Fayoum University

Faculty of Engineering

Electronics & Electrical Communications Department

Report of:

Pipelined MIPS Processor Implementation

For

Prof. Dr. Gihan Naguib

Associate Professor, Department of Electrical Engineering.

Eng. Jihad Awad

Teaching Assistant, Department of Electrical Engineering.

Team Members :-

- **Ahmed Ibrahim Abdel-Razek**
- **Mustafa Mohamed Mustafa**
- **Moaz Adel Hamed**

Students at Department of Electronics and

Electrical Communications.

Table of Content

Part 1:
Single-Cycle Processor

Introduction

- ## Overview

In the realm of computer architecture and digital design, the MIPS (Microprocessor without Interlocked Pipeline Stages) architecture stands as a fundamental framework for understanding and developing modern processors. With its emphasis on simplicity, efficiency, and modularity, the MIPS architecture has been a cornerstone in both academic research and industrial applications.

This report presents the culmination of a project aimed at designing and implementing a Single-Cycle MIPS Processor, leveraging the principles and guidelines set forth by the MIPS architecture. The objective of this endeavor is to create a processor capable of executing instructions within a single clock cycle, thereby maximizing performance and throughput while minimizing latency.

• Instruction Set Architecture

In this project, We have designed a simple 16-bit RISC processor with seven 16-bit general purpose registers: R1 through R7. R0 is hardwired to zero and cannot be written, we are left with seven registers. There is also one special-purpose 16-bit register, which is the program counter (PC). All instructions are only 16 bits. There are three instruction formats, R-type, I-type, and J-type as shown below :

R-type format

5-bit opcode (Op), 3-bit destination register Rd, and two 3-bit source registers Rs & Rt and 2-bit function field F.

Op^5	F^2	Rd^3	Rs^3	Rt^3

I-type format

5-bit opcode (Op), 3-bit destination register Rd, 3-bit source register Rs, and 5-bit immediate.

Op^5	Imm^5	Rs^3	Rt^3

J-type format

5-bit opcode (Op) and 11-bit immediate.

Op^5	Imm^{11}

Register Use

For R-type instructions, Rs and Rt specify the two source register numbers, and Rd specifies the destination register number. The function field F can specify at most four functions for a given opcode. We can reserve several opcodes for R-type instructions.

For I-type instructions, Rs specifies a source register number, and Rt can be a second source or a destination register number. The immediate constant is only 5 bits because of the fixed-size nature of the instruction. The size of the immediate constant is suitable for our uses. The 5-bit immediate constant can be signed or unsigned depending on the opcode. The immediate constant is signed (range is -16 to +15), except for shift and rotate instructions (range is 0 to 31).

The J-type format is used by J (jump), JAL (jump-and-link), and for LUI instructions. The 11-bit immediate is used for PC-relative Addressing and constant formation.

Instruction Description

Opcodes 0 and 1 are used for R-type instructions. Opcode 2 is used for the JR (jump register) instruction. Opcodes 4 through 17 are used for I-type instructions. The 5-bit immediate constant is zero-extended for ANDI, ORI, and XORI. It is sign-extended for the remaining instructions.

There are three shifts and one rotate instruction. To shift or rotate, use the lower 4 bits of Imm^5 as the shift/rotate amount. There is only one rotate left (ROL) instruction. To rotate right by n bits, you can rotate left by 16 – n bits, because registers are 16 bits. The Load Upper Immediate (LUI) is of the J-type to have an 11-bit immediate constant loaded into the upper 11 bits of register R1. The LUI can be combined with ORI to load any 16-bit constant into a register. Although the instruction set is reduced, it is still rich enough to write useful programs.

Processor Architecture

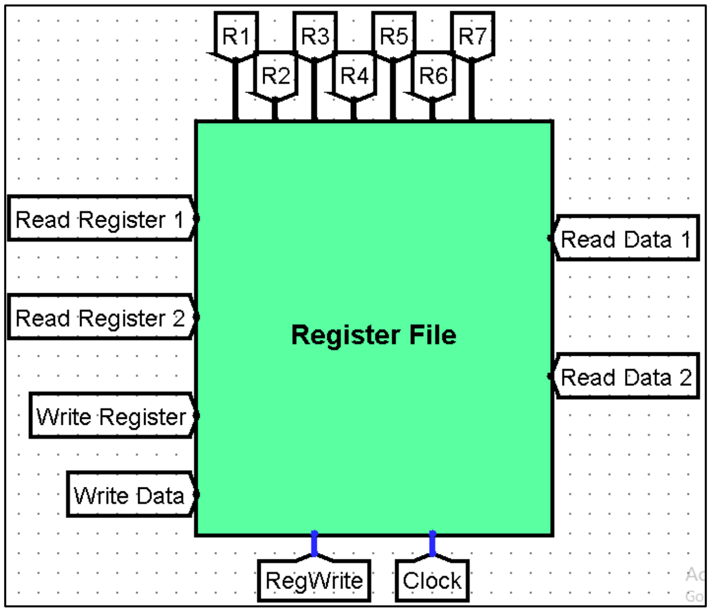
Our processor is meticulously constructed from six main components: Arithmetic Logic Unit (ALU), Register File, Instruction Memory, Data Memory, Control Units, and Program Counter (PC).

These components are intricately interconnected through a datapath to ensure the precise execution of all instructions. In the following subsequent sections, we will provide comprehensive descriptions of each component, elucidating their functionality and the intricate processes involved in executing instructions.

Single-Cycle Architecture

• Register File

Register File Overview



Our implemented Register File boasts seven 16-bit registers denoted as R1 to R7.

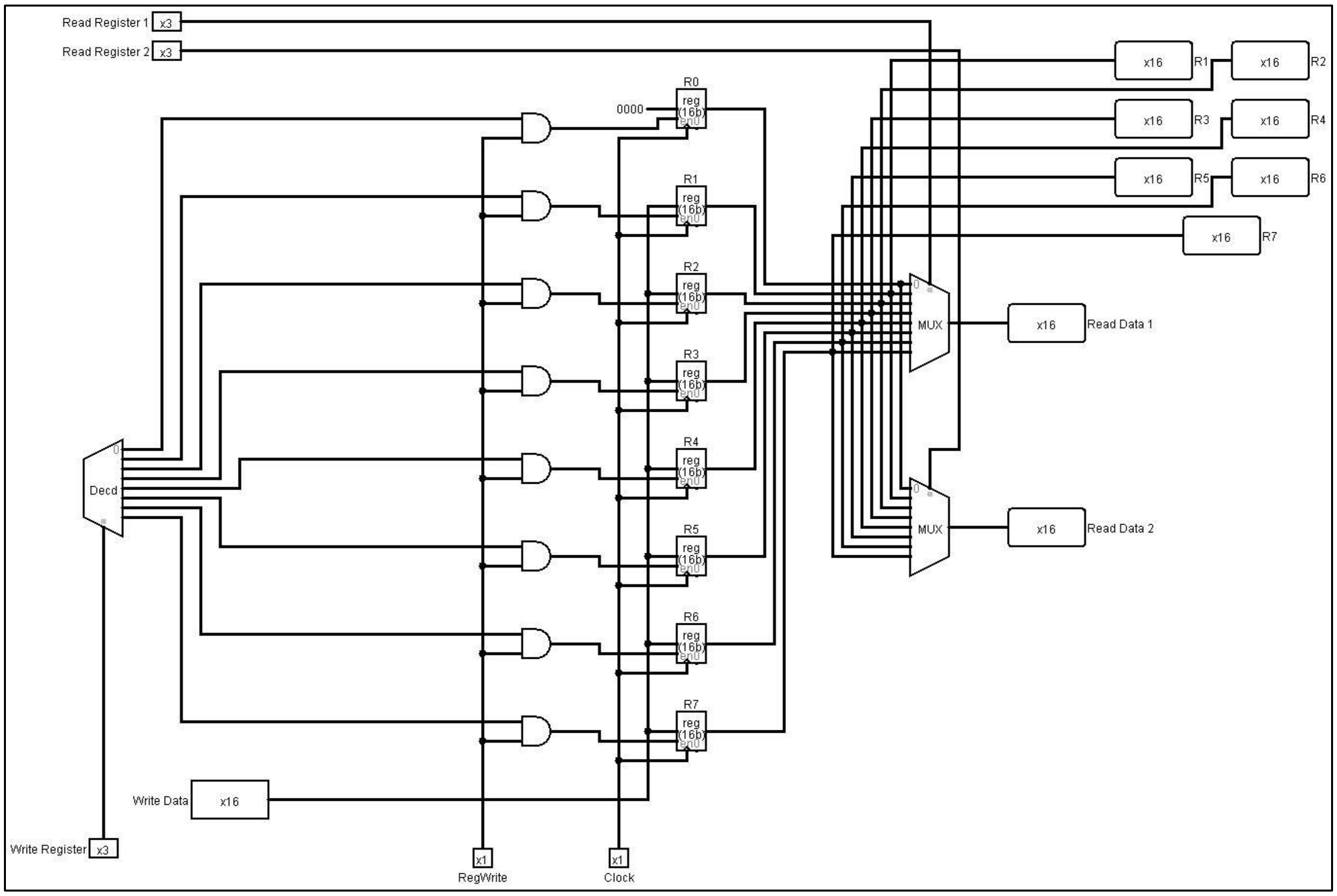
It has two read ports and one write port. Notably, R0 is hardwired set to zero, serving as a default value.

This register file features dual sources, a single destination input, and a data write input. AddItionally, it is outfitted with two output buses to facilitate data output from the register file.

A pivotal control signal, RegWrite, is integrated to regulate the enabling or disabling of writing within the register file. Lastly, the Register File accommodates an input specifically designated for the Clock signal, ensuring synchronous operation.

Register File Structure

In our design, we have implemented a decoder equipped with a Write Register selector, facilitating the seamless designation of the target register for writing operations. This decoder selects from R1 to R7, serves as a pivotal component in directing data to the appropriate register within the Register File.



To ensure precise control over the writing operations, each terminal of the decoder is intricately linked with the RegWrite signal through AND Gates. This configuration allows for effective regulation of writing.

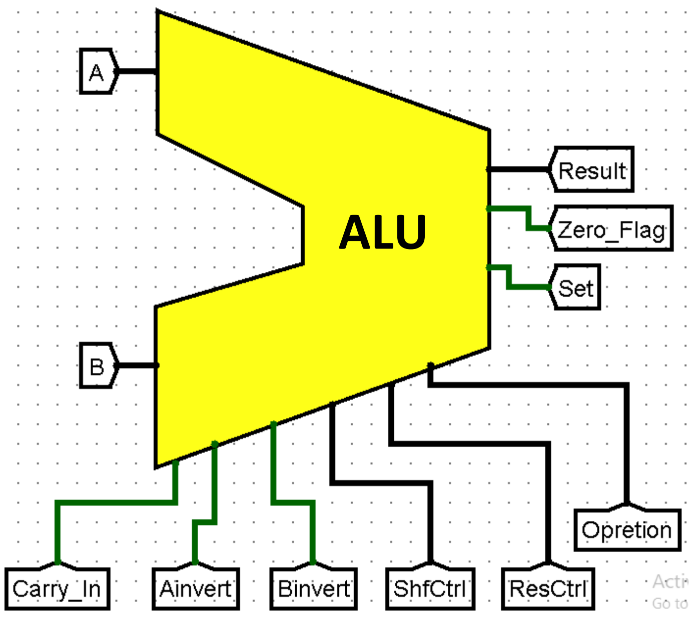
Furthermore, we used two Multiplexers to enrich the functionality of our system by enabling the selection of output ports. This feature grants us to choose between utilizing one or both of the output ports, depending on the instruction.

As part of our validation strategy during simulation, we have included seven output ports, each dedicated to a respective register within the Register File to just reading wile testing processing but it are not main components in the Register File.

• Arithmetic and Logic Unit (ALU)

ALU Overview

Embedded within the CPU, the Arithmetic and Logic Unit (ALU) executes arithmetic and logic operations on operands found in computer instruction words.



ALU has two inputs (A and B) and a single main output for result, so that ALU delivers the outcome of the operation.

It incorporates essential signals like Set and Zero_Flag which used for branch instructions.

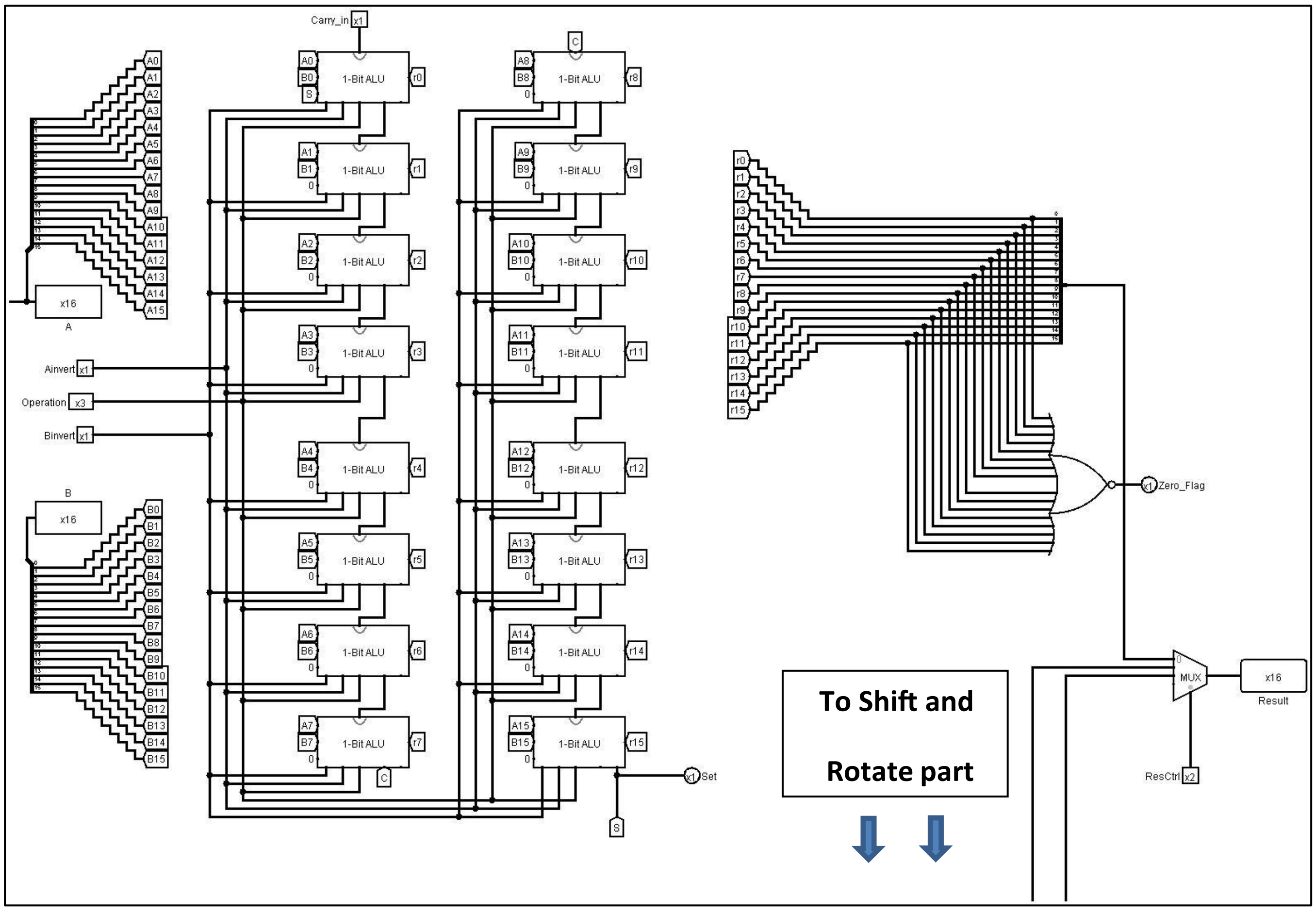
Governed by the ALU control unit (which will be explained later), the ALU receives the following signals (Carry_In, Ainvert, Binvert, ResCtrl, ShfCtrl, and Operation), which determined by ALU control unit due to the instruction needed to execute.

ALU Structure

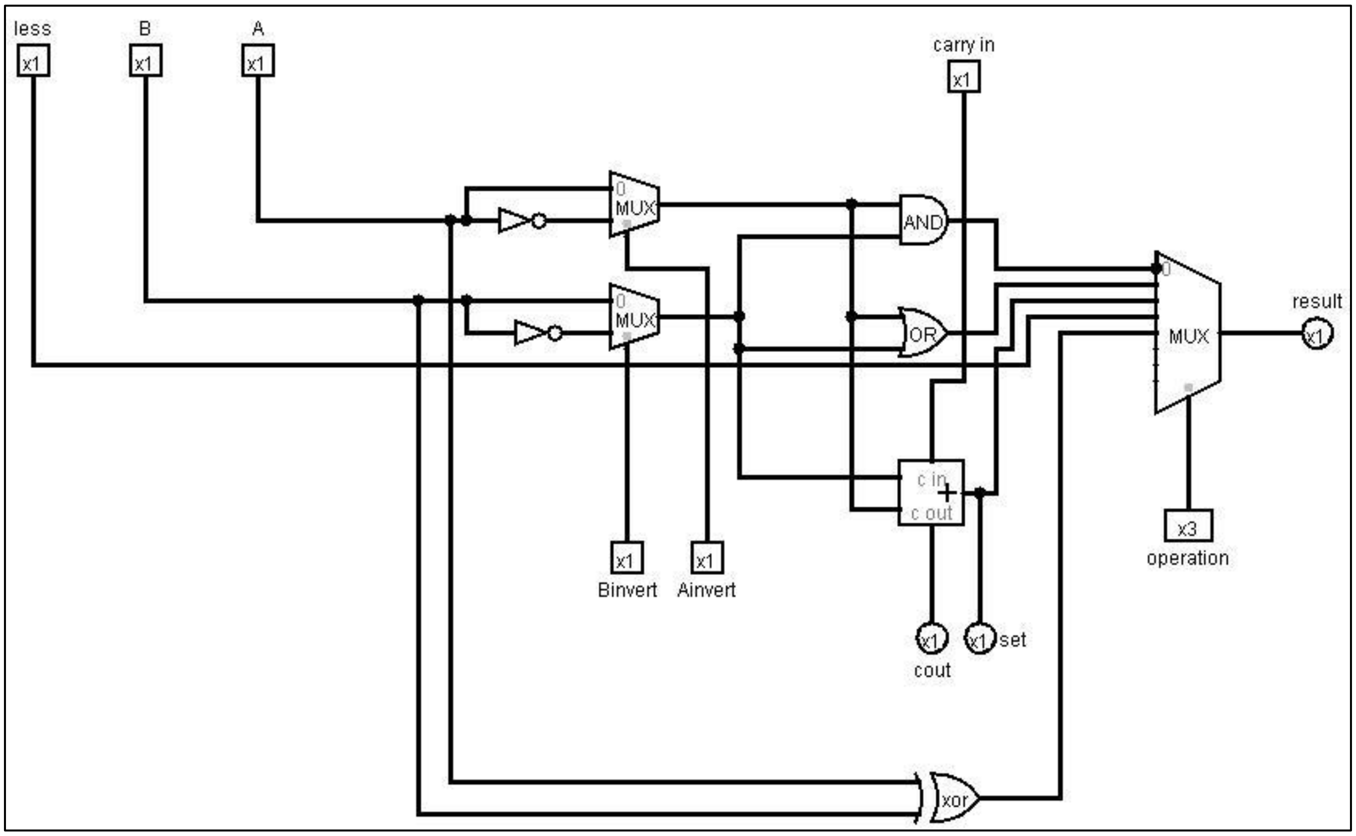
Our ALU consists of 2 main parts, Arithmetic & Logic part and Shift & Rotate Part, we will show each part in details :-

1. Arithmetic & Logic part

This part is responsible for instructions that needs an arithmetical or logical operation (AND, OR, XOR, NOR, Add, Sub, SLT, SLTU, ANDI, ORI, XORI, AddI, LW, SW)



It consists of 16 1-Bit ALU (one for each bit) linked together as carry out of each one is the carry in of the next one.



Each 1-Bit ALU has 2 main inputs for operands A and B, and 1 main output is the result, there are also signals inputs (AInvert, BInvert, Less) which controls different operations

ALU operations divided into two types, Logical and Arithmetical as follow:

1) Logic Operations:

- Employs logic gates (AND, OR, XOR, NOR) for logical operations.

- For AndI, ORI, XORI, and AddI operations, similar to R-Type, the second ALU input originate from the extended immediate, differing only in the control signal (ALU Src) set to one.

2) Arithmetic Operations:

- Utilizes full Adders for AddItion and subtraction.

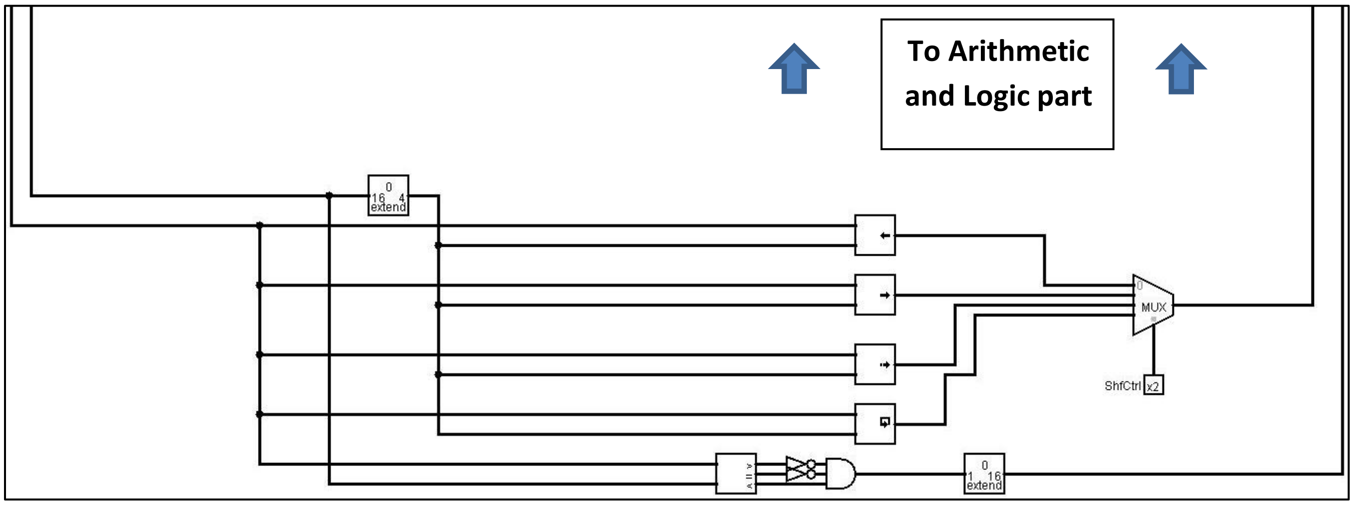
- AddItion involves summing the two inputs (A, B) with a carry-in of zero.

- Subtraction sets Carry in = 1, resulting in the multiplexer output being the inversion of B: A + B' + 1 = A + (B' + 1) = A + (-B) = A - B.

- SLT and SLTU implementation involves setting all output bits except the LSB to 0, with the LSB set to 1 if (A - B) is negative and 0 otherwise, aligning with the sign bit value of (A - B) where 1 denotes negative and 0 denotes positive.(but SLT deals with signed values ,while SLTU deals with unsigned values using comparator).

2. Shift & Rotate part

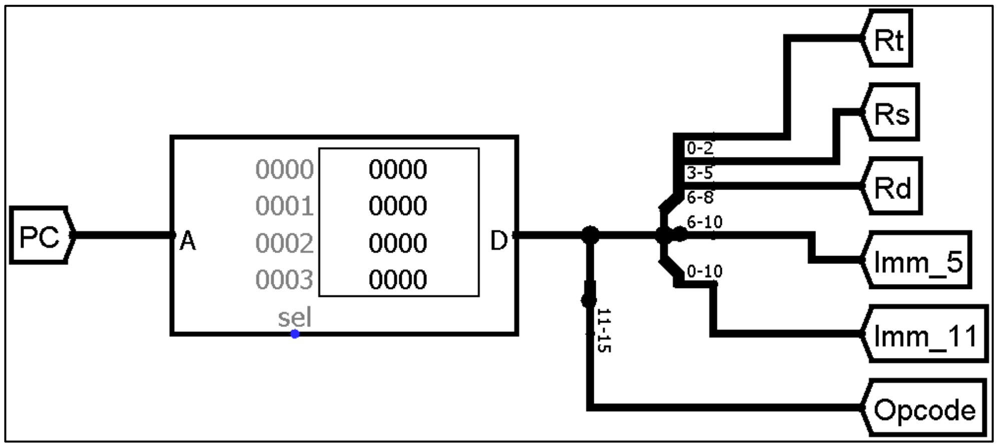
This part is responsible for instructions that use shifters (SLL, SRL, SRA, ROR) it also has a datapath for SLTU instruction.



It Executes logical shift left, logical shift right, shift right arithmatic, and rotate right operations. Shifting involves moving all bits within a register left or right. SLL denotes shift left logical (zero insertion from the right). SRL signifies shift right logical (zero insertion from the left). SRA denotes shift right arithmetic (sign bit insertion from the left).

Utilizes shifters for SLL, SRL, and ROR operations, with the shift and rotate amount being the unsigned immediate 5-bit value (bits 0-4) of the second ALU input (B) ;while shifter for SRA uses the signed immediate 5-bit value (bits 0-4) of the second ALU input (B).

• Instruction Memory



Instruction memory, depicted in the figure, functions as a ROM (Read-Only Memory) crucial for permanent data storage.

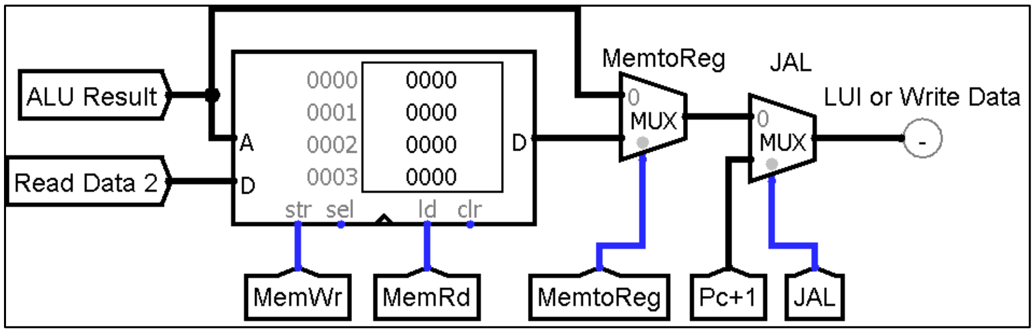
Stored data in ROM consists of four hex numbers (words), providing essential instructions for the system.

It operates on a word-Addressable basis, with input A sourced from the next PC. ensuring access to the memory.

The output comprises 16 bits, representing:

- For R-type format : 5-bit opcode (Op), 3-bit destination register Rd, and two 3-bit source registers Rs & Rt and 2- bit function field.
- For I-type format 5-bit opcode (Op), 3-bit destination register Rd, 3-bit source register Rs, and 5-bit immediate.
- For J-type format 5-bit opcode (Op) and 11-bit Immediate.

• Data Memory

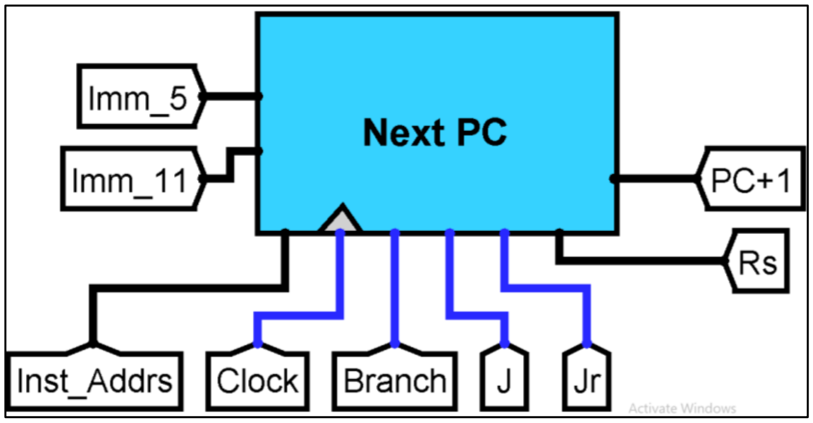


In our Processor, Data Memory serves as the repository for storing data post-instruction execution, facilitating its retrieval for subsequent use. To ensure optimal performance and versatility, we opted for RAM (Random Access Memory) as our choice for Data Memory. RAM's inherent speed and random access capabilities allow the processor to SWiftly access any location within the memory without the need to sequentially search through all Addresses.

Data manipulation within the Data Memory is orchestrated through the execution of LW (Load Word) and SW (Store Word) instructions, governed by the MemRd and MemWr control signals. Two primary inputs drive the Data Memory operations: firstly, the 16-bit result generated by the ALU determines the memory Address to be accessed; secondly, the Read Data 2, sourced from the Register File, is contingent upon the RegDst control signal, which designates whether it originates from Rd or Rt.

At the output stage of the Data Memory, a Multiplexer Selects between the ALU Result (when memory access is unnecessary) and the memory output (during data loading), guided by the MemtoReg control signal. Subsequently, another Mux is employed to determine whether to execute the JAL (Jump and Link) instruction, wherein the value of PC+1 from the Program Counter (PC) Block is selected, or to proceed with the first Mux value, initiating data writing into the Register File or execution of the LUI (Load Upper Immediate) instruction, contingent upon signals from the Control Unit.

• Program Counter

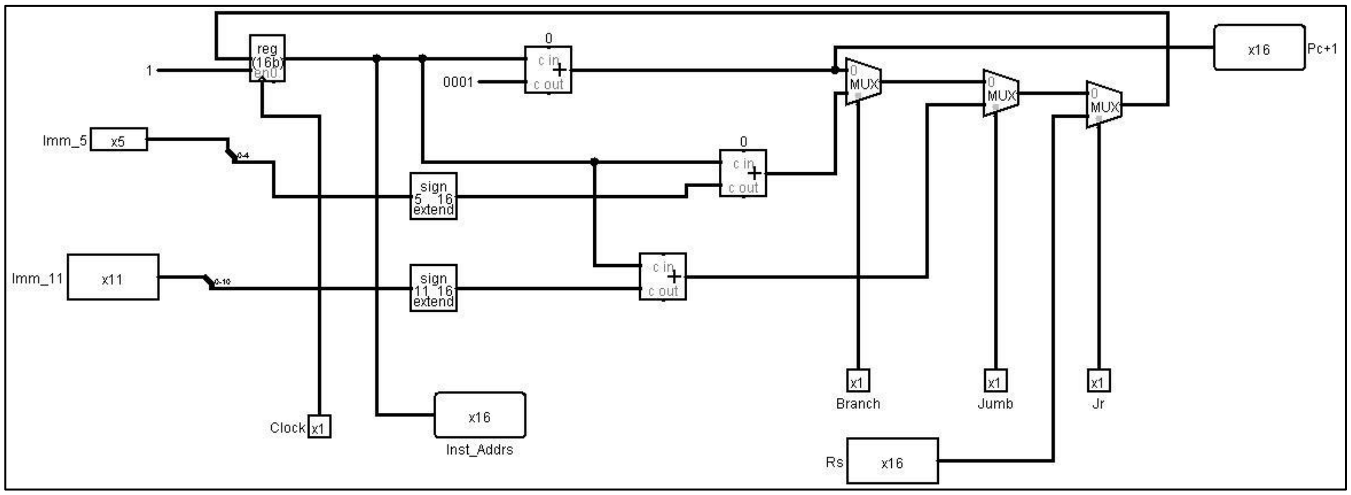


PC Overview

The Program Counter (PC) Block is an integral component of our architecture, housing a dedicated 16-bit register responsible for storing the Address of the current instruction residing in the Instruction Memory, awaiting execution. In AddItion to its primary function as a storage unit for instruction Addresses, the PC Block encompasses a datapath tailored to handle specific instructions like Branch instructions (BEQ, BNE, BLT, BGE), Jump instructions (J and JAL), and Jr instruction.

PC structure & datapath

For Branch instructions, such as BEQ, BNE, BLT, and BGE, a sign-extended 5-Bit immediate value is Added to the current PC value within the datapath. This operation yields the target Address required for branching, facilitating the execution of conditional branches based on specified conditions.



Conversely, for jump instructions (J and JAL), a sign-extended 11-Bit immediate value is Added to the current PC value within the datapath. This calculation effectively determines the target Address for unconditional jumps, enabling the seamless transition to the designated instruction location. Notably, in the case of the JAL instruction, the PC value incremented by 1 is stored in register R7, ensuring the preservation of the Address of the next instruction to be executed after the jump operation. This ensures that the program flow can be resumed accurately following the execution of the jump instruction.

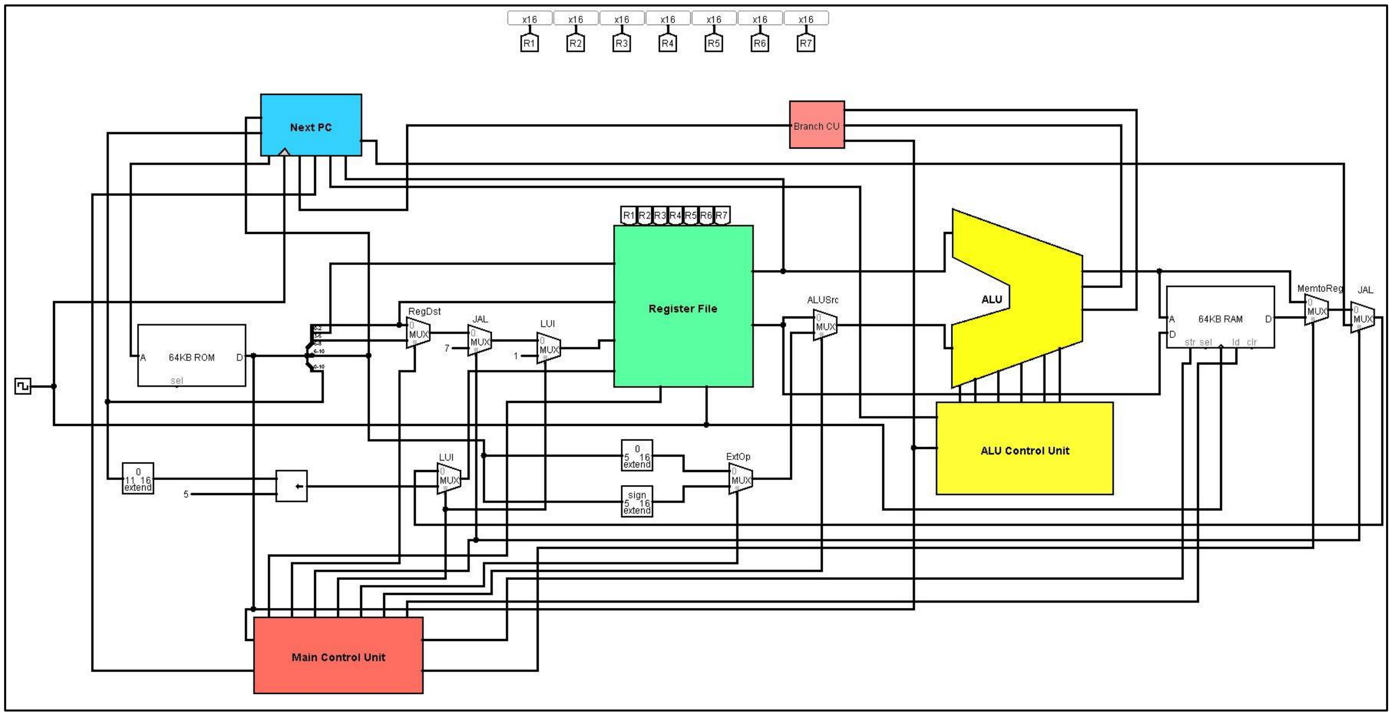
In the final stage of our design, a Multiplexer is incorporated to make a crucial selection between the value stored in the source register Rs and the value derived from the jump multiplexer. This selection determines the content to be stored in the Next Program Counter (PC) register, guided by a signal Jr.

When the signal Jr indicates the value of Rs, the instruction Jr is executed. This instruction transfers the content of the source register Rs directly into the Next PC register, effectively determining the next Address to be accessed in the program sequence.

By incorporating these functionalities into the PC Block's datapath, our architecture enables the seamless execution of branch and jump instructions, essential for program control and flow management. The precise handling of instruction Addresses ensures the correct sequencing of program execution and contributes to the overall efficiency and reliability of the processor's operation.

• Data Path

After comprehensively examining the structure of each component, the next step is to interconnect them to construct the complete processor, achieved through what we refer to as the Data Path.



In our design, the Program Counter (PC) selects the Address of the instruction to be executed. Subsequently, the instruction is fetched from the instruction memory and segmented into distinct parts via a splitter. The first segment, consisting of the first 3 bits (bits 0 to 2), enables the Register File's Read Register 2 (Rt). The next segment, comprising bits 3 to 5, corresponds to Rs (Read Register 1 in the Register File), whiles the third segment, bits 6 to 8, represents Rd. Finally, the last 7 bits (bits 9 to

15) encompass the Function field and Op code, which are relayed to the control units to generate appropriate signals for instruction execution.

A multiplexer labeled RegDst is incorporated to select between Rt and Rd, contingent upon the RegDst signal. Following this selection, another multiplexer determines whether to utilize the chosen register or R7 (in the case of executing the JAL instruction). Subsequently, a final multiplexer selects between the last register and R1 (if executing the LUI instruction), enabling the determination of the register to write into within the Register File via the Write Register input.

AddItionally, a multiplexer labeled LUI is connected to the Write Data input of the Register File, allowing for the selection between the value derived from the JAL multiplexer or the value necessary for executing the LUI instruction.

For the LUI instruction, involving the direct loading of the upper 11 immediate value into R1, a specific sequence is followed: initially, the first 11 bits (bits 0 to 10) are extracted from the instruction, then extended to 16 bits, and finally left-shifted by 5 bits to prepare it for loading into R1.

The Register File boasts two outputs: Read Data 1 and Read Data 2. The former serves as the first input to the Arithmetic Logic Unit (ALU), while the latter is directed to the multiplexer ALUSrc, which selects between Read Data 2 or the immediate value from the ExtOp multiplexer, which determines whether the immediate value is signed or unsigned, depending on the ExtOp signal.

Control over the ALU is governed by the ALU Control Unit, which determines the appropriate signals based on the instruction to be executed. The ALU yields three outputs: Result, Set, and Zero_Flag. The second and third outputs are relayed to the Branch Control Unit to determine the desired Branch instruction. The Result output may also be directed to the Data Memory as an Address or to the MemtoReg multiplexer, contingent upon whether a Memory-Related instruction is being executed.

The input to the Data Memory is linked to the Read Data 2 output from the Register File, ensuring that this value enters the memory for processing.

Furthermore, the ALU Result is directed to the MemtoReg multiplexer, which selects the value to be written back into the Register

File: either the Data Memory output or the ALU result, depending on the instruction being executed. Another multiplexer labeled JAL selects between the chosen value from the previous one or PC+1 (derived from the Next PC block) in the case of executing the JAL instruction.

• Control Units

Control units generate required signals which determine suitable data path to execute each instruction in the processor.

We have 3 control units each one generates special signals as follow:

1- Main Control Unit

Signals: (RegWr, RegDst, J, Jal, Lui, ExtOp, ALUSrc, MemRd, MemWr, MemtoReg)

2- ALU Control Unit

Signals: (Carry_In, Ainvert, Binvert, ShfCtrl, ResCtrl, Operation, Jr)

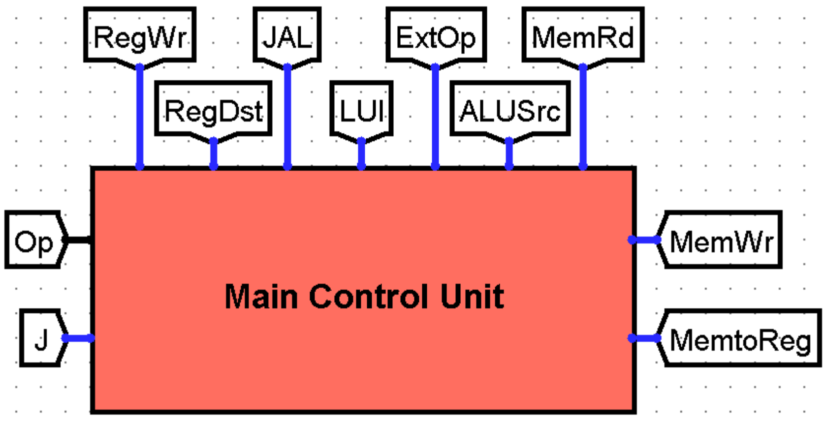
3- Branch Control Unit

Signals: (Branch)

Now we will describe each one in details:

• Main Control Unit

Overview



It is the responsible unit for Data Path signals. It has 1 input which is 5-Bit Op code and 10 outputs signals as follow:

RegWr : Enable/disable writing in Register File.

RegDst : Determine either Rt or Rd will be destination register.

J, Jal, Lui : Responsible for their instructions as we will explain later.

ExtOp : Determine either Imm^{11} will be Signed or Unsigned extended.

ALUSrc : Determine either Rt or Imm^{11} will be directed to ALU.

MemRd : Enable loading from the Data Memory.

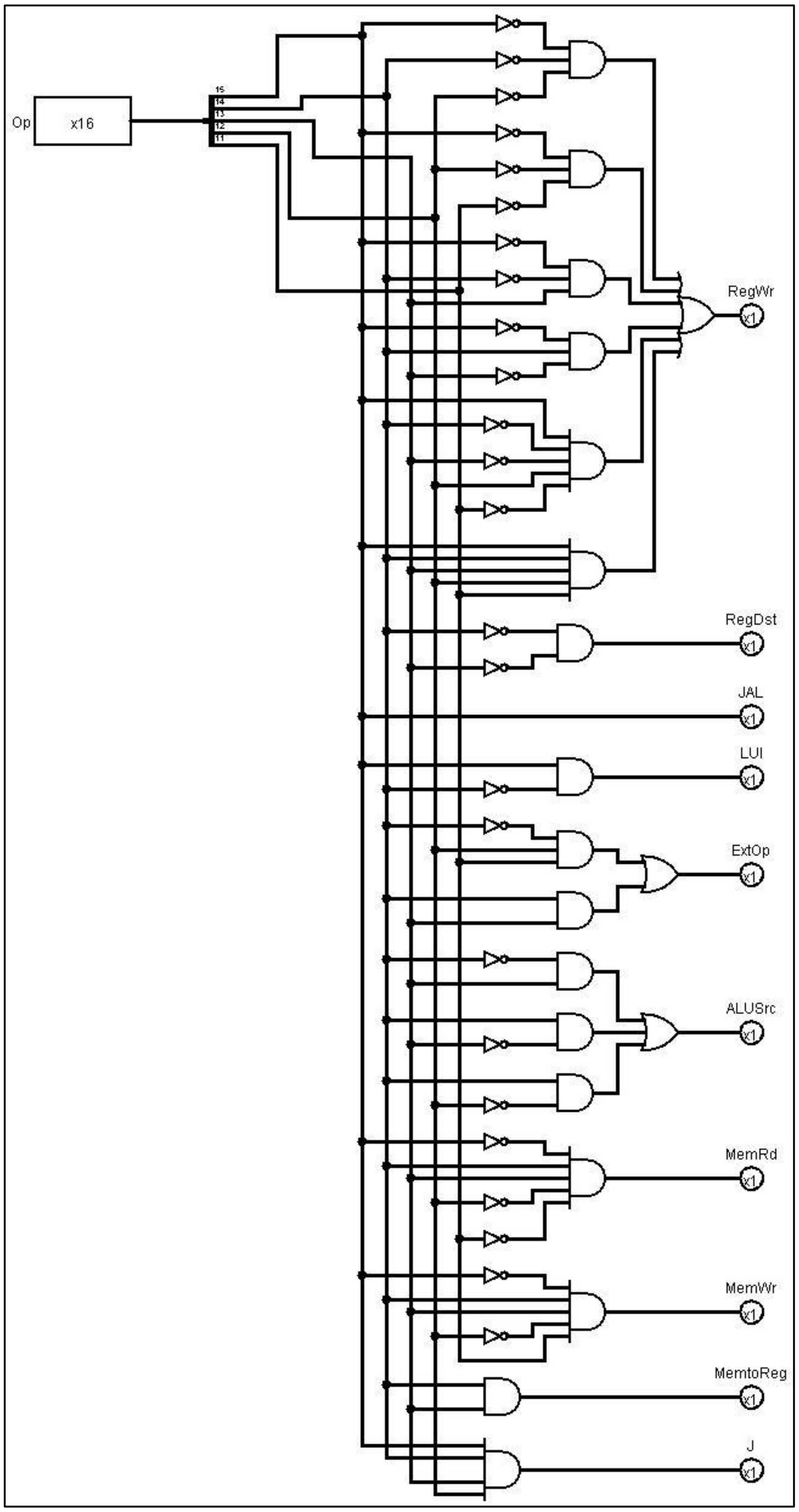
MemWr : Enable storing in the Data Memory.

MemtoReg : Selects either Data Memory output or the ALU result will be written back in the Register File.

Structure

We used a splitter (instead of decoder) to take only last 5 bits in the instructions as the Op code which generates the signals.

We didn't Add R-Type signals as ALU control unit can take last 5 bits in the instructions as the Op code directly.



Instructions Truth Table

NO	Instruction	Op	RegWr	RegDst	ExtOp	ALUSrc	MemRd	MemWr	MemtoReg	Jal	Lui	J
1	R-TYPE	0	1	1	x	0	0	0	0	0	0	0
2	R-TYPE	1	1	1	x	0	0	0	0	0	0	0
3	JR	2	0	x	x	x	0	0	x	x	x	0
4	ANDI	4	1	0	0	1	0	0	0	0	0	0
5	ORI	5	1	0	0	1	0	0	0	0	0	0
6	XORI	6	1	0	0	1	0	0	0	0	0	0
7	ADDI	7	1	0	1	1	0	0	0	0	0	0
8	SLL	8	1	0	0	1	0	0	0	0	0	0
9	SRL	9	1	0	0	1	0	0	0	0	0	0
10	SRA	10	1	0	0	1	0	0	0	0	0	0
11	ROR	11	1	0	0	1	0	0	0	0	0	0
12	LW	12	1	0	1	1	1	0	1	0	0	0
13	SW	13	0	x	1	1	0	1	x	x	x	0
14	BEQ	14	0	x	1	0	0	0	x	x	x	0
15	BNE	15	0	x	1	0	0	0	x	x	x	0
16	BLT	16	0	x	1	0	0	0	x	x	x	0
17	BGE	17	0	x	1	0	0	0	x	x	x	0
18	LUI	18	1	x	x	x	0	0	x	x	1	0
19	J	30	0	x	x	x	0	0	x	x	x	1
20	JAL	31	1	x	x	x	0	0	x	1	0	1

Signals Logic Equations

$\text{RegWr} = \overline{(\text{Jr} + \text{SW} + \text{BEQ} + \text{BNE} + \text{BLT} + \text{BGE} + \text{J})}$

RegDst = (AND + OR + XOR + NOR + Add + Sub + SLT + SLTU)

Jal, J = (JAL)

Lui = (LUI)

ExtOp = (AddI + LW + SW + BEQ + BNE + BLT + BGE)

ALUSrc = (ANDI + ORI + XORI + AddI + SLL + SRL + SRA + ROR + LW + SW)

MemRd, MemtoReg = (LW)

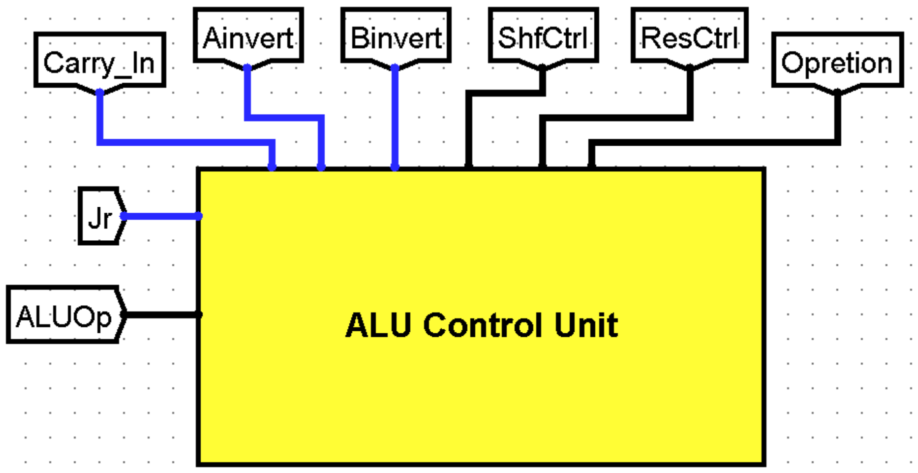
MemWr = (SW)

• ALU Control Unit

Overview

It is the responsible unit for ALU operations.

It has 1 input which is the 5-Bit Op code merged with 2-Bit Function, and 7 outputs as follow:



Carry_In : Adds 1 to 1-Bit ALU.

Ainvert : Inverting operand A.

Binvert : Inverting operand B.

ShfCtrl : Selects between Shift and Rotate instructions.

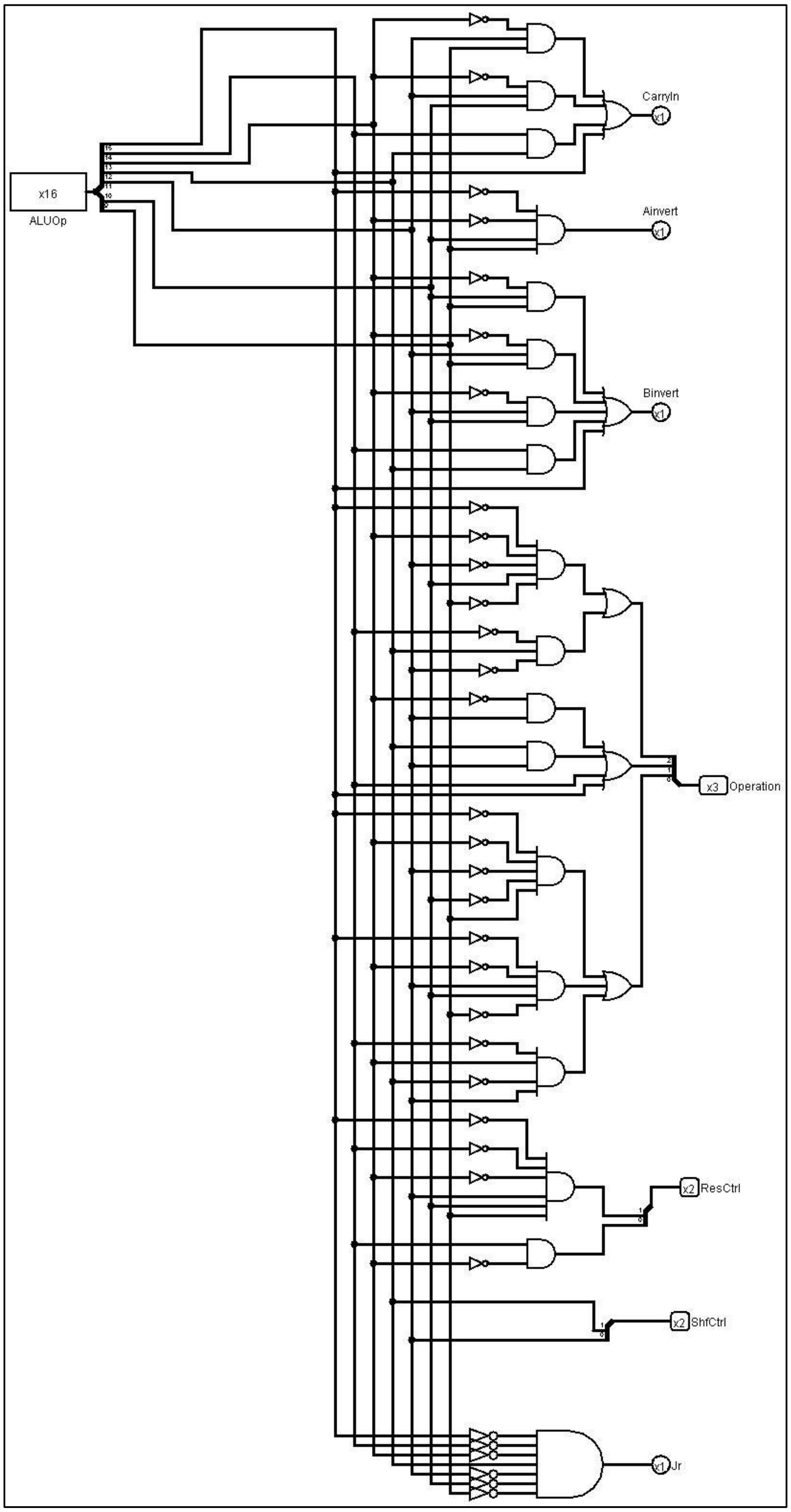
ResCtrl : Selects result type.

Operation : Selects which operation will be done in 1-Bit ALU.

Jr : Selects Register Rs to be loaded in Program Counter.

Structure

We used 2 splitter (instead of decoder), upper one take only last 5 bits in the instructions as the Op code and lower one take Bits 9 and 10 as the Function and both together determine which instruction to be executed in the ALU.



Instructions Truth Table

NO	Instruction	Op	F	Carry_In	AInvert	BInvert	Operation	ResCtrl	ShfCtrl
1	AND	0	0	x	0	0	000	00	xx
2	OR	0	1	x	0	0	001	00	xx
3	XOR	0	2	x	0	0	100	00	xx
4	NOR	0	3	x	1	1	000	00	xx
5	Add	1	0	0	0	0	010	00	xx
6	Sub	1	1	1	0	1	010	00	xx
7	SLT	1	2	1	0	1	011	00	xx
8	SLTU	1	3	x	x	x	xxx	10	xx
9	Jr	2	0	x	x	x	xxx	xx	xx
10	ANDI	4	x	x	0	0	000	00	xx
11	ORI	5	x	x	0	0	001	00	xx
12	XORI	6	x	x	0	0	100	00	xx
13	AddI	7	x	0	0	0	010	00	xx
14	SLL	8	x	x	x	x	xxx	01	00
15	SRL	9	x	x	x	x	xxx	01	01
16	SRA	10	x	x	x	x	xxx	01	10
17	ROR	11	x	x	x	x	xxx	01	11
18	LW	12	x	0	0	0	010	00	xx
19	SW	13	x	0	0	0	010	00	xx
20	BEQ	14	x	1	0	1	010	00	xx
21	BNE	15	x	1	0	1	010	00	xx
22	BLT	16	x	1	0	1	010	00	xx
23	BGE	17	x	1	0	1	010	00	xx

Signals Logic Equations

Carry_In = (Sub + SLT + BEQ + BNE + BLT + BGE)

AInvert = (NOR)

BInvert = (NOR + Sub + SLT + BEQ + BNE + BLT + BGE)

Operation2 = (XOR + XORI)

Operation1 = (Add + Sub + SLT + AddI + LW + SW + BEQ + BNE + BLT + BGE)

Operation0 = (OR + SLT + ORI)

ResCtrl1 = (SLTU)

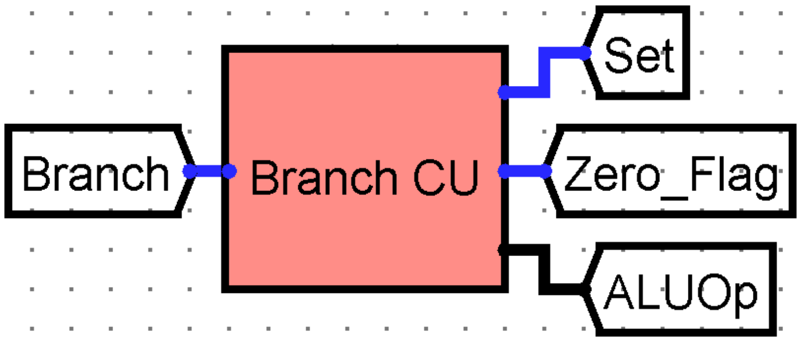
ResCtrl0 = (SLL + SRL + SRA + ROR)

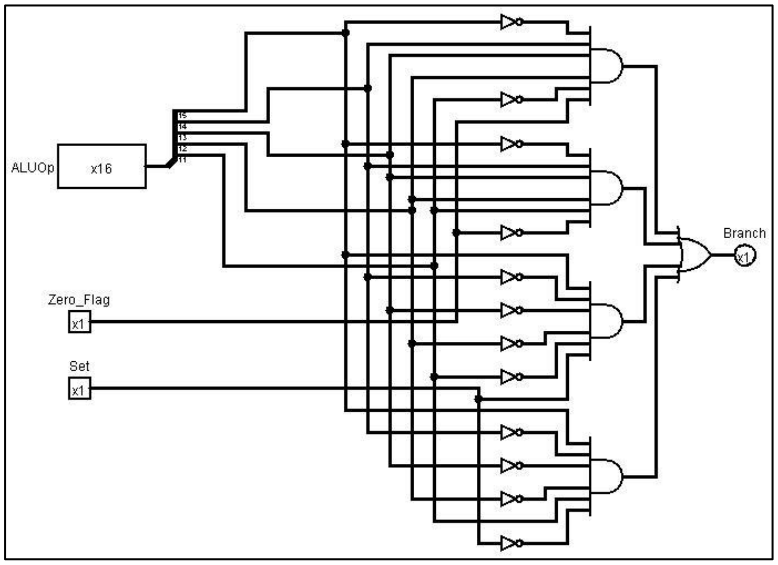
ShfCtrl1 = (SRA + ROR)

ShfCtrl0 = (SRL + ROR)

• Branch Control Unit

It is the unit that executes branch instructions. It has 3 Inputs, Set and Zero_Flag are signals generated by ALU and the same 5-Bit Op code.It has 1 output Branch signal that execute the branch at Next PC Block.





Instructions Truth Table

NO	Instruction	Op	Zero_Flag	Set
1	**BEQ**	**14**	1	x
2	**BNE**	**15**	0	x
3	**BLT**	**16**	x	1
4	**BGE**	**17**	x	0

Signals Logic Equations

$\mathrm{Zero\_Flag} = (\mathrm{BEQ} + \overline{\mathrm{BNE}})$

$\mathrm{Set} = (\mathrm{BLT} + \overline{\mathrm{BGE}})$

$\mathrm{Branch} = \mathrm{BEQ}.\mathrm{Zero\_Flag} + \overline{\mathrm{BNE}}\,.\,\mathrm{Zero\_Flag} + \mathrm{BLT}\,.\,\mathrm{Set} + \overline{\mathrm{BGE}}\,.\,\mathrm{Set}$

Simulation & Testing

- **First Test Code .**

NO	Instruction	Hexa	Expected Value
1	AddI R1, R1, 1	3849	R1=0x1
2	SW R1,0(R0)	6801	mem[0]=0x0001
3	SW R1,1(R0)	6841	mem[1]=0x0001
4	AddI R2, R0, 10	3A82	R2=0x000a
5	SW R2, 2(R0)	6882	mem[2]=0x000a
6	LUI 1	9001	R1=0x0020
7	ORI R2, R1, 28	2F0A	R2=0x003c
8	LUI 536	9218	R1=0x4300
9	ORI R1, R1, 10	2A89	R1=0x430a
10	SW R1 ,0(R2)	6811	mem[60]=0x430a
11	LUI 922	939A	R1=0x7340
12	ORI R3, R1, 2	288B	R3=0x7342
13	SW R3,1(R2)	6853	mem[61]=0x7342
14	AddI R1, R0, 0.	3801	R1=0x0000
15	AddI R2, R0, 0.	3802	R2=0x0000
16	AddI R3, R0, 0.	3803	R3=0x0000
17	Lui 0x384	9384	R1=0X7080
18	AddI R5, R1,13	3B4D	R5=0x708d
19	XOR R3, R1, R5	04CD	R3=0x000d

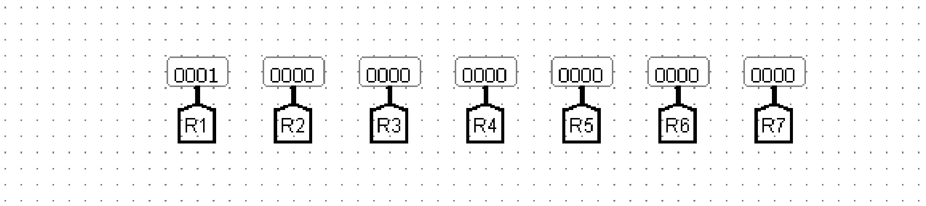
20	LW R1, 0(R0)	6001	R1=0x0001
21	LW R2, 1(R0)	6042	R2=0x0001
22	LW R3, 2(R0)	6083	R3=0x000a
23	AddI R4, R4, 10	3AA4	R4=0x000a
24	Sub R4, R4, R4	0B24	R4=0x0000
25	L2: Add R4, R2, R4	0914	R4=0x0001, for last loop: R4=0x0037
26	Slt R6, R2, R3	0D93	R6=0x0001, for last loop:R6=0x0000
27	BEQ R6, R0, L1	70F0	taken for last loop (br L1)
28	Add R2, R1, R2	088A	R2=0x0002
29	BEQ R0, R0, L2	7700	TAKEN ,(br L2)
30	L1: SW R4, 0(R0)	6804	mem[0]=0x0037
31	Jal Func	F804	R7=0x001f (JUMPING TO FUNC)
32	SLL R3, R2, 6	4193	R3=0xc280
33	ROR R6, R3, 3	58DE	R6=0x1850
34	BEQ R0,R0,-1	7000	stop condition
35	Func: OR R5, R2, R3	0353	R5=0x000a
36	LW R1, 0(R0)	6001	R1=0x0037
37	LW R2, 5(R1)	614A	R2=0x430a
38	LW R3 ,6(R1)	618B	R3=0x7342
39	And R4, R2, R3	0113	R4=0x4302
40	SW R4, 0(R0)	6804	mem[0]=0x4302
41	Jr R7	1038	R7=0x001f (JUMPING TO SLL)

We decode each program according to the format of each instruction type (R- Type, I-Type and J-Type), after that we decode it to binary number

First Code Description

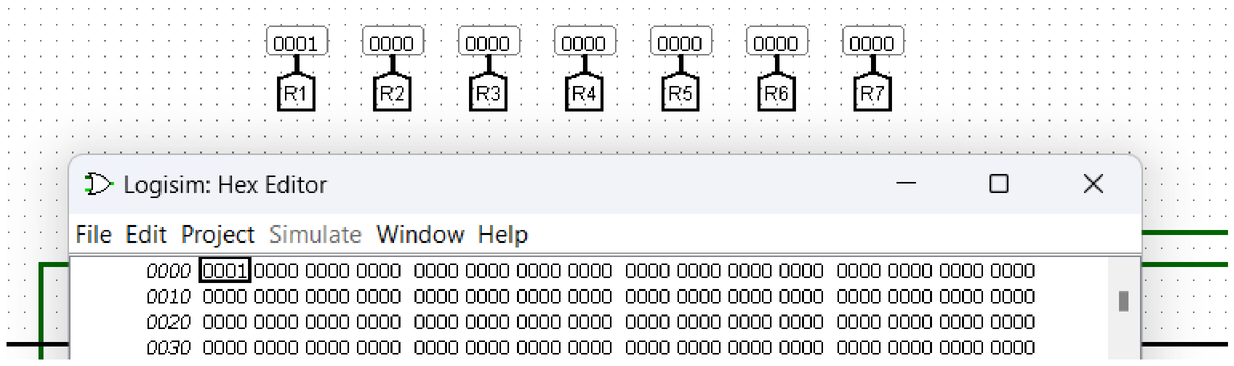
AddI R1, R1, 1

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R1 as Rs and the 5 bits signed extend immediate value which is 1 and AddIng them then writes the result in the destination register R1.



SW R1, 0(R0)

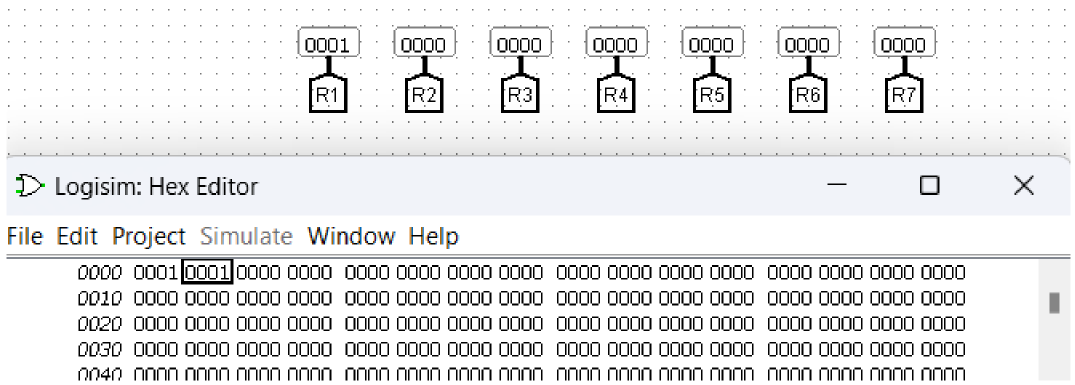
MEM[REg(Rs) + signed(Imm5)]= Reg(Rt), writing the value from Register R1 in the memory by using Displacement Addressing mode.The effective memory Address is the content of R0 Added with the sign_extend (Imm5) which is 0.



```
0000 0001 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0010 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0020 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0030 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

SW R1, 1(R0)

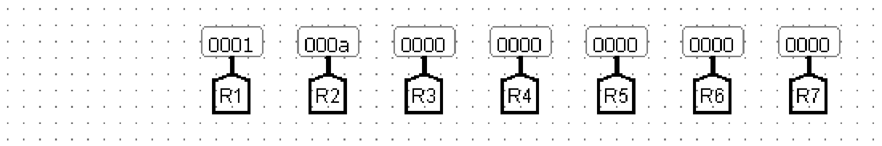
MEM[REg(Rs) + signed(Imm5)]= Reg(Rt), writing the value from Register R1 in the memory by using Displacement Addressing mode.The effective memory Address is the content of R0 Added with the sign_extend (Imm5) which is 1.



```
0000  0001 0001 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0010  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0020  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0030  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0040  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

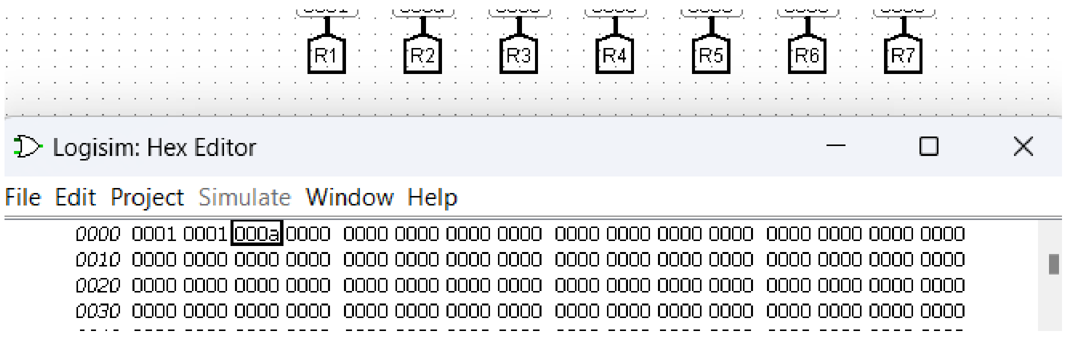
AddI R2, R0, 10

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R0 as Rs and the 5 bits signed extend immediate value which is 10 and AddIng them then writes the result in the destination register R2



SW R2, 2(R0)

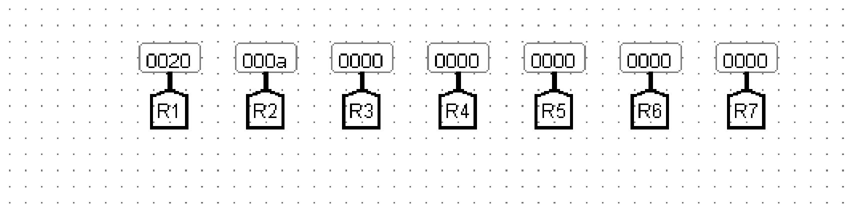
MEM[REg(Rs) + signed(Imm5)]= Reg(Rt), writing the value from Register R2 in the memory by using Displacement Addressing mode.The effective memory Address is the content of R0 Added with the sign_extend (Imm5) which is 2.



```
0000 0001 0001 000a 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0010 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0020 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0030 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

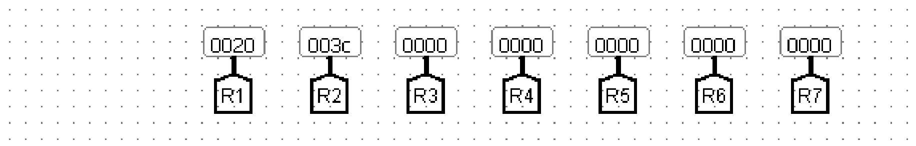
LUI 1

R1 = Imm11 << 5 , The immediate value 1 is loaded in upper 11 bit of R1 with lower 5 bit set to 0s, by shifting the immediate value by 5 bits to the left



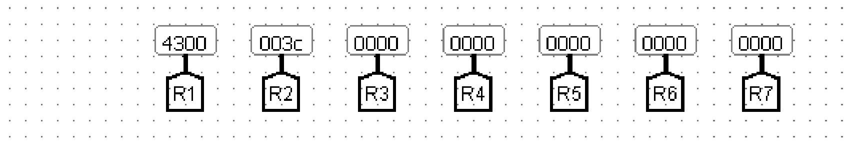
ORI R2, R1, 28

Reg(Rt)= REg(Rs) | (Imm5), reading register R1 as Rs and the 5 bits unsigned extend immediate value which is 28 and oring them then writes the result in the destination register R2.



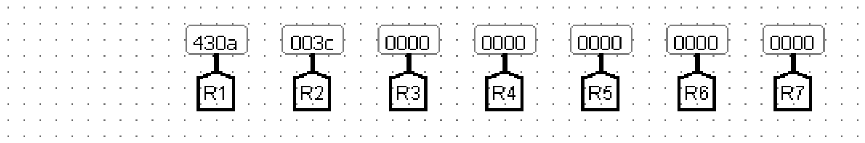
LUI 536

R1 = Imm11 << 5 , The immediate value 536 is loaded in upper 11 bit of R1 with lower 5 bit set to 0s, by shifting the immediate value by 5 bits to the left



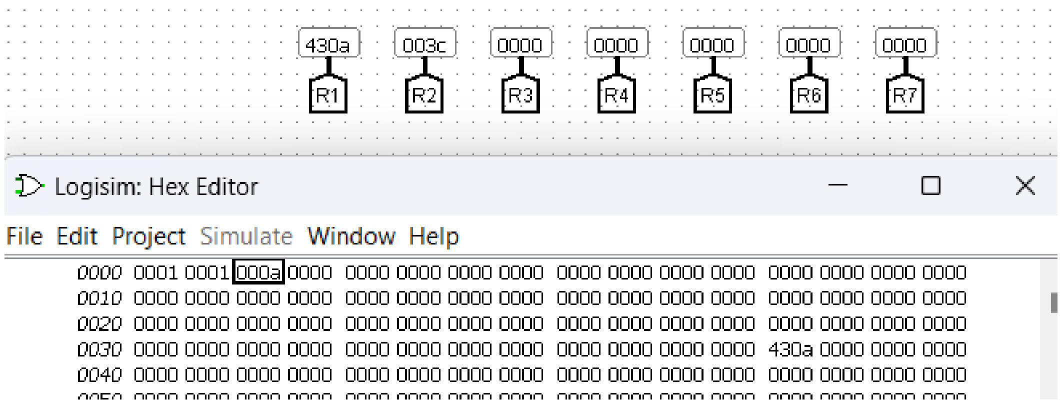
ORI R1, R1, 10

Reg(Rt)= REg(Rs) | (Imm5), reading register R1 as Rs and the 5 bits unsigned extend immediate value which is 10 and oring them then writes the result in the destination register R1.



SW R1, 0(R2)

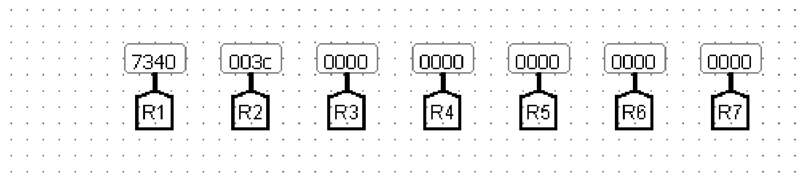
MEM[REg(Rs) + signed(Imm5)]= Reg(Rt), writing the value from Register R1 in the memory by using Displacement Addressing mode.The effective memory Address is the content of R2 Added with the sign_extend (Imm5) which is 0.



```
0000 0001 0001 000a 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0010 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0020 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0030 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  430a 0000 0000 0000
0040 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

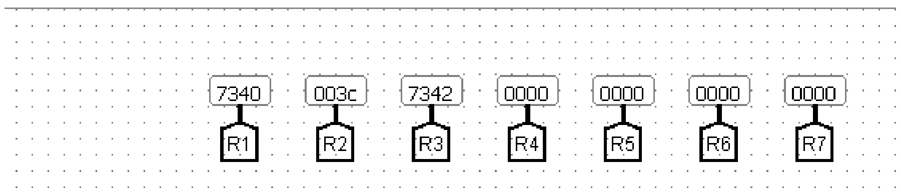
LUI 922

R1 = Imm11 << 5 , The immediate value 922 is loaded in upper 11 bit of R1 with lower 5 bit set to 0s, by shifting the immediate value by 5 bits to the left



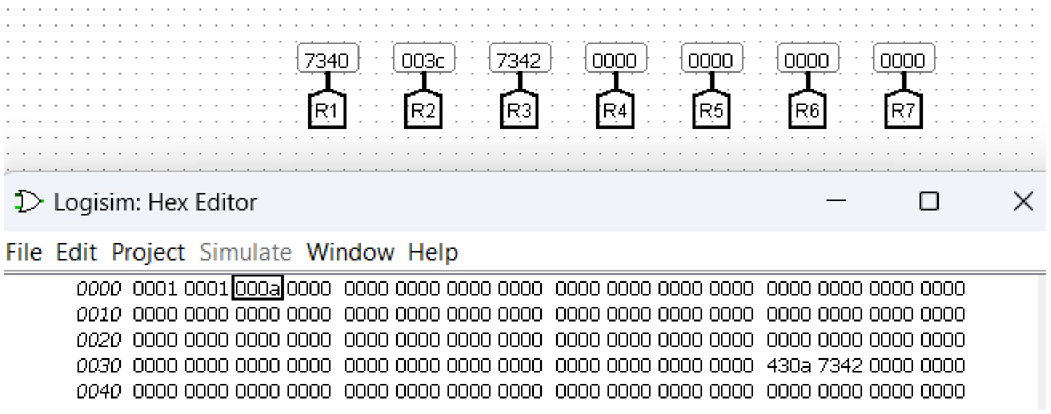
ORI R3, R1, 2

Reg(Rt)= REg(Rs) | (Imm5), reading register R1 as Rs and the 5 bits unsigned extend immediate value which is 2 and oring them then writes the result in the destination register R3.



SW R3, 1(R2)

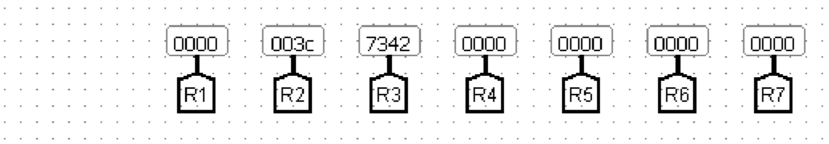
MEM[REg(Rs) + signed(Imm5)]= Reg(Rt), writing the value from Register R3 in the memory by using Displacement Addressing mode.The effective memory Address is the content of R2 Added with the sign_extend (Imm5) which is 1.



```
0000 0001 0001 000a 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0010 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0020 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0030 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  430a 7342 0000 0000
0040 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

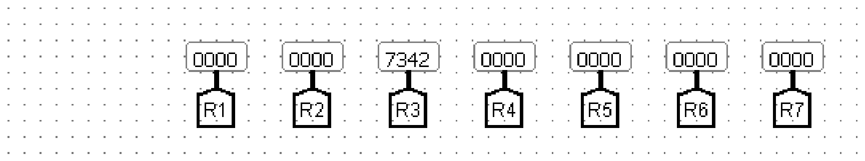
AddI R1,R0, 0

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R0 as Rs and the 5 bits signed extend immediate value which is 0 and AddIng them then writes the result in the destination register R1.



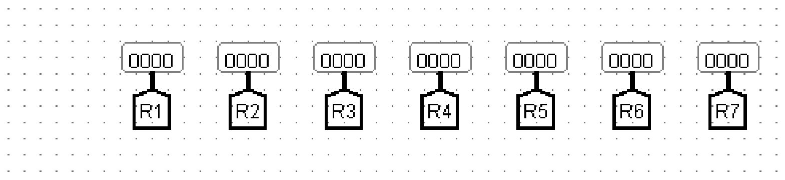
AddI R2, R0, 0

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R0 as Rs and the 5 bits signed extend immediate value which is 0 and AddIng them then writes the result in the destination register R2.



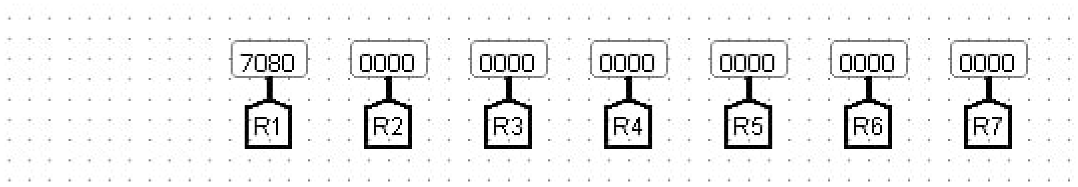
AddI R3, R0, 0

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R0 as Rs and the 5 bits signed extend immediate value which is 0 and AddIng them then writes the result in the destination register R3.



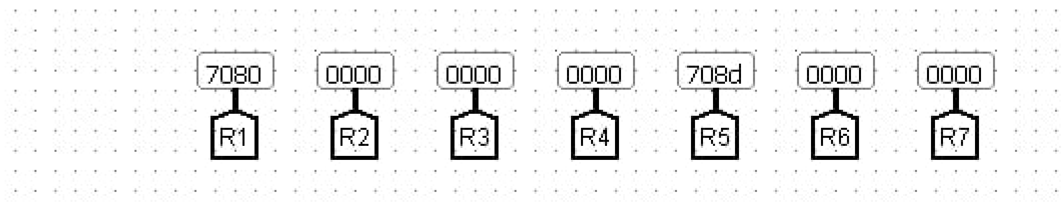
LUI 0x384

R1 = Imm11 << 5 , The immediate value 0x384 is loaded in upper 11 bit of R1 with lower 5 bit set to 0s, by shifting the immediate value by 5 bits to the left



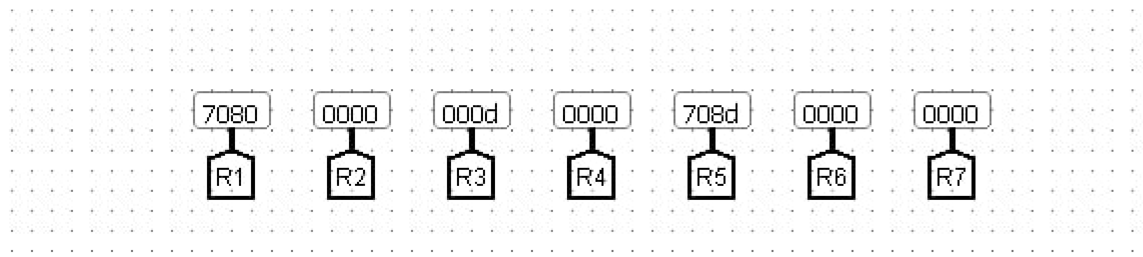
AddI R5, R1,13

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R1 as Rs and the 5 bits signed extend immediate value which is 13 and AddIng them then writes the result in the destination register R5.



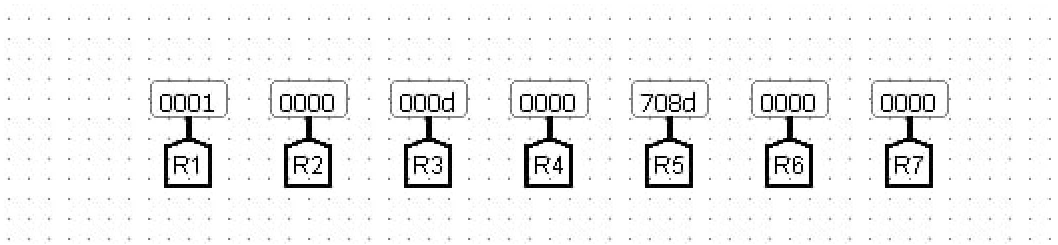
XOR R3, R1, R5

Reg(Rd)= REg(Rs) ^ Reg(Rt), reading register R1 as Rs and register R5 as RT and oring them then writes the result in the destination register R3.



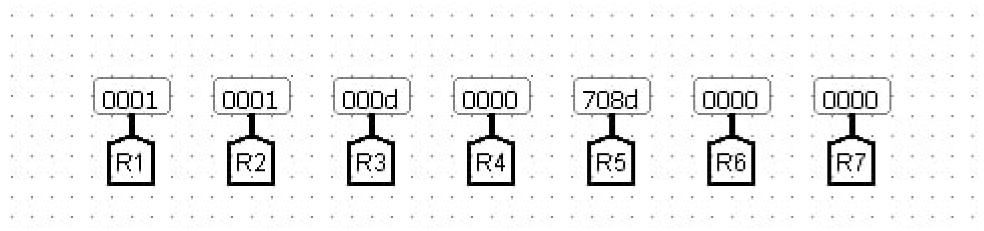
LW R1, 0(R0)

Reg(Rt) = MEM[REg(Rs) + signed(Imm5)], writing the value from the memory in the register R1 as Rt by using Displacement Addressing mode.The effective memory Address is Reg(R0) AddIng with the sign-extend (Imm5) which is 0.



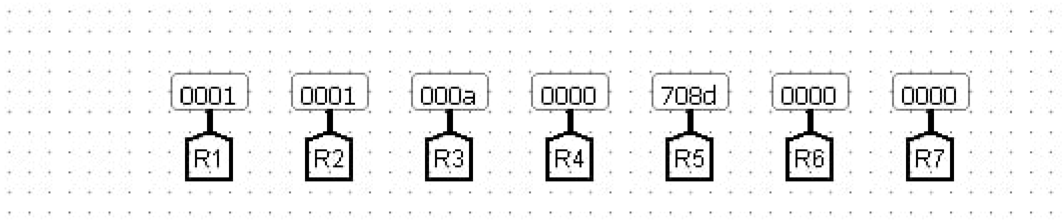
LW R2, 1(R0)

Reg(Rt) = MEM[REg(Rs) + signed(Imm5)], writing the value from the memory in the register R2 as Rt by using Displacement Addressing mode.The effective memory Address is Reg(R0) AddIng with the sign-extend (Imm5) which is 1.



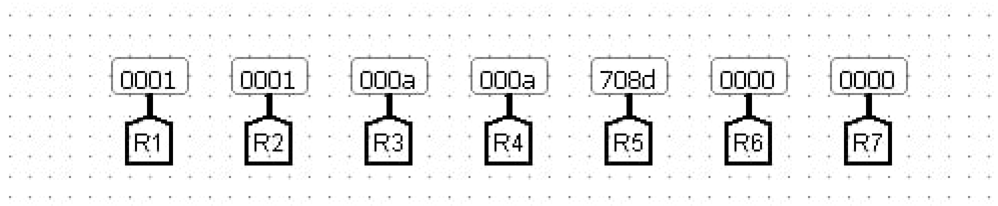
LW R3, 2(R0)

Reg(Rt) = MEM[REg(Rs) + signed(Imm5)], writing the value from the memory in the register R3 as Rt by using Displacement Addressing mode.The effective memory Address is Reg(R0) AddIng with the sign-extend (Imm5) which is 2.



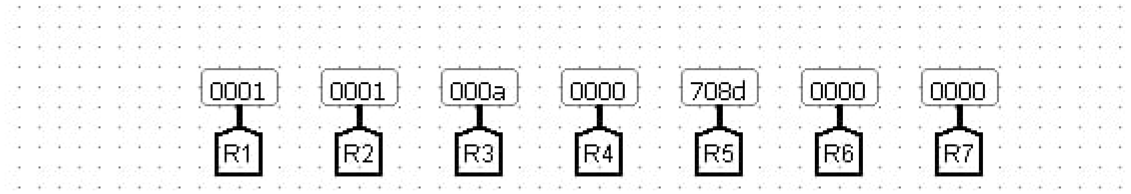
AddI R4, R4, 10

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R4 as Rs and the 5 bits signed extend immediate value which is 10 and AddIng them then writes the result in the destination register R4.



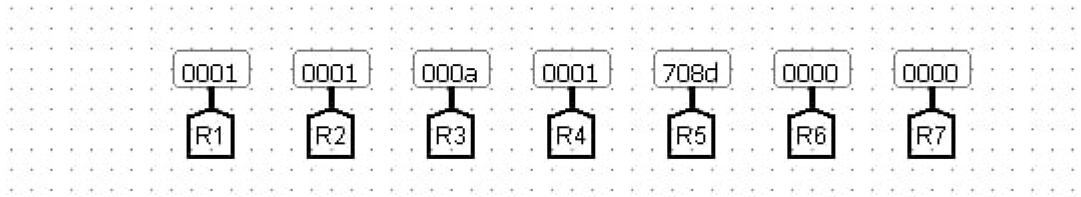
Sub R4, R4, R4

Reg(Rt)= REg(Rs) -REg(rt) ,reading register R4 as Rs and the registerR4 asRt and AddIng them then writes the result in the destination register R4.



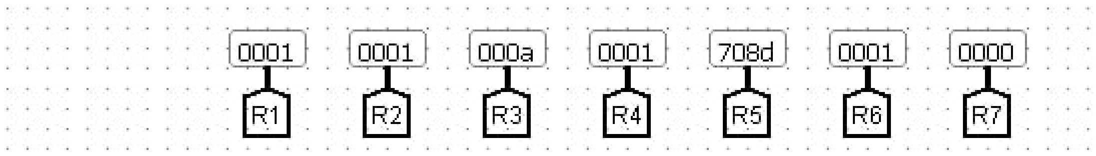
L2: Add R4, R2, R4

Reg(Rt)= REg(Rs) + REg(rt) ,reading register R2 as Rs and the registerR4 as Rt and AddIng them then writes the result in the destination register R4.



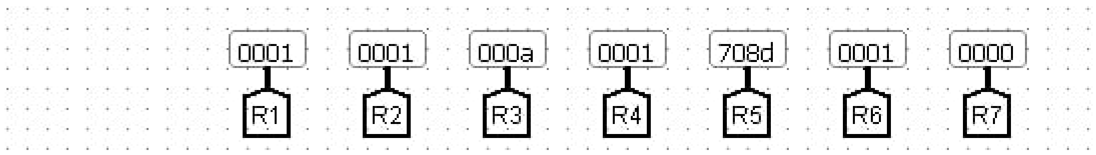
Slt R6, R2, R3

Reg(Rd)= (REg(Rs) <s Reg(Rt)) ?1:0 ,reading register R2 as Rs and register R3 as Rt and compare both of them ;if the condition is true then writes in R6 as RD a value 1;else write 0.



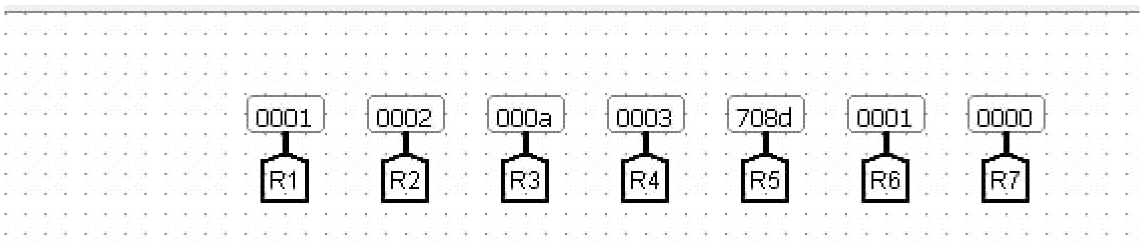
BEQ R6, R0, L1

Branch if REg(Rs) == Reg(Rt) , reading register R0 as Rs, and register R6 as Rt and Branch if "Reg(R0) == Reg(R6)" which is aLWays wrong (until the last loop) then it doesn't branch and goes to next instruction.



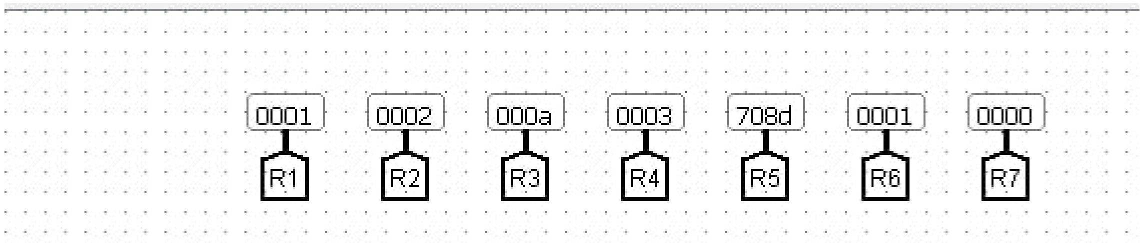
Add R2, R1, R2

Reg(Rt)= REg(Rs) + REg(rt) ,reading register R1 as Rs and the registerR2 as Rt and AddIng them then writes the result in the destination register R2.



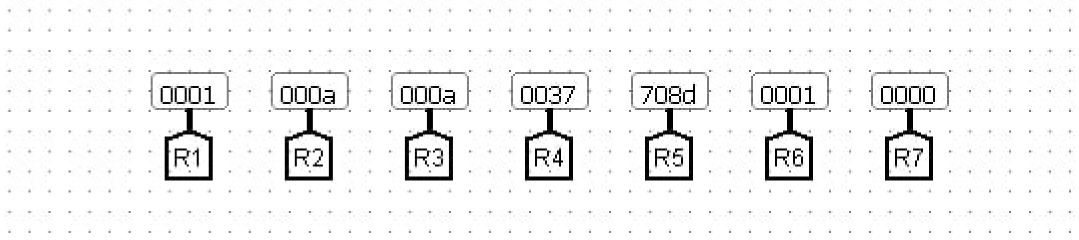
BEQ R0, R0, L2

Branch if REg(Rs) == Reg(Rt) , reading register R0 as Rs, and register R0 as Rt and Branch if "Reg(R0) == Reg(R0)" which is aLWays true then it branch and goes to L2.



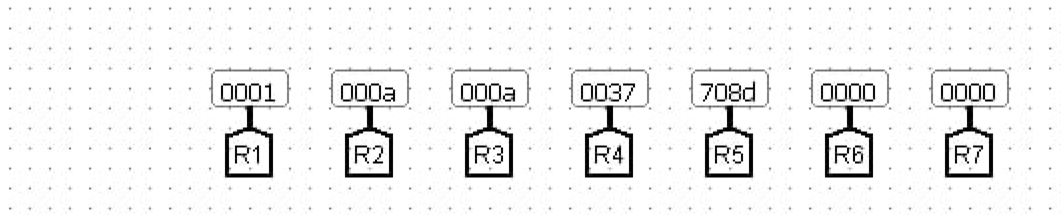
L2: Add R4, R2, R4 (for last loop)

Reg(Rt)= REg(Rs) + REg(rt) ,reading register R2 as Rs and the registerR4 as Rt and AddIng them then writes the result in the destination register R4.



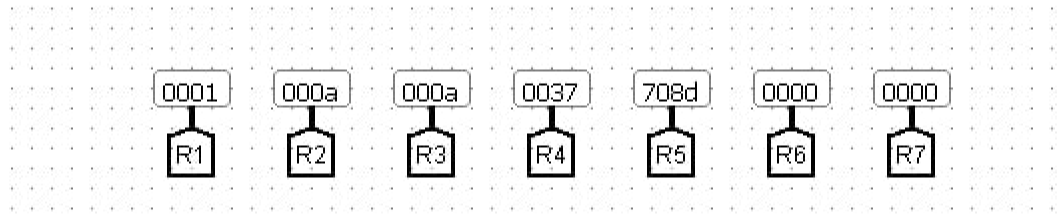
Slt R6, R2, R3 (for last loop)

Reg(Rd)= (REg(Rs) <s Reg(Rt)) ?1:0 ,reading register R2 as Rs and register R3 as Rt and compare both of them ;if the condition is true then writes in R6 as RD a value 1;else write 0.



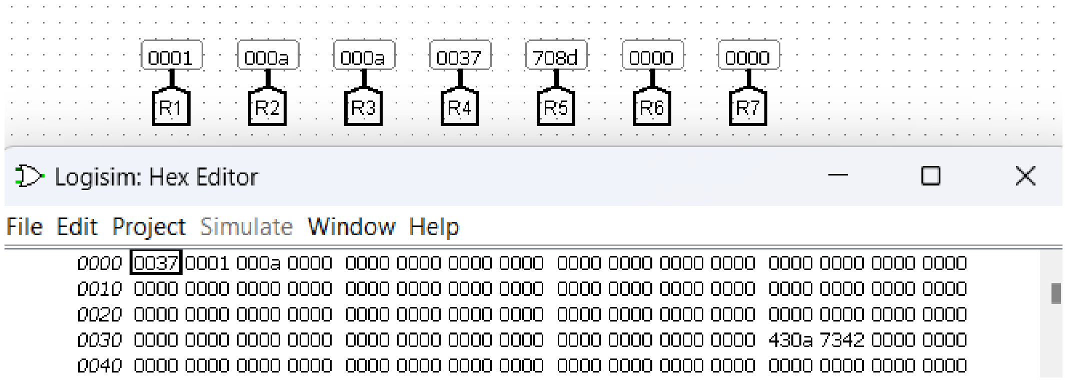
BEQ R6, R0, L1 (for last loop)

Branch if REg(Rs) == Reg(Rt) , reading register R0 as Rs, and register R6 as Rt and Branch if "Reg(R0) == Reg(R0)" which is true then it branch and goes to L1.



L1: SW R4, 0(R0)

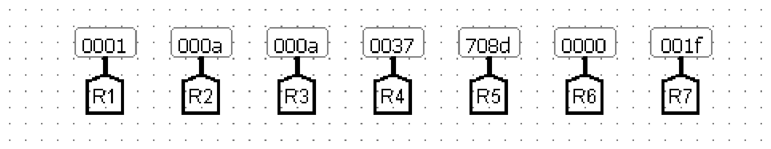
MEM[REg(Rs) + signed(Imm5)]= Reg(Rt), writing the value from Register R4 in the memory by using Displacement Addressing mode.The effective memory Address is the content of R0 Added with the sign_extend (Imm5) which is 0.



```
0000 0037 0001 000a 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0010 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0020 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0030 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  430a 7342 0000 0000
0040 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

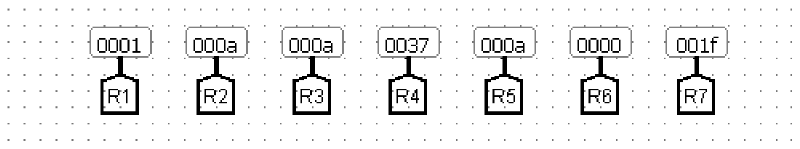
JAL Func

R7 = PC + 1 , PC= PC + Signed(Imm11), writing the return Address (PC+1) in register R7 so it jump to the label func at (instruction in pc = 0x23) set R7 by 0x1f.



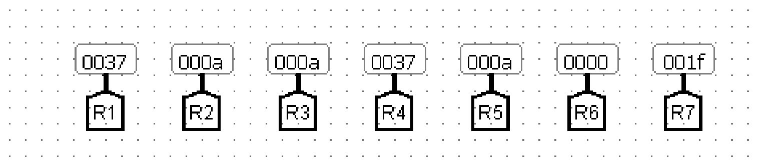
Func: OR R5, R2, R3

Reg(Rd)= REg(Rs) | REg(Rt), reading register R2 as Rs and R3 as Rt and oring them then writes the result in the destination register R5.



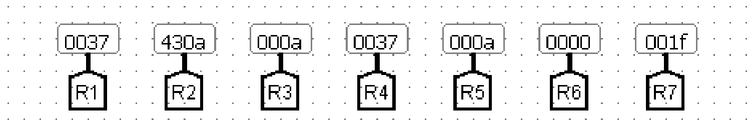
LW R1, 0(R0)

Reg(Rt) = MEM[REg(Rs) + signed(Imm5)], writing the value from the memory in the register R1 as Rt by using Displacement Addressing mode.The effective memory Address is Reg(R0) AddIng with the sign-extend (Imm5) which is 0.



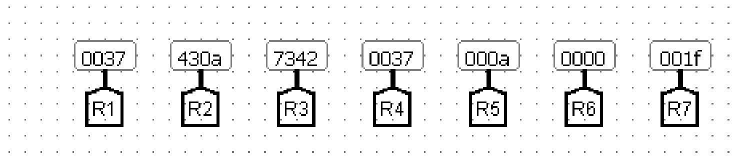
LW R2, 5(R1)

Reg(Rt) = MEM[REg(Rs) + signed(Imm5)], writing the value from the memory in the register R2 as Rt by using Displacement Addressing mode.The effective memory Address is Reg(R1) AddIng with the sign-extend (Imm5) which is 5.



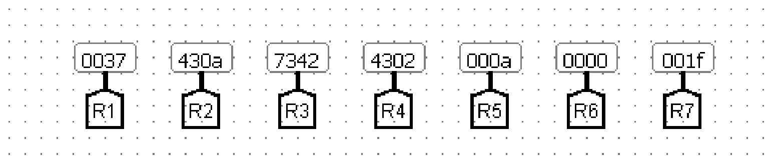
LW R3 ,6(R1)

Reg(Rt) = MEM[REg(Rs) + signed(Imm5)], writing the value from the memory in the register R3 as Rt by using Displacement Addressing mode.The effective memory Address is Reg(R1) AddIng with the sign-extend (Imm5) which is 6.



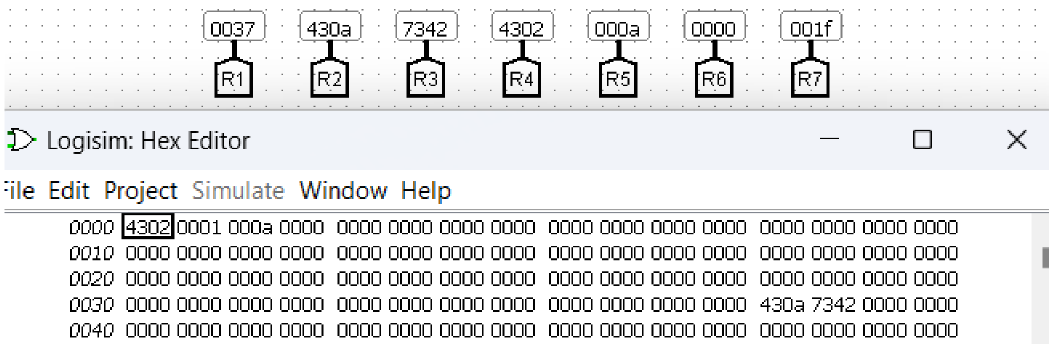
And R4, R2, R3

Reg(Rd)= REg(Rs) & Reg(Rt), reading register R2 as Rs and R3 as Rt and anding them then writes the result in the destination register R4.



SW R4, 0(R0)

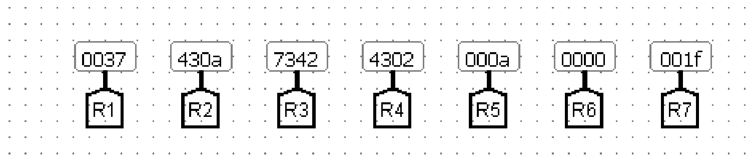
MEM[REg(Rs) + signed(Imm5)]= Reg(Rt), writing the value from Register R4 in the memory by using Displacement Addressing mode.The effective memory Address is the content of R0 Added with the sign_extend (Imm5) which is 0.



```
0000 4302 0001 000a 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0010 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0020 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0030 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  430a 7342 0000 0000
0040 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

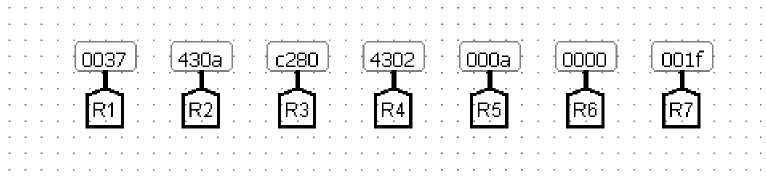
Jr R7

PC=Reg (Rs), that makes PC = Reg(R7) which jump to (instruction in pc = 0x1f).



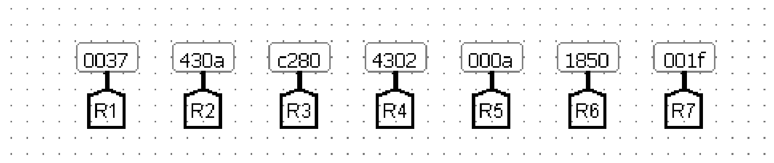
SLL R3, R2, 6

Reg(Rt) = REg(Rs) << (Imm4), reading register R2 as Rs and shifting register R2 by the signed imm = 6 as shift amount to the left by inserting 0's from the right then writes the result in the destination register R3



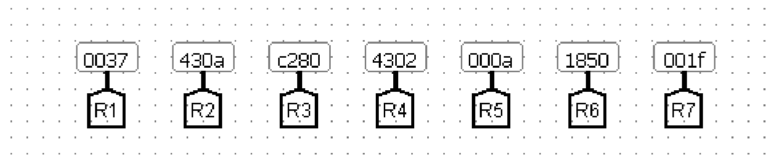
ROR R6, R3, 3

Reg(Rt) = Reg(Rs) rot>> Imm4, reading register R3 as Rs and the 5 bits unsigned extend immediate value which is 3 and shifts the register R3 by 3 bits to Right and reposition each thrown LSB to the MSB of the register then writes the result in the register R6 as Rd.



BEQ R0, R0, 0

Branch if REg(Rs) == Reg(Rt) , reading register R0 as Rs, and register R0 as Rt and Branch if "Reg(R0) == Reg(R0)" which is aLWays true terminating the program.



• Array Test Code

NO	Instruction	Hexa	Expected Value
1	AddI R1, R0, 1	3841	R1=0x0001
2	AddI R2, R0, 2	3882	R2=0x0002
3	AddI R3, R0, 3	38C3	R3=0x0003
4	AddI R4, R0, 4	3904	R4=0x0004
5	AddI R6, R0, 4	3906	R6=0x0004
6	SW R1, 0(R0)	6801	mem[0]=0x0001
7	SW R2, 1(R0)	6842	mem[1]=0x0002
8	SW R3, 2(R0)	6883	mem[2]=0x0003
9	SW R4, 3(R0)	68C4	mem[3]=0x0004
10	jal Sum	F802	R7=0x000a (JUMPING TO SUM)
11	BEQ R0 R0 0	7000	stop condition
12	Sum: AddI R6,R6,4	3936	R6=0x0008
13	SW R1 0(R6)	6831	mem[8]=0x0001
14	SW R2 -1(R6)	6FF2	mem[7]=0x0002
15	SW R3 -2(R6)	6FB3	mem[6]=0x0003
16	SW R4 -3(R6)	6F74	mem[5]=0x0004
17	AddI R4 R0 0	3804	R4=0x0000
18	AddI R3 R0 4	3903	R3=0x0004
19	Loop: LW R2 -1(R1)	67CA	R2=0x0001,for last loop: R2=0x0004
20	add R5 R2 R5	0955	R5=0x0001,for last loop: R5=0x000a

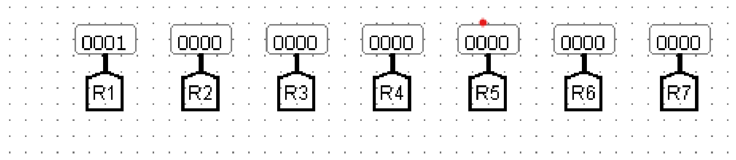
21	AddI R1 R1 1	3849	R1=0x0002,for last loop: R1=0x0005
22	AddI R4 R4 1	3864	R4=0x0001,for last loop: R=0x0004
23	BEQ R4 R3 Exit	70A3	not taken until last loop branch Exit
24	j Loop	F7FB	jumping to label loop (in pc=18)
25	Exit : LW R1 0(R6)	6031	R1=0x0001
26	LW R2 -1(R6)	67F2	R2=0x0002
27	LW R3 -2(R6)	67B3	R3=0x0003
28	LW R4 -3(R6)	6774	R4=0x0004
29	AddI R6 R6 -4	3F36	R6=0x0004
30	Jr R7	1038	pc= 0x000a

Note that, the result of summing of elements of array will be in register R5, while register R6 acts as stack pointer, we take the frame of this function from location 5 to location 8 in Data Memory.

Array Code Description

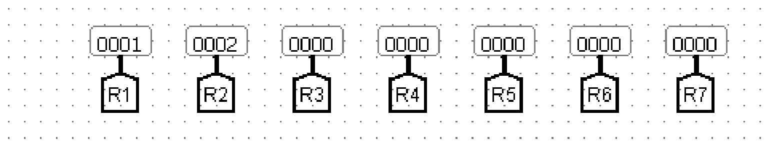
AddI R1, R0, 1

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R0 as Rs and the 5 bits signed extend immediate value which is 1 and AddIng them then writes the result in the destination register R1.



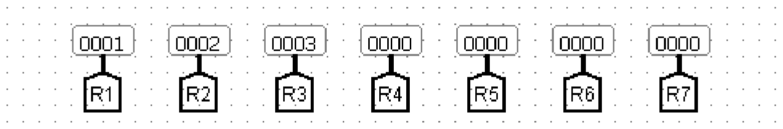
AddI R2, R0, 2

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R0as Rs and the 5 bits signed extend immediate value which is 2 and AddIng them then writes the result in the destination register R2.



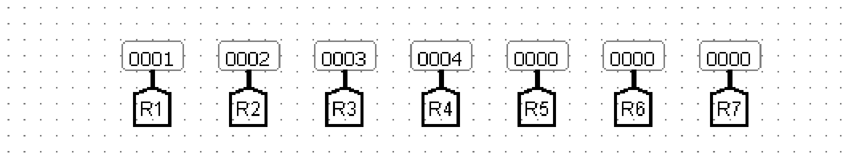
AddI R3, R0, 3

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R0 as Rs and the 5 bits signed extend immediate value which is 3 and AddIng them then writes the result in the destination register R3.



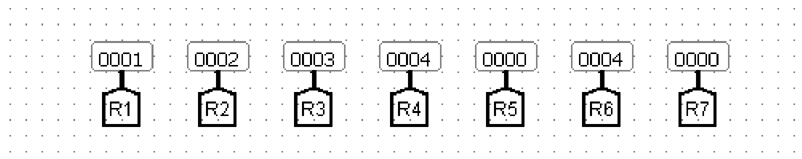
AddI R4, R0, 4

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R0 as Rs and the 5 bits signed extend immediate value which is 4 and AddIng them then writes the result in the destination register R4.



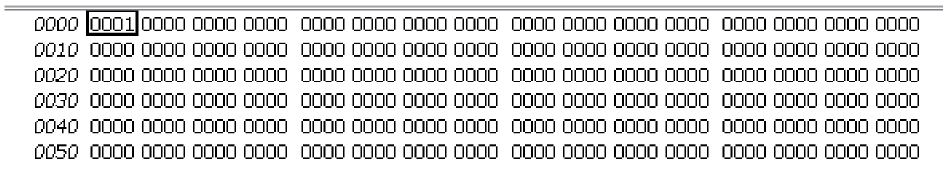
AddI R6, R0, 4

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R0 as Rs and the 5 bits signed extend immediate value which is 4 and AddIng them then writes the result in the destination register R6.



SW R1, 0(R0)

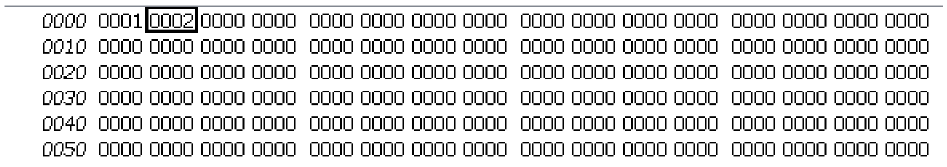
MEM[REg(Rs) + signed(Imm5)]= Reg(Rt), writing the value from Register R1 in the memory by using Displacement addressing mode.The effective memory address is the content of R0 added with the sign_extend (Imm5) which is 0.



```
0000 0001 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0010 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0020 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0030 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0040 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0050 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

SW R2, 1(R0)

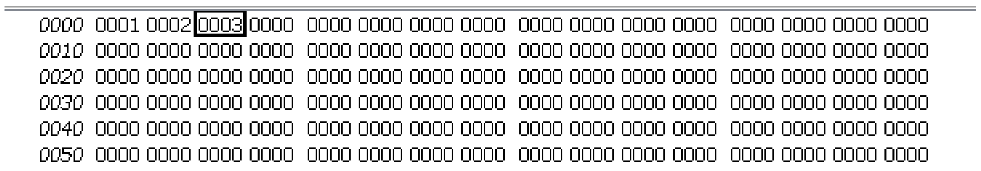
MEM[REg(Rs) + signed(Imm5)]= Reg(Rt), writing the value from Register R2 in the memory by using Displacement addressing mode.The effective memory address is the content of R0 added with the sign_extend (Imm5) which is 1.



```
0000  0001 0002 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0010  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0020  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0030  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0040  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0050  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

SW R3, 2(R0)

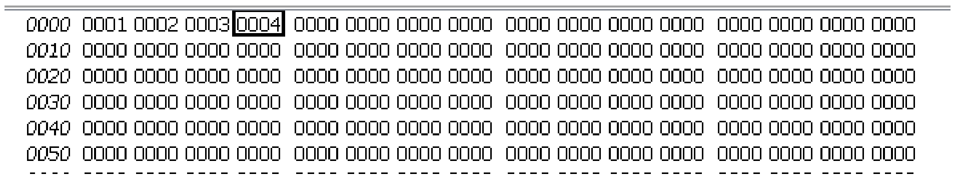
MEM[REg(Rs) + signed(Imm5)]= Reg(Rt), writing the value from Register R3 in the memory by using Displacement addressing mode.The effective memory address is the content of R0 added with the sign_extend (Imm5) which is 2.



```
0000  0001 0002 0003 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0010  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0020  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0030  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0040  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0050  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

SW R4, 3(R0)

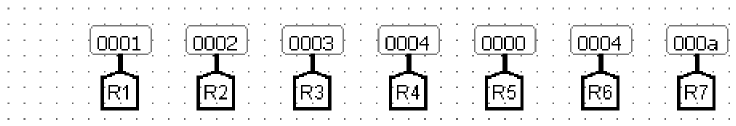
MEM[REg(Rs) + signed(Imm5)]= Reg(Rt), writing the value from Register R4 in the memory by using Displacement addressing mode.The effective memory address is the content of R0 added with the sign_extend (Imm5) which is 3.



```
0000  0001 0002 0003 0004  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0010  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0020  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0030  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0040  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0050  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
----  ---- ---- ---- ----  ---- ---- ---- ----  ---- ---- ---- ----  ---- ---- ---- ----
```

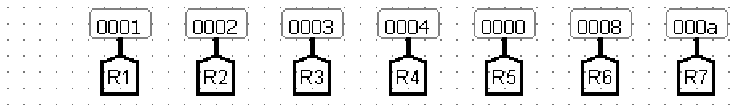
JAL Sum

R7 = PC + 1 , PC= PC + Signed(Imm11), writing the return address (PC+1) in register R7 so it jump to the lapel sum at instruction 0x11and set R7 by 0xa.



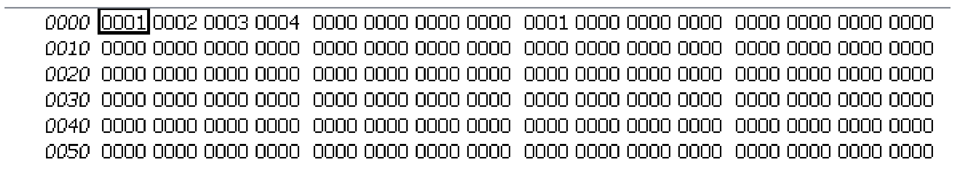
Sum: AddI R6,R6, 4

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R6as Rs and the 5 bits signed extend immediate value which is 4 and AddIng them then writes the result in the destination register R6.



SW R1 0(R6)

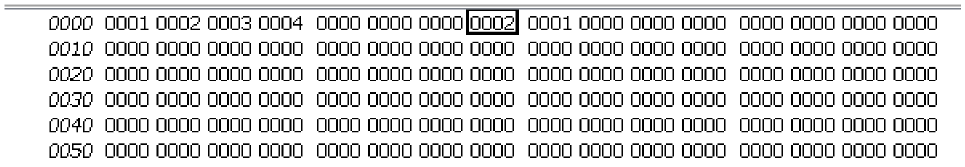
MEM[REg(Rs) + signed(Imm5)]= Reg(Rt), writing the value from Register R1 in the memory by using Displacement addressing mode.The effective memory address is the content of R6 added with the sign_extend (Imm5) which is 0.



```
0000  0001 0002 0003 0004  0000 0000 0000 0000  0001 0000 0000 0000  0000 0000 0000 0000
0010  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0020  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0030  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0040  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0050  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

SW R2 -1(R6)

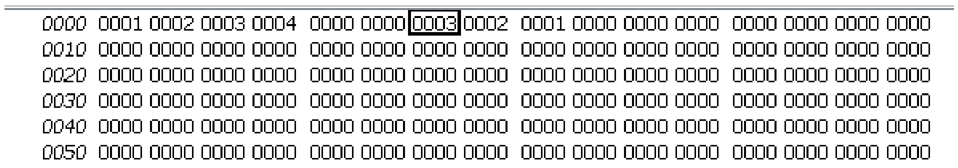
MEM[REg(Rs) + signed(Imm5)]= Reg(Rt), writing the value from Register R2 in the memory by using Displacement addressing mode.The effective memory address is the content of R6 added with the sign_extend (Imm5) which is (-1).



```
0000  0001 0002 0003 0004  0000 0000 0000 0002  0001 0000 0000 0000  0000 0000 0000 0000
0010  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0020  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0030  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0040  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0050  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

SW R3 -2(R6)

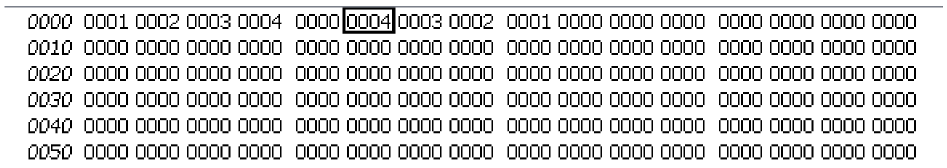
MEM[REg(Rs) + signed(Imm5)]= Reg(Rt), writing the value from Register R3 in the memory by using Displacement addressing mode.The effective memory address is the content of R6 added with the sign_extend (Imm5) which is (-2).



```
0000  0001 0002 0003 0004  0000 0000 0003 0002  0001 0000 0000 0000  0000 0000 0000 0000
0010  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0020  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0030  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0040  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0050  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

SW R4 -3(R6)

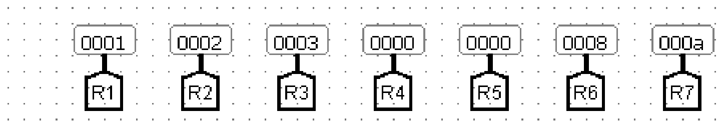
MEM[REg(Rs) + signed(Imm5)]= Reg(Rt), writing the value from Register R4 in the memory by using Displacement addressing mode.The effective memory address is the content of R6 added with the sign_extend (Imm5) which is (-3).



```
0000  0001 0002 0003 0004  0000 0004 0003 0002  0001 0000 0000 0000  0000 0000 0000 0000
0010  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0020  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0030  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0040  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0050  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

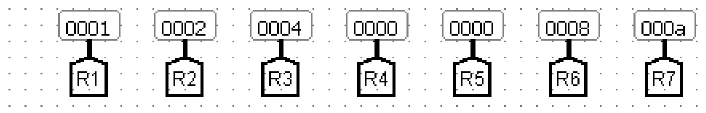
AddI R4 R0 0

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R0 as Rs and the 5 bits signed extend immediate value which is 0 and AddIng them then writes the result in the destination register R4.



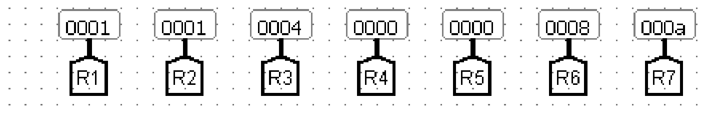
AddI R3 R0 4

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R0 as Rs and the 5 bits signed extend immediate value which is 4 and AddIng them then writes the result in the destination register R3.



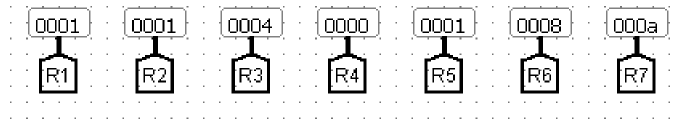
Loop: LW R2 -1(R1)

Reg(Rt) = MEM[REg(Rs) + signed(Imm5)], writing the value from the memory in the register R2 as Rt by using Displacement addressing mode.The effective memory address is Reg(R1) AddIng with the sign-extend (Imm5) which is (-1).



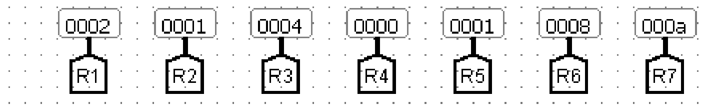
Add R5 R2 R5

Reg(Rt)= REg(Rs) + REg(rt) ,reading register R2 as Rs and the registerR5 as Rt and AddIng them then writes the result in the destination register R5.



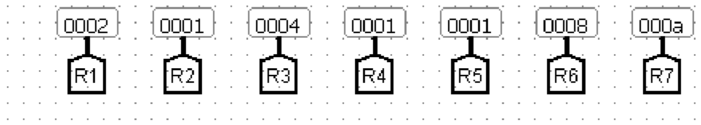
AddI R1 R1 1

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R1 as Rs and the 5 bits signed extend immediate value which is 1 and AddIng them then writes the result in the destination register R1.



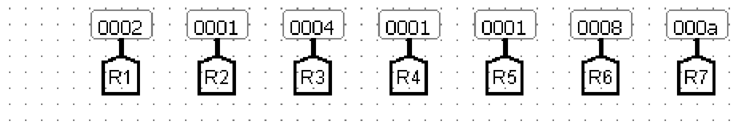
AddI R4 R4 1

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R4 as Rs and the 5 bits signed extend immediate value which is 1 and AddIng them then writes the result in the destination register R4.



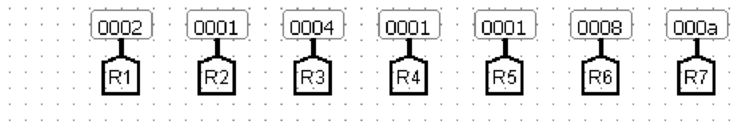
BEQ R4 R3 Exit

Branch if REg(Rs) == Reg(Rt) , reading register R4 as Rs, and register R3 as Rt and Branch if "Reg(R4) == Reg(R3)" which is wrong then it doesn't branch and goes to next instruction.



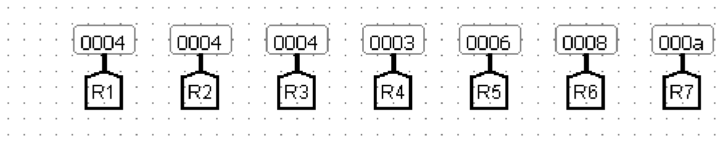
J loop

PC = PC + signed(Imm11), jumping to label **loop** (instruction pc= 18).



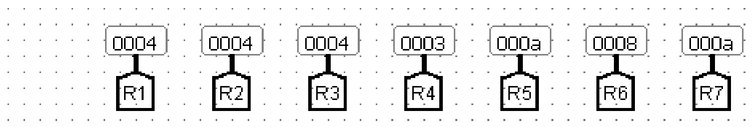
Loop: **LW R2 -1(R1)** (for last loop)

Reg(Rt) = MEM[REg(Rs) + signed(Imm5)], writing the value from the memory in the register R2 as Rt by using Displacement addressing mode.The effective memory address is Reg(R1) AddIng with the sign-extend (Imm5) which is (-1).



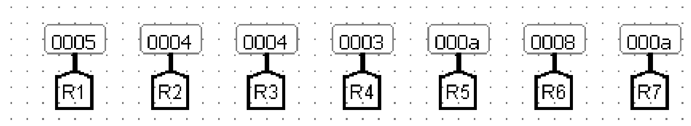
Add R5 R2 R5 (for last loop)

Reg(Rt)= REg(Rs) + REg(rt) ,reading register R2 as Rs and the registerR5 as Rt and AddIng them then writes the result in the destination register R5.



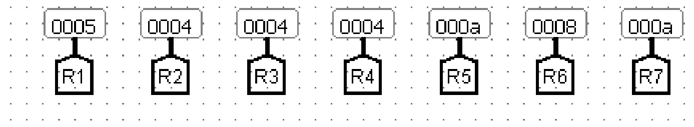
AddI R1 R1 1 (for last loop)

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R1 as Rs and the 5 bits signed extend immediate value which is 1 and AddIng them then writes the result in the destination register R1.



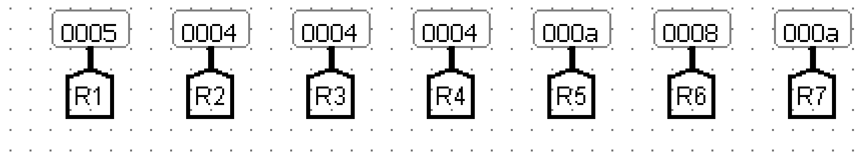
AddI R4 R4 1 (for last loop)

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R4 as Rs and the 5 bits signed extend immediate value which is 1 and AddIng them then writes the result in the destination register R4.



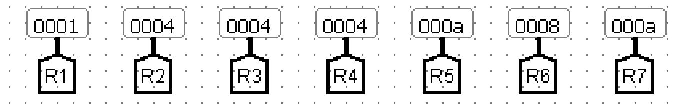
BEQ R4 R3 Exit

Branch if REg(Rs) == Reg(Rt) , reading register R4 as Rs, and register R3 as Rt and Branch if "Reg(R4) == Reg(R3)" which is true then it branch and goes to label **Exit** (instruction in pc=24) .



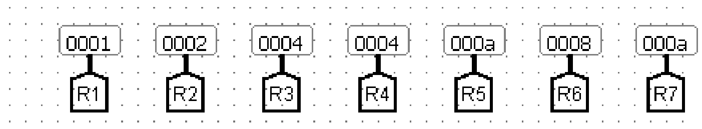
LW R1 0(R6)

Reg(Rt) = MEM[REg(Rs) + signed(Imm5)], writing the value from the memory in the register R1 as Rt by using Displacement addressing mode.The effective memory address is Reg(R6) AddIng with the sign-extend (Imm5) which is (0).



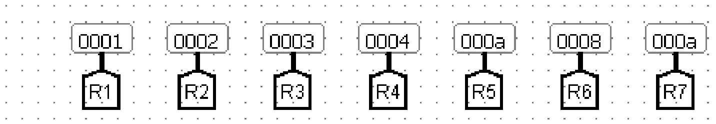
LW R2 -1(R6)

Reg(Rt) = MEM[REg(Rs) + signed(Imm5)], writing the value from the memory in the register R2 as Rt by using Displacement addressing mode.The effective memory address is Reg(R6) AddIng with the sign-extend (Imm5) which is (-1).



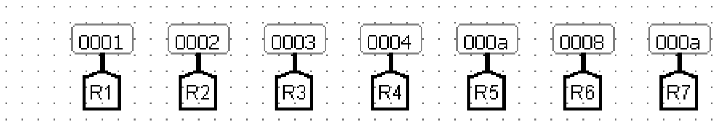
LW R3 -2(R6)

Reg(Rt) = MEM[REg(Rs) + signed(Imm5)], writing the value from the memory in the register R3 as Rt by using Displacement addressing mode.The effective memory address is Reg(R6) AddIng with the sign-extend (Imm5) which is (-2).



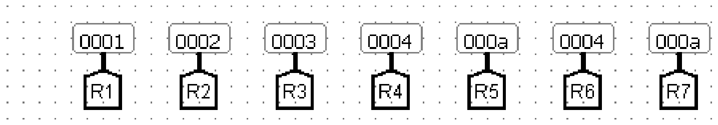
LW R4 -3(R6)

Reg(Rt) = MEM[REg(Rs) + signed(Imm5)], writing the value from the memory in the register R4 as Rt by using Displacement addressing mode.The effective memory address is Reg(R6) AddIng with the sign-extend (Imm5) which is (-3).



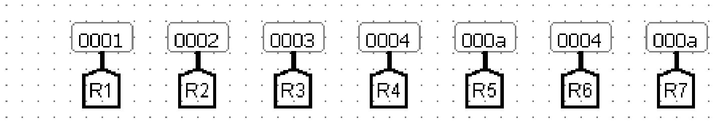
AddI R6 R6 –4

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R6 as Rs and the 5 bits signed extend immediate value which is (-4) and AddIng them then writes the result in the destination register R6.



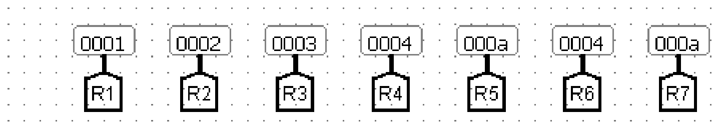
Jr R7

PC=Reg (Rs), that makes PC = Reg(R7) which jump to (instruction in pc= 0xa) .



BEQ R0 R0 0

Branch if REg(Rs) == Reg(Rt) , reading register R0 as Rs, and register R0 as Rt and Branch if "Reg(R0) == Reg(R0)" which is aLWays true terminating the program.



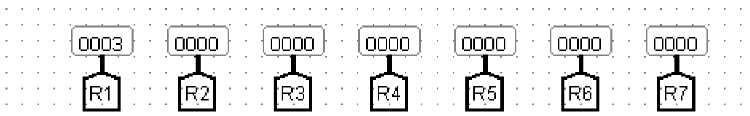
• AddItional Test Code

NO	Instruction	Hexa	Expected Value
1	ADDI R1, R0, 3	38C1	R1=0x0003
2	ADDI R2, R0, 7	39C2	R2=0x0007
3	ADDI R3, R0, 9	3A43	R3=0x0009
4	NOR R4, R1, R3	070B	R4=0xfff4
5	SLL R5, R3, 1	405D	R5=0x0012
6	SRL R5, R5, 2	48AD	R5=0x0004
7	SRA R6, R3, 3	50DE	R6=0x0001
8	BGE R1, R2, L1	888A	branch not taken
9	ADDI R1, R1, -1	3FC9	R1=0x0002
10	L1: ADDI R2, R2, -1	3FD2	R2=0x0006
11	BNE R5 , R0 , 0	7828	stop condition

AddItional Code Description

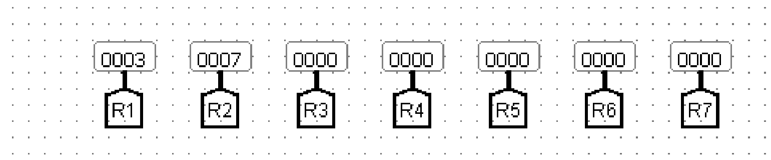
AddI R1, R0, 3

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R0 as Rs and the 5 bits signed extend immediate value which is (3) and AddIng them then writes the result in the destination register R1.



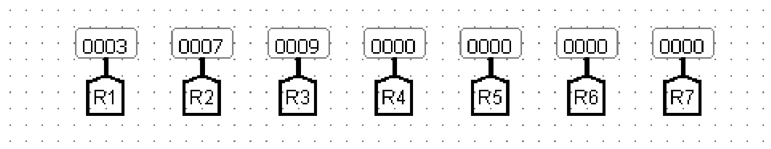
AddI R2, R0, 7

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R0 as Rs and the 5 bits signed extend immediate value which is (7) and AddIng them then writes the result in the destination register R2.

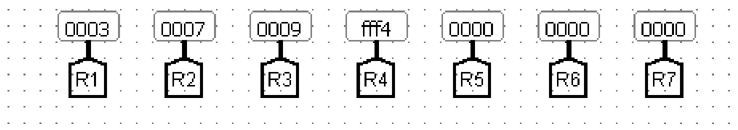


AddI R3, R0, 9

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R0 as Rs and the 5 bits signed extend immediate value which is (9) and AddIng them then writes the result in the destination register R3.

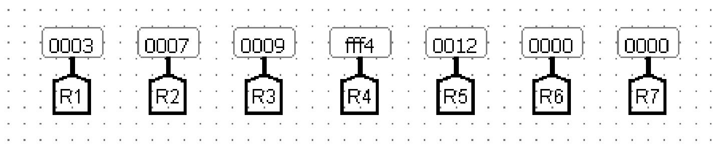


NOR R4, R1, R3



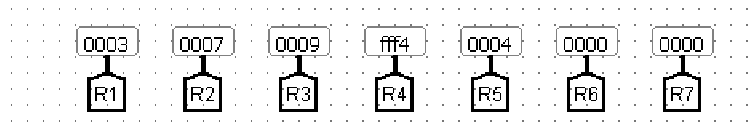
SLL R5, R3, 1

Reg(Rt) = REg(Rs) << (Imm4), reading register R3 as Rs and shifting register R3 by the signed imm = 1 as shift amount to the left by inserting 0's from the right then writes the result in the destination register R5



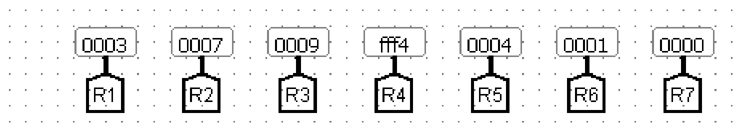
SRL R5, R5, 2

Reg(Rt) = Reg(Rs) zero>> Imm4, reading register R5 as Rs and shifting register R5 by the signed imm = 2 as shift amount to the right by inserting 0’s from the left then writes the result in the destination register R5



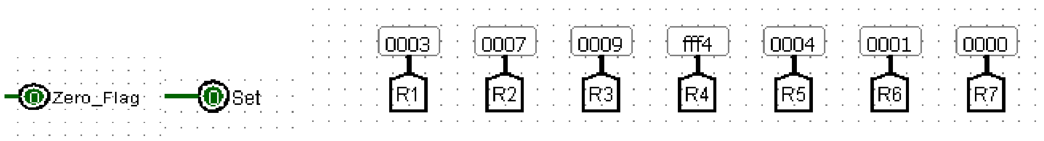
SRA R6, R3, 3

Reg(Rt) = Reg(Rs) sign>> Imm4, reading register R3 as Rs and shifting register R3 by the signed imm = 3 as shift amount to the right by inserting the last sign bit from the left then writes the result in the destination register R6



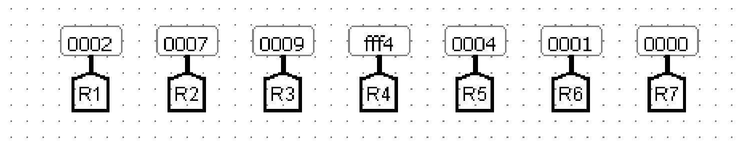
BGE R1, R2, L1

Branch if (Rs >= Rt) , reading register R2 as Rs, and register R1 as Rt and Branch if (Rs >= Rt) which is wrong then it doesn't branch and goes to next instruction.



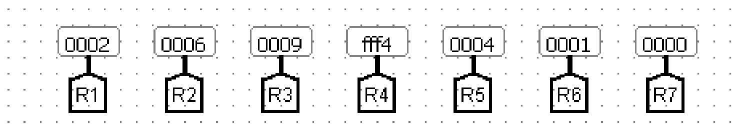
AddI R1, R1, -1

Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R1 as Rs and the 5 bits signed extend immediate value which is (1) and AddIng them then writes the result in the destination register R1.



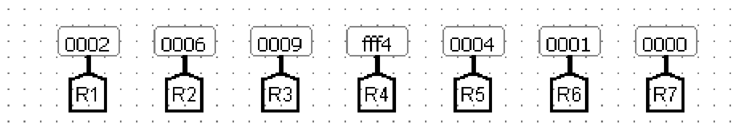
L1: AddI R2, R2, -1

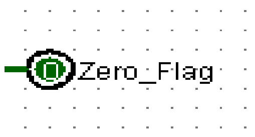
Reg(Rt)= REg(Rs) + signed (Imm5) ,reading register R2 as Rs and the 5 bits signed extend immediate value which is (-1) and AddIng them then writes the result in the destination register R2.



BNE R5, R0, 0

Branch if REg(Rs) != Reg(Rt) , reading register R0 as Rs, and register R5 as Rt and" Branch if REg(R0) != Reg(R5) " which is true terminating the program.





Part 2:
Pipelined Processor

Introduction

In the initial phase of our project, we developed a Single-Cycle Processor, meticulously analyzing its functionality and structure. However, a significant drawback of this processor was its considerable time delay. To address this limitation and optimize performance, we embarked on implementing a Pipelined processor. Pipelining is an implementation technique in which multiple instructions are overlapped in execution.

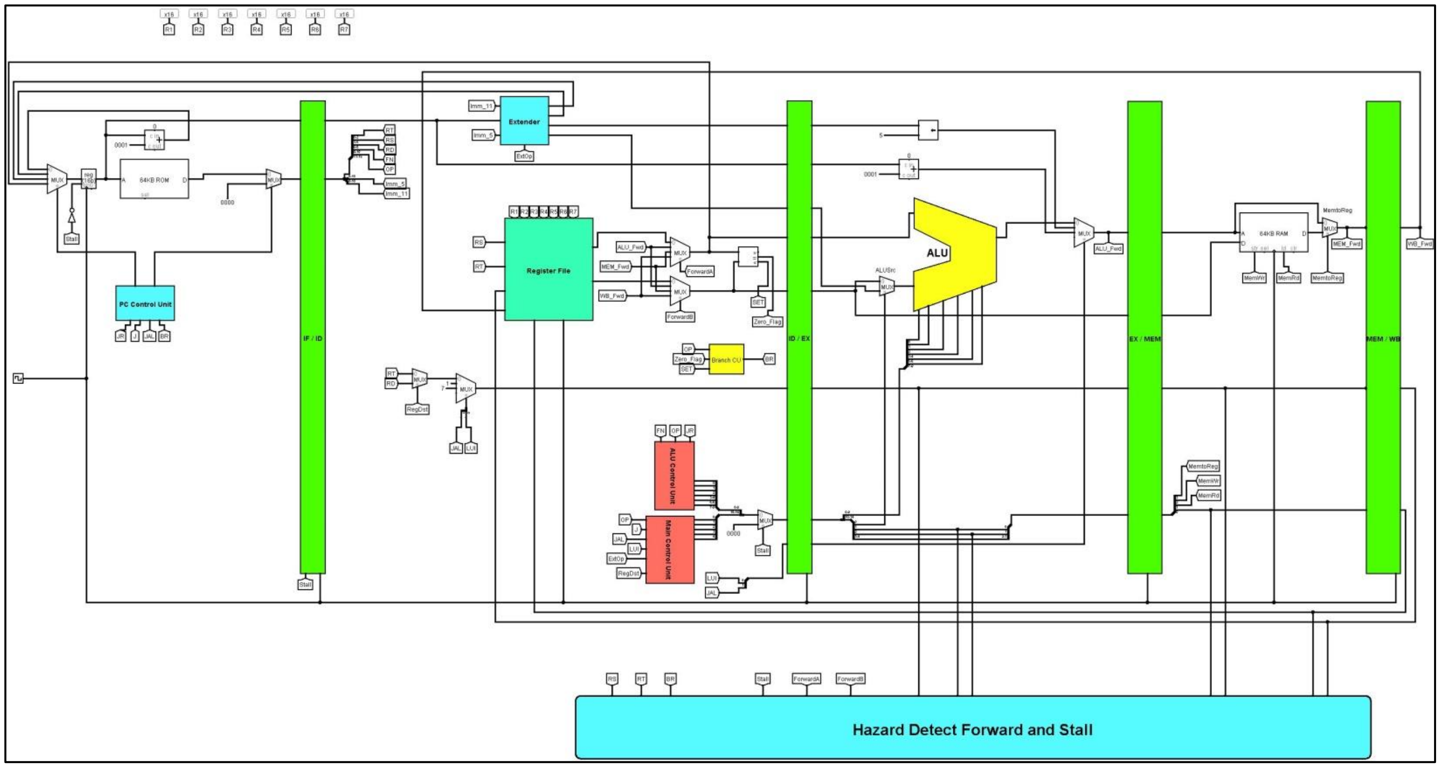
In our endeavor to transition from a Single-Cycle to a Pipelined processor several modifications were made to the existing architecture. Primarily, we introduced five distinct stages, each separated by Stage Registers, to facilitate pipelining. These stages encompassed Fetching, Decoding, Execution, Memory, and Write Back, enabling the parallel execution of multiple instructions.

However, the introduction of pipelining brought forth a new challenge: hazards, which could potentially disrupt the smooth flow of instructions through the pipeline. To mitigate these hazards, we integrated a Hazard Detect Forward and Stall Unit into our design, ensuring the timely identification and resolution of any potential conflicts or dependencies among instructions.

In the subsequent sections, we will delve into the detailed structure and functionality of each component, elucidating their roles in the Pipelined processor and highlighting the optimizations achieved through this innovative design approach.

Pipeline Stages

In the Pipelined Processor each instruction passes through 5 Stages to be executed, between every 2 stages there is a Pipeline Register and we will explain each one in details as following:



• Instruction Fetching (IF) :

In this stage, the primary objective is to retrieve instructions from the instruction memory using the address stored in the PC (Program Counter) register. Subsequently, the fetched instruction is placed into the

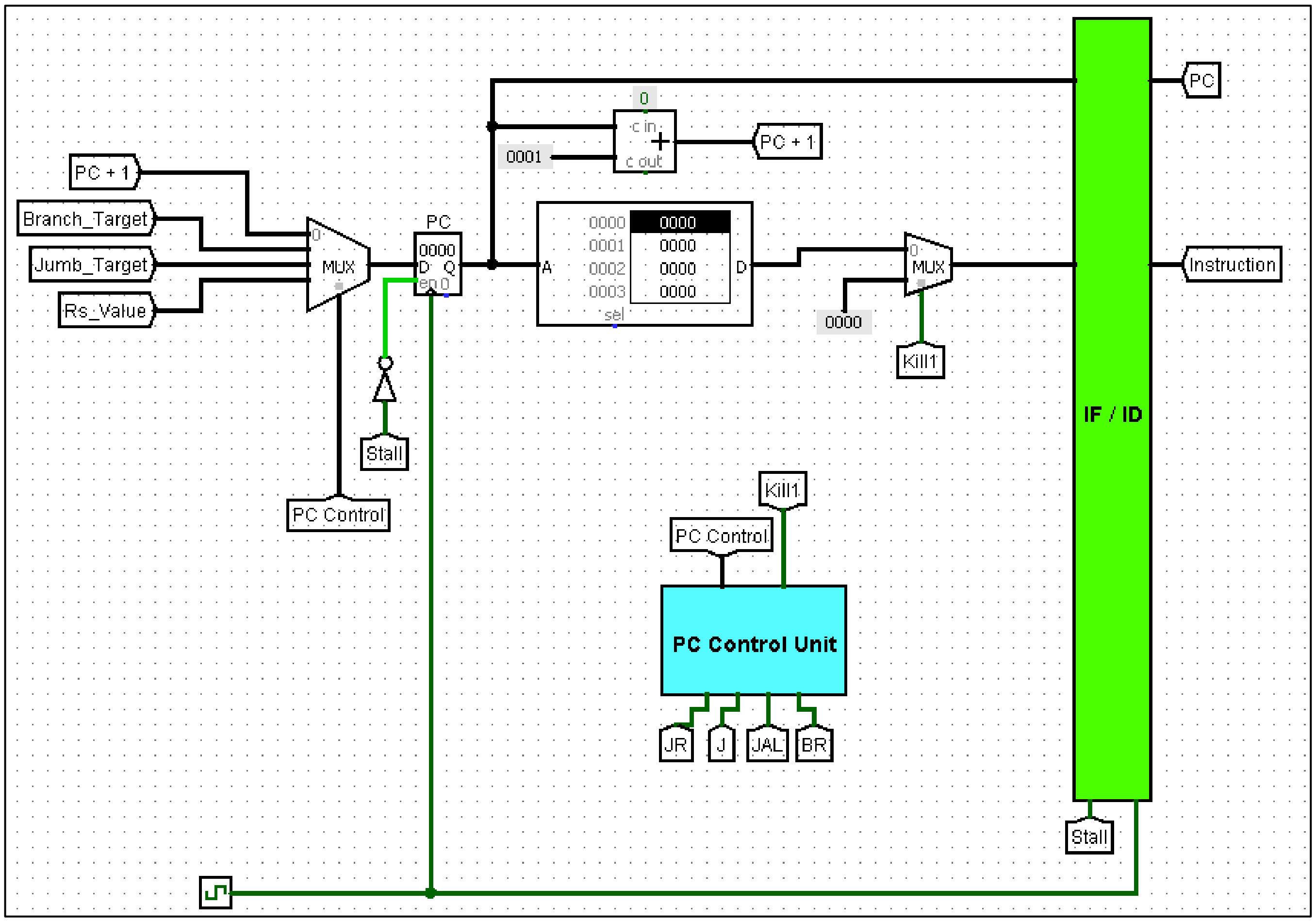
IF/ID pipeline register to initiate the next stage of the pipeline. The address stored in the PC register can vary depending on the type of instruction being executed:

1- PC +1 for the normal flow (next instruction in the memory).

2- Branch_Target for Branch instructions.

3- Jumb_Target for J and JAL instructions.

4- Rs_Value for Jr instruction.

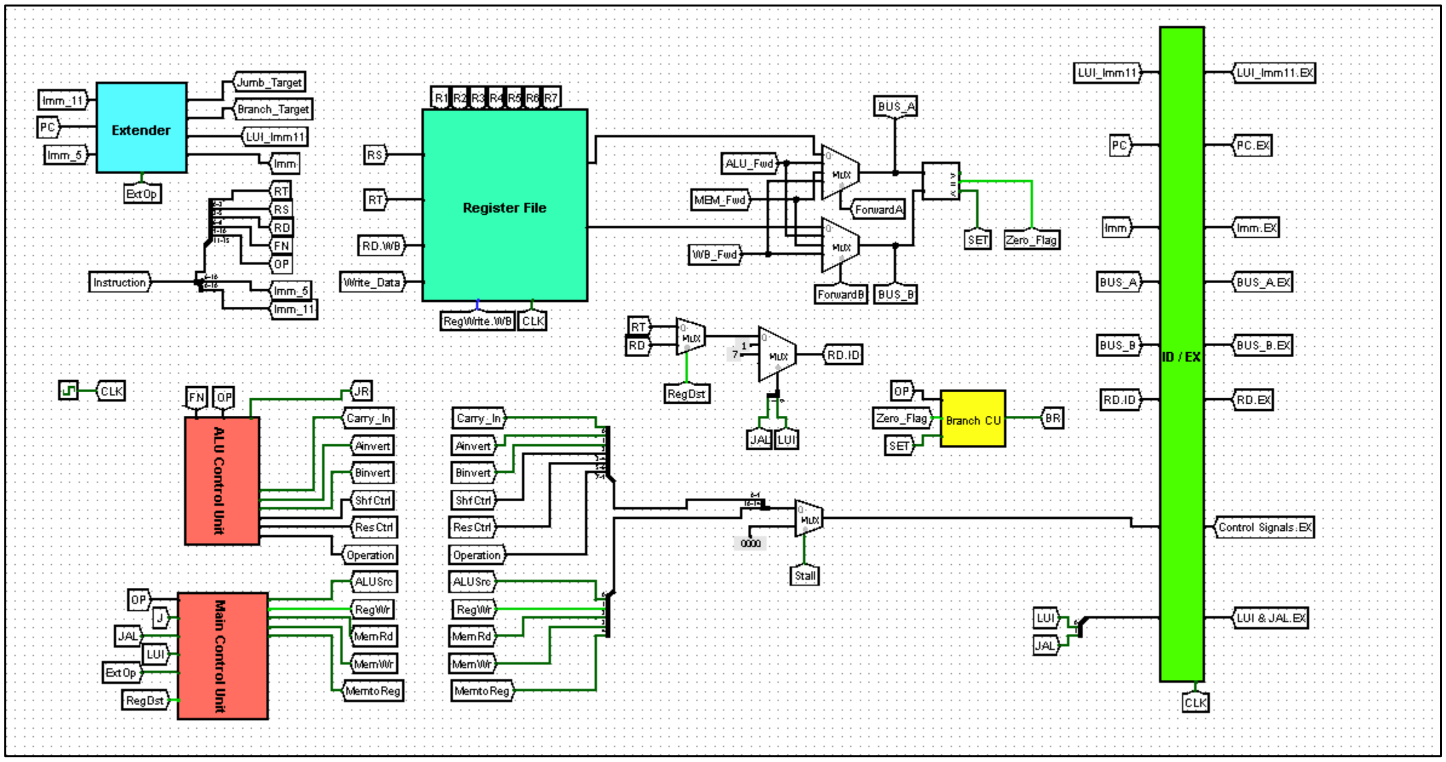


A Multiplexer selects among these addresses for the PC, with its selection controlled by the PC Control signal, which will be explained later. Additionally, this multiplexer features an enable input connected to a stall signal, allowing for the stall of PC writing in the event of hazards.

Furthermore, the PC Control Unit generates a signal known as Kill1, which temporarily stalls the instruction fetching stage for one cycle.

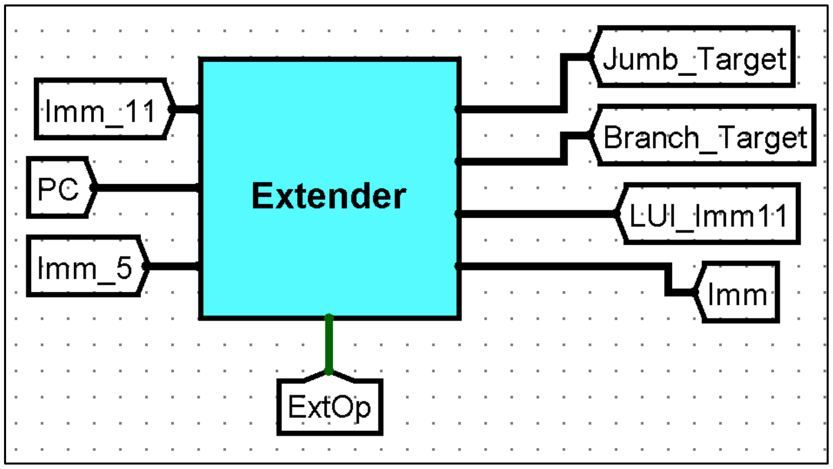
• Instruction Decoding (ID) :

In the Decoding stage of the pipeline, the primary objective is to dissect the fetched instruction into its constituent parts, including the Op code, registers, and immediate values. This disassembly process enables the preparation of the necessary values from the Register File and Extender Unit, which are subsequently placed into the ID/EX pipeline register, ready for the Execution stage.

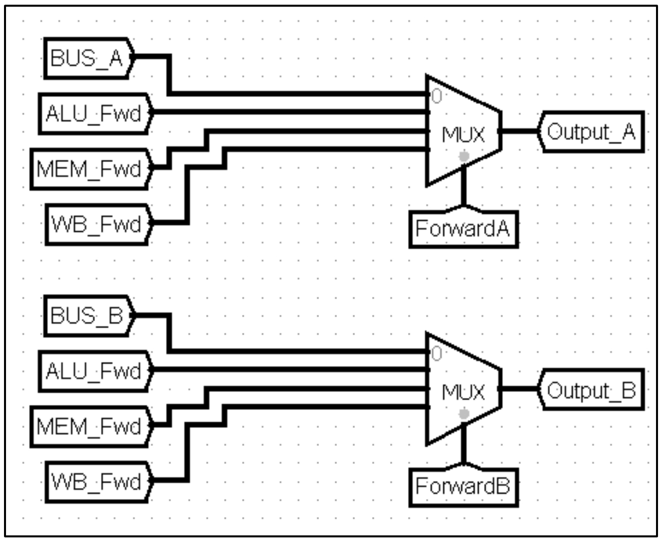


Moreover, in this stage, the Control Units generate the requisite signals for each instruction, as explained in the Single-Cycle part of our design.

To adapt the Datapath from the Single-Cycle to the Pipeline stage, three significant modifications were implemented:

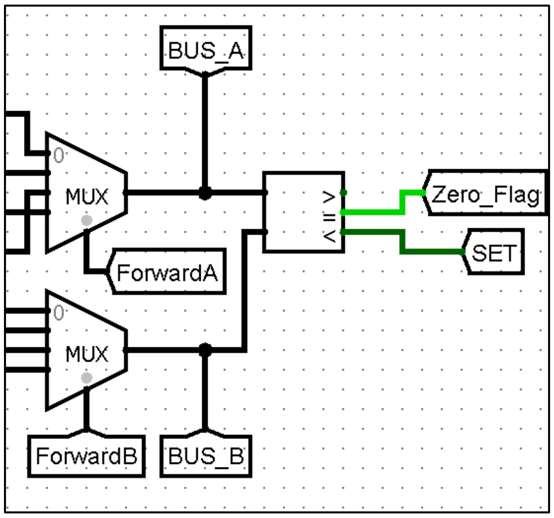


1. Extender Unit Addition: The first modification involves the incorporation of an Extender Unit, with three inputs: PC, Immediate 11, and Immediate 5, sourced from the IF/ID Pipeline Register. This unit features four outputs: Jump_Target, Branch_Target, LUI_Imm11, and Imm. The first two outputs are connected to the PC multiplexer, as discussed in the previous stage. The last two outputs are linked to ID/EX, facilitating the execution of the LUI instruction or immediate instruction in the Execution stage.



2. Forwarding Multiplexers: The second modification is that we added 2 new MUXs, one for BUS_A and one for BUS_B each MUX has 4 inputs as follow: Input 1 for Data BUS that comes from Register File for the

normal flow (in case of no hazards) the other 3 inputs are : ALU_Fwd (forwarding ALU Result), MEM_Fwd (forwarding Memory stage output) and WB_Fwd (forwarding Write Back data value), there are signals ForwardA and ForwardB which generated by Hazard Unit as we will explain later.

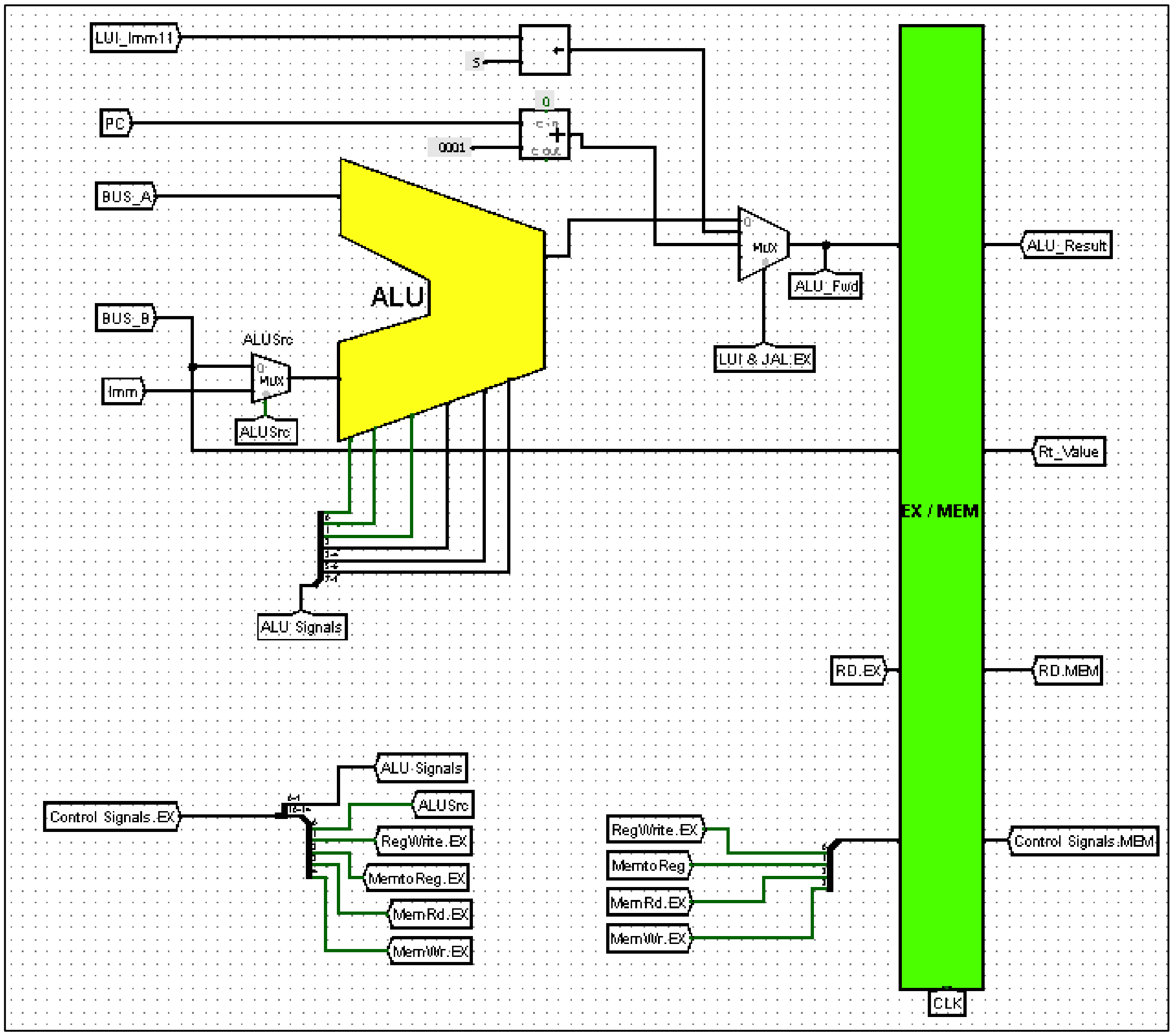


3. Comparator Integration: The third modification entails the addition of a comparator to generate the SET and Zero_Flag signals, replacing the previous method of generating these signals from the ALU. This adjustment aims to optimize clock cycles by eliminating the need for an extra cycle. The comparator's inputs are connected to the forwarding multiplexers which determine what value will pass to the comparator.

While the Control Units in this stage remain unchanged from the Single-Cycle part, signals are simply routed to the Pipeline Register ID/EX to progress to the next stage. These modifications enhance the efficiency and performance of the processor, ensuring seamless execution of instructions within the pipelined architecture.

• Execution Stage (EX) :

In the Execution stage of the pipeline, the main component is the Arithmetic Logic Unit (ALU), which performs various calculations. Similar to the Single-Cycle part, the ALU retains two inputs and produces one output, namely ALU_Result. However, to streamline the pipeline, the outputs Zero_Flag and SET have been removed, as they were already generated in the previous stage, as explained earlier.



A Multiplexer is introduced to select among three inputs:

1. ALU_Result,
2. Unsigned Immediate 11-bit value for LUI instruction (derived from the ID/EX Register),
3. PC + 1 value in the case of the JAL instruction.

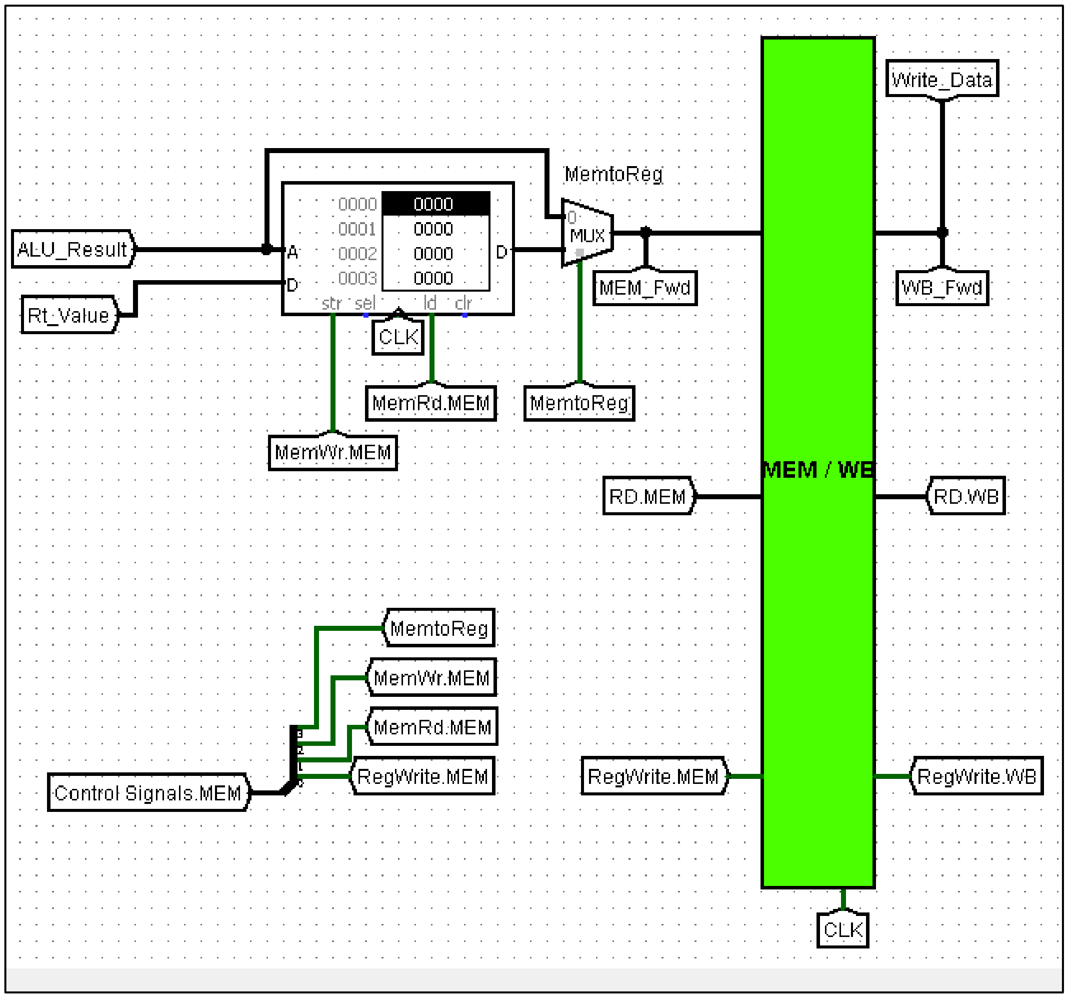
The selection among these inputs is governed by signals LUI and JAL, which are generated by the Main Control Unit. The chosen value is then connected to the EX/MEM Pipeline Register. Furthermore, it is linked to the forwarding multiplexers in the previous stage, ensuring its availability for selection in the event of a hazard.

Additionally, BUS_B, carrying the Rt Value, is connected to EX/MEM to be passed to the subsequent stage. The other two inputs of the EX/MEM Register serve to convey RD and Control Signals, ensuring the seamless transmission of information to the succeeding stage, thus maintaining the continuity of the pipeline flow.

• Memory Access (MEM)

It's the stage in which we store or load in the memory. Memory has the same inputs and outputs as Single-Cycle part, we just took the load and store control signals from the previous stage. We took forwarding from the output of the MemtoReg multiplexer to Decoding Stage due to Hazard cases.

MUX output is connected to Write Data input of MEM/WB Pipeline Register, it also has another 2 Inputs, one for RD and one for signal RegWrite.MEM that comes from previous stage.



• Write Back (WB)

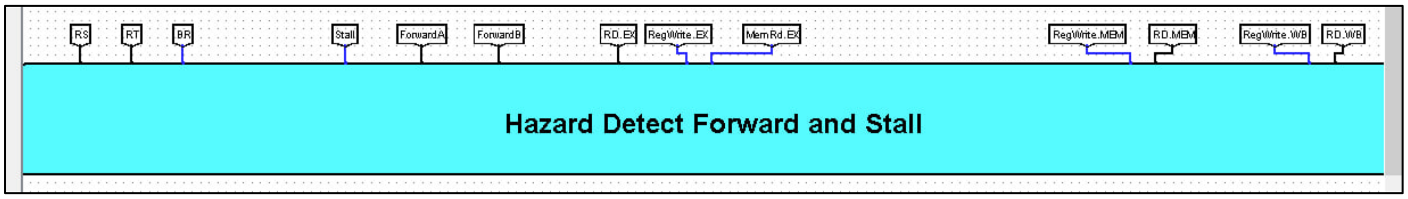
It’s the stage in which we write back in the Register File. MEM/WB Pipeline Register has 3 outputs comes from Memory Stage, First one is Write Data which write in the Register File, second one is Register Destination value which connected to Write Register input in the Register File to select which Register to write in, last output is the RegWrite.WB Signal which enable/disable writing in the Register File, it also connected to Hazard Unit as we will explain later.

Hazards and Solutions

• Hazard Detect Forward and Stall Unit

Forwarding Unit Specification: Forwarding unit is hardware solution to deal with data hazards. The idea is to pass proper values early from the pipeline. It adds special circuitry to the pipeline. This method works because it takes less time for the required values to travel through a wire than it does for a pipeline segment to compute its result. In this design, the instructions we need to forward are: all R-format instructions except JR, and ANDI, ORI, XORI, ADDI, SLL, SRL, SRA, ROR, LW from I-format instructions, and JAL and LUI from J-Format.

Forwarding unit uses to solve data hazards. Data Hazards occur when an instruction depends on the result of previous instruction and that result of instruction has not yet been computed. There are four types of data dependencies; important one of them is Read After Write (RAW).



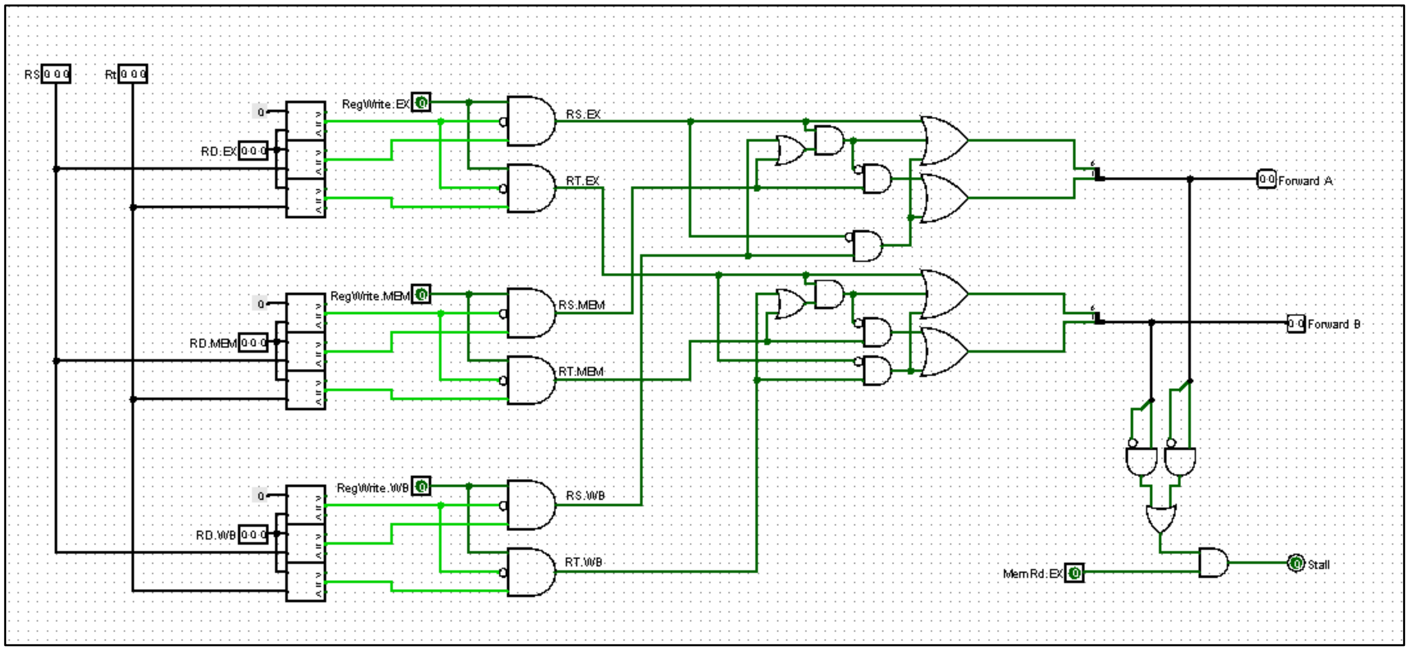
Read After Write (RAW): It is also known as True dependency or Flow dependency. It occurs when the value produced by an instruction is required by a subsequent instruction. For example,

ADD R1, --, --;

SUB --, R1, --;

Stalls or forwarding are required to handle these hazards.

Handling Data Hazards: These are various methods we use to handle hazards: Forwarding, Code reordering, and Stall insertion, in this design we focus on forwarding.



EX hazard:

If ((ID/EX.RegWrite) and (ID/EX.RegisterRd != 0) and(ID/EX.RegisterRd == IF/ID.RegisterRs)) $\rightarrow$ **ForwardA** = 01

If ((ID/EX. RegWrite) and (ID/EX. RegisterRd ! = 0) and(ID/ EX. RegisterRd == IF/ID. RegisterRt)) → **ForwardB** = 01

MEM hazard:

if ((EX/MEM. RegWrite) and (EX/MEM. RegisterRd ! = 0) and (ID/ EX. RegisterRd ! = IF/ID. RegisterRs) and (EX/MEM. RegisterRd == IF/ID. RegisterRs)) → **Forward A** = 10

If ((EX/MEM. RegWrite) and (EX/MEM. RegisterRd ! = 0) and (ID/ EX. RegisterRd ! = IF/ID. RegisterRt) and (EX/MEM. RegisterRd == IF/ID. Register Rt)) → **Forward B** = 10

WB hazard:

If ((MEM/WB. RegWrite) and (MEM/WB. RegisterRd ! = 0) and (ID/ EX. RegisterRd ! = IF/ID. RegisterRs) and (EX/MEM. RegisterRd ! = IF/ID. RegisterRs) and (MEM/WB. RegisterRd == IF/ ID. RegisterRs)) → **Forward A** = 11

If ((MEM/WB. RegWrite) and (MEM/WB. RegisterRd ! = 0) and (ID/ EX. RegisterRd ! = IF/ID. RegisterRt) and (EX/MEM. RegisterRd ! = IF/ID. RegisterRt) and (MEM/WB. RegisterRd == IF/ ID. Register Rt)) → **Forward B** = 11

Forwarding control will be in the ID stage because the forwarding multiplexors are found in that stage. The values of control signals are shown below.

<u>**Signal Explanation**</u>

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)

Unfortunately, not all data hazards can be forwarded
(Load has a delay that cannot be eliminated by forwarding)

Detecting a RAW hazard after a Load instruction:

- The load instruction will be in the EX stage.
- Instruction that depends on the load data is in the decode stage.

Condition for stalling the pipeline :

If ((EX.MemRd == 1) // Detect Load in EX stage
and (ForwardA==1 or ForwardB==1)) → Stall=1 // RAW Hazard

It insert a bubble into the EX stage after a load instruction

- Bubble is a no-op that wastes one clock cycle
- Delays the dependent instruction after load by one cycle
- Allow Load instruction in ALU stage to proceed
- Freeze PC and Instruction registers (NO instruction is fetched)
- Introduce a bubble into the ALU stage (bubble is a NO-OP), then now Load can forward data to next instruction after delaying it.

BRANCH DELAY :

Control logic detects a Branch instruction in the EX Stage, ALU computes the Branch outcome in the 3rd Stage, so Next1 and Next2 instructions will be fetched anyway, and converts Next1 and Next2 into bubbles if branch is taken.

Then if branch taken we waste two clock cycles and If branch not taken there is no wasted cycles, branch delay can be reduced from 2 cycles to just 1 cycle (like with Jump).

By adding hardware to compute the branch target address and evaluate the branch decision to the ID stage. We reduce the number of stall (flush) cycles but now need to add forwarding hardware in ID stage. Now only one instruction (Next1) that follows the branch is fetched.

Rs from EX	Rs from MEM	Rs from WB	Forward A
0	0	0	00
0	0	1	11
0	1	0	10
0	1	1	10
1	0	0	01
1	0	1	01
1	1	0	01
1	1	1	01

$$\text{Rs from EX} = (\text{Regwrite.EX}) \,.\, \overline{(\text{RD.EX} = 0)} \,.\, (\text{RD.EX} = \text{Rs})$$

$$\text{Rs from MEM} = (\text{Regwrite.MEM}) \,.\, \overline{(\text{RD.MEM} = 0)} \,.\, (\text{RD.MEM} = \text{Rs})$$

$$\text{Rs from WB} = (\text{Regwrite.WB}) \,.\, \overline{(\text{RD.WB} = 0)} \,.\, (\text{RD.WB} = \text{Rs})$$

$$\text{Forward A (bit 0)} = (\text{Rs from EX}) \mid ((\text{Rs from EX}) \text{ and } ((\text{Rs from MEM}) \mid (\text{Rs from WB}))) \mid (\overline{(\text{Rs from EX})} \text{ and } (\text{Rs from WB}))$$

$$\text{Forward A (bit 1)} = (\overline{(\text{Rs from EX})} \text{ and } \overline{((\text{Rs from MEM}) \mid (\text{Rs from WB}))}) \text{ and } (\text{Rs from MEM}) \mid (\overline{(\text{Rs from EX})} \text{ and } (\text{Rs from WB}))$$

Rt from EX	**Rt from MEM**	**Rt from WB**	**Forward B**
0	0	0	00
0	0	1	11
0	1	0	10
0	1	1	10
1	0	0	01
1	0	1	01
1	1	0	01
1	1	1	01

$$\text{Rt from EX} = (\text{Regwrite.EX}) \,.\, \overline{(\text{RD.EX} = 0)} \,.\, (\text{RD.EX} = \text{Rt})$$

$$\text{Rt from MEM} = (\text{Regwrite.MEM}) \,.\, \overline{(\text{RD.MEM} = 0)} \,.\, (\text{RD.MEM} = \text{Rt})$$

$$\text{Rt from WB} = (\text{Regwrite.WB}) \,.\, \overline{(\text{RD.WB} = 0)} \,.\, (\text{RD.WB} = \text{Rt})$$

$$\text{Forward B (bit 0)} = (\text{Rt from EX}) \mid ((\text{Rt from EX}) \text{ and } ((\text{Rt from MEM}) \mid (\text{Rt from WB}))) \mid (\overline{(\text{Rt from EX})} \text{ and } (\text{R from WB}))$$

$$\text{Forward B (bit 1)} = (\overline{(\text{Rt from EX})} \text{ and } (\overline{(\text{Rt from MEM}) \mid (\text{Rt from WB})})) \text{ and } (\text{Rs from MEM}) \mid (\overline{(\text{Rs from EX})} \text{ and } (\text{Rs from WB}))$$

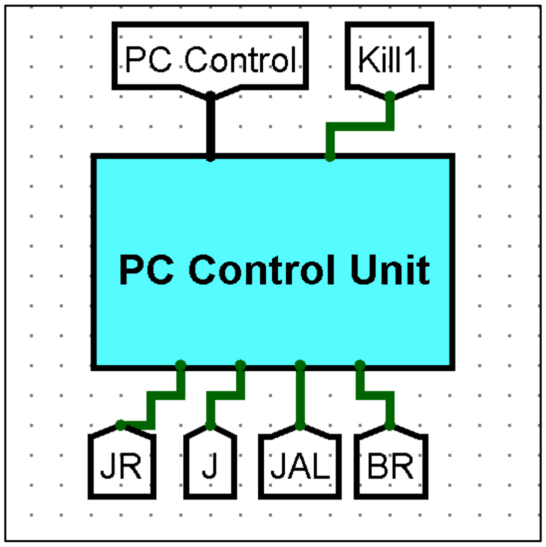
Note that in forwarding if we can forward from many stages at the same moment we use priority method, that forward from EX has more priority than forward from MEM and forward from MEM has more priority than forward from WB

Forward B (1)	Forward B (0)	Forward A (1)	Forward A (0)	MemRd.EX	Stall
1	0	1	0	1	1

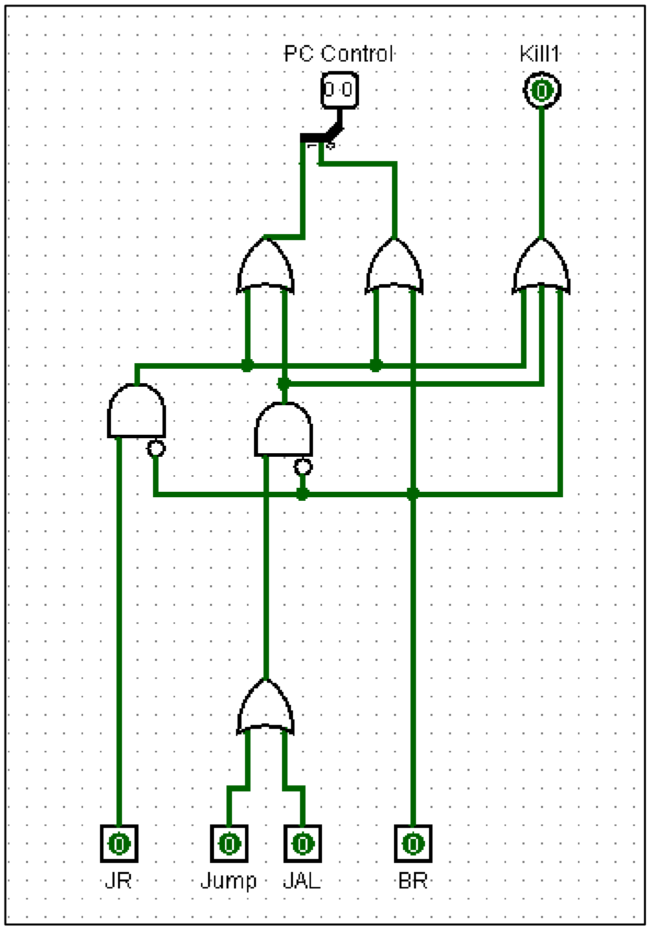
$$\text{Stall} = ((\text{Forward B (bit 1) and } (\text{Forward B (bit 1)})) \mid (((\text{Forward A (bit 1) and } (\text{Forward A (bit 1)})) \text{ and } (\text{MEMRd. EX})$$

This equation observes if there is Load instruction in EX stage.

• PC Control Unit



We added a new component PC Control Unit that was not in Single-Cycle Part to make Datapath more organized; That Unit has 4 inputs which are for 4 Instructions signals (JR, J, JAL, BR).



That Unit generate 2 two outputs control signals: the first control one is used to determine which input will be in the pc next cycle, where pc control is a 2 bit signal which select one address from to Path into next PC register (as explained in IF Stage), while the second one is used to determine if the instruction is to be decoded or killed depending on the instruction, that if the previous instruction is any of (Branch, Jump, Jump and Link or Jump Register) and taken then kill signal will be 1 as the current instruction is fetched then signal 1 will pass 0 which does no operation if not taken then kill signal is 0 passing the next instruction normally.

Instructions Truth Table

instruction	OP	PC1	PC0	Kill1
JR	2	1	1	1
BEQ	14	0	1	1
BNE	15	0	1	1
BLT	16	0	1	1
BGE	17	0	1	1
J	30	1	0	1
JAL	31	1	0	1

Signals Logic Equations

$\mathrm{PC1} = \mathrm{JR}.\mathrm{BR} + (\mathrm{Jump} + \mathrm{JAL}).\mathrm{BR}$

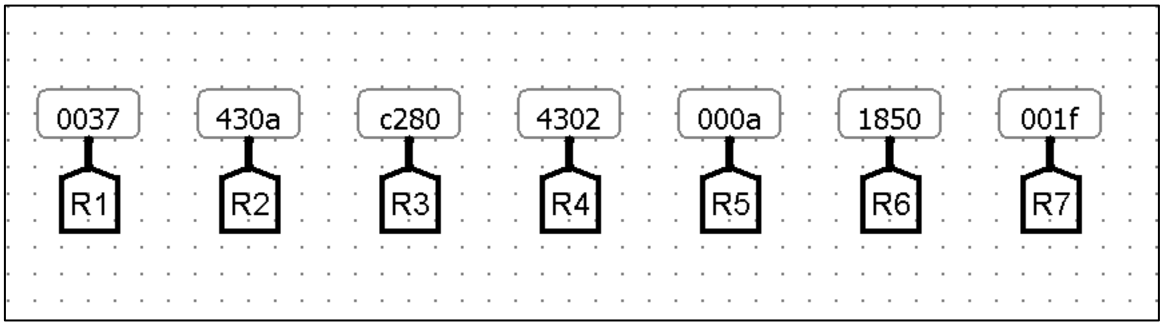
$\mathrm{PC0} = \mathrm{BR}$

$\mathrm{KILL1} = \mathrm{JR}.\overline{\mathrm{BR}} + (\mathrm{Jump} + \mathrm{JAL}).\overline{\mathrm{BR}} + \mathrm{BR}$

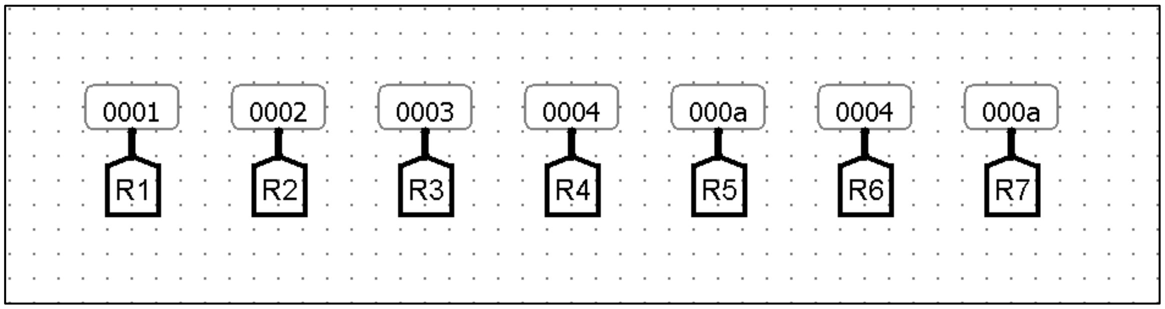
Simulation & Testing

After applying the hardware solutions for Pipeline Hazards that we have explained in the previous part, we tested the Processor by the same 3 test codes from Single-Cycle part and we got the same final registers outputs as follow:

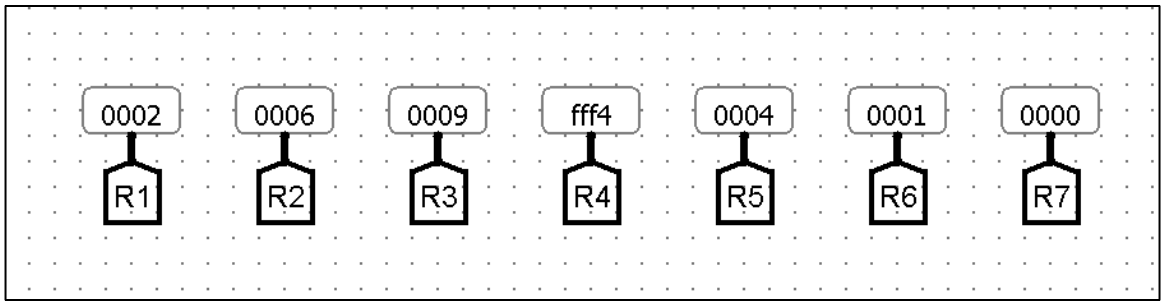
For First Test Code :



For Array Test Code :



For Additional Test Code :

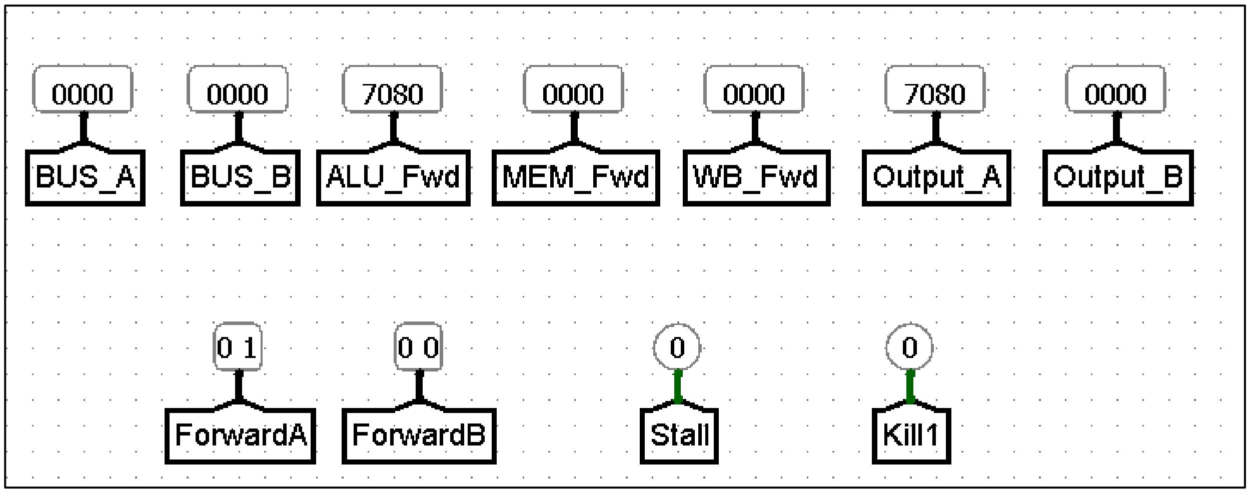


But we also wanted to ensure that Hazard Unit is working correctly and generate required forward and stall signals so we took some examples from the Test Codes to ensure that, and showed each input and output values for forwarding MUXs (explained in ID Stage page 76) as follow:

Example 1 : **LUI 900** **(from first Test Code)**

AddI R5, R1,13

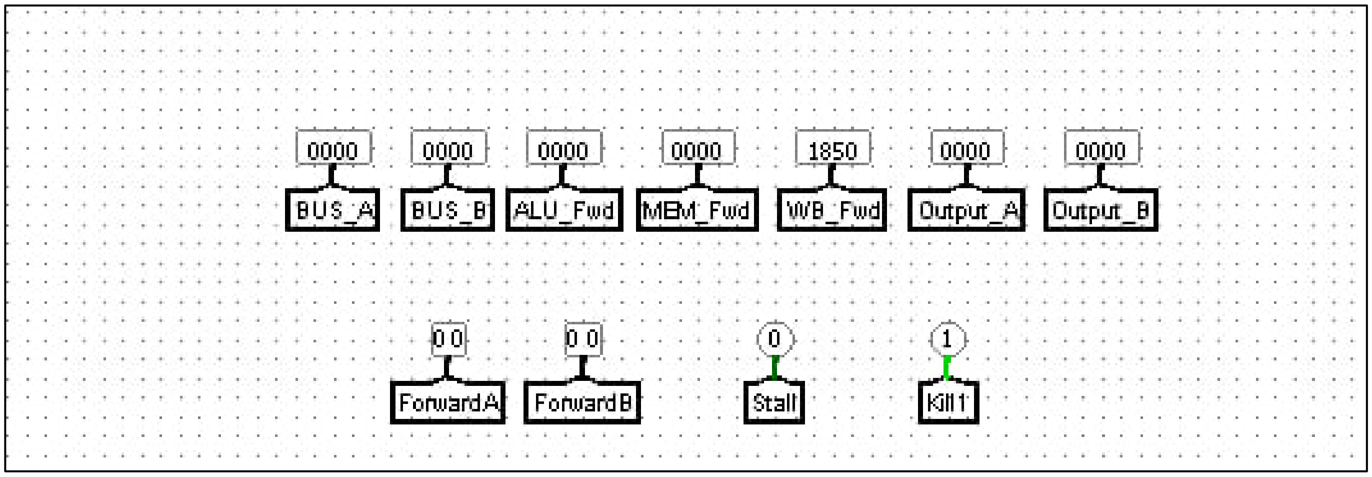
In that example AddI instruction needs the R1 value from the LUI instruction while it is at ID stage so the ALU_Fwd path will forward R1 correct value to ID Stage and signal ForwardA will be 01 as shown below:



Example 2 : **BEQ R0,R0,-1** **(from first Test Code)**

Func: OR R5, R2, R3

In that example branch instruction when decoded needs to kill the fetched instruction as the branche is taken and it keeps repeating as a stop condition

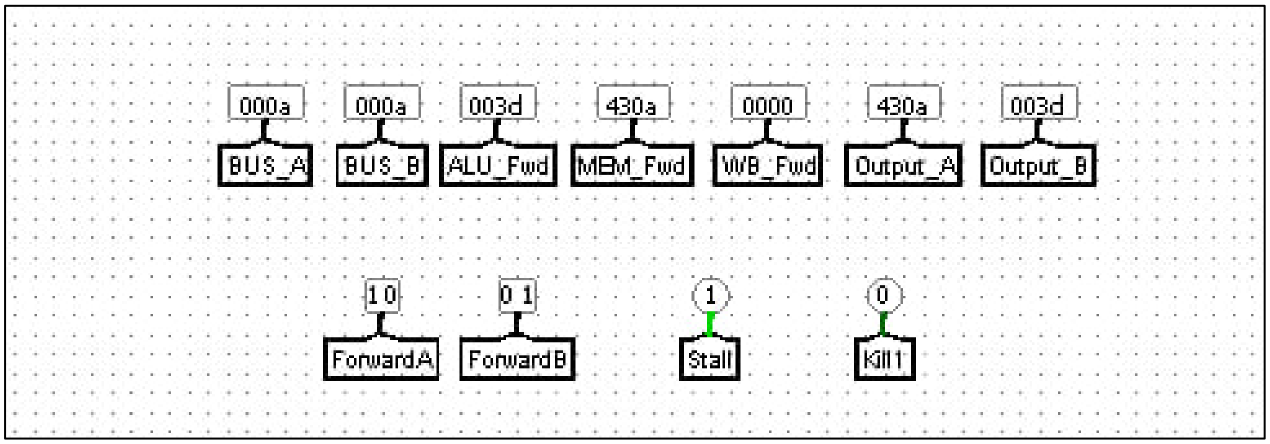


Example 3 : **LW R2, 5(R1) (from first Test Code)**

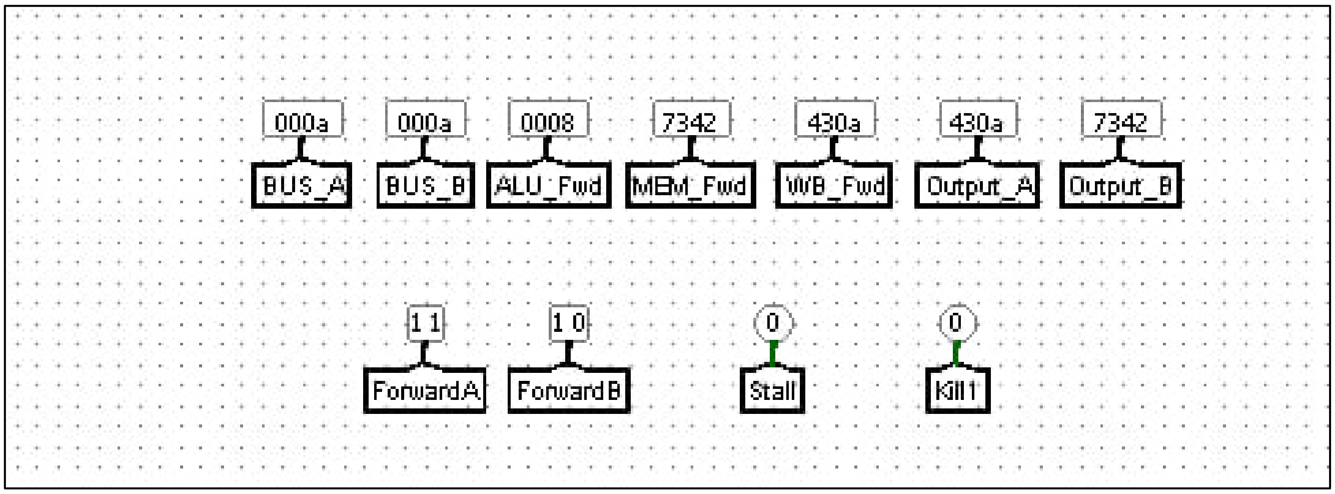
LW R3 ,6(R1)

And R4, R2, R3

In this instruction there is read after write hazard in R2,R3 that when correct value of R2 is in memory the value is forwarded directly but there is aload delay due to presence of the correct value of R3 in Ex stage ;stall is activated keeping this and instruction in decoding stage and send a bubble to the Ex stage in the next cycle



the value of R2 is forwarded from WB stage , and the value of R3 is forwarded from MEM stage



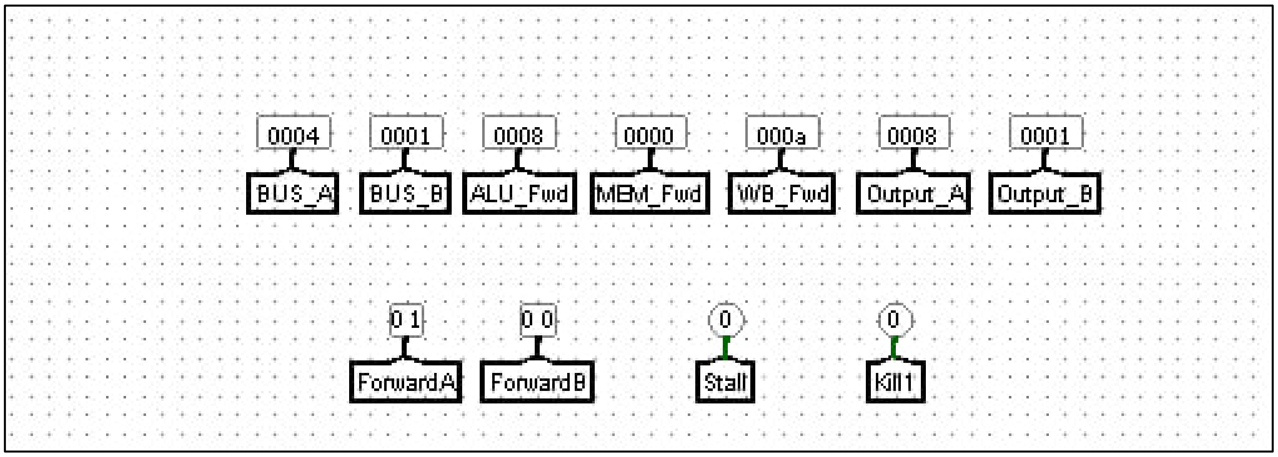
Example 4 : **SW R1 0(R6) (from second Test Code)**

SW R2 -1(R6)

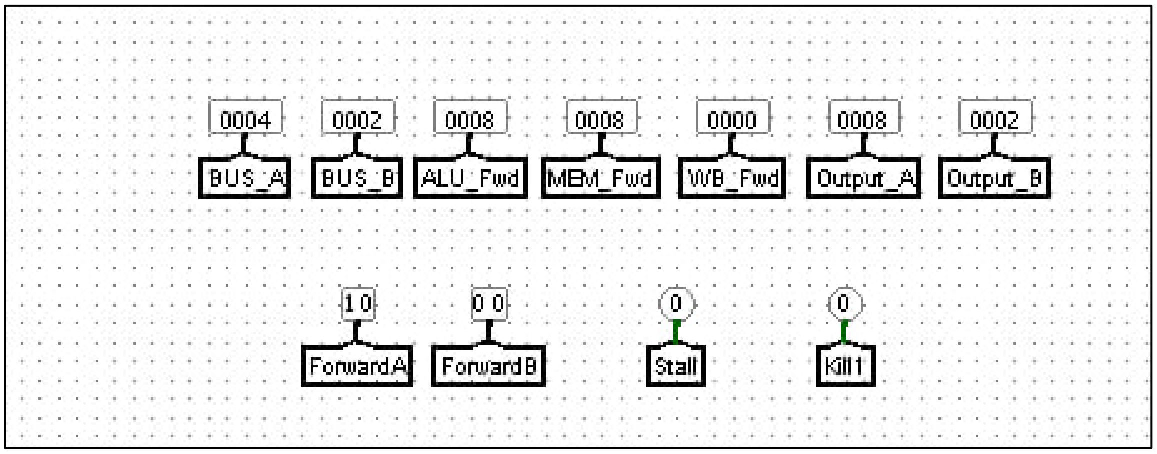
SW R3 -2(R6)

SW R4 -3(R6)

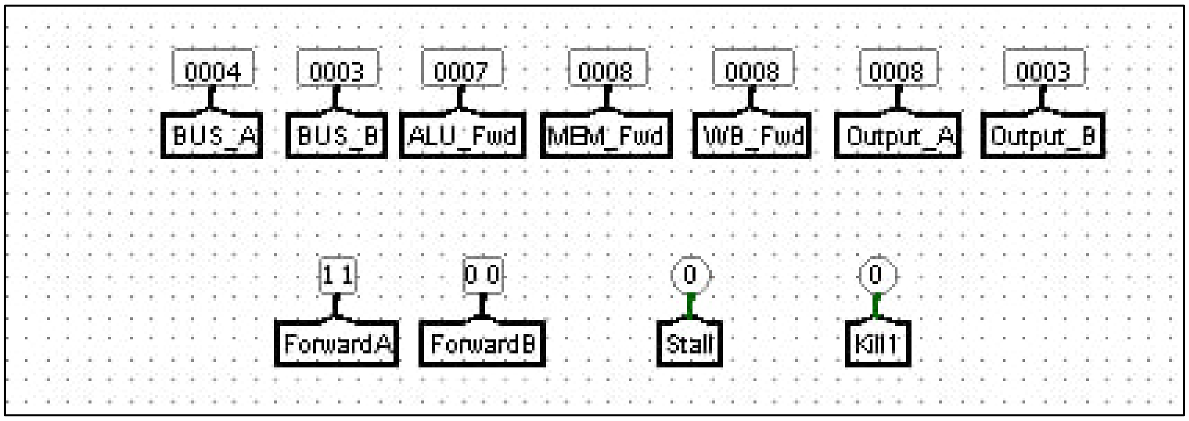
SW R1 0(R6) in this instruction the correct value of R6 is in Ex stage of instruction AddI R6,R6,4 that the value of R6 is forwarded by signal forward A =01



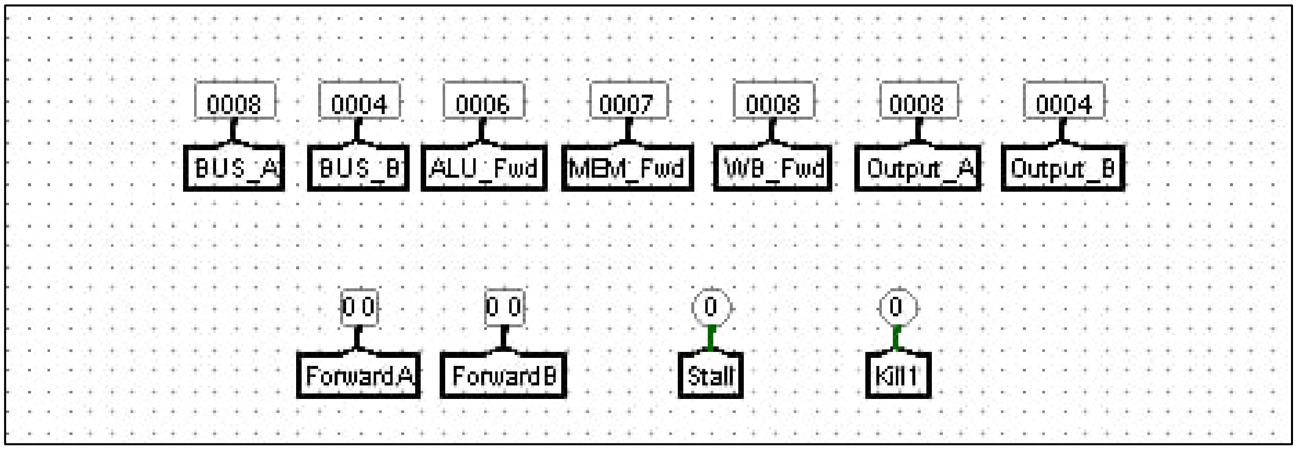
SW R2 -1(R6) in this instruction the correct value of R6 is in Ex stage and mem stage of instruction AddI R6,R6,4 that the value of R6 is forwarded by signal forward A =10 as we only forward from the previous clock cycle



SW R3 -2(R6) in this instruction the correct value of R6 is in Ex stage an, mem stage and WB stage of instruction AddI R6,R6,4 that the value of R6 is forwarded by signal forward A =11 as we only forward from the previous clock cycle



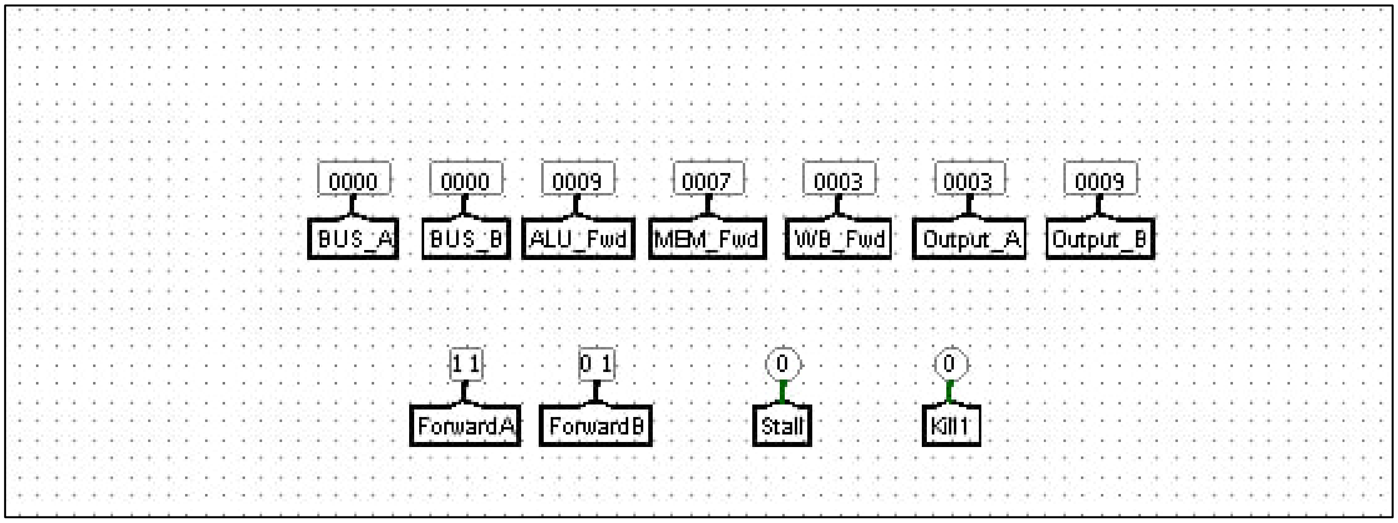
SW R4 -3(R6) in this instruction the correct value of R6 of instruction AddI R6,R6,4 exists in the register file and there is no need for forwarding signals



Example 5 : **NOR R4, R1, R3 (from third Test Code)**

SLL R5, R3, 1

NOR R4, R1, R3 in this example nor instruction needs R1 value from addi instruction and R3 value from addi instruction when it is at ID stage so mem_fwd will forward R1 correct value to ID stage and signal Forward A will be 11 also ALU_Fwd will forward R3 correct value to ID Stage and signal Forward B will be 01 as shown below:



Team Work

Mustafa Mohammed :

Design ALU, Hazard Unit and Big Part of Datapath.

Leads team, gives ideas and problems solver.

Made Array & Additional test codes.

Participate in Report.

Ahmed Ibrahim :

Design Control Units with Moaz Adel.

Designed part of Datapath.

Simulated test codes with Mustafa.

Made Simulation Part in Report.

Moaz Adel :

Designed Register File, Memories and PC Structure and Datapath.

Design Control Units with Ahmed.

Design Part of Datapath and organized all of it.

Write most of Report and format all of it.

-We have about more than 20 meetings most of them are online.

-Total time of working for more than 300 Hours.