ARM* Instruction Set & Assembly Language

Web Sites:

http://www.heyrick.co.uk/assembler/

http://dec.bournemouth.ac.uk/staff/pknaggs/sysarch/ARMBook.pdf

http://www.arm.com/community/academy/university.html

Books (in addition to textbook):

Computers as Components by Wayne Wolf, Morgan Kaufman, 2000.

The ARM RISC Chip – A Programmer's Guide by A. van Someren & C. Atack, Addison-Wesley, 1994.

Jens Gregor, UTK CS Professor.

*Advanced RISC Machines

ARM Instruction Set Overview & Registers

Main Features [1]

- All instructions are 32 bits long
- Registers are 32 bits long
- Memory addresses are 32 bits long
- Memory is byte addressable
- Most instructions execute in a single cycle
- Every instruction can be conditionally executed
- Can be configured at power-up as either little or big endian

Main Features [2]

- A load/store architecture
  - Data processing instructions act only on registers
    - Three operand format
    - Combined ALU and shifter for high speed bit manipulation
  - Specific memory access instructions with powerful auto-indexing addressing modes
    - 32 bit and 8 bit data types
      - And also 16 bit data types on ARM Architecture v4
    - Flexible multiple register load and store instructions

Processor Modes

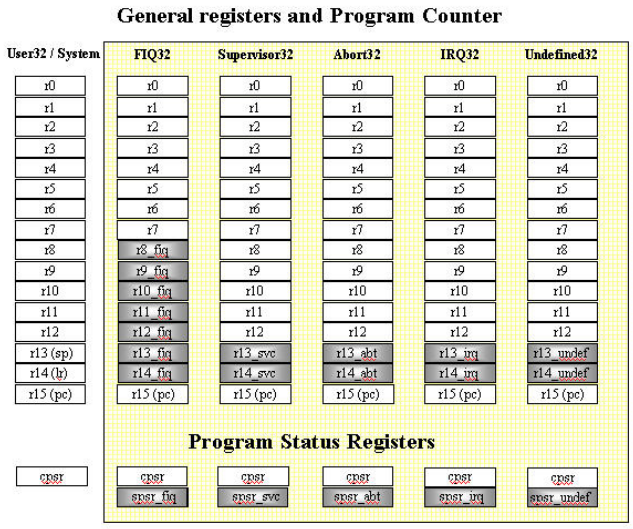
- The ARM has six execution modes
  - *User* (unprivileged mode under which most tasks run)
  - *FIQ* (entered when a high priority (fast) interrupt is raised)
  - *IRQ* (entered when a low priority (normal) interrupt is raised)
  - *Supervisor* (entered on reset and when a Software Interrupt instruction is executed)
  - *Abort* (used to handle memory access violations)
  - *Undef* (used to handle undefined instructions)
- ARM Architecture Version 4 adds a seventh mode
  - *System* (privileged mode using the same registers as user mode)

Registers

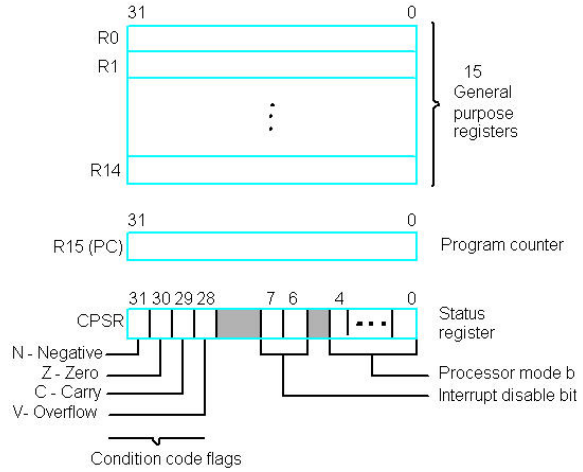
- ARM has 37 registers in total, all of which are 32-bits long
  - 1 dedicated program counter (PC)
  - 1 dedicated current program status register (cpsr)
  - 5 dedicated saved program status registers (spsr)
  - 30 general purpose registers
- However, these are arranged into several banks, with the accessible bank being governed by the processor mode. Each mode can access
  - a particular set of r0-r12 registers
  - a particular r13 (the *stack pointer*) and r14 (*link register*)
  - r15 (the *program counter*)
  - cpsr (the *current program status register* or *status register*)

  and privileged modes can also access
  - a particular spsr (*saved program status register*)

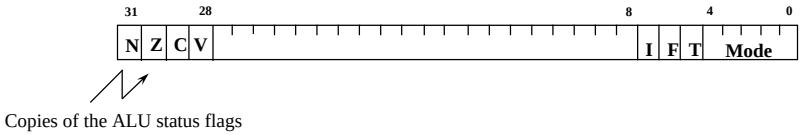
Register Organization



User Mode Registers



Program Status Registers (CPSR, SPSRs)



Condition Code Flags

- N = Negative result from ALU flag
- Z = Zero result from ALU flag
- C = ALU operation Carried out
- V = ALU operation oVerflowed

Mode Bits

- M[4:0] define the processor mode

Interrupt Disable bits

- I = 1, disables the IRQ
- F = 1, disables the FIQ

T Bit (Architecture v4T only)

- T = 0, processor in ARM state
- T = 1, Processor in thumb state

Accessing Registers

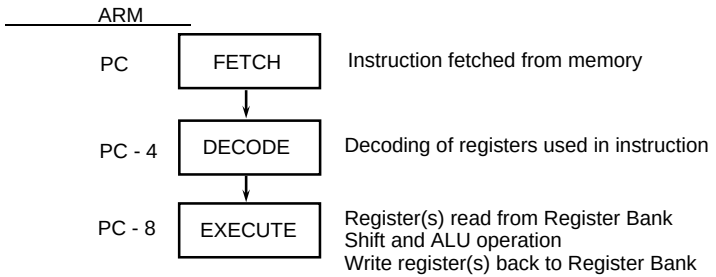
- All instructions can access r0-r14 directly
- Most instructions also allow access of the PC
- Specific instructions allow access to cpsr

Program Counter (r15)

- When the processor is executing in ARM state
  - All instructions are 32 bits in length
  - All instructions must be word aligned
  - Addresses refers to byte (i.e., byte addressable)
  - Therefore, the PC value is stored in bits [31:2] with bits [1:0] equal to zero (as instructions cannot be halfword or byte aligned)
- r14 used as the subroutine link register (lr) and stores the return address when Branch with Link (BL) operations are performed through registers (place on stack in linked branch)
- Thus, to return from a linked branch using registers, contents of r14 must be placed in r15 (from stack).

Instruction Pipeline

- ARM uses a 3-stage pipeline in order to increase the speed of the flow of instructions to the processor

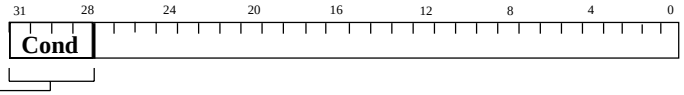


- The PC points to the instruction being fetched

Conditional Execution

- Most instruction sets only allow branches to be executed conditionally.
- However by reusing the condition evaluation hardware, ARM effectively increases number of instructions.
  - All instructions contain a condition field which determines whether the CPU will execute them.
  - Non-executed instructions soak up 1 cycle.
    - Still have to complete cycle so as to allow fetching and decoding of following instructions.
- This removes the need for many branches, which stall the pipeline (3 cycles to refill).
  - Allows very dense in-line code, without branches.
  - The Time penalty of not executing several conditional instructions is frequently less than overhead of the branch or subroutine call that would otherwise be needed.

The Condition Field



0000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI - N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

1101 = LE - Z set, or N set and V clear,or N clear and V set (<, or =)

1110 = AL - always

1111 = NV - reserved.

Using and updating the Condition Field

- To execute an instruction conditionally, simply postfix it with the appropriate condition:
  - For example an add instruction takes the form:
    - `ADD r0,r1,r2 ; r0 = r1 + r2 (ADDAL)`
  - To execute this only if the zero flag is set:
    - `ADDEQ r0,r1,r2 ; If zero flag set then…`
      `; ... r0 = r1 + r2`
- By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect). To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an “S”.
  - For example to add two numbers and set the condition flags:
    - `ADDS r0,r1,r2 ; r0 = r1 + r2`
      `; ... and set flags`

Instruction Formats and Addressing Modes

Basic Instruction Format

- Most instructions use the format

Conditional Execution Code – bits 28-31

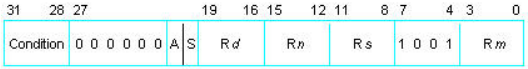
Opcode – bits 20-27

2 or 3 Registers – bits 16-19, 12-15 & 0-3

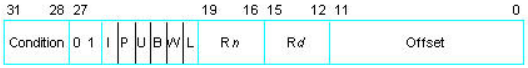
Other information – bits 4-11 & maybe 0-3

ARM Instruction Encoding Formats [1]

(a) Arithmetic, logic, compare, test, and move



(b) Multiply and Multiply Accumulate



(c) Single word or byte transfer from/to memory

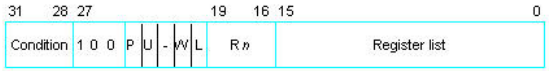
I Immediate
S Set
A Accumulate

P Pre/Post-index
U Up/Down
B Byte/Word

W Writeback
L Load/Store
K Link

From Appendix B

ARM Instruction Encoding Formats [2]



(d) Multiple word transfer from/to memory

(e) Branch and Branch with Link

I Immediate
S Set
A Accumulate

P Pre/Post-index
U Up/Down
B Byte/Word

W Writeback
L Load/Store
K Link

From Appendix B

Addressing Modes [1]

Name	Assembler syntax	Addressing function
With immediate offset:		
Pre-indexed	[Rn, #offset]	EA = [Rn] + offset
Pre-indexed with writeback	[Rn, #offset]!	EA = [Rn] + offset; Rn ← [Rn] + offset
Post-indexed	[Rn], #offset	EA = [Rn]; Rn ← [Rn] + offset

EA = effective address

offset = a signed number contained in the instruction Between ±4095

Examples:

Instruction		Operation
LDR	R0,[R1,#12]	R0 ← [[R1]+12]
STR	R0,[R1,#12]!	& Loc([R1]+12]) ← R0 R1 ← [R1]+12

Addressing Modes [2]

Name	Assembler syntax	Addressing function
With offset magnitude in Rm:		
Pre-indexed	[Rn,± Rm, shift]	EA = [Rn] ± [Rm] shifted
Pre-indexed with writeback	[Rn,± Rm, shift]!	EA = [Rn] ± [Rm] shifted; Rn ← [Rn] ± [Rm] shifted
Post-indexed	[Rn], ± Rm, shift	EA = [Rn]; Rn ← [Rn] ± [Rm] shifted
Relative (Pre-indexed with immediate offset)	Location	EA = Location = [PC] + offset

shift = direction #integer
where direction is LSL for left shift or LSR for right shift, and integer is a 5-bit unsigned number specifying the shift amount

± Rm = the offset magnitude in register Rm can be added to or subtracted from the contents of base register Rn

LDR R0,[R1,R2,LSL#2] R0 ← [[R1]+4*[R2]]

STR R0,[R1],-R2,LSR#4 & Loc([R1]) ← R0
R1 ← [R1] – [R2]/16 (truncated)

More on Addressing Modes

- Do not have to specify an offset or shift

Examples:

LDR R1,[R5] R1← [[R5]]
STR R3,[R0,-R6] Loc([R0]-[R6]) ← R3

- No Direct Addressing mode but assembler turns it into Relative addressing mode:

LDR R0,Address offset ← [Address] - [PC] - 8
R0 ← Loc([PC] + offset)

More on Load & Store

- Can Load & Store bytes rather than words

Use LDRB rather than LDR and STRB rather than STR .

  - Loads & stores from 8 bits in low-order byte position
- Can Load to and Store from multiple registers
  - Can only load and store multiple words (32 bits)
  - Pre and post indexing with or without writeback modes are all available
  - Mnemonic is LDM and STM and may have suffixes such as IA and FD (see next slide)

Example: STMFD R5!,{R0,R1,R2,R3}

Store R3 in [R5-4], R2 in [R5-8], R1 in [R5-12], R0 in [R5-16]

Multiple Word Transfers

Mnemonic (Name)	Instruction bits P U L	Operation performed
LDMIA/LDMFD (Increment after/ Full descending)	0 1 1	$R_{low}, \ldots, R_{high} \leftarrow [[Rn]], [[Rn]+4], \ldots$
LDMIB/LDMED (Increment before/ Empty descending)	1 1 1	$R_{low}, \ldots R_{high} \leftarrow [[Rn]+4], [[Rn]+8], \ldots$
LDMDA/LDMFA (Decrement after/ Full ascending)	0 0 1	$R_{high}, \ldots, R_{low} \leftarrow [[Rn]], [[Rn]-4], \ldots$
LDMDB/LDMEA (Decrement before/ Empty ascending)	1 0 1	$R_{high}, \ldots, R_{low} \leftarrow [[Rn]-4], [[Rn]-8], \ldots$
STMIA/STMEA (Increment after/ Empty ascending)	0 1 0	$[Rn], [Rn]+4, \ldots \leftarrow [R_{low}], \ldots, [R_{high}]$
STMIB/STMFA (Increment before/ Full ascending)	1 1 0	$[Rn]+4, [Rn]+8, \ldots \leftarrow [R_{low}], \ldots, [R_{high}]$
STMDA/STMED (Decrement after/ Empty descending)	0 0 0	$[Rn], [Rn]-4, \ldots \leftarrow [R_{high}], \ldots, [R_{low}]$
STMDB/STMFD (Decrement before/ Full descending)	1 0 0	$[Rn]-4, [Rn]-8, \ldots \leftarrow [R_{high}], \ldots, [R_{low}]$

ARM Assembly Instructions

ARM Assembly Language

- Fairly standard assembly language:

```
        LDR r0,[r8] ; a comment
label   ADD r4,r0,r1
```


ALU Instructions [1]

- Basic format:

  `ADD r0,r1,r2`

  – r0 ← [r1] + [r2]

  – Computes r1+r2, stores in r0.

- Immediate operand:

  `ADD r0,r1,#2`

  – r0 ← [r1] + 2

  – Computes r1+2, stores in r0.

ALU Instructions [2]

Mnemonic (Name)	OP code $b_{24}..b_{21}$	Operation performed	CC flags affected if S = 1			
			N	Z	V	C
ADD (Add)	0 1 0 0	Rd ← [Rn] + Oper2	x	x	x	x
ADC (Add with carry)	0 1 0 1	Rd ← [Rn] + Oper2 + [C]	x	x	x	x
SUB (Subtract)	0 0 1 0	Rd ← [Rn] – Oper2	x	x	x	x
SBC (Subtract with carry)	0 1 1 0	Rd ← [Rn] – Oper2 + [C] – 1	x	x	x	x
RSB (Reverse subtract)	0 0 1 1	Rd ← Oper2 – [Rn]	x	x	x	x
RSC (Reversesubtract with carry)	0 1 1 1	Rd ← Oper2 – [Rn] + [C] – 1	x	x	x	x
MUL (Multiply)	(See Figure B.4)	Rd ← [Rm] × [Rs]	x	x		
MLA (Multiply accumulate)	(See Figure B.4)	Rd ← [Rm] × [Rs] + [Rn]	x	x		

ALU Instructions [3]

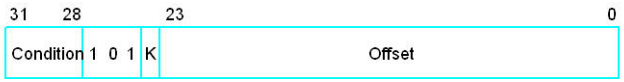
Mnemonic (Name)	OP code $b_{24}..b_{21}$	Operation performed	CC flags affected if S = 1			
			N	Z	V	C
AND (Logical AND)	0 0 0 0	Rd ← [Rn] ∧ Oper2	x	x		x
ORR (Logical OR)	1 1 0 0	Rd ← [Rn] – Oper2	x	x		x
EOR (Exclusive-OR)	0 0 0 1	Rd ← [Rn] ∨ Oper2	x	x		x
BIC (Bit clear)	1 1 1 0	Rd ← [Rn] ∧ ¬ Oper2	x	x		x

Comparison/Test Instructions

Mnemonic (Name)	OP code $b_{24}..b_{21}$	Operation performed	CC flags affected if S = 1			
			N	Z	V	C
CMP (Compare)	1 0 1 0	[Rn] – Oper2	x	x	x	x
CMN (CompareNegative)	1 0 1 1	[Rn] + Oper2	x	x	x	x
TST (Bit test)	1 0 0 0	[Rn] ∧ Oper2	x	x		x
TEQ (Test equal)	1 0 0 1	[Rn] ⊕ Oper2	x	x		x

- These instructions set only the NZCV bits of CPSR.

Branching Instructions



K=0 :Branch (B)
K=1 :Branch with Link (BL); store return address in register R14

Examples:

```
BEQ   LOC1
BNE   LOC2
BL    ROUTINE
```


ARM Move Instructions

Mnemonic (Name)	OP code $b_{24}..b_{21}$	Operation performed	CC flags affected if S = 1			
			N	Z	V	C
MOV (Move)	1 1 0 1	Rd ← Oper2	x	x		x
MVN (Move Complement)	1 1 1 1	Rd ← ¬ Oper2	x	x		x

```
MOV r0,r1 ; sets r0 to r1
```

Load/Store Instructions

- LDR, LDRB : load (word, byte)
- STR, STRB : store (word, byte)
- Addressing modes:
  - register indirect : `LDR r0,[r1]`
  - with second register : `LDR r0,[r1,-r2]`
  - with constant : `LDR r0,[r1,#4]`

ADR pseudo-op

`ADR r1,FOO`

- Loads the 32-bit address FOO into r1
- Not an actual machine instruction
- Assembler replaces with real machine instructions to produce desired results

ARM subroutine linkage

- Branch and link instruction:

  `BL ROUTINE`

  - Copies current PC to r14.
- Initial instructions in ROUTINE
  - Save registers used in subroutine and r14 on stack (allows nested calls); for example:

    ROUTINE STMFD R13!,{r0,r1,r2,r14}
- Final instructions in ROUTINE:
  - Restore saved registers from stack and return r14 address from stack to r15; for example:

    LBMFD R13!,{r0,r1,r2,r15}
- Nested and recursive calls handled properly with this process