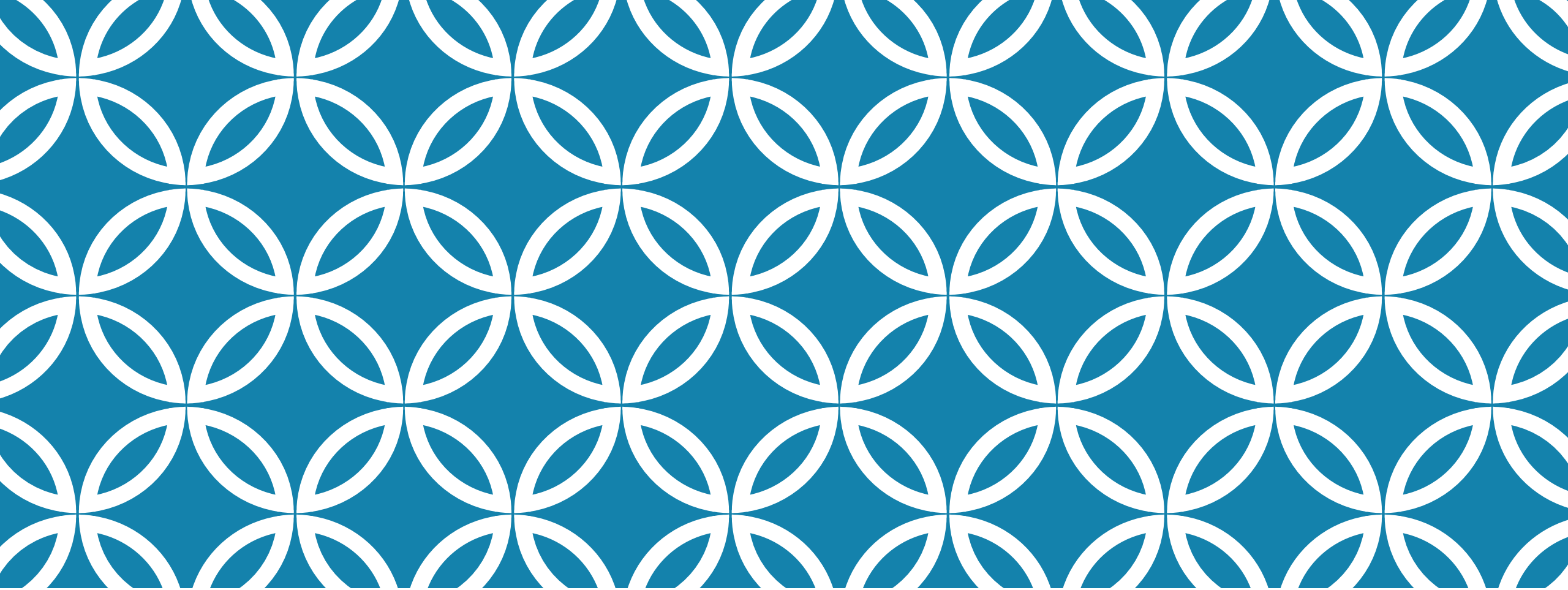


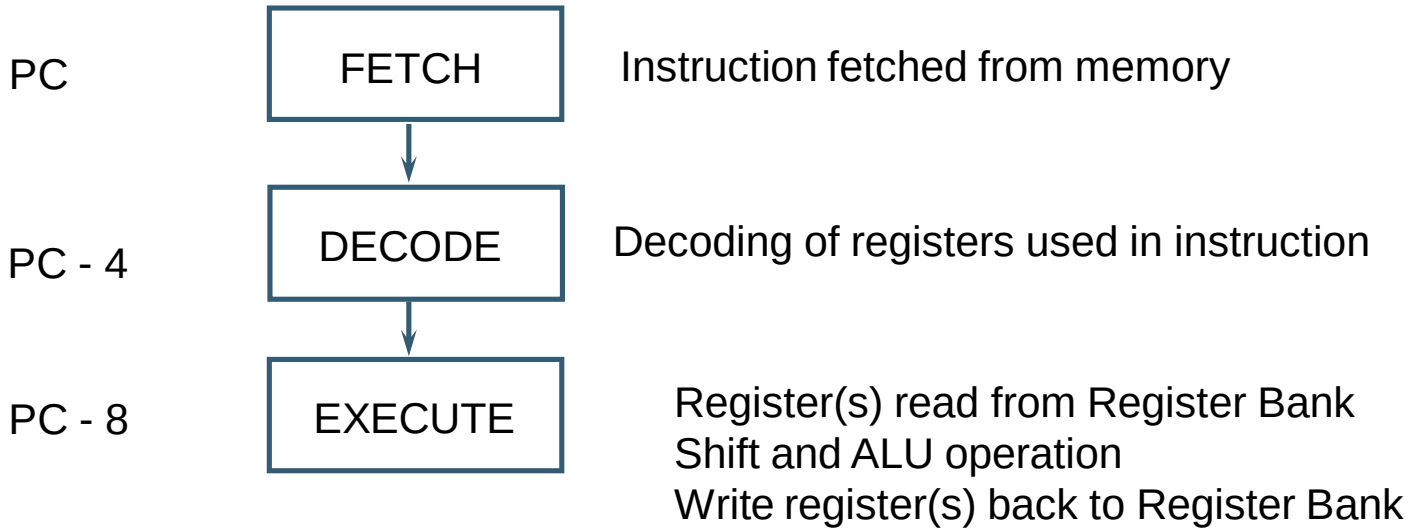
ARM – INSTRUCTION SET ARCHITECTURE

AGENDA

- Instruction Pipeline
- ARM Instruction Formats
- Data Processing Instructions
- Data Transfer Instructions

THE INSTRUCTION PIPELINE

- The ARM uses a pipeline in order to increase the speed of the flow of instructions to the processor.
  - Allows several operations to be undertaken simultaneously, rather than serially.
- Rather than pointing to the instruction being executed, the PC points to the instruction being fetched.



ARM Instruction Set

- ARM defines two separate types of instruction sets,
  - ARM state instruction set – 32 bit wide
  - THUMB state instruction set – 16 bit wide
- ARM Instruction Set Features,
  - Load-store architecture – Only the load and store are supposed to access memory.
  - 3-address instructions
  - Conditional execution of every instruction
  - Possible to combine shift and ALU operations in a single instruction

ARM Instruction Format

- 3 Address Instruction

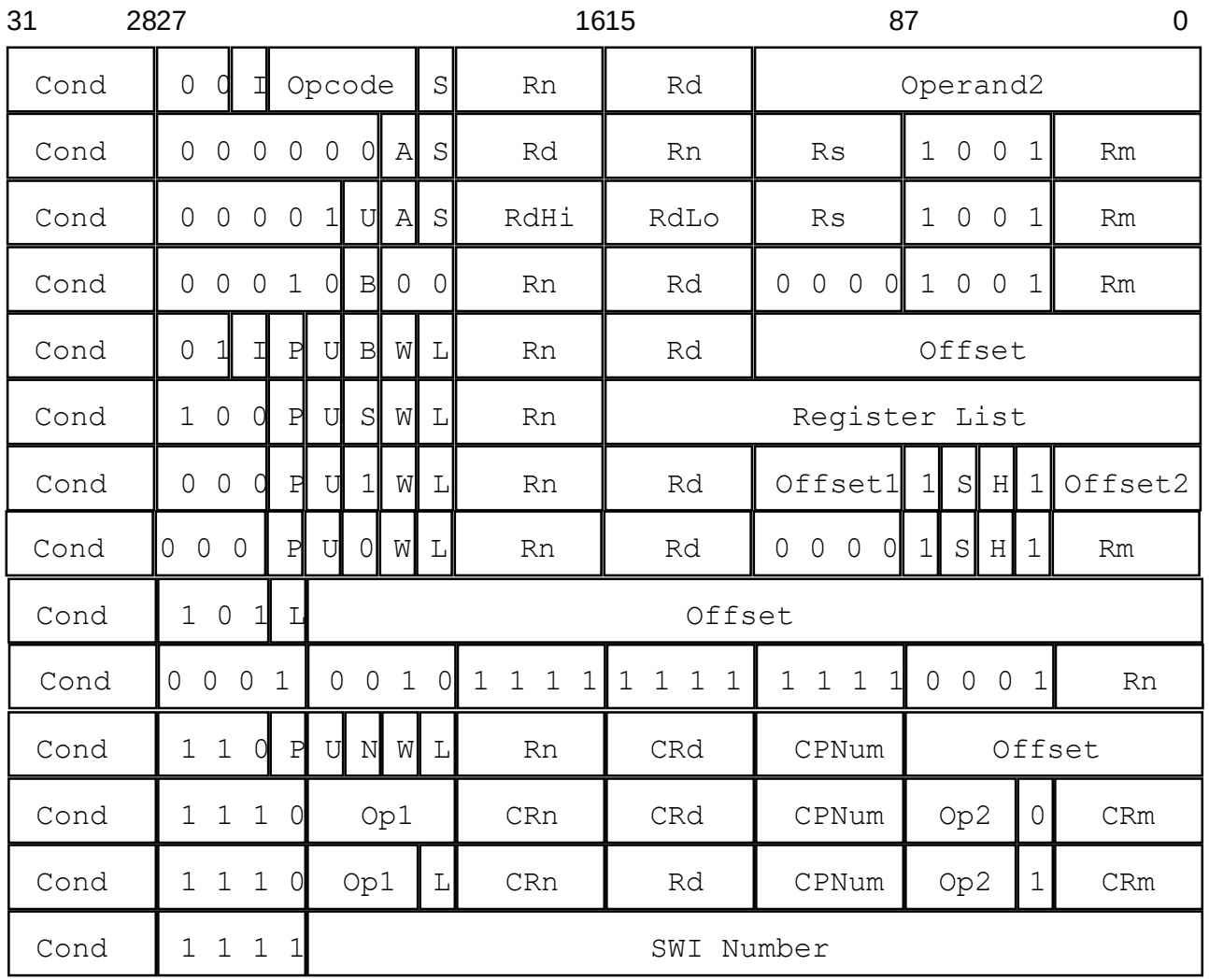
```
MNEMONIC{S}{condition} {Rd}, Operand1, Operand2
```

- MNEMONIC - Short name (mnemonic) of the instruction
- {S} - An optional suffix. If S is specified, the condition flags are updated on the result of the operation
- {condition} - Condition that is needed to be met in order for the instruction to be executed
- {Rd} - Register (destination) for storing the result of the instruction
- Operand1 - First operand. Either a register or an immediate value
- Operand2 - Second (flexible) operand. Can be an immediate value (number) or a register with an optional shift

ARM Instruction Classification

- Data Processing Instructions
- Control flow instructions - Conditional and Branching Instructions
- Data Transfer Instructions - Single-Register Load-Store Instructions
- Stack Instructions
- Software Interrupt Instructions

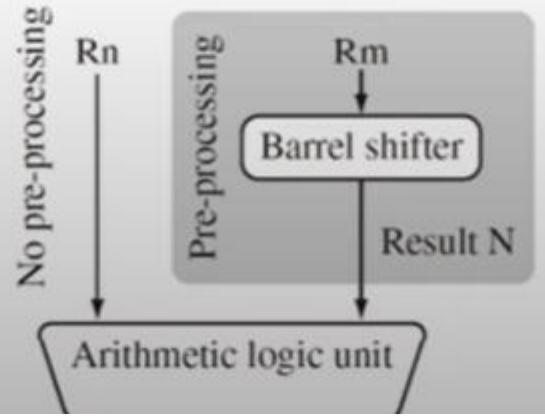
ARM INSTRUCTION FORMATS



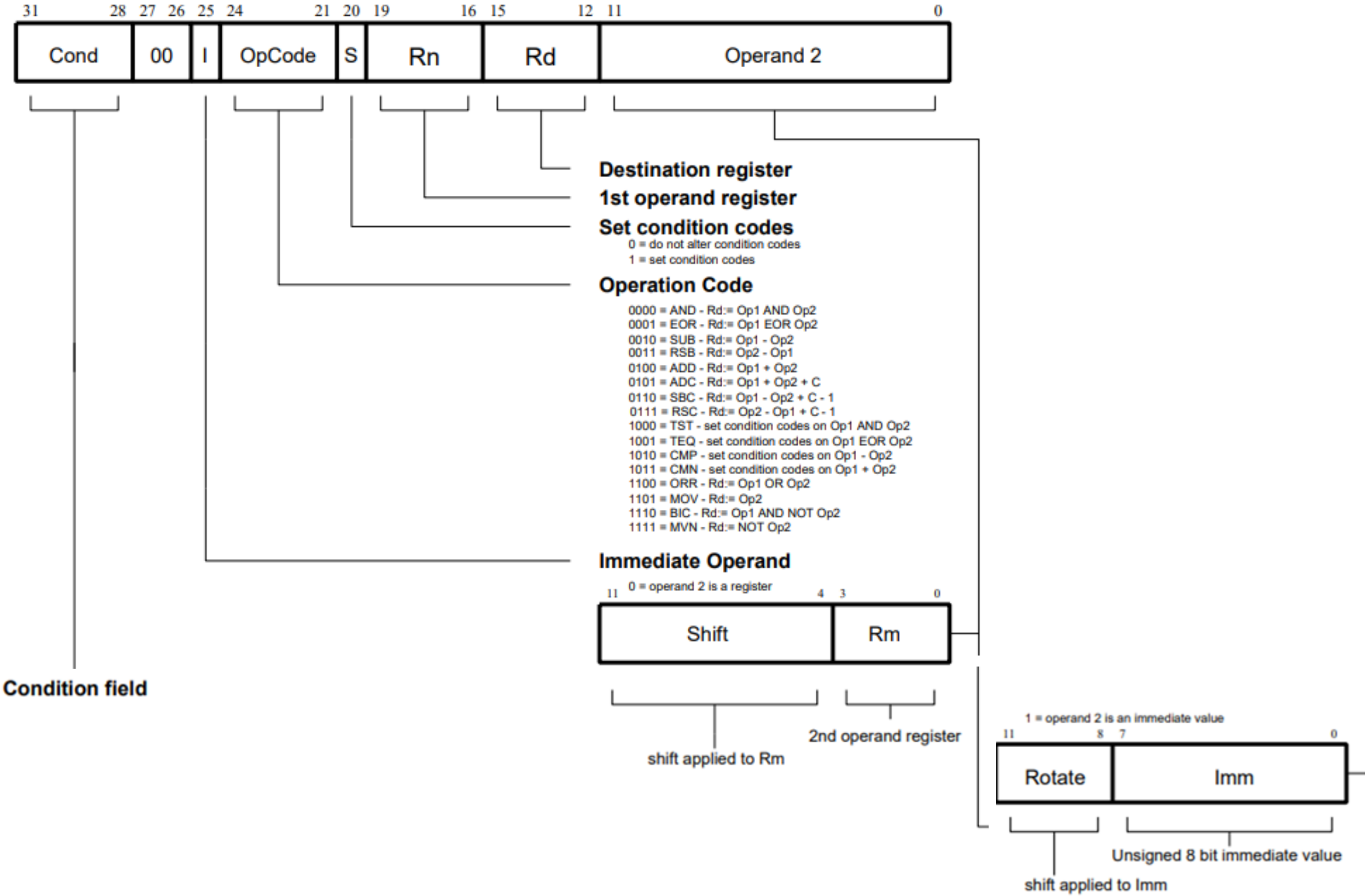
Instruction type
Data processing / PSR Transfer
Multiply
Long Multiply (v3M / v4 only)
Swap
Load/Store Byte/Word
Load/Store Multiple
Halfword transfer : Immediate offset (v4 only)
Halfword transfer: Register offset (v4 only)
Branch
Branch Exchange (v4T only)
Coprocessor data transfer
Coprocessor data operation
Coprocessor register transfer
Software interrupt

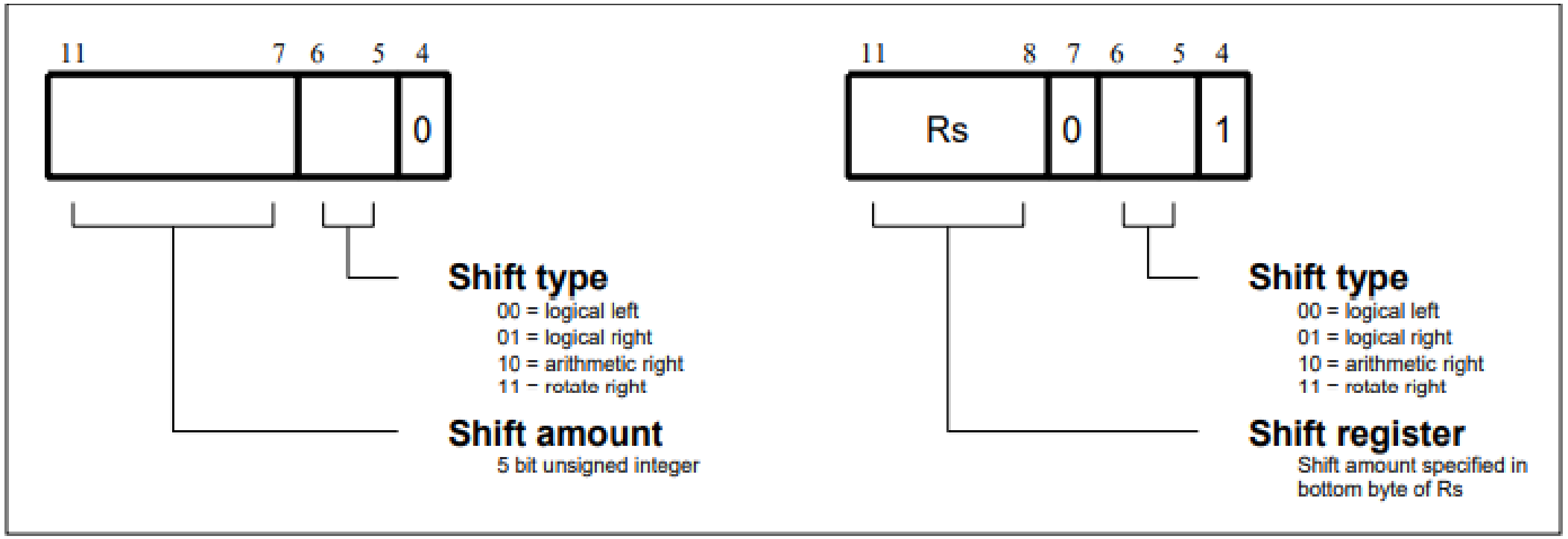
Data Processing Instructions

- Perform move, arithmetic, logical, compare and multiply operations
- All operations except multiply instructions are carried out in ALU
- Multiply instructions are carried out in multiplier block
- Data processing instructions do not access memory
- Instructions operate on two 32-bit operands, produce 32-bit result.
- Most data processing instructions can pre-process one operand using barrel shifter. Approximately there are 16 basic instructions are there.



- Perform
  - Data Movement - **MOV** **MVN**
  - Arithmetic - **ADD** **ADC** **SUB** **SUBC** **RSB** **RSC**
  - Logical - **AND** **ORR** **EOR** **BIC**
  - Compare - **CMP** **CMN** **TST** **TEQ**
    - (No results - Just set condition codes)
- Approximately there are 16 basic instructions are there.


31 28 27 26 25 24 21 20 19 16 15 12 11 0
Cond
00
I
OpCode
S
Rn
Rd
Operand 2
Destination register
1st operand register
Set condition codes
0 = do not alter condition codes
1 = set condition codes
Operation Code
0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2
Immediate Operand
0 = operand 2 is a register
11 4 3 0
Shift
Rm
shift applied to Rm
2nd operand register
1 = operand 2 is an immediate value
11 8 7 0
Rotate
Imm
Unsigned 8 bit immediate value
shift applied to Imm
Condition field


11
7
6
5
4
0
Shift type
00 = logical left
01 = logical right
10 = arithmetic right
11 = rotate right
Shift amount
5 bit unsigned integer
11
8
7
6
5
4
Rs
0
1
Shift type
00 = logical left
01 = logical right
10 = arithmetic right
11 = rotate right
Shift register
Shift amount specified in
bottom byte of Rs

DATA PROCESSING INSTRUCTIONS

Largest family of ARM instructions, all sharing the same instruction format.

Contains:

- Arithmetic operations
- Comparisons (no results - just set condition codes)
- Logical operations
- Data movement between registers

Remember, this is a load / store architecture

- These instruction only work on registers, ***NOT*** memory.

They each perform a specific operation on one or two operands.

- First operand always a register - Rn
- Second operand sent to the ALU via barrel shifter.

ARITHMETIC OPERATIONS

Operations are:

- ADD operand1 + operand2
- ADC operand1 + operand2 + carry
- SUB operand1 - operand2
- SBC operand1 - operand2 + carry -1
- RSB operand2 - operand1
- RSC operand2 - operand1 + carry - 1

Syntax:

- <Operation>{<cond>}{S} Rd, Rn, Operand2

Examples

- ADD r0, r1, r2
- SUBGT r3, r3, #1
- RSBLES r4, r5, #5

COMPARISONS

The only effect of the comparisons is to

- ***UPDATE THE CONDITION FLAGS***. Thus no need to set S bit.

Operations are:

- CMP operand1 - operand2, but result not written
- CMN operand1 + operand2, but result not written
- TST operand1 AND operand2, but result not written
- TEQ operand1 EOR operand2, but result not written

Syntax:

- <Operation>{<cond>} Rn, Operand2

Examples:

- CMP r0, r1
- TSTEQ r2, #5

LOGICAL OPERATIONS

Operations are:

- AND operand1 AND operand2
- EOR operand1 EOR operand2
- ORR operand1 OR operand2
- BIC operand1 AND NOT operand2 [ie bit clear]

Syntax:

- <Operation>{<cond>}{S} Rd, Rn, Operand2

Examples:

- AND r0, r1, r2
- BICEQ r2, r3, #7
- EORS r1,r3,r0

DATA MOVEMENT

Operations are:

- MOV operand2
- MVN NOT operand2

Note that these make no use of operand1.

Syntax:

- <Operation>{<cond>}{S} Rd, Operand2

Examples:

- MOV r0, r1
- MOVS r2, #10
- MVNEQ r1,#0

REGISTERS OPERANDS

The operands are in registers. First register is destination register, second register is operand1 and third register is operand2.

```
ADD R0, R1, R2     ;R0=R1+R2 Add contents of register R1 with
                   register R2 and place the result in
                   register R0.

ADC R0, R1, R2;    ;R0 = R1+R2 +C Add with carry C is carry
                   bit.

SUB R0, R2, R3     ;R0=R2-R3 where R2 is first operand and R3 is
                   second operand

SBC R0, R2, R3;    ;R0=R2-R3+C-1 SUB with carry.

RSB R0, R2, R5     ;R0= R5-R2 Reverse SUB.

RSC R0, R2, R5     ;R0=R5-R2+C-1 Reverse sub with carry.

AND R0, R3, R5     ;R0= R3 AND R5.

ORR R7, R3, R5;    ;R7=R3 OR R5.

EOR R0, R1, R2     ;R0 = R1 Exclusive OR with R2.
```

```
BIC R0, R1, R2    ;Bit clear. The one in second operand clears
                  corresponding bit in first operand and stores the
                  results in destination register.
```

Example 3.1: Assume contents of R1 is 1111111111011111 and R2 is 1000 0100 1110 0011 after execution of **BIC R0,R1, R2** the R0 contains 0111 101100011100

B. Immediate Operand: In immediate operand, operand2 is an immediate value and maximum can be 12 bits

```
ADD R1, R2, #0x25     ;R1=R2+&25, # means immediate and & means
                      the immediate value is in hexadecimal.

AND R2, R3, #0x45     ;R2 = R3 AND &45.

EOR R2, R3, #0x45     ;R2 = R3 Exclusive OR &45.
```

Example 3.2: What is contents of R1 after executing following instruction, assume R2 contains 0x12345678

```
ADD R1, R2, #0x345
```

The ADD instruction will add contains of R2 with 0x2345 and store the result in R1, then R1 = 0x123459BD

Setting Flag Bits of PSR: The above instructions do not affect the flag bit of PSR because the instructions do not have option S. By adding suffix S to the instruction, the instruction would affect the flag bit.

```
ADDS R1, R2, R3      ;The suffix S means set appropriate flag
                     bit.

SUBS R1, R2, R2;     ;The will set zero flag to 1.
```

A. MOV{S} {condition} Rd, Rm; Move the contents of Rm to Rd

Example 3.6: What is contents of R1 after Execution of following instruction

Assume R2 contains 0X0000FFFF

a. `MOV R1, R2;R1 ← R2`
`R2=0x0000FFFF`

b. `MVN R1, R2; R1 ← NOT R2`
`R2= 0xFFFF0000`

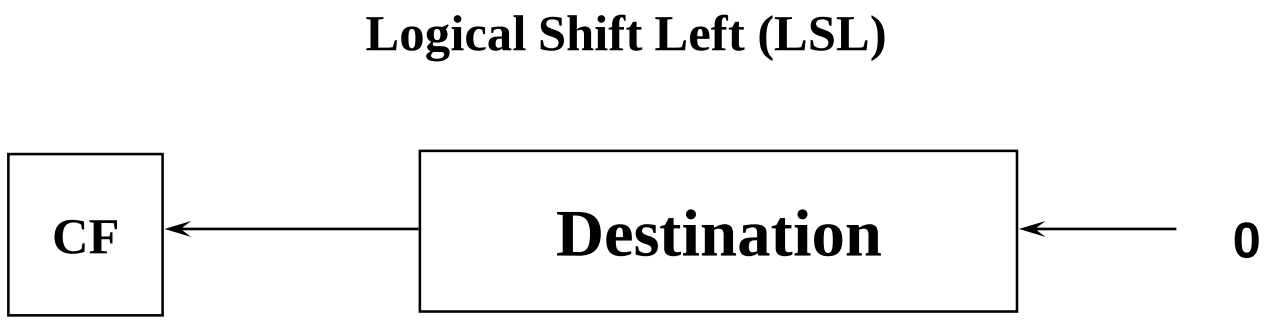
THE BARREL SHIFTER

- The ARM doesn't have actual shift instructions.
- Instead it has a barrel shifter which provides a mechanism to carry out shifts as part of other instructions.
- So what operations does the barrel shifter support?

BARREL SHIFTER - LEFT SHIFT

Shifts left by the specified amount (multiplies by powers of two) e.g.

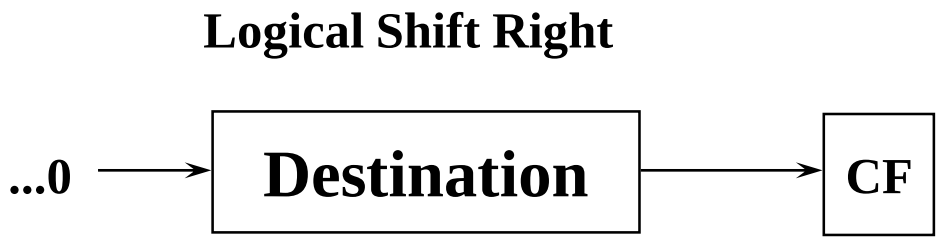
LSL #5 = multiply by 32



BARREL SHIFTER - RIGHT SHIFTS

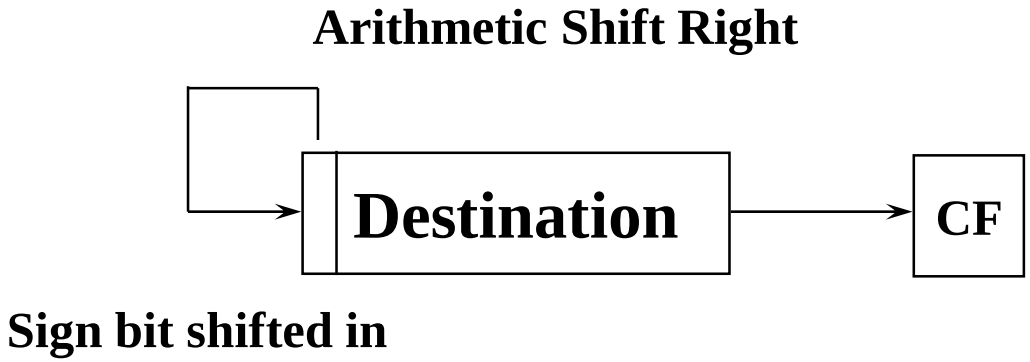
Logical Shift Right

- Shifts right by the specified amount (divides by powers of two)
- e.g. LSR #5 = divide by 32



Arithmetic Shift Right

- Shifts right (divides by powers of two) and preserves the sign bit, for 2's complement operations.
- e.g. ASR #5 = divide by 32

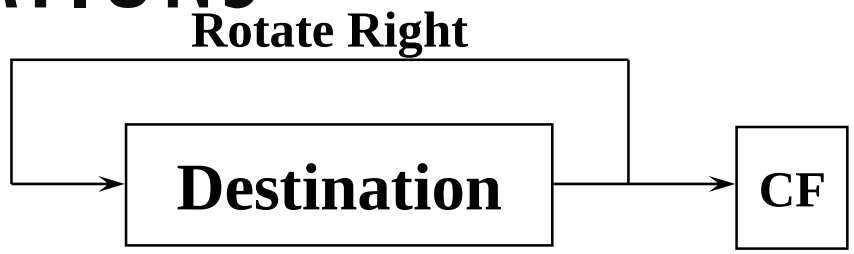


BARREL SHIFTER - ROTATIONS

Rotate Right (ROR)

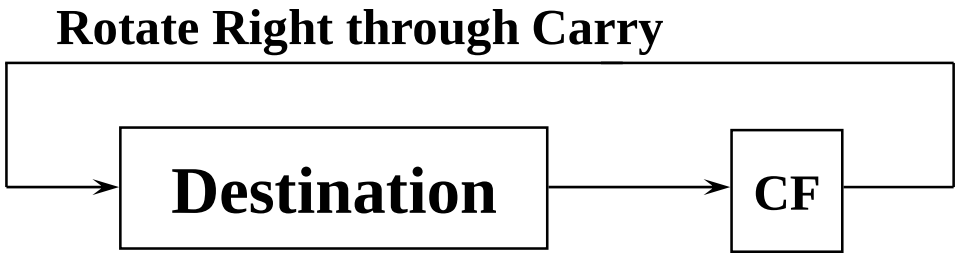
- Similar to an ASR but the bits wrap around as they leave the LSB and appear as the MSB. Note the last bit rotated is also used as the Carry Out.

e.g. ROR #5

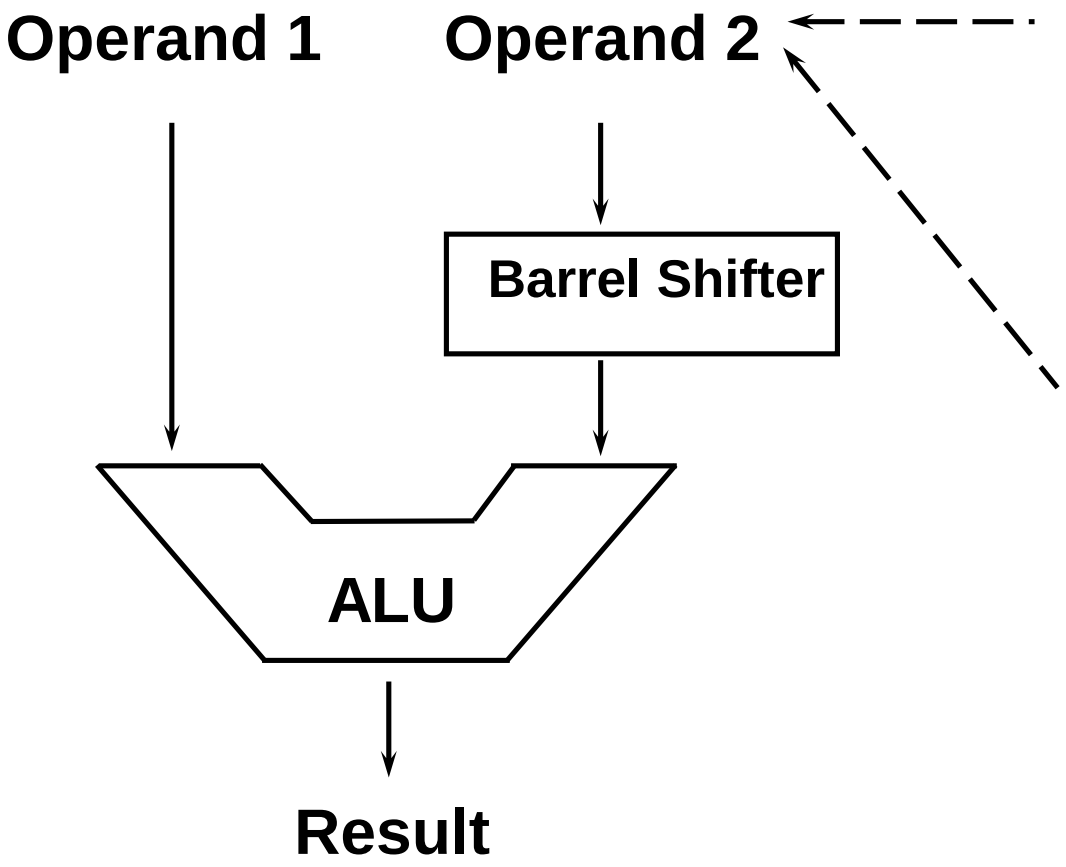


Rotate Right Extended (RRX)

- This operation uses the CPSR C flag as a 33rd bit.
- Rotates right by 1 bit. Encoded as ROR #0.



USING THE BARREL SHIFTER : THE SECOND OPERAND



Register, optionally with shift operation applied.

Shift value can be either be:

- 5 bit unsigned integer
- Specified in bottom byte of another register.

* **Immediate value**

- 8 bit number
- Can be rotated right through an even number of positions.
- Assembler will calculate rotate for you from constant.

SECOND OPERAND : SHIFTED REGISTER

The amount by which the register is to be shifted is contained in either:

- the immediate 5-bit field in the instruction
  - *NO OVERHEAD*
  - Shift is done for free - executes in single cycle.
- the bottom byte of a register (not PC)
  - Then takes extra cycle to execute
  - ARM doesn't have enough read ports to read 3 registers at once.
  - Then same as on other processors where shift is separate instruction.

If no shift is specified then a default shift is applied: LSL #0

- i.e. barrel shifter has no effect on value in register.

SECOND OPERAND : USING A SHIFTED REGISTER

Using a multiplication instruction to multiply by a constant means first loading the constant into a register and then waiting a number of internal cycles for the instruction to complete.

A more optimum solution can often be found by using some combination of MOVs, ADDs, SUBs and RSBs with shifts.

- Multiplications by a constant equal to a ((power of 2) ± 1) can be done in one cycle.

Example: r0 = r1 * 5
= r1 + (r1 * 4)

⇦ ADD r0, r1, r1, LSL #2

Example: r2 = r3 * 105
= r3 * 15 * 7
= r3 * (16 - 1) * (8 - 1)
⇦ RSB r2, r3, r3, LSL #4 ; r2 = r3 * 15
⇦ RSB r2, r2, r2, LSL #3 ; r2 = r2 * 7

SECOND OPERAND : IMMEDIATE VALUE (1)

There is no single instruction which will load a 32 bit immediate constant into a register without performing a data load from memory.

- All ARM instructions are 32 bits long
- ARM instructions do not use the instruction stream as data.

The data processing instruction format has 12 bits available for operand2

- If used directly this would only give a range of 4096.

Instead it is used to store 8 bit constants, giving a range of 0 - 255.

These 8 bits can then be rotated right through an even number of positions (ie RORs by 0, 2, 4,..30).

This gives a much larger range of constants that can be directly loaded, though some constants will still need to be loaded from memory.

SECOND OPERAND : IMMEDIATE VALUE (2)

This gives us:

- 0 - 255 [0 - 0xff]
- 256,260,264,..,1020 [0x100-0x3fc, step 4, 0x40-0xff ror 30]
- 1024,1040,1056,..,4080 [0x400-0xff0, step 16, 0x40-0xff ror 28]
- 4096,4160, 4224,..,16320 [0x1000-0x3fc0, step 64, 0x40-0xff ror 26]

These can be loaded using, for example:

- MOV r0, #0x40, 26 ; => MOV r0, #0x1000 (ie 4096)

To make this easier, the assembler will convert to this form for us if simply given the required constant:

- MOV r0, #4096 ; => MOV r0, #0x1000 (ie 0x40 ror 26)

The bitwise complements can also be formed using MVN:

- MOV r0, #0xFFFFFFFF ; assembles to MVN r0, #0

If the required constant cannot be generated, an error will be reported.

LOADING FULL 32 BIT CONSTANTS

Although the MOV/MVN mechansim will load a large range of constants into a register, sometimes this mechansim will not generate the required constant.

Therefore, the assembler also provides a method which will load *ANY* 32 bit constant:

- `LDR rd,=numeric constant`

If the constant can be constructed using either a MOV or MVN then this will be the instruction actually generated.

Otherwise, the assembler will produce an LDR instruction with a PC-relative address to read the constant from a literal pool.

- `LDR r0,=0x42   ; generates MOV r0,#0x42`
- `LDR r0,=0x55555555      ; generate LDR r0,[pc, offset to lit pool]`

As this mechanism will always generate the best instruction for a given case, it is the recommended way of loading constants.

MULTIPLICATION INSTRUCTIONS

The Basic ARM provides two multiplication instructions.

Multiply

- MUL{<cond>}{S} Rd, Rm, Rs ; Rd = Rm * Rs

Multiply Accumulate - does addition for free

- MLA{<cond>}{S} Rd, Rm, Rs,Rn ; Rd = (Rm * Rs) + Rn

Restrictions on use:

- Rd and Rm cannot be the same register
  - Can be avoid by swapping Rm and Rs around. This works because multiplication is commutative.
- Cannot use PC.

These will be picked up by the assembler if overlooked.

Operands can be considered signed or unsigned

- Up to user to interpret correctly.

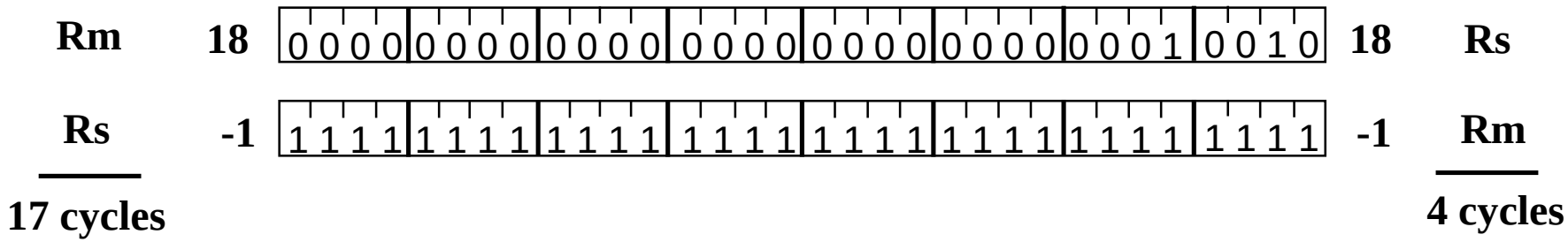
MULTIPLICATION IMPLEMENTATION

The ARM makes use of Booth's Algorithm to perform integer multiplication.

On non-M ARMs this operates on 2 bits of Rs at a time.

- For each pair of bits this takes 1 cycle (plus 1 cycle to start with).
- However when there are no more 1's left in Rs, the multiplication will early-terminate.

Example: Multiply 18 and -1 : Rd = Rm * Rs



Note: Compiler does not use early termination criteria to decide on which order to place operands.

EXTENDED MULTIPLY INSTRUCTIONS

M variants of ARM cores contain extended multiplication hardware. This provides three enhancements:

- An *8 bit Booth's Algorithm* is used
  - Multiplication is carried out faster (maximum for standard instructions is now 5 cycles).
- *Early termination method improved* so that now completes multiplication when all remaining bit sets contain
  - all zeroes (as with non-M ARMs), or
  - all ones.

  Thus the previous example would early terminate in 2 cycles in both cases.
- *64 bit results* can now be produced from two 32bit operands
  - Higher accuracy.
  - Pair of registers used to store result.

MULTIPLY-LONG AND MULTIPLY-ACCUMULATE LONG

Instructions are

- MULL which gives RdHi,RdLo:=Rm*Rs
- MLAL which gives RdHi,RdLo:=(Rm*Rs)+RdHi,RdLo

However the full 64 bit of the result now matter (lower precision multiply instructions simply throws top 32bits away)

- Need to specify whether operands are signed or unsigned

Therefore syntax of new instructions are:

- UMULL{<cond>}{S} RdLo,RdHi,Rm,Rs
- UMLAL{<cond>}{S} RdLo,RdHi,Rm,Rs
- SMULL{<cond>}{S} RdLo, RdHi, Rm, Rs
- SMLAL{<cond>}{S} RdLo, RdHi, Rm, Rs

Not generated by the compiler.

Warning : Unpredictable on non-M ARMs.

LOAD / STORE INSTRUCTIONS

The ARM is a Load / Store Architecture:

- Does not support memory to memory data processing operations.
- Must move data values into registers before using them.

This might sound inefficient, but in practice isn't:

- Load data values from memory into registers.
- Process data in registers using a number of data processing instructions which are not slowed down by memory access.
- Store results from registers out to memory.

The ARM has three sets of instructions which interact with main memory. These are:

- Single register data transfer (LDR / STR).
- Block data transfer (LDM/STM).
- Single Data Swap (SWP).

SINGLE REGISTER DATA TRANSFER

The basic load and store instructions are:

- Load and Store Word or Byte
  - LDR / STR / LDRB / STRB

ARM Architecture Version 4 also adds support for halfwords and signed data.

- Load and Store Halfword
  - LDRH / STRH
- Load Signed Byte or Halfword - load value and sign extend it to 32 bits.
  - LDRSB / LDRSH

All of these instructions can be conditionally executed by inserting the appropriate condition code after STR / LDR.

- e.g. LDREQB

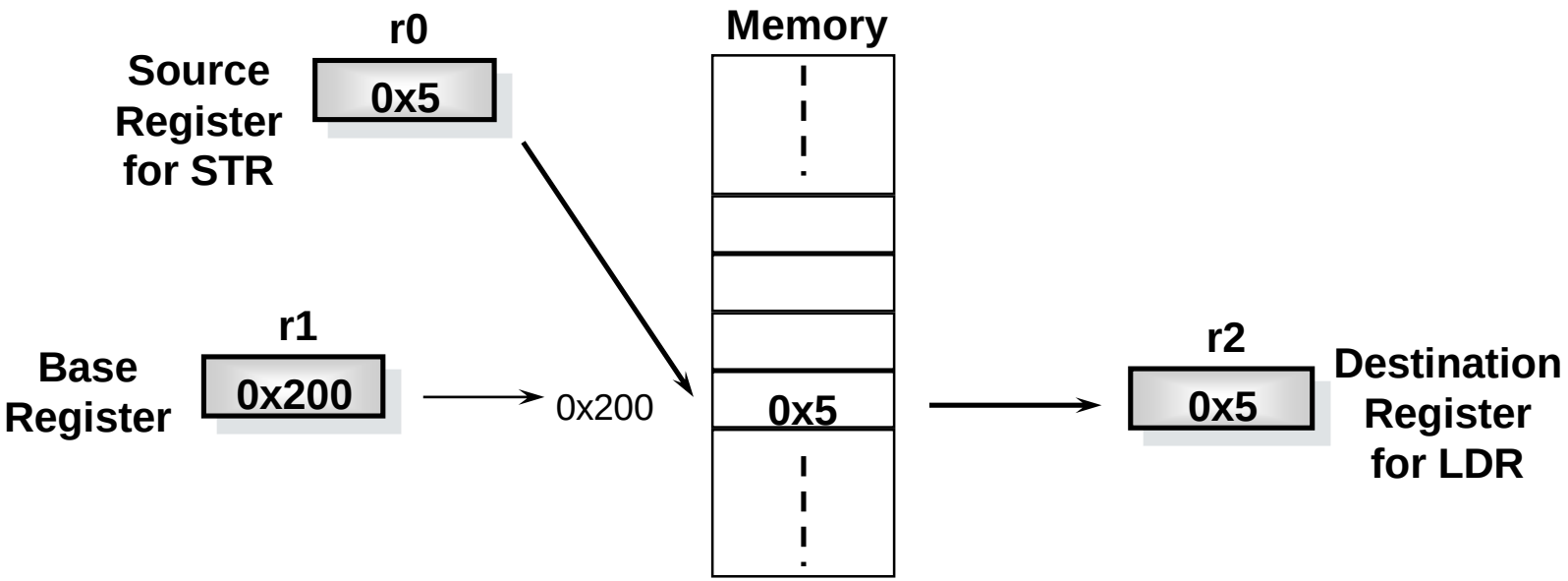
Syntax:

- <LDR|STR>{<cond>}{<size>} Rd, <address>

LOAD AND STORE WORD OR BYTE: BASE REGISTER

The memory location to be accessed is held in a base register

- STR r0, [r1] ; Store contents of r0 to location pointed to
  ; by contents of r1.
- LDR r2, [r1] ; Load r2 with contents of memory location
  ; pointed to by contents of r1.



LOAD AND STORE WORD OR BYTE: OFFSETS FROM THE BASE REGISTER

As well as accessing the actual location contained in the base register, these instructions can access a location offset from the base register pointer.

This offset can be

- An unsigned 12bit immediate value (ie 0 - 4095 bytes).
- A register, optionally shifted by an immediate value

This can be either added or subtracted from the base register:

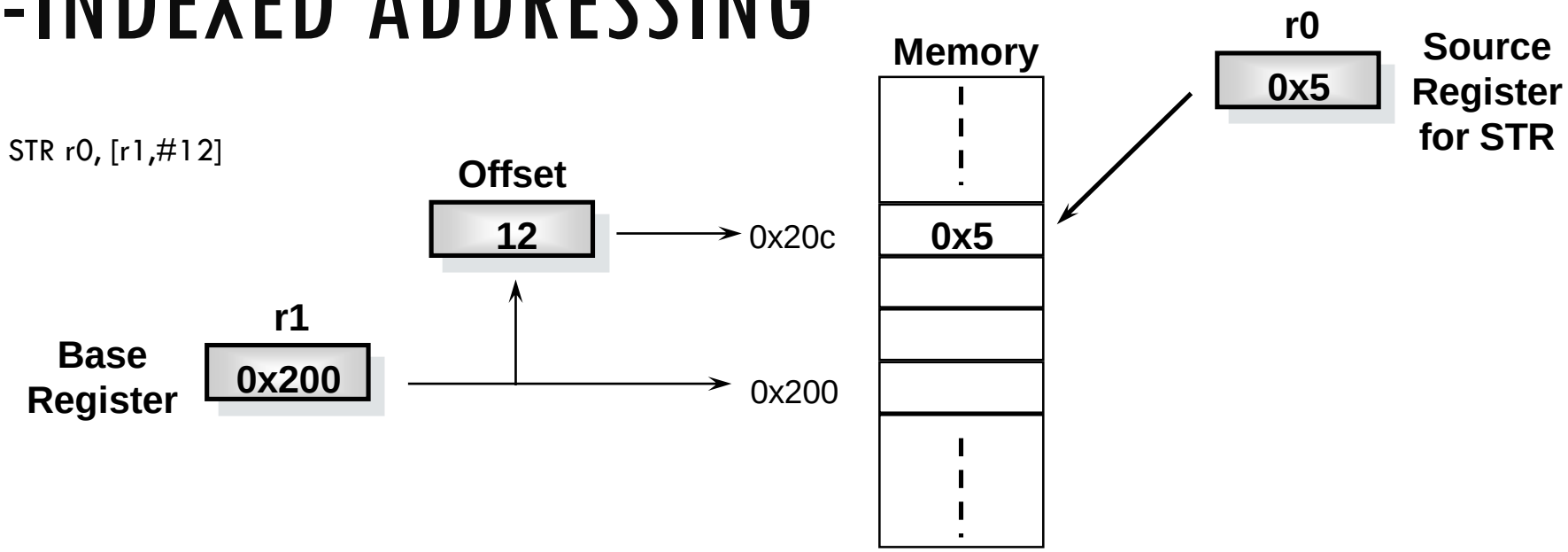
- Prefix the offset value or register with '**+**' (default) or '**-**'.

This offset can be applied:

- before the transfer is made: ***Pre-indexed addressing***
  - optionally *auto-incrementing* the base register, by postfixing the instruction with an '**!**'.
- after the transfer is made: ***Post-indexed addressing***
  - causing the base register to be *auto-incremented*.

LOAD AND STORE WORD OR BYTE: PRE-INDEXED ADDRESSING

Example: STR r0, [r1,#12]



To store to location 0x1f4 instead use: STR r0, [r1,#-12]

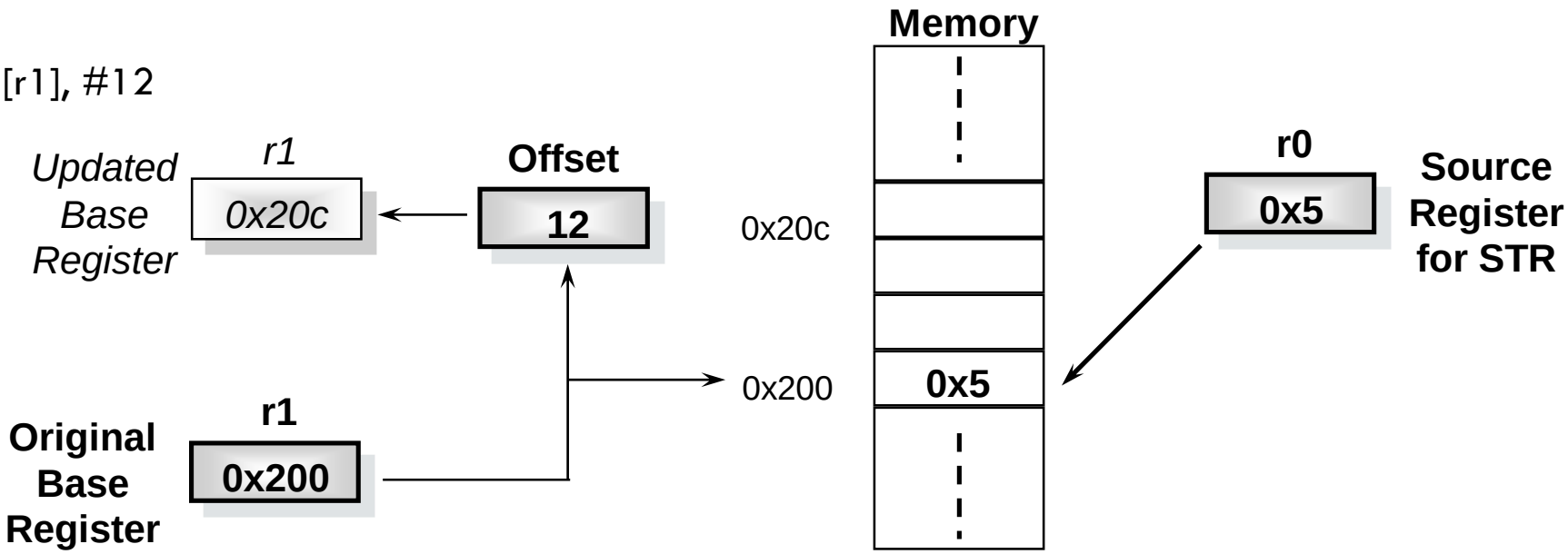
To auto-increment base pointer to 0x20c use: STR r0, [r1, #12]!

If r2 contains 3, access 0x20c by multiplying this by 4:

- STR r0, [r1, r2, LSL #2]

LOAD AND STORE WORD OR BYTE: POST-INDEXED ADDRESSING

Example: STR r0, [r1], #12



To auto-increment the base register to location 0x1f4 instead use:

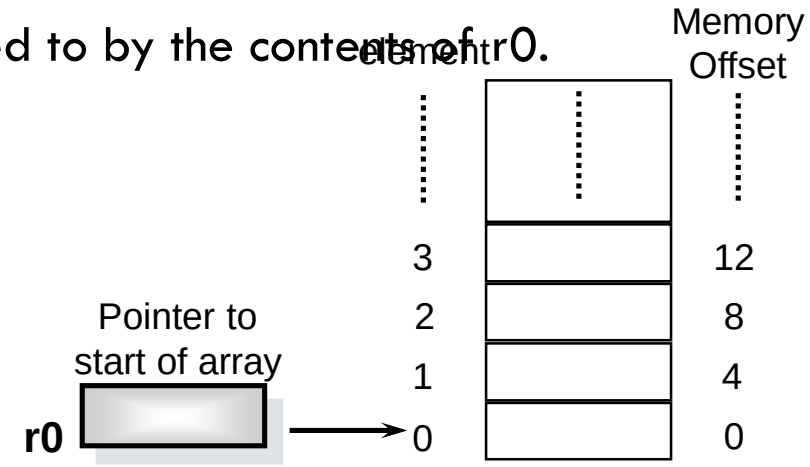
- STR r0, [r1], #-12

If r2 contains 3, auto-incremenet base register to 0x20c by multiplying this by 4:

- STR r0, [r1], r2, LSL #2

EXAMPLE USAGE OF ADDRESSING MODES

Imagine an array, the first element of which is pointed to by the contents of r0.



If we want to access a particular element, then we can use pre-indexed addressing:

- r1 is element we want.
- LDR r2, [r0, r1, LSL #2]

If we want to step through every element of the array, for instance to produce sum of elements in the array, then we can use post-indexed addressing within a loop:

- r1 is address of current element (initially equal to r0).
- LDR r2, [r1], #4

Use a further register to store the address of final element, so that the loop can be correctly terminated.

BLOCK DATA TRANSFER (1)

The Load and Store Multiple instructions (LDM / STM) allow betweeen 1 and 16 registers to be transferred to or from memory.

The transferred registers can be either:

- Any subset of the current bank of registers (default).
- Any subset of the user mode bank of registers when in a priviledged mode (postfix instruction with a '^').