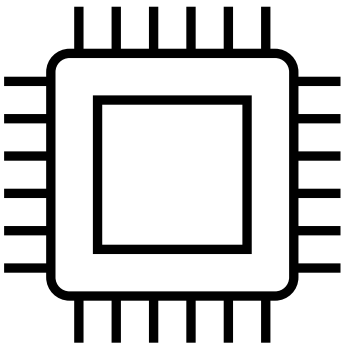
Curtin University

Creating a 16 Bit Processor in VHDL

Design Report

Written by Khaled Ali Zulfikar

20166322



Top Level CPU Explanation and CPUv1.0 Simulation:

https://youtu.be/YZlNr3CSA7o

Arithmetic Unit, ROM, Counter Register, Control Unit and CPUv2.0 Simulations:

https://youtu.be/-nhVXcA7opo

October 2023

Contents

Table of Figures

Table of Tables

1.0 Introduction

During the semester, I have created a 16-Bit CPU in Quartus Prime software. The aim of this assignment is to implement several of the CPU components, such as memory management and arithmetic operations, as seen in Eddiewastakens' 8-bit Discrete CPU which can be found here: [GitHub - eddiewastaken/logisim-discrete-CPU: An 8-Bit (mostly) discrete CPU, built in Logisim.] The aim is to construct the CPU entirely in VHDL, and then simulate the code on an Alterra DE-10-LITE FPGA.

2.0 CPU Components

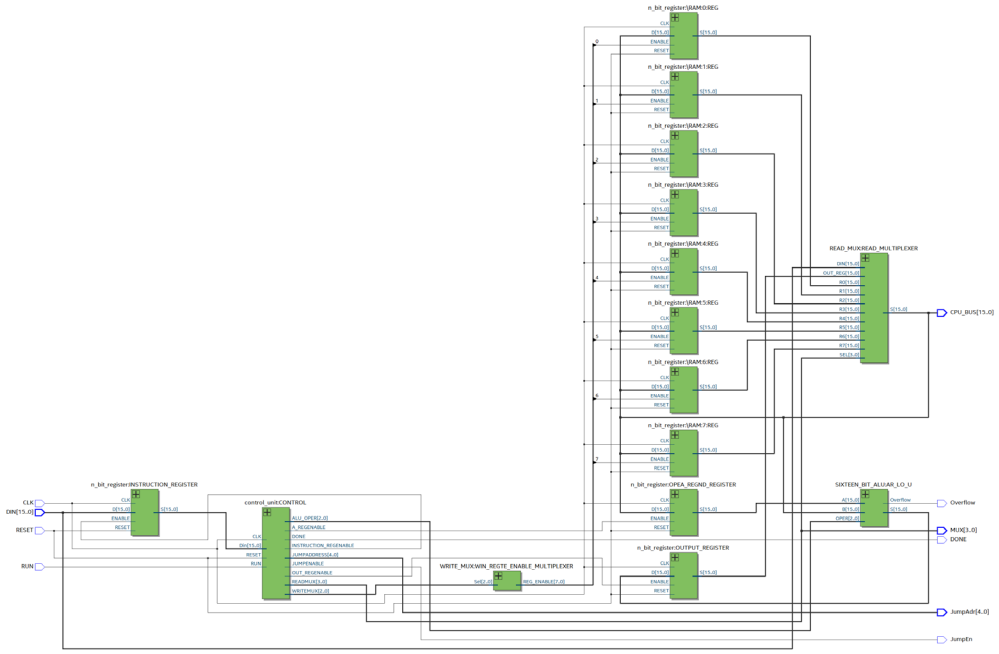


Figure 1: RTL Viewer of the 16-bit CPU

2.1 The Arithmetic Logic Unit (ALU)

The basic building block of an ALU is a one-bit full adder. This full adder takes in 3 inputs, A, B and Carry in. Its outputs, S and Carry Out will depend on the sum of the three inputs. These full adders can then be daisy chained to create an N-bit full adder.

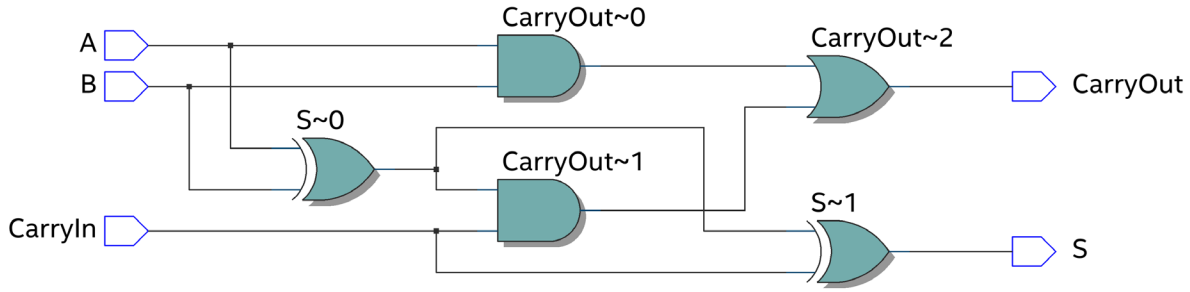


Figure 2: RTL Viewer of 1-bit Full Adder

Subtraction can be performed in the ALU by a 2's complement, which is done by setting 'not B' and setting the carry in to 1. The ALU that I have created is a similar implementation to the reference design, that includes addition and subtraction, but I have added extra logic functionality (AND, OR, NOT, XOR). The ALU takes in two 16-bit signals as operands and returns the result of their selected operation. The operation to be carried out must be first selected by the control unit. The ALU is controlled by the control unit through a 3-bit operation signal.

The ALU contains two registers, namely register A at the A input of the ALU and an output register at the calculation output of the ALU. These registers are used to temporarily store the data in memory, so the CPU cycles run correctly. The control unit will signal operand A to be stored in register A while it waits for operand B. Similarly, the result is sorted in the output register as it waits for the control unit to activate its transfer to the registers.

ALU Signal IN	Resultant Output of Operands
000	ADD
001	SUB
011	AND
100	OR
101	NOT
110	XOR

Table 1: ALU Operation

2.2 Registers (RAM)

The CPU is made up of 8 registers that can each store 16 bits of memory. Cumulatively, these 8 registers correspond to the processors Random Access memory (RAM). These registers store the calculation data, whether it is the operands for the ALU input or the results of the calculations. The data is processed through the program's instructions, which is implemented by the control unit. The control unit decides which registers interact with, either to write or to move data from one register to another. The enable signal for a given register is sent by the register multiplexer, where it enables writing to that register after a clock cycle, and the signal travels from an output of another component via the bus of the processor. The information is stored until the reset key is pressed. Other registers in the CPU include the instruction register, the A register (for the ALU) as well as the output register.

2.3 The Control Unit

The control unit is fed the 16-bit input data via the instruction register, and the most significant 4 bits, bit 15 to 12 are the opcode, which describe the function that the CPU will perform in that cycle based on the instruction word. The following is the table of the opcodes as well as their function:

Opcode	Mnemonic	Function
0000	MV	Allows the user to write a given register with the read memory from another register
0001	DMV	Allows the user to write the instruction into a selected register
0010	ADD	Add two selected registers A and B and output write to register C
0011	SUB	Subtract two selected registers A and B and output write to register C
0100	LD1	Loads a selected register A with binary 1
0101	AND	Logic AND two selected registers A and B and output write to register C
0110	OR	Logic OR two selected registers A and B and output write to register C
0111	XOR	Logic XOR two selected registers A and B and output write to register C
1000	NOT	Logic NOT two selected registers A and B and output write to register C
1001	DADD	Add the instruction and a given register A, store at register C
1010	DSUB	Sub the instruction and a given register A, store at register C
1011	JUMP	Jumps to the given address in ROM
1100	DAND	Logic AND the instruction and a given register A, store at register C
1101	DOR	Logic OR the instruction and a given register A, store at register C
1110	DXOR	Logic XOR the instruction and a given register A, store at register C
1111	DNOT	Logic NOT the instruction and a given register A, store at register C

Table 2: Table of Opcodes

The control unit is the brain of the CPU, it is responsible for translating instructions into the operation of the CPU. The control unit can recognise the instructions and activate the respective components required by the instruction code, such as the ALU, RAM, multiplexers as well as the instruction register to load the next instruction.

2.3.1 The Data Path (Fetch, Decode, Execute, Write-back)

The control unit was constructed in VHDL using 3 processes. The first process, code_future_state defines the next state based on the input opcode. As some instructions only require two states (move and direct move), these can skip to the final stage from the initial stage and do not need to pass through state1 and state2 like the rest of the instructions. The next process changes the state of the processor from present state to its future state based on the clock, reset and enable signals. The final process determines the signals sent out to the various components for each state of the CPU based on the instruction code.

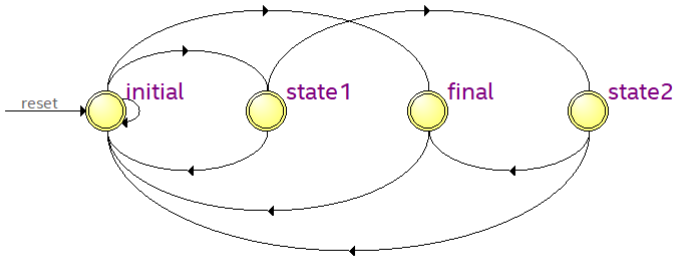


Figure 3: Quartus State Machine Viewer of the Control Unit

Using the above diagram as a reference, the initial state represents the loading of the instruction from the instruction register, or the 'fetch' process. State1 represents the 'decoding' of the instruction and storing an operand in the AREG. State 2 is where the second operand is sent by the control unit and the 'execution' of the CPU takes place and the result is stored in the ALU's output register. The final state represents the write back, where the signal from the output register is written to memory.

2.3.2 Using ADD as an Example

The add instruction will enter all four states, hence it is a good example for the operation of the control unit. In the following table it will demonstrate the signals sent to all of the control units outputs given an instruction add r0 r1 to r3.

CONTROL UNIT STATE	INSTRUCTION REGISTER ENABLE	OUTPUT REGISTER ENABLE	A REGISTER ENABLE	DONE SIGNAL	ALU	WRITE MUX	READ MUX
INITIAL	1	0	0	0	NULL	NULL	NULL
STATE1	0	0	1	0	NULL	NULL	REG0
STATE2	0	1	0	0	ADD	NULL	REG1
FINAL	0	0	0	1	ADD	REG3	REGOUT

Table 3: Operation of the Control Unit for Opcode "0010 0010 0011 0111" i.e. add r0 r1 r3

2.4 The WRITE ENABLE Register Multiplexer

The write register multiplexer is in charge of sending the enable write signals to a single selected 16-bit register out of the total 8. It takes in a 3-bit signal from the control unit, which is specified in the instruction code.

Register Number	WRITE Address (Control Unit Signal)

0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Table 4: WRITE Address Control Unit Signal

2.5 The READ Multiplexer (Memory access)

The read multiplexer is in charge of selecting which signal to display on the CPU's output bus. This is determined by the control unit, and there are 11 signals to choose from using a 4-bit input. The combination of the read and write multiplexers allows the user to write whatever signal required to memory. This includes information already stored on memory, the input 16-bit signal and the output register from ALU operations.

Signal	READ Address (Control Unit Signal)
ALU OUTPUT REG	0000
INPUT	0010
R0	0011
R1	0100
R2	0101
R3	0110
R4	0111
R5	1000
R6	1001
R7	1011
NULL (set to display BCD ADED)	ELSE

Table 5: READ Address Control Unit Signal

2.6 ROM Memory, Counter and Instruction Register

All the instructions are loaded onto the processor before the program is ready for use. These instructions are stored in the processors memory (ROM). The ROM contains a program counter, that contains 32 instruction addresses, allowing for 32 instruction words to be stored in the ROM before compilation. The 16-bit instruction lines are then sent one by one from the memory and loaded onto the instruction register. At all times, the instruction register will contain an instruction being executed by the control unit. It is only until after an instruction has been processed, the control unit will send a signal to the instruction register to load the next instruction.

CPU v2.0

In CPUv2.0, I have made the ROM automatically cycle to the next instruction once the previous instruction has been executed, by using the done signal from the control unit. This will allow the CPU to continually sequence through the entire code stored in ROM without user input. Additionally, adding a 'jump' opcode in the ROM will cause the CPU to iterate through a user programmed loop.

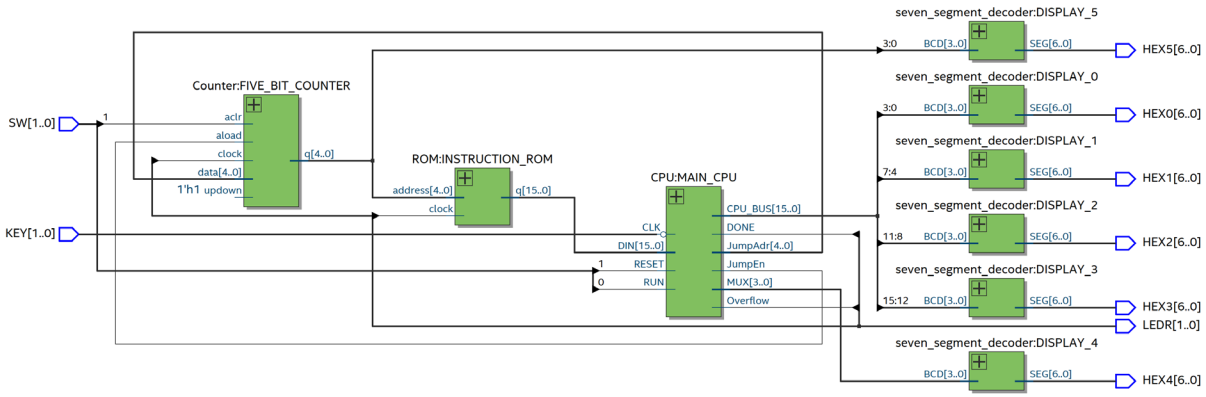


Figure 4: RTL Viewer of Top Level CPUv2.0

2.7 Binary to 7-Seg Decoder

For testing purposes on our DE-10 LITE FPGA, I used a 16-bit binary to 7 seg decoder so I can represent any binary signal that I choose as a reference and display it on my FPGA for results analysis as well as programming and debugging. The DE-10 LITE uses very specific pin assignments for its seven-segment display, including being an active low. I have represented the decoder in the following table:

7-Segment Display Output	7 Segment Display Signal	4-Bit Binary Input signal
0	1000000	0000
1	1111001	0001
2	0100100	0010
3	0110000	0011
4	0011001	0100
5	0010010	0101
6	0000010	0110
7	1111000	0111
8	0000000	1000
9	0010000	1001
A	0001000	1010
B	0000000	1011
C	1000110	1100
D	1000000	1101
E	0000110	1110
F	0001110	ELSE

3.0 CPU Programming

3.1 The Instruction Code Format

The instruction code format for the most significant bits will vary based on the selected of the 14 operations. The remaining bits have no function unless it is a direct operation with the instruction code (DMV, DADD etc.).

Instruction	16-BIT Instruction Code Format			
MV	4-BIT OPCODE	4-BIT READ address	3-BIT WRITE address	
DMV	4-BIT OPCODE	3-BIT WRITE address		
ADD	4-BIT OPCODE	4- BIT READ address	4- BIT READ address	3-BIT WRITE address
SUB	4-BIT OPCODE	4- BIT READ address	4- BIT READ address	3-BIT WRITE address
LD1	4-BIT OPCODE	3-BIT WRITE address		
AND	4-BIT OPCODE	4- BIT READ address	4- BIT READ address	3-BIT WRITE address
OR	4-BIT OPCODE	4- BIT READ address	4- BIT READ address	3-BIT WRITE address
XOR	4-BIT OPCODE	4- BIT READ address	4- BIT READ address	3-BIT WRITE address
NOT	4-BIT OPCODE	4- BIT READ address	3-BIT WRITE address	
DADD	4-BIT OPCODE	4- BIT READ address	3-BIT WRITE address	
DSUB	4-BIT OPCODE	4- BIT READ address	3-BIT WRITE address	
JUMP	4-BIT OPCODE	5-BIT INSTRUCTION ADDRESS IN ROM		
DAND	4-BIT OPCODE	4- BIT READ address	3-BIT WRITE address	
DOR	4-BIT OPCODE	4- BIT READ address	3-BIT WRITE address	
DXOR	4-BIT OPCODE	4- BIT READ address	3-BIT WRITE address	
DNOT	4-BIT OPCODE	3-BIT WRITE address		

Table 6: Instruction Code Format

3.2 Writing ROM Program Using Quartus

A program can be written and loaded onto the CPU using a memory initialisation file (.mif) in Quartus. Since the counter chosen for the ROM is 5-bit, there is a maximum of 32 allowed instruction words of 16-bit per instruction able to be loaded onto the ROM. Once a memory file has been created, it can be loaded into the program by selecting 1 port ROM from the Quartus IP catalogue and entering the parameters and specifying the memory file location on your desktop.

3.3 The Fibonacci Program

To demonstrate the functionality of my CPU, I have created a program that will output the Fibonacci sequence, similar to the one available on eddiwastaken's Fibonacci ROM. Addresses 3 and 4 will need to be looped in order to achieve the Fibonacci sequence. This program is available at the appendix of my project report.

Instruction Address in ROM	Mnemonic	16-Bit Instruction Code
0x0	LD1 R3	0100 0110 0000 0000
0x1	LD1 R4	0100 1000 0000 0000
0x2	ADD R3 R4 R1	0010 0110 0111 0010
0x3	ADD R3 R1 R3	0010 0110 0100 0110
0x4	ADD R1 R3 R1	0010 0100 0110 0010

Table 7: Fibonacci Sequence Program

4.0 Testing and Simulation

After the code for the CPU's components has been written, the user must carry out the simulation process to test each component of the CPU, to validate the functionalities and obtain timing information. This is done either by using a testbench file through Modelsim, which can be used to simulate the waveform of various signals in each component, as well as the use of a Field Programmable Gate Array (FPGA), where the VHDL code can be simulated on the board. The use of an FPGA generally will require the user to create a second testing VHDL file, where the boards pin assignments can be configured as well as components such as the ROM, counter, clock and others.

An FPGA is an integrated circuit that can be reprogrammed after manufacturing. It contains a series of logic blocks and interconnections which can be programmed to perform various digital functions.[1]

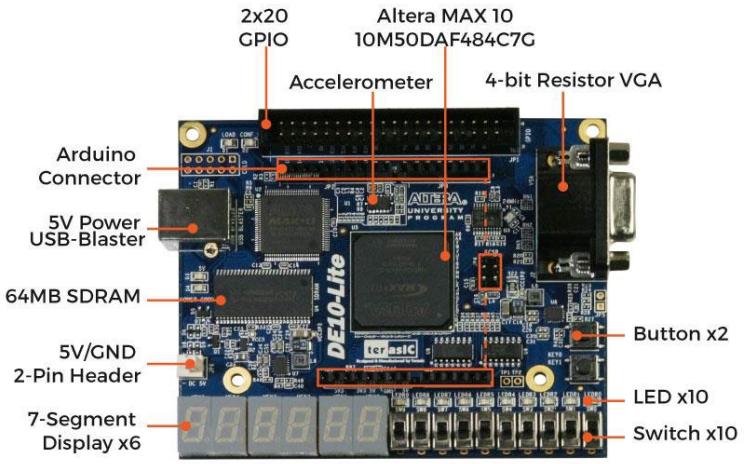


Figure 5: Terasic DE10-Lite FPGA

4.1 FPGA Testing Process

In my experience, I found that the simulation/debugging process on the FPGA was much more convenient than Modelsim and intermediate testing before the top-level design was the most beneficial. In doing so, I have created the following test files for my FPGA which are available in appendix A of this report:

- Test_Full_Adder_1bit.vhd
- Test_Full_Adder_nbit.vhd

- Test_AddSub_nbit.vhd
- Test_Register_1bit.vhd
- Test_Register_nbit.vhd
- Test_7segdecoder.vhd
- Test_5bitcounter.vhd
- Test_ROM.vhd (which can be loaded with any .mif file)
- Test_ControlUnit.vhd
- Test_CPU.vhd
- Test_CPUROM.vhd (which can be loaded with any .mif file)

All the above test files will be demonstrated in the 5-minute YouTube video attached to this project report.

5.0 Future Improvements

The Processor design that I have created is very capable but can be further improved to better functionality and ease of programming and debugging.

5.1 Multiplication Function

Future improvements to this CPU would be to include a multiplication function, which would need a second output register to display more bits.

5.2 Clock

Another improvement is to implement an autonomous clock which cycles through the CPU's instructions at a given rate (eg. 1Hz) so the functionality can be displayed without the need of manually pressing the button for the input clock.

5.3 Output Display Improvement

Other improvements can be to include a functionality to cycle between different outputs of the seven seg display. For an example, the switches on the FPGA can be used to switch between the multiplexer (bus) output, any of the selected registers, or even an internal signal in one of the components.

6.0 Conclusion

During this project I have learned to use VHDL language to describe electronic circuits, which includes their behaviours and their structures. I have found VHDL is a more practical and technical language and offers functionality far greater than tedious block diagram files. During this project, I also improved my knowledge of the structure and flow of the CPU.

References

GitHub - eddiewastaken/logisim-discrete-CPU: An 8-Bit (mostly) discrete CPU, built in Logisim.