Instruction Set Architecture

One of the most crucial early-stage design decisions that were required was the finalisation of an instruction set architecture (ISA) that the CPU's implementation would be based on. Upon researching different ISAs, the decision was made to proceed with an ISA similar to that of ARMv8 [1] that would have two types of encoding for instructions – load/store encoding and 3 operand instruction encoding. This general format (similar to many RISC machines) was chosen as it fit the functional requirement of 16-bit long instruction words the best and provided space for up to 8 registers in the CPU.

In general, all instructions can be classified into one of these categories:

- Loads/stores
- Arithmetic
- Logical
- Shifts
- Jumps
- Stack manipulators
- Others

The CPU can accommodate up to 64 different commands due to the size of the opcode field and therefore can be expanded by adding new instructions and/or hardware when the current design does not meet a specific need or if a high degree of specialisation is required for a given implementation. For instructions other than the memory load/store operations, the encoding can be summarised in terms of bits as:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	Opcode						Rd – destination register			Rs1 – source register 1			Rs2 – source register 2		

Load/store Operations

These instructions are used to manipulate the data RAM in order to retrieve values or save new values into it from one of the CPU's registers. The LDA and STA instructions require 11 bits in the instruction word for the memory address and due to this need, their format is different from all other instructions of this ISA. The most significant bit (MSB) of the instruction word is 1 for both, which distinguishes them from all other commands. The other two available commands follow the standard pattern and start with 0. The instructions available in this category are:

- **LDA (load direct address):** load Rx with the value found at the specified memory address in the data RAM (Rx = Mem[Memory Address])
- **STA (store direct address:** store the value of Rx at the specified memory address in the data RAM (Mem[Memory Address] = Rx)
- **LDR (load indirect address):** load Rd with the value found at the memory address specified by Rs1 in the data RAM (Rd = Mem[Rs1])
- **STR (store indirect address):** store the value of Rs1 at the memory address specified by Rd in the data RAM (Mem[Rd] = Rs1)

Their encoding is summarised below. Rx represents the register on which the load/store operation is performed in the case of LDA and STA. In the case of LDR and STR, Rd is the register to which the value will be read from/stored to (respectively) and Rs1 is the register which contains the memory address.

Instruction	Bits of Instruction Word															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LDA	1	0	Rx			Memory Address										
STA		1														
LDR	0	1	0	1	0	1	0	Rd			Rs1			Unused		
STR						1	1									

Arithmetic Operations

The CPU's Arithmetic Logic Unit (ALU) is able to perform the basic arithmetic operations on 16-bit numbers like addition, subtraction and multiplication (division was not implemented due to its high hardware requirements). The implementation of the MLA command in particular was important as it combines two operations into one command which shortens the operation A = A*B+C from taking 3 clock cycles (multiply then add) to just 2. This instruction was included in the ISA as this specific operation was required to compute one of the given C++ code tests (see Functional Requirements and Appendix … for details). The operations available are:

- **ADD (add):** Rd = Rs1 + Rs2
- **ADC (add with carry):** Rd = Rs1 + Rs2 + CARRY
- **ADO (add 1):** Rd = Rs1 + 1
- **SUB (subtract):** Rd = Rs1 – Rs2
- **SBC (subtract with carry):** Rd = Rs1 – Rs2 + CARRY – 1
- **MUL (multiply):** Rd = Rs1 * Rs2
- **MLA (multiply and add):** Rd = (Rd * Rs1) + Rs2
- **MLS (multiply and subtract):** Rd = Rs2 – (Rd * Rs1)

The format of each of these instructions is summarised in the following table. Rd represents the destination register while Rs1 and Rs2 represent the source registers 1 and 2 respectively.

Instruction	Bits of Instruction Word															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD	0	0	1	0	1	0	0	Rd			Rs1			Rs2		
ADC						0	1									
ADO						1	0							Unused		
SUB		0	1	1	0	0	0							Rs2		
SBC						0	1									
SBO						1	0							Unused		
MUL		0	1	1	1	0	0							Rs2		
MLA						0	1									
MLS						1	0									

Logical Operations

The CPU is also capable of performing some basic logical operations. Although these are not needed to execute the specified tasks outlined in the three test codes (see Functional Requirements and

Appendix …), they are included in the ISA for the sake of completeness and versatility. The operations that fall under this category are:

- **AND:** Rd = Rs1 AND Rs2
- **OR:** Rd = Rs1 OR Rs2
- **XOR:** Rd = Rs1 XOR Rs2
- **NOT:** Rd = NOT Rs1
- **NAND (NND):** Rd = NOT (Rs1 AND Rs2)
- **NOR:** Rd = NOT (Rs1 OR Rs2)
- **XNOR (XNR):** Rd = NOT (Rs1 XOR Rs2)

The format of each of these instructions is summarised in the following table. Rd represents the destination register while Rs1 and Rs2 represent the source registers 1 and 2 respectively.

<table>
<tr><th rowspan="2">Instruction</th><th colspan="16">Bits of Instruction Word</th></tr>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>AND</td><td rowspan="8">0</td><td rowspan="4">0</td><td rowspan="4">0</td><td rowspan="4">1</td><td rowspan="4">1</td><td>0</td><td>0</td><td colspan="4" rowspan="8">Rd</td><td colspan="4" rowspan="8">Rs1</td><td colspan="3" rowspan="3">Rs2</td></tr>
<tr><td>OR</td><td>0</td><td>1</td></tr>
<tr><td>XOR</td><td>1</td><td>0</td></tr>
<tr><td>NOT</td><td>1</td><td>1</td><td colspan="3">Unused</td></tr>
<tr><td>NND</td><td rowspan="4">0</td><td rowspan="4">1</td><td rowspan="4">0</td><td rowspan="4">0</td><td>0</td><td>0</td><td colspan="3" rowspan="4">Rs2</td></tr>
<tr><td>NOR</td><td>0</td><td>1</td></tr>
<tr><td>XNR</td><td>1</td><td>1</td></tr>
</table>

Shift Operations

The ALU can perform asynchronous shift operations by a specified number of places. These are not directly required for any of the testbench codes (see Functional Requirements and Appendix …), however, they provide a useful addition to the CPU to complement the lack of a division operation (an arithmetic shift right by n places is equivalent to division by 2^n) as well as a 1 cycle alternative to a multiplication by a power of 2 (a logical shift left by n places is equivalent to multiplication by 2^n). For a detailed description of the functionality of the ASR, ROR, and RRC instructions, consult Appendix …. The available operations in this category are:

- **LSL (logical shift left):** Rd = Rs1 shifted left by Rs2 number of places
- **LSR (logical shift right):** Rd = Rs1 shifted right by Rs2 number of places without copying the most significant bit
- **ASR (arithmetic shift right):** Rd = Rs1 shifted right by Rs2 number of places with the sign of Rs1 preserved through sign extension (shifting in the most significant bit of Rs1 for every shift)
- **ROR (shift right loop):** Rd = Rs1 shifted right by Rs2 number of places with the value shifted in being the least significant bit of Rs1 with every shift
- **RRC (shift right loop with carry):** Rd = Rs1 shifted right by Rs2 number of places with the value of the carry flip-flop of the ALU shifted in and the least significant bit of Rs1 saved into the carry flip-flop with every shift

The format of each of these instructions is summarised in the following table. Rd represents the destination register while Rs1 and Rs2 represent the source registers 1 and 2 respectively.

Instruction	Bits of Instruction Word															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LSL	0	1	0	0	0	0	0	Rd			Rs1			Rs2		
LSR						0	1									
ASR						1	0									
ROR		1	0	0	1	0	0									
RRC						0	1									

Jump Operations

The jump operations allow the user to directly manipulate the value of the program counter (PC) register in order to skip certain instruction words or create conditional loops. Both conditional jumps and an unconditional jump are implemented to allow for a high degree of flexibility. The conditional jumps also provide an exhaustive range of comparisons of 2 values. In their case, if the comparison is true, the jump is executed. The available jump instructions are:

- **JMP (unconditional jump):** unconditional jump to instruction with index Rd
- **JC1 (conditional jump 1):** jump to instruction with index Rd if Rs1 < Rs2
- **JC2 (conditional jump 2):** jump to instruction with index Rd if Rs1 > Rs2
- **JC3 (conditional jump 3):** jump to instruction with index Rd if Rs1 = Rs2
- **JC4 (conditional jump 4):** jump to instruction with index Rd if Rs1 = 0
- **JC5 (conditional jump 5):** jump to instruction with index Rd if Rs1 ≥ Rs2
- **JC6 (conditional jump 6):** jump to instruction with index Rd if Rs1 ≤ Rs2
- **JC7 (conditional jump 7):** jump to instruction with index Rd if Rs1 ≠ Rs2
- **JC8 (conditional jump 8):** jump to instruction with index Rd if Rs1 < 0

The format of each of these instructions is summarised in the following table. Rd represents the destination register (in this case, the destination of the jump) while Rs1 and Rs2 represent the source registers 1 and 2 respectively.

Instruction	Bits of Instruction Word															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	0	0	0	0	0	0	0	Rd			Unused					
JC1		0	0	0	1	0	0				Rs1			Rs2		
JC2						0	1									
JC3						1	0									
JC4						1	1							Unused		
JC5		0	0	1	0	0	0							Rs2		
JC6						0	1									
JC7						1	0									
JC8						1	1							Unused		

Stack Operations:

These operations manipulate the stack directly, either by pushing (saving) a new value to it from a specified register or by retrieving the latest value pushed onto the stack and saving it in a specified register. These simple operations allow for primitive subroutines to be constructed in assembly language; however, they require the user to think more intensively about their layout in comparison to the simple-to-use BX and BL instructions of the ARMv8 architecture. The stack is a last-in-first-out buffer, meaning that the value available for retrieval is the latest value pushed to it. For a detailed description of the stack's operation and the reasoning behind its implementation, see LIFO Stack. The operations in this category are:

- **PSH (push):** push the value of a register (Rs1) onto the stack
- **POP (pop):** save the latest value pushed onto the stack in a register (Rd)

The format of these instructions is summarised in the following table. Rd represents the destination register while Rs1 represents the source register 1. These instructions only take one register operand each. Their register fields are different in order to make it clear what functionality they perform (PSH requires a source, POP requires a destination).

Instruction	Bits of Instruction Word															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSH	0	1	0	1	0	0	0	Unused			Rs1			Unused		
POP						0	1	Rd			Unused					

Other Operations:

The operations in this category cannot be classified into any of the categories mentioned previously. Nevertheless, their importance in the correct and efficient operation of the CPU is crucial. The operations in this category include:

- **MOV (move):** Rd = Rs1
- **NOP (no operation):** do nothing for a clock cycle/wait
- **STP (stop):** halts all CPU operations, clears the stack, and signifies the end of the list of instructions/program

The format of these instructions is summarised in the following table. Rd represents the destination register while Rs1 represents the source register 1.

Instruction	Bits of Instruction Word															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOV	0	0	1	0	0	1	1	Rd			Rs1			Unused		
NOP		1	1	1	1	1	0	Unused								
STP						1	1									

Bibliography:

ARM, "ARM Information Center". Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/Cjafgdih.html [Accessed: May 18, 2020].