# TRINITY COLLEGE DUBLIN
# THE UNIVERSITY OF DUBLIN

## Faculty of Engineering, Mathematics and Science

### School of Computer Science & Statistics

**Integrated Engineering** **Trinity Term 2015**
**Year 3 Annual Examinations**

## 3D2 Microprocessor Systems 2

**Thursday 30 April 2015** **Sports Centre** **09:30 – 11:30**

**Dr Mike Brady**

---

**Instructions to Candidates:**
Attempt **two** questions. All questions carry equal marks. Each question is scored out of a total of 20 marks.

You may not start this examination until you are instructed to do so by the Invigilator.

**Materials permitted for this examination:**
A two-page document, entitled *"Pthread Types and Function Prototypes"* accompanies this examination paper.

Non-programmable calculators are permitted for this examination — please indicate the make and model of your calculator on each answer book used.

1. (a) Give an account of pipelining and how it can improve the performance of a processor. In your answer, explain how and why the full potential of pipelines is not always achievable. [10 marks]

   (b) Explain how a cache works. Give a worked example of how cache would speed up the execution of a program loop designed to calculate the average of 1,000 integers stored sequentially in memory, given 10 ns memory access time, cache with an idealised access time of 0 ns and a processor that can execute a complete instruction every 1 ns, provided the instruction and data are in cache. [10 marks]

2. (a) Compare and contrast polled and interrupt-driven I/O. In your answer, highlight the advantages and drawbacks of each approach. For example, given that polling is so simple, why is it not used all the time, and given that interrupt handling is so processor efficient, why is it not used all the time? [8 marks]

   (b) A system has four push-button switches S1, S2, S3 and S4 and four light-emitting diodes (LEDs) L1, L2, L3 and L4, much as you would have seen on the ARM boards. Write a fragment of ARM assembly language, complete with any equates and memory reservations needed, so that whenever a switch is pressed, the state of the corresponding LED should toggle, that is, it should change state — if it was lit it should go dim, and if it was dim it should be lit.

   Assume the switches are connected to location 0xE0040002 in bit positions 0, 1, 2 and 3 respectively such that the a bit is 1 if its switch is pressed and 0 otherwise, with no switch bounce.

   Assume that the LEDs are connected to location 0xE0040004 in bit positions 0, 1, 2 and 3 respectively. To light a LED, set its bit to 1; to make it dim, set its bit to 0. Assume the other four bits in that location are "don't cares" — i.e. it doesn't matter what values you set them to. Assume that 0xE0040004 is write-only, i.e. that you can't read back values from it, and explain why this assumption is important. [12 marks]

3. (a) What is the purpose of different modes of operation of a processor, such as the different modes that are provided on the ARM processor? What are the extra registers for? [4 marks]

   (b) What are desirable properties of an interrupt handler and why are they desirable? [4 marks]

   (c) Write an interrupt handler which is called every 1 ms and which provides a debounced version of a push button switch's input. When pressed or release, the switch's output may bounce between its initial value and its final value for up to 7 ms. The switch is connected to bit 0 of location 0xE0040006 and the debounced version of it should be maintained at bit 0 of a location in RAM labelled `CLEANSWITCH`. [12 marks]

# ASCII Code

| | Column Number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Row Number | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0000 | *NUL* | *DLE* | ◊ | 0 | @ | P | ` | p |
| 0001 | *SOH* | *DC1* | ! | 1 | A | Q | a | q |
| 0010 | *STX* | *DC2* | " | 2 | B | R | b | r |
| 0011 | *ETX* | *DC3* | # | 3 | C | S | c | s |
| 0100 | *EOT* | *DC4* | $ | 4 | D | T | d | t |
| 0101 | *ENQ* | *NAK* | % | 5 | E | U | e | u |
| 0110 | *ACK* | *SYN* | & | 6 | F | V | f | v |
| 0111 | *BELL* | *ETB* | ' | 7 | G | W | g | w |
| 1000 | *BS* | *CAN* | ( | 8 | H | X | h | x |
| 1001 | *HT* | *EM* | ) | 9 | I | Y | i | y |
| 1010 | *LF* | *SUB* | * | : | J | Z | j | z |
| 1011 | *VT* | *ESC* | + | ; | K | [ | k | { |
| 1100 | *FF* | *FS* | , | < | L | \ | l | \| |
| 1101 | *CR* | *GS* | - | = | M | ] | m | } |
| 1110 | *SO* | *RS* | . | > | N | ^ | n | ~ |
| 1111 | *SI* | *US* | / | ? | O | _ | o | *DEL* |

The ASCII code of a character is found by combining its Column Number (given in 3-bit binary) with its Row Number (given in 4-bit binary).
The Column Number forms bits 6, 5 and 4 of the ASCII, and the Row Number forms bits 3, 2, 1 and 0 of the ASCII.

Example of use: to get ASCII code for letter "n", locate it in Column **110**, Row **1110.** Hence its ASCII code is **1101110**.

The **Control Code** mnemonics are given in italics above; e.g. *CR* for Carriage Return, *LF* for Line Feed, *BELL* for the Bell, *DEL* for Delete.

The Space is ASCII 0100000, and is shown as ◊ here.

# ARM® Instruction Set Quick Reference Card

| Key to Tables | |
|---|---|
| `{cond}` | Refer to Table **Condition Field**. Omit for unconditional execution. |
| `<Operand2>` | Refer to Table **Flexible Operand 2**. Shift and rotate are only available as part of Operand2. |
| `<fields>` | Refer to Table **PSR fields**. |
| `<PSR>` | Either CPSR (Current Processor Status Register) or SPSR (Saved Processor Status Register) |
| `{S}` | Updates condition flags if S present. |
| `C*, V*` | Flag is unpredictable in Architecture v4 and earlier, unchanged in Architecture v5 and later. |
| `Q` | Sticky flag. Always updates on overflow (no S option). Read and reset using MRS and MSR. |
| `GE` | Four Greater than or Equal flags. Always updated by parallel adds and subtracts. |
| `x,y` | B meaning half-register [15:0], or T meaning [31:16]. |
| `<immed_8r>` | A 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits. |
| `{X}` | RsX is Rs rotated 16 bits if X present. Otherwise, RsX is Rs. |
| `<prefix>` | Refer to Table **Prefixes for Parallel instructions** |
| `<p_mode>` | Refer to Table **Processor Modes** |
| `R13m` | R13 for the processor mode specified by `<p_mode>` |

| | |
|---|---|
| `{endianness}` | Can be BE (Big Endian) or LE (Little Endian). |
| `<a_mode2>` | Refer to Table **Addressing Mode 2.** |
| `<a_mode2P>` | Refer to Table **Addressing Mode 2 (Post-indexed only).** |
| `<a_mode3>` | Refer to Table **Addressing Mode 3.** |
| `<a_mode4L>` | Refer to Table **Addressing Mode 4 (Block load or Stack pop).** |
| `<a_mode4S>` | Refer to Table **Addressing Mode 4 (Block store or Stack push).** |
| `<a_mode5>` | Refer to Table **Addressing Mode 5.** |
| `<reglist>` | A comma-separated list of registers, enclosed in braces { and }. |
| `<reglist-PC>` | As `<reglist>`, must not include the PC. |
| `<reglist+PC>` | As `<reglist>`, including the PC. |
| `{!}` | Updates base register after data transfer if ! present. |
| `+/-` | + or –. (+ may be omitted.) |
| `§` | Refer to Table **ARM architecture versions.** |
| `<iflags>` | Interrupt flags. One or more of `a`, `i`, `f` (abort, interrupt, fast interrupt). |
| `{R}` | Rounds result to nearest if R present, otherwise truncates result. |

| Operation | | § | Assembler | S updates | Q | Action |
|---|---|---|---|---|---|---|
| **Arithmetic** | Add | | `ADD{cond}{S} Rd, Rn, <Operand2>` | N Z C V | | Rd := Rn + Operand2 |
| | with carry | | `ADC{cond}{S} Rd, Rn, <Operand2>` | N Z C V | | Rd := Rn + Operand2 + Carry |
| | saturating | 5E | `QADD{cond} Rd, Rm, Rn` | | Q | Rd := SAT(Rm + Rn) |
| | double saturating | 5E | `QDADD{cond} Rd, Rm, Rn` | | Q | Rd := SAT(Rm + SAT(Rn * 2)) |
| | Subtract | | `SUB{cond}{S} Rd, Rn, <Operand2>` | N Z C V | | Rd := Rn – Operand2 |
| | with carry | | `SBC{cond}{S} Rd, Rn, <Operand2>` | N Z C V | | Rd := Rn – Operand2 – NOT(Carry) |
| | reverse subtract | | `RSB{cond}{S} Rd, Rn, <Operand2>` | N Z C V | | Rd := Operand2 – Rn |
| | reverse subtract with carry | | `RSC{cond}{S} Rd, Rn, <Operand2>` | N Z C V | | Rd := Operand2 – Rn – NOT(Carry) |
| | saturating | 5E | `QSUB{cond} Rd, Rm, Rn` | | Q | Rd := SAT(Rm – Rn) |
| | double saturating | 5E | `QDSUB{cond} Rd, Rm, Rn` | | Q | Rd := SAT(Rm – SAT(Rn * 2)) |
| | Multiply | 2 | `MUL{cond}{S} Rd, Rm, Rs` | N Z C* | | Rd := (Rm * Rs)[31:0] |
| | and accumulate | 2 | `MLA{cond}{S} Rd, Rm, Rs, Rn` | N Z C* | | Rd := ((Rm * Rs) + Rn)[31:0] |
| | unsigned long | M | `UMULL{cond}{S} RdLo, RdHi, Rm, Rs` | N Z C* V* | | RdHi,RdLo := unsigned(Rm * Rs) |
| | unsigned accumulate long | M | `UMLAL{cond}{S} RdLo, RdHi, Rm, Rs` | N Z C* V* | | RdHi,RdLo := unsigned(RdHi,RdLo + Rm * Rs) |
| | unsigned double accumulate long | 6 | `UMAAL{cond} RdLo, RdHi, Rm, Rs` | | | RdHi,RdLo := unsigned(RdHi + RdLo + Rm * Rs) |
| | Signed multiply long | M | `SMULL{cond}{S} RdLo, RdHi, Rm, Rs` | N Z C* V* | | RdHi,RdLo := signed(Rm * Rs) |
| | and accumulate long | M | `SMLAL{cond}{S} RdLo, RdHi, Rm, Rs` | N Z C* V* | | RdHi,RdLo := signed(RdHi,RdLo + Rm * Rs) |
| | 16 * 16 bit | 5E | `SMULxy{cond} Rd, Rm, Rs` | | | Rd := Rm[x] * Rs[y] |
| | 32 * 16 bit | 5E | `SMULWy{cond} Rd, Rm, Rs` | | | Rd := (Rm * Rs[y])[47:16] |
| | 16 * 16 bit and accumulate | 5E | `SMLAxy{cond} Rd, Rm, Rs, Rn` | | Q | Rd := Rn + Rm[x] * Rs[y] |
| | 32 * 16 bit and accumulate | 5E | `SMLAWy{cond} Rd, Rm, Rs, Rn` | | Q | Rd := Rn + (Rm * Rs[y])[47:16] |
| | 16 * 16 bit and accumulate long | 5E | `SMLALxy{cond} RdLo, RdHi, Rm, Rs` | | | RdHi,RdLo := RdHi,RdLo + Rm[x] * Rs[y] |
| | Dual signed multiply, add | 6 | `SMUAD{X}{cond} Rd, Rm, Rs` | | Q | Rd := Rm[15:0] * RsX[15:0] + Rm[31:16] * RsX[31:16] |
| | and accumulate | 6 | `SMLAD{X}{cond} Rd, Rm, Rs, Rn` | | Q | Rd := Rn + Rm[15:0] * RsX[15:0] + Rm[31:16] * RsX[31:16] |
| | and accumulate long | 6 | `SMLALD{X}{cond} RdHi, RdLo, Rm, Rs` | | Q | RdHi,RdLo := RdHi,RdLo + Rm[15:0] * RsX[15:0] + Rm[31:16] * RsX[31:16] |
| | Dual signed multiply, subtract | 6 | `SMUSD{X}{cond} Rd, Rm, Rs` | | Q | Rd := Rm[15:0] * RsX[15:0] – Rm[31:16] * RsX[31:16] |
| | and accumulate | 6 | `SMLSD{X}{cond} Rd, Rm, Rs, Rn` | | Q | Rd := Rn + Rm[15:0] * RsX[15:0] – Rm[31:16] * RsX[31:16] |
| | and accumulate long | 6 | `SMLSLD{X}{cond} RdHi, RdLo, Rm, Rs` | | Q | RdHi,RdLo := RdHi,RdLo + Rm[15:0] * RsX[15:0] – Rm[31:16] * RsX[31:16] |
| | Signed most significant word multiply | 6 | `SMMUL{R}{cond} Rd, Rm, Rs` | | | Rd := (Rm * Rs)[63:32] |
| | and accumulate | 6 | `SMMLA{R}{cond} Rd, Rm, Rs, Rn` | | | Rd := Rn + (Rm * Rs)[63:32] |
| | and subtract | 6 | `SMMLS{R}{cond} Rd, Rm, Rs, Rn` | | | Rd := Rn – (Rm * Rs)[63:32] |
| | Multiply with internal 40-bit accumulate | XS | `MIA{cond} Ac, Rm, Rs` | | | Ac := Ac + Rm * Rs |
| | packed halfword | XS | `MIAPH{cond} Ac, Rm, Rs` | | | Ac := Ac + Rm[15:0] * Rs[15:0] + Rm[31:16] * Rs[31:16] |
| | halfword | XS | `MIAxy{cond} Ac, Rm, Rs` | | | Ac := Ac + Rm[x] * Rs[y] |
| | Count leading zeroes | 5 | `CLZ{cond} Rd, Rm` | | | Rd := number of leading zeroes in Rm |

# ARM Addressing Modes
Quick Reference Card

| Operation | | § | Assembler | S updates | Q | GE | Action |
|---|---|---|---|---|---|---|---|
| **Parallel arithmetic** | Halfword-wise addition | 6 | `<prefix>ADD16{cond} Rd, Rn, Rm` | | | GE | Rd[31:16] := Rn[31:16] + Rm[31:16], Rd[15:0] := Rn[15:0] + Rm[15:0] |
| | Halfword-wise subtraction | 6 | `<prefix>SUB16{cond} Rd, Rn, Rm` | | | GE | Rd[31:16] := Rn[31:16] – Rm[31:16], Rd[15:0] := Rn[15:0] – Rm[15:0] |
| | Byte-wise addition | 6 | `<prefix>ADD8{cond} Rd, Rn, Rm` | | | GE | Rd[31:24] := Rn[31:24] + Rm[31:24], Rd[23:16] := Rn[23:16] + Rm[23:16], Rd[15:8] := Rn[15:8] + Rm[15:8], Rd[7:0] := Rn[7:0] + Rm[7:0] |
| | Byte-wise subtraction | 6 | `<prefix>SUB8{cond} Rd, Rn, Rm` | | | GE | Rd[31:24] := Rn[31:24] – Rm[31:24], Rd[23:16] := Rn[23:16] – Rm[23:16], Rd[15:8] := Rn[15:8] – Rm[15:8], Rd[7:0] := Rn[7:0] – Rm[7:0] |
| | Halfword-wise exchange, add, subtract | 6 | `<prefix>ADDSUBX{cond} Rd, Rn, Rm` | | | GE | Rd[31:16] := Rn[31:16] + Rm[15:0], Rd[15:0] := Rn[15:0] – Rm[31:16] |
| | Halfword-wise exchange, subtract, add | 6 | `<prefix>SUBADDX{cond} Rd, Rn, Rm` | | | GE | Rd[31:16] := Rn[31:16] – Rm[15:0], Rd[15:0] := Rn[15:0] + Rm[31:16] |
| | Unsigned sum of absolute differences | 6 | `USAD8{cond} Rd, Rm, Rs` | | | | Rd := Abs(Rm[31:24] – Rs[31:24]) + Abs(Rm[23:16] – Rs[23:16]) + Abs(Rm[15:8] – Rs[15:8]) + Abs(Rm[7:0] – Rs[7:0]) |
| | and accumulate | 6 | `USADA8{cond} Rd, Rm, Rs, Rn` | | | | Rd := Rn + Abs(Rm[31:24] – Rs[31:24]) + Abs(Rm[23:16] – Rs[23:16]) + Abs(Rm[15:8] – Rs[15:8]) + Abs(Rm[7:0] – Rs[7:0]) |
| **Move** | Move | | `MOV{cond}{S} Rd, <Operand2>` | N Z C | | | Rd := Operand2 |
| | NOT | | `MVN{cond}{S} Rd, <Operand2>` | N Z C | | | Rd := 0xFFFFFFFF EOR Operand2 |
| | PSR to register | 3 | `MRS{cond} Rd, <PSR>` | | | | Rd := PSR |
| | register to PSR | 3 | `MSR{cond} <PSR>_<fields>, Rm` | | | | PSR := Rm (selected bytes only) |
| | immediate to PSR | 3 | `MSR{cond} <PSR>_<fields>, #<immed_8r>` | | | | PSR := immed_8r (selected bytes only) |
| | 40-bit accumulator to register | XS | `MRA{cond} RdLo, RdHi, Ac` | | | | RdLo := Ac[31:0], RdHi := Ac[39:32] |
| | register to 40-bit accumulator | XS | `MAR{cond} Ac, RdLo, RdHi` | | | | Ac[31:0] := RdLo, Ac[39:32] := RdHi |
| | Copy | 6 | `CPY{cond} Rd, <Operand2>` | | | | Rd := Operand2 |
| **Logical** | Test | | `TST{cond} Rn, <Operand2>` | N Z C | | | Update CPSR flags on Rn AND Operand2 |
| | Test equivalence | | `TEQ{cond} Rn, <Operand2>` | N Z C | | | Update CPSR flags on Rn EOR Operand2 |
| | AND | | `AND{cond}{S} Rd, Rn, <Operand2>` | N Z C | | | Rd := Rn AND Operand2 |
| | EOR | | `EOR{cond}{S} Rd, Rn, <Operand2>` | N Z C | | | Rd := Rn EOR Operand2 |
| | ORR | | `ORR{cond}{S} Rd, Rn, <Operand2>` | N Z C | | | Rd := Rn OR Operand2 |
| | Bit Clear | | `BIC{cond}{S} Rd, Rn, <Operand2>` | N Z C | | | Rd := Rn AND NOT Operand2 |
| **Compare** | Compare | | `CMP{cond} Rn, <Operand2>` | N Z C V | | | Update CPSR flags on Rn – Operand2 |
| | negative | | `CMN{cond} Rn, <Operand2>` | N Z C V | | | Update CPSR flags on Rn + Operand2 |
| **Saturate** | Signed saturate word, right shift | 6 | `SSAT{cond} Rd, #<sat>, Rm{, ASR <sh>}` | | Q | | Rd := SignedSat((Rm ASR sh), sat). `<sat>` range 0-31, `<sh>` range 1-32. |
| | left shift | | `SSAT{cond} Rd, #<sat>, Rm{, LSL <sh>}` | | Q | | Rd := SignedSat((Rm LSL sh), sat). `<sat>` range 0-31, `<sh>` range 0-31. |
| | Signed saturate two halfwords | 6 | `SSAT16{cond} Rd, #<sat>, Rm` | | Q | | Rd[31:16] := SignedSat(Rm[31:16], sat), Rd[15:0] := SignedSat(Rm[15:0], sat). `<sat>` range 0-15. |
| | Unsigned saturate word, right shift | 6 | `USAT{cond} Rd, #<sat>, Rm{, ASR <sh>}` | | Q | | Rd := UnsignedSat((Rm ASR sh), sat). `<sat>` range 0-31, `<sh>` range 1-32. |
| | left shift | | `USAT{cond} Rd, #<sat>, Rm{, LSL <sh>}` | | Q | | Rd := UnsignedSat((Rm LSL sh), sat). `<sat>` range 0-31, `<sh>` range 0-31. |
| | Unsigned saturate two halfwords | 6 | `USAT16{cond} Rd, #<sat>, Rm` | | Q | | Rd[31:16] := UnsignedSat(Rm[31:16], sat), Rd[15:0] := UnsignedSat(Rm[15:0], sat). `<sat>` range 0-15. |

## ARM Instruction Set
## Quick Reference Card

| Operation | | § | Assembler | Action | Notes |
|---|---|---|---|---|---|
| **Pack** | Pack halfword bottom + top | 6 | PKHBT{cond} Rd, Rn, Rm{, LSL #<sh>} | Rd[15:0] := Rn[15:0], Rd[31:16] := (Rm LSL sh)[31:16]. sh 0-31. | |
| | Pack halfword top + bottom | 6 | PKHTB{cond} Rd, Rn, Rm{, ASR #<sh>} | Rd[31:16] := Rn[31:16], Rd[15:0] := (Rm ASR sh)[15:0]. sh 1-32. | |
| **Signed extend** | Halfword to word | 6 | SXTH{cond} Rd, Rm{, ROR #<sh>} | Rd[31:0] := SignExtend((Rm ROR (8 * sh))[15:0]). sh 0-3. | |
| | Two bytes to halfwords | 6 | SXTB16{cond} Rd, Rm{, ROR #<sh>} | Rd[31:16] := SignExtend((Rm ROR (8 * sh))[23:16]), Rd[15:0] := SignExtend((Rm ROR (8 * sh))[7:0]). sh 0-3. | |
| | Byte to word | 6 | SXTB{cond} Rd, Rm{, ROR #<sh>} | Rd[31:0] := SignExtend((Rm ROR (8 * sh))[7:0]). sh 0-3. | |
| **Unsigned extend** | Halfword to word | 6 | UXTH{cond} Rd, Rm{, ROR #<sh>} | Rd[31:0] := ZeroExtend((Rm ROR (8 * sh))[15:0]). sh 0-3. | |
| | Two bytes to halfwords | 6 | UXTB16{cond} Rd, Rm{, ROR #<sh>} | Rd[31:16] := ZeroExtend((Rm ROR (8 * sh))[23:16]), Rd[15:0] := ZeroExtend((Rm ROR (8 * sh))[7:0]). sh 0-3. | |
| | Byte to word | 6 | UXTB{cond} Rd, Rm{, ROR #<sh>} | Rd[31:0] := ZeroExtend((Rm ROR (8 * sh))[7:0]). sh 0-3. | |
| **Signed extend with add** | Halfword to word, add | 6 | SXTAH{cond} Rd, Rn, Rm{, ROR #<sh>} | Rd[31:0] := Rn[31:0] + SignExtend((Rm ROR (8 * sh))[15:0]). sh 0-3. | |
| | Two bytes to halfwords, add | 6 | SXTAB16{cond} Rd, Rn, Rm{, ROR #<sh>} | Rd[31:16] := Rn[31:16] + SignExtend((Rm ROR (8 * sh))[23:16]), Rd[15:0] := Rn[15:0] + SignExtend((Rm ROR (8 * sh))[7:0]). sh 0-3. | |
| | Byte to word, add | 6 | SXTAB{cond} Rd, Rn, Rm{, ROR #<sh>} | Rd[31:0] := Rn[31:0] + SignExtend((Rm ROR (8 * sh))[7:0]). sh 0-3. | |
| **Unsigned extend with add** | Halfword to word, add | 6 | UXTAH{cond} Rd, Rn, Rm{, ROR #<sh>} | Rd[31:0] := Rn[31:0] + ZeroExtend((Rm ROR (8 * sh))[15:0]). sh 0-3. | |
| | Two bytes to halfwords, add | 6 | UXTAB16{cond} Rd, Rn, Rm{, ROR #<sh>} | Rd[31:16] := Rn[31:16] + ZeroExtend((Rm ROR (8 * sh))[23:16]), Rd[15:0] := Rn[15:0] + ZeroExtend((Rm ROR (8 * sh))[7:0]). sh 0-3. | |
| | Byte to word, add | 6 | UXTAB{cond} Rd, Rn, Rm{, ROR #<sh>} | Rd[31:0] := Rn[31:0] + ZeroExtend((Rm ROR (8 * sh))[7:0]). sh 0-3. | |
| **Reverse bytes** | In word | 6 | REV{cond} Rd, Rm | Rd[31:24] := Rm[7:0], Rd[23:16] := Rm[15:8], Rd[15:8] := Rm[23:16], Rd[7:0] := Rm[31:24] | |
| | In both halfwords | 6 | REV16{cond} Rd, Rm | Rd[15:8] := Rm[7:0], Rd[7:0] := Rm[15:8], Rd[31:24] := Rm[23:16], Rd[23:16] := Rm[31:24] | |
| | In low halfword, sign extend | 6 | REVSH{cond} Rd, Rm | Rd[15:8] := Rm[7:0], Rd[7:0] := Rm[15:8], Rd[31:16] := Rm[7] * &FFFF | |
| **Select** | Select bytes | 6 | SEL{cond} Rd, Rn, Rm | Rd[7:0] := Rn[7:0] if GE[0] = 1, else Rd[7:0] := Rm[7:0] Bits[15:8], [23:16], [31:24] selected similarly by GE[1], GE[2], GE[3] | |
| **Branch** | Branch | | B{cond} label | R15 := label | label must be within ±32Mb of current instruction. |
| | with link | | BL{cond} label | R14 := address of next instruction, R15 := label | label must be within ±32Mb of current instruction. |
| | and exchange | 4T,5 | BX{cond} Rm | R15 := Rm, Change to Thumb if Rm[0] is 1 | |
| | with link and exchange (1) | 5T | BLX label | R14 := address of next instruction, R15 := label, Change to Thumb | Cannot be conditional. label must be within ±32Mb of current instruction. |
| | with link and exchange (2) | 5 | BLX{cond} Rm | R14 := address of next instruction, R15 := Rm[31:1] Change to Thumb if Rm[0] is 1 | |
| | and change to Java state | 5J, 6 | BXJ{cond} Rm | Change to Java state | |
| **Processor state change** | Change processor state | 6 | CPSID <iflags> {, #<p_mode>} | Disable specified interrupts, optional change mode. | Cannot be conditional. |
| | | 6 | CPSIE <iflags> {, #<p_mode>} | Enable specified interrupts, optional change mode. | Cannot be conditional. |
| | Change processor mode | 6 | CPS #<p_mode> | | Cannot be conditional. |
| | Set endianness | 6 | SETEND <endianness> | Sets endianness for loads and saves. <endianness> can be BE (Big Endian) or LE (Little Endian). | Cannot be conditional. |
| | Store return state | 6 | SRS<a_mode4S> #<p_mode>{!} | [R13m] := R14, [R13m + 4] := CPSR | Cannot be conditional. |
| | Return from exception | 6 | RFE<a_mode4L> Rn{!} | PC := [Rn], CPSR := [Rn + 4] | Cannot be conditional. |
| | Breakpoint | 5 | BKPT <immed_16> | Prefetch abort *or* enter debug state. | Cannot be conditional. |
| **Software interrupt** | Software interrupt | | SWI{cond} <immed_24> | Software interrupt processor exception. | 24-bit value encoded in instruction. |
| **No Op** | No operation | 5 | NOP | None | |

## ARM Addressing Modes
## Quick Reference Card

| Operation | | § | Assembler | Action | Notes |
|---|---|---|---|---|---|
| **Load** | Word | | `LDR{cond} Rd, <a_mode2>` | Rd := [address] | Rd must not be R15. |
| | User mode privilege | | `LDR{cond}T Rd, <a_mode2P>` | | Rd must not be R15. |
| | branch (§ 5T: and exchange) | | `LDR{cond} R15, <a_mode2>` | R15 := [address][31:1]<br>(§ 5T: Change to Thumb if [address][0] is 1) | |
| | Byte | | `LDR{cond}B Rd, <a_mode2>` | Rd := ZeroExtend[byte from address] | Rd must not be R15. |
| | User mode privilege | | `LDR{cond}BT Rd, <a_mode2P>` | | Rd must not be R15. |
| | signed | 4 | `LDR{cond}SB Rd, <a_mode3>` | Rd := SignExtend[byte from address] | Rd must not be R15. |
| | Halfword | 4 | `LDR{cond}H Rd, <a_mode3>` | Rd := ZeroExtent[halfword from address] | Rd must not be R15. |
| | signed | 4 | `LDR{cond}SH Rd, <a_mode3>` | Rd := SignExtend[halfword from address] | Rd must not be R15. |
| | Doubleword | 5E* | `LDR{cond}D Rd, <a_mode3>` | Rd := [address], R(d+1) := [address + 4] | Rd must be even, and not R14. |
| **Load multiple** | Pop, or Block data load | | `LDM{cond}<a_mode4L> Rn{!}, <reglist-PC>` | Load list of registers from [Rn] | |
| | return (and exchange) | | `LDM{cond}<a_mode4L> Rn{!}, <reglist+PC>` | Load registers, R15 := [address][31:1]<br>(§ 5T: Change to Thumb if [address][0] is 1) | |
| | and restore CPSR | | `LDM{cond}<a_mode4L> Rn{!}, <reglist+PC>^` | Load registers, branch (§ 5T: and exchange), CPSR := SPSR | Use from exception modes only. |
| | User mode registers | | `LDM{cond}<a_mode4L> Rn, <reglist-PC>^` | Load list of User mode registers from [Rn] | Use from privileged modes only. |
| **Soft preload** | Memory system hint | 5E* | `PLD <a_mode2>` | Memory may prepare to load from address | Cannot be conditional. |
| **Load exclusive** | Semaphore operation | 6 | `LDREX{cond} Rd, [Rn]` | Rd := [Rn], tag address as exclusive access<br>Outstanding tag set if not shared address | Rd, Rn must not be R15. |
| **Store** | Word | | `STR{cond} Rd, <a_mode2>` | [address] := Rd | |
| | User mode privilege | | `STR{cond}T Rd, <a_mode2P>` | [address] := Rd | |
| | Byte | | `STR{cond}B Rd, <a_mode2>` | [address][7:0] := Rd[7:0] | |
| | User mode privilege | | `STR{cond}BT Rd, <a_mode2P>` | [address][7:0] := Rd[7:0] | |
| | Halfword | 4 | `STR{cond}H Rd, <a_mode3>` | [address][15:0] := Rd[15:0] | |
| | Doubleword | 5E* | `STR{cond}D Rd, <a_mode3>` | [address] := Rd, [address + 4] := R(d+1) | Rd must be even, and not R14. |
| **Store multiple** | Push, or Block data store | | `STM{cond}<a_mode4S> Rn{!}, <reglist>` | Store list of registers to [Rn] | |
| | User mode registers | | `STM{cond}<a_mode4S> Rn{!}, <reglist>^` | Store list of User mode registers to [Rn] | Use from privileged modes only. |
| **Store exclusive** | Semaphore operation | 6 | `STREX{cond} Rd, Rm, [Rn]` | [Rn] := Rm if allowed,<br>Rd := 0 if successful, else 1 | Rd, Rm, Rn must not be R15. |
| **Swap** | Word | 3 | `SWP{cond} Rd, Rm, [Rn]` | temp := [Rn], [Rn] := Rm, Rd := temp | |
| | Byte | 3 | `SWP{cond}B Rd, Rm, [Rn]` | temp := ZeroExtend([Rn][7:0]),<br>[Rn][7:0] := Rm[7:0], Rd := temp | |

# ARM Addressing Modes
# Quick Reference Card

**Addressing Mode 2 - Word and Unsigned Byte Data Transfer**

| | | | |
|---|---|---|---|
| Pre-indexed | Immediate offset | `[Rn, #+/-<immed_12>]{!}` | |
| | Zero offset | `[Rn]` | Equivalent to [Rn,#0] |
| | Register offset | `[Rn, +/-Rm]{!}` | |
| | Scaled register offset | `[Rn, +/-Rm, LSL #<shift>]{!}` | Allowed shifts 0-31 |
| | | `[Rn, +/-Rm, LSR #<shift>]{!}` | Allowed shifts 1-32 |
| | | `[Rn, +/-Rm, ASR #<shift>]{!}` | Allowed shifts 1-32 |
| | | `[Rn, +/-Rm, ROR #<shift>]{!}` | Allowed shifts 1-31 |
| | | `[Rn, +/-Rm, RRX]{!}` | |
| Post-indexed | Immediate offset | `[Rn], #+/-<immed_12>` | |
| | Register offset | `[Rn], +/-Rm` | |
| | Scaled register offset | `[Rn], +/-Rm, LSL #<shift>` | Allowed shifts 0-31 |
| | | `[Rn], +/-Rm, LSR #<shift>` | Allowed shifts 1-32 |
| | | `[Rn], +/-Rm, ASR #<shift>` | Allowed shifts 1-32 |
| | | `[Rn], +/-Rm, ROR #<shift>` | Allowed shifts 1-31 |
| | | `[Rn], +/-Rm, RRX` | |

**Addressing Mode 2 (Post-indexed only)**

| | | | |
|---|---|---|---|
| Post-indexed | Immediate offset | `[Rn], #+/-<immed_12>` | |
| | Zero offset | `[Rn]` | Equivalent to [Rn],#0 |
| | Register offset | `[Rn], +/-Rm` | |
| | Scaled register offset | `[Rn], +/-Rm, LSL #<shift>` | Allowed shifts 0-31 |
| | | `[Rn], +/-Rm, LSR #<shift>` | Allowed shifts 1-32 |
| | | `[Rn], +/-Rm, ASR #<shift>` | Allowed shifts 1-32 |
| | | `[Rn], +/-Rm, ROR #<shift>` | Allowed shifts 1-31 |
| | | `[Rn], +/-Rm, RRX` | |

**Addressing Mode 3 - Halfword, Signed Byte, and Doubleword Data Transfer**

| | | | |
|---|---|---|---|
| Pre-indexed | Immediate offset | `[Rn, #+/-<immed_8>]{!}` | |
| | Zero offset | `[Rn]` | Equivalent to [Rn,#0] |
| | Register | `[Rn, +/-Rm]{!}` | |
| Post-indexed | Immediate offset | `[Rn], #+/-<immed_8>` | |
| | Register | `[Rn], +/-Rm` | |

**Addressing Mode 4 - Multiple Data Transfer**

| Block load | | Stack pop | |
|---|---|---|---|
| IA | Increment After | FD | Full Descending |
| IB | Increment Before | ED | Empty Descending |
| DA | Decrement After | FA | Full Ascending |
| DB | Decrement Before | EA | Empty Ascending |
| **Block store** | | **Stack push** | |
| IA | Increment After | EA | Empty Ascending |
| IB | Increment Before | FA | Full Ascending |
| DA | Decrement After | ED | Empty Descending |
| DB | Decrement Before | FD | Full Descending |

**Addressing Mode 5 - Coprocessor Data Transfer**

| | | | |
|---|---|---|---|
| Pre-indexed | Immediate offset | `[Rn, #+/-<immed_8*4>]{!}` | |
| | Zero offset | `[Rn]` | Equivalent to [Rn,#0] |
| Post-indexed | Immediate offset | `[Rn], #+/-<immed_8*4>` | |
| Unindexed | No offset | `[Rn], {8-bit copro. option}` | |

**ARM architecture versions**

| | |
|---|---|
| *n* | ARM architecture version *n* and above. |
| *n*T, *n*J | T or J variants of ARM architecture version *n* and above. |
| M | ARM architecture version 3M, and 4 and above, except xM variants. |
| *n*E | All E variants of ARM architecture version *n* and above. |
| *n*E* | E variants of ARM architecture version *n* and above, except xP variants. |
| XS | XScale coprocessor instruction |

**Flexible Operand 2**

| | | |
|---|---|---|
| Immediate value | `#<immed_8r>` | |
| Logical shift left immediate | `Rm, LSL #<shift>` | Allowed shifts 0-31 |
| Logical shift right immediate | `Rm, LSR #<shift>` | Allowed shifts 1-32 |
| Arithmetic shift right immediate | `Rm, ASR #<shift>` | Allowed shifts 1-32 |
| Rotate right immediate | `Rm, ROR #<shift>` | Allowed shifts 1-31 |
| Register | `Rm` | |
| Rotate right extended | `Rm, RRX` | |
| Logical shift left register | `Rm, LSL Rs` | |
| Logical shift right register | `Rm, LSR Rs` | |
| Arithmetic shift right register | `Rm, ASR Rs` | |
| Rotate right register | `Rm, ROR Rs` | |

**PSR fields** (use at least one suffix)

| Suffix | Meaning | |
|---|---|---|
| c | Control field mask byte | PSR[7:0] |
| f | Flags field mask byte | PSR[31:24] |
| s | Status field mask byte | PSR[23:16] |
| x | Extension field mask byte | PSR[15:8] |

**Condition Field**

| Mnemonic | Description | Description (VFP) |
|---|---|---|
| EQ | Equal | Equal |
| NE | Not equal | Not equal, or unordered |
| CS / HS | Carry Set / Unsigned higher or same | Greater than or equal, or unordered |
| CC / LO | Carry Clear / Unsigned lower | Less than |
| MI | Negative | Less than |
| PL | Positive or zero | Greater than or equal, or unordered |
| VS | Overflow | Unordered (at least one NaN operand) |
| VC | No overflow | Not unordered |
| HI | Unsigned higher | Greater than, or unordered |
| LS | Unsigned lower or same | Less than or equal |
| GE | Signed greater than or equal | Greater than or equal |
| LT | Signed less than | Less than, or unordered |
| GT | Signed greater than | Greater than |
| LE | Signed less than or equal | Less than or equal, or unordered |
| AL | Always (normally omitted) | Always (normally omitted) |

**Processor Modes**

| | |
|---|---|
| 16 | User |
| 17 | FIQ Fast Interrupt |
| 18 | IRQ Interrupt |
| 19 | Supervisor |
| 23 | Abort |
| 27 | Undefined |
| 31 | System |

**Prefixes for Parallel Instructions**

| | |
|---|---|
| S | Signed arithmetic modulo $2^8$ or $2^{16}$, sets CPSR GE bits |
| Q | Signed saturating arithmetic |
| SH | Signed arithmetic, halving results |
| U | Unsigned arithmetic modulo $2^8$ or $2^{16}$, sets CPSR GE bits |
| UQ | Unsigned saturating arithmetic |
| UH | Unsigned arithmetic, halving results |

# ARM Addressing Modes Quick Reference Card

| Coprocessor operations | § | Assembler | Action | Notes |
|---|---|---|---|---|
| Data operations | 2 | `CDP{cond} <copr>, <op1>, CRd, CRn, CRm{, <op2>}` | Coprocessor dependent | |
| Alternative data operations | 5 | `CDP2 <copr>, <op1>, CRd, CRn, CRm{, <op2>}` | Coprocessor dependent | Cannot be conditional. |
| Move to ARM register from coprocessor | 2 | `MRC{cond} <copr>, <op1>, Rd, CRn, CRm{, <op2>}` | Coprocessor dependent | |
| Alternative move | 5 | `MRC2 <copr>, <op1>, Rd, CRn, CRm{, <op2>}` | Coprocessor dependent | Cannot be conditional. |
| Two ARM register move | 5E* | `MRRC{cond} <copr>, <op1>, Rd, Rn, CRm` | Coprocessor dependent | |
| Alternative two ARM register move | 6 | `MRRC2 <copr>, <op1>, Rd, Rn, CRm` | Coprocessor dependent | Cannot be conditional. |
| Move to coproc from ARM reg | 2 | `MCR{cond} <copr>, <op1>, Rd, CRn, CRm{, <op2>}` | Coprocessor dependent | |
| Alternative move | 5 | `MCR2 <copr>, <op1>, Rd, CRn, CRm{, <op2>}` | Coprocessor dependent | Cannot be conditional. |
| Two ARM register move | 5E* | `MCRR{cond} <copr>, <op1>, Rd, Rn, CRm` | Coprocessor dependent | |
| Alternative two ARM register move | 6 | `MCRR2 <copr>, <op1>, Rd, Rn, CRm` | Coprocessor dependent | Cannot be conditional. |
| Load | 2 | `LDC{cond} <copr>, CRd, <a_mode5>` | Coprocessor dependent | |
| Alternative loads | 5 | `LDC2 <copr>, CRd, <a_mode5>` | Coprocessor dependent | Cannot be conditional. |
| Store | 2 | `STC{cond} <copr>, CRd, <a_mode5>` | Coprocessor dependent | |
| Alternative stores | 5 | `STC2 <copr>, CRd, <a_mode5>` | Coprocessor dependent | Cannot be conditional. |

## Proprietary Notice

## Document Number

ARM QRC 0001H

## Change Log

| Issue | Date | By | Change |
|---|---|---|---|
| A | June 1995 | BJH | First Release |
| B | Sept 1996 | BJH | Second Release |
| C | Nov 1998 | BJH | Third Release |
| D | Oct 1999 | CKS | Fourth Release |
| E | Oct 2000 | CKS | Fifth Release |
| F | Sept 2001 | CKS | Sixth Release |
| G | Jan 2003 | CKS | Seventh Release |
| H | Oct 2003 | CKS | Eighth Release |