registers
  - control flow instructions
    - change the program counter (PC)

Data transfer instructions

- The ARM has 3 types of data transfer instructions:
  - single register loads and stores
    - flexible byte, half-word, and word transfers
  - multiple register loads and stores
    - less flexible, multiple words, higher transfer rate
  - single register - memory swap
    - mainly for system use, so ignore it for now

Data transfer instructions

- The ARM has 3 types of data transfer instructions:
  - **single register loads and stores**
    - flexible byte, half-word, and word transfers
  - **multiple register loads and stores**
    - less flexible, multiple words, higher transfer rate
  - single register - memory swap
    - mainly for system use, so ignore it for now

Data transfer instructions

- Addressing memory
  - All ARM data transfer instructions use **register indirect** addressing.
  - Examples of load and store instructions:

```
LDR   r0, [r1]        ; r0 := mem[r1]
STR   r0, [r1]        ; mem[r1] := r0
```

  - Therefore before any data transfer is possible:
    a register must be initialized with an address close to the target

Data transfer instructions

- **Single register loads and stores**
  - The simplest form is just registered indirect:

```
LDR     r0,   [r1]              ;  r0  :=  mem[r1]
```

  - This is a special form of base plus offset:

```
LDR    r0,  [r1,#4]  ;r0  :=  mem[r1+4]
```

    - the offset is within 4 Kbytes
    - auto-indexing/preindexing is also possible:

```
LDR    r0, [r1,#4]!   ; r0 := mem[r1+4]
                      ; r1 := r1  +  4
```

ARM

Data transfer instructions

- **Single register loads and stores (…cntd.)**
  - another form uses post-indexing

```
LDR   r0, [r1],#4    @ r0 := mem[r1]
                     @ r1 := r1 + 4
```

  - finally, a byte or half-word can be loaded instead of a word (with some restrictions):

```
LDRB    r0,   [r1]        @ r0 := mem8[r1]
LDRSH   r0,   [r1]        @ r0 := mem16[r1](signed)
```

  - stores (STR) have the same forms

ARM

Data transfer instructions

- Initializing an address pointer
  - any register can be used for an address
  - the assembler has special ‘pseudo instructions’ to initialize address registers:

```
          ADR    r1, TABLE1       @ r1 points to TABLE1
          ..
                                  @ LABEL
TABLE1
```

– ADR will result in a single ARM instruction

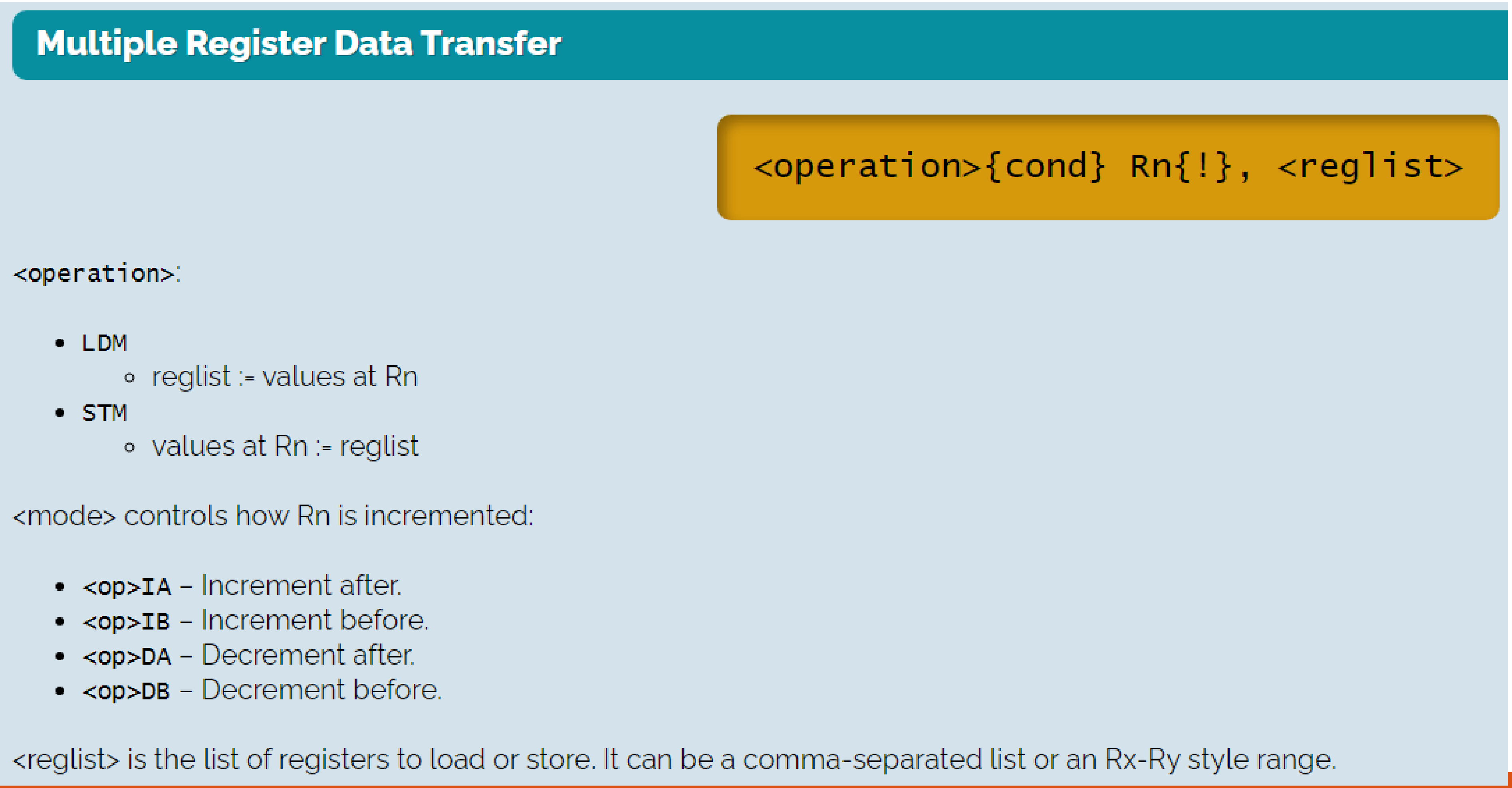
```
ADRL    r1,    TABLE1
```

– ADRL will handle cases that ADR can’t

ARM

Data transfer instructions

❑ **Multiple register loads and stores**



Multiple Register Data Transfer

```
<operation>{cond} Rn{!}, <reglist>
```

`<operation>`:

- `LDM`
  - reglist := values at Rn
- `STM`
  - values at Rn := reglist

<mode> controls how Rn is incremented:

- `<op>IA` – Increment after.
- `<op>IB` – Increment before.
- `<op>DA` – Decrement after.
- `<op>DB` – Decrement before.

<reglist> is the list of registers to load or store. It can be a comma-separated list or an Rx-Ry style range.

ARM

Data transfer instructions

❑ **Multiple register loads and stores**

- ARM also supports instructions that transfer several registers:

```
LDMIA  r1, {r0,r2,r5} ; r0  := mem[r1]

                      ; r2  := mem[r1+4]
                      ; r5  := mem[r1+8]
```

  - the {…} list may contain any or all of r0 - r15
  - the lowest register **always** uses the lowest address, and so on, in increasing order
  - it doesn’t matter how the registers are ordered in {…}

ARM

The ARM instruction set

- ARM instructions fall into three categories:
  - data processing instructions
    - operate on values in registers
  - data transfer instructions
    - move values between memory and registers
  - ➜ **control flow instructions**
    - change the program counter (PC)

Control flow instructions

```
<operation>{cond} <address>
```

- `B` – *Branch*
  - PC := <address>
- `BL` – *Branch with Link*
  - R14 := address of next instruction, PC := <address>

How do we return from the subroutine which `BL` invoked?

```
MOV pc, r14
```

or

`BX r14` (on ARMv4T or later)

Control flow instructions

- Control flow instructions just switch execution around the program:
- normal execution of program is sequential
- branches are used to change this
  - to move forwards or backwards
- Note: data ops and loads can also change the PC!

ARM

Control flow instructions

- Conditional branches
  - sometimes whether or not a branch is taken depends on the condition codes:

```
          MOV    r0, #0          ; initialise  counter
LOOP      ..

          ADD    r0, r0, #1      ; increment  counter
            CMP  r0, #10         ; compare  with  limit

            BNE  LOO             ; repeat  if  not  equal
         ..      P              ; else continue
```

    – here the branch depends on how CMP sets Z

ARM

Branch conditions

Branch	Interpretation	Normal uses
B BAL	Unconditio nal Always	Always take this branch Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC BLO	Carry clear Lower	Arithmetic operation did not give carry-out Unsigned comparison gave lower
BCS BHS	Carry set Higher or same	Arithmetic operation gave carry-out Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

“Missing” instructions

Some familiar mnemonics are not present in the ARM instructions:

NOT	MVN R0, R0	@ R0 := not(R0)
NEG	RSB R0, R0, #0	@ R0 := 0 - R0
RET	MOV PC, LR	@ ‘Leaf’ return
or (e.g.)	LDR PC, [SP], #4	@ Unstack PC
PUSH & POP	STMFDSP!, {register list};	Push
	LDMFDSP!, {register list};	Pop
LSL, etc.	MOV R0, R0, LSL #5	
	MOV R0, R0, ASR R1	

– Note that the shift can be combined with other operations too.