LITE-16

A 16-bit Instruction Set Architecture

- Index

1.- Introduction

LITE-16 is a RISC-type, Harvard-based 16-bit architecture, mainly inspired by RISC-V, the open-source Instruction Set Architecture by the University of California, Berkeley. It contains a total of 16 instructions and it works with 16 16-bit registers, along with a 16-bit program and data memory. The aim of LITE-16 is to be a simple and easy-to-use architecture with just enough resources to be a fully-functioning ISA.

The architecture is divided into five segments: The program memory and counter, the Register Fetch Unit (RFU), the Arithmetic Logic Unit (ALU), the data memory, and the Control Unit (CU). Each of the segments will be described in their specific chapter.

Every instruction can be divided into four equal length segments of 4 bits, usually having the same purpose between instructions. These characteristics were intentionally implemented to ensure more simplicity in both the architecture design and in the use of the instructions, and perhaps to potentially extend its bit design with more ease. Every instruction contains an operation code (opcode) segment to identify the instruction to be executed. This segment contains the lowest 4 bits of every instruction. The other three segments left are mainly used to fetch registers but also to define immediate 8-bit values. More on the instructions in their chapter.

The software used to design the digital architecture is Logisim, an open-source, cross-platform graphical tool programmed in Java by Carl Burch. The fully-functioning circuit project can be found in the LITE-16 repository, and it can be simulated with the same software. More on the software and its download can be found in Logisim's official page.

The repository will also include a LITE-16 assembler and simulator programmed in C to build and assemble code that a LITE-16 processor could run.

2.- Instruction Set

As mentioned in the Introduction, the LITE-16 Instruction Set Architecture contains 16 instructions: 15 of them are dedicated to operations inside the processor, and one of them is dedicated to invoke system calls.

The instructions are distributed in a certain way to divide them into groups in function of their opcode. This facilitates the creation of control bits in the Control Unit (CU). The groups are ALU Instructions, jump (JMP) instructions, and miscellaneous (MISC) instructions.

- **ALU Instructions:**

The ALU instructions are the ones that use the ALU to load in a register (rd) an output, using two registers (s1 and s2) as their operands. There are a total of 6 ALU instructions and they can be identified by reading the highest bit of the opcode. If it's 0, it will be an ALU Instruction. The system call instruction also has the last opcode bit unset, but this instruction can be ignored because this project won't implement specific syscalls for the time being.

- **JMP Instructions:**

The JMP Instructions operate with the Program Counter (PC) to invoke jumps if they fulfill the condition specified. These instructions also use the CMP part of the ALU, which determines if s1 is higher, lower or equal than s2. There are three JMP Instructions, one for each CMP flag, and one instruction that invokes a subroutine that jumps unconditionally.

- **MISC Instructions:**

MISC Instructions are instructions that operate with the Data Memory or load immediate values to registers. There are two memory instructions, one for loading and another one for storing. There are also two immediate value instructions, one for loading an 8-bit immediate value to the higher half of the register, and another one to add to the register itself an 8-bit immediate value in the lower half.

In the following page there is a table with specifications for each instruction:

Operation	Function	(3-0)	(7-4)	(11-8)	(15-12)
add	rd <- s1 + s2, pc <- pc + 1	0000	s1	s2	rd
or	rd <- s1 or s2, pc <- pc + 1	0001	s1	s2	rd
xor	rd <- s1 xor s2, pc <- pc + 1	0010	s1	s2	rd
and	rd <- s1 and s2, pc <- pc + 1	0011	s1	s2	rd
sl	rd <- s1 << dist, pc <- pc + 1	0100	s1	dist	rd
srl	rd <- s1 >> dist, pc <- pc + 1	0101	s1	dist	rd
sra	rd <- s1 >>> dist, pc <- pc + 1	0110	s1	dist	rd
syscall	(invokes syscall)	0111	s1	s2	s3
je	pc <- pc + (A==B) ? roff : 1	1000	A	B	roff
jl	pc <- pc + (A<B) ? roff : 1	1001	A	B	roff
jg	pc <- pc + (A>B) ? roff : 1	1010	A	B	roff
fn	pc <- pc + roff, rd <- pc + 1	1011	X	rd	roff
ld	rd <- (addr), pc <- pc + 1	1100	X	addr	rd
st	(addr) <- val, pc <- pc + 1	1101	X	addr	val
mv	rd <- (lo & hi) + rd, pc <- pc + 1	1110	lo	hi	rd
mvu	rd(upp) <- lo & hi, pc <- pc + 1	1111	lo	hi	rd

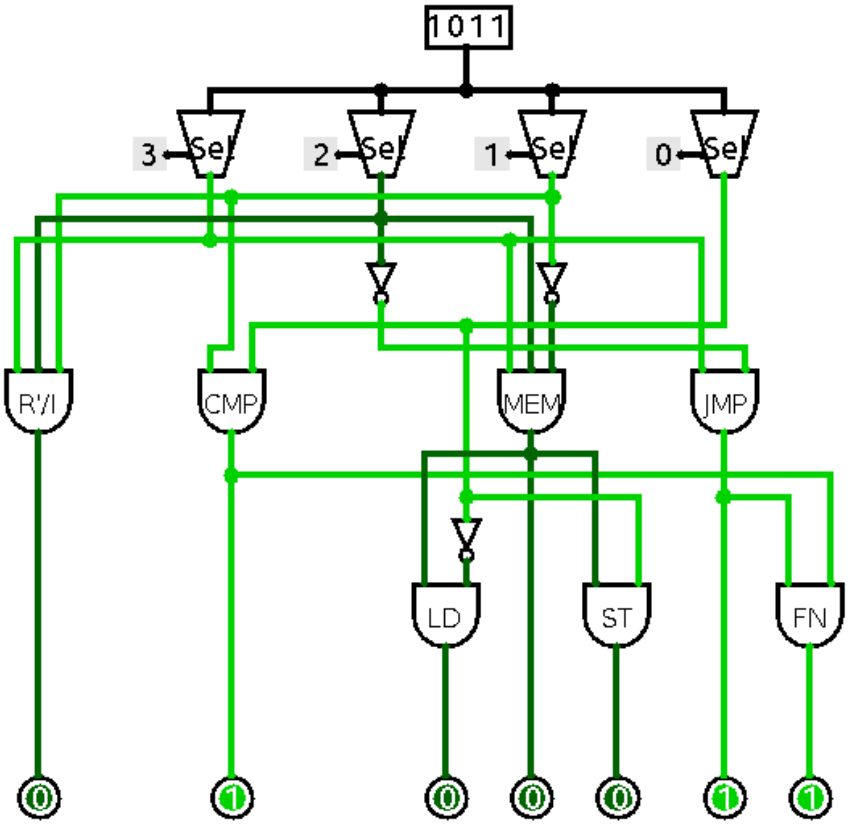
3.- Control Unit

The Control Unit is an essential part to control the actions of the processor. By the use of seven 1-bit signals, the processor knows what and where to do a specific operation. This is achieved by connecting the appropriate signals to some specific multiplexers all around the processor.

The way these signals are set or unset is by checking the codeop bits. The instructions are ordered in a way that facilitates obtaining the signals. For instance, as mentioned in the Instruction Set chapter, if the third bit of the codeop is zero, it means that the instruction is an ALU instruction. If the third bit is set but the second one is not, the instruction is a JMP instruction. Using this method, these are the signals the processor needs:

- R'/I: If 0, the instruction requires two registers as operands. If 1, it requires the immediate values.
- CMP: This signal is used as an operand to check if the instruction is a FN instruction.
- JMP: If 1, the instruction is a JMP instruction.
- FN: If 1, the instruction is a FN instruction.
- MEM: If 1, the instruction requires access to the data memory.
- LD: If 1, the instruction is a LD instruction. This is connected to the "ld" input of the data memory, which enables memory reading.
- ST: If 1, the instruction is a ST instruction. This is connected to the "st" input of the data memory, which enables memory writing.

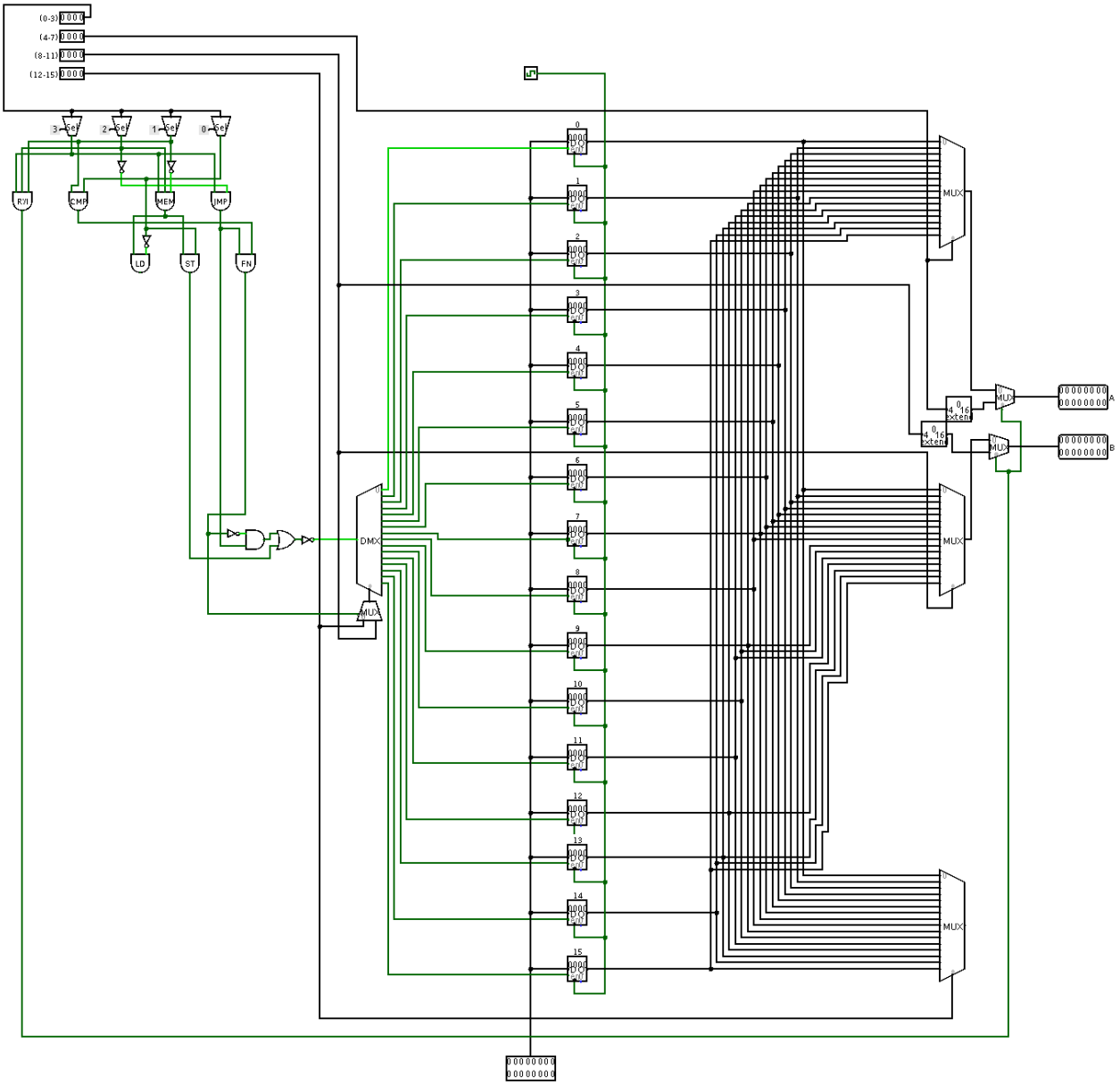
In the following page there is a representation of the circuitry of the Control Unit.



In this example, the codeop is 1011, meaning that it's a FN instruction. In this case, the CMP, JMP and FN signals turn on.

4.- Register Fetch Unit

The RFU is responsible for getting the values of the appropriate registers and routing them to the A and B ALU operands if necessary. In order to doso, the key instruction segments needed are the three last ones. Generally, s1, s2 and rd. Here's the circuitry for this section of the processor.

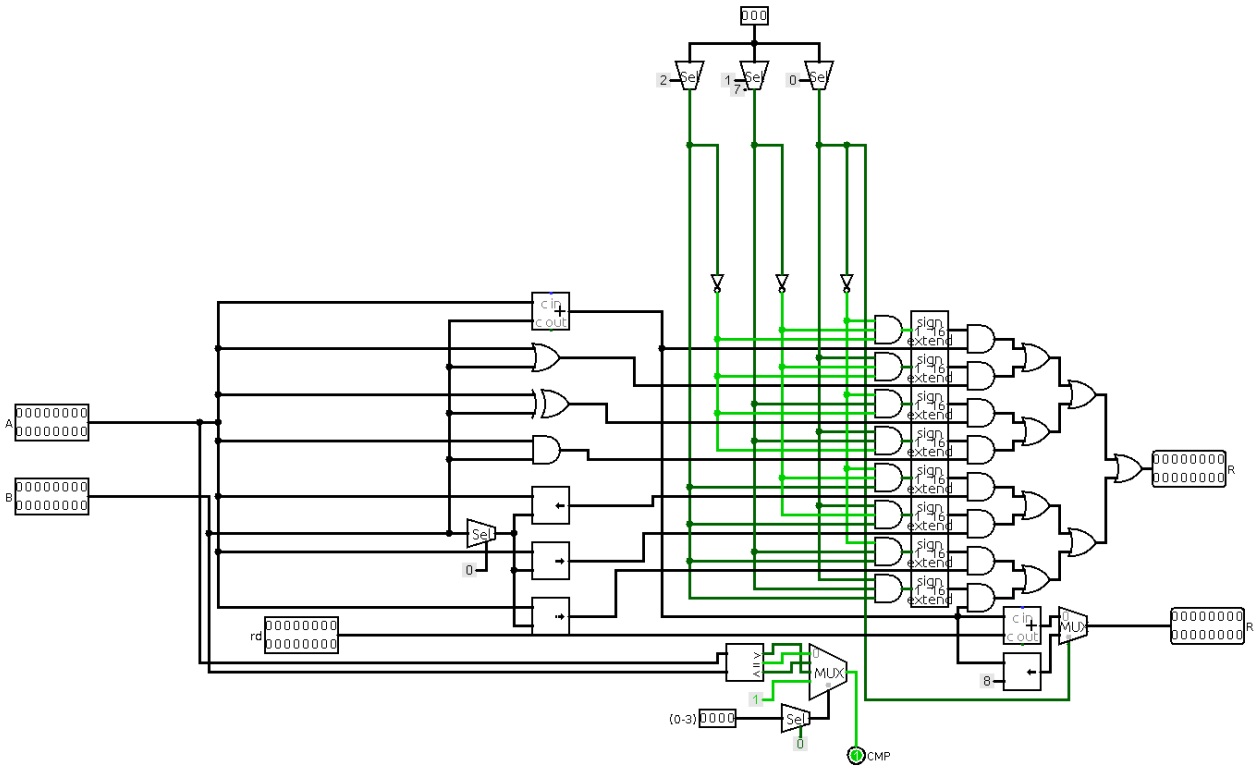


The input of the demultiplexer (on the left) will be a 1 if the instruction is not a JMP type or a storing instruction. Then, the 1 (or 0) will be routed to the enable input of the rd register (as seen in the select bit of the demultiplexer). The other registers won't be receiving that 1, meaning that the value received from the ALU or memory (in this case, the input square at the bottom of the circuit) will only be stored in the desired register (rd). The three multiplexers (on the right) receive as inputs the values of every register, and will give as an output the value of the desired registers. The one on the top is in charge of s1, the middle one of s2, and the last one at the bottom, of rd, which can be used as a source too. The s1 and s2 to multiplexers are connected to two outputs, A and B. These outputs are the inputs the ALU receives to operate. However, the mov and movu instructions don't need registers, as they are immediate value instructions. As a result,

two multiplexers are needed at the end of the circuit to determine whether what gets to A and B are the values of the desired registers, or the immediate values of the instruction. For this, the R'/I signal is used.

5.- ALU

The ALU has two main outputs: R and CMP. R is the result of the operation specified at the last three bits of the codeop. CMP has three outputs: lower, equal or higher, but only one of them is sent as output, depending on the JMP operation. Here's the circuit:

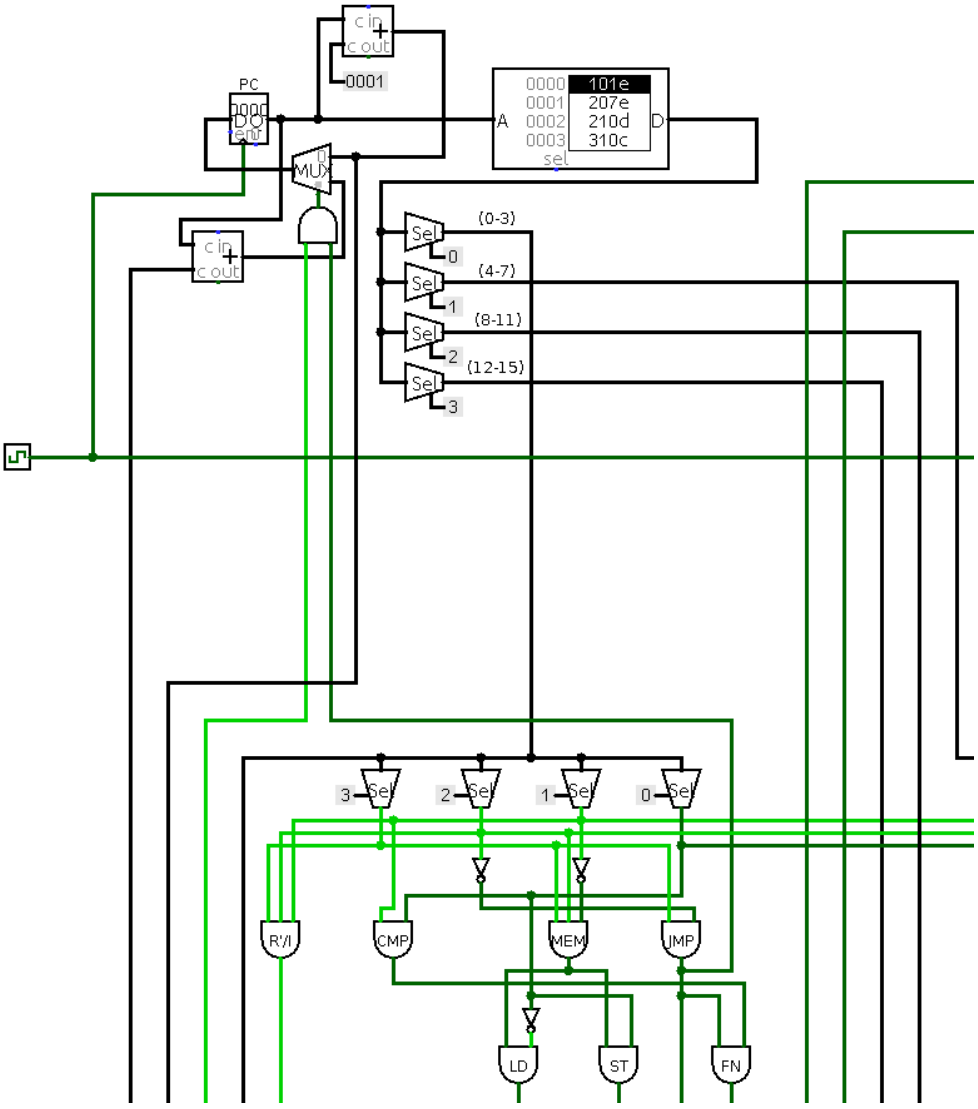


Firstly, every operation is calculated: add, or, xor, and, shift left, shift right logical and shift right arithmetic. Then, depending on the opcode bits, one of them is selected as the R output. It is worth noting that the add operation is used in the multiplexer's inputs twice. This is because it is used in both 000 and 111 operations. An adder is used to concatenate the two 4-bit segments of the mov and movu operations.

As for the CMP output, both A and B go through a comparator to calculate whether A is higher, lower or equal than B, and then an output out of the three is chosen via multiplexer (using the last two bits of the codeop as a select input).

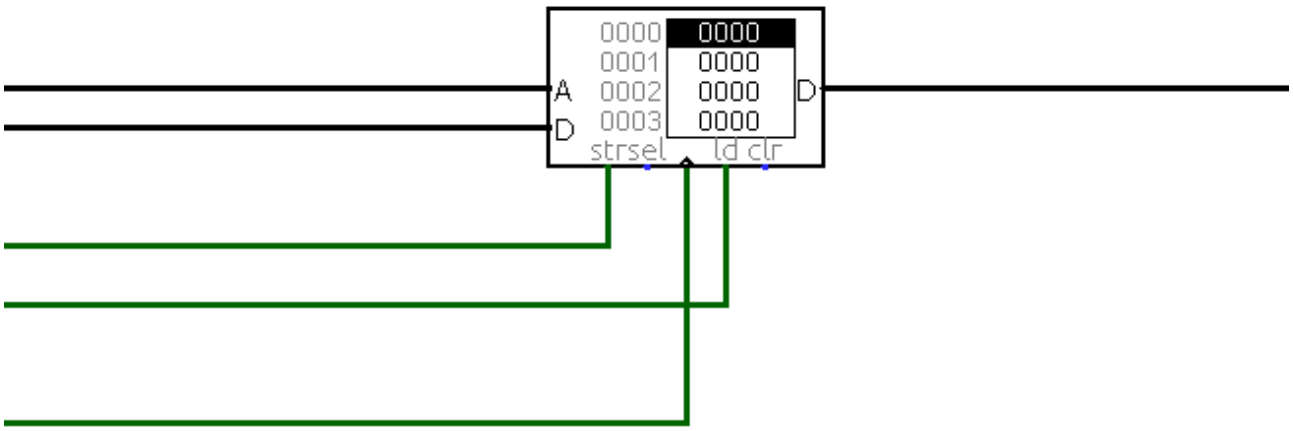
6.- Other parts of the processor

- **Program Counter:**



The PC is directly connected to the address input of the Program Memory, and it will update (PC + 1) each tick of the clock. The output of the Program Memory is the instruction to be executed, whose segments go to their respective multiplexers and the CU. The multiplexer that can be seen below the PC dictates whether the PC updates to the jump address or to the next recursive instruction. The jump address is selected if the instruction is a JMP type and if the CMP bit taken out of the ALU is set.

- **Data Memory:**



The Data Memory circuitry is very straight-forward. The value of the register equivalent to the s2 segment of the instruction (The bit set from 8 to 11) is connected to the A (address) input of the memory. The D input on the left is the value that will be stored in A if the instruction type is st. The value connected to this input is the register value that is equivalent to the last 4 bits of the instruction. The str and ld inputs are connected to the ST and LD signals respectively, for the memory to know whether the operation type requires a load or a store.