APPENDIX

THUMB Instruction Set Listing A

This appendix lists all of the instructions of the THUMB instruction set. This material is a summary of the information presented in the *ARM Architecture Reference Manual*, which can be obtained by contacting ARM Limited (www.arm.com). The instructions are listed in alphabetical order, with the instruction code and a short description given for each instruction. Instruction parameters are denoted using the ⟨*param*⟩ notation, where ⟨*param*⟩ is to be replaced with an actual operand or data value. Immediate operands are denoted using #immedX, where X is the number of bits occupied by the immediate operand.

A.1 ADC ⟨Rd⟩, ⟨Rm⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  0  0  0  1  0  1|   Rm   |   Rd
```

This is an "Add with Carry" instruction. It adds the data in the register ⟨*Rd*⟩, with the data in the register ⟨*Rm*⟩, and the carry (C) flag bit, and stores the result in ⟨*Rd*⟩. This instruction affects the N, Z, C, and V condition-code flags.

Let us consider an example of the usage of this instruction. Suppose that the 64-bit data in {*R*2, *R*3} (high 32 bits in register *R*2 and low 32 bits in register *R*3) is to be added with the 64-bit data in {*R*6, *R*7} (high 32 bits in register *R*6 and low 32 bits in register *R*7). This can be done by first adding *R*3 with *R*7, which will set the C flag bit if there is a carry-out, and then executing "ADC R2, R6".

A.2 ADD ⟨Rd⟩, ⟨Rn⟩, #immed3

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  0  0  1  1  1  0| immed3 |   Rn   |   Rd
```

This is an "Add" instruction. It simply adds the data in the register ⟨*Rn*⟩, with the 3-bit immediate data *immed*3, and then stores the result in ⟨*Rd*⟩. Note that since it is a simple ADD and not an ADC, it does not add the carry (C) flag bit. It affects the N, Z, C, and V condition-code flags.

ADD_2

A.3 ADD ⟨Rd⟩, immed8

Rd = Rd + Imm8

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  0  1  1  0|   Rd   |        immed8
```

This "Add" instruction adds the 8-bit immediate data *immed*8 with the data in register ⟨*Rd*⟩ and stores the result in ⟨*Rd*⟩. It affects the N, Z, C, and V condition-code flags.

ADD_3

A.4 ADD ⟨Rd⟩, ⟨Rn⟩, ⟨Rm⟩

Rd = Rn + Rm

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  0  0  1  1  0  0|   Rm   |   Rn   |   Rd
```

This "Add" instruction adds the data in register ⟨*Rm*⟩ with the data in register ⟨*Rn*⟩ and stores the result in register ⟨*Rd*⟩. It affects the N, Z, C, and V condition-code flags.

A.5 ADD ⟨Rd⟩, ⟨Rm⟩

Rd = Rd (R0~R15) + Rm (R0~R15)

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  0  1  0  0|H1|H2|   Rm   |   Rd
```

This "Add" instruction adds the data in register ⟨*Rd*⟩, with the data in register ⟨*Rm*⟩, and stores the result in register ⟨*Rd*⟩. The condition codes (flags) are not affected. A special feature of this instruction is that the registers ⟨*Rd*⟩ and ⟨*Rm*⟩ can range from *R*0 to *R*15. In order to accommodate 16 possible registers, the *H*1 bit is used as the most significant bit for the binary encoding of the *Rd* register, while the *H*2 bit is used as the most significant bit for the binary encoding of the *Rm* register.

Note that this instruction is not implemented in our HDL code, since the instruction uses the high registers *R*8 through *R*15, which we did not wish to support.

ADD_5

A.6 ADD ⟨Rd⟩, PC, #immed8 × 4

Rd = PC + Imm8 x4

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   1  0  1  0  0|   Rd   |        immed8
```

This "Add" instruction adds the data in the Program Counter, (PC), with the immediate value *immed*8 × 4, and stores the result in the register ⟨*Rd*⟩. Also, the two least significant bits in the PC are ANDed with 0 before storing the result in ⟨*Rd*⟩. This instruction is used to create and store a 32-bit word-aligned PC-relative address (the address is a multiple of 4). Note also that the condition codes (flags) are not affected.

ADD_6 Rd = SP + Imm8 × 4

A.7 ADD ⟨Rd⟩, SP, #immed8 × 4

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   1  0  1  0  1|   Rd   |        immed8
```

This "Add" instruction adds the data in the Stack Pointer (SP), with the immediate value *immed*8 × 4, and stores the result in the register ⟨*Rd*⟩. This instruction is used to create and store a 32-bit word-aligned SP-relative address (the address is a multiple of 4). Note also that the condition code flags are not affected.

ADD_7 Rd = SP + Imm7 × 4

A.8 ADD SP, #immed7 × 4

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   1  0  1  1  0  0  0  0  0|       immed7
```

This "Add" instruction adds the data in the Stack Pointer (SP), with the immediate value *immed*7 × 4, and stores the result back into SP. Note that the condition code flags are not affected.

A.9 AND ⟨Rd⟩, ⟨Rm⟩

Rd = Rd & Rm

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  0  0  0  0  0  0|   Rm   |   Rd
```

This is a logical instruction that ANDs the data in register ⟨*Rd*⟩, with the data in register ⟨*Rm*⟩, and stores the result back into ⟨*Rd*⟩. This instruction updates the N and Z flags based on its final result.

ASR_1 Rd = Rm >> Imm5

A.10 ASR ⟨Rd⟩, ⟨Rm⟩, #immed5

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  0  0  1  0|    immed5    |   Rm   |   Rd
```

This is an arithmetic-shift instruction. An arithmetic shift differs from a logical shift in that an arithmetic shift of a data word retains the *sign* of the data word when performing the shift (by copying the sign bit into the *msb* position when shifting right), while a logical shift simply shifts in 0-bits into the empty slots left by the shift. The data in the register ⟨*Rm*⟩ is arithmetic-shifted right by the amount *immed*5, and then stored into the register (*Rd*). The shift amount ranges from 1 to 32, where a shift of 32 bits is denoted by *immed*5 = 00000, and shifts from 1 to 31 bits are denoted by the binary value of *immed*5. This instruction updates the N, Z, and C flags based on the result of the shift. The most recent shifted-out bit is stored as the C flag.

ASR_2

A.11 ASR ⟨Rd⟩, ⟨Rs⟩

Rd = Rd >> Rs

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  0  0  0  1  0  0|   Rs   |   Rd
```

This is an arithmetic-shift-right instruction that shifts register ⟨*Rd*⟩ by the amount indicated in register ⟨*Rs*⟩, where the value of the latter is interpreted as an unsigned binary number. An arithmetic-shift-right operation is a shift-right operation in which the sign bit is copied into the *msb* bit position as the register is shifted right. Thus, a positive value remains positive after the shift, and a negative value remains negative after the shift. This instruction updates the N, Z, and C flags based on the result of the shift. The C flag attains the value of the most recent shifted-out bit.

B_1

A.12 B⟨cond⟩, ⟨target_address⟩

(cond.)?
PC = PC + Imm8 x 2
(signed)

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   1  1  0  1|    cond   |     signed_immed8
```

This is a conditional-branch instruction that causes a branch to a PC-relative address if the condition being checked is true. The address to branch to is computed as $PC \leftarrow PC+$ the 32-bit sign-extended form of *signed_immed*8 $\times$ 2. The n-bit sign-extended form of a k-bit number is that number extended to n bits by copying the sign bit (*msb*) of the original k-bit number into all bit positions to the left of the original sign bit. By using sign extension, a k-bit number can be extended to a larger number of bits while retaining its original 2's complement signed value. The condition to be checked for this conditional-branch instruction uses the condition code (flag) checks shown in Table A.1.

Table A.1: ▶ Condition codes for ARM and THUMB instructions.

Code	Mnemonic	Meaning	Flags Checked
0000	EQ	Equal	$Z = 1$
0001	NE	Not equal	$Z = 0$
0010	CS/HS	Carry set	$C = 1$
0011	CC/LO	Carry clear	$C = 0$
0100	MI	Minus	$N = 1$
0101	PL	Plus	$N = 0$
0110	VS	Overflow	$V = 1$
0111	VC	No overflow	$V = 0$
1000	HI	Unsigned higher	$(C = 1)$ and $(Z = 0)$
1001	LS	Unsigned less or equal	$(C = 0)$ or $(Z = 1)$
1010	GE	Greater or equal	$(N = V)$
1011	LT	Less	$(N \neq V)$
1100	GT	Greater	$(Z = 0)$ and $(N = V)$
1101	LE	Less or equal	$(Z = 1)$ or $(N \neq V)$
1110	AL	Always	None (unconditional)

B_2 A.13 B ⟨target_address⟩

PC = PC + Imm11 x 2 (signed)

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   1  1  1  0  0|          signed_immed11
```

This is an unconditional-branch instruction that causes a branch to a PC-relative address. The address to branch to is computed as $PC \leftarrow PC+$ the 32-bit sign-extended form of *signed_immed*11 $\times$ 2. The n-bit sign-extended form of a k-bit number is that number extended to n bits by copying the sign bit (*msb*) of the original k-bit number into all bit positions to the left of original sign bit. By using sign extension, a k-bit number can be extended to a larger number of bits while retaining its original 2's complement signed value.

Rd = Rd & $\overline{Rm}$

A.14 BIC ⟨Rd⟩, ⟨Rm⟩

BIC

Bits:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	0	0	1	1	1	0	Rm			Rd		

This is a "bit clear" instruction that performs the bit-wise AND of the value in register ⟨*Rd*⟩, with the complement (all bits inverted) of the value in register ⟨*Rm*⟩, and stores the result back into ⟨*Rd*⟩. The N and Z condition flags are affected by this instruction according to the final value of the ⟨*Rd*⟩ register.

A.15 BKPT ⟨immed8⟩

Bits:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Code:	1	0	1	1	1	1	1	0					immed8			

This is a software-breakpoint instruction. The execution of this instruction results in a subroutine call to a breakpoint exception handler. The ⟨*immed*8⟩ value is an immediate value that is ignored by the hardware but can be used by a debugger to store additional information about the breakpoint.

Note that this instruction is not implemented in our HDL code, since it requires additional software and hardware support.

if H=10 LR = PC + offset<<12 (signed)

if H=11, PC = LR + offset<<1

LR = addr. next instr. OR 00...01

if H=00, PC = PC + offset11 x2

A.16 BL ⟨target_address⟩ or BLX ⟨target_address⟩

Bits:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Code:	1	1	1	H						offset11						

These two instructions are subroutine-call instructions. BL is a "Branch with Link" instruction, used to call a THUMB subroutine, while BLX is a "Branch with Link and Exchange" instruction, used to call an ARM subroutine. BL or BLX is used depending on whether the subroutine being called uses THUMB or ARM assembly code. The "H" field is used to select variants of these instructions. To allow for a large offset for the target address, this instruction is translated by the assembler into a set of two consecutive THUMB instructions. The first THUMB instruction uses $H = 10$ (this mode causes *offset*11 to be used as the high part of the branch offset), and the second THUMB instruction uses $H = 01$ or $H = 11$ (this mode causes *offset*11 to be used as the low part of the branch offset).

The operation of this set of instructions varies depending on the H value. If $H = 10$, then the Link Register (LR) is updated as $LR \leftarrow PC + sign\_extend$ (*offset*11 << 12). If $H = 11$, then the Program Counter (PC) is updated as

$PC \leftarrow LR + sign\_extend(offset11 << 1)$ and the Link Register is updated as $LR \leftarrow$ (*address of next instruction*) ORed with 1 (bitwise-OR). If $H = 00$, then the instruction is an unconditional-branch instruction (the encoding is equivalent to B ⟨target_address⟩). The $H = 01$ mode is only supported in T variants of 5 and above and results in the same sequence of operations as the $H = 11$ mode, except that the T (trap) flag is also cleared.

Note that the BLX form of this instruction is not implemented in our HDL code, since only THUMB code, and not ARM code, is supported.

A.17 BLX ⟨Rm⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  0  1  1  1  1|H2|   Rm   |   SBZ
```

This is a "Branch with Link and Exchange" (subroutine call) instruction that is used to branch to a THUMB or ARM subroutine at the address stored in the register ⟨*Rm*⟩ (with bit *H2* used as the high bit of this register number). The T flag is also set to bit 0 of ⟨*Rm*⟩. Thus, the sequence of operations for this instruction are $LR \leftarrow$ (*address of next instruction*) ORed with 1, $T\ flag \leftarrow Rm(0)$, and $PC \leftarrow Rm(31:1) << 1$.

Note that this instruction is not implemented in our HDL code since this instruction accesses ARM code and uses registers $R8$ through $R15$, which are two features that are not supported in our HDL code.

A.18 BX ⟨Rm⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  0  1  1  1  0|H2|   Rm   |   SBZ
```

This is a "Branch with Exchange" instruction that is used to branch between ARM code and THUMB code. The instruction `BX R41` commonly is used as a subroutine return instruction since $R14$ is the Link Register (LR), which typically stores the subroutine return address. As before, bit $H2$ contains the high bit and ⟨*Rm*⟩ contains the low three bits of the number of the register containing the branch address. This instruction first performs $T\ flag \leftarrow Rm(0)$ and then $PC \leftarrow Rm(31:1) << 1$.

Note that this instruction is not implemented in our HDL code, since this instruction accesses ARM code and uses registers $R8$ through $R15$, which are two features that are not supported in our HDL code.

A.19 CMN ⟨Rn⟩, ⟨Rm⟩

$\{Z, N, C, V\} \xleftarrow{\text{set}} Rn + Rm$

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  0  0  1  0  1  1|   Rm   |   Rn
```

This is a "Compare Negative" instruction that compares the value of register ⟨*Rn*⟩ with the negation of the value of register ⟨*Rm*⟩. Compare instructions typically are implemented as subtract instructions that do not store the subtraction result but only set the condition flags based on the subtraction result. Thus, this instruction first performs *Rn* + *Rm* (subtraction of the negative of *Rm* is the same as addition of *Rm*) and then sets the Z, N, C, and V condition-code flags based on the result of the addition.

A.20 CMP ⟨Rn⟩, #immed8

$\{Z, N, C, V\} \xleftarrow{\text{set}} Rn - Imm8$

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  0  1  0  1|   Rn   |        immed8
```

This "Compare" instruction subtracts the immediate value *immed*8 from the value of the register ⟨*Rn*⟩ and then sets the Z, N, C, and V condition-code flags based on the result of the subtraction.

A.21 CMP ⟨Rn⟩, ⟨Rm⟩

$\{Z, N, C, V\} \xleftarrow{\text{set}} Rn - Rm$

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  0  0  1  0  1  0|   Rm   |   Rn
```

This "Compare" instruction subtracts the value in register ⟨*Rm*⟩ from the value in register ⟨*Rn*⟩ and then sets the Z, N, C, and V condition-code flags based on the result of the subtraction.

A.22 CMP ⟨Rn⟩, ⟨Rm⟩

$\left\{\begin{matrix} Z \\ N \\ C \\ V \end{matrix}\right\}$ set $\leftarrow Rn - Rn$
$\{R_0 \sim R_{15})\ (R_0 \sim R_{15})$

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  0  1  0  1|H1|H2|   Rm   |   Rn
```

This "Compare" instruction subtracts the value in register ⟨*Rm*⟩ from the value in register ⟨*Rn*⟩ and then sets the Z, N, C, and V condition-code flags based on the result of the subtraction. The difference with the previous instruction is that registers *R*0 through *R*15 can be used in this instruction. The fields *H*1 and *H*2 store the high bits for registers ⟨*Rn*⟩ and ⟨*Rm*⟩, respectively. At least one of the two registers must be a register in the range *R*8 through *R*15.

Note that this instruction is not supported in our HDL code, since the registers *R*8 through *R*15 are not supported.

A.23 EOR ⟨Rd⟩, ⟨Rm⟩

$Rd = Rd \wedge Rm$

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  0  0  0  0  0  1|   Rm   |   Rd
```

This is an "Exclusive-OR" instruction that forms the exclusive-OR of the contents of ⟨*Rd*⟩ with the contents of ⟨*Rm*⟩ and stores the result back into register ⟨*Rd*⟩. The operation performed is a bit-wise (bit by bit) logical operation. This instruction affects the Z and N condition-code flags.

A.24 LDMIA ⟨Rn⟩!, ⟨registers⟩

$R_i = Mem([Rn])$
$R_{i+1} \quad Mem([Rn]) + 4$
$\vdots$
R_j

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   1  1  0  0  1|   Rn   |     register_list
```

This instruction, which stands for "Load Multiple Increment After," is a multiple load instruction that loads a non-empty set of registers from memory starting at the address specified in ⟨*Rn*⟩. LDMIA, when used with STMIA (store multiple), permits efficient block copies. The parameter ⟨*registers*⟩ is a list of the registers to be loaded, separated by commas and surrounded by braces (e.g., *R*0, *R*5, *R*6). The parameter ⟨*Rn*⟩ contains the address of the memory location where the data is to be loaded from. Consecutive 32-bit data words are stored into the registers listed in ⟨*registers*⟩, starting from the lowest numbered register. ⟨*Rn*⟩ then is updated to point to the next memory location after the last data item loaded. In the instruction

encoding, *register_list* is an 8-bit vector, with bit i $(0 \leq i \leq 7)$ set to 1 if register i is included in ⟨*registers*⟩.

Note that this instruction is not supported in our HDL code, since this instruction requires multiple accesses to memory, which implies variable instruction execution times that result in complications in the design of the instruction pipeline.

A.25 LDR ⟨Rd⟩, ⟨Rn⟩, #immed5 × 4

Rd = Mem{⟨Rn⟩ + Imm5 x4}

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  1  0  1|    immed5    |   Rn   |   Rd
```

This "Load Register" instruction loads the register ⟨*Rd*⟩ with the 32-bit data contained in the memory location addressed by ⟨*Rn*⟩ + (*immed*5 × 4). For aligned memory word access, it is required that the two least significant bits of ⟨*Rn*⟩ be 0.

A.26 LDR ⟨Rd⟩, ⟨Rn⟩, ⟨Rm⟩

Rd = Mem{⟨Rn⟩ + ⟨Rm⟩}

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  1  1  0  0|   Rm   |   Rn   |   Rd
```

This "Load Register" instruction loads the register ⟨*Rd*⟩ with the 32-bit data contained in the memory location addressed by ⟨*Rn*⟩ + ⟨*Rm*⟩. For aligned memory word access, it is required that the two least significant bits of the address thus formed be 0.

A.27 LDR ⟨Rd⟩, PC, #immed8 × 4

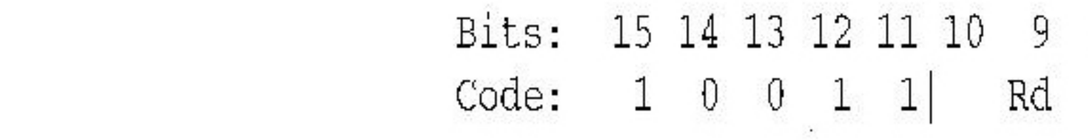
Rd = Mem{PC(31:2)<<2 + Imm8 x4}

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  1|   Rd   |          immed8
```

This "Load Register" instruction loads the register ⟨*Rd*⟩ with the 32-bit data contained in the memory location addressed by $(PC(31:2) << 2) + (immed8 \times 4)$.

Rd = Mem{SP + Imm8×4}

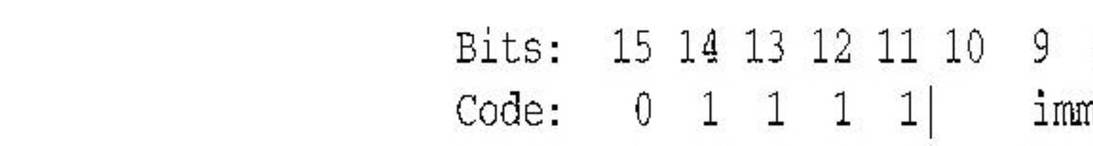
A.28 LDR ⟨Rd⟩, SP, #immed8 × 4



This form of the "Load Register" instruction is useful for accessing stack data. The register $\langle Rd \rangle$ is loaded with the 32-bit data stored at the address $SP + (immed8 \times 4)$. For aligned memory word access, it is required that the two least significant bits of the address thus formed be 0.

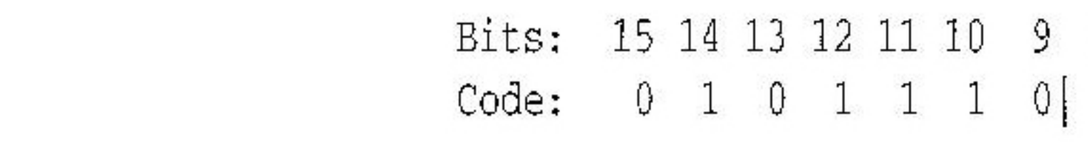
Rd = {0...0, Mem{⟨Rn⟩+Imm5}}
(0...0: 24-b; Mem{⟨Rn⟩+Imm5}: 8-bit)

A.29 LDRB ⟨Rd⟩, ⟨Rn⟩, #immed5



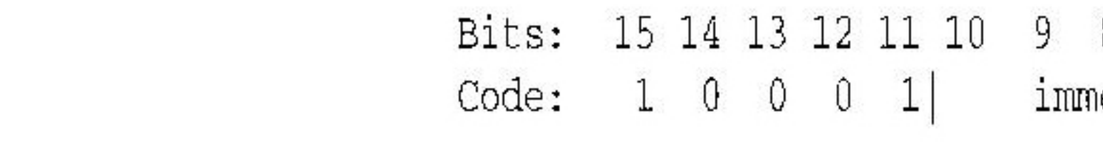
This "Load Register Byte" instruction loads the register $\langle Rd \rangle$ with the data byte at the memory location addressed by $\langle Rn \rangle + immed5$. The data byte retrieved is zero-extended to a 32-bit word (zeroes are inserted into the 24 most significant bit positions) before it is stored into the destination register.

A.30 LDRB ⟨Rd⟩, ⟨Rn⟩, ⟨Rm⟩



This "Load Register Byte" instruction loads the register $\langle Rd \rangle$ with the data byte at the memory location addressed by $\langle Rn \rangle + \langle Rm \rangle$. The data byte retrieved is zero-extended to a 32-bit word (zeroes are inserted into the 24 most significant bit positions) before it is stored into the destination register.

A.31 LDRH ⟨Rd⟩, ⟨Rn⟩, #immed5 × 2



This "Load Register Halfword" instruction loads the register $\langle Rd \rangle$ with the 16-bit data at the memory location addressed by $\langle Rn \rangle + (immed5 \times 2)$. For aligned memory access, the address thus formed must be an even number. The 16-bit data retrieved is zero-extended to a 32-bit word (the most significant 16 bits are filled with zeroes) before it is stored into the destination register.

A.32 LDRH ⟨Rd⟩, ⟨Rn⟩, ⟨Rm⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  1  1  0  1|   Rm   |   Rn   |   Rd
```

This "Load Register Halfword" instruction loads the register ⟨*Rd*⟩ with the 16-bit data at the memory location addressed by ⟨*Rn*⟩ + ⟨*Rm*⟩. For aligned memory access, the address thus formed must be an even number. The 16-bit data retrieved is zero-extended to a 32-bit word (zeroes are inserted into the 16 most significant bit positions) before it is stored into the destination register.

A.33 LDRSB ⟨Rd⟩, ⟨Rn⟩, ⟨Rm⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  1  0  1  1|   Rm   |   Rn   |   Rd
```

This "Load Register Signed Byte" instruction loads the register ⟨*Rd*⟩ with the data byte at the memory location addressed by ⟨*Rn*⟩+⟨*Rm*⟩. The data byte retrieved is sign extended to a 32-bit word (the sign bit, bit 7 of the original data byte, is copied into the 24 most significant bit positions) before it is stored into the destination register.

A.34 LDRSH ⟨Rd⟩, ⟨Rn⟩, ⟨Rm⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  1  1  1  1|   Rm   |   Rn   |   Rd
```

This "Load Register Signed Halfword" instruction loads the register ⟨*Rd*⟩ with the 16-bit data at the memory location addressed by ⟨*Rn*⟩ + ⟨*Rm*⟩. For aligned memory access, the address thus formed must be an even number. The 16-bit data retrieved is sign extended to a 32-bit word (the sign bit, bit 15 of the original data halfword, is copied into the 16 most significant bit positions) before it is stored into the destination register.

A.35 LSL ⟨Rd⟩, ⟨Rm⟩, #immed5

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  0  0  0  0|    immed5    |   Rm   |   Rd
```

This "Logical Shift Left" instruction shifts the value in the register ⟨*Rm*⟩ left by the amount *immed5* and then stores the result in register ⟨*Rd*⟩. Zeroes are shifted into the bit positions vacated by the shift, and the Z, N, and C condition-code flags are updated based on the result of the shift. In particular, the C flag contains the most recent shifted-out bit.

A.36 LSL ⟨Rd⟩, ⟨Rs⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  0  0  0  0  1  0|   Rs   |   Rd
```

This "Logical Shift Left" instruction shifts the value in the register ⟨*Rd*⟩ left by the amount indicated in the least significant byte of ⟨*Rs*⟩. Zeroes are shifted into the bit positions vacated by the shift, and the Z, N, and C condition-code flags are updated based on the result of the shift. In particular, the C flag contains the most recent shifted-out bit.

A.37 LSR ⟨Rd⟩, ⟨Rm⟩, #immed5

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  0  0  0  1|    immed5    |   Rm   |   Rd
```

This "Logical Shift Right" instruction shifts the value in the register ⟨*Rm*⟩ right by the amount indicated in *immed5* and stores the result in register ⟨*Rd*⟩. The value *immed5* = 0 is used to indicate a shift of 32 bits, while all other shift values are equal to the binary value of *immed5*. Zeroes are shifted into the bit positions vacated by the shift, and the Z, N, and C condition-code flags are updated based on the result of the shift. In particular, the C flag contains the most recent shifted-out bit.

A.38 LSR ⟨Rd⟩, ⟨Rs⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  0  0  0  0  1  1|   Rs   |   Rd
```

This "Logical Shift Right" instruction shifts the value in the register ⟨*Rd*⟩ right by the amount indicated in the least significant byte of ⟨*Rs*⟩. Zeroes are shifted into the bit positions vacated by the shift, and the Z, N, and C condition-code flags are updated based on the result of the shift. In particular, the C flag contains the most recent shifted-out bit.

A.39 MOV ⟨Rd⟩, #immed8

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  0  1  0  0|   Rd   |         immed8
```

This "Move" instruction moves the 8-bit immediate value *immed*8 into the register ⟨*Rd*⟩. The Z and N condition-code flags are updated based on the final ⟨*Rd*⟩ value.

A.40 MOV ⟨Rd⟩, ⟨Rn⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  0  0  1  1  1  0  0  0  0|   Rn   |   Rd
```

This "Move" instruction copies the value in register ⟨*Rn*⟩ into the register ⟨*Rd*⟩. The Z and N condition-code flags are updated based on the final ⟨*Rd*⟩ value. Note that the encoding for this instruction is the same as the `ADD Rd, Rn, #0` instruction.

A.41 MOV ⟨Rd⟩, ⟨Rm⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  0  1  1  0|H1|H2|   Rm   |   Rd
```

This form of the "Move" instruction is used for moves from or to high registers (one the registers in the range *R*8 through *R*15). The destination register is indicated by ⟨*Rd*⟩ (least significant three bits) and *H*1 (most significant bit). The source register is indicated by ⟨*Rm*⟩ (least significant three bits) and *H*2 (most significant bit). For proper operation, at least one of *H*1 or *H*2 must be a 1. The condition-code flags are not affected by this instruction.

Note that this instruction is not supported in our HDL code, since the registers *R*8 through *R*15 are not supported.

A.42 MUL ⟨Rd⟩, ⟨Rm⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  0  0  1  1  0  1|   Rm   |   Rd
```

This "Multiply" instruction multiplies the contents of register ⟨*Rm*⟩, with the contents of register ⟨*Rd*⟩, and stores the result in ⟨*Rd*⟩. Although the multiplication of two 32-bit numbers results in a 64-bit number in general, only the least significant 32 bits are stored into ⟨*Rd*⟩ by this instruction. The Z and N condition-code flags are updated according to the final result stored in ⟨*Rd*⟩.

A.43 MVN ⟨Rd⟩, ⟨Rm⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  0  0  1  1  1  1|   Rm   |   Rd
```

This "Move NOT" instruction moves the complement (all bits inverted) of the value of ⟨*Rm*⟩ into the register ⟨*Rd*⟩. The Z and N condition-code flags are updated according to the final result stored in ⟨*Rd*⟩.

A.44 NEG ⟨Rd⟩, ⟨Rm⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  0  0  1  0  0  1|   Rm   |   Rd
```

This "Negate" instruction stores 0 − ⟨*Rm*⟩ into the register ⟨*Rd*⟩. The Z, N, C, and V condition-code flags are updated based on the result of the subtraction.

A.45 ORR ⟨Rd⟩, ⟨Rm⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  0  0  0  1  1  0  0|   Rm   |   Rd
```

This "Logical OR" instruction computes the bit-wise OR of the value in ⟨*Rd*⟩, with the value in ⟨*Rm*⟩, and stores the result in register ⟨*Rd*⟩. The Z and N condition-code flags are updated based on the result of this logical operation.

A.46 POP ⟨registers⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   1  0  1  1  1  1  0| R|    register_list
```

This "Pop Multiple Registers" instruction pops a sequence of 32-bit data items from the stack and stores them into a non-empty set of registers, which may include the registers $R0$ through $R7$ and PC. The registers into which the data items are to be popped are indicated by *register_list*, where a 1 in bit i indicates that register i is to loaded. The R bit is set to 1 if the PC is to be loaded and to 0 otherwise. At least one register must be included in ⟨*registers*⟩. Depending on the bits that are set, the registers are loaded in order starting from $R0$ to $R7$ to PC.

Note that this instruction only is supported in our HDL code as a single POP instruction. The register corresponding to the first bit found, when scanning *register_list* from bit 0 to bit 7, is loaded from the stack and all other registers are left unaffected. This change is made since pops of a variable number of data items require a variable amount of execution time, which complicates the design of the instruction pipeline.

A.47 PUSH ⟨registers⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   1  0  1  1  0  1  0| R|    register_list
```

This "Push Multiple Registers" instruction pushes a non-empty set of registers, which may include the registers $R0$ through $R7$ and LR, onto the stack. The registers to be pushed onto the stack are indicated by *register_list*, where a 1 in bit i indicates that register i is to be pushed. The R bit is set to 1 if LR is to be pushed onto the stack, and 0 otherwise. At least one register must be included in ⟨*registers*⟩. Depending on the bits that are set, the registers are pushed in order starting from LR to $R7$ to $R0$.

Note that this instruction only is supported in our HDL code as a single PUSH instruction. The register corresponding to the first bit found, when scanning *register_list* from bit 0 to bit 7, is pushed onto the stack and all other registers are ignored. This change is made since pushes of a variable number of data items requires a variable amount of execution time, which complicates the design of the instruction pipeline.

A.48 ROR ⟨Rd⟩, ⟨Rs⟩

Bits:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Code:	0	1	0	0	0	0	0	1	1	1	Rs			Rd		

This "Rotate Right" instruction rotates the number in the register ⟨*Rd*⟩ right by the amount indicated in the least significant byte of ⟨*Rs*⟩, and then stores the result in ⟨*Rd*⟩. Since this is a *rotate* instruction, each bit shifted out of the *lsb* position of ⟨*Rd*⟩ is stored back into the *msb* position (which is vacated by the shift operation) of ⟨*Rd*⟩. The Z, N, and C condition-code flags are updated based on the final value stored in ⟨*Rd*⟩. In particular, the C flag attains the value of the most recently shifted-out bit.

A.49 SBC ⟨Rd⟩, ⟨Rm⟩ $Rd = Rd - Rm - \overline{C}$

Bits:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Code:	0	1	0	0	0	0	0	1	1	0	Rm			Rd		

This "Subtract with Carry" instruction stores the value of ⟨*Rd*⟩ − ⟨*Rm*⟩ − (*not C*) into the ⟨*Rd*⟩ register, where *C* refers to the value of the C condition-code flag. This instruction is useful for multiple word subtractions. The carry bit (C) behaves as the complement of the "borrow" after the right halves of two 64-bit numbers are subtracted from one another. The Z, N, C, and V condition-code flags are updated based on the result of the subtraction operation performed for this instruction. Note that the C flag will again be the complement of the "borrow" resulting from this subtraction.

A.50 STMIA ⟨Rn⟩!, ⟨registers⟩

Bits:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Code:	1	1	0	0	0	Rn			register_list							

This "Store Multiple Increment After" instruction stores the contents of a non-empty set of registers into memory starting at the address specified in ⟨*Rn*⟩. Bit *i* of *register_list* is set to 1 if register *i* is to be stored into memory. The registers are stored in sequence from *R*0 to *R*7 (provided that the corresponding bit *i* in *register_list* is 1), and ⟨*Rn*⟩ is then updated to point to the memory address after the location where the contents of the last register are stored.

Note that this instruction is not supported in our HDL code, since this instruction requires multiple accesses to memory, which implies variable instruction execution times that result in complications in the design of the instruction pipeline.

A.51 STR ⟨Rd⟩, ⟨Rn⟩, #immed5 × 4

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  1  0  0|    immed5    |   Rn   |   Rd
```

This "Store" instruction stores the 32-bit data in the register ⟨*Rd*⟩ into memory at the address specified by ⟨*Rn*⟩ + (*immed*5 × 4). For proper aligned memory access, the least significant two bits of the address must be equal to 0.

A.52 STR ⟨Rd⟩, ⟨Rn⟩, ⟨Rm⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  1  0  0  0|   Rm   |   Rn   |   Rd
```

This "Store" instruction stores the 32-bit data in the register ⟨*Rd*⟩ into memory at the address specified by ⟨*Rn*⟩ + ⟨*Rm*⟩. For proper aligned memory access, the least significant two bits of the address must be equal to 0.

A.53 STR ⟨Rd⟩, SP, #immed8 × 4

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   1  0  0  1  0|   Rd   |         immed8
```

This "Store" instruction stores the 32-bit data in the register ⟨*Rd*⟩ into memory at the address specified by *SP* + (*immed*8 × 4). For proper aligned memory access, the least significant two bits of the address must be equal to 0. This instruction is useful for storing items onto the stack.

A.54 STRB ⟨Rd⟩, ⟨Rn⟩, #immed5

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  1  1  0|    immed5    |   Rn   |   Rd
```

This "Store" instruction stores the least significant byte (8 bits) of the data in the register ⟨*Rd*⟩ into memory at the address specified by ⟨*Rn*⟩ + *immed*5.

A.55 STRB ⟨Rd⟩, ⟨Rn⟩, ⟨Rm⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  1  0  1  0|   Rm   |   Rn   |   Rd
```

This "Store" instruction stores the least significant byte (8 bits) of the data in the register ⟨*Rd*⟩ into memory at the address specified by ⟨*Rn*⟩ + ⟨*Rm*⟩.

A.56 STRH ⟨Rd⟩, ⟨Rn⟩, #immed5 × 2

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   1  0  0  0  0|    immed5    |   Rn   |   Rd
```

This "Store" instruction stores the least significant 16 bits of the data in the register ⟨*Rd*⟩ into memory at the address specified by ⟨*Rn*⟩ + (*immed*5 × 2). For proper aligned memory access, the least significant bit of the address must be equal to 0.

A.57 STRH ⟨Rd⟩, ⟨Rn⟩, ⟨Rm⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  1  0  1  0  0  1|   Rm   |   Rn   |   Rd
```

This "Store" instruction stores the least significant 16 bits of the data in the register ⟨*Rd*⟩ into memory at the address specified by ⟨*Rn*⟩ + ⟨*Rm*⟩. For proper aligned memory access, the least significant bit of the address must be equal to 0.

A.58 SUB ⟨Rd⟩, ⟨Rn⟩, #immed3

Rd = Rn − Imm3

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  0  0  1  1  1  1| immed3 |   Rn   |   Rd
```

This "Subtract" instruction computes ⟨*Rn*⟩ − *immed*3 and stores the result in register ⟨*Rd*⟩. The Z, N, C, and V flags are updated based on the results of this subtraction operation.

A.59 SUB ⟨Rd⟩, #immed8

Rd = Rd - Imm8

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  0  1  1  1|   Rd   |        immed8
```

This "Subtract" instruction computes ⟨*Rd*⟩ – *immed*8 and stores the result in register ⟨*Rd*⟩. The Z, N, C, and V flags are updated based on the results of this subtraction operation.

A.60 SUB ⟨Rd⟩, ⟨Rn⟩, ⟨Rm⟩

Rd = Rn - Rm

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   0  0  0  1  1  0  1|   Rm   |   Rn   |   Rd
```

This "Subtract" instruction computes ⟨*Rn*⟩ – ⟨*Rm*⟩ and stores the result in register ⟨*Rd*⟩. The Z, N, C, and V flags are updated based on the results of this subtraction operation.

A.61 SUB SP, #immed7 × 4

SP = SP - Imm7×4

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   1  0  1  1  0  0  0  0  1|        immed7
```

This "Subtract" instruction computes *SP* – (*immed*7 × 4) and stores the result back into the SP register. This instruction is used to allocate memory on the top of the stack. It does not affect the condition-code flags.

A.62 SWI ⟨immed8⟩

```
Bits:  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Code:   1  1  0  1  1  1  1  1|        immed8
```

This is a "Software Interrupt" instruction that is used to call operating system service routines. The parameter ⟨*immed*8⟩ is an 8-bit immediate value that is used by the operating system to determine the service being requested. This parameter simply is ignored by the CPU.

Note that this instruction is not supported, as described previously by our HDL code. The above implementation would require a close interface with an operating system, which is outside the scope of the CPU design being attempted. Instead, this "Software Interrupt" instruction simply is used as a "Halt" instruction, which suspends the operation of the CPU in our HDL code.

A.63 TST ⟨Rn⟩, ⟨Rm⟩

$\{N, Z\} \xleftarrow{\text{set}} Rn \;\&\; Rm$

Bits:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Code:	0	1	0	0	0	0	1	0	0	0	Rm			Rn		

This "Test" instruction forms the bit-wise AND of the contents of register ⟨*Rn*⟩ with the contents of register ⟨*Rm*⟩. The N and Z condition-code flags are then updated based on the result of this bit-wise AND operation. This instruction is used to check if a particular set of bits in a register includes 1 bits. Similar to a "Compare" (CMP) instruction, it does not store its operation result in a destination register; it simply updates the N and Z flags to indicate the result of its test.