Project 08: Assembler

In our last project, the concept of CS organizations upon which we focused our effort was the construction and utilization of an assembler. An assembler is an essential device in computing; it's a computer program that translates a list of instructions, expressed in a simple mnemonic notation, into machine code suitable for running on a computer. In this project, we implemented a 2-pass assembler specifically designed for the CPU that we had developed in our previous project. To verify the correctness and functionality of our assembler, we thoroughly tested it with a sample example provided in the project guidelines. Also, to demonstrate its efficiency further, we wrote and tested assembly language programs for the Fibonacci series and recursive sumN function. These tests not only verified the capability of the assembler to handle complex tasks and also strengthened our understanding regarding assembler operations in software development.

Assembler:

Our task involved the development of a two-pass assembler for the CPU designed in our previous project. The core function of a two-pass assembler begins with tokenization, where the assembly program is processed to eliminate white space and comments. This step breaks down the program into tokens, creating distinct elements for each part of the file and organizing numbers and symbols as individual strings. The result of this tokenization is a structured list of lists, each containing the necessary instructions for the program. The first pass of the assembler plays a crucial role in interpreting symbols used in assembly language, particularly for branching operations. It maps each label within the program to its corresponding line number, derived from the tokenized list. The output from this pass is twofold: a refined list of lists containing all the tokens, excluding the labels, and a dictionary that pairs each label with its respective line number. The final phase is the second pass, which utilizes the processed list of tokens and the label dictionary from the first pass. This step is responsible for generating the actual machine instructions that are compatible with our CPU. To ensure the accuracy and efficacy of our assembler, we conducted tests using a provided example assembly program.

Test.txt:

```
MOVEI 4 RA

start:

MOVEI 1 RE

SUB RA RE RA

BRAZ after
```

```
BRA start

after:

MOVEI 1 RD
```

Test.mif:

```
-- program memory file for test.a

DEPTH = 256;

WIDTH = 16;

ADDRESS_RADIX = HEX;

DATA_RADIX = BIN;

CONTENT

BEGIN

00 : 1111100000100000;

01 : 1111100000001100;

02 : 1001000100111000;

03 : 0011000000000101;

04 : 0010111100000001;

05 : 1111100000001011;

[06..FF] : 1111111111111111;

END
```

For the next test, we wrote a fibonacci program in assembly that generates the first 10 fibonacci numbers. Below is a screenshot of the GTKwave simulation using the machine instructions generated from our assembler and our CPU:

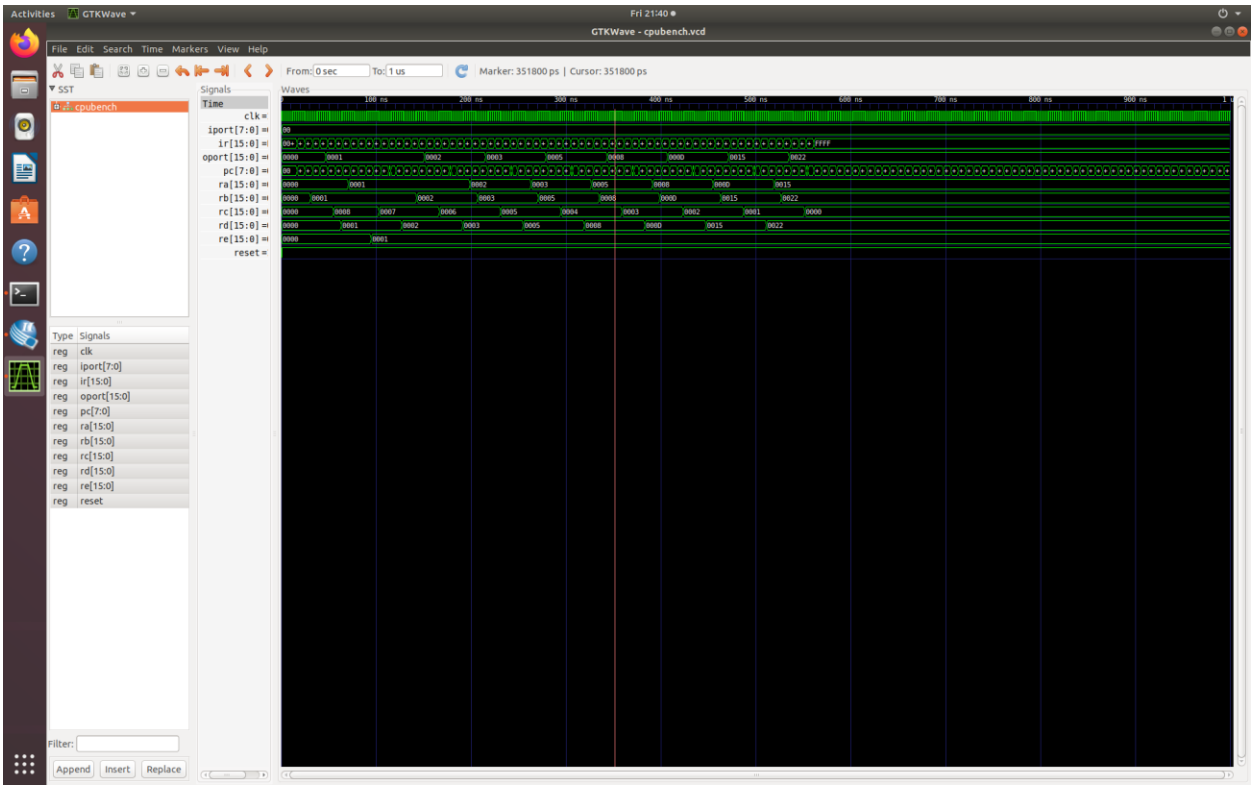


Figure: FibonacciGTK.png

In our simulation, we get a clear view of various internal registers of our CPU in action, reflecting the execution of the assembly program we crafted. This program begins with initializing the registers: setting RA to 0, RB to 1, and subsequently storing these values to the output. We then establish a counter by assigning the value 4 to RC and commencing a loop. Within this loop, we have implemented a conditional branch: if RC equals 0, the program halts; otherwise, it proceeds with the Fibonacci sequence calculations. The loop's core functionality involves adding RA and RB, storing the result in RA (and subsequently to the output), and then repeating this process with RB. After updating both RA and RB, we decrement RC by adding it to all 1’s and then loop back to the start. Upon completion of the loop, the program exits from the CPU. A key observation from Picture 3 is the output on the oport signal, as well as the values in the RA and RB registers, which showcase the first 10 numbers of the Fibonacci sequence. This visual confirmation validates that the machine code, generated by our assembler, is functioning as intended, accurately performing the Fibonacci sequence calculations and demonstrating the successful operation of our assembler and CPU.

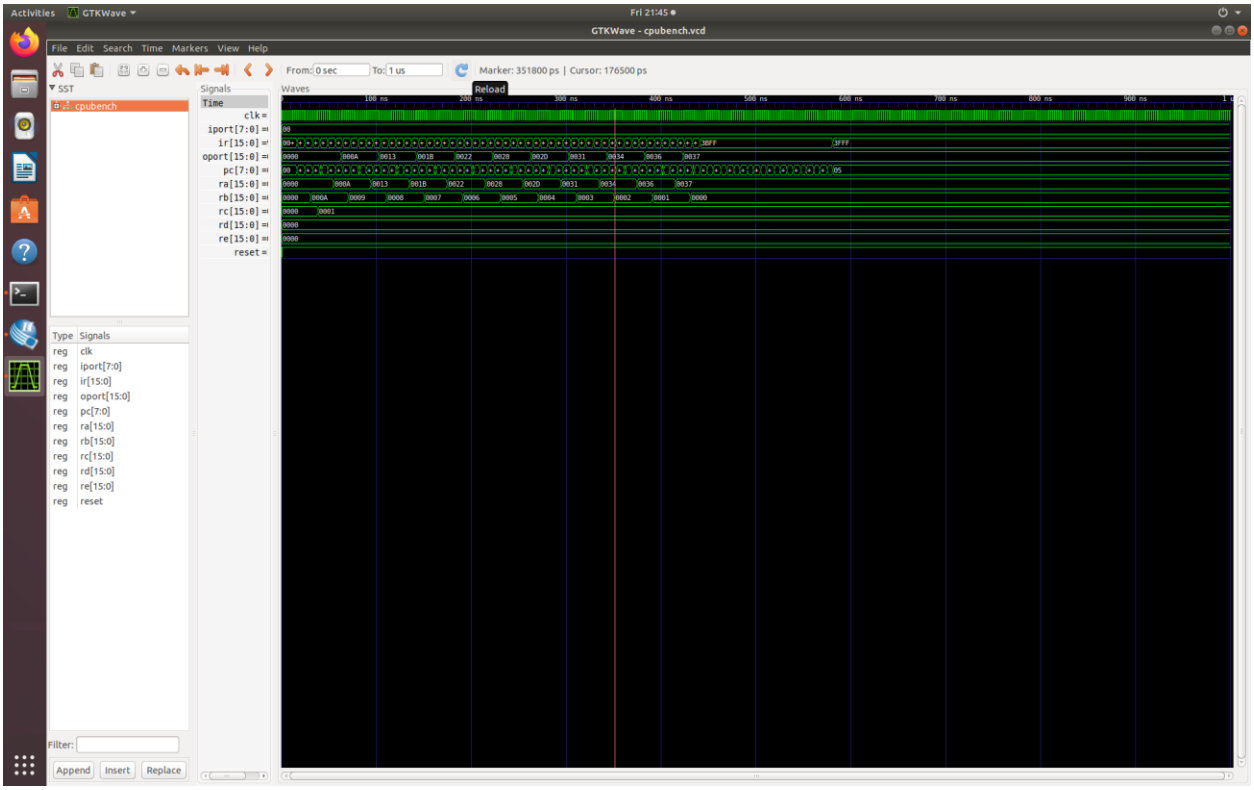
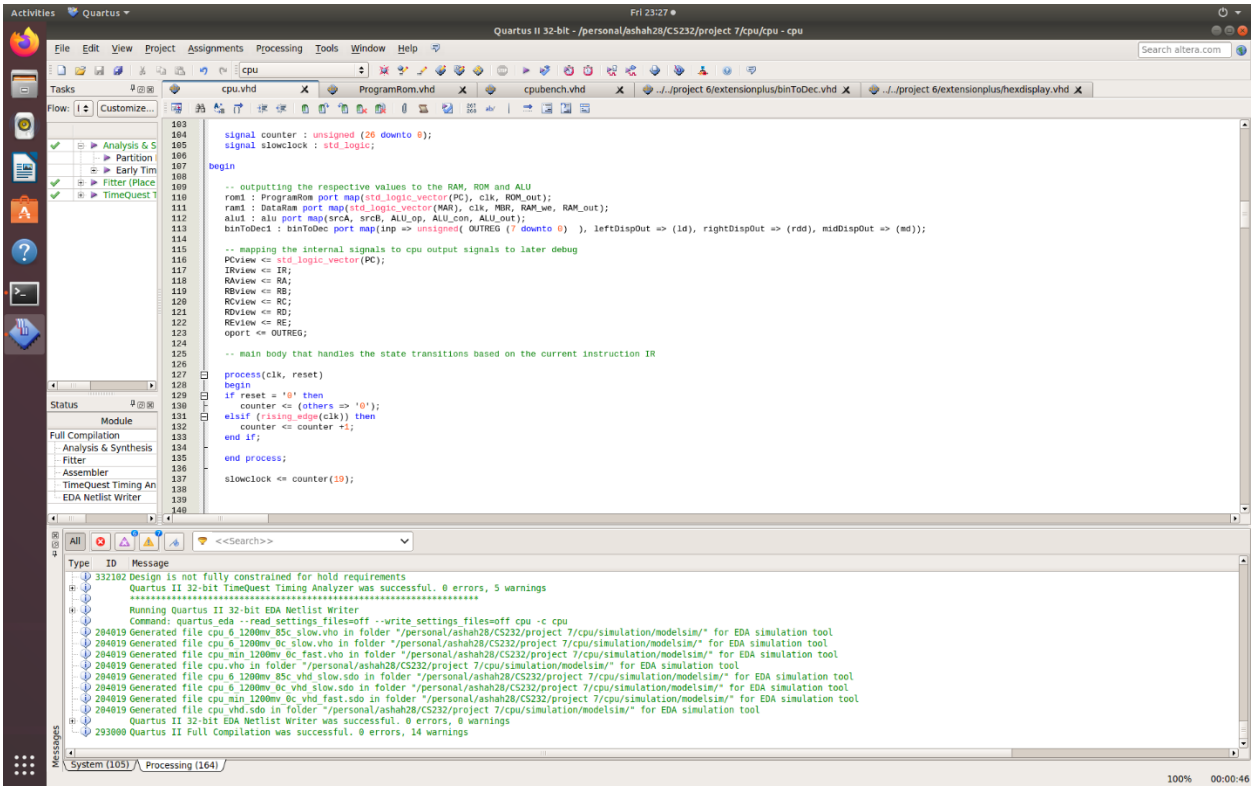


Figure: Recursive.png

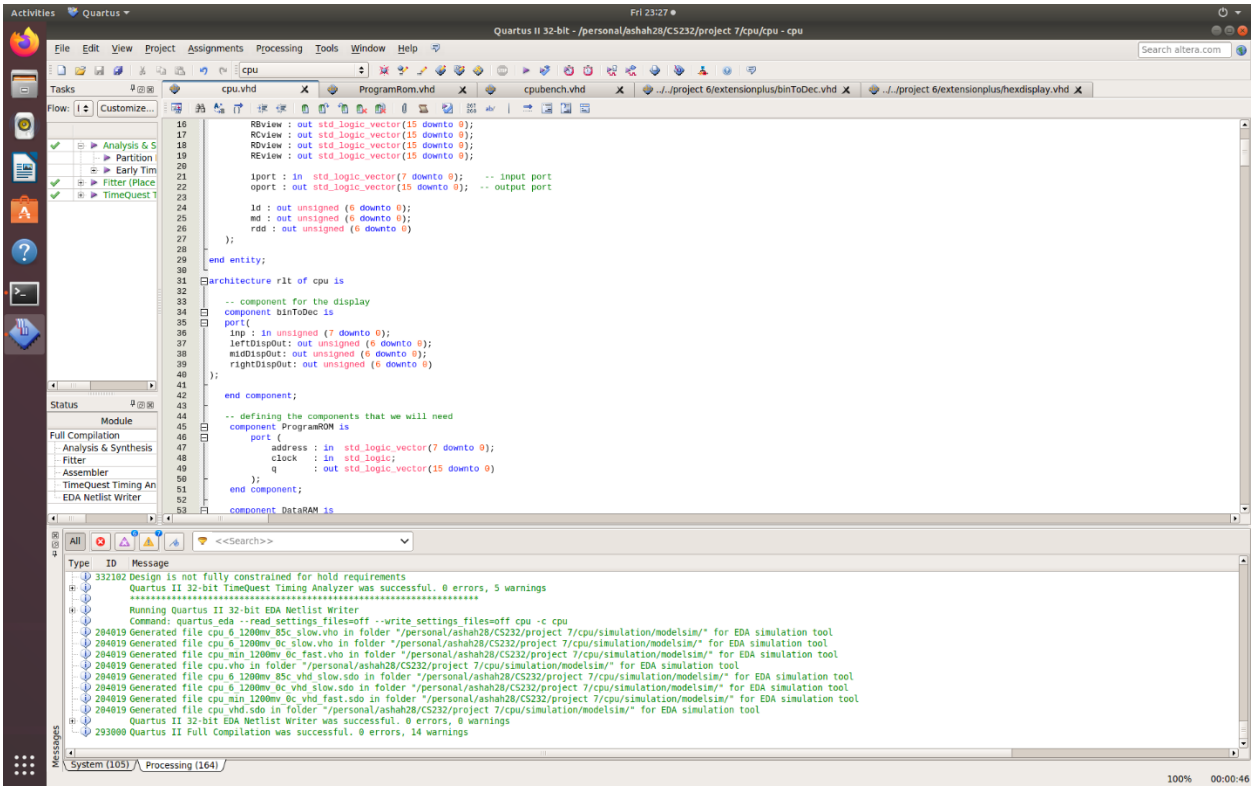
In our above figure, the internal registers of the CPU are engaged in executing an assembly program designed to sum numbers from 1 to 10. We initialized the process by setting RA to 0, RB (representing our variable N) to 10, and RC to 1. The program then enters a recursive routine, starting at the 'sum' label, where it adds RA and RB, outputs the sum to RA, and decrements RB by RC. A base case check follows, where if RB reaches 0, the program branches to the 'end' label to execute the return instruction; otherwise, it recursively calls 'sum' again. Upon satisfying the base case, the program halts after completing the return sequence. The expected result of this summation, 0x37 (55), is accurately reflected in the oport signal, affirming the successful operation and correctness of the machine code generated by our assembler and the CPU's ability to efficiently handle recursive functions.

Extension 1:

For my first extension, I used the boards to show both of my programs. For this, I slowed down the clocks:



I also imported my binary to decimal components from previous projects:



These implementations are presented in the videos titled:

Fibonacci.mp4

Sum55.mp4

Extension 2:

For my second extension, I implemented a game using my CPU and Assembler that allows a user to guess a number.

```
MOVEI 7 rb
loop:
IPORT ra
SUB ra rb rc
BRAZ correct
BRAO low
high:
MOVEI 2 rd
OPORT rd
BRA loop
low:
MOVEI 1 rd
OPORT rd
BRA loop
correct:
MOVEI 0 rd
OPORT rd
HALT
```

Figure: Game Assembly File

This simple number guessing game runs on a low-level assembly-like instruction set where a fixed secret number (in this case, 7) is stored in a register. The player provides guesses through the input port, which are read into a register. The program then subtracts the guess from the secret number and uses condition flags to determine the result: if the guess is equal, it outputs 0 and halts the program; if the guess is too low, it outputs 1; and if it's too high, it outputs 2. The process loops, continually reading input and providing feedback via the output port, until the correct number is guessed. This interaction leverages conditional branching (BRAO, BRAZ), arithmetic (SUB), and I/O instructions (IPORT, OPORT) to simulate a minimal interactive game within a constrained instruction set.

```
-- program memory file for game.a
DEPTH = 256;
WIDTH = 16;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN
00 : 1111100000111001;
01 : 0111000111111111;
02 : 1001000001111010;
03 : 0011000000001011;
04 : 0011001000001000;
05 : 1111100000010011;
06 : 0110011111111111;
07 : 0010111100000001;
08 : 1111100000001011;
09 : 0110011111111111;
0A : 0010111100000001;
0B : 1111100000000011;
0C : 0110011111111111;
0D : 0011111111111111;
[0E..FF] : 1111111111111111;
END
```

Figure: Game.mif

If the guessed number is less than the set number, the display will show 1.

If the guessed number is more than the set number, the display will show 2.

If the guess is right, we will display 0.

The game is tested by my friend and showcased in the video titled

Game.mp4

Acknowledgements:

In this project, I used help from my amazing classmates Maya, Azam, Aleksandra and my friend Muneeb and Manish for helping with testing. This would not have been possible without their help and I am grateful along with help form professors and Tas.