Kush Parmar
I pledge my honor that I have abided by the stevens honor system

# Ori Architecture Instruction Manual

## Introduction:

Ori architecture is currently an elementary type of CPU. Currently, it is only capable of performing 2 arithmetic operations: add and (instruction and). At the time, it can also load an element from a register, with an immediate number as its offset. Lastly, it can run an empty function that does not do anything called nop. However, the CPU can not store values back to memory, as it is limited to the fact that each instruction only uses 8 bits or 1 byte.

The CPU design has 4 general-purpose registers: x0,x1,x2,x3, an instruction memory, and data memory.

## Instructions:

Ori assembly is similar to other assembly languages, the exact syntax of the instruction is very minimalist. Before discussing the syntax of the Ori assembly language the table below introduces each instruction and what it performs.

*(Table 1)*

| Instruction | Action |
|---|---|
| LDR | Loads a value using operand register 1 as an address, which is then added by an offset. The value found after adding the offset is then placed into a target register. |
| and | Compares the bits of register 1 and register 2, each bit is compared ( 1&1=1 anything else 0), the resulting bits are concatenated and placed into the target register |
| add | Adds register 1 and register 2 and places the total into target register |
| nop | Is an empty instruction that does not write back any value to the registers |

The and, add, and nop instruction all share a similar format when they are written in the Ori assembly language. The only difference LDR has is that it uses an immediate number to calculate an offset. Before writing Ori assembly language some basic facts the user should know. Capitalization and the number of spaces does not matter, however, commas must be in their correct place, and each instruction must be written in one line and separately. Instructions can not use registers above x3, and cannot use a negative number as a register number. The amount of

instructions the CPU can perform is limited by the amount of memory we have, going over the 256 instructions will result in instructions being overlapped, thereby deleting some instructions. Using instructions not available in the CPU will result in an error while compiling the assembly code.

*(Table 2) The general structure of instructions, and values they take.*

| Instruction | Structure of instruction | Immediate/ Register 2 values | Register 1 | Target Register |
|---|---|---|---|---|
| ldr | ldr, immediate, reg1, target reg | Immediate:(-2,1) Register 2: N/A | Register 1: (0,3) | Target Register: (0,3) |
| nop | nop, reg2, reg1, target reg | Immediate: N/A Register 2: (0,3) | Register 1: (0,3) | Target Register: (0,3) |
| add | add, reg2, reg1, target reg | Immediate: N/A Register 2: (0,3) | Register 1: (0,3) | Target Register: (0,3) |
| and | and, reg2, reg1, target reg | Immediate: N/A Register 2: (0,3) | Register 1: (0,3) | Target Register: (0,3) |

When we state registers 1 and 2, it does not mean we are only using registers 1 and 2 and not 0 or 3 in the CPU. We are instead stating that registers 1 and 2 are operands and can use any of the 4 registers.

*(Table 3)Examples of instructions, in words and how they are encoded in Ori Assembly language*

| Instructions in words | Instructions translated |
|---|---|
| Using the address from register 2, load the value from memory at an offset of 1 into register 3 | ldr, 1, x2, x3 |
| Using the values from register 1 and register 0, add them and place it in register 3 | add, x1, x0, x3 |
| Using the values from register 2 and register 3, compare the bits using the and comparator them and place it in register 2 | and, x2, x3, x2 |
| Perform an empty operation which does not write back to a register | nop, x0, x0, x0 |

All these instructions are written inside a file called toTranslate.txt, which is then encoded by the compiler named OriTranslator (which was coded in java) which produces a file named translated or will update the translated file. If someone accidentally deleted the toTranslate.txt file, by running the compiler a new toTranslate.txt file is created. The translate.file can be loaded into the instruction memory for the Ori CPU by right-clicking the ram using the mouse and selecting the load image option. The encoding process turns all these instructions into machine bits, becoming binary. This binary is then translated into hexadecimal bits that can be loaded and interpreted by the CPU.

The Ori Translator mainly uses 2 built-in java libraries, Io and Util. From Java.io we import BufferedReader, BufferedWriter, File, FileReader, FileWriter, IOException, and FileNotFoundException. From Java.util we import Scanner and Arraylist. All the functions are imported from the java.io library and are used in the reading and writing process of converting toTranslate.txt to translated.file. The ArrayList data structure was imported and it is used to hold the instructions, since arrays themselves are not dynamic, it is easier to use an ArrayList to hold our instructions. The Scanner is used so the user can insert the path (which folder) in which you would want the toTranslated.txt and translated.file to be saved.

**Instructions to Cpu**

After understanding what each instruction does, it is important to make clear how the Cpu understands these values.
00000000 - These are 8 bits or 1 byte
[00][00][00][00]- The bit is broken down into two bits so the CPU can understand what task is being performed

*(Table 4) 7th bit most significant bit (or right-most) 0th bit least significant bit.*

| Bits | 7-6 | 5-4 | 3-2 | 1-0 |
|---|---|---|---|---|
| Action of bits | Instruction Alu opcode | Register 2 or immediate number | Register 1 | Target Register |

To restate, the CPU currently only performed nop, add, and LDR operations. And the table below shows the Alu opcodes for each instruction

*(Table 5)*

| Alu opcodes | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| Action of bits | Nop | Add | And | LDR |

The reason why the Alu opcode is distributed like this, instead of creating a separate bit is to determine whether there is an immediate number or not, comes from the limitations of a 2 bit Alu opcode. Instead, we have multiplexors that make a decision based on the first 2-bit number, sending out outputs about whether we are using an immediate number or using register 2.

*(Table 6) The binary representation of the 4 registers*

| Registers | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Binary | 00 | 01 | 10 | 11 |

*(Table 7) The values of immediate numbers, these are signed numbers*

| Registers | -2 | -1 | 0 | 1 |
|---|---|---|---|---|
| Binary | 10 | 11 | 00 | 01 |

When we combine these bits individually, in the table below we can see examples of translated instructions

*(Table 8)*

| Instruction | AluOp | imm/reg 2 | reg1 | Target reg | Binary | Hex digits |
|---|---|---|---|---|---|---|
| ldr, 1, x2, x3 | 11 | 01 | 10 | 11 | 11011011 | DB |
| add, x1, x0, x3 | 01 | 01 | 00 | 11 | 01010011 | 53 |
| and, x2, x3, x2 | 10 | 10 | 11 | 10 | 10101110 | AE |
| nop, x0, x0, x0 | 00 | 00 | 00 | 00 | 00000000 | 00 |

From the table above we see that we concatenate these bits from each part of the instruction to get the 8-bit instruction. The hex digits are a way of representing the binary, which is how the instruction is shown in the instruction memory. The same idea also applies when loading values into the data memory, all the data must be manually translated into hex digits, to be utilized by the CPU. The way we add values manually to the data memory is by using the hand-looking figure on the top and by clicking on each address you can add a value to that address in memory. Just a reminder that hex digits can be represented from 0-9 and a-f.

To understand how the Ori CPU works. We first write our instructions into toTranslate.txt, compile our instructions and load them into the CPU by loading translated.file into the instruction memory. Those corresponding values must then be added to data memory manually. After that, we can start running the program. The CPU will split up all the bits from instruction memory, as shown in Table 4. Each of these bits is then used, in identifying which registers are going to be used and where the output will be stored. The Alu opcode will be used to identify whether we are using an immediate number or not, which operation we will doing, and if we are writing back to memory. These are passed through multiplexors as control signals. If we are loading for memory, then the outputs from instruction add and instruction and are disregarded, only accepting the load instruction since the control signal will indicate the multiplexor to choose the load value. These same principles can be applied to all the other instructions except we would disregard the value from memory since the values for instructions and, and instruction add are found in the registers. After the instruction is processed, on the rising edge of the clock cycle of the next instruction, the value is stored.