Department of Electrical Engineering

Indian Institute of Technology Bombay

EE-739: Processor Design

ARM7TDMI Instruction Set Reference

Table of Contents

1 Instruction Encoding 1
1.1 ARM7TDMI – ARM Instructions 1
1.2 ARM7TDMI – THUMB Instructions 2
2 Conditional Execution 2
2.1 Condition Field 2
2.2 Condition Codes 2
3 Addressing, Operands and Directives 3
3.1 General Notes 3
3.2 Shifter Operands 3
3.3 Load/Store Register Addressing Modes 6
3.4 Miscellaneous Load/Store Addressing Modes 8
3.5 Memory Allocation and Operand Alignment 9
3.6 Miscellaneous Assembler Directives 10
4 Instruction Descriptions 12
4.1 General Information 12
4.2 ADC – Add with Carry 12
4.3 ADD - Add 13
4.4 AND – Bit-wise AND 13
4.5 B, BL – Branch, Branch and Link 14
4.6 BIC – Bit Clear 15
4.7 BX – Branch and Exchange 15
4.8 CMN – Compare Negative 16
4.9 CMP - Compare 16
4.10 EOR – Bit-wise Exclusive-OR 17
4.11 LDM – Load Multiple 17
4.12 LDR – Load Register 19
4.13 LDRB – Load Register Byte 20
4.14 LDRH – Load Register Halfword 20
4.15 LDRSB – Load Register Signed Byte 21
4.16 LDRSH – Load Register Signed Halfword 21
4.17 MLA – Multiply-Accumulate 22
4.18 MOV – Move 22
4.19 MRS – Move PSR into General-Purpose Register 23
4.20 MSR – Move to Status Register from ARM Register 23
4.21 MUL – Multiply 24
4.22 MVN – Move Negative 25
4.23 ORR – Bit-wise Inclusive-OR 25
4.24 RSB – Reverse Subtract 26
4.25 RSC – Reverse Subtract with Carry 26
4.26 SBC – Subtract with Carry 27
4.27 SMLAL – Signed Multiply-Accumulate Long 27
4.28 SMULL – Signed Multiply Long 28
4.29 STM – Store Multiple 28
4.30 STR – Store Register 30
4.31 STRB – Store Register Byte 30
4.32 STRH – Store Register Halfword 31
4.33 SUB - Subtract 31
4.34 SWI – Software Interrupt 32
4.35 SWP - Swap 32
4.36 SWPB – Swap Byte 33
4.37 TEQ – Test Equivalence 34
4.38 TST - Test 34
4.39 UMLAL – Unsigned Multiply-Accumulate Long 35
4.40 UMULL – Unsigned Multiply Long 35
5 Pseudo-Instructions 36
5.1 ADR – Load Address (short-range) 36
5.2 ADRL – Load Address (medium-range) 36
5.3 ASR – Arithmetic Shift Right 36
5.4 LDR – Load Register 37
5.5 LSL – Logical Shift Left 37
5.6 LSR – Logical Shift Right 37
5.7 NOP – No Operation 38
5.8 POP - Pop 38
5.9 PUSH - Push 38
5.10 ROR – Rotate Right 38
5.11 RRX – Rotate Right with Extend 39

1 Instruction Encoding

1.1 ARM7TDMI – ARM Instructions

The ARM7TDMI uses a fixed-length, 32-bit instruction encoding scheme for all ARM instructions. The basic encoding for all ARM7TDMI instructions is shown below. Individual instruction descriptions and encodings are shown in section 4 of this document.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Multiply (accumulate)	cond				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm			
Multiply (accumulate) long	cond				0	0	0	0	1	U	A	S	Rd_MSW				Rd_LSW				Rn				1	0	0	1	Rm			
Branch and exchange	cond				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn			
Single data swap	cond				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm			
Halfword data transfer, register offset	cond				0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	0	1	1	Rm			
Halfword data transfer, immediate offset	cond				0	0	0	P	U	1	W	L	Rn				Rd				offset				1	0	1	1	offset			
Signed data transfer (byte/halfword)	cond				0	0	0	P	U	B	W	L	Rn				Rd				addr_mode				1	1	H	1	addr_mode			
Data processing and PSR transfer	cond				0	0	I	opcode				S	Rn				Rd				operand2											
Load/store register/unsigned byte	cond				0	1	I	P	U	B	W	L	Rn				Rd				addr_mode											
Undefined	cond				0	1	1																					1				
Block data transfer	cond				1	0	0	P	U	0	W	L	Rn				register list															
Branch	cond				1	0	1	L	offset																							
Coprocessor data transfer	cond				1	1	0	P	U	N	W	L	Rn				CRd				CP#				offset							
Coprocessor data operation	cond				1	1	1	0	CP opcode				CRn				CRd				CP#				CP			0	CRm			
Coprocessor register transfer	cond				1	1	1	0	CP opc			L	CRn				Rd				CP#				CP			1	CRm			
Software interrupt	cond				1	1	1	1	ignored by processor																							

1.2 ARM7TDMI – THUMB Instructions

The ARM7TDMI uses a fixed-length, 16-bit instruction encoding scheme for all Thumb instructions. The Thumb instruction set is a subset of the ARM instruction set, and is intended to permit a higher code density (smaller memory requirement) than the ARM instruction set in many applications. The processor executes in Thumb mode when bit 5 of the CPSR is 1. Exception processing is always done in ARM mode; the processor automatically switches to ARM mode when entering an exception mode. Use of the Thumb instruction set will not be required in ECE 353/315, and so is not covered here.

2 Conditional Execution

2.1 Condition Field

All ARM7TDMI instructions can be executed conditionally, based on a 4-bit condition field in the instruction. The processor tests the state of the condition flags in the CPSR (N, Z, V, C), and if the condition flag state matches the condition, the instruction executes normally. If the condition flag state does not match the condition, the instruction is executed as a NOP (no operation).

2.2 Condition Codes

The condition codes and use are shown below. If the condition is omitted in instructions, the AL (always) condition is used to specify that the instruction should always execute.

Opcode [31:28]	Mnemonic Extension	Meaning	Condition flag state
0000	EQ	Equal	Z==1
0001	NE	Not equal	Z==0
0010	CS/HS	Carry set / unsigned higher or same	C==1
0011	CC/LO	Carry clear / unsigned lower	C==0
0100	MI	Minus / negative	N==1
0101	PL	Plus / positive or zero	N==0
0110	VS	Overflow	V==1
0111	VC	No overflow	V==0
1000	HI	Unsigned higher	(C==1) AND (Z==0)
1001	LS	Unsigned lower or same	(C==0) OR (Z==1)
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	(Z==0) AND (N==V)
1101	LE	Signed less than or equal	(Z==1) OR (N!=V)
1110	AL	Always (unconditional)	Not applicable
1111	(NV)	Never	Obsolete, unpredictable in ARM7TDMI

3 Addressing, Operands and Directives

3.1 General Notes

In general, using R15 (PC) as the destination register is not appropriate for most instructions. Many instructions will have unpredictable behavior if R15 is the destination.

The ARM supports instruction set extensions by reserving certain bit combinations in the operand fields of otherwise valid instructions. The assembler will ensure that these bit combinations are not used, but these must be avoided when hand-coding instructions.

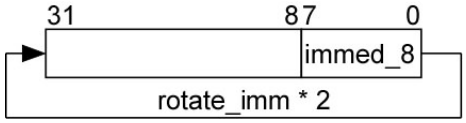
The notation SBZ means “should be zeros”, SBO means “should be ones”.

3.2 Shifter Operands

The shifter operand is represented by the least-significant 12 bits of the instruction. It can take one of eleven forms, as listed below. For illustration, each form has one or more examples based on the Compare instruction (CMP <Rn>, <shifter_operand>). For instructions that use shifter operands, the C flag update is dependent on the form of the operand used.

3.2.1 Immediate Operands

Immediate values are signified by a leading # symbol. The operand is actually stored in the instruction as an 8-bit value with a 4-bit rotation code. The resultant value is the 8-bit value rotated right 0-30 bits (twice the rotation code amount), as illustrated below. Only values that can be represented in this form can be encoded as immediate operands.



The assembler will make substitutions of comparable instructions if it makes it possible to create the desired immediate operand. For example, CMP R0, #-1 is not a legal instruction since it is not possible to specify -1 (0xFFFFFFFF) as an immediate value, but it can be replaced by CMN R0, #1. If the rotate value is non-zero, the C flag is set to bit 31 of the immediate value, otherwise it is unchanged.

Syntax: #<immediate>
Example: CMP R0, #7
Encoding:

11	10	9	8	7	6	5	4	3	2	1	0
rotate_imm				immediate_8							

3.2.2 Register Operands

The register value is used directly. The C flag is unchanged. Note that this is actually a form of the Register Operand, Logical Shift Left by Immediate option (see below) with a 0-bit shift.

Syntax: <Rm>
Example: CMP R0, R1
Encoding:

11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	Rm			

3.2.3 Register Operand, Logical Shift Left by Immediate

The register value is shifted left by an immediate value in the range 0-31. Note that a shift of zero is identical to the encoding for a register operand with no shift. The C flag will be updated with the last value shifted out of Rm unless the shift count is 0.

Syntax: <Rm>, LSL #<immediate>
Example: CMP R0, R1, LSL #7
Encoding:

11	10	9	8	7	6	5	4	3	2	1	0
shift_imm					0	0	0	Rm			

3.2.4 Register Operand, Logical Shift Left by Register

The register value is shifted left by a value contained in a register. The C flag will be updated with the last value shifted out of Rm unless the value in Rs is 0.

Syntax: <Rm>, LSL <Rs>
Example: CMP R0, R1, LSL R2
Encoding:

11	10	9	8	7	6	5	4	3	2	1	0
Rs				0	0	0	1	Rm			

3.2.5 Register Operand, Logical Shift Right by Immediate

The register value is shifted right by an immediate value in the range 1-32. The C flag will be updated with the last value shifted out of Rm.

Syntax: <Rm>, LSR #<immediate>
Example: CMP R0, R1, LSR #7
Encoding:

11	10	9	8	7	6	5	4	3	2	1	0
shift_imm					0	1	0	Rm			

3.2.6 Register Operand, Logical Shift Right by Register

The register value is shifted right by a value contained in a register. The C flag will be updated with the last value shifted out of Rm unless the value in Rs is 0.

Syntax: <Rm>, LSR <Rs>
Example: CMP R0, R1, LSR R2
Encoding:

11	10	9	8	7	6	5	4	3	2	1	0
Rs				0	0	1	1	Rm			

3.2.7 Register Operand, Arithmetic Shift Right by Immediate

The register value is arithmetically shifted right by an immediate value in the range 1-32. The arithmetic shift fills from the left with the sign bit, preserving the sign of the number. The C flag will be updated with the last value shifted out of Rm.

Syntax: <Rm>, ASR #<immediate>
Example: CMP R0, R1, ASR #7
Encoding:

11	10	9	8	7	6	5	4	3	2	1	0
shift_imm					1	0	0	Rm			

3.2.8 Register Operand, Arithmetic Shift Right by Register

The register value is arithmetically shifted right by a value contained in a register. The arithmetic shift fills from the left with the sign bit, preserving the sign of the number. The C flag will be updated with the last value shifted out of Rm unless the value in Rs is 0.

Syntax: <Rm>, ASR <Rs>
Example: CMP R0, R1, ASR R2
Encoding:

11	10	9	8	7	6	5	4	3	2	1	0
Rs				0	1	0	1	Rm			

3.2.9 Register Operand, Rotate Right by Immediate

The register value is rotated right by an immediate value in the range 1-31. [A rotate value of 0 in this instruction encoding will cause an RRX operation to be performed.] The C flag will be updated with the last value shifted out of Rm.

Syntax: <Rm>, ROR #<immediate>
Example: CMP R0, R1, ROR #7
Encoding:

11	10	9	8	7	6	5	4	3	2	1	0
shift_imm					1	1	0	Rm			

3.2.10 Register Operand, Rotate Right by Register

The register value is rotated right by a value contained in a register. The C flag will be updated with the last value shifted out of Rm unless the value in Rs is 0.

Syntax: <Rm>, ROR <Rs>
Example: CMP R0, R1, ROR R2
Encoding:

11	10	9	8	7	6	5	4	3	2	1	0
Rs				0	1	1	1	Rm			

3.2.11 Register Operand, Rotate Right with Extend

The register value is rotated right by one bit through the C flag, i.e. C←Rm[0], Rm[31]←C, Rm[30]←Rm[29], etc.

Syntax: <Rm>, RRX
Example: CMP R0, R1, RRX
Encoding:

11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	Rm			

3.3 Load/Store Register Addressing Modes

There are nine options that can be specified for the addressing mode, as listed below. All options include a base register. See the ARM Architecture Reference Manual section 5.2 for more detailed information.

Note that the ARM processor does not support a direct addressing mode for load/store operation. (Direct addressing is where the address is encoded in the instruction.) However, a label (i.e. a memory location name) can be specified as the memory address in a load/store instruction. In this case, the assembler will attempt to reach the label using the “base register with immediate offset” mode, using the PC as the base register and computing the required offset.

3.3.1 Base register with immediate offset

[Rn, #+/-<offset12>]

memory_address = Rn +/- offset12
Rn is unchanged after instruction

3.3.2 Base register with register offset

[Rn, +/-<Rm>]

memory_address = Rn +/- Rm
Rn is unchanged after instruction

3.3.3 Base register with shifted register offset

[Rn, +/-<Rm>, <shift> #<shift_immediate>]

memory_address = Rn +/- shifted_Rm (see below)

Rn is unchanged after instruction

3.3.4 Base register with immediate offset, pre-indexed

[Rn, #+/-<offset12>]!

memory_address = Rn +/- offset12

Rn = memory_address after instruction

3.3.5 Base register with register offset, pre-indexed

[Rn, +/-<Rm>]!

memory_address = Rn +/- Rm

Rn = memory_address after instruction

3.3.6 Base register with shifted register offset, pre-indexed

[Rn, +/-<Rm>, <shift> #<shift_immediate>]!

memory_address = Rn +/- shifted_Rm (see below)

Rn = memory_address after instruction

3.3.7 Base register with immediate offset, post-indexed

[Rn], #+/-<offset12>

memory_address = Rn

Rn = Rn +/- offset12 after instruction

3.3.8 Base register with register offset, post-indexed

[Rn], +/-<Rm>

memory_address = Rn

Rn = Rn +/- Rm after instruction

3.3.9 Base register with shifted register offset, post-indexed

[Rn], +/-<Rm>, <shift> #<shift_immediate>

memory_address = Rn

Rn = Rn +/- shifted_Rm after instruction (see below)

The *shifted register offset* modes are specified in the same manner as other shifter operands, where the *<shift> #<shift_immediate>* fields can be one of the below.

Logical shift left	- LSL #0-31
Logical shift right	- LSR #1-32
Arithmetic shift right	- ASR #1-32
Rotate right	- ROR #1-32
Rotate right with extend	- RRX

3.4 Miscellaneous Load/Store Addressing Modes

These options apply only to the LDRSB, LDRH, LDRSH, STRH instructions. There are six options that can be specified for the addressing mode, as listed below. All options include a base register. See the ARM Architecture Reference manual section 5.3 for more detailed information.

Note that the ARM processor does not support a direct addressing mode for load/store operation. (Direct addressing is where the address is encoded in the instruction.) However, a label (i.e. a memory location name) can be specified as the memory address. In this case, the assembler will attempt to reach the label using the base register with immediate offset mode, using the PC as the base register and computing the required offset.

3.4.1 Base register with immediate offset

[Rn, #+/-<offset8>]

memory_address = Rn +/- offset8
Rn is unchanged after instruction

3.4.2 Base register with register offset

[Rn, +/-<Rm>]

memory_address = Rn +/- Rm
Rn is unchanged after instruction

3.4.3 Base register with immediate offset, pre-indexed

[Rn, #+/-<offset8>]!

memory_address = Rn +/- offset8
Rn = memory_address after instruction

3.4.4 Base register with register offset, pre-indexed

[Rn, +/-<Rm>]!

memory_address = Rn +/- Rm
Rn = memory_address after instruction

3.4.5 Base register with immediate offset, post-indexed

[Rn], #+/-<offset8>

memory_address = Rn
Rn = Rn +/- offset8 after instruction

3.4.6 Base register with register offset, post-indexed

[Rn], +/-<Rm>

memory_address = Rn
Rn = Rn +/- Rm after instruction

3.5 Memory Allocation and Operand Alignment

This section presents an abridged list of the most commonly used assembler directives. Full documentation is available in the ARM Assembler manuals.

3.5.1 Literals

Literal substitutions are indicated with the EQU directives.
Syntax: name EQU expr {,type}
Description: The EQU directive informs the assembler to perform a literal substitution whenever it encounters *name* in the source file, replacing it with *expr*. The expression can be a numeric constant, a relative address, or an absolute address. If an absolute address is specified, then the type field can be one of ARM, THUMB, CODE16, CODE32, or DATA. (Note that only ARM or DATA are applicable to code written in ECE 353/315.)

3.5.2 Numeric Expressions

Numeric values can be expressed in a number of ways. The most common are;
Decimal – assumed if not otherwise qualified (i.e. 12345)
Hexadecimal – signified by a leading 0x (i.e. 0x12A3) or a leading & (i.e. &12A3)
Arbitrary radix – an arbitrary number base between 2 and 9 can be specified in the form *base_value* (i.e. 2_10110110 represents the binary number 10110110).
Character constants – signified by a character in single quotes (i.e. 'A')

3.5.3 String Expressions

String expressions are indicated by double quotes (i.e. "this is a string!"). To include a double quote or dollar sign in the string, use two of the character in sequence (i.e. "dollar sign = $$". C string escape sequences are also supported (i.e. \n for a newline character).

3.5.4 Memory Allocation Directives

ALIGN – Align
Syntax: ALIGN
Description: ALIGN with no arguments causes that location to be aligned on an instruction (4-byte) boundary.

DCB – Allocate Byte
Syntax: {label} DCB expr {, expr}
Description: DCB allocates bytes of memory, and initializes them to the values given. The *expr* fields can either be numeric constants and/or a quoted string. If DCB is followed by a label that is a branch target, use the ALIGN directive to ensure that the label is properly aligned.

DCD/DCDU – Allocate Words
Syntax: {label} DCD expr {, expr}
Description: DCD allocates words of memory, padding as necessary to ensure word-alignment, and initializes them to the values given. The *expr* fields can either be

numeric constants or program-relative expressions (i.e. labels). DCDU allocates without ensuring alignment. If DCDU is followed by a label that is a branch target, use the ALIGN directive to ensure that the label is properly aligned.

DCW/DCWU – Allocate Halfwords
Syntax: {label} DCW expr {, expr}
Description: DCW allocates halfwords of memory, padding as necessary to ensure halfword-alignment, and initializes them to the values given. The *expr* fields must be numeric constants in the range -32678 to +65535. DCWU allocates without ensuring alignment. If DCW/DCWU is followed by a label that is a branch target, use the ALIGN directive to ensure that the label is properly aligned.

SPACE – Allocate Memory Space
Syntax: {label} SPACE number_of_bytes
Description: SPACE allocates the given number of zero-initialized bytes. If SPACE is followed by a label that is a branch target, use the ALIGN directive to ensure that the label is properly aligned.

3.5.5 Operand Alignment

In general, all memory accesses by the ARM7TDMI must be aligned. For a word (4-byte) access, the target must be aligned on a 4-byte boundary. Halfwords must be aligned on a 2-byte boundary.

3.6 Miscellaneous Assembler Directives

3.6.1 AREA – Area Directive

Syntax: AREA {section_name} {attr} {,attr}...
Description: The AREA directive establishes indivisible memory regions that are manipulated by the linker. Key attribute fields include CODE (area includes only instructions), DATA (area includes only data), READONLY (the default for CODE areas), READWRITE (the default for DATA areas), NOINIT (must only contain uninitialized data or data initialized to 0, the linker settings determine if initialization is actually done.) When linking for the ADuC7026, the default behavior of the linker will place CODE areas into flash memory and DATA areas into SRAM.

3.6.2 ARM – Use ARM Encoding

Syntax: ARM
Description: Informs the assembler to assemble instructions for ARM execution. CODE32 is a synonym for ARM.

3.6.3 END – End of File

Syntax: END
Description: Informs the assembler that the end of the source file has been reached. Every assembly language file must have an END directive.

3.6.4 ENTRY – Code Entry Point

Syntax: ENTRY

Description: The ENTRY directive indicates the point in the code where program execution should begin. There should be only ONE entry point per complete program. Note that in developing the software for an embedded system, execution will begin at the reset vector, so the code entry point will be determined by what code is linked at that address and the ENTRY directive is not used.

3.6.5 EXPORT – Export

Syntax: EXPORT symbol

Description: Instructs the assembler to include the symbol description in the output file so that it can be used by the linker to resolve external references.

3.6.6 EXTERN – External Symbol

Syntax: EXTERN symbol

Description: Informs the assembler that the symbol is defined in another source file. If no reference is made to the symbol, it is not imported into the file.

3.6.7 GLOBAL – Global

Synonym for EXPORT.

3.6.8 INCLUDE – Include File

Syntax: INCLUDE filename

Description: The listed file is read in by the assembler as though it were part of the source file before it proceeds to the next line in the source file. INCLUDEs can be nested.

3.6.9 IMPORT – Import

Syntax: IMPORT symbol

Description: Informs the assembler that the symbol is defined in another source file. The symbol is imported whether it is referenced in the file or not.

3.6.10 KEEP – Keep Local Symbols

Syntax: KEEP {symbol}

Description: Forces the assembler to describe the symbol in its output file, so it will be visible to the debugger. If symbol is blank, all local symbols are preserved. By default, the ARM assembler only preserves symbols in the output file if they are EXPORTed or require relocation.

4 Instruction Descriptions

4.1 General Information

A number of ARM7TDMI instructions will not be used in conjunction with the ADuC7026 processor used in ECE 353 and ECE 315. These instructions are listed below, and do not have detailed information in this section. Further information on these instructions can be obtained from the ARM Architecture Reference Manual.

CDP – Coprocessor Data Processing
LDC – Load Coprocessor
LDRBT – Load Register Byte with Translation
LDRT – Load Register with Translation
MCR - Move to Coprocessor from ARM Register
MRC - Move to ARM Register from Coprocessor
STC - Store Coprocessor
STRBT - Store Register Byte with Translation
STRT - Store Register with Translation

4.2 ADC – Add with Carry

Syntax:
ADC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

RTL:
if(cond)
Rd ← Rn + shifter_operand + C

Flags updated if S used:
N, Z, V, C

Encoding:

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

Usage and Examples:
The ADC instruction is used to implement efficient multiword addition. For example, if 64-bit numbers are stored in R1:0 and R3:2, their sum can be stored in R5:4 as shown below.

```
ADDS  R4, R2, R0        ;add least significant words
ADC   R5, R3, R1        ;add most significant words plus carry
```

4.3 ADD - Add

Syntax:
ADD{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

RTL:
if(cond)
Rd ← Rn + shifter_operand

Flags updated if S used:
N, Z, V, C

Encoding:

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

Usage and Examples:

```
ADDS  R0, R0, #1              ;increments R0, updates flags
ADD   R0, R0, R0, ASR #2      ;multiply R0 by 1.25
```

4.4 AND – Bit-wise AND

Syntax:
AND{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

RTL:
if(cond)
Rd ← Rn AND shifter_operand

Flags updated if S used:
N, Z, C

Encoding:

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

Usage and Examples:

```
AND   R0, R0, #0x8000         ;mask bit D15 of R0
```

4.5 B, BL – Branch, Branch and Link

Syntax:
B{<cond>} <target_address>, BL{<cond>} <target_address>

RTL:
if(cond)
    if(L==1)
        R14 ← address of next instruction
    PC ← PC + (signed_immediate_24 << 2)

Flags updated:
None

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	1	L	signed_immediate_24																							

Usage and Examples:
The B/BL instructions are used to branch to a target address, based on an optional condition. The BL instruction supports subroutine calls by storing the next instruction's address in R14, the link register. (The assembler defines LR as a pseudonym for R14.) Since the offset value is a signed 24-bit value, the branch target must be within approximately +/-32MB.

To return from a subroutine after using BL to call it, the preferred method is to use BX LR. If the subroutine used the STM instruction to store a group of registers and the return address on the stack (i.e. STMFD R13!, {R0-R5, LR}), then the return should be executed in the complementary fashion by placing the saved link register value into the PC (i.e. LDMFD R13!, {R0-R5, PC}).

4.6 BIC – Bit Clear

Syntax:
BIC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

RTL:
if(cond)
Rd ← Rn AND NOT shifter_operand

Flags updated if S used:
N, Z

Encoding:

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

Usage and Examples:
BIC R0, R0, #0x8000 ;clears bit D15 of R0

4.7 BX – Branch and Exchange

Syntax:
BX{<cond>} <Rm>

RTL:
if(cond)
T flag ← Rm[0]
PC ← Rm & 0xFFFFFFFE

Flags updated:
None

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	0	1	0	SB0				SB0				SB0				0	0	0	1	Rm			

Usage and Examples:
The BX instruction is used to branch to a target address stored in a register, based on an optional condition. If bit 0 of the register is set to 1, then the processor will switch to Thumb execution. (Bit 0 is forced to 0 in before the branch address is stored in the PC.) The sample code below shows a call to a Thumb subroutine.

```
ADR   R0, sub     ;get subroutine address
ORR   R0, #1      ;set bit 1
MOV   LR, PC      ;load link register with PC (this address + 8)
BX    R0          ;branch to Thumb subroutine
;subroutine returns here
```

4.8 CMN – Compare Negative

Syntax:
CMN{<cond>} <Rn>, <shifter_operand>

RTL:
if(cond)
Rn + shifter_operand

Flags updated:
N, Z, V, C

Encoding:

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

Usage and Examples:
The CMN instruction performs an addition of the operands (equivalent to a subtraction of the negative), but does not store the result. The flags are always updated.

```
CMP   R0, #1        ;Z=1 if R0=-1, N=1 if R0<-1
```

4.9 CMP - Compare

Syntax:
CMP{<cond>} <Rn>, <shifter_operand>

RTL:
if(cond)
Rn - shifter_operand

Flags updated:
N, Z, V, C

Encoding:

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

Usage and Examples:
The CMP instruction performs a subtraction, but does not store the result. The flags are always updated.

```
CMP   R0, #1        ;Z=1 if R0=1, N=0 if R0>1
```

4.10 EOR – Bit-wise Exclusive-OR

Syntax:
EOR{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

RTL:
if(cond)
Rd ← Rn XOR shifter_operand

Flags updated if S used:
N, Z, C

Encoding:

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

Usage and Examples:

```
EOR    R0, R0, #0x8000          ;toggle bit D15 of R0
```

4.11 LDM – Load Multiple

There are three distinct variants of the LDM instruction. Two of them are for use in conjunction with exception processing, and are not described here. Further information can be obtained in the ARM Architecture Reference Manual.

Syntax:
LDM{<cond>}<addressing_mode>, <Rn>{!}, <registers>

RTL:

```
if(cond)
    start_address ← Rn
    for i = 0 to 14
        if(register_list[i] == 1)

                Ri ← memory[next_address]
    if(register_list[15] == 1)
        PC ← memory[next_address] & 0xFFFFFFFC
    if(writeback)
        Rn ← end_address
```

Flags updated:
None

Encoding:

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

Usage and Examples:
The LDM instruction permits block moves of memory to the registers and enables efficient stack operations. The registers may be listed in any order, but the registers are always loaded in order with the lowest numbered register getting the value form the lowest memory address. If Rn is also listed in the register list and register writeback (W bit) is set, the final value in Rn is unpredictable.
The addressing_mode field determines how *next_address* is calculated (bits P & W), which control how the address is updated in conjunction with each register load. The four addressing_mode values are;

- IA - increment address by 4 after each load (post-increment)
- IB - increment address by 4 before each load (pre-increment)
- DA - decrement address by 4 after each load (post-decrement)
- DB - decrement address by 4 before each load (pre-decrement)

The "!" following Rn controls the value of the writeback bit (bit W), and signifies that Rn should be updated with the ending address at the end of the instruction. If the "!" is not present (W=0), the value of Rn will be unchanged at the end of the instruction.

```
LDMIA     R7, {R0, R2-R4}     ;R0 ← memory[R7]
                              ;R2 ← memory[R7+4]
                              ;R3 ← memory[R7+8]
                              ;R4 ← memory[R7+12]
                              ;R7 is unchanged

LDMDB     R7!, {R0, R2-R4}    ;R0 ← memory[R7-16]
                              ;R2 ← memory[R7-12]
                              ;R3 ← memory[R7-8]
                              ;R4 ← memory[R7-4]
                              ;R7 ← R7 - 16
```

For use in conjunction with stack addressing, four alternative names can be used for the addressing modes. These names are based on the type of stack being used instead of the addressing mode being used. This eliminates confusion in coding stack push and pop operations, since the type of stack will be the same for both the LDM and STM instructions. In ARM syntax, a full stack is one where the stack pointer points to the last used (full) location. An empty stack is one where the stack pointer points to the next available (empty) stack location. As well, a stack can grow through increasing memory addresses (ascending), or downward through decreasing memory addresses (descending). The table below summarizes the stack addressing modes.

- FA (full ascending) - post-decrement (DA) on pop
- FD (full descending) - post-increment (IA) on pop
- EA (empty ascending) - pre-decrement (DB) on pop
- ED (empty descending) - pre-increment on (IB) pop

The instructions below demonstrate a push operation followed by a pop operation assuming an empty-ascending stack. Note that by including the link register (R14) in the push operation, and the PC in the pop operation, a subroutine will return to the caller as part of the context save/restore.

```
STMEA    R13!, {R0, R2-R3, LR}    ;memory[R13] ← R0
                                  ;memory[R13+4] ← R2
                                  ;memory[R13+8] ← R3
                                  ;memory[R13+12] ← R14
                                  ;R13 ← R13 + 16

LDMEA    R13!, {R0, R2-R4, PC}    ;R0 ← memory[R13-16]
                                  ;R2 ← memory[R13-12]
                                  ;R3 ← memory[R13-8]
                                  ;PC ← memory[R13-4]
                                  ;R13 ← R13 - 16
```

4.12 LDR – Load Register

Syntax:
LDR{<cond>} <Rd>, <addressing_mode>

RTL:
if(cond)
    Rd ← memory[memory_address]
    if(writeback)
        Rn ← end_address
Flags updated:
None

Encoding:

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

Usage and Examples:
The LDR instruction reads a word from memory and writes it to the destination register. See the section *Load/Store Register Addressing Modes* for a description of the available addressing modes.

```
LDR      R0, [R1]                 ;R0 = memory[R1]
```

If the memory address is not word-aligned, the value read is rotated right by 8 times the value of bits [1:0] of the memory address. If R15 is specified as the destination, the value is loaded from memory and written to the PC, effecting a branch.

4.13 LDRB – Load Register Byte

Syntax:
LDRB{<cond>} <Rd>, <addressing_mode>

RTL:
if(cond)
    Rd[7:0] ← memory[memory_address], Rd[31:8] ← 0
    if(writeback)
        Rn ← end_address
Flags updated:
None

Encoding:

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

Usage and Examples:
The LDRB instruction reads a byte from memory and zero-extends it into the destination register. See the section *Load/Store Register Addressing Modes* for a description of the available addressing modes.

```
LDRB    R0, [R1]            ;R0 = memory[R1] (zero-extended)
```

4.14 LDRH – Load Register Halfword

Syntax:
LDRH{<cond>} <Rd>, <addressing_mode>

RTL:
if(cond)
    Rd[15:0] ← memory[memory_address], Rd[31:16] ← 0
    if(writeback)
        Rn ← end_address
Flags updated:
None

Encoding:

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

Usage and Examples:
The LDRH instruction reads a halfword from memory, and zero-extends it to 32-bits in the register. See the section *Miscellaneous Load/Store Addressing Modes* for a description of the available addressing modes.

```
LDRH    R0, [R1]            ;R0 = zero-extended memory[R1]
```

4.15 LDRSB – Load Register Signed Byte

Syntax:
LDRSB{<cond>} <Rd>, <addressing_mode>

RTL:
if(cond)
    Rd[7:0] ← memory[memory_address]
    Rd[31:8] ← Rd[7] (sign-extension)
    if(writeback)
        Rn ← end_address
Flags updated:
None

Encoding:

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

Usage and Examples:
The LDRSB instruction reads a byte from memory, and sign-extends it to 32-bits in the register. See the section *Miscellaneous Load/Store Addressing Modes* for a description of the available addressing modes.

```
LDRSB    R0, [R1]          ;R0 = sign-extended memory[R1]
```

4.16 LDRSH – Load Register Signed Halfword

Syntax:
LDRSH{<cond>} <Rd>, <addressing_mode>

RTL:
if(cond)
    Rd[15:0] ← memory[memory_address], Rd[31:16] ← Rd[15] (sign-extension)
    if(writeback)
        Rn ← end_address
Flags updated:
None

Encoding:

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

Usage and Examples:
The LDRSH instruction reads a halfword from memory, and sign-extends it to 32-bits in the register. See the section *Miscellaneous Load/Store Addressing Modes* for a description of the available addressing modes.

```
LDRSH    R0, [R1]          ;R0 = sign-extended memory[R1]
```

4.17 MLA – Multiply-Accumulate

Syntax:
MLA{<cond>}{S} <Rd >, <Rm>, <Rs>, <Rn>

RTL:
if(cond)
Rd ← Rn + (Rs • Rm)

Flags updated if S used:
N, Z (C is unpredictable)

Encoding:

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

Usage and Examples:
MLA performs a 32x32 multiply operation, then stores the sum of Rn and the 32-bit multiplication result to Rd. Since only the least significant 32-bits of the multiplication are used, the result is the same for signed and unsigned numbers.
The instruction below adds the product of R1 and R2 to R0.

```
MLA     R0, R1, R2, R0
```

4.18 MOV – Move

Syntax:
MOV{<cond>}{S} <Rd>, <shifter_operand>

RTL:
if(cond)
Rd ← shifter_operand
if(S==1 and Rd==R15)
CPSR ← SPSR

Flags updated if S used and Rd is not R15 (PC):
N, Z, C

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	I	1	1	0	1	S	SBZ				Rd				shifter operand												

Usage and Examples:
MOV performs a move to a register from another register or an immediate value.

```
MOV   R1, R0, LSL #2          ;R1 <- R0 * 4
MOV   R1, #1                  ;R1 <- 0x0000001
```

If the S bit is set and the destination is R15 (the PC), the SPSR is also copied to CPSR. This form of the instruction used to return from an exception mode.

4.19 MRS – Move PSR into General-Purpose Register

Syntax:
MRS{<cond>} <Rd >, CPSR
MRS{<cond>} <Rd >, SPSR

RTL:
if(cond)
Rd ← CPSR/SPSR

Flags updated:
None

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	R	0	0	SB0				Rd				SB0											

Usage and Examples:
Moves the value of CPSR or the current SPSR into a general-purpose register.

```
MRS         R0, CPSR
```

4.20 MSR – Move to Status Register from ARM Register

Syntax:
MSR{<cond>} CPSR_<fields>, #<immediate>
MSR{<cond>} CPSR_<fields>, <Rm>
MSR{<cond>} SPSR_<fields>, #<immediate>
MSR{<cond>} SPSR_<fields>, <Rm>

RTL:
if(cond)
CPSR/SPSR ← immediate/register value

Flags updated:
N/A

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	R	1	0	field_mask				SB0				rotate_imm				immediate							
cond				0	0	0	1	0	R	1	0	field_mask				SB0				SBZ				0	0	0	0	Rm			

Usage and Examples:
Moves the value of a register or immediate operand into the CPSR or the current SPSR. This instruction is typically used by supervisory mode code. Further details on this instruction can be found in the ARM Architecture Reference Manual.

The <fields> indicate which fields of the CPSR/SPSR be written to should be allowed to be changed. This limits any changes just to the fields intended by the programmer. The allowed fields are;

c sets the control field mask bit (bit 16)
x sets the extension field mask bit (bit 17)
s sets the status field mask bit (bit 18)
f sets the flags field mask bit (bit 19)

One or more fields may be specified.

4.21 MUL – Multiply

Syntax:
MUL{<cond>}{S} <Rd >, <Rm>, <Rs>

RTL:
if(cond)
Rd ← Rs • Rm

Flags updated if S used:
N, Z (C is unpredictable)

Encoding:

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

Usage and Examples:
MUL performs a 32x32 multiply operation, and stores a 32-bit result. Since only the least significant 32-bits are stored, the result is the same for signed and unsigned numbers.
The instruction below stores the product of R1 and R2 to R0.

```
MUL     R0, R1, R2
```

4.22 MVN – Move Negative

Syntax:
MVN{<cond>}{S} <Rd>, <shifter_operand>

RTL:
if(cond)
Rd ← NOT shifter_operand
if(S==1 and Rd==R15)
CPSR ← SPSR

Flags updated if S used and Rd is not R15 (PC):
N, Z, C

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	I	1	1	1	1	S	SBZ				Rd				shifter operand												

Usage and Examples:
MVN complements the value of a register or an immediate value and stores it in the destination register.

```
MVN   R1, R0, LSL #2          ;R1 <- NOT (R0 * 4)
MVN   R1, #1                  ;R1 <- 0xFFFFFFFE
```

If the S bit is set and the destination is R15 (the PC), the SPSR is also copied to CPSR. This form of the instruction used to return from an exception mode.

4.23 ORR – Bit-wise Inclusive-OR

Syntax:
ORR{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

RTL:
if(cond)
Rd ← Rn OR shifter_operand

Flags updated if S used:
N, Z, C

Encoding:

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

Usage and Examples:

```
OR    R0, R0, #0x8000         ;sets bit D15 of R0
```

4.24 RSB – Reverse Subtract

Syntax:
RSB{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

RTL:
if(cond)
Rd ← shifter_operand - Rn

Flags updated if S used:
N, Z, V, C

Encoding:

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

Usage and Examples:

```
RSB    R0, R0, #0              ;negate R0 (2's complement)
RSB    R0, R0, R0, LSL #3      ;multiply R0 by 7
```

Note that the carry flag (C) is the complement of a borrow flag. If a borrow is required by the operation, C will be set to 0.

4.25 RSC – Reverse Subtract with Carry

Syntax:
RSC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

RTL:
if(cond)
Rd ← shifter_operand – Rn – NOT C

Flags updated if S used:
N, Z, V, C

Encoding:

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

Usage and Examples:
If a 64-bit number is stored in R1:0, it can be negated (2's complement) as shown below.

```
RSCS   R0, R0, #0              ;negate least significant word
RSC    R1, R1, #0              ;negate most significant words minus borrow
```

Note that the carry flag (C) is the complement of a borrow flag. If a borrow is required by the operation, C will be set to 0.

4.26 SBC – Subtract with Carry

Syntax:
SBC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

RTL:
if(cond)
Rd ← Rn - shifter_operand – NOT C

Flags updated if S used:
N, Z, V, C

Encoding:

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

Usage and Examples:
The SUB instruction is used to implement efficient multiword subtraction. For example, if 64-bit numbers are stored in R1:0 and R3:2, their difference can be stored in R5:4 as shown below.

```
SUBS  R4, R2, R0          ;subtract least significant words
SUB   R5, R3, R1          ;subtract most significant words minus borrow
```

Note that the carry flag (C) is the complement of a borrow flag. If a borrow is required by the operation, C will be set to 0.

4.27 SMLAL – Signed Multiply-Accumulate Long

Syntax:
SMLAL{<cond>}{S} <Rd_LSW>, <Rd_MSW>, <Rm>, <Rs>

RTL:
if(cond)
Rd_MSW:Rd_LSW ← Rd_MSW:Rd_LSW + (Rs • Rm)

Flags updated if S used:
N, Z (V, C are unpredictable)

Encoding:

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

Usage and Examples:
SMLAL performs a signed 32x32 multiply operation with a 64-bit accumulation. The product of Rm and Rs is added to the 64-bit signed value contained in the register pair Rd_MSW:Rd_LSW. All values are interpreted as 2's-complement.
The instruction below adds the product of R2 and R3 to the 64-bit number stored in R1:0.

```
SMLAL     R0, R1, R2, R3
```

4.28 SMULL – Signed Multiply Long

Syntax:
SMULL{<cond>}{S} <Rd_LSW>, <Rd_MSW>, <Rm>, <Rs>

RTL:
if(cond)
    Rd_MSW:Rd_LSW ← Rs • Rm

Flags updated if S used:
N, Z (V, C are unpredictable)

Encoding:

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

Usage and Examples:
SMULL performs a signed 32x32 multiply operation. The product of Rm and Rs is stored as a 64-bit signed value in the register pair Rd_MSW:Rd_LSW. All values are interpreted as 2's-complement.
The instruction below stores the product of R2 and R3 as a 64-bit number in R1:0.

```
SMULL    R0, R1, R2, R3
```

4.29 STM – Store Multiple

There are two distinct variants of the STM instruction. One of them is for use in conjunction with exception processing, and is not described here. Further information can be obtained in the ARM Architecture Reference Manual.
Syntax:
STM{<cond>}<addressing_mode>, <Rn>{!}, <registers>

RTL:

```
if(cond)
    start_address ← Rn
    for i = 0 to 15
        if(register_list[i] == 1)
            memory[next_address] ← Ri
    if(writeback)
        Rn ← end_address
```

Flags updated:
None

Encoding:

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

Usage and Examples:
The STM instruction permits block moves of registers to memory and enables efficient stack operations. The registers may be listed in any order, but the registers are always stored in order with the lowest numbered register going to the lowest memory address. If Rn is also listed in the register list and register writeback (W bit) is set, the final value stored for Rn can be unpredictable.
The addressing_mode field determines how *next_address* is calculated (bits P & W), which control how the address is updated in conjunction with each register store. The four addressing_mode values are;

- IA - increment address by 4 after each load (post-increment)
- IB - increment address by 4 before each load (pre-increment)
- DA - decrement address by 4 after each load (post-decrement)
- DB - decrement address by 4 before each load (pre-decrement)

The "!" following Rn controls the value of the writeback bit (bit W), and signifies that Rn should be updated with the ending address at the end of the instruction. If the "!" is not present (W=0), the value of Rn will be unchanged at the end of the instruction.

```
STMIA     R7, {R0, R2-R4}      ;memory[R7] ← R0
                               ;memory[R7+4] ← R2
                               ;memory[R7+8] ← R3
                               ;memory[R7+12] ← R4
                               ;R7 is unchanged

STMDB     R7!, {R0, R2-R4}     ;memory[R7-16] ← R0
                               ;memory[R7-12] ← R2
                               ;memory[R7-8] ← R3
                               ;memory[R7-4] ← R4
                               ;R7 ← R7 - 16
```

For use in conjunction with stack addressing, four alternative names can be used for the addressing modes. See the LDM instruction for a detailed discussion and example of usage.

4.30 STR – Store Register

Syntax:
STR{<cond>} <Rd>, <addressing_mode>

RTL:
if(cond)
    memory[memory_address] ← Rd
    if(writeback)
        Rn ← end_address

Flags updated:
None

Encoding:

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

Usage and Examples:
The STR instruction stores a single register to memory. See the section *Load/Store Register Addressing Modes* for a description of the available addressing modes.

```
STR     R0, [R1], -R2, LSL #2     ;memory[R1] = R0, R1 = R1 – (R2 * 4)
STR     R0, [R1, #4]              ;memory[R1+4] = R0, R1 unchanged
```

4.31 STRB – Store Register Byte

Syntax:
STRB{<cond>} <Rd>, <addressing_mode>

RTL:
if(cond)
    memory[memory_address] ← Rd[7:0]
    if(writeback)
        Rn ← end_address
Flags updated:
None

Encoding:

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

Usage and Examples:
The STRB instruction stores the least significant byte of a register to memory. See the section *Load/Store Register Addressing Modes* for a description of the available addressing modes.

```
STRB    R0, [R1, #4]!             ;memory[R1+4] = R0, R1=R1+4
```

4.32 STRH – Store Register Halfword

Syntax:
STRH{<cond>} <Rd>, <addressing_mode>

RTL:
if(cond)
    memory[memory_address] ← Rd[15:0]
    if(writeback)
        Rn ← end_address
Flags updated:
None

Encoding:

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

Usage and Examples:
The STRH instruction stores the least significant halfword (2 bytes) of a register to memory. See the section *Miscellaneous Load/Store Addressing Modes* for a description of the available addressing modes.

```
STRH    R0, [R1], #2        ;memory[R1] = R0, R1 = R1 + 2
STRH    R0, [R1, #-2]       ;memory[R1 – 2]=R0, R1 unchanged
```

4.33 SUB - Subtract

Syntax:
SUB{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

RTL:
if(cond)
    Rd ← Rn - shifter_operand

Flags updated if S used:
N, Z, V, C

Encoding:

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

Usage and Examples:

```
SUBS  R0, R0, #1              ;decrements R0, updates flags
SUB   R0, R0, R0, ASR #2      ;multiply R0 by 0.75
```

Note that the carry flag (C) is the complement of a borrow flag. If a borrow is required by the operation, C will be set to 0.

4.34 SWI – Software Interrupt

Syntax:
SWI{<cond>} <immediate_24>

RTL:
if(cond)
R14_svc ← address of next instruction after SWI instruction
SPSR_svc ← CPSR ; save current CPSR
CPSR[4:0] ← 10011b ; supervisor mode
CPSR[5] ← 0 ; ARM execution
CPSR[7] ← 1 ; disable interrupts
PC ← 0x00000008 ; jump to exception vector

Flags updated:
N/A

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	1	immediate_24																							

Usage and Examples:
The SWI instruction causes a SWI exception. The processor disables interrupts, switches to ARM execution if previously in Thumb, and enters supervisory mode. Execution starts at the SWI exception address.

The 24-bit immediate value is ignored by the instruction, but the value in the instruction can be determined by the exception handler if desired. Parameters can also be passed to the SWI handler in general-purpose registers or memory locations.

4.35 SWP - Swap

Syntax:
SWP{<cond>} <Rd>, <Rm>, [<Rn>]

RTL:
if(cond)
temp ← [Rn]
[Rn] ← Rm
Rd ← temp

Flags updated:
None

Encoding:

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

Usage and Examples:
The SWP instruction exchanges a word between a register and memory. This instruction is intended to support semaphore manipulation by completing the transfers as a single, atomic operation.

```
ADR   R1, semaphore          ;semaphore address
MOV   R0, #1
SWP   R0, R0, [R1]           ;make swap
CMPS  R0, #0                 ;test result
```

4.36 SWPB – Swap Byte

Syntax:
SWPB{<cond>} <Rd>, <Rm>, [<Rn>]

RTL:
if(cond)
temp ← [Rn]
[Rn] ← Rm
Rd ← temp

Flags updated:
None

Encoding:

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

Usage and Examples:
The SWPB instruction exchanges a single byte between a register and memory. This instruction is intended to support semaphore manipulation by completing the transfers as a single, atomic operation.

```
ADR   R1, semaphore          ;semaphore address
MOV   R0, #1
SWPB  R0, R0, [R1]           ;make swap
TST   R0, #0xFF              ;test result
```

4.37 TEQ – Test Equivalence

Syntax:
TEQ{<cond>} <Rn>, <shifter_operand>

RTL:
if(cond)
Rn XOR shifter_operand

Flags updated:
N, Z, C

Encoding:

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

Usage and Examples:
The TEQ instruction performs a non-destructive bit-wise XOR (the result is not stored). The flags are always updated. The most common use for this instruction is to determine if two operands are equal without affecting the V flag. It can also be use to tell if two values have the same sign, since the N flag will be the logical XOR of the two sign bits.

```
TEQ   R0, #0x8000        ;sets Z = 1 if R0 contains the value 0x00008000
TEQ   R0, R1             ;sets N = 1 if signs are different
```

4.38 TST - Test

Syntax:
TST{<cond>} <Rn>, <shifter_operand>

RTL:
if(cond)
Rn AND shifter_operand

Flags updated:
N, Z, C

Encoding:

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

Usage and Examples:
The TST instruction performs a non-destructive AND (the result is not stored). The flags are always updated. The most common use for this instruction is to determine the value of an individual bit of a register.

```
TST   R0, #0x8000        ;sets Z = the complement of R0[15]
```

4.39 UMLAL – Unsigned Multiply-Accumulate Long

Syntax:
UMLAL{<cond>}{S} <Rd_LSW>, <Rd_MSW>, <Rm>, <Rs>

RTL:
if(cond)
Rd_MSW:Rd_LSW ← Rd_MSW:Rd_LSW + (Rs • Rm)

Flags updated if S used:
N, Z (V, C are unpredictable)

Encoding:

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

Usage and Examples:
UMLAL performs an unsigned 32x32 multiply operation with a 64-bit accumulation. The product of Rm and Rs is added to the 64-bit unsigned value contained in the register pair Rd_MSW:Rd_LSW. All values are interpreted as unsigned binary.
The instruction below adds the product of R2 and R3 to the 64-bit number stored in R1:0.

```
UMLAL    R0, R1, R2, R3
```

4.40 UMULL – Unsigned Multiply Long

Syntax:
UMULL{<cond>}{S} <Rd_LSW>, <Rd_MSW>, <Rm>, <Rs>

RTL:
if(cond)
Rd_MSW:Rd_LSW ← Rs • Rm

Flags updated if S used:
N, Z (V, C are unpredictable)

Encoding:

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

Usage and Examples:
UMULL performs an unsigned 32x32 multiply operation. The product of Rm and Rs is stored as a 64-bit unsigned value in the register pair Rd_MSW:Rd_LSW. All values are interpreted as unsigned binary.
The instruction below stores the product of R2 and R3 as a 64-bit number in R1:0.

```
UMULL    R0, R1, R2, R3
```

5 Pseudo-Instructions

5.1 ADR – Load Address (short-range)

Syntax:
ADR{cond} <Rd>, <label>

Description:
The ADR pseudo-instruction assembles to a single ADD or SUB instruction, normally with the PC as an operand. This produces position-independent code. The assembler will report an error if it cannot create a valid instruction to load the address. If the label is program-relative, it must be in the same assembler area as the ADR instruction. (The ADRL pseudo-instruction can reach a wider address range.)

5.2 ADRL – Load Address (medium-range)

Syntax:
ADRL{cond} <Rd>, <label>

Description:
The ADRL pseudo-instruction will always generate a two-instruction sequence to load the address of the given label into the destination register, giving it a wider target range than the ADR instruction. The code is position-independent. The assembler will report an error if it cannot create a valid instruction sequence. (The LDR pseudo-instruction with a label argument can reach any address.)

5.3 ASR – Arithmetic Shift Right

Syntax:
ASR{cond}{S} <Rd>, <Rm>, <Rs>
ASR{cond}{S} <Rd>, <Rm>, <#shift_count>

Description:
ASR is a synonym for the MOV instruction with an ASR-shifted register operand. If an immediate shift count is used, it is limited to the range 1-32. If Rm is not included, the assembler will assume it is the same as Rd.

ASR R0, R1 is equivalent to MOV R0, R0, ASR R1
ASR R0, R1, R2 is equivalent to MOV R0, R1, ASR R2

5.4 LDR – Load Register

Syntax:
LDR{cond} <Rd>, =<expression>
LDR{cond} <Rd>, =<label-expression>

Description:
The LDR pseudo-instruction will generate an instruction to load the destination register with the desired value.
The <expression> field must evaluate to a numeric constant. If the constant is an allowable immediate expression (or the complement of one), a MOV or MVN instruction will be generated. If it is not, the assembler will place the value in a memory location, and generate a PC-relative load instruction to load it from that memory location.
If a label is specified, the assembler will generate a local memory location to store the label address, and include the appropriate linker directives so that the correct address will be in that location after linking.

5.5 LSL – Logical Shift Left

Syntax:
LSL{cond}{S} <Rd>, <Rm>, <Rs>
LSL{cond}{S} <Rd>, <Rm>, <#shift_count>

Description:
LSL is a synonym for the MOV instruction with an LSL shifter operand. If an immediate shift count is used, it is limited to the range 0-31. If Rm is not included, the assembler will assume it is the same as Rd.

LSL R0, R1 is equivalent to MOV R0, R0, LSL R1
LSL R0, R1, R2 is equivalent to MOV R0, R1, LSL R2

5.6 LSR – Logical Shift Right

Syntax:
LSR{cond}{S} <Rd>, <Rm>, <Rs>
LSR{cond}{S} <Rd>, <Rm>, <#shift_count>

Description:
LSR is a synonym for the MOV instruction with an LSR shifter operand. If an immediate shift count is used, it is limited to the range 1-32. If Rm is not included, the assembler will assume it is the same as Rd.

LSR R0, R1 is equivalent to MOV R0, R0, LSR R1
LSR R0, R1, R2 is equivalent to MOV R0, R1, LSR R2

5.7 NOP – No Operation

Syntax:
NOP

Description:
There are numerous ways to encode a NOP (no operation) instruction for the ARM7TDMI processor, such as adding 0 to a register, ORing a register with 0, branching to the next instruction, etc. The actual encoding of the NOP is assembler-dependent.

5.8 POP - Pop

Syntax:
POP{cond} reg_list

Description:
POP is a pseudonym for the LDMIA instruction, with R13! specified for the base register (Rn). The PUSH/POP instructions assume a full-descending (FD) stack organization.

5.9 PUSH - Push

Syntax:
PUSH{cond} reg_list

Description:
PUSH is a pseudonym for the STMDB instruction, with R13! specified for the base register (Rn). The PUSH/POP instructions assume a full-descending (FD) stack organization.

5.10 ROR – Rotate Right

Syntax:
ROR{cond}{S} <Rd>, <Rm>, <Rs>
ROR{cond}{S} <Rd>, <Rm>, <#shift_count>
Description:
ROR is a synonym for the MOV instruction with an ROR shifter operand. If an immediate shift count is used, it is limited to the range 1-31. If Rm is not included, the assembler will assume it is the same as Rd.

ROR R0, R1 is equivalent to MOV R0, R0, ROR R1
ROR R0, R1, R2 is equivalent to MOV R0, R1, ROR R2

5.11 RRX – Rotate Right with Extend

Syntax:

RRX{cond}{S} <Rd>, <Rm>

Description:

RRX is a synonym for the MOV instruction with an RRX shifter operand. If Rm is not included, the assembler will assume it is the same as Rd.

RRX R0 is equivalent to MOV R0, R0, RRX
RRX R0, R1 is equivalent to MOV R0, R1, RRX