Computer and Systems Engineering Department
3rd Year CSE, Zagazig University
Course: Computer Integrated Circuits (IC/DSP)
Project final report 2021

8-Bit RISC Processor
Group 1

Name: Alzahraa Mohamed Abdel Hamid Shaheen
Code: 20812017200023
Sec: 1, Num: 25

Introduction

This project aims to apply what we have learned in computer organization and logic gates courses, so we have chosen to make an 8-bit processor as it full of challenges. We have enjoyed learning how the computer works with all details.

Requirements:

- 8-bit data bus.
- 16-bit address bus.
- Eight 8-bit general purpose registers which can be used in pairs as four 16-bit register.
- Instruction formatted.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode					Source Type		Destination Type	Source Register			Destination Register			Condition	

- Instruction set.

Instruction	Opcode	Operands
NOP	0x00	-
Add	0x01	reg, reg/imm
Subtract	0x02	reg, reg/imm
Multiply	0x03	reg, reg/imm
Logical AND	0x04	reg, reg/imm
Logical OR	0x05	reg, reg/imm
Logical shift right	0x06	reg
Load	0x10	reg, imm/address
Store	0x11	address, reg
Move	0x12	address, address
Jump	0x0F	address, condition
Halt	0x1F	-

- Jump conditions codes.

Condition	Bit designation
Always	00
Carry	01
Zero	10
Negative	11

- Source Addressing.

Source	Bit code
Register	00
Memory	01
Immediate	10

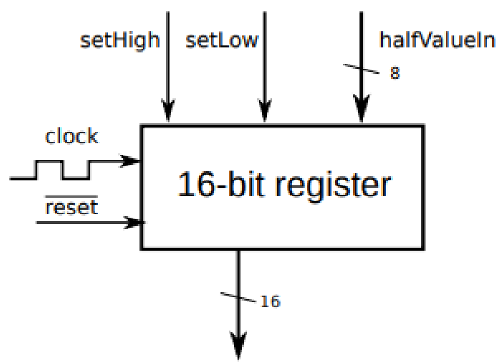
General Architecture

Where multi-byte values are used, they are stored in big-endian format, with the bits ordered from most significant to least significant. The processor has an 8-bit internal data bus which transfers data between the individual modules and connects to external RAM and ROM.

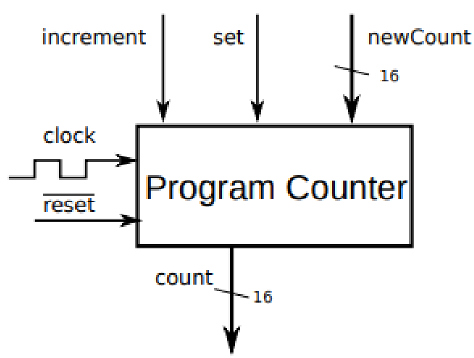
Modules:

- Generic 16-bit register:

  A 16-bit register is used three times in the design: the jump register, the memory address register, and the instruction register.

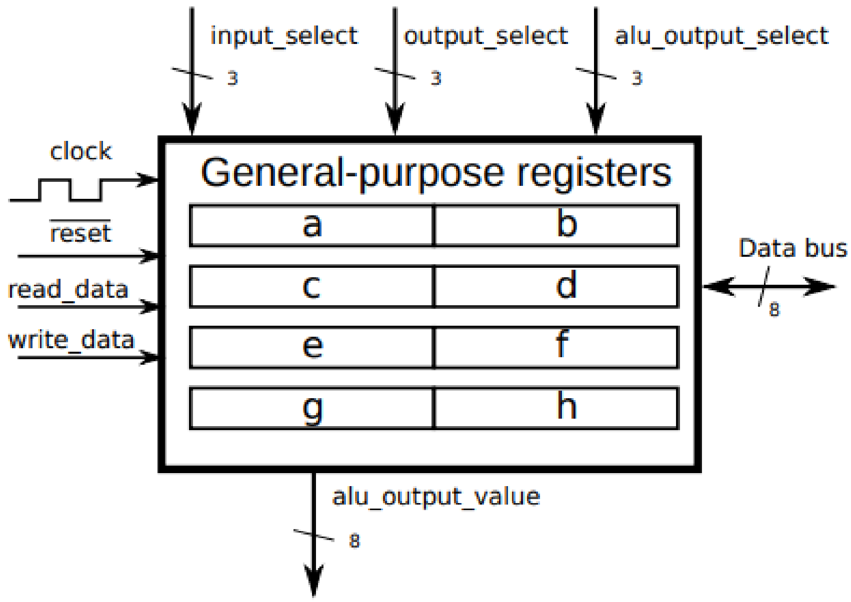


- Program counter:



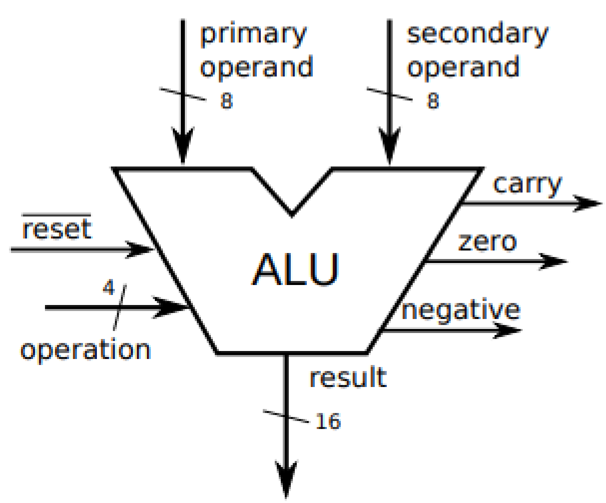
- General-purpose register block:

  The general-purpose register block contains 8 eight-bit registers which are used for data manipulation. They are grouped into pairs, creating 4 sixteen-bit registers which can receive the result of a multiply operation.



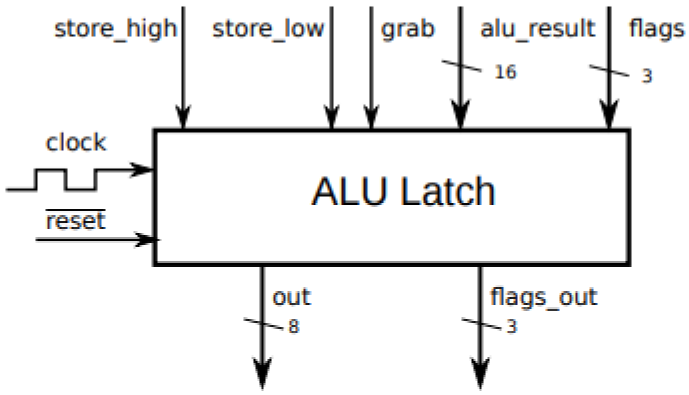
- Arithmetic logic unit (ALU)

  The arithmetic logic unit implements all of the arithmetic operations specified: addition, subtraction, multiplication, logical AND and OR, left logical shifts. Output flags are set based on the results of the operation.



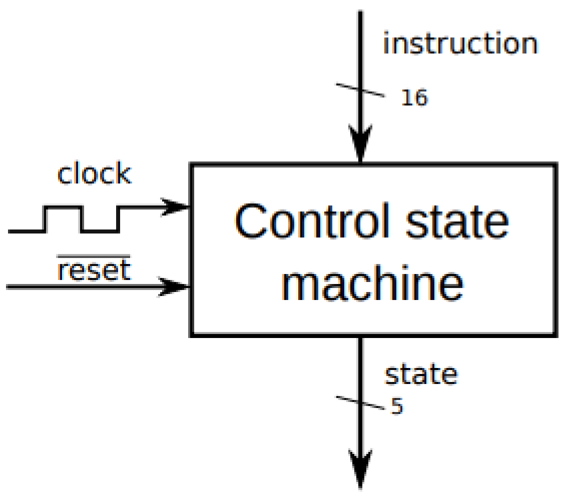
- ALU Latch:

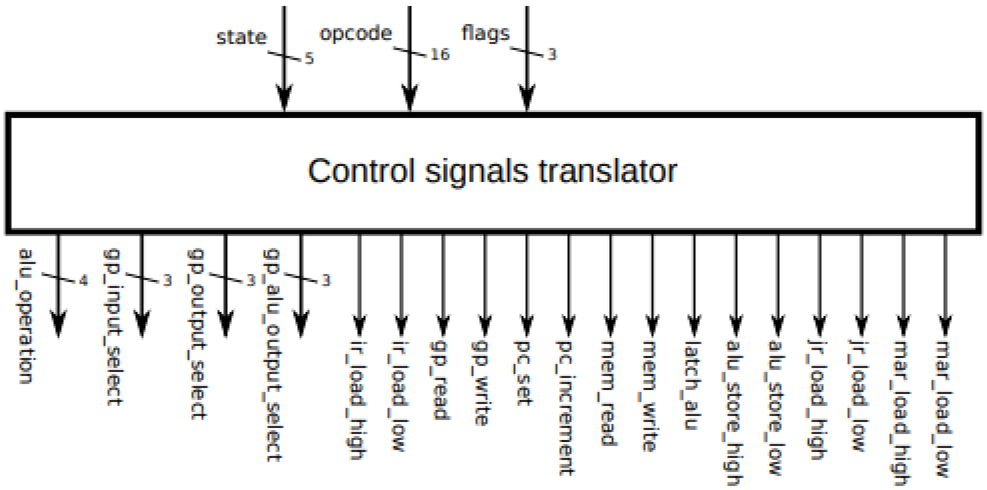
  The ALU latch grabs the result of the ALU operation, holds it, and then puts it on the databus when the store signals are asserted. It also latches the flags, so that a jump operates based on the last time the result was grabbed.



- Control module:

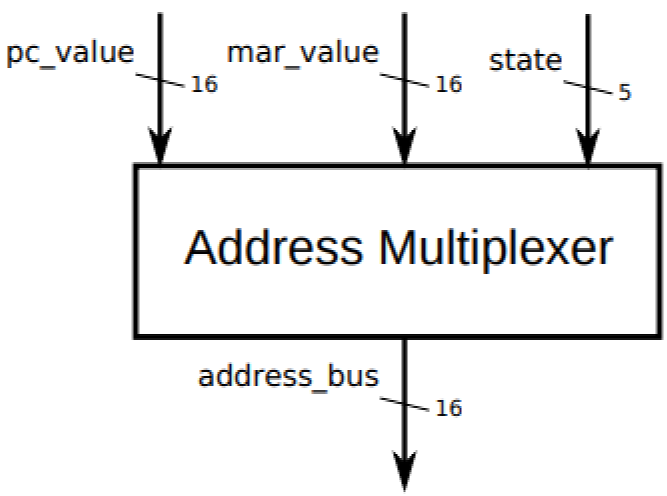
  The control module consists of two separate modules: a state machine which reads the output of the instruction register and determines what to do on the next clock cycle, and a signal translation module which maps the control state into controls signals for all other modules.





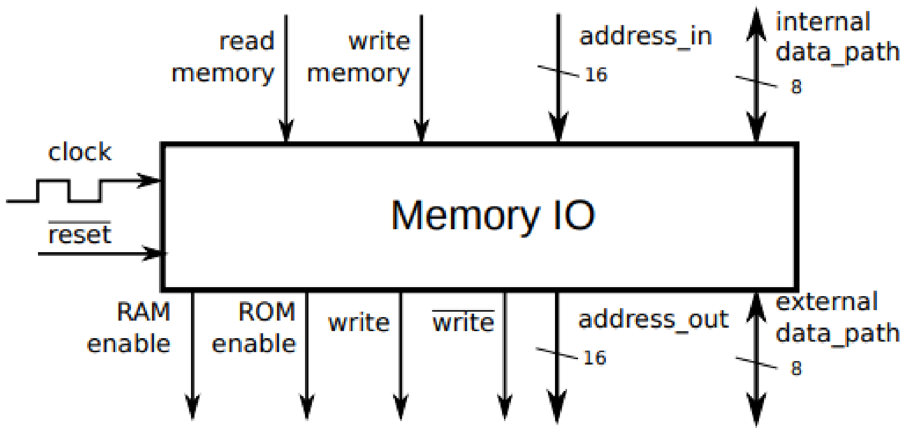
- Address multiplexer:

  The address multiplexer switches the output address between the program counter and memory address register based on the state.

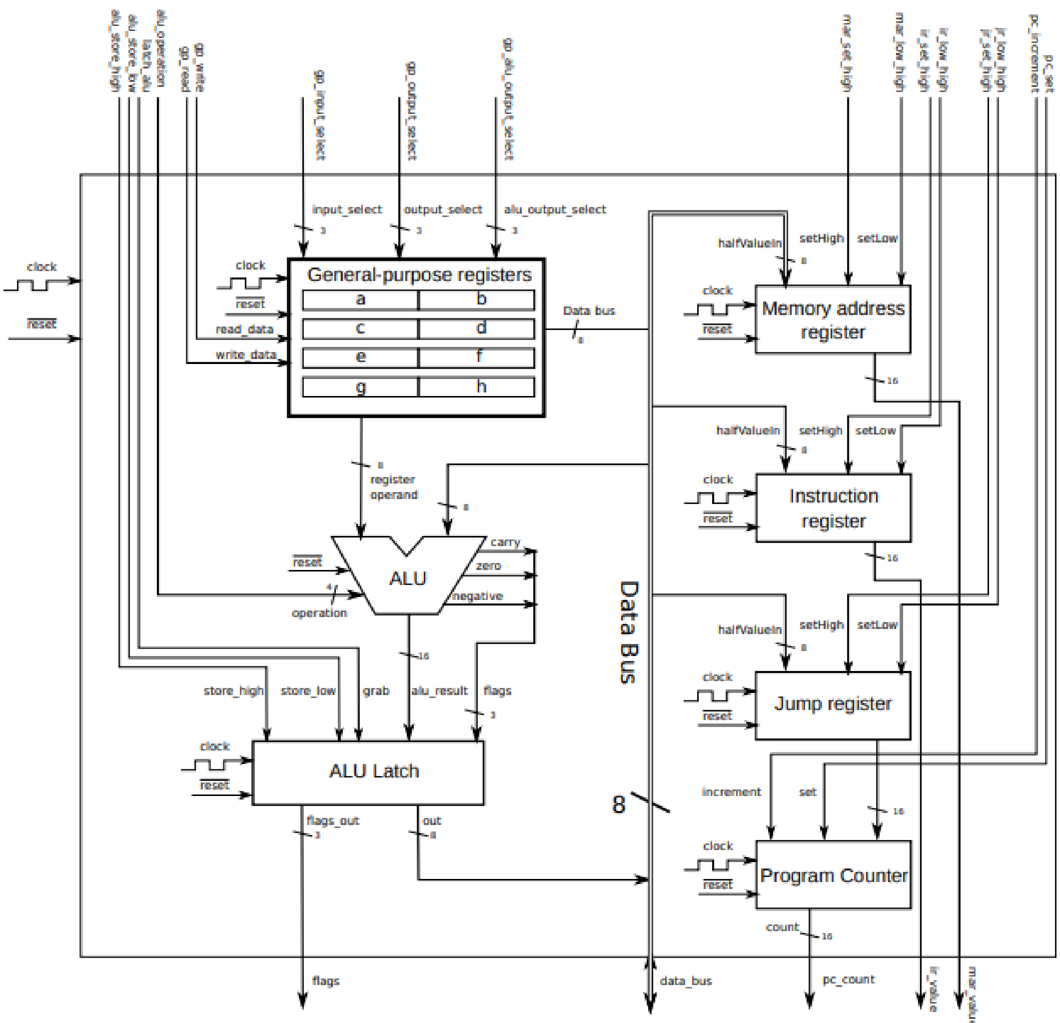


- Memory IO:

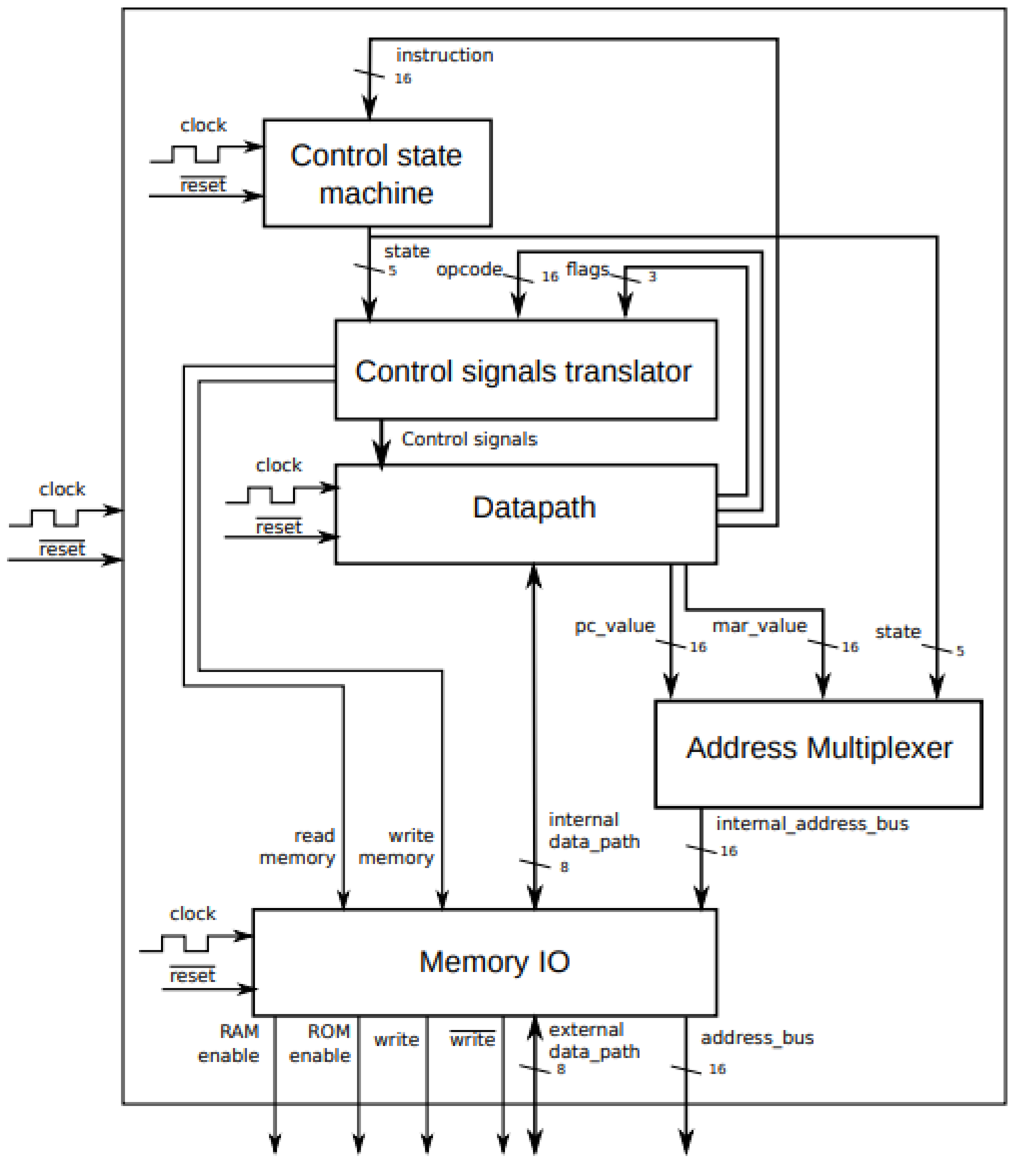
  The memory IO module performs memory mapping to locate the ROM and RAM in address space and translates the read and write signals into ROM and RAM chip enable signals.


read memory
write memory
address_in
16
internal data_path
8
clock
reset
Memory IO
RAM enable
ROM enable
write
write
address_out
16
external data_path
8

➢ Datapath


General-purpose registers
a
b
c
d
e
f
g
h
ALU
ALU Latch
Memory address register
Instruction register
Jump register
Program Counter
Data Bus
Data bus
8
clock
reset
input_select
output_select
alu_output_select
read_data
write_data
register operand
carry
zero
negative
operation
store_high
store_low
grab
alu_result
flags
flags_out
out
halfValueIn
setHigh
setLow
increment
set
count
data_bus
pc_count
ir_value
mar_value

➢ CPU Block



I have worked on the Control State Machine Module let us talk about it in details.

Control State Machine

Objective:

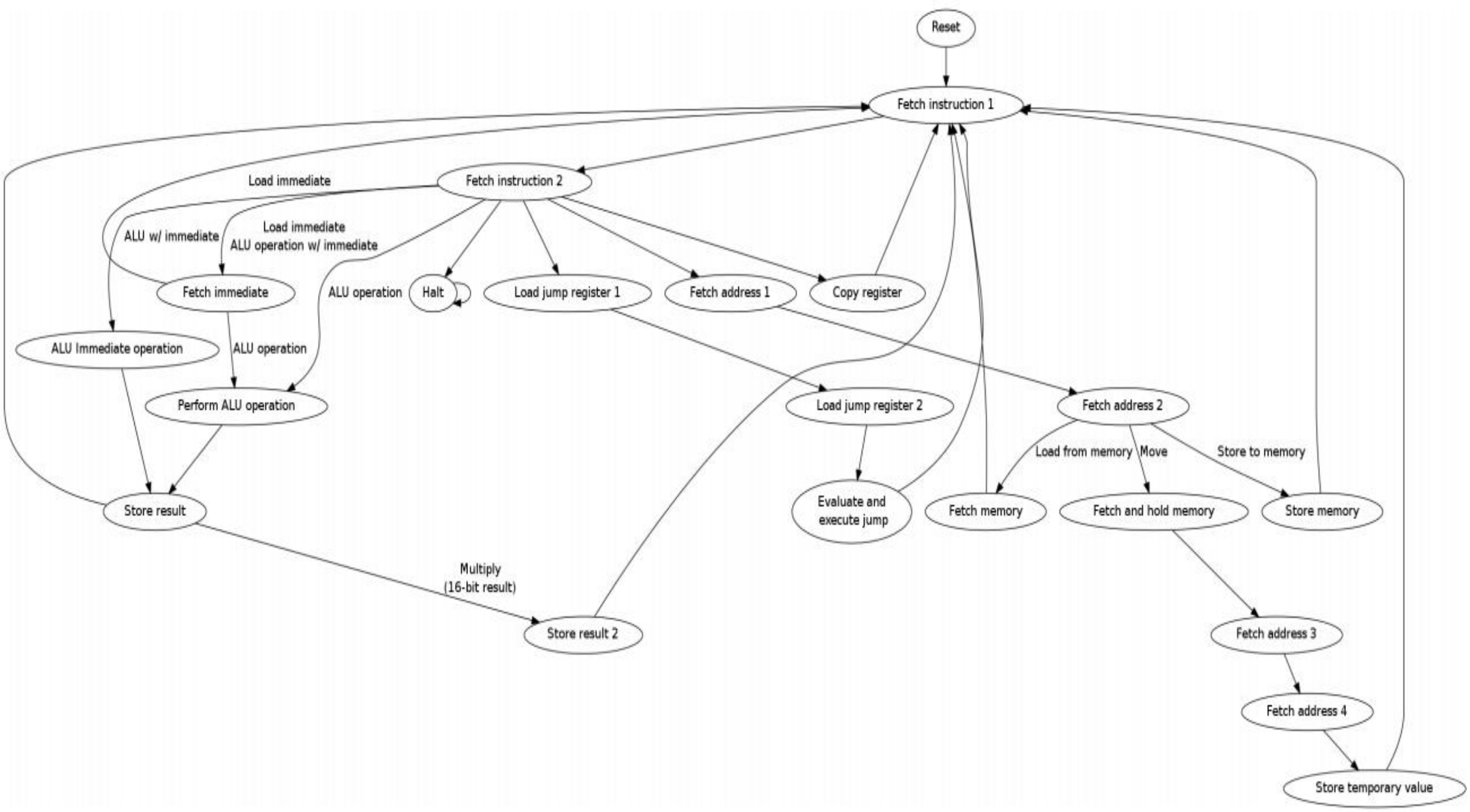
The Computer Executes Programs, the program is a sequence of instructions (Instruction cycle). Each Instruction is made up smaller units (SubCycles), Each subcycle has a number of Steps, each step has a number of micro-operations. Each step takes **one clock cycle** so we can call it the state of the processor.

The State Machine determines what is the next state, depend on the instruction register and the current state.

Control states:

State	Decimal representation	Binary Code
RESET	0	00000
FETCH_1	1	00001
FETCH_2	2	00010
ALU_OPERATION	3	00011
STORE_RESULT_1	4	00100
STORE_RESULT_2	5	00101
FETCH_IMMEDIATE	6	00110
COPY_REGISTER	7	00111
FETCH_ADDRESS_1	8	01000
FETCH_ADDRESS_2	9	01001
FETCH_MEMORY	10	01010
STORE_MEMORY	11	01011
TEMP_FETCH	12	01100
FETCH_ADDRESS_3	13	01101
FETCH_ADDRESS_4	14	01110
TEMP_STORE	15	01111
LOAD_JUMP_1	16	10000
LOAD_JUMP_2	17	10001
EXECUTE_JUMP	18	10010
HALT	19	10011
ALU_IMMEDIAT	20	10100

State Transition Diagram:



Generating the Next States:

In our processor the bits of Instruction register that affects the states are the opcode bits [15:11] and the source addressing mode bits [10:9], so every change in those 7 bits with respect to the current state will cause a different next state.

Each clock cycle has an Instruction Register input must achieve some distinguish conditions to move to another state.

I named those inputs IN(index) to facilitate Calculating the output equations.

I used decimal representation (D) for the states to reduce the table size.

CS(index) is Current states.

Q*(index) is the bit in Next states.

Current State(D)	Conditions	Input	Q4*	Q3*	Q2*	Q1*	Q0*	Next State(D)
0	reset		0	0	0	0	1	1
1	No Cond.		0	0	0	1	0	2
2	[15:11]==00000	IN0	0	0	0	0	1	1
2	[15:11]==01111	IN1	1	0	0	0	0	16
2	[15]==0&& [15:11]!=01101&& [15:11]!=01110&& [10:9]==00	IN2	0	0	0	1	1	3
2	[15]==0&& [15:11]!=01101&& [15:11]!=01110&& [10:9]!=00	IN3	1	0	1	0	0	20
2	[15:11]==10000&& [10:9]==00	IN4	0	0	1	1	1	7
2	[15:11]==10000&& [10:9]==10	IN5	0	0	1	1	0	6
2	[15:11]==10000&& [10:9]==01	IN6	0	1	0	0	0	8
2	[15:11]==10001|| [15:11]==10010	IN7	0	1	0	0	0	8
2	[15:11]==11111	IN8	1	0	0	1	1	19
2	Else	IN9	1	0	0	1	1	19
3	No Cond.		0	0	1	0	0	4
20	No Cond.		0	0	1	0	0	4
4	[15:11]==10011	IN10	0	0	1	0	1	5
4	Else	IN11	0	0	0	0	1	1
5	No Cond.		0	0	0	0	1	1
6	[15:11]==10000	IN12	0	0	0	0	1	1
6	Else	IN13	0	0	0	1	1	3
7	No Cond.		0	0	0	0	1	1
8	No Cond.		0	1	0	0	1	9
9	[15:11]==10000	IN14	0	1	0	1	0	10
9	[15:11]==10001	IN15	0	1	0	1	1	11
9	[15:11]==10010	IN16	0	1	1	0	0	12
10	No Cond.		0	0	0	0	1	1

11	No Cond.		0	0	0	0	1	1
12	No Cond.		0	1	1	0	1	13
13	No Cond.		0	1	1	1	0	14
14	No Cond.		0	1	1	1	1	15
15	No Cond.		0	0	0	0	1	1
16	No Cond.		1	0	0	0	1	17
17	No Cond.		1	0	0	1	0	18
18	No Cond.		0	0	0	0	1	1
19	No Cond.		1	0	0	1	1	19

With Calculating the Minterms of each output column:

Q0* = CS0+CS2(IN0+IN2+IN4+IN8+IN9)+CS4(IN10+IN11)+CS5+CS6(IN12+IN13)
+CS7+CS8+CS9(IN15)+CS10+CS11+CS12+CS14+CS15+CS16+CS18+CS19

Q1* = CS1+CS2(IN2+IN4+IN5+IN8+IN9)+CS6(IN13)+CS9(IN14+IN15)+CS13+CS14
+CS17+CS19

Q2* = CS2(IN3+IN4+IN5)+CS3+CS20+CS4(IN10)+CS9(IN16)+CS12+CS13+CS14

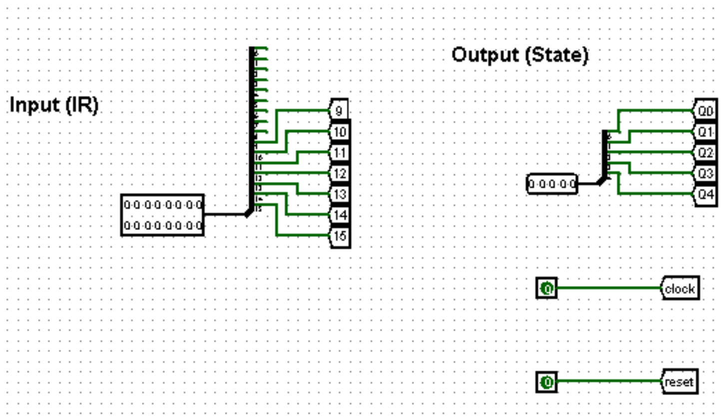
Q3* = CS2(IN6+IN7)+CS8+CS9(IN14+IN15+IN16)+CS12+CS13+CS14

Q4* = CS2(IN1+IN3+IN8+IN9)+CS16+CS17+CS19

Design Simulation:

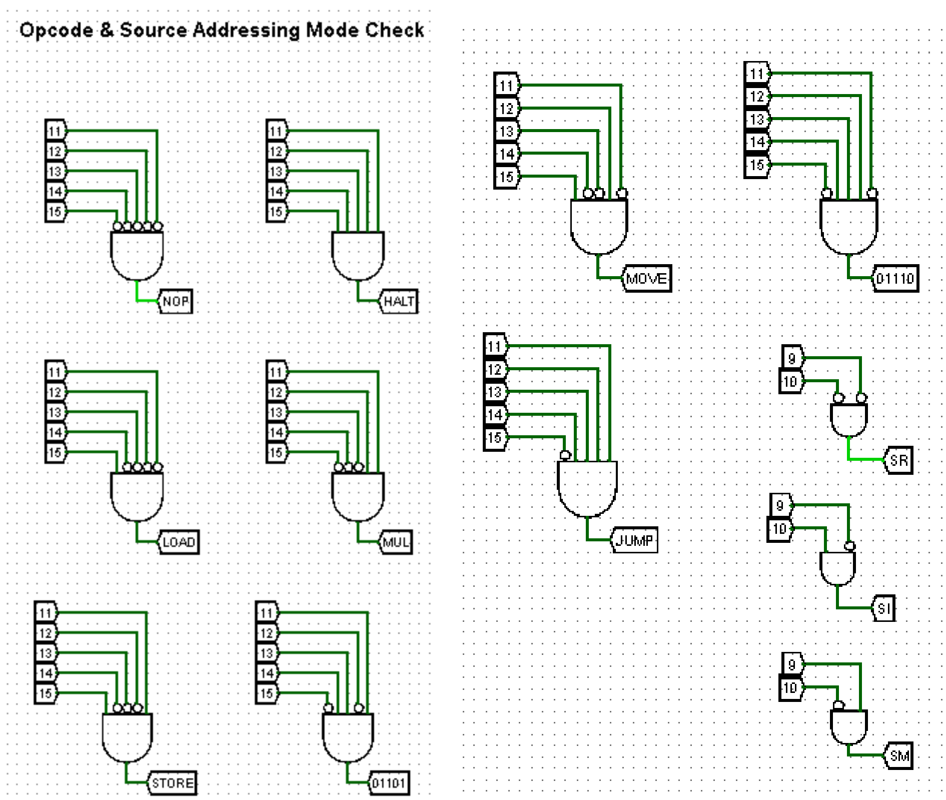
- Input & Output

  The module input is instruction register ,reset and clock pins. And the output is the state pins.



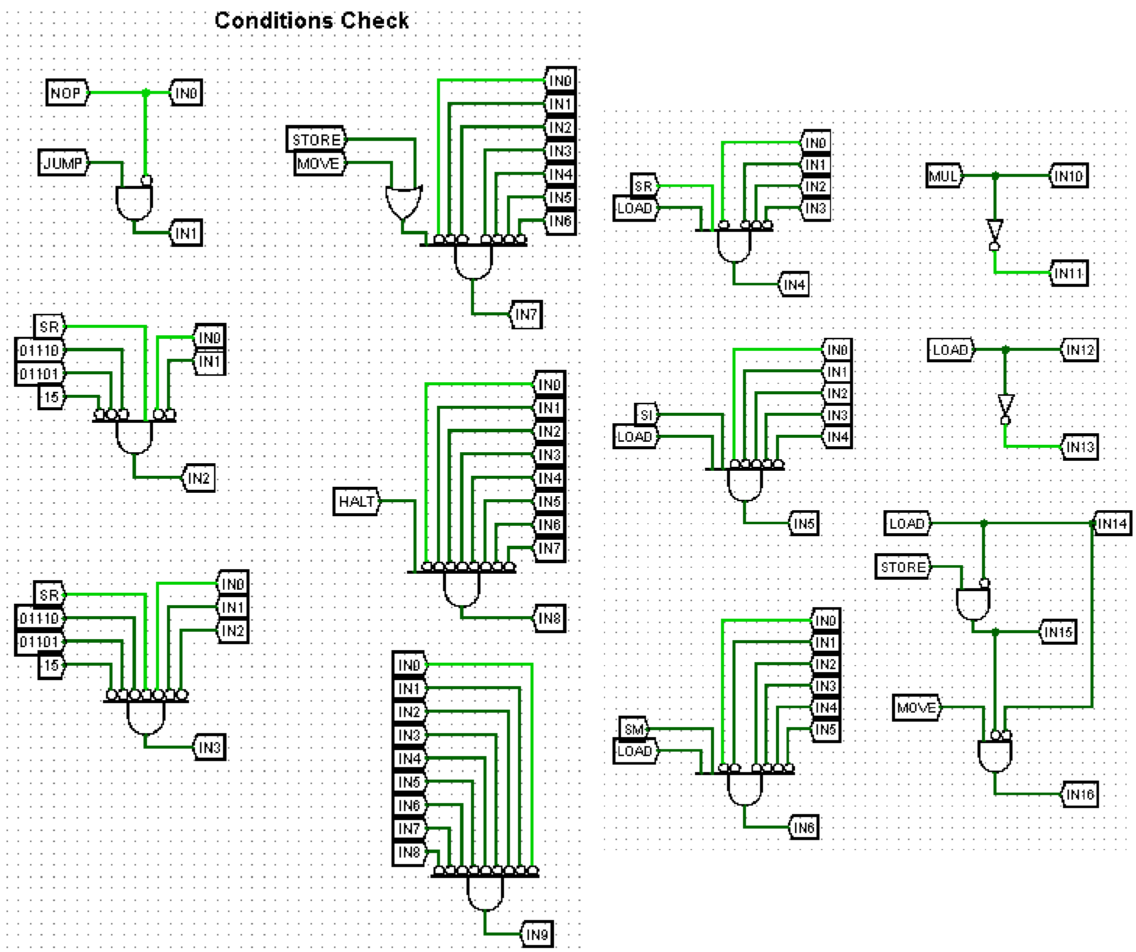
- Input Check

  What the type of instruction represented in the opcode, also what the type of source addressing mode.



- ## Conditions Check

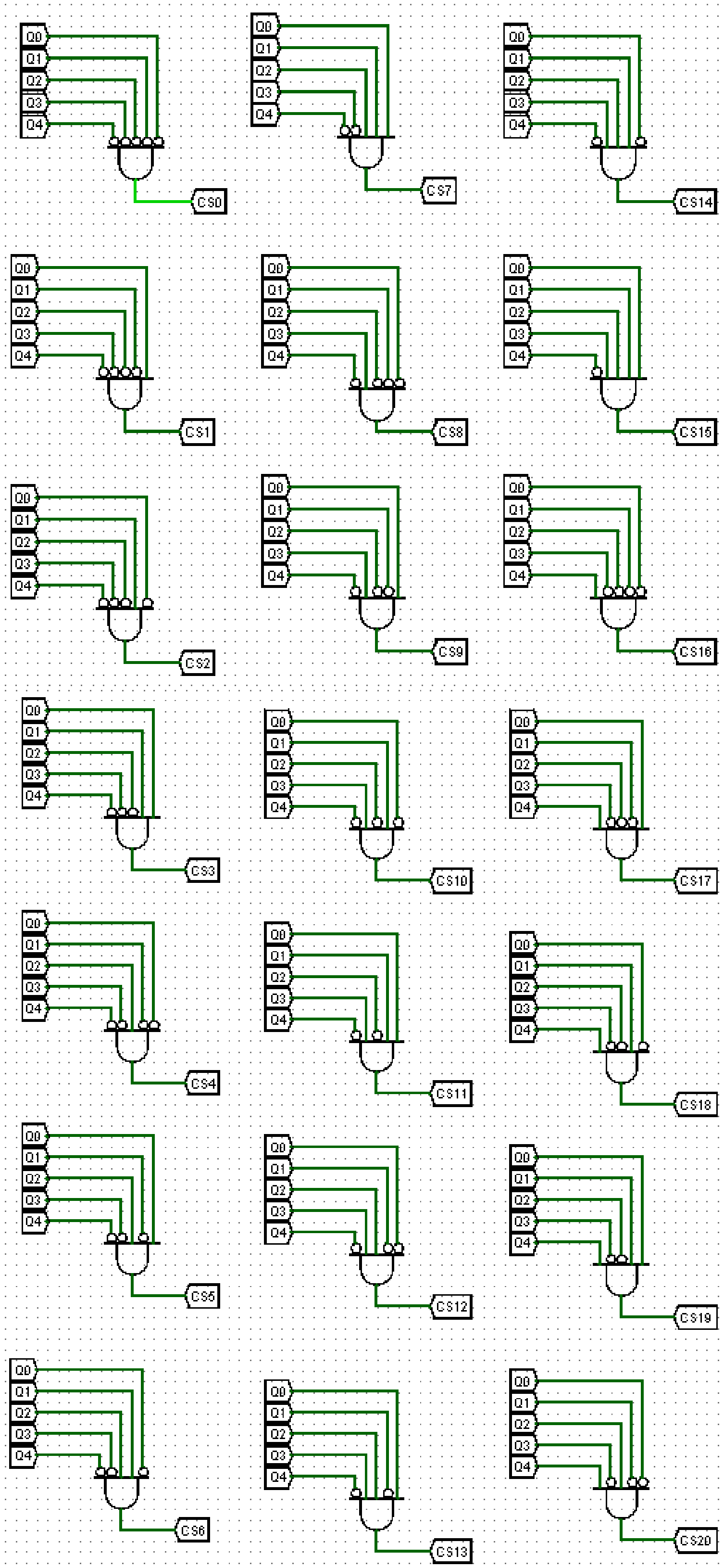
  To reach some states it is not enough to know the current state or IR, it must achieve a certain condition depend on input and the previous conditions also. Those conditions are what I named in the table IN[16:0].



- ## Current State Check

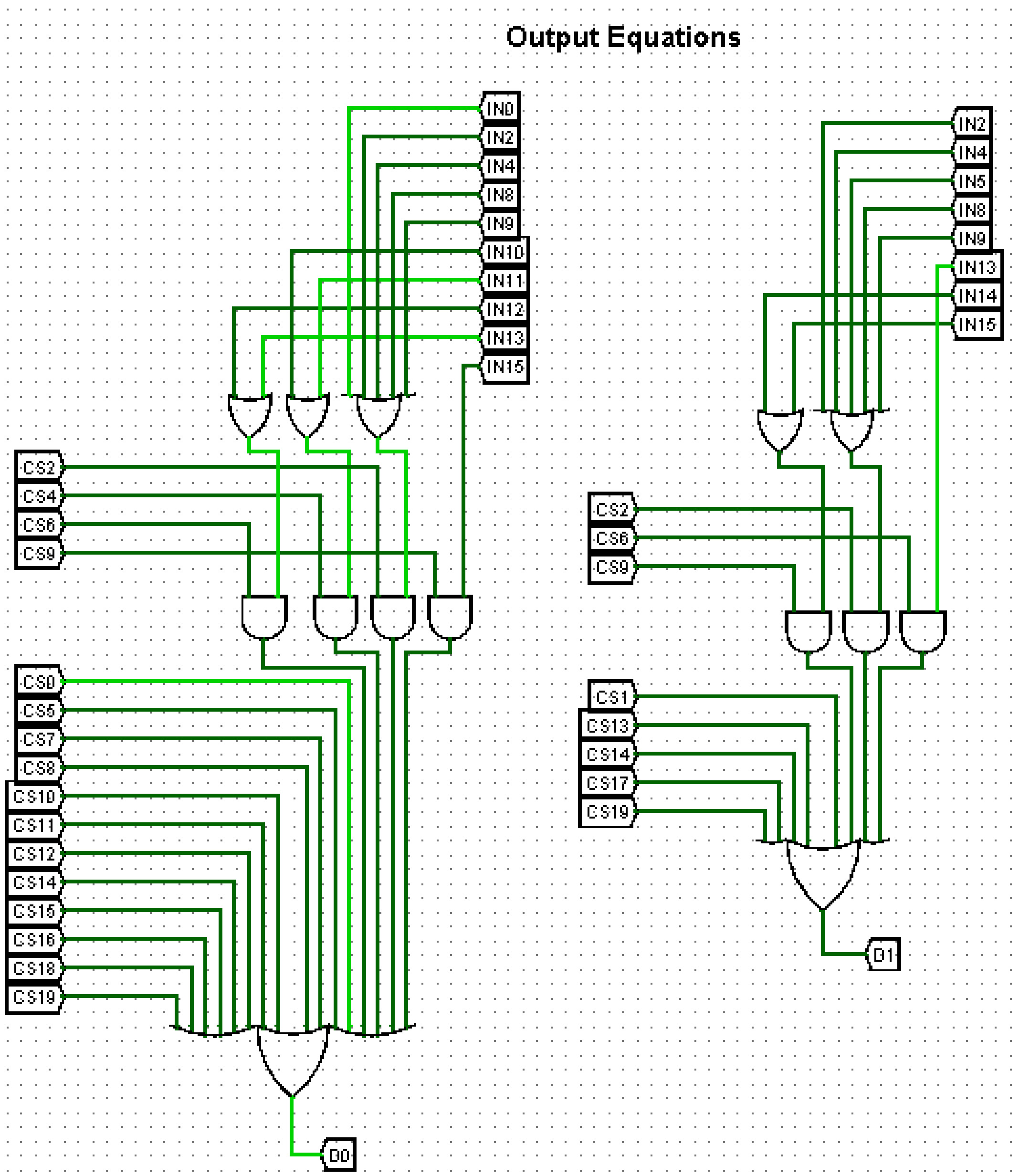
  Identify which state is the previous next state (the current state in this clock cycle).Q[4:0] is state bits and CS[20:0] is states.

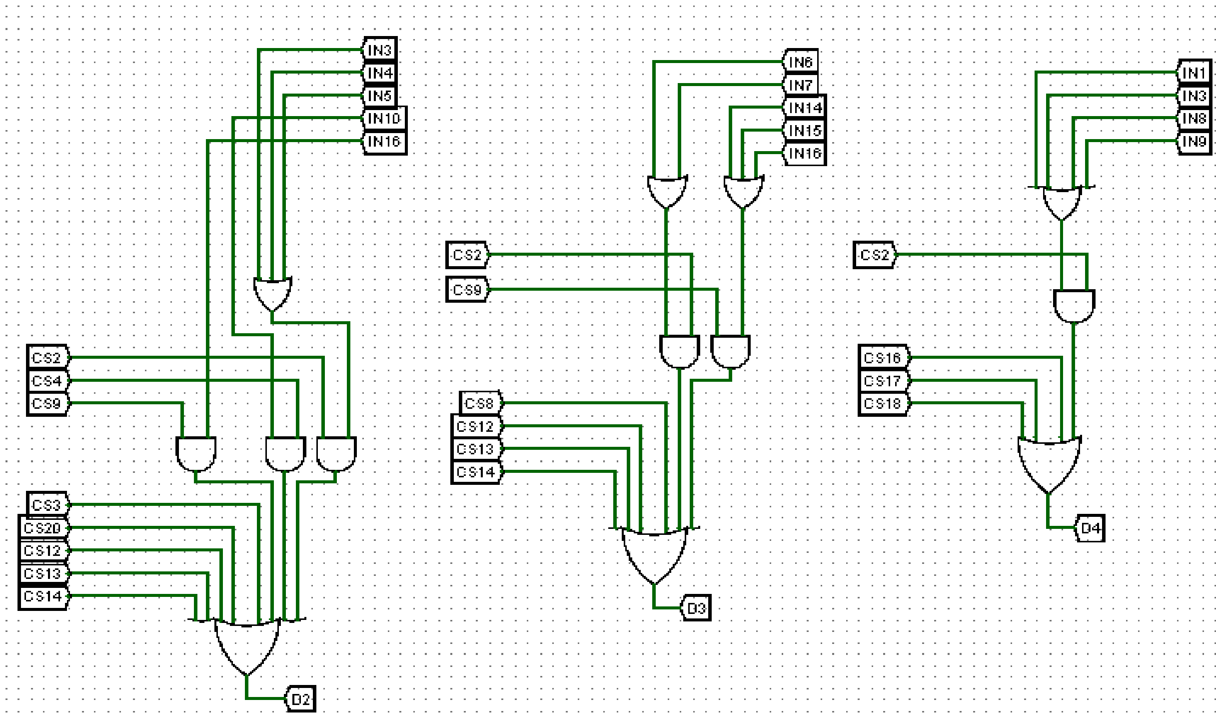
Current States



- Output Equations

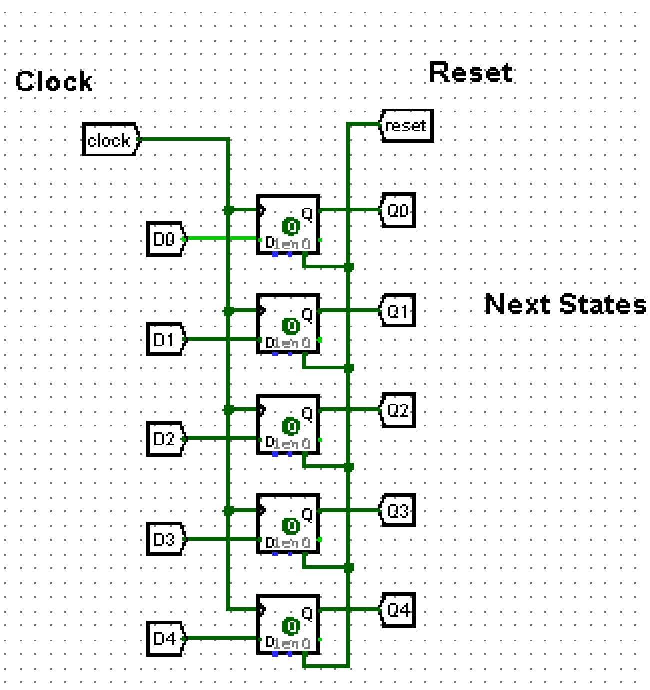
  I have used tunnels in the previous levels of the circuit to enable me to represent the output equations easier. Each equation is simulated in a collection of basic logic gates with inputs of IN[index]and CS[index] according to the table I have made before.





- Next States

To generate the next state on every clock cycle I have used D flipflops.



[Project Reference]()