


KWAME NKRUMAH UNIVERSITY OF SCIENCE AND TECHNOLOGY

KUMASI, GHANA

COLLEGE OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

COE 381: MICROPROCESSORS

MICROPROCESSOR CIRCUIT DESIGN

MAY 2022

GROUP ALTERA

	Name	**Index**
1.	Sven Dzeble	8258719
2.	Max Tetteh Otuteye	8265019
3.	Prince Elikplim Kofi	8251119
4.	Dillys Naa Kai Annan	8253919
5.	Danso Eric Obeng Akese	8257919
6.	Biritwum-Nyarko Kwasi Manu	8257119
7.	Hansen Kwadwo Koduah Junior	8260019
8.	Frank Kwamena Nana Edu Kotsia Appiah	8254419

DECLARATION

We, the undersigned solemnly declare that the project report dubbed 'MICROPROCESSOR CIRCUIT DESIGN' is based on work conducted during study.

We assert the statements made and conclusions drawn are an outcome of our research work. We further certify that

I. The work contained in the report is original and has been done by this team.

II. The work has not been submitted to any other Institution for any other degree/certificate in this university or any other university.

III. The guidelines provided by the university in authoring the report have been followed.

IV. Whenever materials (data, theoretical analysis, and text) have been used from other sources, due credit to them in the text of the report and giving their details in the references have been given.

ABSTRACT

A microprocessor is a computer processor where the data processing logic and control is included on a single integrated circuit, or a small number of integrated circuits. The microprocessor contains the arithmetic, logic, and control circuitry required to perform the functions of a computer's central processing unit. “Altera” integrated this concept into designing a simple microprocessor that performs operation on operands to produce a desired output.

DEDICATION

We dedicate this project to God Almighty our creator, strong pillar, source of inspiration, wisdom, knowledge and understanding. He has been the source of our strength throughout this programme and on His wings only have we soared.

ACKNOWLEDGEMENTS

We would like to express our special thanks of gratitude to Mr. J. Yankey, our Microprocessors lecturer who gave us the excellent opportunity to work on this project, 'MICROPROCESSOR DESIGN CIRCUIT.' This project has helped learn and understand microprocessors better through research. We are very thankful.

Secondly, we would like to thank each group member for their various contributions and commitment to making this project a success.

TABLE OF CONTENT

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

NitroCompute is a **16-bit processor** designed based on **Vonn Neuman architecture**. It consists of a **power supply**, an **Algorithm and Logic Unit** (**ALU**), **Registers**, a **Control Unit** and **Status Flags**. Components were designed **modularly**; only the **Control Unit** determines what should be done at what instance. **NitroCompute** helps to perform basic operations with corresponding **opcodes**, together forming an **instruction set**.

POWER SUPLLY CIRCUIT

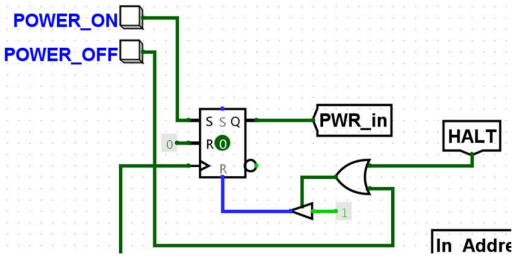

Figure 1.0

The **power supply** consists of a **SR-flip flop**. This is to ensure that the circuit stays active even if **Set** and **Reset** is **zero** once Set has an **initial one value**. The circuit is powered down either **when Set and Reset is zero**, a **halt instruction from the Control Unit** or **the power off button which toggles reset**

Set	Reset	Output
0	0	No change
0	1	Reset
1	0	Set
1	1	invalid

Table 1.0

FLAGS

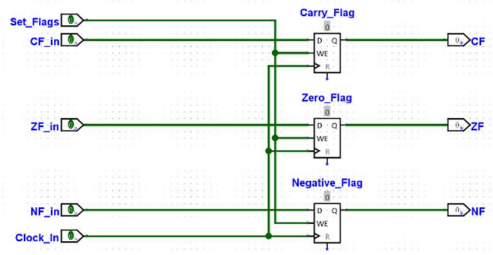

Figure 2.0

Flags are a modified kind of register that record the condition of a microprocessor's calculation. The status of each flag determines the microprocessor's next action, thus enabling it to make decisions.
The **Set flag** is connected to the **write enabled** which determines whether to set flag or not
Clock determines when values are updated

Parameter	Description
D	Data in
WE	Write Enabled
Q	Output
R	Reset (unused)

Table 2.0

CARRY FLAG

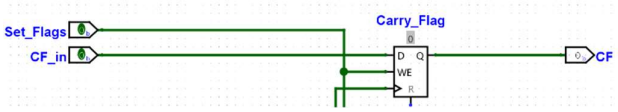

Figure 2.1

The **Carry Flag** has a register to store the value of the flag. The **Carry Flag in (CF_in)** receives carry data and sends it to the register. **Carry flag (CF)** outputs flag state to the control unit.

ZERO FLAG

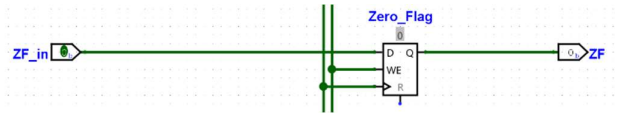

Figure 2.2

The **Zero Flag** has a register to store the value of the flag. The **Zero Flag in (ZF_in)** receives data and send to register. **Zero Flag(ZF)** outputs flag state to the control unit.

NEGATIVE FLAG

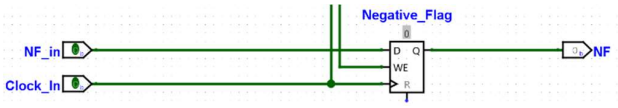

Figure 2.3

The **Negative Flag** has a register to store the value of the flag. The **Negative Flag in (NF_in)** receives data and send to register. **Negative Flag(ZF)** outputs flag state to the control unit.

REGISTERS

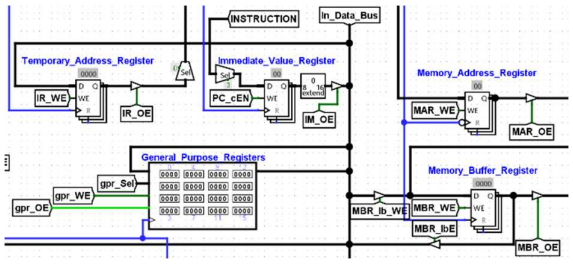

Figure 3.0

The temporary storage locations inside the **Central processing Unit** to store **data** and **addresses** is called a **register**.

GENERAL PURPOSE REGISTERS

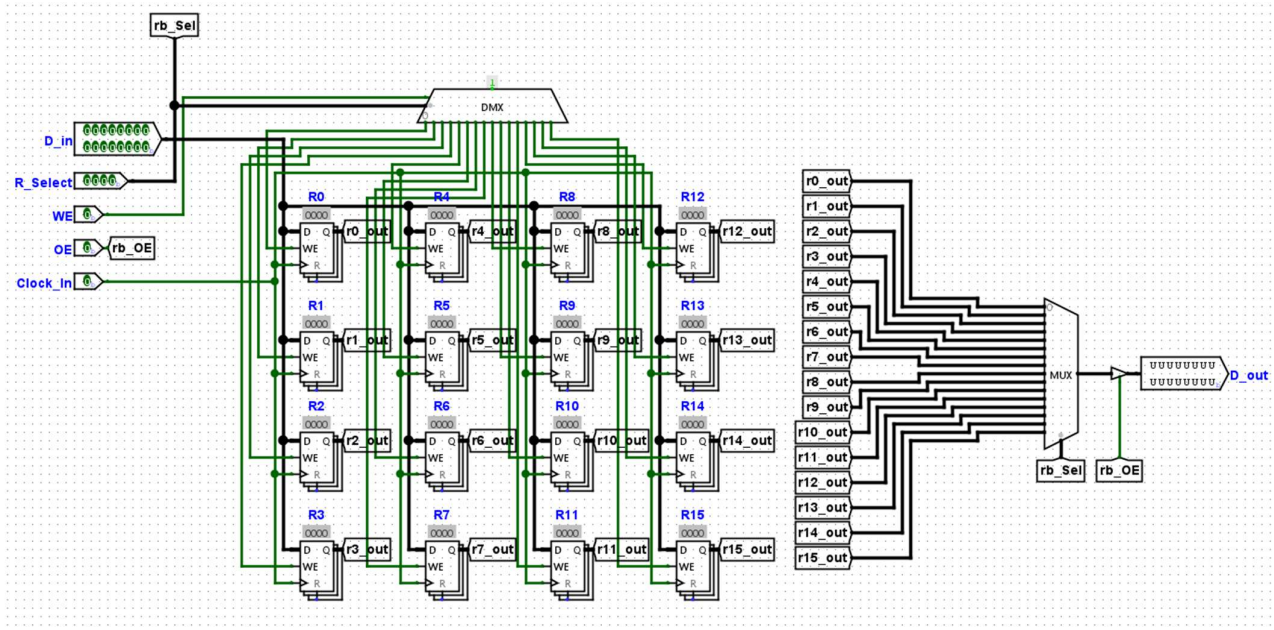

Figure 3.1

The **general-purpose register** can store a data or a memory location address. It is a **multipurpose register.** They can be used either by programmer or by a user. For **NitroCompute**, **Write Enabled** (**WE**) determines whether to store or not from **Data In** (**D_in**). **Output Enabled** (**OE**) determines whether to output the content of selected register. Register Bank Select (**Rb_sel**) tunnel helps to select a register whose content should be displayed. The **Demultiplexer** helps to select a specific register based on **rb_sel** value
The Multiplexer is used in selecting data to be outputted**. Register Bank Output Enabled** (**rb_OE**) determines whether data stored in register should be outputted. **Data out** (**D_out)** is the data being outputted

TEMPORARY ADDRESS REGISTER

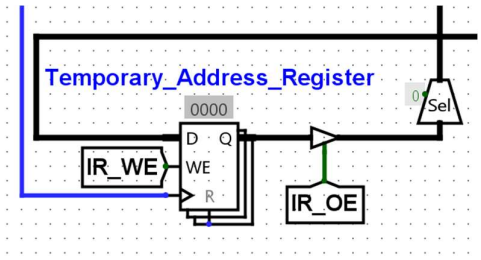

Figure 3.2

This **Address Register** was previously called the **Instruction Register**. It receives data from **the internal bars**. Based on the **bit select** value, either the first eight bits or second eight bits would be selected. It consists of a **latch** as well as **write and output enablers** to decide whether to write data or to output its result. It also consists of a **controlled buffer** which determines whether the output should be sent to the internal address bars.

IMMEDIATE VALUE REGISTER

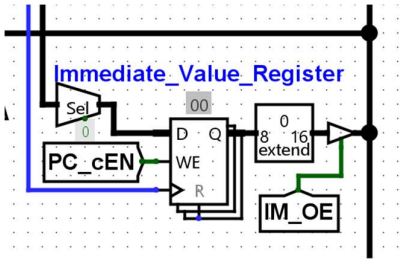

Figure 3.3

The **immediate value register** stores the immediate value in every instruction (**first eight bits**).
It receives input from instruction and outputs data to the internal data bars. It consists of a **bit extender** to push additional zeros to the instruction bits to enable data to be compatible with the number of bits received on the data bars.

MEMORY ADDRESS REGISTER

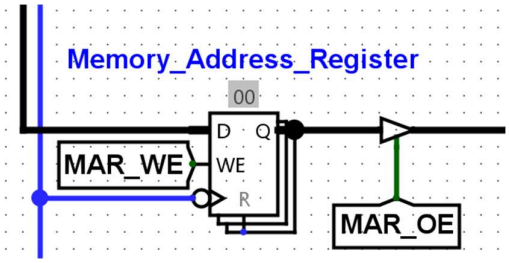

Figure 3.4

This register receives eight bits of data. It is **negatively clocked** (**enabled on falling edge**) to enable the program counter successfully to clock (**synchronization**). It also has a write enabler and output enabler to decide whether to write and store data values at each instance.
The memory address register pushes data to **the external address bus (RAM).**

MEMORY BIFFER REGISTER

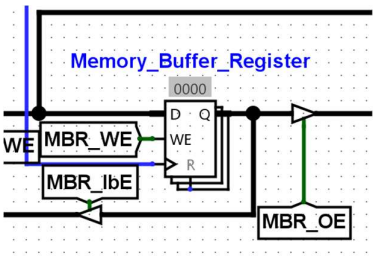

Figure 3.5

The **memory buffer register** is a **bridge** between **the internal and external data bars**. (**Two-way bridge**). It can **write** to both **the internal and external data bars** as well as **receive input from either data bars.** It has an **output enabler** which determines whether data should be pushed out of this register.

RANDOM ACCESS MEMORY

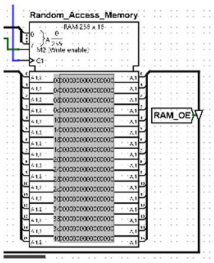

Figure 4.0

The **RAM** has a total of **256 x 16 bits** of data Its **first 128 bits** are for storing **addresses** and **second 128, for data**. It has a **write enabler** to determine whether data should be written to the **RAM** and an **output enabler** to determine whether data should be fed to the **external data bars**.

REGISTER A AND B

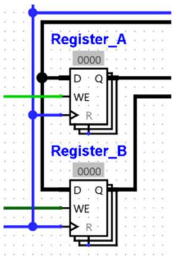

Figure 5.0

These are the only inputs to Arithmetic and Logic Unit. They serve as data input for ALU operations.

PROGRAM COUNTER

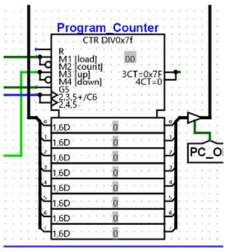

Figure 6.0

The **program counter** determines in what **order** instructions are fetched and executed. It can either **load and count but not both at an instance** and **count upwards or and downwards**. It also has a **count enable** as well as a **reset**. It has **memory to store count values**. It has an **output enabler** to decide whether to **write** to **internal data bus**. The Program Counter has max value **7f** (**hexadecimal**) after which it **restarts counting**

ARITHMETIC AND LOGIC UNIT

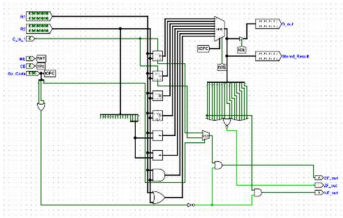

Figure 7.0

The Arithmetic and Logic Unit is a digital circuit used to perform arithmetic and logic operations. It represents the fundamental building block of the central processing unit.
The Instruction Format for an ALU is:

[OPCODE][SOURCE1][SOURCE2][DESTINATION]

Under this ALU, there are eight operations.

- **0** - **ADD** - ADDITION
- **1** - **SUB** - SUBTRACTION
- **2** - **MULT** - MULTIPLICATION
- **3** - **DIV** - DIVISION
- **4** - **SHL** - LOGICAL SHIFT LEFT
- **5** - **SHR** - LOGICAL SHIFT RIGHT
- **6** - **AND** - LOGICAL MULTIPLICATION
- **7** - **OR** - LOGICAL ADDITION

The ALU receives data from **Register 1** and **Register 2** (**Register A and B**). Every operation aside **Logical Shift Left,** and **Logical Shift Right** use 16-bit values. All eight operations are performed on the operands. A **Multiplexer** and **Write Enabler** (**WE**) determine which output should be displayed and whether an operation should be performed. **Data Out** (**D_out**) sends data to internal address bus. **Stored_Result** shows current result. The **Negative Flag** and **Carry Flag** are only triggered by the **Add** and **Sub** operations. If the result of the operation exceeds sixteen bits, the **Carry Flag** is triggered. The **Negative flag** is triggered when the **Logical Union** of the result is **zero**.
For the **logical shift left and logical shift**, the **first operand** is **sixteen bits** and second is a **4-bit value** (**LSB**). **The Operation Code** received from **CU** determines which operation output to be displayed.
The **Output Enable (OE)** determines whether the result should be displayed.

CONTROL UNIT

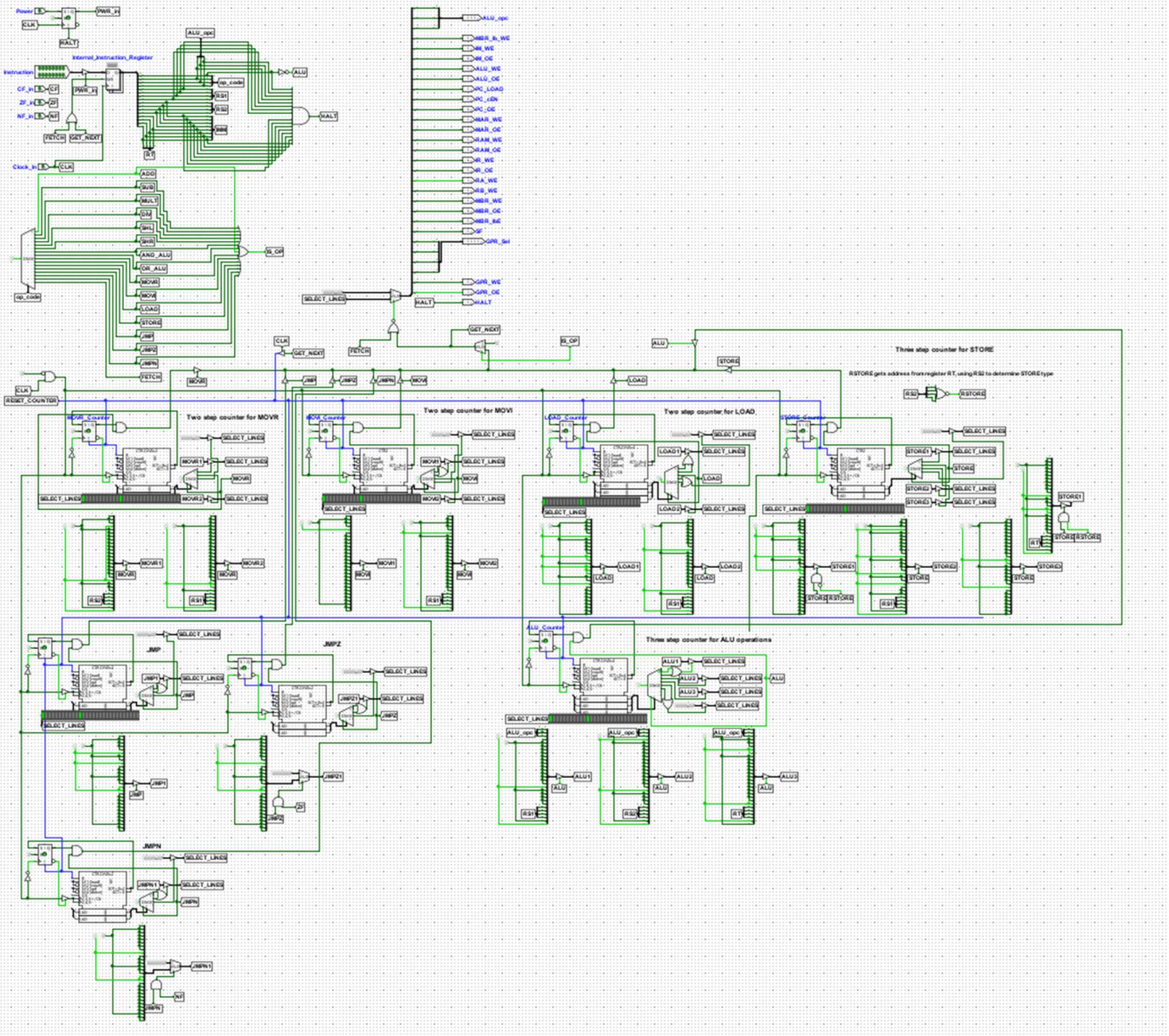

Figure 8.0

The **control unit** is the core of the entire **Central Processing Unit.** Its main operation is to **generate control signals** throughout the CPU to control when certain operations are performed. This control unit has **twenty-nine control signals from zero to twenty-eight.** For every operation, it checks for the **enabled control signals** and **generates a control sequence** for it to be performed.

The twenty-nine control signals are:

ALU_OPCODE	**-**	**0**
ALU_OPCODE	**-**	**1**
ALU_OPCODE	**-**	**2**
MEMORYADDRESSREGISTER_INTERNALBUS_WRITEENABLED	**-**	**3**
IMMEDIATEVALUEREGISTER_WRITEENABLED	**-**	**4**
IMMEDIATEVALUEREGISTER_OUTPUTENABLED	**-**	**5**

ALU_WRITEENABLED	**-**	**6**
ALU_OUTPUTENABLED	**-**	**7**
PC_LOAD	**-**	**8**
PC_COUNTENABLED	**-**	**9**
PC_OUTPUTENABLED	**-**	**10**
MEMORYADDRESSREGISTER_WRITEENABLED	**-**	**11**
MEMORYADDRESSREGISTER_OUTPUTENABLED	**-**	**12**
RAM_WRITEENABLED	**-**	**13**
RAM_OUTPUTENABLED	**-**	**14**
INSTRUCTIONREGISTER_WRITEENABLED	**-**	**15**
INSTRUCTIONREGISTER_OUTPUTENABLED	**-**	**16**
REGISTERA_WRITEENABLED	**-**	**17**
REGISTERB_WRITEENABLED	**-**	**18**
MEMORYBUFFERREGISTER_WRITEENABLED	**-**	**19**
MEMORYBUFFERREGISTER_OUTPUTENABLED	**-**	**20**
MEMORYBUFFERREGISTER_INTERNALBUSENABLED	**-**	**21**
SET FLAG	**-**	**22**
GENERALPURPOSEREGISTER_SELECT	**-**	**23**
GENERALPURPOSEREGISTER_SELECT	**-**	**24**
GENERALPURPOSEREGISTER_SELECT	**-**	**25**
GENERALPURPOSEREGISTER_SELECT	**-**	**26**
GENERALPURPOSEREGISTER_WRITEENABLED	**-**	**27**
GENERALPURPOSEREGISTER_OUTPUTENABLED	**-**	**28**

The Control Unit performs the following operations with the following selected signals.

MOVR

This operation allows us to move the contents from one register to another register.
The syntax for a MOVR operation is,

[MOVR] [RA],[RB]

Where:

MOVR is the operation being performed
RA is the **SOURCE** register
RB is the **DESTINATION** register

MOVR1 = 3,19,23,24,25,26

MOVR2 = 20,21,23,24,25,26,27

MOVI

This operation allows us to move an immediate value into a register
The syntax for a MOVI operation is,

[MOVI] [RA],[IMME]

Where:

MOVI is the operation being performed
RA is the **DESTINATION** register
IMME is the immediate value

MOVI1 = 4

MOVI2 = 5,23,24,25,26,27

LOAD

This operation allows us to load contents in memory into a register
The syntax for a MOVI operation is,

[LOAD] [RA],[ADDR]

Where:

LOAD is the operation being performed
RA is the **DESTINATION** register
ADDR is the address location of the data in memory.

LOAD1 = 5,11,12,14,15,16,19

LOAD2 = 5,21,23,24,25,26,27

STORE

This operation allows us to store the data from a register into memory.
The syntax for a STORE operation is,

[STORE] [RA],[ADDR]

Where:

STORE is the operation being performed
RA field is the **SOURCE** register
ADDR is the address location to store the value

STORE1= 5,11,15,16

STORE1* = 11,15,16,23,24,25,26,28

STORE2 = 3,11,16,23,24,25,26,28

STORE3 = 12,13,20

JMP

This operation allows us to jump to a specified instruction location in memory.
The syntax for a JMP operation is,

[JMP] [ADDR]

Where:

JMP is the operation being performed
ADDR is the address location of the instruction to jump to

JMP1 = 5,15,16

JMPZ

This operation allows us to jump to a specified instruction location in memory if the zero flag is enabled,
The syntax for a JMPZ operation is,

[JMPZ] [ADDR]

Where:

JMPZ is the operation being performed
ADDR is the address location of the instruction to jump to

JMPZ = 5,15,16

JMPN

This operation allows us to jump to a specified instruction location in memory if the negative flag is enabled.
The syntax for a JMPN operation is,

[JMPN] [ADDR]

Where:

JMPN is the operation being performed
ADDR is the address location of the instruction to jump to

JMPN = 5,15,16

HALT

This opcode halts the microprocessor
0x5e00 is fed into the Multiplexer
The syntax for a HALT operation is.

[HALT]

Where:

HALT is the operation being performed

ALU OPERATIONS

[OPCODE][SOURCE1][SOURCE2][DESTINATION]

ALU1 = 17,23,24,25,26,28

ALU2 = 3,18,23,24,24,25,26,28

ALU3 = 6,7,22,23,24,25,26,27

ASSEMBLER

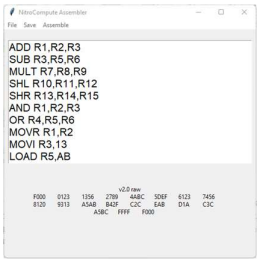

Figure 9.0

The NitroCompute Assembler was built using Python programming language and converts Assembly Language into hexadecimal code fed into The NitroCompute microprocessor. When executed, an application console which has a menu bar, text area and title bar appears.

The menu bar contains the following drop-down menus.

- File
    - Open: Allow user to provide input from a text file
    - Exit: Close application

- Save
    - Save: Store the content in the text area into a specified file.

- Assemble
    - Assemble: Allow user to convert the assembly code into a hexadecimal code and store the output into a specified file.

The text area is a rectangular are with a white background where the user can type the assembly code.

CONCLUSION

In designing NitroCompute, efficiency and cost were considered. Although it is not an ideal microprocessor, it is a start-up to undergo reviews and performance improvements together with The NitroCompute Assembler.

REFRENCES

- KNUST logo by source, Fair use, https://en.wikipedia.org/w/index.php?curid=24888272
- Definition of Microprocessor, Microprocessor - Wikipedia
- Definition of Registers, www.educba.com/register-in-microprocessor/

DEVELOPERS

maxotuteye (Max Otuteye) (github.com)
hanskod (Hansen Koduah) (github.com)
dzeble (github.com)
Adaks (github.com)
Dillys3567 (github.com)

APPENDIX

https://github.com/maxotuteye/nitro-compute.git

ADD - ADDITION
ADDR - ADDRESS LOCATION OF THE DATA IN MEMORY.
ALU - ARITHMETIC AND LOGIC UNIT
AND - LOGICAL MULTIPLICATION
CF_ in - CARRY FLAG IN
D_out - DATA OUT
DIV - DIVISION
HALT - STOP MICROPROCESSOR
IMME - IMMEDIATE VALUE
JMP - EXECUTE FROM SPECIFIED LOCATION IN MEMORY
JMPN - EXECUTE FROM LOCATION IF NEGATIVE FLAG IS TRIGGERED
JMPZ - EXECUTE FROM SPECIFIED LOCATION IF ZERO FLAG IS TRIGGERED
LOAD - PUSH CONTENT IN MEMORY TO REGISTER
MOVI - IMMEDIATE VALUE TO REGISTER MOVE
MOVR - REGISTER TO REGISTER MOVE
MULT - MULTIPLICATION
NF_in - NEGATIVE FLAG IN
NOP - STOP MICROPROCESSOR (NO OPERATION)
OE - OUTPUT ENABLED
OR - LOGICAL ADDITION
R1 - REGISTER 1
R2 - REGISTER 2
R3 - REGISTER 3
RA - REGISTER A
RB - REGISTER B
Rb_OE - REGISTER BANK OUTPUT ENABLED
Rb_sel - REGISTER BANK SELECT
RC - REGISTER C
SHL - SHIFT LEFT
SHR - SHIFT RIGHT
STORE - SAVE DATA FROM REGISTER TO MEMORY
SUB - SUBTRACTION
WE - WRITE ENABLED
ZF_in - ZERO FLAG IN