4 Instruction Set Encoding

Table 5 Overall instruction set encoding table.

	Type	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
Data Processing	REGOP	Cond				0	0	**I**	Opcode				**S**	Rn				Rd				shifter_operand											
Multiply	MULT	Cond				0	0	0	0	0	0	**A**	**S**	Rd				Rn				Rs				1	0	0	1	Rm			
Single Data Swap	SWAP	Cond				0	0	0	1	0	**B**	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm			
Single Data Transfer	TRANS	Cond				0	1	**I**	**P**	**U**	**B**	**W**	**L**	Rn				Rd				Offset											
Block Data Transfer	MTRANS	Cond				1	0	0	**P**	**U**	**S**	**W**	**L**	Rn				Register List															
Branch	BRANCH	Cond				1	0	1	**L**	Offset																							
Coprocessor Data Transfer	CODTRANS	Cond				1	1	0	**P**	**U**	**N**	**W**	**L**	Rn				CRd				CP#				Offset							
Coprocessor Data Operation	COREGOP	Cond				1	1	1	0	CP Opcode				CRn				CRd				CP#				CP			0	CRm			
Coprocessor Register Transfer	CORTRANS	Cond				1	1	1	0	CP Opcode			**L**	CRn				Rd				CP#				CP			1	CRm			
Software Interrupt	SWI	Cond				1	1	1	1	Ignored by processor																							
		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

Where

I_{25} = Immediate form of shifter_operand

L_{24} = Link; Save PC to LR

U_{23} = 1; address = Rn + offset_12

= 0; address = Rn - offset_12

B_{22} = Byte (0 = word)

A_{21} = Accumulate

L_{20} = Load (0 = store)

S_{20} = Update Condition flags

P_{24}, W_{21} : Select different modes of operation

4.1 Condition Encoding

All instructions include a 4-bit condition execution code. The instruction is only executed if the condition specified in the instruction agrees with the current value of the status flags.

Table 6 Cond: Condition Encoding

Condition	Mnemonic extension	Meaning	Condition flag state
4'h0	eq	Equal	Z set
4'h1	ne	Not equal	Z clear
4'h2	cs / hs	Carry set / unsigned higher or same	C set
4'h3	cc / lo	Carry clear / unsigned lower	C clear

Condition	Mnemonic extension	Meaning	Condition flag state
4'h4	mi	Minus / negative	N set
4'h5	pl	Plus / positive or zero	N clear
4'h6	vs	Overflow	V set
4'h7	vc	No overflow	V clear
4'h8	hi	Unsigned higher	C set and Z clear
4'h9	ls	Unsigned lower or same	C clear or Z set
4'h10	ge	Signed greater than or equal	N == V
4'h11	lt	Signed less than	N != V
4'h12	gt	Signed greater than	Z == 0,N == V
4'h13	le	Signed less than or equal	Z == 1 or N != V
4'h14	al	Always (unconditional)	-
4'h15	-	Invalid condition	-

4.2 Opcode Encoding

Table 7 REGOP: Opcode Encoding

Opcode	Mnemonic extension	Operation	Action	Flags affected
4'h0	**and**	Logical AND	Rd := Rn AND shifter_operand	N, Z, C
4'h1	**eor**	Logical XOR	Rd := Rn XOR shifter_operand	N, Z, C
4'h2	**sub**	Subtract	Rd := Rn - shifter_operand	N, Z, C, V
4'h3	**rsb**	Reverse subtract	Rd := shifter_operand - Rn	N, Z, C, V
4'h4	**add**	Add	Rd := Rn + shifter_operand	N, Z, C, V
4'h5	**adc**	Add with carry	Rd := Rn + shifter_operand + Carry Flag	N, Z, C, V
4'h6	**sbc**	Subtract with carry	Rd := Rn - shifter_operand - NOT(Carry Flag)	N, Z, C, V
4'h7	**rsc**	Reverse subtract with carry	Rd := shifter_operand - Rn - NOT(Carry Flag)	N, Z, C, V
4'h8	**tst**	Test	Update flags after Rn AND shifter_operand S bit always set	N, Z, C
4'h9	**teq**	Test equivalence	Update flags after Rn EOR shifter_operand S bit always set	N, Z, C
4'ha	**cmp**	Compare	Update flags after Rn – shifter_operand S bit always set	N, Z, C, V
4'hb	**cmn**	Compare negated	Update flags after Rn + shifter_operand S bit always set	N, Z, C, V
4'hc	**orr**	Logical (inclusive) OR	Rd := Rn OR shifter_operand	N, Z, C
4'hd	**mov**	Move	Rd := shifter_operand (no first operand)	N, Z, C
4'he	**bic**	Bit clear	Rd := Rn AND NOT(shifter_operand)	N, Z, C
4'hf	**mvn**	Move NOT	Rd := NOT shifter_operand (no first operand)	N, Z, C

4.3 Shifter Operand Encoding

This section describes the encoding of the shifter operand for register instructions.

Table 8 REGOP: Shifter Operand Encoding

Format	Syntax	25 'I'	11	10	9	8	7	6	5	4	3	2	1	0
32-bit immediate	#<immediate>	1	encode_imm				imm_8							
Immediate shifts	<Rm>	0	5'h0					2'h0		0	Rm			
	<Rm>, lsl #<shift_imm>	0	shift_imm					Shift		0	Rm			
	<Rm>, lsr #<shift_imm>													
	<Rm>, asr #<shift_imm>													
	<Rm>, ror #<shift_imm>													
	<Rm>, rrx	0	5'h0					2'b11		0	Rm			
Register Shifts	<Rm>, lsl <Rs>	0	Rs				0	Shift		1	Rm			
	<Rm>, lsr <Rs>													
	<Rm>, asr <Rs>													
	<Rm>, ror <Rs>													

4.3.1 Encode immediate value

Table 9 REGOP: Encode Immediate Value Encoding

Value	32-bit immediate value
4'h0	{ 24'h0, imm_8[7:0] }
4'h1	{ imm_8[1:0], 24'h0, imm_8[7:2] }
4'h2	{ imm_8[3:0], 24'h0, imm_8[7:4] }
4'h3	{ imm_8[5:0], 24'h0, imm_8[7:6] }
4'h4	{ imm_8[7:0], 24'h0 }
4'h5	{ 2'h0, imm_8[7:0], 22'h0 }
4'h6	{ 4'h0, imm_8[7:0], 20'h0 }
4'h7	{ 6'h0, imm_8[7:0], 18'h0 }
4'h8	{ 8'h0, imm_8[7:0], 16'h0 }
4'h9	{ 10'h0, imm_8[7:0], 14'h0 }
4'h10	{ 12'h0, imm_8[7:0], 12'h0 }
4'h11	{ 14'h0, imm_8[7:0], 10'h0 }
4'h12	{ 16'h0, imm_8[7:0], 8'h0 }
4'h13	{ 18'h0, imm_8[7:0], 6'h0 }
4'h14	{ 20'h0, imm_8[7:0], 4'h0 }
4'h15	{ 22'h0, imm_8[7:0], 2'h0 }

4.4 Register transfer offset encoding

Table 10 TRANS: Offset Encoding

Category	Type	Syntax	25 'I'	24 'P'	23 'U'	22 'B'	21 'W'	20 'L'	11	10	9	8	7	6	5	4	3	2	1	0
Immediate offset / index	Immediate offset	[<Rn>, #+/-<offset_12>]	0	1	-	-	0	-	offset_12											
	Immediate pre-indexed	[<Rn>, #+/-<offset_12>]!	0	1	-	-	1	-	offset_12											
	Immediate post-indexed	[<Rn>], #+/-<offset_12>	0	0	-	-	0	-	offset_12											
	Immediate post-indexed, unprivilaged memory access	[<Rn>], #+/-<offset_12>	0	0	-	-	1	-	offset_12											
Register offset /	Register offset	[<Rn>, +/-<Rm>]	1	1	-	-	0	-	8'h0								Rm			

Category	Type	Syntax	25 'I'	24 'P'	23 'U'	22 'B'	21 'W'	20 'L'	11	10	9	8	7	6	5	4	3	2	1	0
index	Register pre-indexed	[<Rn>, +/-<Rm>]!	1	1	-	-	1	-	8'h0								Rm			
	Register post-indexed	[<Rn>], +/-<Rm>	1	0	-	-	0	-	8'h0								Rm			
	Register post-indexed, unprivilaged memory access	[<Rn>], +/-<Rm>	1	0	-	-	1	-	8'h0								Rm			
Scaled register offset / index	Scaled register offset	[<Rn>, +/-<Rm>, <shift> #<shift_imm>]	1	1	-	-	0	-	shift_imm					Shift		0	Rm			
	Scaled register pre-indexed	[<Rn>, +/-<Rm>, <shift> #<shift_imm>]!	1	1	-	-	1	-	shift_imm					Shift		0	Rm			
	Scaled register post-indexed	[<Rn>], +/-<Rm>, <shift> #<shift_imm>	1	0	-	-	0	-	shift_imm					Shift		0	Rm			
	Scaled register post-indexed, unprivilaged memory access	[<Rn>], +/-<Rm>, <shift> #<shift_imm>	1	0	-	-	1	-	shift_imm					Shift		0	Rm			

Where;

Pre-indexed: Address adjusted before access

Post-indexed: Address adjusted after access

I_{25}, P_{24} and W_{21} encode the instruction as shown in the table above.

U_{23} = 1; address = Rn + offset_12

= 0; address = Rn – offset_12

B_{22} = 0; data type is 32-bit word

= 1; data type is byte

L_{20} = 1; load

= 0; store

4.5 Shift Encoding

This encoding is used in both register and single data transfer instructions.

Table 11 REGOP, TRANS: Shift Encoding

Condition	Type	Syntax
2'h0	Logical Shift Left	lsl
2'h1	Logical Shift Right	lsr
2'h2	Arithmetic Shift Right (sign extend)	asr
2'h3	Rotate Right with Extent (CO -> bit 31, bit 0 -> CO), if shift amount = 0, else Rotate Right	ror, rrx

4.6 Load & Store Multiple

Table 12 MTRANS: Index options with ldm and stm

Mode	Stack Load Equivalent	Stack Store Equivalent	Instructions	24 'P'	23 'U'	22 'S'	21 'W'	20 'L'
Increment After (ia)	Full Descending (fd)	Empty Ascending (ea)	ldmia, stmia, ldmfd, stmea	0	1	-	-	-
Increment Before (ib)	Empty Descending (ed)	Full Ascending (fa)	lmdib, stmib, ldmed, stmfa	1	1	-	-	-
Decrement After (da)	Full Ascending (fa)	Empty Descending (ed)	ldmda, stmda, ldmfa, stmed	0	0	-	-	-

Mode	Stack Load Equivalent	Stack Store Equivalent	Instructions	24 'P'	23 'U'	22 'S'	21 'W'	20 'L'
Decrement Before (db)	Empty Ascending (ea)	Full Descending (fd)	lmddb, stmdb, ldmea, stmfd	1	0	-	-	-

S_{22}

The S bit for ldm that loads the PC, the S bit indicates that the status bits loaded. For ldm instructions that do not load the PC and all stm instructions, the S bit indicates that when the processor is in a privileged mode, the User mode banked registers are transferred instead of the registers of the current mode. Ldm with the S bit set is unpredictable in User mode.

W_{21}

Indicates that the base register is updated after the transfer.

L_{20}

Distinguishes between Load (L==1) and Store (L==0) instructions.

4.7 Branch offset

Branch instructions contain an offset in the lower 24 bits of the instruction. This offset is combined with the current pc value to calculate the branch target, as follows:

1. Shift the 24-bit signed immediate value left two bits to form a 26-bit value.
2. Add this to the pc.

4.8 Booth's Multiplication Algorithm

Booth's algorithm involves repeatedly adding one of two predetermined values A and S to a product P, then performing a rightward arithmetic shift on P. Let m and r be the multiplicand and multiplier, respectively; and let x and y represent the number of bits in m and r.

1. Determine the values of A and S, and the initial value of P. All of these numbers should have a length equal to (x + y + 1).
    1. A: Fill the most significant (leftmost) bits with the value of m. Fill the remaining (y + 1) bits with zeros.
    2. S: Fill the most significant bits with the value of (−m) in two's complement notation. Fill the remaining (y + 1) bits with zeros.
    3. P: Fill the most significant x bits with zeros. To the right of this, append the value of r. Fill the least significant (rightmost) bit with a zero.
2. Examine the two least significant (rightmost) bits of P.
    1. If they are 01, find the value of P + A. Ignore any overflow.
    2. If they are 10, find the value of P + S. Ignore any overflow.

3. If they are 00, do nothing. Use P directly in the next step.

4. If they are 11, do nothing. Use P directly in the next step.

3. Arithmetically shift the value obtained in the 2nd step by a single place to the right. Let P now equal this new value.

4. Repeat steps 2 and 3 until they have been done y times.

5. Drop the least significant (rightmost) bit from P. This is the product of m and r.

Here is the algorithm in C-code form;

```
unsigned int mul ( unsigned int Rm, unsigned int Rs )
{
unsigned int multiply_result_hi, multiply_result_lo, n, booth_bits;

for (n=0;n<33;n++){
    if (n==0) {
        booth_bits          = ((Rs & 1)<<1);
        multiply_result_lo  = Rs;
        if      (booth_bits == 1) { multiply_result_hi = Rm;     }
        else if (booth_bits == 2) { multiply_result_hi = ~Rm + 1;}
        else                      { multiply_result_hi = 0;      }
        }
    else {
        booth_bits         =  multiply_result & 3;
        multiply_result_lo = (multiply_result_lo >>1) | (( multiply_result_hi & 1)<<31);
        multiply_result_hi = (multiply_result_hi >>1) | (multiply_result_hi & 0x80000000);
        if (booth_bits == 1) { multiply_result_hi = multiply_result_hi + Rm;        }
        if (booth_bits == 2) { multiply_result_hi = multiply_result_hi + (~Rm + 1); }
        }
    }

return multiply_result_lo;
}
```