ARM CPU Overview

The ARM7TDMI is a 32bit RISC (Reduced Instruction Set Computer) CPU, designed by ARM (Advanced RISC Machines), and designed for both high performance and low power consumption.

Fast Execution
Depending on the CPU state, all opcodes are sized 32bit or 16bit (that's counting both the opcode bits and its parameters bits) providing fast decoding and execution. Additionally, pipelining allows - (a) one instruction to be executed while (b) the next instruction is decoded and (c) the next instruction is fetched from memory - all at the same time.

Data Formats
The CPU manages to deal with 8bit, 16bit, and 32bit data, that are called:

```
   8bit - Byte
  16bit - Halfword
  32bit - Word
```

The two CPU states
As mentioned above, two CPU states exist:
- ARM state: Uses the full 32bit instruction set (32bit opcodes)
- THUMB state: Uses a cutdown 16bit instruction set (16bit opcodes)
Regardless of the opcode-width, both states are using 32bit registers, allowing 32bit memory addressing as well as 32bit arithmetic/logical operations.

When to use ARM state
Basically, there are two advantages in ARM state:

```
- Each single opcode provides more functionality, resulting
  in faster execution when using a 32bit bus memory system
  (such like opcodes stored in GBA Work RAM).
- All registers R0-R15 can be accessed directly.
```

The downsides are:

```
- Not so fast when using 16bit memory system
  (but it still works though).
- Program code occupies more memory space.
```

When to use THUMB state
There are two major advantages in THUMB state:

```
- Faster execution up to approx 160% when using a 16bit bus
  memory system (such like opcodes stored in GBA GamePak ROM).
- Reduces code size, decreases memory overload down to approx 65%.
```

The disadvantages are:

```
- Not as multi-functional opcodes as in ARM state, so it will
  be sometimes required use more than one opcode to gain a
  similar result as for a single opcode in ARM state.
- Most opcodes allow only registers R0-R7 to be used directly.
```

Combining ARM and THUMB state
Switching between ARM and THUMB state is done by a normal branch (BX) instruction which takes only a handful of cycles to execute (allowing to change states as often as desired - with almost no overload).

Also, as both ARM and THUMB are using the same register set, it is possible to pass data between ARM and THUMB mode very easily.

The best memory & execution performance can be gained by combining both states: THUMB for normal program code, and ARM code for timing critical subroutines (such like interrupt handlers, or complicated algorithms).

Note: ARM and THUMB code cannot be executed simultaneously.

Automatic state changes
Beside for the above manual state switching by using BX instructions, the following situations involve automatic state changes:
- CPU switches to ARM state when executing an exception
- User switches back to old state when leaving an exception

ARM CPU Register Set

Overview
The following table shows the ARM7TDMI register set which is available in each mode. There's a total of 37 registers (32bit each), 31 general registers (Rxx) and 6 status registers (xPSR).
Note that only some registers are 'banked', for example, each mode has it's own R14 register: called R14, R14_fiq, R14_svc, etc. for each mode respectively.
However, other registers are not banked, for example, each mode is using the same R0 register, so writing to R0 will always affect the content of R0 in other modes also.

System/User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13 (SP)	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14 (LR)	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15	R15	R15	R15	R15
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
--	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

R0-R12 Registers (General Purpose Registers)
These thirteen registers may be used for whatever general purposes. Basically, each is having same functionality and performance, ie. there is no 'fast accumulator' for arithmetic operations, and no 'special pointer register' for memory addressing.
However, in THUMB mode only R0-R7 (Lo registers) may be accessed freely, while R8-R12 and up (Hi registers) can be accessed only by some instructions.

R13 Register (SP)
This register is used as Stack Pointer (SP) in THUMB state. While in ARM state the user may decided to use R13 and/or other register(s) as stack pointer(s), or as general purpose register.
As shown in the table above, there's a separate R13 register in each mode, and (when used as SP) each exception handler may (and MUST!) use its own stack.

R14 Register (LR)
This register is used as Link Register (LR). That is, when calling to a sub-routine by a Branch with Link (BL) instruction, then the return address (ie. old value of PC) is saved in this register.
Storing the return address in the LR register is obviously faster than pushing it into memory, however, as there's only one LR register for each mode, the user must manually push its content before issuing 'nested' subroutines.
Same happens when an exception is called, PC is saved in LR of new mode.
Note: In ARM mode, R14 may be used as general purpose register also, provided that above usage as LR register isn't required.

R15 Register (PC)
R15 is always used as program counter (PC). Note that when reading R15, this will usually return a value of PC+nn because of read-ahead (pipelining), whereas 'nn' depends on the instruction and on the CPU state (ARM or THUMB).

CPSR and SPSR (Program Status Registers) (ARMv3 and up)
The current condition codes (flags) and CPU control bits are stored in the CPSR register. When an exception arises, the old CPSR is saved in the SPSR of the respective exception-mode (much like PC is saved in LR).
For details refer to chapter about CPU Flags.

ARM CPU Flags & Condition Field (cond)

ARM Condition Field {cond}
The opcode {cond} suffixes can be used for conditionally executed code based on the C,N,Z,V flags in CPSR register. For example: BEQ = Branch if Equal, MOVMI = Move if Signed.
In ARM mode, {cond} can be used with all opcodes (except for a few newer ARMv5 instructions: BKPT, PLD, CDP2, LDC2, MCR2, MRC2, STC2, and BLX_imm are nonconditional; however BLX_reg can be conditional).
In THUMB mode, {cond} can be used only for branch opcodes.

Code	Suffix	Flags	Meaning
0:	EQ	Z=1	equal (zero) (same)
1:	NE	Z=0	not equal (nonzero) (not same)
2:	CS/HS	C=1	unsigned higher or same (carry set)
3:	CC/LO	C=0	unsigned lower (carry cleared)
4:	MI	N=1	negative (minus)
5:	PL	N=0	positive or zero (plus)
6:	VS	V=1	overflow (V set)
7:	VC	V=0	no overflow (V cleared)
8:	HI	C=1 and Z=0	unsigned higher
9:	LS	C=0 or Z=1	unsigned lower or same
A:	GE	N=V	greater or equal
B:	LT	N<>V	less than
C:	GT	Z=0 and N=V	greater than
D:	LE	Z=1 or N<>V	less or equal
E:	AL	-	always (the "AL" suffix can be omitted)
F:	NV	-	never (ARMv1,v2 only) (Reserved ARMv3 and up)

Execution Time: If condition=false: 1S cycle. Otherwise: as specified for the respective opcode.

Current Program Status Register (CPSR)

```
Bit   Expl.
31    N - Sign Flag       (0=Not Signed, 1=Signed)               ;\
30    Z - Zero Flag       (0=Not Zero, 1=Zero)                   ; Condition
29    C - Carry Flag      (0=Borrow/No Carry, 1=Carry/No Borrow) ; Code Flags
28    V - Overflow Flag   (0=No Overflow, 1=Overflow)            ;/
27    Q - Sticky Overflow (1=Sticky Overflow, ARMv5TE and up only)
26-8  Reserved            (For future use) - Do not change manually!
7     I - IRQ disable     (0=Enable, 1=Disable)                    ;\
6     F - FIQ disable     (0=Enable, 1=Disable)                    ; Control
```

```
  5     T - State Bit       (0=ARM, 1=THUMB) - Do not change manually!; Bits
  4-0   M4-M0 - Mode Bits   (See below)                                  ;/
```

Bit 31-28: Condition Code Flags (N,Z,C,V)
These bits reflect results of logical or arithmetic instructions. In ARM mode, it is often optionally whether an instruction should modify flags or not, for example, it is possible to execute a SUB instruction that does NOT modify the condition flags.
In ARM state, all instructions can be executed conditionally depending on the settings of the flags, such like MOVEQ (Move if Z=1). While In THUMB state, only Branch instructions (jumps) can be made conditionally.

Bit 27: Sticky Overflow Flag (Q) - ARMv5TE and ARMv5TExP and up only
Used by QADD, QSUB, QDADD, QDSUB, SMLAxy, and SMLAWy only. These opcodes set the Q-flag in case of overflows, but leave it unchanged otherwise. The Q-flag can be tested/reset by MSR/MRS opcodes only.

Bit 27-8: Reserved Bits (except Bit 27 on ARMv5TE and up, see above)
These bits are reserved for possible future implementations. For best forwards compatibility, the user should never change the state of these bits, and should not expect these bits to be set to a specific value.

Bit 7-0: Control Bits (I,F,T,M4-M0)
These bits may change when an exception occurs. In privileged modes (non-user modes) they may be also changed manually.
The interrupt bits I and F are used to disable IRQ and FIQ interrupts respectively (a setting of "1" means disabled).
The T Bit signalizes the current state of the CPU (0=ARM, 1=THUMB), this bit should never be changed manually - instead, changing between ARM and THUMB state must be done by BX instructions.
The Mode Bits M4-M0 contain the current operating mode.

```
  Binary Hex Dec  Expl.
  0xx00b 00h 0  - Old User       ;\26bit Backward Compatibility modes
  0xx01b 01h 1  - Old FIQ        ; (supported only on ARMv3, except ARMv3G,
  0xx10b 02h 2  - Old IRQ        ; and on some non-T variants of ARMv4)
  0xx11b 03h 3  - Old Supervisor ;/
  10000b 10h 16 - User (non-privileged)
  10001b 11h 17 - FIQ
  10010b 12h 18 - IRQ
  10011b 13h 19 - Supervisor (SWI)
  10111b 17h 23 - Abort
  11011b 1Bh 27 - Undefined
  11111b 1Fh 31 - System (privileged 'User' mode) (ARMv4 and up)
```

Writing any other values into the Mode bits is not allowed.

Saved Program Status Registers (SPSR_<mode>)
Additionally to above CPSR, five Saved Program Status Registers exist:
SPSR_fiq, SPSR_svc, SPSR_abt, SPSR_irq, SPSR_und
Whenever the CPU enters an exception, the current status register (CPSR) is copied to the respective SPSR_<mode> register. Note that there is only one SPSR for each mode, so nested exceptions inside of the same mode are allowed only if the exception handler saves the content of SPSR in memory.
For example, for an IRQ exception: IRQ-mode is entered, and CPSR is copied to SPSR_irq. If the interrupt handler wants to enable nested IRQs, then it must first push SPSR_irq before doing so.

ARM CPU 26bit Memory Interface

The 26bit Memory Interface was used by ARMv1 and ARMv2. The 32bit interface is used by ARMv3 and newer, however, 26bit backward compatibility was included in all ARMv3 (except ARMv3G), and optionally in some non-T variants of ARMv4.

Format of R15 in 26bit Mode (Program Counter Register)

```
Bit   Name     Expl.
31-28 N,Z,C,V  Flags (Sign, Zero, Carry, Overflow)
27-26 I,F      Interrupt Disable bits (IRQ, FIQ) (1=Disable)
25-2  PC       Program Counter, 24bit, Step 4 (64M range)
1-0   M1,M0    Mode (0=User, 1=FIQ, 2=IRQ, 3=Supervisor)
```

Branches with +/-32M range wrap the PC register, and can reach all 64M memory.

Reading from R15

If R15 is specified in bit16-19 of an opcode, then NZCVIF and M0,1 are masked (zero), otherwise the full 32bits are used.

Writing to R15

ALU opcodes with S=1, and LDM opcodes with PSR=1 can write to all 32bits in R15 (in 26bit mode, that is allowed even in user mode, though it does then affect only NZCF, not the write protected IFMM bits ???), other opcodes which write to R15 will modify only the program counter bits. Also, special CMP/CMN/TST/TEQ{P} opcodes can be used to write to the PSR bits in R15 without modifying the PC bits.

Exceptions

SWIs, Reset, Data/Prefetch Aborts and Undefined instructions enter Supervisor mode. Interrupts enter IRQ and FIQ mode. Additionally, a special 26bit Address Exception exists, which enters Supervisor mode on accesses to memory addresses>=64M as follows:

```
R14_svc = PC ($+8, including old PSR bits)
M1,M0 = 11b = supervisor mode, F=same, I=1, PC=14h,
to continue at the fault location, return by SUBS PC,LR,8.
```

32bit CPUs with 26bit compatibility mode can be configured to switch into 32bit mode when encountering exceptions.

ARM CPU Exceptions

Exception Vectors

The following are the exception vectors in memory. That is, when an exception arises, CPU is switched into ARM state, and the program counter (PC) is loaded by the respective address.

```
Address  Prio  Exception                  Mode on Entry      Interrupt Flags
BASE+00h 1     Reset                      Supervisor (_svc)  I=1, F=1
BASE+04h 7     Undefined Instruction      Undefined  (_und)  I=1, F=unchanged
BASE+08h 6     Software Interrupt (SWI)   Supervisor (_svc)  I=1, F=unchanged
BASE+0Ch 5     Prefetch Abort             Abort      (_abt)  I=1, F=unchanged
BASE+10h 2     Data Abort                 Abort      (_abt)  I=1, F=unchanged
BASE+14h ??    Address Exceeds 26bit      Supervisor (_svc)  I=1, F=unchanged
BASE+18h 4     Normal Interrupt (IRQ)     IRQ        (_irq)  I=1, F=unchanged
BASE+1Ch 3     Fast Interrupt (FIQ)       FIQ        (_fiq)  I=1, F=1
```

BASE is normally 00000000h, but may be optionally FFFF0000h in some ARM CPUs. Priority for simultaneously occuring exceptions ranges from Prio=1=Highest to Prio=7=Lowest.
As there's only space for one ARM opcode at each of the above addresses, it'd be usually recommended to deposit a Branch opcode into each vector, which'd then redirect to the actual exception handlers address.

Actions performed by CPU when entering an exception

```
- R14_<new mode>=PC+nn   ;save old PC, ie. return address
- SPSR_<new mode>=CPSR   ;save old flags
- CPSR new T,M bits      ;set to T=0 (ARM state), and M4-0=new mode
- CPSR new I bit         ;IRQs disabled (I=1), done by ALL exceptions
- CPSR new F bit         ;FIQs disabled (F=1), done by Reset and FIQ only
- PC=exception_vector    ;see table above
```

Above "PC+nn" depends on the type of exception. Basically, in ARM state that nn-offset is caused by pipelining, and in THUMB state an identical ARM-style 'offset' is generated (even though the 'base address' may be only halfword-aligned).

Required user-handler actions when returning from an exception
Restore any general registers (R0-R14) which might have been modified by the exception handler. Use return-instruction as listed in the respective descriptions below, this will both restore PC and CPSR - that automatically involves that the old CPU state (THUMB or ARM) as well as old state of FIQ and IRQ disable flags are restored. As mentioned above (see action on entering...), the return address is always saved in ARM-style format, so that exception handler may use the same return-instruction, regardless of whether the exception has been generated from inside of ARM or THUMB state.

FIQ (Fast Interrupt Request)
This interrupt is generated by a LOW level on the nFIQ input. It is supposed to process timing critical interrupts at a high priority, as fast as possible.
Additionally to the common banked registers (R13_fiq,R14_fiq), five extra banked registers (R8_fiq-R12_fiq) are available in FIQ mode. The exception handler may freely access these registers without modifying the main programs R8-R12 registers (and without having to save that registers on stack).
In privileged (non-user) modes, FIQs may be also manually disabled by setting the F Bit in CPSR.

IRQ (Normal Interrupt Request)
This interrupt is generated by a LOW level on the nIRQ input. Unlike FIQ, the IRQ mode is not having its own banked R8-R12 registers.
IRQ is having lower priority than FIQ, and IRQs are automatically disabled when a FIQ exception becomes executed. In privileged (non-user) modes, IRQs may be also manually disabled by setting the I Bit in CPSR.
To return from IRQ Mode (continuing at following opcode):
```
SUBS PC,R14,4   ;both PC=R14_irq-4, and CPSR=SPSR_irq
```

Software Interrupt
Generated by a software interrupt instruction (SWI). Recommended to request a supervisor (operating system) function. The SWI instruction may also contain a parameter in the 'comment field' of the opcode:
In case that your main program issues SWIs from both inside of THUMB and ARM states, then your exception handler must separate between 24bit comment fields in ARM opcodes, and 8bit comment fields in THUMB opcodes (if necessary determine old state by examining T Bit in SPSR_svc); However, in Little Endian mode, you could use only the most significant 8bits of the 24bit ARM comment field (as done in the GBA, for example) - the exception handler could then process the BYTE at [R14-2], regardless of whether it's been called from ARM or THUMB state.
To return from Supervisor Mode (continuing at following opcode):
```
MOVS PC,R14   ;both PC=R14_svc, and CPSR=SPSR_svc
```
Note: Like all other exceptions, SWIs are always executed in ARM state, no matter whether it's been caused by an ARM or THUMB state SWI instruction.

Undefined Instruction Exception (supported by ARMv3 and up)
This exception is generated when the CPU comes across an instruction which it cannot handle. Most likely signalizing that the program has locked up, and that an errormessage should be displayed.
However, it might be also used to emulate custom functions, ie. as an additional 'SWI' instruction (which'd use R14_und and SPSR_und though, and it'd thus allow to execute the Undefined Instruction handler from inside of Supervisor mode without having to save R14_svc and SPSR_svc).
To return from Undefined Mode (continuing at following opcode):
```
MOVS PC,R14   ;both PC=R14_und, and CPSR=SPSR_und
```
Note that not all unused opcodes are necessarily producing an exception, for example, an ARM state Multiply instruction with Bit6=1 would be blindly accepted as 'legal' opcode.

Abort (supported by ARMv3 and up)
Aborts (page faults) are mostly supposed for virtual memory systems (ie. not used in GBA, as far as I know),

otherwise they might be used just to display an error message. Two types of aborts exists:
- Prefetch Abort (occurs during an instruction prefetch)
- Prefetch Abort (also occurs on BKPT opcodes, ARMv5 and up)
- Data Abort (occurs during a data access)
A virtual memory systems abort handler would then most likely determine the fault address: For prefetch abort that's just "R14_abt-4". For Data abort, the THUMB or ARM instruction at "R14_abt-8" needs to be 'disassembled' in order to determine the addressed data in memory.
The handler would then fix the error by loading the respective memory page into physical memory, and then retry to execute the SAME instruction again, by returning as follows:

```
  prefetch abort: SUBS PC,R14,#4   ;PC=R14_abt-4, and CPSR=SPSR_abt
  data abort:     SUBS PC,R14,#8   ;PC=R14_abt-8, and CPSR=SPSR_abt
```

Separate exception vectors for prefetch/data abort exists, each should use the respective return instruction as shown above.

Address Exceeds 26bit
This exception can occur only on old ARM CPUs with 26bit address scheme (or in 26bit backwards compatibility mode).

Reset
Forces PC=VVVV0000h, and forces control bits of CPSR to T=0 (ARM state), F=1 and I=1 (disable FIQ and IRQ), and M4-0=10011b (Supervisor mode).

ARM CPU Memory Alignments

The CPU does NOT support accessing mis-aligned addresses (which would be rather slow because it'd have to merge/split that data into two accesses).
When reading/writing code/data to/from memory, Words and Halfwords must be located at well-aligned memory address, ie. 32bit words aligned by 4, and 16bit halfwords aligned by 2.

Mis-aligned STR,STRH,STM,LDM,LDRD,STRD,PUSH,POP (forced align)
The mis-aligned low bit(s) are ignored, the memory access goes to a forcibly aligned (rounded-down) memory address.
For LDRD/STRD, it isn't clearly defined if the address must be aligned by 8 (on the NDS, align-4 seems to be okay) (align-8 may be required on other CPUs with 64bit databus).

Mis-aligned LDR,SWP (rotated read)
Reads from forcibly aligned address "addr AND (NOT 3)", and does then rotate the data as "ROR (addr AND 3)*8". That effect is internally used by LDRB and LDRH opcodes (which do then mask-out the unused bits).
The SWP opcode works like a combination of LDR and STR, that means, it does read-rotated, but does write-unrotated.

Mis-aligned LDRH,LDRSH (does or does not do strange things)
On ARM9 aka ARMv5 aka NDS9:

```
  LDRH Rd,[odd]   -->  LDRH Rd,[odd-1]        ;forced align
  LDRSH Rd,[odd]  -->  LDRSH Rd,[odd-1]       ;forced align
```

On ARM7 aka ARMv4 aka NDS7/GBA:

```
  LDRH Rd,[odd]   -->  LDRH Rd,[odd-1] ROR 8  ;read to bit0-7 and bit24-31
  LDRSH Rd,[odd]  -->  LDRSB Rd,[odd]         ;sign-expand BYTE value
```

Mis-aligned PC/R15 (branch opcodes, or MOV/ALU/LDR with Rd=R15)
For ARM code, the low bits of the target address should be usually zero, otherwise, R15 is forcibly aligned by clearing the lower two bits.
For THUMB code, the low bit of the target address may/should/must be set, the bit is (or is not) interpreted as

thumb-bit (depending on the opcode), and R15 is then forcibly aligned by clearing the lower bit.
In short, R15 will be always forcibly aligned, so mis-aligned branches won't have effect on subsequent opcodes that use R15, or [R15+disp] as operand.

ARM Instruction Summary

Modification of CPSR flags is optional for all {S} instructions.

Logical ALU Operations

```
  Instruction                    Cycles   Flags  Expl.
  MOV{cond}{S} Rd,Op2            1S+x+y    NZc-  Rd = Op2
  MVN{cond}{S} Rd,Op2            1S+x+y    NZc-  Rd = NOT Op2
  ORR{cond}{S} Rd,Rn,Op2         1S+x+y    NZc-  Rd = Rn OR Op2
  EOR{cond}{S} Rd,Rn,Op2         1S+x+y    NZc-  Rd = Rn XOR Op2
  AND{cond}{S} Rd,Rn,Op2         1S+x+y    NZc-  Rd = Rn AND Op2
  BIC{cond}{S} Rd,Rn,Op2         1S+x+y    NZc-  Rd = Rn AND NOT Op2
  TST{cond}{P}    Rn,Op2         1S+x      NZc-  Void = Rn AND Op2
  TEQ{cond}{P}    Rn,Op2         1S+x      NZc-  Void = Rn XOR Op2
```

Add x=1I cycles if Op2 shifted-by-register. Add y=1S+1N cycles if Rd=R15.
Carry flag affected only if Op2 contains a non-zero shift amount.

Arithmetic ALU Operations

```
  Instruction                    Cycles   Flags  Expl.
  ADD{cond}{S} Rd,Rn,Op2         1S+x+y    NZCV  Rd = Rn+Op2
  ADC{cond}{S} Rd,Rn,Op2         1S+x+y    NZCV  Rd = Rn+Op2+Cy
  SUB{cond}{S} Rd,Rn,Op2         1S+x+y    NZCV  Rd = Rn-Op2
  SBC{cond}{S} Rd,Rn,Op2         1S+x+y    NZCV  Rd = Rn-Op2+Cy-1
  RSB{cond}{S} Rd,Rn,Op2         1S+x+y    NZCV  Rd = Op2-Rn
  RSC{cond}{S} Rd,Rn,Op2         1S+x+y    NZCV  Rd = Op2-Rn+Cy-1
  CMP{cond}{P}    Rn,Op2         1S+x      NZCV  Void = Rn-Op2
  CMN{cond}{P}    Rn,Op2         1S+x      NZCV  Void = Rn+Op2
```

Add x=1I cycles if Op2 shifted-by-register. Add y=1S+1N cycles if Rd=R15.

Multiply

```
  Instruction                    Cycles   Flags  Expl.
  MUL{cond}{S} Rd,Rm,Rs          1S+mI     NZx-  Rd = Rm*Rs
  MLA{cond}{S} Rd,Rm,Rs,Rn       1S+mI+1I  NZx-  Rd = Rm*Rs+Rn
  UMULL{cond}{S} RdLo,RdHi,Rm,Rs 1S+mI+1I  NZx-  RdHiLo = Rm*Rs
  UMLAL{cond}{S} RdLo,RdHi,Rm,Rs 1S+mI+2I  NZx-  RdHiLo = Rm*Rs+RdHiLo
  SMULL{cond}{S} RdLo,RdHi,Rm,Rs 1S+mI+1I  NZx-  RdHiLo = Rm*Rs
  SMLAL{cond}{S} RdLo,RdHi,Rm,Rs 1S+mI+2I  NZx-  RdHiLo = Rm*Rs+RdHiLo
  SMLAxy{cond}  Rd,Rm,Rs,Rn     ARMv5TE(xP)  ----q Rd=HalfRm*HalfRs+Rn
  SMLAWy{cond}  Rd,Rm,Rs,Rn     ARMv5TE(xP)  ----q Rd=(Rm*HalfRs)/10000h+Rn
  SMULWy{cond}  Rd,Rm,Rs        ARMv5TE(xP)  ----  Rd=(Rm*HalfRs)/10000h
  SMLALxy{cond} RdLo,RdHi,Rm,Rs ARMv5TE(xP)  ----  RdHiLo=RdHiLo+HalfRm*HalfRs
  SMULxy{cond}  Rd,Rm,Rs        ARMv5TE(xP)  ----  Rd=HalfRm*HalfRs
```

Memory Load/Store

```
  Instruction                    Cycles    Flags  Expl.
  LDR{cond}{B}{T} Rd,<Address>   1S+1N+1I+y ----  Rd=[Rn+/-<offset>]
  LDR{cond}H      Rd,<Address>   1S+1N+1I+y ----  Load Unsigned halfword
  LDR{cond}D      Rd,<Address>              ----  Load Dword ARMv5TE
  LDR{cond}SB     Rd,<Address>   1S+1N+1I+y ----  Load Signed byte
  LDR{cond}SH     Rd,<Address>   1S+1N+1I+y ----  Load Signed halfword
  LDM{cond}{amod} Rn{!},<Rlist>{^} nS+1N+1I+y ----  Load Multiple
  STR{cond}{B}{T} Rd,<Address>   2N         ----  [Rn+/-<offset>]=Rd
  STR{cond}H      Rd,<Address>   2N         ----  Store halfword
  STR{cond}D      Rd,<Address>              ----  Store Dword ARMv5TE
  STM{cond}{amod} Rn{!},<Rlist>{^} (n-1)S+2N  ----  Store Multiple
```

```
SWP{cond}{B}    Rd,Rm,[Rn]       1S+2N+1I   ----  Rd=[Rn], [Rn]=Rm
PLD             <Address>        1S         ----  Prepare Cache ARMv5TE
```

For LDR/LDM, add y=1S+1N if Rd=R15, or if R15 in Rlist.

Jumps, Calls, CPSR Mode, and others

```
Instruction                 Cycles  Flags  Expl.
B{cond}   label             2S+1N    ----  PC=$+8+/-32M
BL{cond}  label             2S+1N    ----  PC=$+8+/-32M, LR=$+4
BX{cond}  Rn                2S+1N    ----  PC=Rn, T=Rn.0 (THUMB/ARM)
BLX{cond} Rn                2S+1N    ----  PC=Rn, T=Rn.0, LR=PC+4, ARM9
BLX       label             2S+1N    ----  PC=PC+$+/-32M, LR=$+4, T=1, ARM9
MRS{cond} Rd,Psr            1S       ----  Rd=Psr
MSR{cond} Psr{_field},Op    1S      (psr)  Psr[field]=Op
SWI{cond} Imm24bit          2S+1N    ----  PC=8, ARM Svc mode, LR=$+4
BKPT      Imm16bit          ???      ----  PC=C, ARM Abt mode, LR=$+4 ARM9
The Undefined Instruction   2S+1I+1N ----  PC=4, ARM Und mode, LR=$+4
cond=false                  1S       ----  Any opcode with condition=false
NOP                         1S       ----  R0=R0

CLZ{cond} Rd,Rm             ???      ----    Count Leading Zeros ARMv5
QADD{cond} Rd,Rm,Rn                  ----q   Rd=Rm+Rn       ARMv5TE(xP)
QSUB{cond} Rd,Rm,Rn                  ----q   Rd=Rm-Rn       ARMv5TE(xP)
QDADD{cond} Rd,Rm,Rn                 ----q   Rd=Rm+Rn*2     ARMv5TE(xP)
QDSUB{cond} Rd,Rm,Rn                 ----q   Rd=Rm-Rn*2     ARMv5TE(xP)
```

Coprocessor Functions (if any)

```
Instruction                          Cycles  Flags  Expl.
CDP{cond} Pn,<cpopc>,Cd,Cn,Cm{,<cp>} 1S+bI   ----  Coprocessor specific
STC{cond}{L} Pn,Cd,<Address>          (n-1)S+2N+bI  [address] = CRd
LDC{cond}{L} Pn,Cd,<Address>          (n-1)S+2N+bI  CRd = [address]
MCR{cond} Pn,<cpopc>,Rd,Cn,Cm{,<cp>} 1S+bI+1C      CRn = Rn {<op> CRm}
MRC{cond} Pn,<cpopc>,Rd,Cn,Cm{,<cp>} 1S+(b+1)I+1C  Rn = CRn {<op> CRm}
CDP2,STC2,LDC2,MCR2,MRC2 - ARMv5 Extensions similar above, without {cond}
MCRR{cond} Pn,<cpopc>,Rd,Rn,Cm  ;write Rd,Rn to coproc ARMv5TE
MRRC{cond} Pn,<cpopc>,Rd,Rn,Cm  ;read Rd,Rn from coproc ARMv5TE
```

ARM Binary Opcode Format

```
|..3 .................2 .................1 .................0|
|1_0_9_8_7_6_5_4_3_2_1_0_9_8_7_6_5_4_3_2_1_0_9_8_7_6_5_4_3_2_1_0|
|_Cond__|0_0_0|___Op__|S|__Rn___|__Rd___|__Shift__|Typ|0|__Rm___| DataProc
|_Cond__|0_0_0|___Op__|S|__Rn___|__Rd___|__Rs___|0|Typ|1|__Rm___| DataProc
|_Cond__|0_0_1|___Op__|S|__Rn___|__Rd___|_Shift_|___Immediate___| DataProc
|_Cond__|0_0_1_1_0|P|1|0|_Field_|__Rd___|_Shift_|___Immediate___| PSR Imm
|_Cond__|0_0_0_1_0|P|L|0|_Field_|__Rd___|0_0_0_0|0_0_0_0|__Rm___| PSR Reg
|_Cond__|0_0_0_1_0_0_1_0_1_1_1_1_1_1_1_1_1_1_1_1|0_0|L|1|__Rn___| BX,BLX
|1_1_1_0|0_0_0_1_0_0_1_0|_____immediate_________|0_1_1_1|_immed_| BKPT ARM9
|_Cond__|0_0_0_1_0_1_1_0_1_1_1_1|__Rd___|1_1_1_1|0_0_0_1|__Rm___| CLZ  ARM9
|_Cond__|0_0_0_1_0|Op_|0|__Rn___|__Rd___|0_0_0_0|0_1_0_1|__Rm___| QALU ARM9
|_Cond__|0_0_0_0_0_0|A|S|__Rd___|__Rn___|__Rs___|1_0_0_1|__Rm___| Multiply
|_Cond__|0_0_0_0_1|U|A|S|_RdHi__|_RdLo__|__Rs___|1_0_0_1|__Rm___| MulLong
|_Cond__|0_0_0_1_0|Op_|0|Rd/RdHi|Rn/RdLo|__Rs___|1|y|x|0|__Rm___| MulHalfARM9
|_Cond__|0_0_0_1_0|B|0_0|__Rn___|__Rd___|0_0_0_0|1_0_0_1|__Rm___| TransSwp12
|_Cond__|0_0_0|P|U|0|W|L|__Rn___|__Rd___|0_0_0_0|1|S|H|1|__Rm___| TransReg10
|_Cond__|0_0_0|P|U|1|W|L|__Rn___|__Rd___|OffsetH|1|S|H|1|OffsetL| TransImm10
|_Cond__|0_1_0|P|U|B|W|L|__Rn___|__Rd___|________Offset________| TransImm9
|_Cond__|0_1_1|P|U|B|W|L|__Rn___|__Rd___|__Shift__|Typ|0|__Rm___| TransReg9
|_Cond__|0_1_1|_____________xxx_____________________|1|__xxx__| Undefined
|_Cond__|1_0_0|P|U|S|W|L|__Rn___|_________Register_List________| BlockTrans
|_Cond__|1_0_1|L|_____________________Offset____________________| B,BL,BLX
|_Cond__|1_1_0|P|U|N|W|L|__Rn___|__CRd__|__CP#__|____Offset_____| CoDataTrans
|_Cond__|1_1_0_0_0_1_0|L|__Rn___|__Rd___|__CP#__|_CPopc_|__CRm__| CoRR ARM9
|_Cond__|1_1_1_0|_CPopc_|__CRn__|__CRd__|__CP#__|_CP__|0|__CRm__| CoDataOp
```

```
|_Cond__|1_1_1_0|CPopc|L|__CRn__|__Rd___|__CP#__|_CP__|1|__CRm__| CoRegTrans
|_Cond__|1_1_1_1|_____________Ignored_by_Processor_____________| SWI
```

ARM Opcodes: Branch and Branch with Link (B, BL, BX, BLX, SWI, BKPT)

Branch and Branch with Link (B, BL, BLX_imm)
Branch (B) is supposed to jump to a subroutine. Branch with Link is meant to be used to call to a subroutine, return address is then saved in R14.

```
  Bit    Expl.
  31-28  Condition (must be 1111b for BLX)
  27-25  Must be "101" for this instruction
  24     Opcode (0-1) (or Halfword Offset for BLX)
          0: B{cond} label    ;branch            PC=PC+8+nn*4
          1: BL{cond} label   ;branch/link       PC=PC+8+nn*4, LR=PC+4
          H: BLX label ;ARM9  ;branch/link/thumb PC=PC+8+nn*4+H*2, LR=PC+4, T=1
  23-0   nn - Signed Offset, step 4      (-32M..+32M in steps of 4)
```

Branch with Link can be used to 'call' to a sub-routine, which may then 'return' by MOV PC,R14 for example.
Execution Time: 2S + 1N
Return: No flags affected.

Branch and Exchange (BX, BLX_reg)

```
  Bit    Expl.
  31-28  Condition
  27-8   Must be "0001.0010.1111.1111.1111" for this instruction
  7-4    Opcode
          0001b: BX{cond}  Rn    ;PC=Rn, T=Rn.0  (ARMv4T and ARMv5 and up)
          0011b: BLX{cond} Rn    ;PC=Rn, T=Rn.0, LR=PC+4    (ARMv5 and up)
  3-0    Rn - Operand Register  (R0-R14)
```

Switching to THUMB Mode: Set Bit 0 of the value in Rn to 1, program continues then at Rn-1 in THUMB mode.
Results in undefined behaviour if using R15 (PC+8 itself) as operand. Using BLX R14 is possible (sets PC=Old_LR, and New_LR=retadr).
Execution Time: 2S + 1N
Return: No flags affected.

Branch via ALU, LDR, LDM
Most ALU, LDR, LDM opcodes can also change PC/R15.

Software Interrupt (SWI/BKPT) (svc/abt exceptions)
SWI supposed for calls to the operating system - Enter Supervisor mode (SVC) in ARM state. BKPT intended for debugging - enters Abort mode in ARM state via Prefetch Abort vector.

```
  Bit    Expl.
  31-28  Condition (must be 1110b for BKPT, ie. Condition=always)
  27-24  Opcode
          1111b: SWI{cond} nn   ;software interrupt
          0001b: BKPT      nn   ;breakpoint (ARMv5 and up)
  For SWI:
   23-0   nn - Comment Field, ignored by processor (24bit value)
  For BKPT:
   23-20  Must be 0010b for BKPT
   19-8   nn - upper 12bits of comment field, ignored by processor
   7-4    Must be 0111b for BKPT
   3-0    nn - lower 4bits of comment field, ignored by processor
```

Execution Time: 2S+1N
The exception handler may interprete the SWI Comment Field by examining the lower 24bit of the 32bit opcode opcode at [R14_svc-4]. If your are also using SWI's from inside of THUMB, then the SWI handler must

examine the T Bit SPSR_svc in order to determine whether it's been a THUMB SWI - and if so, examine the lower 8bit of the 16bit opcode opcode at [R14_svc-2].
For Returning from SWI use "MOVS PC,R14", that instruction does restore both PC and CPSR, ie. PC=R14_svc, and CPSR=SPSR_svc.
Nesting SWIs: SPSR_svc and R14_svc should be saved on stack before either invoking nested SWIs, or (if the IRQ handler uses SWIs) before enabling IRQs.
Execution SWI/BKPT:

```
R14_svc=PC+4     R14_abt=PC+4   ;save return address
SPSR_svc=CPSR    SPSR_abt=CPSR  ;save CPSR flags
CPSR=<changed>   CPSR=<changed> ;Enter svc/abt, ARM state, IRQs disabled
PC=VVVV0008h     PC=VVVV000Ch   ;jump to SWI/PrefetchAbort vector address
```

Undefined Instruction (und exception)

```
Bit    Expl.
31-28  Condition
27-25  Must be 011b for this instruction
24-5   Reserved for future use
4      Must be 1b for this instruction
3-0    Reserved for future use
```

No assembler mnemonic exists, following bitstreams are (not) reserved.

```
cond011xxxxxxxxxxxxxxxxxxxx1xxxx - reserved for future use (except below).
cond01111111xxxxxxxxxxxx1111xxxx - free for user.
```

Execution time: 2S+1I+1N.

ARM Opcodes: Data Processing (ALU)

Opcode Format

```
Bit    Expl.
31-28  Condition
27-26  Must be 00b for this instruction
25     I - Immediate 2nd Operand Flag (0=Register, 1=Immediate)
24-21  Opcode (0-Fh)               ;*=Arithmetic, otherwise Logical
         0: AND{cond}{S} Rd,Rn,Op2    ;AND logical       Rd = Rn AND Op2
         1: EOR{cond}{S} Rd,Rn,Op2    ;XOR logical       Rd = Rn XOR Op2
         2: SUB{cond}{S} Rd,Rn,Op2 ;* ;subtract          Rd = Rn-Op2
         3: RSB{cond}{S} Rd,Rn,Op2 ;* ;subtract reversed Rd = Op2-Rn
         4: ADD{cond}{S} Rd,Rn,Op2 ;* ;add               Rd = Rn+Op2
         5: ADC{cond}{S} Rd,Rn,Op2 ;* ;add with carry    Rd = Rn+Op2+Cy
         6: SBC{cond}{S} Rd,Rn,Op2 ;* ;sub with carry    Rd = Rn-Op2+Cy-1
         7: RSC{cond}{S} Rd,Rn,Op2 ;* ;sub cy. reversed  Rd = Op2-Rn+Cy-1
         8: TST{cond}{P}    Rn,Op2    ;test            Void = Rn AND Op2
         9: TEQ{cond}{P}    Rn,Op2    ;test exclusive  Void = Rn XOR Op2
         A: CMP{cond}{P}    Rn,Op2 ;* ;compare         Void = Rn-Op2
         B: CMN{cond}{P}    Rn,Op2 ;* ;compare neg.    Void = Rn+Op2
         C: ORR{cond}{S} Rd,Rn,Op2    ;OR logical        Rd = Rn OR Op2
         D: MOV{cond}{S} Rd,Op2       ;move              Rd = Op2
         E: BIC{cond}{S} Rd,Rn,Op2    ;bit clear         Rd = Rn AND NOT Op2
         F: MVN{cond}{S} Rd,Op2       ;not               Rd = NOT Op2
20     S - Set Condition Codes (0=No, 1=Yes) (Must be 1 for opcode 8-B)
19-16  Rn - 1st Operand Register (R0..R15) (including PC=R15)
           Must be 0000b for MOV/MVN.
15-12  Rd - Destination Register (R0..R15) (including PC=R15)
           Must be 0000b (or 1111b) for CMP/CMN/TST/TEQ{P}.
When above Bit 25 I=0 (Register as 2nd Operand)
  When below Bit 4 R=0 - Shift by Immediate
    11-7   Is - Shift amount   (1-31, 0=Special/See below)
  When below Bit 4 R=1 - Shift by Register
    11-8   Rs - Shift register (R0-R14) - only lower 8bit 0-255 used
    7      Reserved, must be zero  (otherwise multiply or undefined opcode)
  6-5    Shift Type (0=LSL, 1=LSR, 2=ASR, 3=ROR)
```

```
    4      R - Shift by Register Flag (0=Immediate, 1=Register)
    3-0    Rm - 2nd Operand Register (R0..R15) (including PC=R15)
  When above Bit 25 I=1 (Immediate as 2nd Operand)
    11-8   Is - ROR-Shift applied to nn (0-30, in steps of 2)
    7-0    nn - 2nd Operand Unsigned 8bit Immediate
```

Second Operand (Op2)
This may be a shifted register, or a shifted immediate. See Bit 25 and 11-0.
Unshifted Register: Specify Op2 as "Rm", assembler converts to "Rm,LSL#0".
Shifted Register: Specify as "Rm,SSS#Is" or "Rm,SSS Rs" (SSS=LSL/LSR/ASR/ROR).
Immediate: Specify as 32bit value, for example: "#000NN000h", assembler should automatically convert into "#0NNh,ROR#0ssh" as far as possible (ie. as far as a section of not more than 8bits of the immediate is non-zero).

Zero Shift Amount (Shift Register by Immediate, with Immediate=0)

```
  LSL#0: No shift performed, ie. directly Op2=Rm, the C flag is NOT affected.
  LSR#0: Interpreted as LSR#32, ie. Op2 becomes zero, C becomes Bit 31 of Rm.
  ASR#0: Interpreted as ASR#32, ie. Op2 and C are filled by Bit 31 of Rm.
  ROR#0: Interpreted as RRX#1 (RCR), like ROR#1, but Op2 Bit 31 set to old C.
```

In source code, LSR#32, ASR#32, and RRX#1 should be specified as such - attempts to specify LSR#0, ASR#0, or ROR#0 will be internally converted to LSL#0 by the assembler.

Using R15 (PC)
When using R15 as Destination (Rd), note below CPSR description and Execution time description.
When using R15 as operand (Rm or Rn), the returned value depends on the instruction: PC+12 if I=0,R=1 (shift by register), otherwise PC+8 (shift by immediate).

Returned CPSR Flags
If S=1, Rd<>R15, logical operations (AND,EOR,TST,TEQ,ORR,MOV,BIC,MVN):

```
  V=not affected
  C=carryflag of shift operation (not affected if LSL#0 or Rs=00h)
  Z=zeroflag of result
  N=signflag of result (result bit 31)
```

If S=1, Rd<>R15, arithmetic operations (SUB,RSB,ADD,ADC,SBC,RSC,CMP,CMN):

```
  V=overflowflag of result
  C=carryflag of result
  Z=zeroflag of result
  N=signflag of result (result bit 31)
```

IF S=1, with unused Rd bits=1111b, {P} opcodes (CMPP/CMNP/TSTP/TEQP):

```
  R15=result  ;modify PSR bits in R15, ARMv2 and below only.
  In user mode only N,Z,C,V bits of R15 can be changed.
  In other modes additionally I,F,M1,M0 can be changed.
  The PC bits in R15 are left unchanged in all modes.
```

If S=1, Rd=R15; should not be used in user mode:

```
  CPSR = SPSR_<current mode>
  PC = result
  For example: MOVS PC,R14  ;return from SWI (PC=R14_svc, CPSR=SPSR_svc).
```

If S=0: Flags are not affected (not allowed for CMP,CMN,TEQ,TST).

The instruction "MOV R0,R0" is used as "NOP" opcode in 32bit ARM state.
Execution Time: (1+p)S+rI+pN. Whereas r=1 if I=0 and R=1 (ie. shift by register); otherwise r=0. And p=1 if Rd=R15; otherwise p=0.

ARM Opcodes: Multiply and Multiply-Accumulate (MUL, MLA)

Opcode Format

```
  Bit    Expl.
```

```
  31-28  Condition
  27-25  Must be 000b for this instruction
  24-21  Opcode
          0000b: MUL{cond}{S}   Rd,Rm,Rs        ;multiply   Rd = Rm*Rs
          0001b: MLA{cond}{S}   Rd,Rm,Rs,Rn     ;mul.& accumulate Rd = Rm*Rs+Rn
          0100b: UMULL{cond}{S} RdLo,RdHi,Rm,Rs ;multiply   RdHiLo=Rm*Rs
          0101b: UMLAL{cond}{S} RdLo,RdHi,Rm,Rs ;mul.& acc. RdHiLo=Rm*Rs+RdHiLo
          0110b: SMULL{cond}{S} RdLo,RdHi,Rm,Rs ;sign.mul.  RdHiLo=Rm*Rs
          0111b: SMLAL{cond}{S} RdLo,RdHi,Rm,Rs ;sign.m&a.  RdHiLo=Rm*Rs+RdHiLo
          1000b: SMLAxy{cond}   Rd,Rm,Rs,Rn     ;Rd=HalfRm*HalfRs+Rn
          1001b: SMLAWy{cond}   Rd,Rm,Rs,Rn     ;Rd=(Rm*HalfRs)/10000h+Rn
          1001b: SMULWy{cond}   Rd,Rm,Rs        ;Rd=(Rm*HalfRs)/10000h
          1010b: SMLALxy{cond}  RdLo,RdHi,Rm,Rs ;RdHiLo=RdHiLo+HalfRm*HalfRs
          1011b: SMULxy{cond}   Rd,Rm,Rs        ;Rd=HalfRm*HalfRs
  20     S - Set Condition Codes (0=No, 1=Yes) (Must be 0 for Halfword mul)
  19-16  Rd (or RdHi) - Destination Register (R0-R14)
  15-12  Rn (or RdLo) - Accumulate Register  (R0-R14) (Set to 0000b if unused)
  11-8   Rs - Operand Register               (R0-R14)
  For Non-Halfword Multiplies
    7-4  Must be 1001b for these instructions
  For Halfword Multiplies
    7    Must be 1 for these instructions
    6    y - Rs Top/Bottom flag (0=B=Lower 16bit, 1=T=Upper 16bit)
    5    x - Rm Top/Bottom flag (as above), or 0 for SMLAW, or 1 for SMULW
    4    Must be 0 for these instructions
  3-0    Rm - Operand Register               (R0-R14)
```

Multiply and Multiply-Accumulate (MUL, MLA)
Restrictions: Rd may not be same as Rm. Rd,Rn,Rs,Rm may not be R15.
Note: Only the lower 32bit of the internal 64bit result are stored in Rd, thus no sign/zero extension is required and MUL and MLA can be used for both signed and unsigned calculations!
Execution Time: 1S+mI for MUL, and 1S+(m+1)I for MLA. Whereas 'm' depends on whether/how many most significant bits of Rs are all zero or all one. That is m=1 for Bit 31-8, m=2 for Bit 31-16, m=3 for Bit 31-24, and m=4 otherwise.
Flags (if S=1): Z=zeroflag, N=signflag, C=destroyed (ARMv4 and below) or C=not affected (ARMv5 and up), V=not affected. MUL/MLA supported by ARMv2 and up.

Multiply Long and Multiply-Accumulate Long (MULL, MLAL)
Optionally supported, INCLUDED in ARMv3M, EXCLUDED in ARMv4xM/ARMv5xM.
Restrictions: RdHi,RdLo,Rm must be different registers. R15 may not be used.
Execution Time: 1S+(m+1)I for MULL, and 1S+(m+2)I for MLAL. Whereas 'm' depends on whether/how many most significant bits of Rs are "all zero" (UMULL/UMLAL) or "all zero or all one" (SMULL,SMLAL). That is m=1 for Bit31-8, m=2 for Bit31-16, m=3 for Bit31-24, and m=4 otherwise.
Flags (if S=1): Z=zeroflag, N=signflag, C=destroyed (ARMv4 and below) or C=not affected (ARMv5 and up), V=destroyed??? (ARMv4 and below???) or V=not affected (ARMv5 and up).

Signed Halfword Multiply (SMLAxy,SMLAWy,SMLALxy,SMULxy,SMULWy)
Supported by E variants of ARMv5 and up, ie. ARMv5TE(xP).
Q-flag gets set on 32bit SMLAxy/SMLAWy addition overflows, however, the result is NOT truncated (as it'd be done with QADD opcodes).
Q-flag is NOT affected on (rare) 64bit SMLALxy addition overflows.
SMULxy/SMULWy cannot overflow, and thus leave Q-flag unchanged as well.
NZCV-flags are not affected by Halfword multiplies.
Execution Time: 1S+Interlock (SMULxy,SMLAxy,SMULWx,SMLAWx)
Execution Time: 1S+1I+Interlock (SMLALxy)

ARM Opcodes: Special ARM9 Instructions (CLZ, QADD/QSUB)

Count Leading Zeros (CLZ)

```
Bit    Expl.
31-28  Condition
27-16  Must be 0001.0110.1111b for this instruction
       Opcode (fixed)
         CLZ{cond} Rd,Rm  ;Rd=Number of leading zeros in Rm
15-12  Rd - Destination Register              (R0-R14)
11-4   Must be 1111.0001b for this instruction
3-0    Rm - Source Register                   (R0-R14)
```

CLZ supported by ARMv5 and up. Execution time: 1S.
Return: No Flags affected. Rd=0..32.

Opcode Format (QADD/QSUB)

```
Bit    Expl.
31-28  Condition
27-24  Must be 0001b for this instruction
23-20  Opcode
        0000b: QADD{cond}  Rd,Rm,Rn    ;Rd=Rm+Rn
        0010b: QSUB{cond}  Rd,Rm,Rn    ;Rd=Rm-Rn
        0100b: QDADD{cond} Rd,Rm,Rn    ;Rd=Rm+Rn*2 (doubled)
        0110b: QDSUB{cond} Rd,Rm,Rn    ;Rd=Rm-Rn*2 (doubled)
19-16  Rn - Second Source Register            (R0-R14)
15-12  Rd - Destination Register              (R0-R14)
11-4   Must be 00000101b for this instruction
3-0    Rm - First Source Register             (R0-R14)
```

Supported by E variants of ARMv5 and up, ie. ARMv5TE(xP).
Execution time: 1S+Interlock.
Results truncated to signed 32bit range in case of overflows, with the Q-flag being set (and being left unchanged otherwise). NZCV flags are not affected.
Note: Rn*2 is internally processed first, and may get truncated - even if the final result would fit into range.

ARM Opcodes: PSR Transfer (MRS, MSR)

Opcode Format

These instructions occupy an unused area (TEQ,TST,CMP,CMN with S=0) of ALU opcodes.

```
Bit    Expl.
31-28  Condition
27-26  Must be 00b for this instruction
25     I - Immediate Operand Flag  (0=Register, 1=Immediate) (Zero for MRS)
24-23  Must be 10b for this instruction
22     Psr - Source/Destination PSR  (0=CPSR, 1=SPSR_<current mode>)
21     Opcode
         0: MRS{cond} Rd,Psr          ;Rd = Psr
         1: MSR{cond} Psr{_field},Op  ;Psr[field] = Op
20     Must be 0b for this instruction (otherwise TST,TEQ,CMP,CMN)
For MRS:
  19-16   Must be 1111b for this instruction (otherwise SWP)
  15-12   Rd - Destination Register  (R0-R14)
  11-0    Not used, must be zero.
For MSR:
  19      f  write to flags field     Bit 31-24 (aka _flg)
  18      s  write to status field    Bit 23-16 (reserved, don't change)
  17      x  write to extension field Bit 15-8  (reserved, don't change)
  16      c  write to control field   Bit 7-0   (aka _ctl)
  15-12   Not used, must be 1111b.
For MSR Psr,Rm (I=0)
  11-4    Not used, must be zero. (otherwise BX)
  3-0     Rm - Source Register <op>  (R0-R14)
For MSR Psr,Imm (I=1)
```

```
  11-8    Shift applied to Imm   (ROR in steps of two 0-30)
  7-0     Imm - Unsigned 8bit Immediate
  In source code, a 32bit immediate should be specified as operand.
  The assembler should then convert that into a shifted 8bit value.
```

MSR/MRS and CPSR/SPSR supported by ARMv3 and up.
ARMv2 and below contained PSR flags in R15, accessed by CMP/CMN/TST/TEQ{P}.
The field mask bits specify which bits of the destination Psr are write-able (or write-protected), one or more of these bits should be set, for example, CPSR_fsxc (aka CPSR aka CPSR_all) unlocks all bits (see below user mode restriction though).
Restrictions:
In non-privileged mode (user mode): only condition code bits of CPSR can be changed, control bits can't.
Only the SPSR of the current mode can be accessed; In User and System modes no SPSR exists.
The T-bit may not be changed; for THUMB/ARM switching use BX instruction.
Unused Bits in CPSR are reserved for future use and should never be changed (except for unused bits in the flags field).
Execution Time: 1S.

Note: The A22i assembler recognizes MOV as alias for both MSR and MRS because it is practically not possible to remember whether MSR or MRS was the load or store opcode, and/or whether it does load to or from the Psr register.

ARM Opcodes: Memory: Single Data Transfer (LDR, STR, PLD)

Opcode Format

```
Bit    Expl.
31-28  Condition (Must be 1111b for PLD)
27-26  Must be 01b for this instruction
25     I - Immediate Offset Flag (0=Immediate, 1=Shifted Register)
24     P - Pre/Post (0=post; add offset after transfer, 1=pre; before trans.)
23     U - Up/Down Bit (0=down; subtract offset from base, 1=up; add to base)
22     B - Byte/Word bit (0=transfer 32bit/word, 1=transfer 8bit/byte)
When above Bit 24 P=0 (Post-indexing, write-back is ALWAYS enabled):
  21     T - Memory Management (0=Normal, 1=Force non-privileged access)
When above Bit 24 P=1 (Pre-indexing, write-back is optional):
  21     W - Write-back bit (0=no write-back, 1=write address into base)
20     L - Load/Store bit (0=Store to memory, 1=Load from memory)
        0: STR{cond}{B}{T} Rd,<Address>   ;[Rn+/-<offset>]=Rd
        1: LDR{cond}{B}{T} Rd,<Address>   ;Rd=[Rn+/-<offset>]
       (1: PLD <Address> ;Prepare Cache for Load, see notes below)
        Whereas, B=Byte, T=Force User Mode (only for POST-Indexing)
19-16  Rn - Base register               (R0..R15) (including R15=PC+8)
15-12  Rd - Source/Destination Register (R0..R15) (including R15=PC+12)
When above I=0 (Immediate as Offset)
  11-0   Unsigned 12bit Immediate Offset (0-4095, steps of 1)
When above I=1 (Register shifted by Immediate as Offset)
  11-7   Is - Shift amount      (1-31, 0=Special/See below)
  6-5    Shift Type             (0=LSL, 1=LSR, 2=ASR, 3=ROR)
  4      Must be 0 (Reserved, see The Undefined Instruction)
  3-0    Rm - Offset Register   (R0..R14) (not including PC=R15)
```

Instruction Formats for <Address>

An expression which generates an address:

```
<expression>                ;an immediate used as address
;*** restriction: must be located in range PC+/-4095+8, if so,
;*** assembler will calculate offset and use PC (R15) as base.
```

Pre-indexed addressing specification:

```
[Rn]                        ;offset = zero
[Rn, <#{+/-}expression>]{!}   ;offset = immediate
```

```
  [Rn, {+/-}Rm{,<shift>} ]{!}   ;offset = register shifted by immediate
```

Post-indexed addressing specification:

```
  [Rn], <#{+/-}expression>      ;offset = immediate
  [Rn], {+/-}Rm{,<shift>}       ;offset = register shifted by immediate
```

Whereas...

```
  <shift>  immediate shift such like LSL#4, ROR#2, etc. (see ALU opcodes).
  {!}      exclamation mark ("!") indicates write-back (Rn will be updated).
```

Notes

Shift amount 0 has special meaning, as described for ALU opcodes.
When writing a word (32bit) to memory, the address should be word-aligned.
When reading a byte from memory, upper 24 bits of Rd are zero-extended.
LDR PC,<op> on ARMv4 leaves CPSR.T unchanged.
LDR PC,<op> on ARMv5 sets CPSR.T to <op> Bit0, (1=Switch to Thumb).

When reading a word from a halfword-aligned address (which is located in the middle between two word-aligned addresses), the lower 16bit of Rd will contain [address] ie. the addressed halfword, and the upper 16bit of Rd will contain [Rd-2] ie. more or less unwanted garbage. However, by isolating lower bits this may be used to read a halfword from memory. (Above applies to little endian mode, as used in GBA.)

In a virtual memory based environment (ie. not in the GBA), aborts (ie. page faults) may take place during execution, if so, Rm and Rn should not specify the same register when post-indexing is used, as the abort-handler might have problems to reconstruct the original value of the register.

Return: CPSR flags are not affected.
Execution Time: For normal LDR: 1S+1N+1I. For LDR PC: 2S+2N+1I. For STR: 2N.

PLD <Address> ;Prepare Cache for Load

PLD must use following settings cond=1111b, P=1, B=1, W=0, L=1, Rd=1111b, the address may not use post-indexing, and may not use writeback, the opcode is encoded identical as LDRNVB R15,<Address>.
PLD signalizes to the memory system that a specific memory address will be soon accessed, the memory system may use this hint to prepare caching/pipelining, aside from that, PLD does not have any affect to the program logic, and behaves identical as NOP.
PLD supported by ARMv5TE only, not ARMv5, not ARMv5TExP.

ARM Opcodes: Memory: Halfword, Doubleword, and Signed Data Transfer

Opcode Format

```
  Bit    Expl.
  31-28  Condition
  27-25  Must be 000b for this instruction
  24     P - Pre/Post (0=post; add offset after transfer, 1=pre; before trans.)
  23     U - Up/Down Bit (0=down; subtract offset from base, 1=up; add to base)
  22     I - Immediate Offset Flag (0=Register Offset, 1=Immediate Offset)
  When above Bit 24 P=0 (Post-indexing, write-back is ALWAYS enabled):
    21     Not used, must be zero (0)
  When above Bit 24 P=1 (Pre-indexing, write-back is optional):
    21     W - Write-back bit (0=no write-back, 1=write address into base)
  20     L - Load/Store bit (0=Store to memory, 1=Load from memory)
  19-16  Rn - Base register               (R0-R15) (Including R15=PC+8)
  15-12  Rd - Source/Destination Register  (R0-R15) (Including R15=PC+12)
  11-8   When above Bit 22 I=0 (Register as Offset):
           Not used. Must be 0000b
         When above Bit 22 I=1 (immediate as Offset):
           Immediate Offset (upper 4bits)
```

```
  7      Reserved, must be set (1)
  6-5    Opcode (0-3)
         When Bit 20 L=0 (Store) (and Doubleword Load/Store):
          0: Reserved for SWP instruction
          1: STR{cond}H  Rd,<Address>  ;Store halfword   [a]=Rd
          2: LDR{cond}D  Rd,<Address>  ;Load Doubleword  R(d)=[a], R(d+1)=[a+4]
          3: STR{cond}D  Rd,<Address>  ;Store Doubleword [a]=R(d), [a+4]=R(d+1)
         When Bit 20 L=1 (Load):
          0: Reserved.
          1: LDR{cond}H  Rd,<Address>  ;Load Unsigned halfword (zero-extended)
          2: LDR{cond}SB Rd,<Address>  ;Load Signed byte (sign extended)
          3: LDR{cond}SH Rd,<Address>  ;Load Signed halfword (sign extended)
  4      Reserved, must be set (1)
  3-0    When above Bit 22 I=0:
           Rm - Offset Register            (R0-R14) (not including R15)
         When above Bit 22 I=1:
           Immediate Offset (lower 4bits)  (0-255, together with upper bits)
```

STRH,LDRH,LDRSB,LDRSH supported on ARMv4 and up.
STRD/LDRD supported on ARMv5TE only, not ARMv5, not ARMv5TExP.
STRD/LDRD: base writeback: Rn should not be same as R(d) or R(d+1).
STRD: index register: Rm should not be same as R(d) or R(d+1).
STRD/LDRD: Rd must be an even numbered register (R0,R2,R4,R6,R8,R10,R12).
STRD/LDRD: Address must be double-word aligned (multiple of eight).

Instruction Formats for <Address>

An expression which generates an address:

```
  <expression>                  ;an immediate used as address
  ;*** restriction: must be located in range PC+/-255+8, if so,
  ;*** assembler will calculate offset and use PC (R15) as base.
```

Pre-indexed addressing specification:

```
  [Rn]                          ;offset = zero
  [Rn, <#{+/-}expression>]{!}   ;offset = immediate
  [Rn, {+/-}Rm]{!}              ;offset = register
```

Post-indexed addressing specification:

```
  [Rn], <#{+/-}expression>      ;offset = immediate
  [Rn], {+/-}Rm                 ;offset = register
```

Whereas...

```
  {!}      exclamation mark ("!") indicates write-back (Rn will be updated).
```

Return: No Flags affected.
Execution Time: For Normal LDR, 1S+1N+1I. For LDR PC, 2S+2N+1I. For STRH 2N.

ARM Opcodes: Memory: Block Data Transfer (LDM, STM)

Opcode Format

```
  Bit    Expl.
  31-28  Condition
  27-25  Must be 100b for this instruction
  24     P - Pre/Post (0=post; add offset after transfer, 1=pre; before trans.)
  23     U - Up/Down Bit (0=down; subtract offset from base, 1=up; add to base)
  22     S - PSR & force user bit (0=No, 1=load PSR or force user mode)
  21     W - Write-back bit (0=no write-back, 1=write address into base)
  20     L - Load/Store bit (0=Store to memory, 1=Load from memory)
          0: STM{cond}{amod} Rn{!},<Rlist>{^}  ;Store (Push)
          1: LDM{cond}{amod} Rn{!},<Rlist>{^}  ;Load  (Pop)
          Whereas, {!}=Write-Back (W), and {^}=PSR/User Mode (S)
  19-16  Rn - Base register               (R0-R14) (not including R15)
  15-0   Rlist - Register List
  (Above 'offset' is meant to be the number of words specified in Rlist.)
```

Return: No Flags affected.
Execution Time: For normal LDM, nS+1N+1I. For LDM PC, (n+1)S+2N+1I. For STM (n-1)S+2N. Where n is the number of words transferred.

Addressing Modes {amod}
The IB,IA,DB,DA suffixes directly specify the desired U and P bits:

```
IB  increment before          ;P=1, U=1
IA  increment after           ;P=0, U=1
DB  decrement before          ;P=1, U=0
DA  decrement after           ;P=0, U=0
```

Alternately, FD,ED,FA,EA could be used, mostly to simplify mnemonics for stack transfers.

```
ED  empty stack, descending   ;LDM: P=1, U=1  ;STM: P=0, U=0
FD  full stack,  descending   ;     P=0, U=1  ;     P=1, U=0
EA  empty stack, ascending    ;     P=1, U=0  ;     P=0, U=1
FA  full stack,  ascending    ;     P=0, U=0  ;     P=1, U=1
```

Ie. the following expressions are aliases for each other:

```
STMFD=STMDB=PUSH   STMED=STMDA   STMFA=STMIB   STMEA=STMIA
LDMFD=LDMIA=POP    LDMED=LDMIB   LDMFA=LDMDA   LDMEA=LDMDB
```

Note: The equivalent THUMB functions use fixed organization:

```
PUSH/POP: full descending     ;base register SP (R13)
LDM/STM:  increment after     ;base register R0..R7
```

Descending is common stack organization as used in 80x86 and Z80 CPUs, SP is decremented when pushing/storing data, and incremented when popping/loading data.

When S Bit is set (S=1)
If instruction is LDM and R15 is in the list: (Mode Changes)

```
While R15 loaded, additionally: CPSR=SPSR_<current mode>
```

Otherwise: (User bank transfer)

```
Rlist is referring to User Bank Registers, R0-R15 (rather than
register related to the current mode, such like R14_svc etc.)
Base write-back should not be used for User bank transfer.
Caution - When instruction is LDM:
If the following instruction reads from a banked register (eg. R14_svc),
then CPU might still read R14 instead; if necessary insert a dummy NOP.
```

Notes
The base address should be usually word-aligned.
LDM Rn,...,PC on ARMv4 leaves CPSR.T unchanged.
LDR Rn,...,PC on ARMv5 sets CPSR.T to <op> Bit0, (1=Switch to Thumb).

Transfer Order
The lowest Register in Rlist (R0 if its in the list) will be loaded/stored to/from the lowest memory address. Internally, the rlist register are always processed with INCREASING addresses (ie. for DECREASING addressing modes, the CPU does first calculate the lowest address, and does then process rlist with increasing addresses; this detail can be important when accessing memory mapped I/O ports).

Strange Effects on Invalid Rlist's
Empty Rlist: R15 loaded/stored (ARMv4 only), and Rb=Rb+/-40h (ARMv4-v5).
Writeback with Rb included in Rlist: Store OLD base if Rb is FIRST entry in Rlist, otherwise store NEW base (STM/ARMv4), always store OLD base (STM/ARMv5), no writeback (LDM/ARMv4), writeback if Rb is "the ONLY register, or NOT the LAST register" in Rlist (LDM/ARMv5).

ARM Opcodes: Memory: Single Data Swap (SWP)

Opcode Format

```
Bit    Expl.
```

```
  31-28  Condition
  27-23  Must be 00010b for this instruction
         Opcode (fixed)
           SWP{cond}{B} Rd,Rm,[Rn]      ;Rd=[Rn], [Rn]=Rm
  22     B - Byte/Word bit (0=swap 32bit/word, 1=swap 8bit/byte)
  21-20  Must be 00b for this instruction
  19-16  Rn - Base register                    (R0-R14)
  15-12  Rd - Destination Register             (R0-R14)
  11-4   Must be 00001001b for this instruction
  3-0    Rm - Source Register                  (R0-R14)
```

SWP/SWPB supported by ARMv2a and up.
Swap works properly including if Rm and Rn specify the same register.
R15 may not be used for either Rn,Rd,Rm. (Rn=R15 would be MRS opcode).
Upper bits of Rd are zero-expanded when using Byte quantity. For info about byte and word data memory addressing, read LDR and STR opcode description.
Execution Time: 1S+2N+1I. That is, 2N data cycles, 1S code cycle, plus 1I.

ARM Opcodes: Coprocessor Instructions (MRC/MCR, LDC/STC, CDP, MCRR/MRRC)

Coprocessor Register Transfers (MRC, MCR) (with ARM Register read/write)

```
  Bit    Expl.
  31-28  Condition (or 1111b for MRC2/MCR2 opcodes on ARMv5 and up)
  27-24  Must be 1110b for this instruction
  23-21  CP Opc - Coprocessor operation code         (0-7)
  20     ARM-Opcode (0-1)
          0: MCR{cond} Pn,<cpopc>,Rd,Cn,Cm{,<cp>}   ;move from ARM to CoPro
          0: MCR2      Pn,<cpopc>,Rd,Cn,Cm{,<cp>}   ;move from ARM to CoPro
          1: MRC{cond} Pn,<cpopc>,Rd,Cn,Cm{,<cp>}   ;move from CoPro to ARM
          1: MRC2      Pn,<cpopc>,Rd,Cn,Cm{,<cp>}   ;move from CoPro to ARM
  19-16  Cn     - Coprocessor source/dest. Register  (C0-C15)
  15-12  Rd     - ARM source/destination Register    (R0-R15)
  11-8   Pn     - Coprocessor number                 (P0-P15)
  7-5    CP     - Coprocessor information            (0-7)
  4      Reserved, must be one (1) (otherwise CDP opcode)
  3-0    Cm     - Coprocessor operand Register       (C0-C15)
```

MCR/MRC supported by ARMv2 and up, MCR2/MRC2 by ARMv5 and up.
A22i syntax allows to use MOV with Rd specified as first (dest), or last (source) operand. Native MCR/MRC syntax uses Rd as middle operand, <cp> can be ommited if <cp> is zero.
When using MCR with R15: Coprocessor will receive a data value of PC+12.
When using MRC with R15: Bit 31-28 of data are copied to Bit 31-28 of CPSR (ie. N,Z,C,V flags), other data bits are ignored, CPSR Bit 27-0 are not affected, R15 (PC) is not affected.
Execution time: 1S+bI+1C for MCR, 1S+(b+1)I+1C for MRC.
Return: For MRC only: Either R0-R14 modified, or flags affected (see above).
For details refer to original ARM docs. The opcodes irrelevant for GBA/NDS7 because no coprocessor exists (except for a dummy CP14 unit). However, NDS9 includes a working CP15 unit.
ARM CP14 ICEbreaker Debug Communications Channel
ARM CP15 System Control Coprocessor

Coprocessor Data Transfers (LDC, STC) (with Memory read/write)

```
  Bit    Expl.
  31-28  Condition (or 1111b for LDC2/STC2 opcodes on ARMv5 and up)
  27-25  Must be 110b for this instruction
  24     P - Pre/Post (0=post; add offset after transfer, 1=pre; before trans.)
  23     U - Up/Down Bit (0=down; subtract offset from base, 1=up; add to base)
  22     N - Transfer length (0-1, interpretation depends on co-processor)
  21     W - Write-back bit (0=no write-back, 1=write address into base)
```

```
  20     Opcode (0-1)
          0: STC{cond}{L} Pn,Cd,<Address>  ;Store to memory (from coprocessor)
          0: STC2{L}      Pn,Cd,<Address>  ;Store to memory (from coprocessor)
          1: LDC{cond}{L} Pn,Cd,<Address>  ;Read from memory (to coprocessor)
          1: LDC2{L}      Pn,Cd,<Address>  ;Read from memory (to coprocessor)
          whereas {L} indicates long transfer (Bit 22: N=1)
  19-16  Rn     - ARM Base Register              (R0-R15)     (R15=PC+8)
  15-12  Cd     - Coprocessor src/dest Register  (C0-C15)
  11-8   Pn     - Coprocessor number             (P0-P15)
  7-0    Offset - Unsigned Immediate, step 4     (0-1020, in steps of 4)
```

LDC/STC supported by ARMv2 and up, LDC2/STC2 by ARMv5 and up.
Execution time: (n-1)S+2N+bI, n=number of words transferred.
For details refer to original ARM docs, irrelevant in GBA because no coprocessor exists.

Coprocessor Data Operations (CDP) (without Memory or ARM Register operand)

```
  Bit    Expl.
  31-28  Condition (or 1111b for CDP2 opcode on ARMv5 and up)
  27-24  Must be 1110b for this instruction
         ARM-Opcode (fixed)
           CDP{cond} Pn,<cpopc>,Cd,Cn,Cm{,<cp>}
           CDP2      Pn,<cpopc>,Cd,Cn,Cm{,<cp>}
  23-20  CP Opc - Coprocessor operation code       (0-15)
  19-16  Cn     - Coprocessor operand Register     (C0-C15)
  15-12  Cd     - Coprocessor destination Register (C0-C15)
  11-8   Pn     - Coprocessor number               (P0-P15)
  7-5    CP     - Coprocessor information          (0-7)
  4      Reserved, must be zero (otherwise MCR/MRC opcode)
  3-0    Cm     - Coprocessor operand Register     (C0-C15)
```

CDP supported by ARMv2 and up, CDP2 by ARMv5 and up.
Execution time: 1S+bI, b=number of cycles in coprocessor busy-wait loop.
Return: No flags affected, no ARM-registers used/modified.
For details refer to original ARM docs, irrelevant in GBA because no coprocessor exists.

Coprocessor Double-Register Transfer (MCRR, MRRC) - ARMv5TE only

```
  Bit    Expl.
  31-28  Condition
  27-21  Must be 1100010b for this instruction
  20     L - Opcode (Load/Store)
          0: MCRR{cond} Pn,opcode,Rd,Rn,Cm  ;write Rd,Rn to coproc
          1: MRRC{cond} Pn,opcode,Rd,Rn,Cm  ;read Rd,Rn from coproc
  19-16  Rn - Second source/dest register      (R0-R14)
  15-12  Rd - First source/dest register       (R0-R14)
  11-8   Pn     - Coprocessor number           (P0-P15)
  7-4    CP Opc - Coprocessor operation code   (0-15)
  3-0    Cm     - Coprocessor operand Register (C0-C15)
```

Supported by ARMv5TE only, not ARMv5, not ARMv5TExP.