King Fahd University of Petroleum & Minerals
College of Computer Sciences and Engineering
Information and Computer Science Department

ICS233 (Computer Arch. & Assembly Lang)

Term: 172

Single Cycle and Pipelined Design

Section: 52
Group #6

Instructor in Charge:
Mr. Saleh AlSaleh

Group Members:

Name	ID
Yousef Majeed	201568070
Ryan Gadhi	201532590
Abdulrahman Abutayli	201459620

Wednesday, May 2, 2018

"6th Team" ® - *"Success is a journey not a destination!"*

Table of Contents

1. Single Cycle Design 3

1.1 Component Description 3

1.1.1 Register File 3

1.1.2 Arithmetic and Logic Unit (ALU) 4

1.1.3 Main Control Unit 6

1.2 Testing Programs 11

1.2.1 Bubble Sort procedure. 11

1.2.2 Program that Counts the number of 1's in a register 11

1.2.3 Sequence of instructions to verify the correctness of ALL instructions. 12

2. Pipelined Design 14

2.1 Component Description 14

2.1.1 Main Control Unit (modified) 14

2.1.2 Hazard Detection Unit 17

2.1.3 PC Control Unit 18

2.2 Testing Programs 21

2.2.1 Independent Instructions 21

2.2.2 Dependent Instructions 22

2.2.3 Test dependent instructions including 'LW' 22

2.2.4 Program to test the branching logic 23

3. Teamwork 24

Single Cycle and Pipelined Design

1. Single Cycle Design

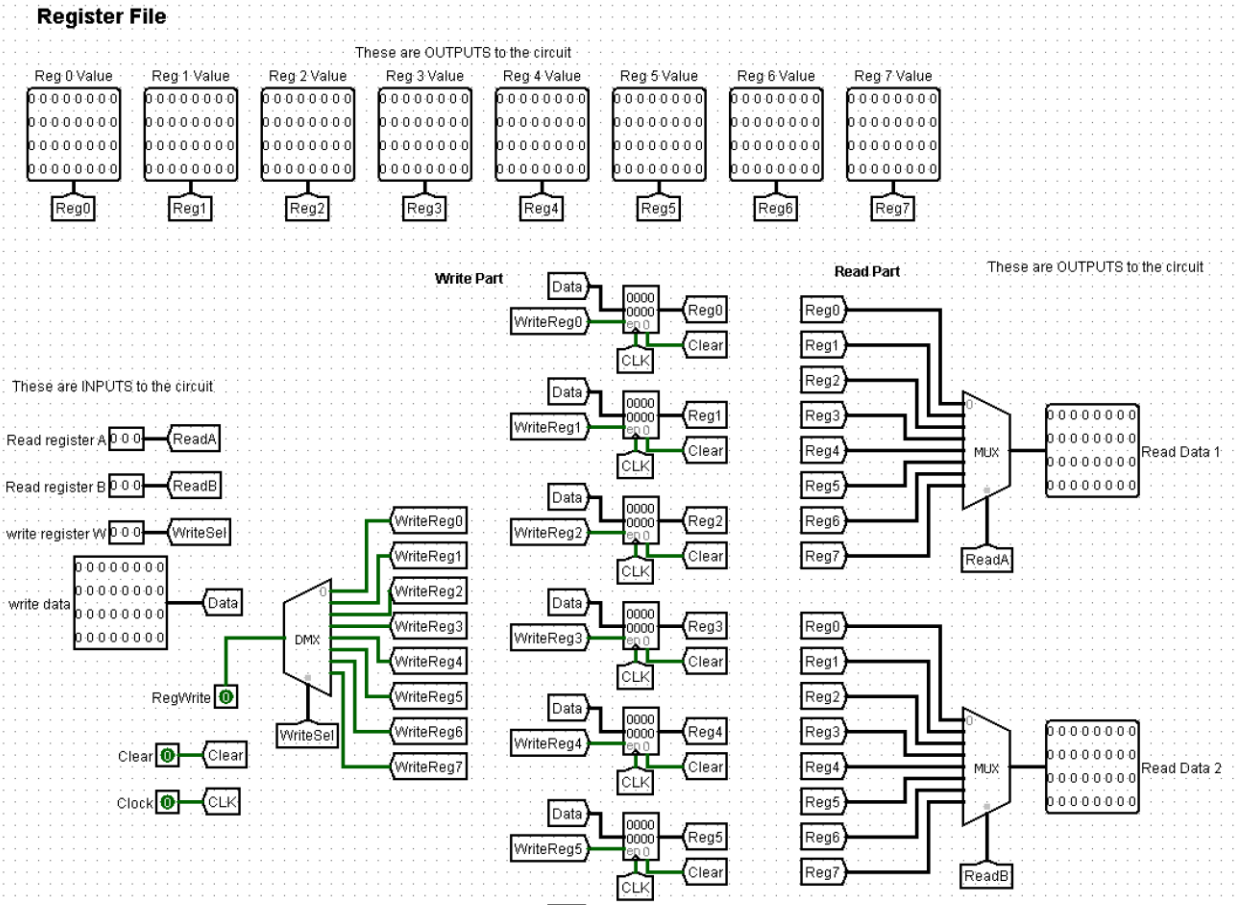
1.1 Component Description

1.1.1 Register File

- **Brief Description:** the register file is the component that is responsible for reading data from registers, writing, and updating the registers data. The decision of reading from or writing to a register is decided by the instruction that is being fetched. It can also read the old data and update the data from/to the same register in a single instruction.

- **Alternative Design:** our register file uses two MUXs (as shown below) to decide which data is being read for A and B. However, we can use a decoder, AND-Gates and OR-Gates to replace each MUX. The problem with alternative design is that it takes a lot of space and it is Inefficient way to design such this.

- **Input/Output:** It is shown in the circuit picture below.



1.1.2 Arithmetic and Logic Unit (ALU)

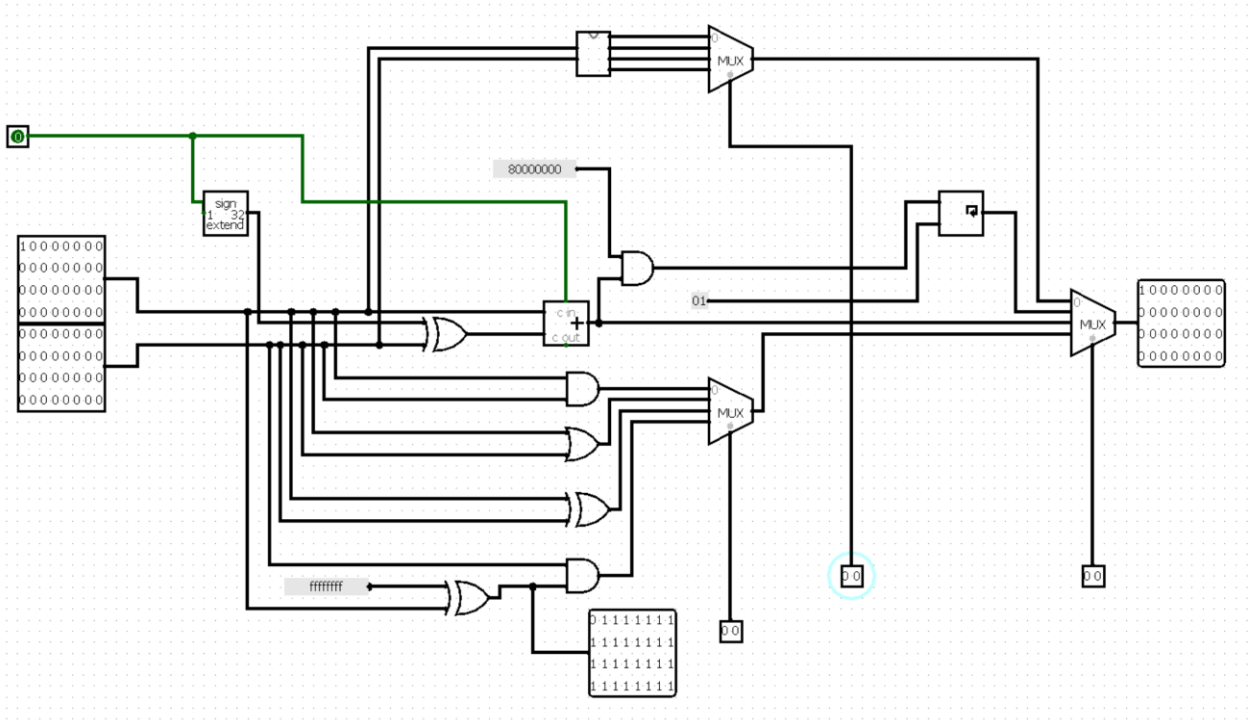
The Arithmetic and Logical Unit consists mainly of four parts:

1- Logical Gates
2- Shifters
3- Comparators
4- Multiplexers

- **Brief description:** The ALU receives the signals from the main control that determine the type of the instruction to be executed, namely: AND, CAND, OR, XOR, ADD, NADD, SLT and SLTU type of instruction. We used one large mux that has a fan in count of 16 to deliver the result of the execution of each operation to the mux. Using the ALUop - the select line for the Multiplexer- we connected the control signal directly with the ALUop to choose the required operation. We used an already designed shifters and comparator by the Logisim library.

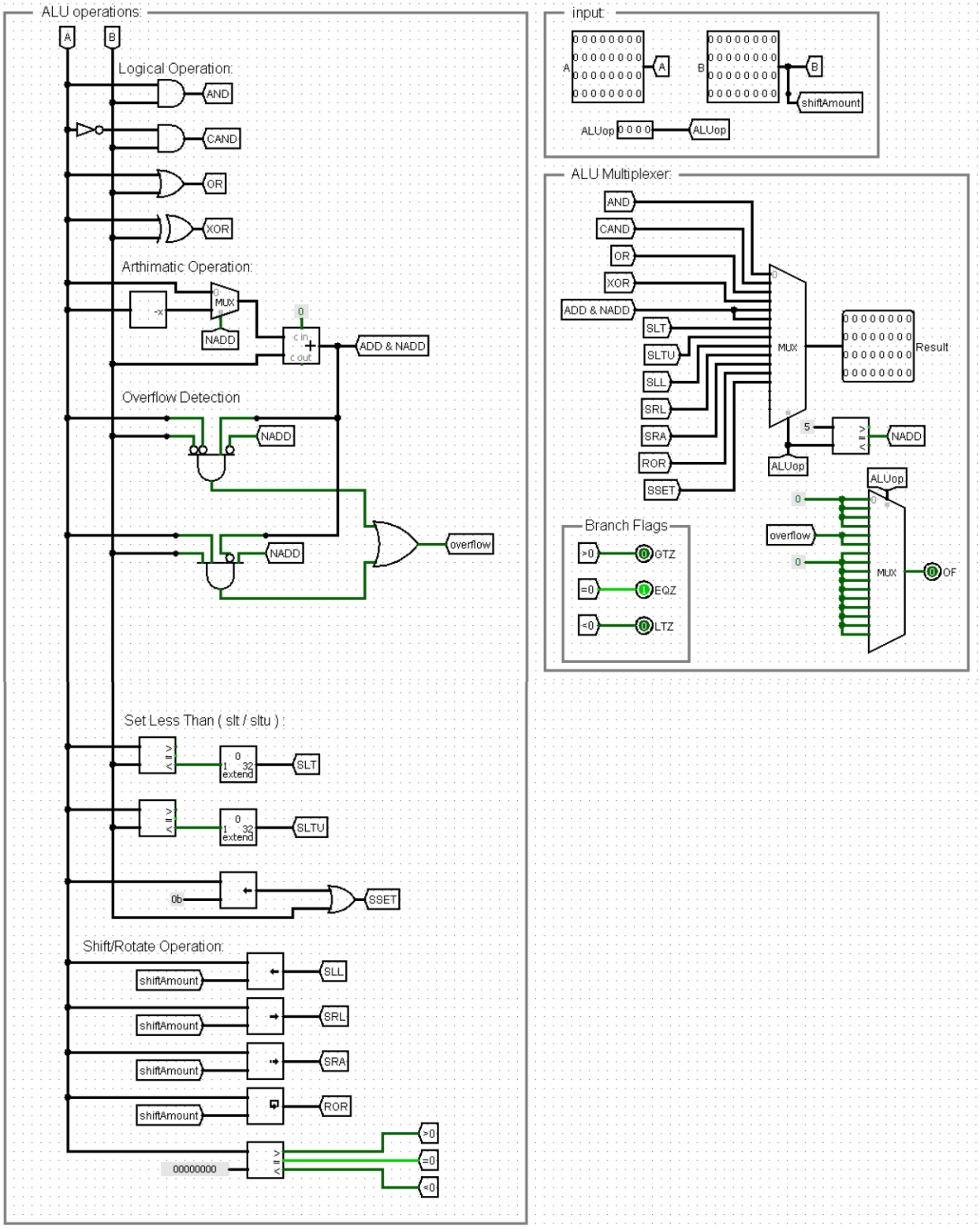
- **Alternative Design:** Our design can be improved and be more realistic if we reduce the number of the fan in of the multiplexer. We introduce this design in the figure below. We worked on this part shortly, but it was not considered because of some issues in the design.

***Screenshot of the alternative design of the ALU**



*Please note that most of the ALU instructions work as expected by this design. Some instructions that may output inaccurate results such (in SLT for example) are due to errors in the logic behind their design. Thus, we depended on the predefined Logisim circuits and used them to solve these issues.

- **Input/Output:** It is shown in the circuit picture below.

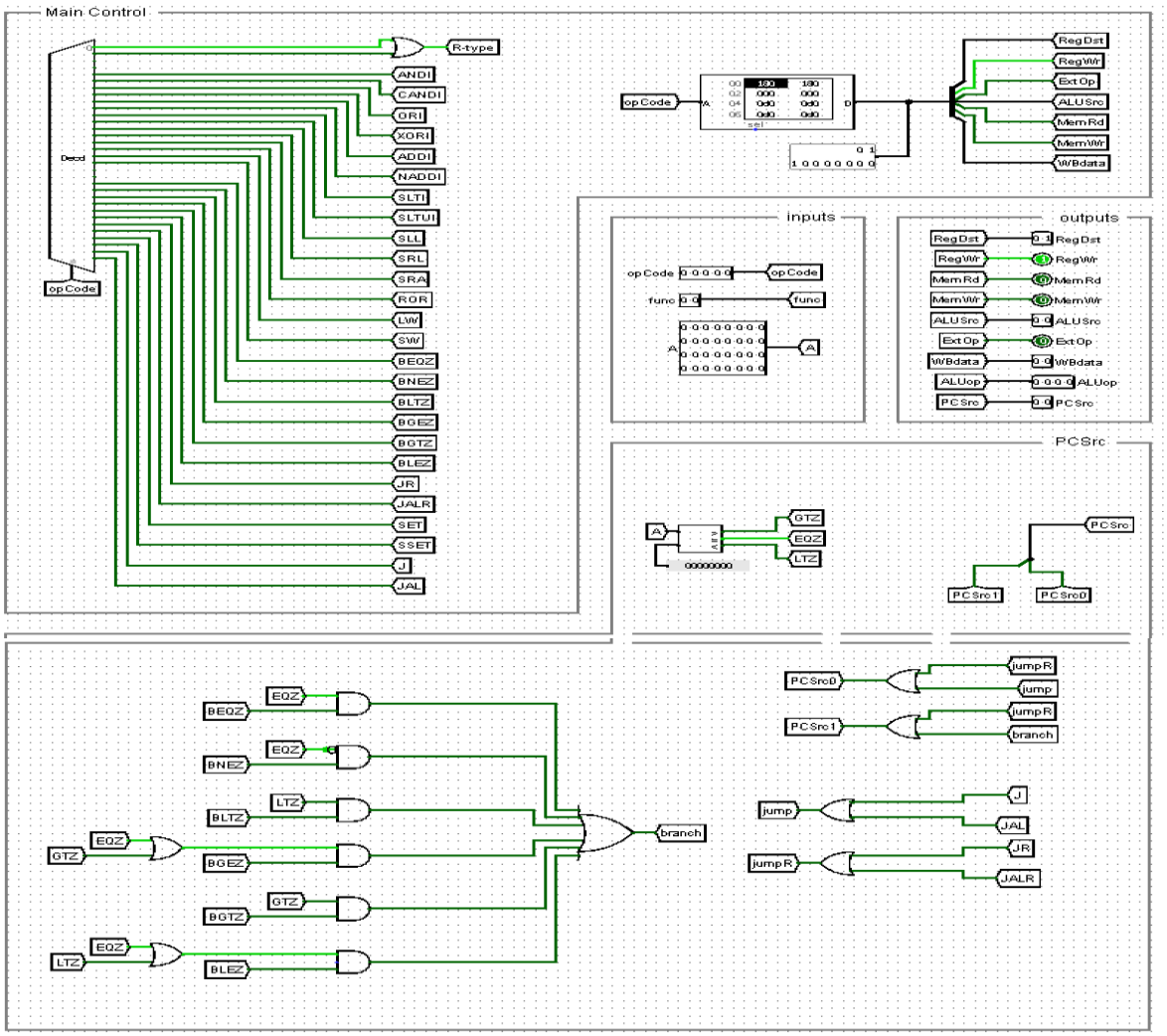


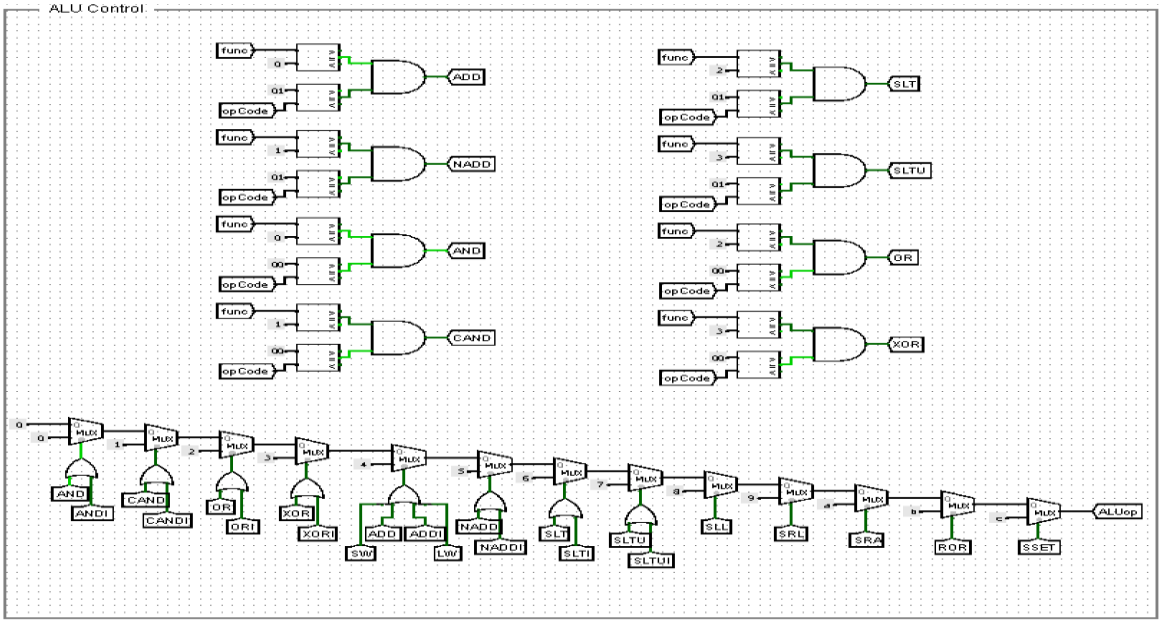
1.1.3 Main Control Unit

- **Brief Description:** the main control unit is the unit that will generate all the signals that our single cycle CPU need to control the main component of the CPU, such as Register File needs a signal that determine where the data should be written in register and needs also a write enable, ALU needs a signal that will determine what operation should be used for specific instruction, Data Memory needs signal that will make the memory writable for SW instruction and readable for LW instruction, Next PC needs a signal that will determine which kind of Next PC (Incremented PC, Jump, Branch, …) should be used for a given instruction, Finally the Write Back Stage needs a signal that will determine which data output should be written to the given register (ALU output, Data Memory output, return address, …).

- **Alternative Design:** we could use a lot of comparators to generate the instruction from Opcode instead of using a decoder and we have used a decoder because it will make a lot of space in the circuit and it is more efficient. Also, we could use logic equations to generate the signal instead of using a ROM, and we have used a ROM because it is a very fabulous idea to implement and too easy to use compare to the logic equations which will be a little hard to obtain. (Part of the alternative design is shown below with the suggested design).

- **Input/Output:** It is shown in the circuit picture below.





- **Control Tables:** Main Control Signals, ALU Control Table, Main Control Table shown below.

Main Control Signals

Signal	Effect when '0'	Effect when '1'	Effect when '2'	Effect when '3'
RegDst	Destination register = Rt	Destination register = Rd	Destination register = R7	Destination register = R0
RegWr	No register is written	Destination register (Rt, Rd, R7, or R0) is written with the data on BusW	-	-
ExtOp	Immediate is zero-extended	Immediate is sign-extended	-	-
ALUSrc	Second ALU operand is the value of register Rt that appears on BusB	Second ALU operand is the value of the extended 5-bit immediate	Second ALU operand is the value of the extended 11-bit immediate	Second ALU operand is the value of the extended 11-bit immediate
MemRd	Data memory is NOT read	Data memory is read Data_out ← Memory[address]	-	-
MemWr	Data Memory is NOT written	Data memory is written Memory[address] ← Data_in	Data Memory is NOT written	Data Memory is NOT written
WBdata	BusW = ALU result	BusW = Data_out from Memory	BusW = PC + 1 (Return Address)	BusW = value of the extended 11-bit immediate

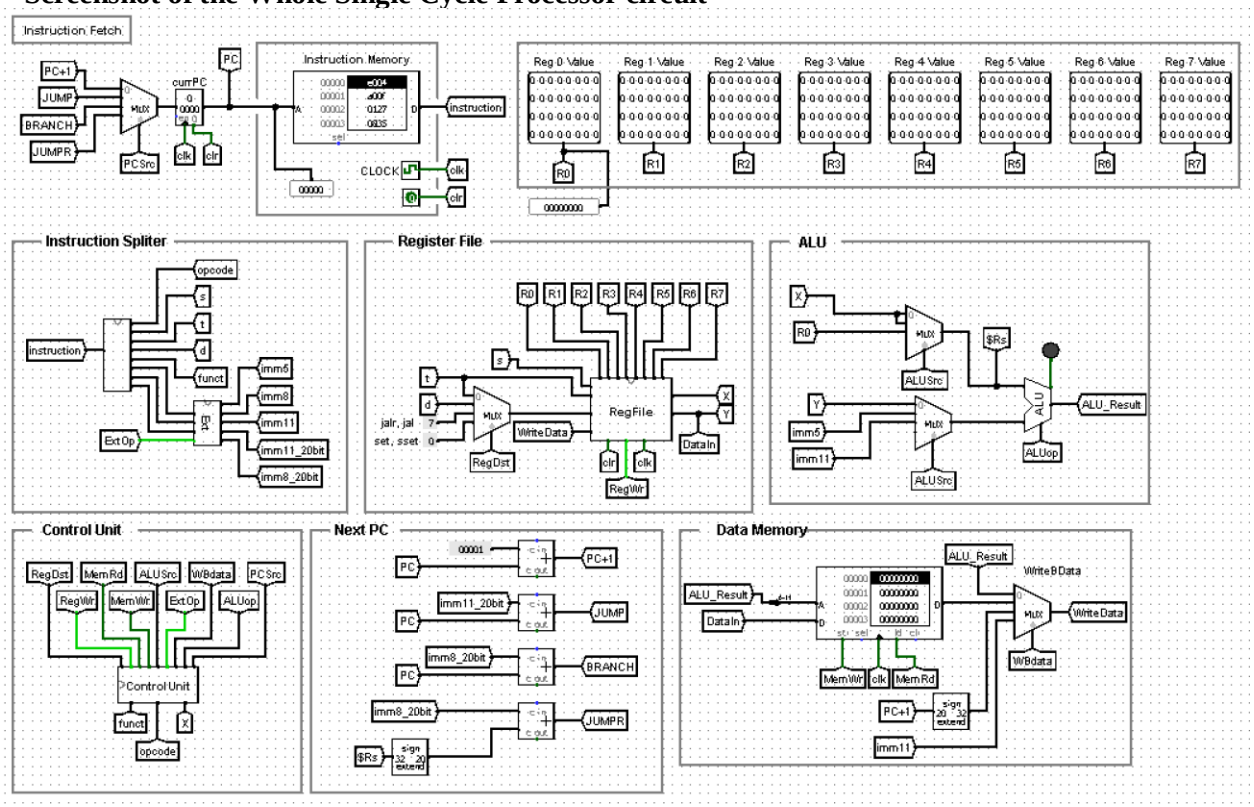
ALU Control Truth Table

Op	Op (dec)	funct	funct (dec)	ALUOp	4-bit Coding	ALUOp (dec)
R-type	0	AND	0	AND	0000	0
R-type	0	CAND	1	CAND	0001	1
R-type	0	OR	2	OR	0010	2
R-type	0	XOR	3	XOR	0011	3
R-type	1	ADD	0	ADD	0100	4
R-type	1	NADD	1	NADD	0101	5
R-type	1	SLT	2	SLT	0110	6
R-type	1	SLTU	3	SLTU	0111	7
ANDI	4	X	X	AND	0000	0
CANDI	5	X	X	CAND	0001	1
ORI	6	X	X	OR	0010	2
XORI	7	X	X	XOR	0011	3
ADDI	8	X	X	ADD	0100	4
NADDI	9	X	X	NADD	0101	5
SLTI	10	X	X	SLT	0110	6
SLTUI	11	X	X	SLTU	0111	7
SLL	12	X	X	SLL	1000	8
SRL	13	X	X	SRL	1001	9
SRA	14	X	X	SRA	1010	10
ROR	15	X	X	ROR	1011	11
LW	16	X	X	ADD	0100	4
SW	17	X	X	ADD	0100	4
SSET	29	X	X	SSET	1100	12

Main Control Truth Table

Op	Op (dec)	RegDst	RegWr	ExtOp	ALUSrc	MemRd	MemWr	WBdata	ALUop	Binary	Hex
R-type	0 OR 1	01 = Rd	1	X	00 = BusB	0	0	00 = ALU	0 – 7	0110000000	180
ANDI	4	00 = Rt	1	1 = sign	01 = imm5	0	0	00 = ALU	0	0011010000	0d0
CANDI	5	00 = Rt	1	1 = sign	01 = imm5	0	0	00 = ALU	1	0011010000	0d0
ORI	6	00 = Rt	1	1 = sign	01 = imm5	0	0	00 = ALU	2	0011010000	0d0
XORI	7	00 = Rt	1	1 = sign	01 = imm5	0	0	00 = ALU	3	0011010000	0d0
ADDI	8	00 = Rt	1	1 = sign	01 = imm5	0	0	00 = ALU	4	0011010000	0d0
NADDI	9	00 = Rt	1	1 = sign	01 = imm5	0	0	00 = ALU	5	0011010000	0d0
SLTI	10	00 = Rt	1	1 = sign	01 = Imm5	0	0	00 = ALU	6	0011010000	0d0
SLTUI	11	00 = Rt	1	1 = sign	01 = Imm5	0	0	00 = ALU	7	0011010000	0d0
SLL	12	00 = Rt	1	0 = zero	01 = imm5	0	0	00 = ALU	8	0010010000	090
SRL	13	00 = Rt	1	0 = zero	01 = imm5	0	0	00 = ALU	9	0010010000	090
SRA	14	00 = Rt	1	0 = zero	01 = imm5	0	0	00 = ALU	10	0010010000	090
ROR	15	00 = Rt	1	0 = zero	01 = imm5	0	0	00 = ALU	11	0010010000	090
LW	16	00 = Rt	1	1 = sign	01 = Imm5	1	0	01 = Mem	4	0011011001	0d9
SW	17	X	0	1 = sign	01 = Imm5	0	1	X	4	0001010100	054
BEQZ	20	X	0	1 = sign	X	0	0	X	X	0001000000	040
BNEZ	21	X	0	1 = sign	X	0	0	X	X	0001000000	040
BLTZ	22	X	0	1 = sign	X	0	0	X	X	0001000000	040
BGEZ	23	X	0	1 = sign	X	0	0	X	X	0001000000	040
BGTZ	24	X	0	1 = sign	X	0	0	X	X	0001000000	040
BLEZ	25	X	0	1 = sign	X	0	0	X	X	0001000000	040
JR	26	X	0	1 = sign	X	0	0	X	X	0001000000	040
JALR	27	10 = $7	1	1 = sign	X	0	0	10 = PC+1	X	1011000010	2c2
SET	28	11 = $0	1	1 = sign	X	0	0	11 = Imm11	X	1111000011	3c3
SSET	29	11 = $0	1	0 = zero	10 = R0 <<	0	0	00 = ALU	12	1110100000	3a0
J	30	X	0	1 = sign	X	0	0	X	X	0001000000	040
JAL	31	10 = $7	1	1 = sign	X	0	0	10 = PC+1	X	1011000010	2c2

*** Screenshot of the Whole Single Cycle Processor circuit**



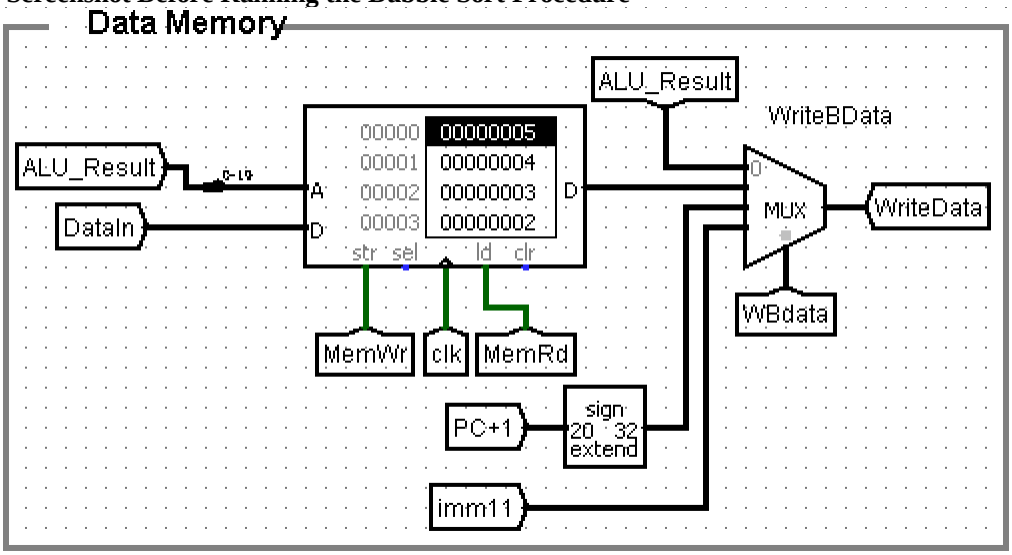
- **Brief Description:** as shown above, this is the whole single cycle processor that will do the complete functionalities of our Project Requirement in a proper way. First the instruction will be fetched using a ROM then the it will be decoding to its main parts that will run our CPU starting from the Register File Stage and finishing with the Write Back Data Stage in one clock cycle only. And we used a MUX before the Register File Stage to determine what type of instruction that will be executed in this Clock Cycle, and there is another two MUXs placed before the ALU stage to determine what should type of data be passed inside the ALU Unit and the opcode and the generated signal will determine that. Also, there is a MUXs after the Data Memory Stage that will determine which kind of data should be written back in the Write Register after passing all the stages and the opcode and the generating signals will determine that also. Finally, there a unit that will calculate the Next PC based on what instruction is executing at the same time and there a MUX before the instruction Fetching-ROM Memory that will determine which kind of Address or Next PC should be passed in the next Clock Cycle.

1.2 Testing Programs

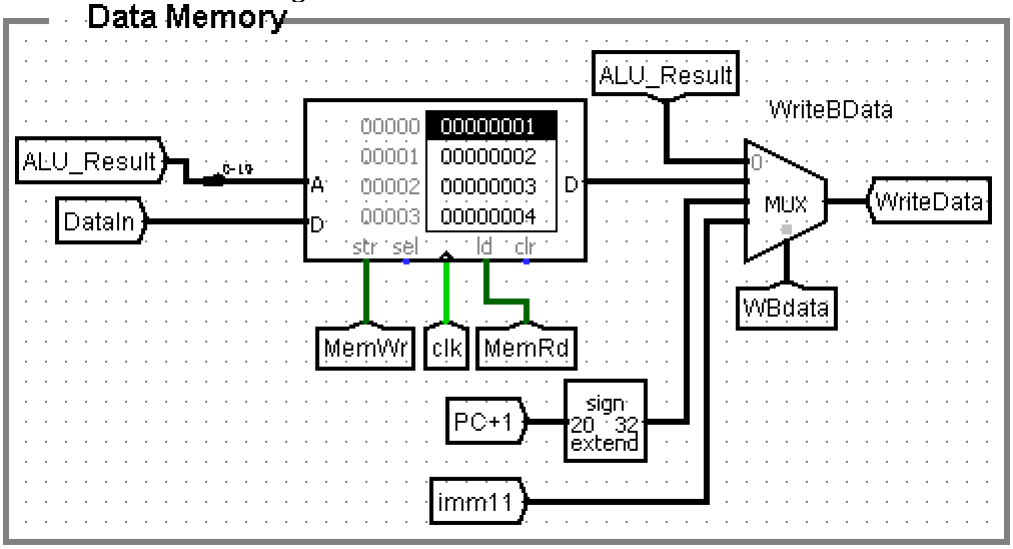
1.2.1 Bubble Sort procedure.

The number of comparisons will be stored in R0 which is 5 elements, and the array content will be placed in the data memory in a random order starting from index zero until index 4, the output will be an ascending sorted data array in the data memory. The picture below will illustrate the data memory content before running the procedure and after it.

***Screenshot Before Running the Bubble Sort Procedure**



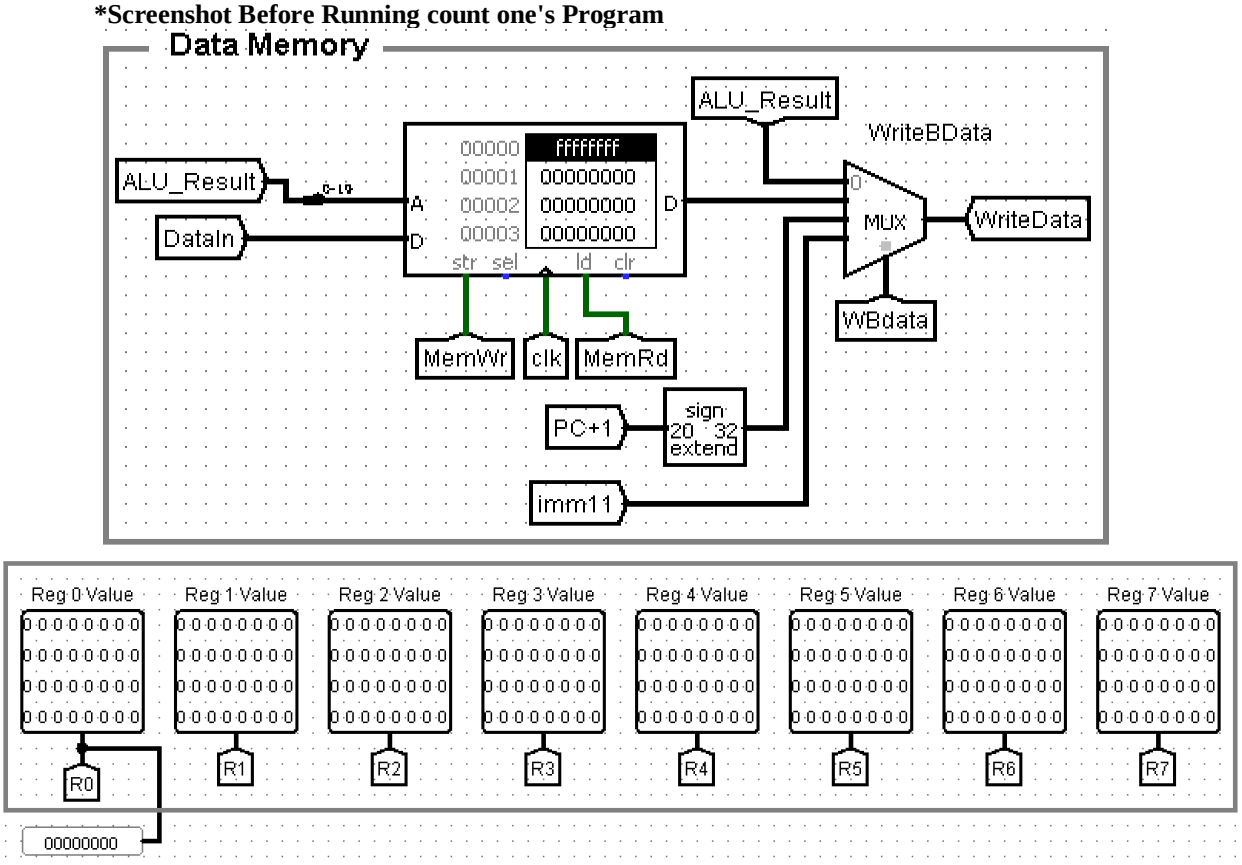
***Screenshot After Running the Bubble Sort Procedure**



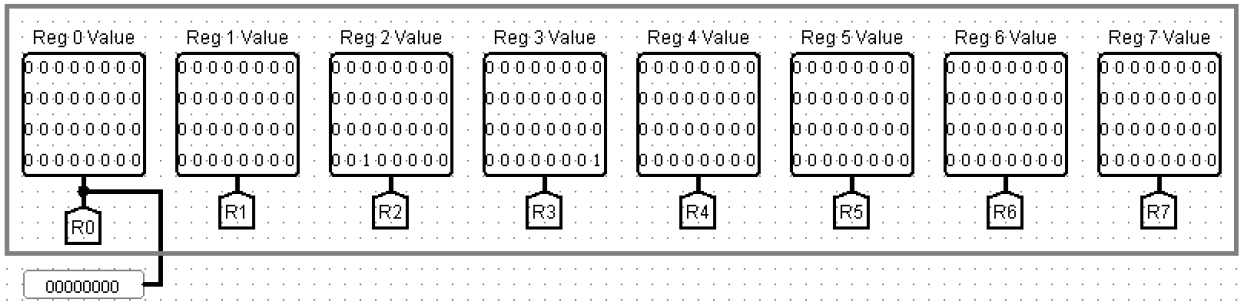
1.2.2 Program that Counts the number of 1's in a register

This program will take the input number that we want to count its ones from the data memory at index or location zero and will copy it to register 1 then will count the ones and put the count in register 2. The picture below will illustrate the registers content before running the program and after it.

***Screenshot Before Running count one's Program**



***Screenshot After Running count one's Program**



1.2.3 Sequence of instructions to verify the correctness of ALL instructions.

*The Register $0 will be set to 7.

Assembly	Binary	Hex	Comment/Effect
AND $0, $0, $0	0000_0000_0000_0000	0000	Anding register $0 with itself and store the result in register $0, $0 = $0 & $0;
CAND $1, $0, $1	0000_0000_0010_0101	0025	Complement And $0 with $1 and store the result in $1, $1 = ~$0 & $1
OR $1, $1, $0	0000_0001_0000_0110	0106	Oring Register $1 with $0 and store the result in $1, $1 = $1 | $0
XOR $1, $1, $1	0000_0001_0010_0111	0127	Xoring register $1 with itself and store the result in register $1, $1 = $1 ^ $1;
ADD $2, $0, $1	0000_1000_0010_1000	0828	Adds register $0 to $1 and store the result in register $2, $2 = $0 + $1

NADD $3, $2, $1	0000_1010_0010_1101	0a2d	Negate $2 then Adds it to $1 and store the result in register $3, $3 = -$2 + $1
SLT $4, $2, $1	0000_1010_0011_0010	0a32	Check signed if $2 is less than $1, if it is, then put 1 in $4, else put 0.
SLTU $4, $1, $3	0000_1001_0111_0011	0973	Check unsigned if $1 is less than $3, if it is, then put 1 in $4, else put 0.
ANDI $4, $0, 5	0010_0000_1000_0101	2085	Anding register $0 with 5 and store the result in register $4, $4 = $0 & 5;
CANDI $3, $1, 3	0010_1001_0110_0011	2963	Complement And $1 with 3 and store the result in $3, $3 = ~$1 & 3
ORI $5, $5, 15	0011_0101_1010_1111	35af	Oring Register $5 with 15 and store the result in $5, $5 = $5 | 15
XORI $4, $4, 1	0011_1100_1000_0001	3c81	Xoring register $4 with 1 and store the result in register $4, $4 = $4 ^ 1;
ADDI $1, $1, 8	0100_0001_0010_1000	4128	Adds register $1 to 8 and store the result in register $1, $1 = $1 + 8
NADDI $0, $0, 10	0100_1000_0000_1010	480a	Negate $0 then Adds it to 10 and store the result in register $0, $0 = -$0 + 10
SLTI $4, $0, 2	0101_0000_1000_0010	5082	Check signed if $0 is less than 2, if it is, then put 1 in $4, else put 0.
SLTUI $4, $0, 5	0101_1000_1000_0101	5885	Check unsigned if $0 is less than 5, if it is, then put 1 in $4, else put 0.
SLL $3, $0, 2	0110_0000_0110_0010	6062	Shift register $0 to left by 2-bit and store the result in $3, $3 = $0 << 2
SRL $4, $0, 2	0110_1000_1000_0010	6882	Shift register $0 to right by 2-bit and store the result in $4, $4 = $0 zero >> 2
SRA $3, $0, 2	0111_0000_0110_0010	7062	Shift register $0 to right by 2-bit, extend it by sign-bit then store it in $3, $3 = $0 >> 2
ROR $4, $0, 5	0111_1000_1000_0101	7885	Rotate register $0 to Right by 5-bit and store the result in $4, $4 = $0 rotate >> 5
SW $3, 0($1)	1000_1001_0110_0000	8960	Store a data from register $3 to memory at address 0.
LW $0, 0($1)	1000_0001_0000_0000	8100	Load a data from memory at address 0 to register $0.
BEQZ $5, label1	1010_0101_0000_0110	a506	Check if $5 is equal to zero then branch to label1, else go to next instruction
BNEZ $3, label2	1010_1011_0000_0111	ab07	Check if $3 is not equal to zero then branch to label2, else go to next instr.
BLTZ $1, label3	1011_0001_0000_1000	b108	Check if $1 is less than zero then branch to label3, else go to next instruction
BGEZ $2, label4	1011_1010_0000_1001	ba09	Check if $2 is greater than or equal to 0 then branch to label4, else go next instr.
BGTZ $2, label5	1100_0010_0000_1010	c20a	Check if $2 is greater than zero then branch to label5, else go to next instr.
BLEZ $4, label6	1011_0100_0000_1011	b40b	Check if $4 is less than or equal to 0 then branch to label4, else go next instr.
JR $7, 0	1101_0111_0000_0000	d700	Jump to the return address which is PC+1 that stored in $7 with offset of 0
JALR $2, 5	1101_1111_0000_0000	df00	Jump and link to Register $2 with adding immediate of 5, address = &$2 + 5
SET 5	1110_0000_0000_0101	e005	Put an immediate of 5 in Register $0 with zero extended.
SSET 7	1110_1000_0000_0111	e807	Shift left the content of $0 by 11 then put the immediate of 7 in lower Bound of $0
J next	1111_0000_0000_0010	f002	Jump to address of the label 'next' which is equal to PC + address of the label
JAL procedure	1111_1000_0000_0010	f802	Jump and link to the label "procedure" and store the return address in $0

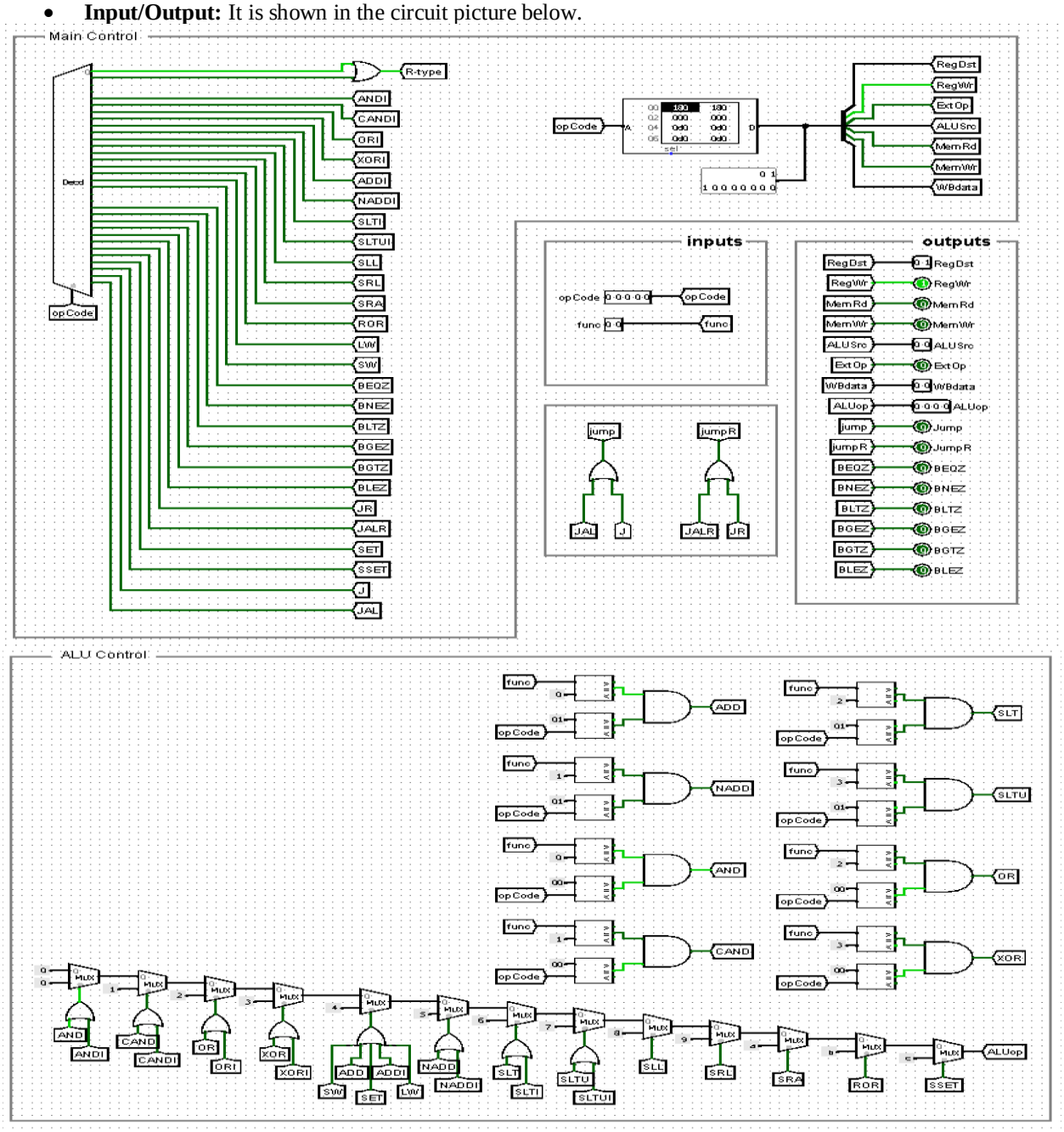
2. Pipelined Design

2.1 Component Description

2.1.1 Main Control Unit (modified)

- **Modification:** Since the pipelined Design need to have a PC control Unit as a separate Component and we choose to be as much as course slides says, so we decided to separate the PCSrc signal from the main control unit and design a new unit that handle the PCSrc signal and all its logic behind it, so we modify the main control unit by removing the PCSrc signal and separate every kind of branches as an isolated signal so we can optimize it easily when we dealing with the PC Control Unit, and we change small thing with the SET and SSET instructions.

- **Input/Output:** It is shown in the circuit picture below.



- **Control Tables:** ALU Control Table, Main Control Table are shown below.

ALU Control Truth Table (Modified)

Op	Op (dec)	funct	funct (dec)	ALUOp	4-bit Coding	ALUOp (dec)
R-type	0	AND	0	AND	0000	0
R-type	0	CAND	1	CAND	0001	1
R-type	0	OR	2	OR	0010	2
R-type	0	XOR	3	XOR	0011	3
R-type	1	ADD	0	ADD	0100	4
R-type	1	NADD	1	NADD	0101	5
R-type	1	SLT	2	SLT	0110	6
R-type	1	SLTU	3	SLTU	0111	7
ANDI	4	X	X	AND	0000	0
CANDI	5	X	X	CAND	0001	1
ORI	6	X	X	OR	0010	2
XORI	7	X	X	XOR	0011	3
ADDI	8	X	X	ADD	0100	4
NADDI	9	X	X	NADD	0101	5
SLTI	10	X	X	SLT	0110	6
SLTUI	11	X	X	SLTU	0111	7
SLL	12	X	X	SLL	1000	8
SRL	13	X	X	SRL	1001	9
SRA	14	X	X	SRA	1010	10
ROR	15	X	X	ROR	1011	11
LW	16	X	X	ADD	0100	4
SW	17	X	X	ADD	0100	4
SSET	29	X	X	SSET	1100	12
SET	28	X	X	ADD	0100	4

Main Control Truth Table (Modified)

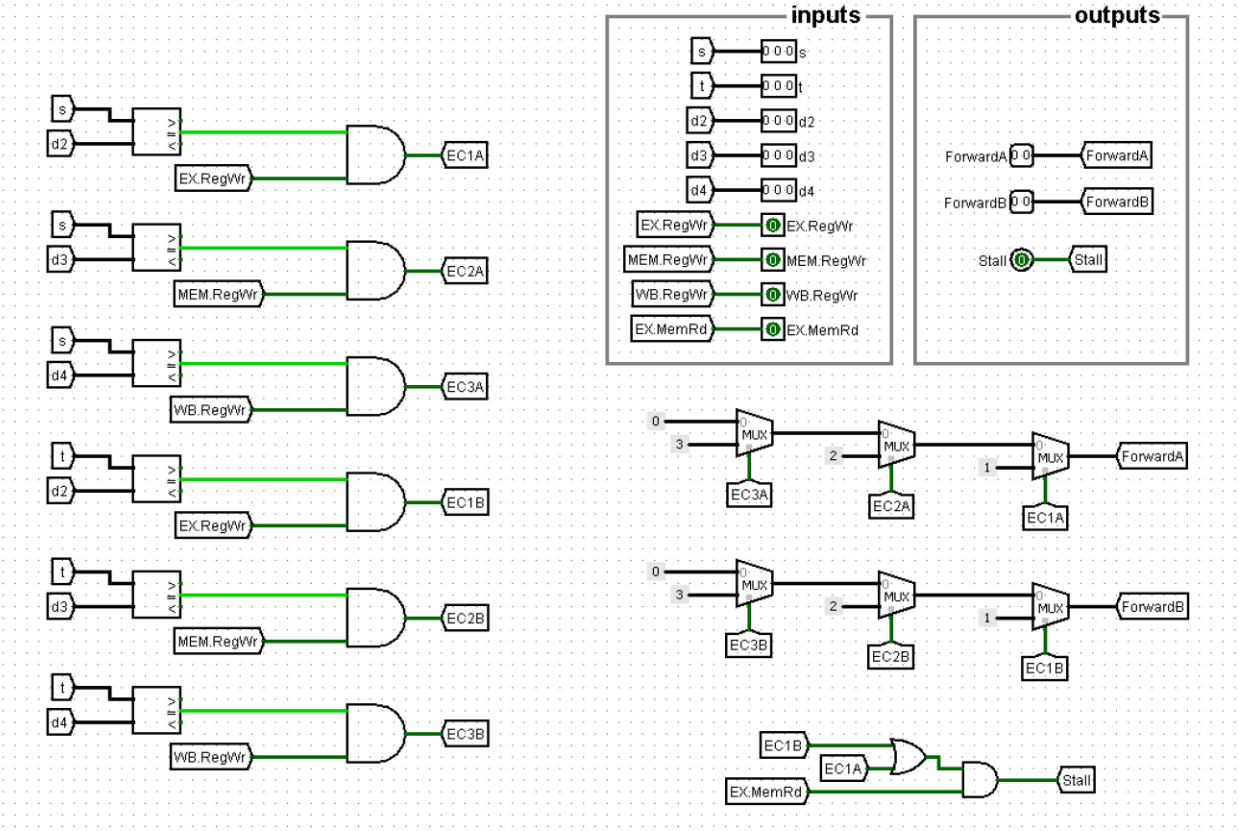
Op	Op (dec)	RegDst	RegWr	ExtOp	ALUSrc	MemRd	MemWr	WBdata	ALUop	Binary	Hex
R-type	0 OR 1	01 = Rd	1	X	00 = BusB	0	0	00 = ALU	0 – 7	0110000000	180
ANDI	4	00 = Rt	1	1 = sign	01 = imm5	0	0	00 = ALU	0	0011010000	0d0
CANDI	5	00 = Rt	1	1 = sign	01 = imm5	0	0	00 = ALU	1	0011010000	0d0
ORI	6	00 = Rt	1	1 = sign	01 = imm5	0	0	00 = ALU	2	0011010000	0d0
XORI	7	00 = Rt	1	1 = sign	01 = imm5	0	0	00 = ALU	3	0011010000	0d0
ADDI	8	00 = Rt	1	1 = sign	01 = imm5	0	0	00 = ALU	4	0011010000	0d0
NADDI	9	00 = Rt	1	1 = sign	01 = imm5	0	0	00 = ALU	5	0011010000	0d0
SLTI	10	00 = Rt	1	1 = sign	01 = Imm5	0	0	00 = ALU	6	0011010000	0d0
SLTUI	11	00 = Rt	1	1 = sign	01 = Imm5	0	0	00 = ALU	7	0011010000	0d0
SLL	12	00 = Rt	1	0 = zero	01 = imm5	0	0	00 = ALU	8	0010010000	090
SRL	13	00 = Rt	1	0 = zero	01 = imm5	0	0	00 = ALU	9	0010010000	090
SRA	14	00 = Rt	1	0 = zero	01 = imm5	0	0	00 = ALU	10	0010010000	090
ROR	15	00 = Rt	1	0 = zero	01 = imm5	0	0	00 = ALU	11	0010010000	090
LW	16	00 = Rt	1	1 = sign	01 = Imm5	1	0	01 = Mem	4	0011011001	0d9
SW	17	X	0	1 = sign	01 = Imm5	0	1	X	4	0001010100	054
BEQZ	20	X	0	1 = sign	X	0	0	X	X	0001000000	040
BNEZ	21	X	0	1 = sign	X	0	0	X	X	0001000000	040
BLTZ	22	X	0	1 = sign	X	0	0	X	X	0001000000	040
BGEZ	23	X	0	1 = sign	X	0	0	X	X	0001000000	040
BGTZ	24	X	0	1 = sign	X	0	0	X	X	0001000000	040
BLEZ	25	X	0	1 = sign	X	0	0	X	X	0001000000	040
JR	26	X	0	1 = sign	X	0	0	X	X	0001000000	040
JALR	27	10 = $7	1	1 = sign	X	0	0	10 = PC+1	X	1011000010	2c2
SET	28	11 = $0	1	1 = sign	11 = A = 0-32bit, B=Imm11,	0	0	00 = ALU	X	1111110000	3f3
SSET	29	11 = $0	1	0 = zero	10 = A= A <<11, B=imm11,	0	0	00 = ALU	12	1110100000	3a0
J	30	X	0	1 = sign	X	0	0	X	X	0001000000	040
JAL	31	10 = $7	1	1 = sign	X	0	0	10 = PC+1	X	1011000010	2c2

2.1.2 Hazard Detection Unit

- **Brief Description:** Since the pipeline processor can start an instruction per a cycle, and each instruction has a period of five cycles, and since there are some instructions that depend on previous instructions whose final outputs have not been written to the destination register yet, the Hazard Unit is introduced to solve this issue. The Hazard Unit prepares the output of last four instructions, which have not been written to the registers yet, and choose the appropriate value to pass it to the ALU for the current instruction if there is dependency between the previous instruction and the current instruction.

- **Alternative Design:** To implement the same Hazard Unit with the same functionalities, we can do some modifications to the current design. We can replace the three MUXs for each forward with two AND-Gates and one splitter. The output of the splitter will be one of four possible outputs: 000, 001, 010 and 100. The problem with this design is that it requires bigger MUXs and that will waste the full use of some of the hardware. For example, the MUXs mentioned above uses 2-bit while we need to make it a 3-bit.

- **Input/Output:** It is shown in the circuit picture below.



- **Control Tables:** Forwarding Control Signal Table is shown below.

Signal	Explanation
ForwardA = 0	First ALU operand comes from register file = Value of (Rs)
ForwardA = 1	Forward result of previous instruction to A (from ALU stage)
ForwardA = 2	Forward result of 2nd previous instruction to A (from MEM stage)
ForwardA = 3	Forward result of 3rd previous instruction to A (from WB stage)

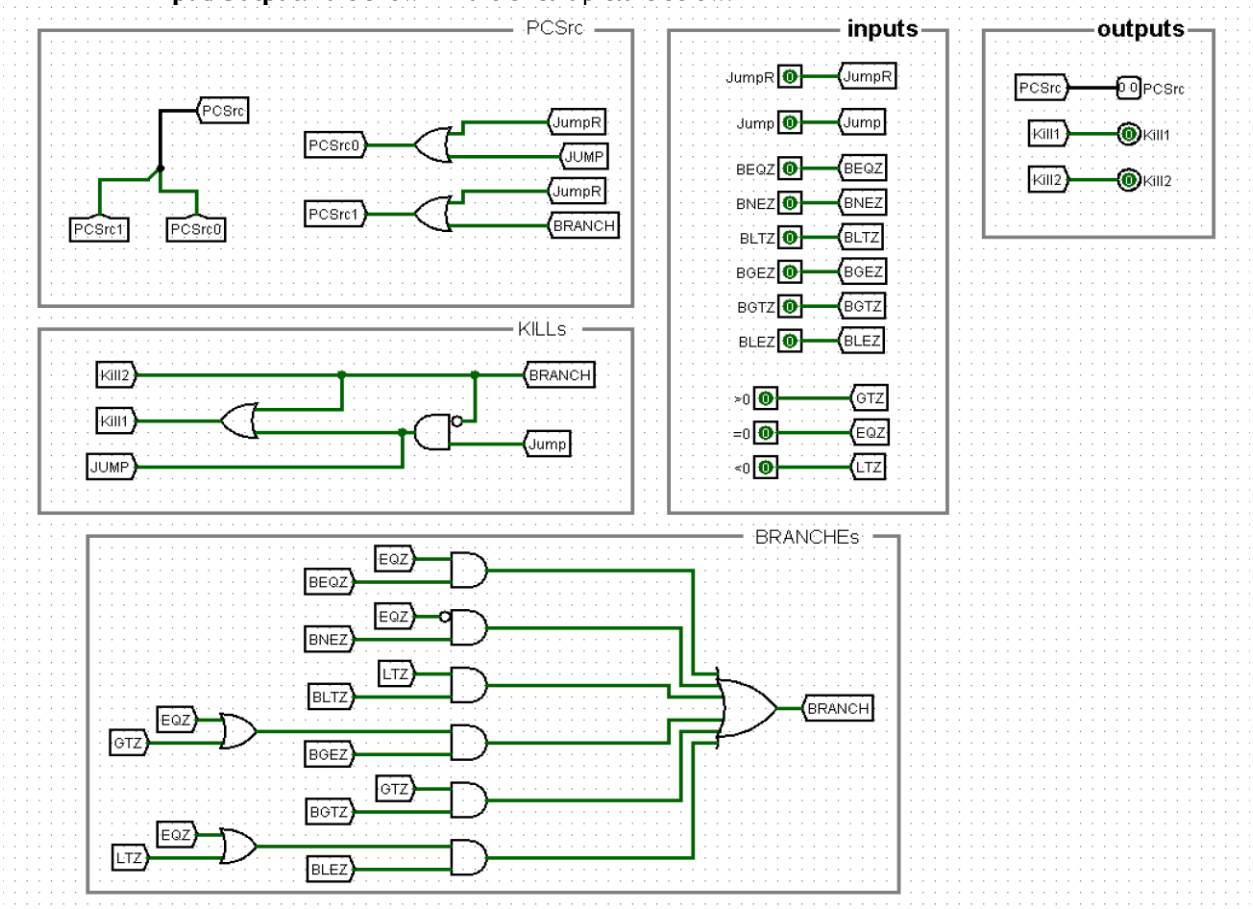
ForwardB = 0	Second ALU operand comes from register file = Value of (Rt)
ForwardB = 1	Forward result of previous instruction to B (from ALU stage)
ForwardB = 2	Forward result of 2nd previous instruction to B (from MEM stage)
ForwardB = 3	Forward result of 3rd previous instruction to B (from WB stage)

2.1.3 PC Control Unit

- **Brief Description:** since the pipelined design need two extra signals which are kill1 and kill2, so we decided to be consistent with course slides and just do the design as it is. So, this unit will be responsible for the generating the PCSrc signal to determine which kind of next program count should be used for each instruction. i.e. for the branch if it is taken then the PCSrc will be 2 and the Next PC will branch to the given address (or label), and if branch not taken then the Next PC will be the incremented PC, and for the jump, it is always must be taken so the next PC will be the jump target address (everything related to this will be illustrated in PC Control Table Below). Also, this unit is responsible for generate the Kill1 and Kill2 which response for bubbling the instruction that will come after a jump or if branch taken so they will not be executed, because we replace it with zeros.

- **Alternative Design:** We could implement this unit in the mine control unit instead of implemented as a separate unit as we did with the single cycle, so we can generate the PCSrc, Kill1 and Kill2 from the main control unit.

- **Input/Output:** It is shown in the circuit picture below.

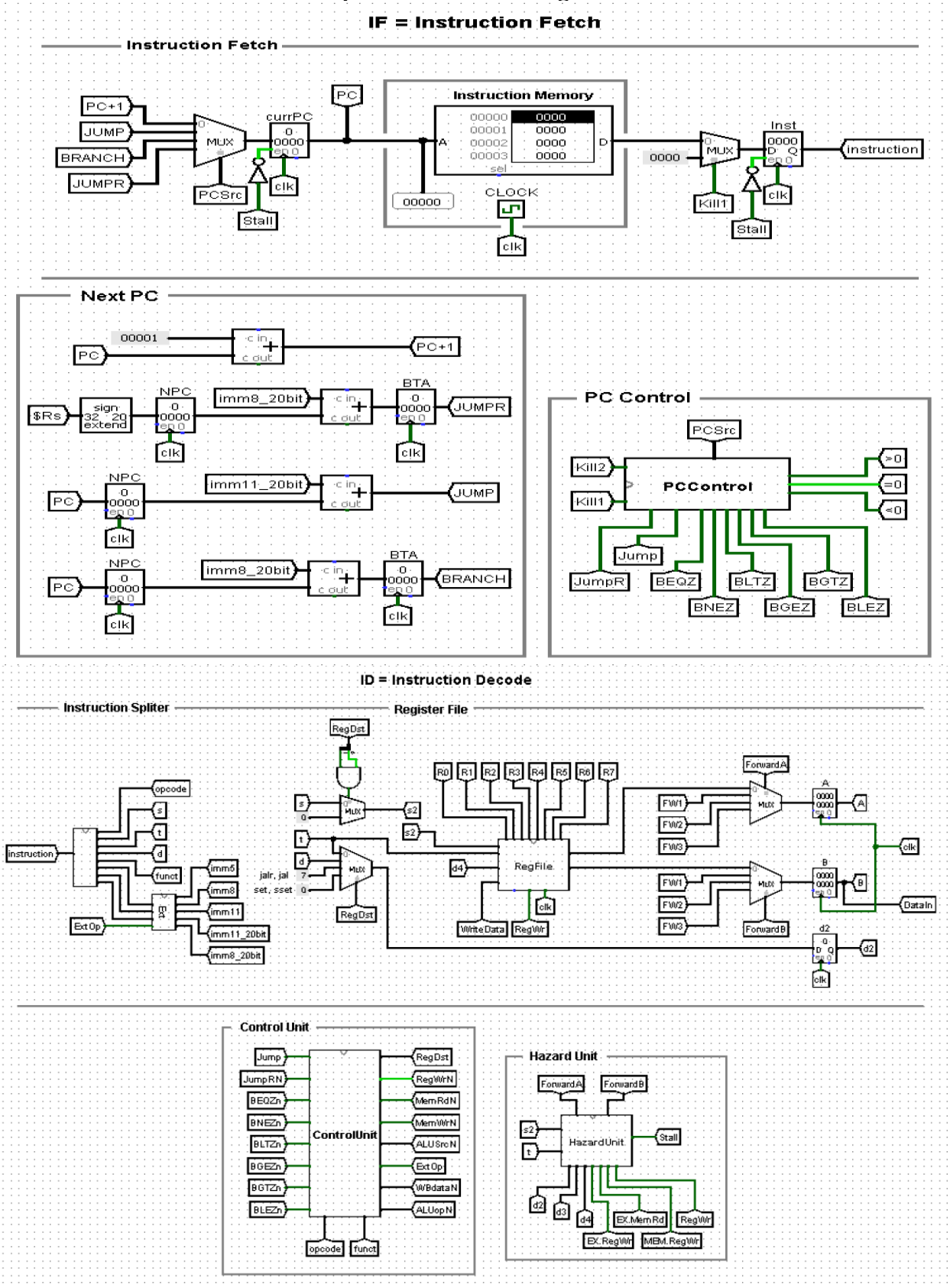


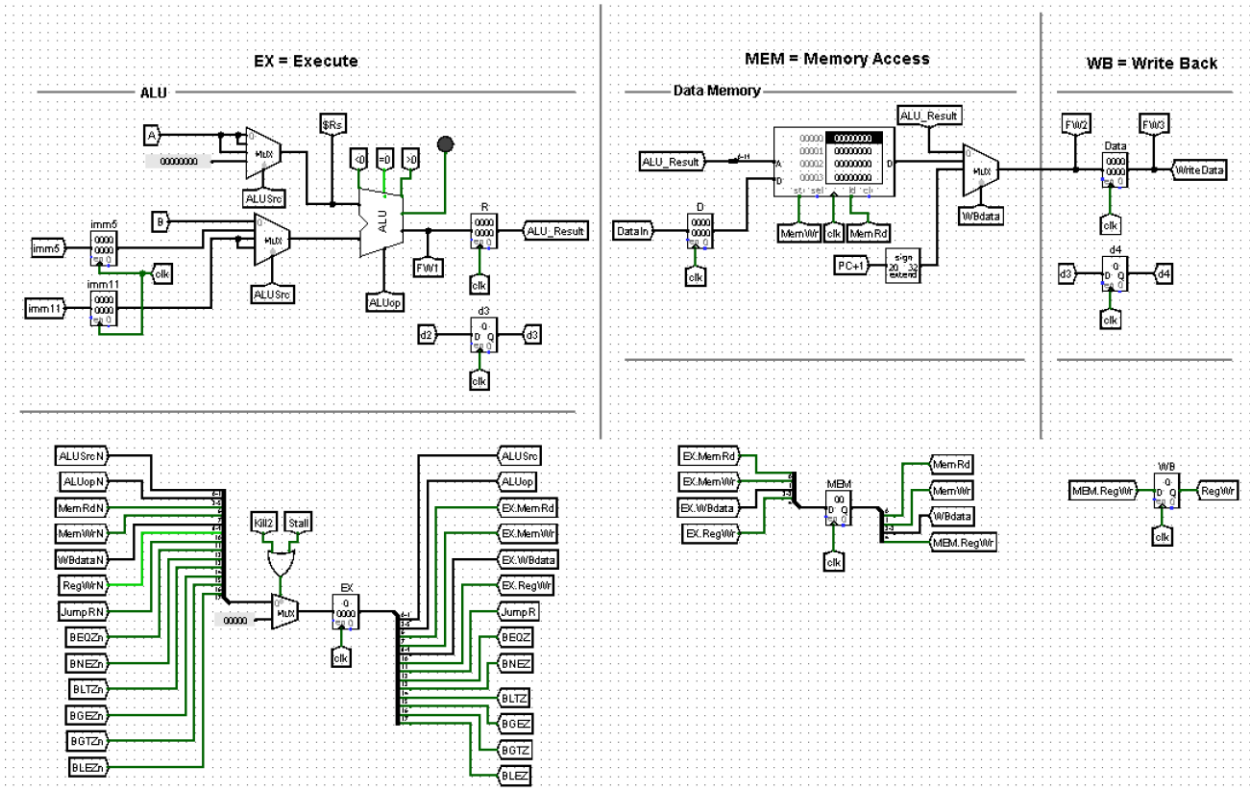
- **Control Tables:** PC Control Truth Table is shown below.

PC Control Truth Table

Opcode	Zero flag	PCSrc
R-type	X	0 = Increment PC
J	X	1 = Jump Target Address
JAL	X	1 = Jump Target Address
BEQZ	0	0 = Increment PC
BEQZ	1	2 = Branch Target Address
BNEZ	0	0 = Increment PC
BNEZ	1	2 = Branch Target Address
BLEZ	0	0 = Increment PC
BLEZ	1	2 = Branch Target Address
BGTZ	0	0 = Increment PC
BGTZ	1	2 = Branch Target Address
BGEZ	0	0 = Increment PC
BGEZ	1	2 = Branch Target Address
BLEZ	0	0 = Increment PC
BLEZ	1	2 = Branch Target Address
JR	X	3 = Jump Register Target Address
JALR	X	3 = Jump Register Target Address
Other than Jump and Branch	X	0 = Increment PC

*** Screenshot of the Whole Pipelined Processor Design.**



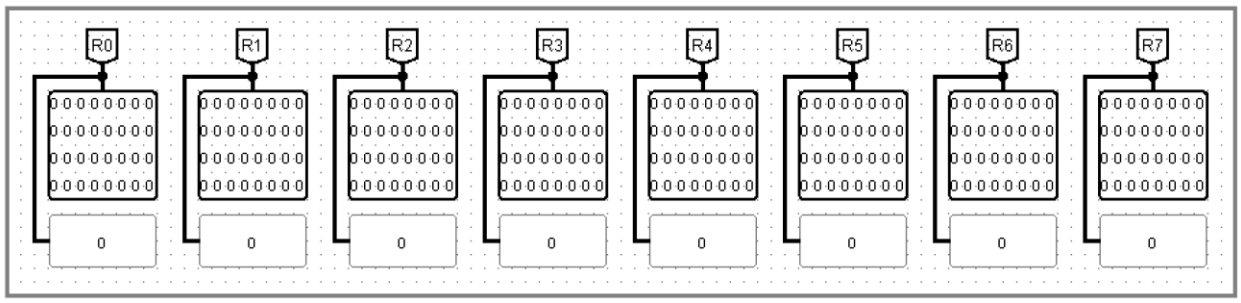


2.2 Testing Programs

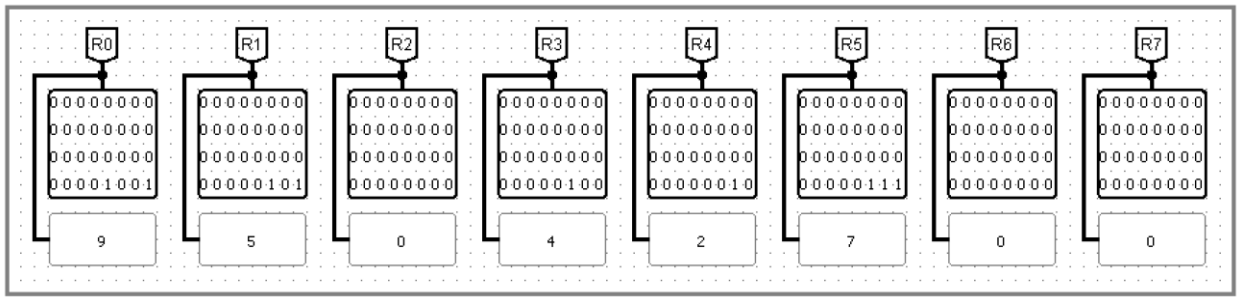
2.2.1 Independent Instructions

This program has five independent instructions that tests the basic functionality of the pipelined processor.

***Screenshot Before Running independent instructions program**



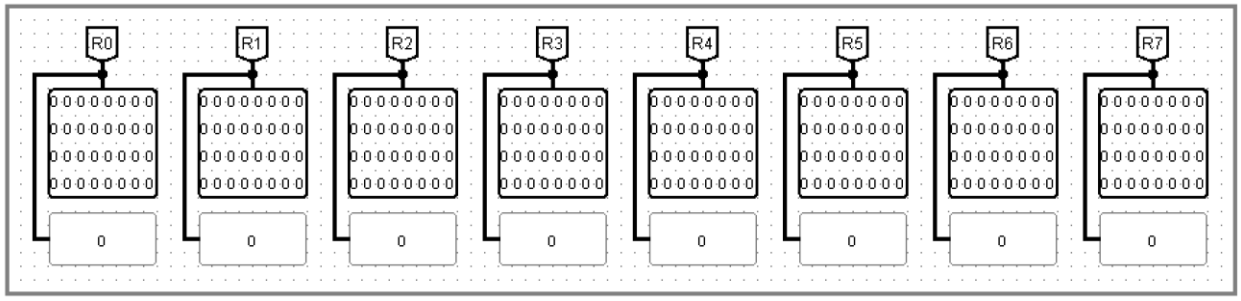
***Screenshot After Running independent instructions program**



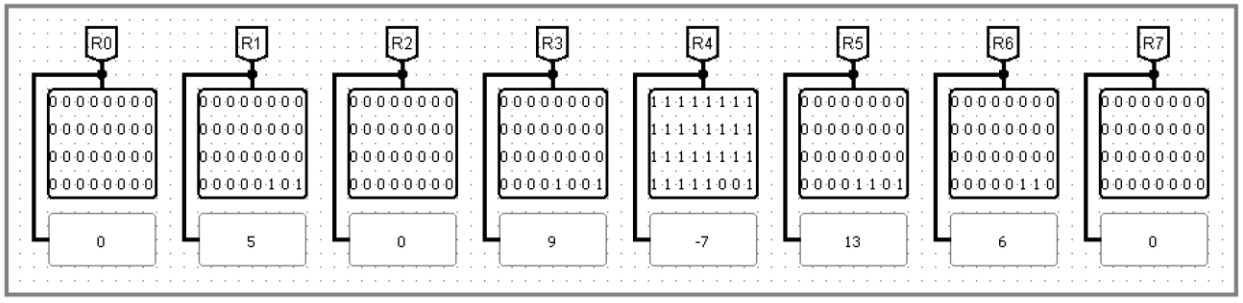
2.2.2 Dependent Instructions

This program has more than five dependent instructions that tests forwarding logic.

***Screenshot Before Running dependent instructions program**



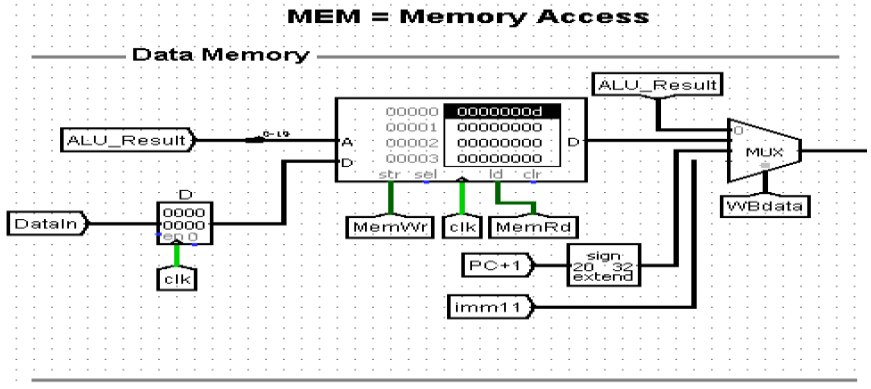
***Screenshot After Running dependent instructions program**

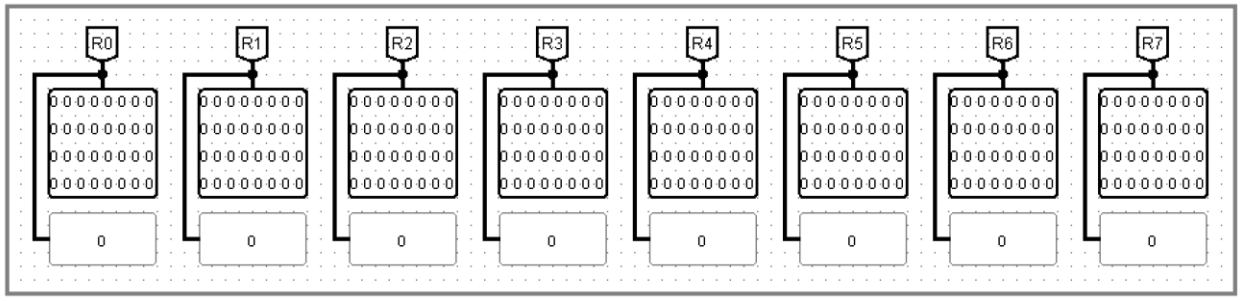


2.2.3 Test dependent instructions including 'LW'

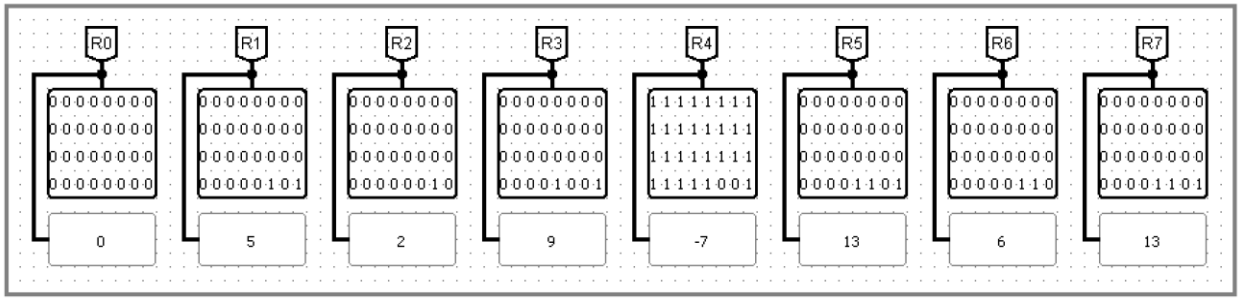
This program will have 'LW' and many dependent instructions follows it to test and show how the processor is stalled for one cycle.

***Screenshot Before Running dependent instructions with 'LW' program**





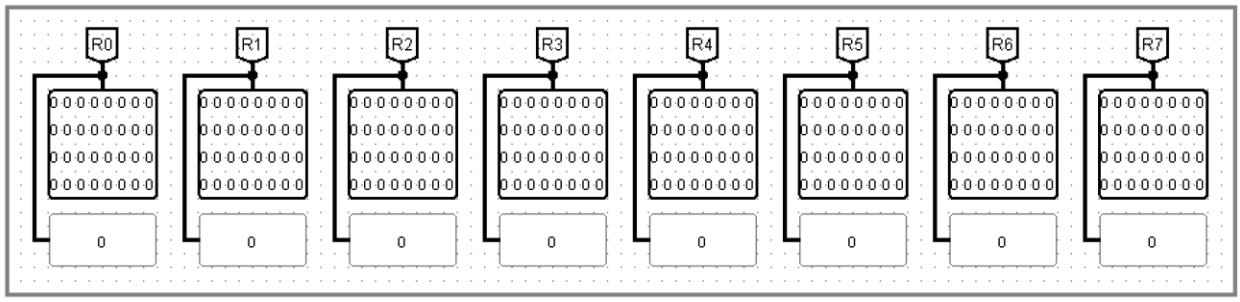
***Screenshot After Running dependent instructions with 'LW' program**



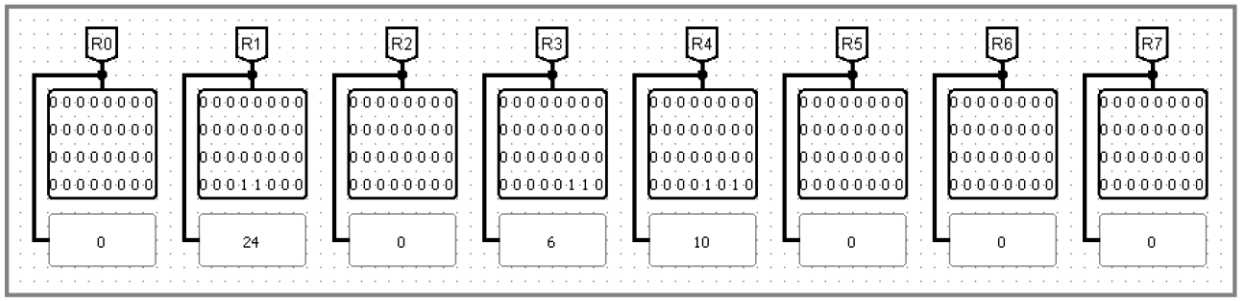
2.2.4 Program to test the branching logic

This program is to test the branching logic and the bubble insert into the system.

***Screenshot Before Running the branching instructions program**



***Screenshot After Running the branching instructions program**



***Note that all the assembly codes and its hexadecimal values of these programs are in the '172_52_Group6_Pipelined.zip' file inside the Tests folder.**

3. Teamwork

Single Cycle Contribution						
Team Member	Register File	ALU	Control Unit	Next PC	Main Processor	Testing Programs
Yousef	33.3%	10%	100%	30%	90%	100%
Ryan	33.3%	60%	0%	0%	5%	0%
Abdulrahman	33.3%	30%	0%	70%	5%	0%

Pipelined Design Contribution						
Team Member	Next PC	PC Control	Control Unit	Hazard Detection Unit	Main Processor	Testing Programs
Yousef	100%	10%	100%	10%	90%	50%
Ryan	0%	90%	0%	0%	5%	0%
Abdulrahman	0%	0%	0%	90%	5%	50%

Report Contribution					
Team Member	S.C. Main Component Section	S.C. Testing Programs Section	P.D. Main Component Section	P.D. Testing Programs Section	Overall Document Design Templet
Yousef	33.3%	100%	66.6%	100%	100%
Ryan	33.3%	0%	0%	0%	0%
Abdulrahman	33.3%	0%	33.4%	0%	0%

*S.C. is an acronym for Single Cycle and P.D. is an acronym for Pipelined Design.