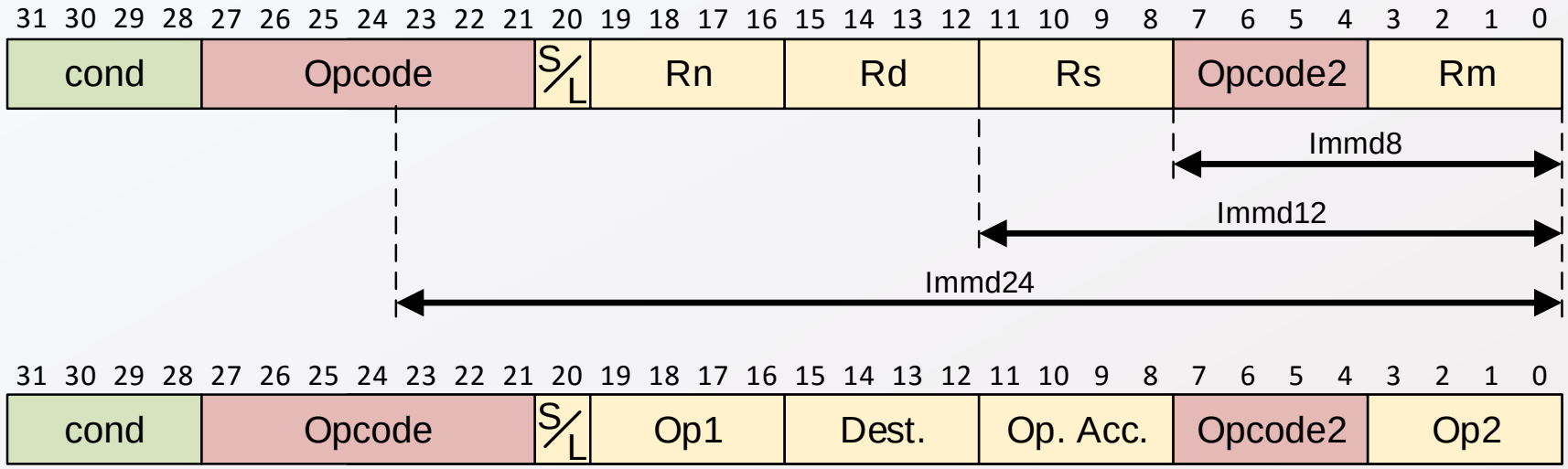
ARM Cortex-A9
ARM v7-A

A programmer’s perspective
Part 2

ARM Instructions



General Format

Inst Rd, Rn, Rm, Rs

Inst Rd, Rn, #0ximm

Instruction Classes

Data Processing (largest class): ADD, AND, BIC, CMP, EOR, MOV, ORR, RSB, SUB, TEQ, TST
ASR, ASL, LSL, LSR, ROR, MLA, MLS, MUL, PKH, SDIV, SXT

Branch Instructions: B, BL, BX

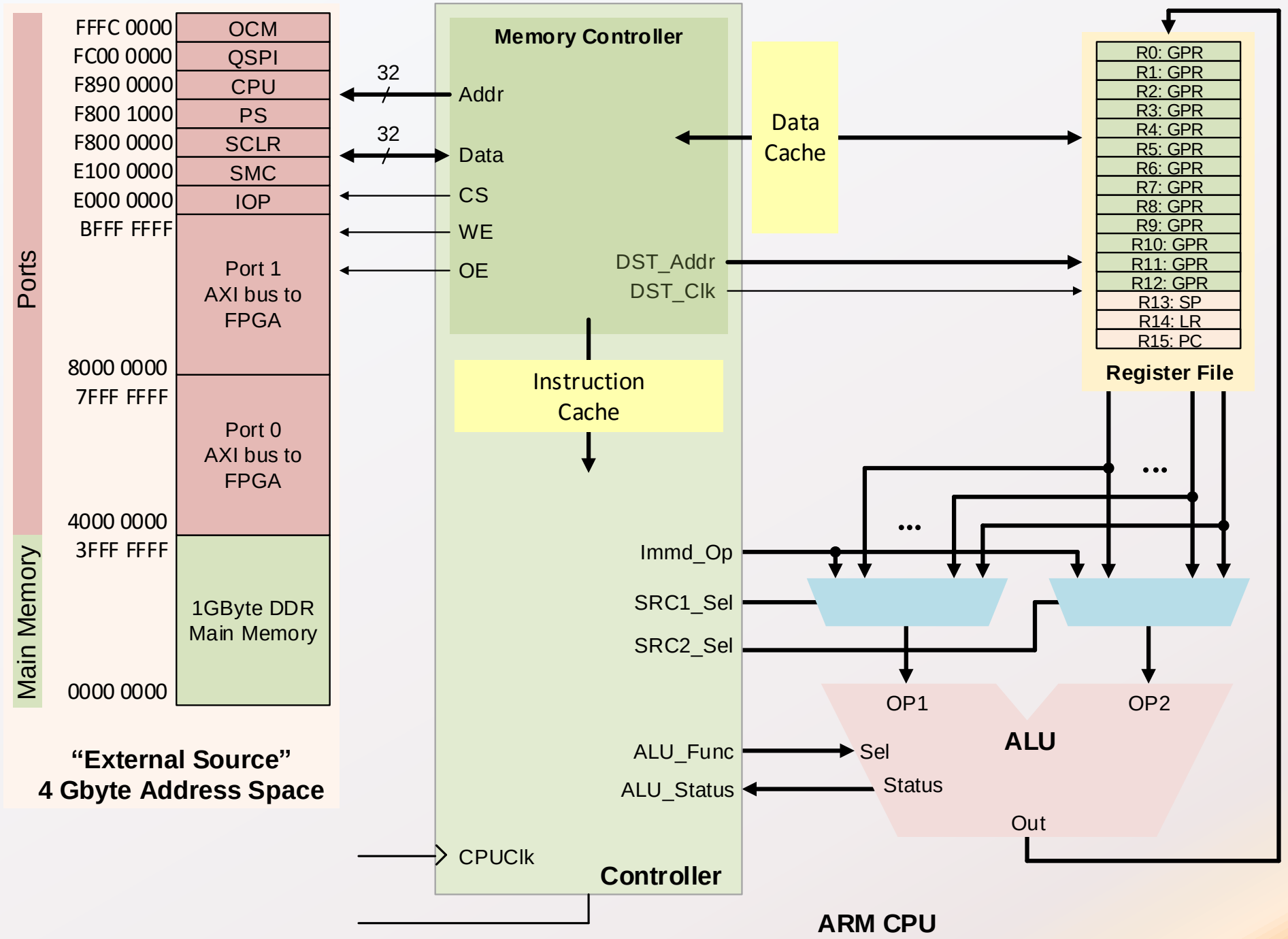
Load/Store: LDR, LDRB, LDRW, LDRD, STR, STRB, STRH, STRD, LDM, LDMIA, LDMDA, LDMDB, LDMIB, STM, STMIA, STMDA, STMDB, STMIB, POP, PUSH

Plus exception handling, coprocessor calls, SIMD, floating point, vector

ARM Instruction Set Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Instruction Type
Condition				0	0	I	OPCODE				S	Rn				Rs				OPERAND-2												Data processing
Condition				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm				Multiply
Condition				0	0	0	0	1	U	A	S	Rd HIGH				Rd LOW				Rs				1	0	0	1	Rm				Long Multiply
Condition				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm				Swap
Condition				0	1	I	P	U	B	W	L	Rn				Rd				OFFSET												Load/Store - Byte/Word
Condition				1	0	0	P	U	B	W	L	Rn				REGISTER LIST																Load/Store Multiple
Condition				0	0	0	P	U	1	W	L	Rn				Rd				OFFSET 1				1	S	H	1	OFFSET 2				Halfword Transfer Imm Off
Condition				0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				Halfword Transfer Reg Off
Condition				1	0	1	L	BRANCH OFFSET																								Branch
Condition				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn				Branch Exchange
Condition				1	1	0	P	U	N	W	L	Rn				CRd				CPNum				OFFSET								COPROCESSOR DATA XFER
Condition				1	1	1	0	Op-1				CRn				CRd				CPNum				OP-2			0	CRm				COPROCESSOR DATA OP
Condition								OP-1			L	CRn				Rd				CPNum				OP-2			1	CRm				COPROCESSOR REG XFER
Condition				1	1	1	1	SWI NUMBER																								Software Interrupt

Loads and Stores



Load and Store

32-bit external addresses (4 Gbytes). ZYNQ supports 30-bit external addresses (1 Gbyte).

Loads and stores can operate on words (4 consecutive bytes), half words (2 bytes), or bytes.

Operands residing in memory must be loaded into a register before they can be used, and new values stored in main memory must be stored from a register.

Three sets of instructions that interact with main memory

- Single register data transfer (LDR / STR)
- Block register data transfer (LDM / STM)
- Single data swap (SWP)

NO memory-to-memory data operations

Load and Store

Basic load and store operations are: LDR/STR, LDRH/STRH, LDRB/STRB, LDRD/STRD

All load/store instructions require a base address pointer placed in a GPR (Rn). Rn is a "pointer", or a 32-bit memory address. Square brackets [] designate a pointer.

```
LDR   Rt, [Rn]          @ Load Rt with location pointed at by Rn
STR   Rt, [Rn]          @ Store Rt at location pointed at by Rn
```

The base address can be modified with an offset applied before access. Examples:

```
LDR   Rt, [Rn,#<imm>]       @ Load Rt with location pointed at by Rn + imm value
STR   Rt, [Rn,#- <imm>]     @ Store Rt at location pointed at by Rn - imm value
```

Imm is a 12-bit "immediate" value; if omitted, default is 0. A minus sign subtracts the immediate value, a plus sign (or no sign) adds the immediate value.

Load and Store

Load at store operations can use the PC to load or store “literal”

```
LDR   Rt, label          @ Load Rt with location at label (can be +/- 4096 from PC)
```

Base address can be modified with an offset stored in a 2nd GPR.

```
LDR   Rt, [Rn, Rm]       @ Load Rn from location [Rn + Rn]
STR   Rt, [Rn, -Rm]      @ Store Rn at location [Rn - Rm]
```

Load and Store with Indexing

Indexing means modifying the base address, and writing the modified value back into the base address register, as a part of the execution cycle.

Pre-indexing means doing a transfer first, then updating Rn. Pre-indexing is used when a '!' is added to the end of any load instruction.

```
LDR   Rt, [Rn, #4]!      @ Rt <- [Rn + 4], then Rn is updated with address that was used
STR   Rt, [Rn, Rm]!      @ Rt <- [Rn + Rm], then Rn is updated with address that was used
```

Post-indexing means updating Rn first, then doing the transfer. Post-indexing is used when only the base address is enclosed in square bracket.

```
LDR   Rt, [Rn], #4       @ Rn updated with (Rn+4), then Rt loaded from new address
STR   Rt, [Rn], Rm       @ Rn is updated with (Rn + Rm), then Rt loaded from new address
```

Indexing provides a powerful tool for working with regular memory structures like arrays.

A quick aside… the barrel shifter

ARMs shifter is located in one operand data path, in front of the ALU.

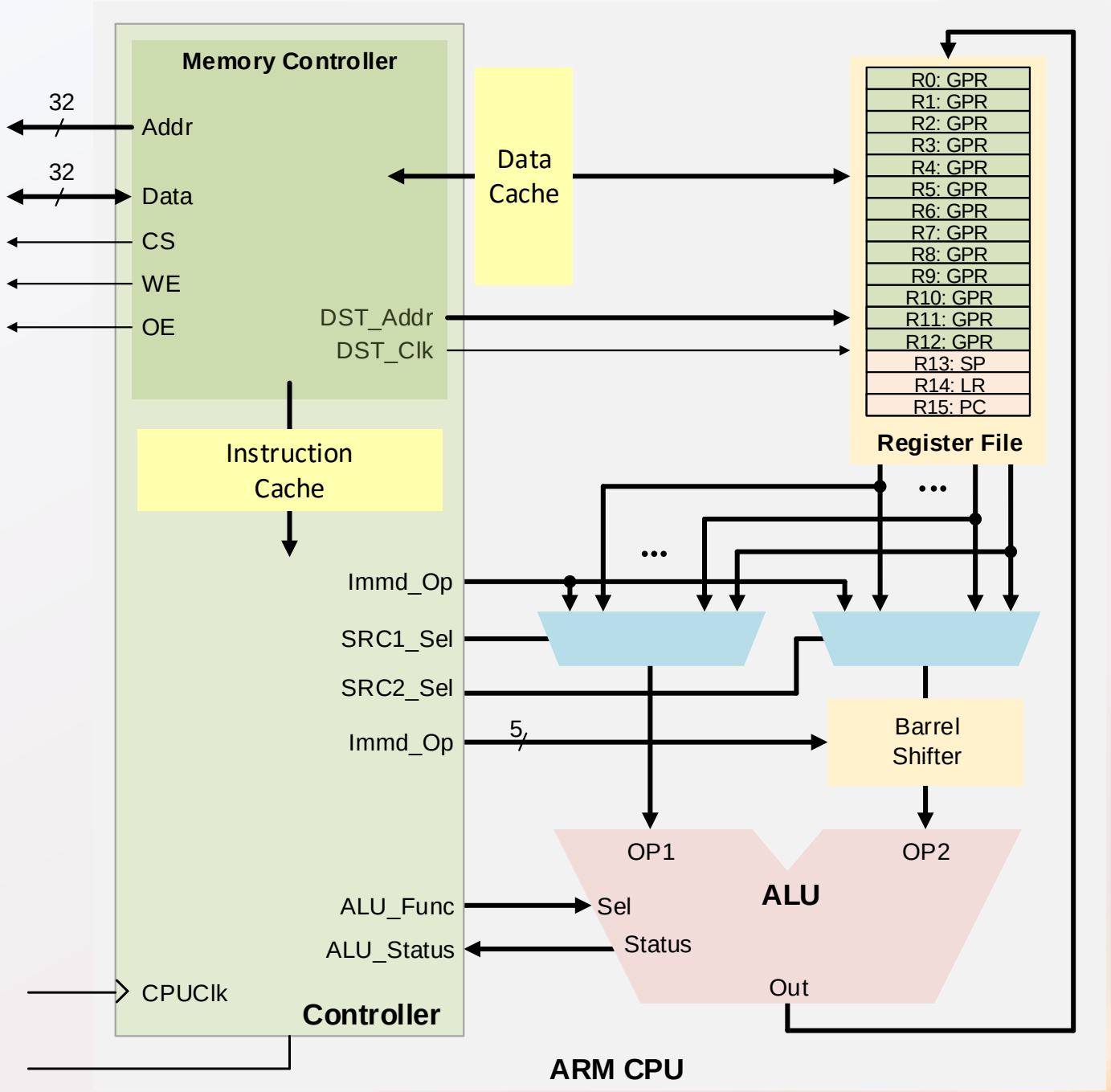
Shifts can occur as a part of almost any instruction

Shift instructions (LSR, LSL, ASR, ASL) work as expected:

```
LSR     Rd, Rm, #imm
LSLS    Rd, Rn, Rm
```

Same with Rotate (ROR):

```
RORS    Rd, Rm, #imm
ROR     Rd, Rn, Rm
```



Load and Store using Barrel Shifter

The shifter can also modify the base address used for load and store operations. A 5-bit immediate field in the opcode encodes shift amount. No extra CPU time is needed.

```
LDR   Rt, [Rn, Rm, LSL #0x4]
STR   Rt, [Rn, Rm, LSR #0x3]!
```

Assembler syntax

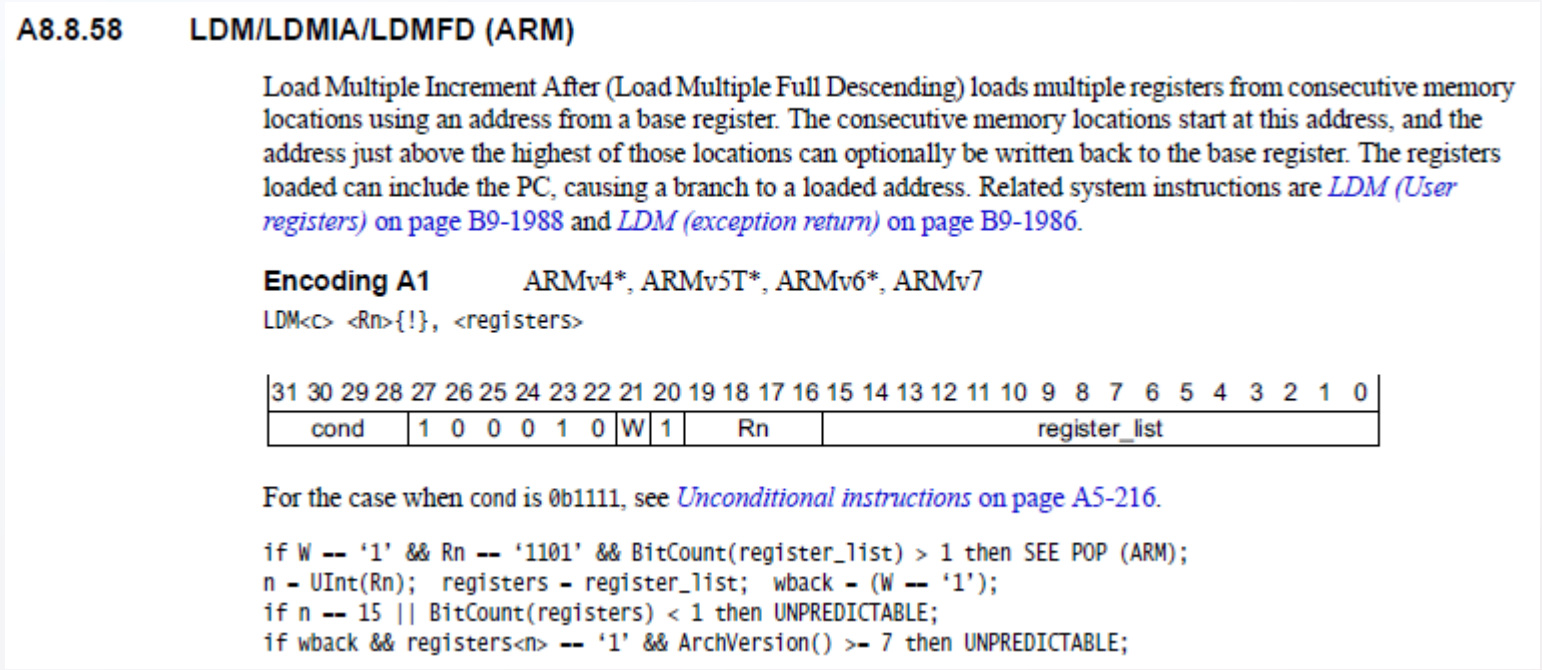
`LDR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]`	Offset: `index==TRUE, wback==FALSE`
`LDR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!`	Pre-indexed: `index==TRUE, wback==TRUE`
`LDR{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}`	Post-indexed: `index==FALSE, wback==TRUE`

where:

`<c>, <q>`	See *Standard assembler syntax fields* on page A8-287.
`<Rt>`	The destination register. The SP can be used. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. In ARMv5T and above, this branch is an interworking branch, see *Pseudocode details of operations on ARM core registers* on page A2-47.
`<Rn>`	The base register. The SP can be used. The PC can be used for offset addressing only.
`+/-`	If + or omitted, the optionally shifted value of `<Rm>` is added to the base register value (`add == TRUE` encoded as U == 1). If –, the optionally shifted value of `<Rm>` is subtracted from the base register value (`add == FALSE` encoded as U == 0).
`<Rm>`	The offset that is optionally shifted and applied to the value of `<Rn>` to form the address.
`<shift>`	The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see *Shifts applied to a register* on page A8-291.

Load and Store

Additional load and store commands operate on lists of registers. More information in the Arm Architectural Reference starting on page A8-396.



A8.8.58 LDM/LDMIA/LDMFD (ARM)

Load Multiple Increment After (Load Multiple Full Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the highest of those locations can optionally be written back to the base register. The registers loaded can include the PC, causing a branch to a loaded address. Related system instructions are *LDM (User registers)* on page B9-1988 and *LDM (exception return)* on page B9-1986.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDM<c> <Rn>{!}, <registers>
```

31 30 29 28	27 26 25 24	23	22	21	20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0 0 0	1	0	W	1	Rn	register_list

For the case when cond is 0b1111, see *Unconditional instructions* on page A5-216.

```
if W == '1' && Rn == '1101' && BitCount(register_list) > 1 then SEE POP (ARM);
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' && ArchVersion() >= 7 then UNPREDICTABLE;
```

LDMIA/LDMFD

LDMDA/LDMFA

LDMDB/LDMEA

LDMIB/LDMED

STMIA/STMEA

STMDA/STMED

STMDB/STMFD

STMIB/STMFA

```
STMDB   R13, {r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14}
LDMIA   R13, {r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14}
```

Load and Store

No instruction can load a 32-bit immediate constant into a register without performing a data load from memory (ARM does not embed immediate in the instruction stream).

8 or 12 bit immediates can be loaded and rotated to give a wider range of numbers that can be generated from immediates.

```
MOV r0, #0x40, 26          @ load #0x1000 into R0
```

The pseudo-instruction shown can move any declared constant into a register:

```
.set myconstant, 0xAAAA5555
LDR  r0, =myconstant       @ load #0xAAAA5555 into R0
```

The pseudo-instruction shown below can also be used. In this case, the assembler will use a MOV or MVN instruction if possible; otherwise, it will create a constant and then load it:

```
LDR  r0, =0xFFFF0C0C
```

PC-relative Load and Store

The PC can be used as the base address.

```
LDR    R0, [PC, #4]                @ load R0 from location PC + 4
```

The PC points 8 bytes ahead of executing instructing (why)?

Calculating relative addresses can be tedious. An ARM pseudo-instruction can help.

```
ADR    Rd, label                   @ load Rd from location label
```

The assembler turns this instruction into the following (the assembler calculates offset)

```
ADD    PC, #offset
```

MOV

A MOV instruction can move data between registers, or from an immediate to a register. A MVN instruction also moves information, but does a bit-wise negation in the process.

```
MOV   Rd, Rm          @ Rd <= Rm.
MOVS  Rd, Rm          @ Rd <= Rm and updates status bits.
MOV   Rd, #0xFFF      @ Rd <= FFF (up to 12-bit immediates can be used).
MOVW Rd, #0xFFFF      @ Rd <= FFFF (up to 16-bit immediates can be used)
MVN   Rd, Rs          @ Rd <= NOT Rm (bit-wise inversion).
MOVT Rd, #0xAA        @ Top halfword of Rd <= AA; bottom half unchanged
```

There are other flavors of MOV instructions to move data into special registers and to coprocessors. You can get more information about these move instructions from the text book of from the Arm Architectural Reference starting on page A8-484.

Branch

Conditional and unconditional braches in program execution create loops or if-then constructs. “Branch with Link” (BL) additionally copies the PC to the Link register, so a subroutine return can resume execution immediately after where it was called. “Branch and Exchange” braches to an address stored in a register.

```
B  <label>      @ Unconditional branch to instruction after label
BNE  <label>    @ Branch to instruction after label
BL <label>      @ Branch to label and copy PC into R14 (the LR)
BX LR           @ Branch to location stored in LR (R14)
```

Examples:

```
Loop_point:   @ label
        LDR   r0, [r1]
        B     loop_point
```

```
              MOV   R0, #10
Loop_point1:  SUBS  r0, r0, #1
              BNE   loop_point
```

Conditional Branches

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	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	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	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

ADD

The ADD and ADC (add with carry) instructions add the contents in two 32-bit “source” registers and place the result in a 32-bit “destination” register (the destination register can be the same as one of the sources).

```
ADD     r0, r1, r2              @ r0 <= r1 + r2
ADC     r0, r1, r2              @ r0 <= r1 + r2 + C
ADCS    r3, r2, #0xABC          @ r3 <= r2 + 0xABC + C and set status bits (up to 12 bit imm)
ADDEQ   r0, r1, r2              @ r0 <=  r1 + r2 if Z bit is set
ADD     r0, r1, r2, LSL #0x4    @ r0 <= r1 + r2 shifted left four bits
ADD     r0, r1, r2, ASR r3      @ r0 <= r1 + r2 shifted left (sign ext.) by number in r3
ADD     r1, r1, #1              @ increment value in r1
```

More information about add instructions in text book and the Arm Architectural Reference starting on page A8-300.

Subtract

The SUB and SBC (subtract with NOT of carry bit), and reverse-subtract RSB and RSC instructions subtract the contents of one 32-bit register from another register, and place the result in a 32-bit destination register (the destination register can be the same as one of the sources).

```
SUB     r0, r1, r2              @ r0 <= r1 - r2
SBCS    r0, r1, r2              @ r0 <= r1 - r2 - Not C, and update status bits
SUB     r0, r1, #0xABC          @ r0 <= r1 - 0xABC (up to 12 bit immediate)
SUBNE   r0, r1, r2              @ r0 <=  r1 - r2 if Z bit is not set
SUB     r0, r1, r2, LSL #0x4    @ r0 <= r1 - r2 shifted left four bits
RSB     r0, r1, r2              @ r0 <= r2 – r1
SUB     r2, r2, #1              @ Decrement value in r2
```

More information about sub instructions in text book and the Arm Architectural Reference.

Shift

The ARM can do arithmetic and logical shifts by up to 32 bits. Arithmetic shifts are right-shift and sign-extended (i.e, the sign bit is placed in all vacated bits). Logical shift right or left (LSR or LSL) can shift up to 32 bits, and a ‘0’ will be placed in all vacated bits. The number of bits to shift can be an immediate or placed in a register.

```
ASR   r0, r1 ,#0x04      @ r0 <= r1 contents shifted right four bits with sign extend
ASR   r0, r1, r2         @ r0 <= r1 contents shifted right by the number of bits in the bottom
                                 of r2, sign extends, and places result in r0
LSL   r0, r1, #0x06      @ r0 <= r1 shifted left 6 bits with 0 fill
LSR   r0, r1, r2         @ r0 <= r1 shifted right by number of bits in bottom of r2, 0 fill
```

Comparisons

CMP compares by subtracting the operands, updating the status bits, and discarding the results. No need to use “S” mnemonic extension. CMN adds operands, updates status, and discards results. TST (test) ANDs operand 1 with operand 2, updates status bits, and discards results. TEQ (test equivalence) XORs operand 1 with operand 2, updates status bits, and discards results.

```
CMP   r0, #0ximm        @ Subtract imm (12 bit) from r0, discard results, update status bits
CMP   r0, r1            @ Subtract r1 from r0, discard results, update status bits
CMP   r0, r1, LSL #0x06 @ Subtract left-shifted r1 from r0, discard results, update status bits
CMP   r0, r1, ROR r2    @ Subtract r1 (right-rotated by number in r2) from r0, discard, update
CMN   r0, #0ximm        @ Add imm to r0, discard results, update status bits
CMN   r0, r1            @ Add r1 and r0, discard results, update status bits
TST   r0, #1            @ ANDs r0 with 1, discard results, update status bits (why?)
TEQ   r2, r3            @ Bitwise XOR of r2 and r3, discard results, updates status bits
TEQ   r2, #5            @ Bitwise XOR of r2 and 5 (Z = 1 if equal)
```

Logical Operations

The AND, BIC, EOR, and ORR data processing operators all use the same general addressing modes as the other data processing instructions.

```
AND   r0, r1, #0ximm   @ Bitwise AND r1 with 12-bit immediate and write result to r0
ANDS r0, r1, r2          @ Bitwise AND r1 with r2, write result to r0, update status bits
AND   r0, r1, r2 LSR r3 @ Bitwise AND r1 with left-shifted r2, write result to r0
BIC    r0, r1, #0ximm   @ Bitwise AND r1 with inverse of imm, write to r0, update status bits
BIC    r0, r1, r2, LSL 0x#2 @ Bitwise AND r1 with left-shifted r2, write to r0, update status bits
EORS  r0, r1, #0ximm   @ Bitwise XOR r1 with 12-b imm, write result to r0, update status bits
EOR    r0, r1, r2, LSR r3 @ Bitwise XOR r1 with r2 shifted left by r3, write to r0
ORR    r2, r3, #0xAA     @ Bitwise XOR of r3 and AA, write to r2, update status bits
```

Multiplication

ARM has three multiplication instructions: MUL to multiply two 32-bit registers, MLA (multiply and accumulate) to multiply two 32-bit registers and add a third register; and MLS (multiply and subtract) to multiply two registers and subtract the result from a third register. For each instruction, the destination register holds the least-significant 32 bits of the result.

```
MUL    r0, r1, r2       @ r0 <= r1 x r2
MUL    r0, r1           @ r0 <= r1 x r0
MULS   r0, r1, r2       @ r0 <= r1 x r2 and status bits are updated
MLA    r0, r1, r2, r3   @ r0 <= r1 x r2 + r3
MLAS   r0, r1, r2, r3   @ r0 <= r3 - r1 x r2
MLS    r0, r1, r2, r3
```