CODE DIARY

Mini-Verilog 8-bit CPU

Overview:

0. Introduction:

This project consists of the design and implementation of a mini processor using Verilog[1] for its description. The main objective was to understand the intern functionality of a simple architecture, from the definition of the ISA[2] (Instruction Set Architecture) to the execution of very simple programs on simulation.

On the next points it is described the main elements that were developed during the project:

1. Definition of the Architecture:

The defined architecture consists of:

Datapath of 8 bits.

- 8 general registers (R0-R7).
- Data memory of 8 bits.
- Instructions memory apart from data memory.
- Instructions of 16 bits.

Even though the processor works with 8-bit data, the instructions are composed by 16 bits for:

- The Opcode (4 bits)
- Destination Register (3 bits)
- Two source registers (3 bits)

This decision comes from:

- We have 10 different instructions, to define each one of them as a different instruction we will need at least 4 bits ($2^4 = 16$ possible combinations, $2^3 = 8$, which wouldn't be enough to define 10 instructions).
- At least 3 bits to define 7 different registers ($2^3 = 8$). To be able to specify each one of the registers that we need (for a non immediate operation) we need to specify a different register, 9 bits in total.

2. Design of the ISA[2]:

The definition of the format follows the next structure:

```
[15:12] opcode
[11:9] rd
[8:6] rs1
[5:3] rs2
[2:0] not used - useful in case the processor becomes "bigger"
```

Implemented instructions:

- NOP
- ADD
- SUB
- AND
- OR

- LDI (load immediate)
- LD (load from memory)
- ST (store to memory)
- JMP (incondicional jump)
- HALT

3. Implementation of the Datapath[3]:

Using Verilog[1], the following modules have been defined:

- ALU (alu.v)
  Implements arithmetico-logic operations of 8 bits.
- Register bank (regfile.v)
  8 registers of 8 bits with two read ports and one write port.
- Data memoria (dmem.v)
  Simple RAM for LD and ST operations.
- Instruction memory (imem.v)
  Loads the program from a .hex file.
- CPU (cpu.v)
  Implements the control unit by a multi-cycle FSM[4].

4. Implementation of the FSM[4] (Control Unit):

The CPU is based on a finite state machine with the next phases:

- FETCH
- DECODE
- EXECUTE
- MEMORY (if we need it)
- WRITEBACK (if we need it)

This structure lets the user correctly execute simple instructions, such as ADD or even instructions with additional words, such as LDI, LD, etc.

5. Hexadecimal[5] Programs Creation:

In the beginning, hexadecimals were created by hand, manually codifying the instructions following the defined format previously.
Such as:

```
5200
0003
5400
0005
1650
F000
```

Which loads values to the registers, adds them and finally ends the execution of the program.

6. Development of an Assembler[6] in Python:

To make easier running programs on this mini-processor, its been developed an assembler[6] which:

- ❖ Converts assembly code (which is easier to use to code a program) to hexadecimal.
- ❖ Manages labels.
- ❖ Generates automatically the binary codification.
- ❖ Creates a .asm.lst file with the information of the depuration.

7. Generation of the List File:

To make the depuration of a code easier, the assembler[6] generates the "lst" file, which was previously mentioned, but in this point of the documentation will be defined a little bit better.
This file includes:

- ❖ Address of each instruction
- ❖ Codification in hexadecimal
- ❖ Original line in Assembly

This makes possible to verify that the assembly translation to machine code is correct.

8. Automatization with Makefile:

In the repository a Makefile can be found. This integrates the automatization of the execution of the whole machine by:
Executing the assembler[6] (`make asm`)
Compiling and simulating with Icarus Verilog[1] (`make sim`)
Execute the whole flow of the machine (`make run`)

9. Simulation and Verification:

The CPU gets tested by:

- ❖ Testbenching it with Verilog[1].
- ❖ Simulating it with iverilog[7].
- ❖ Generating a ".vcd" to visualize signaling.

10. References:

[1] https://www.verilog.com
[2] https://www.geeksforgeeks.org/computer-organization-architecture/microarchitecture-and-instruction-set-architecture/
[3] https://www.youtube.com/watch?v=5ox9Yt3z5d0
[4] https://www.freecodecamp.org/news/finite-state-machines/
[5] https://www.freecodecamp.org/news/finite-state-machines/
[6] https://www.lenovo.com/in/en/glossary/assembler/?orgRef=https%253A%252F%252Fwww.google.com%252F
[7] https://github.com/steveicarus/iverilog