RV32IM Pipeline Implementation

GitHub Repository
Project Page

Damsy De Silva (E/16/069)
Shirly Ekanayaka (E/16/094)
Buddhi Perera (E/16/276)

Table of Contents

1. Pipeline Diagram

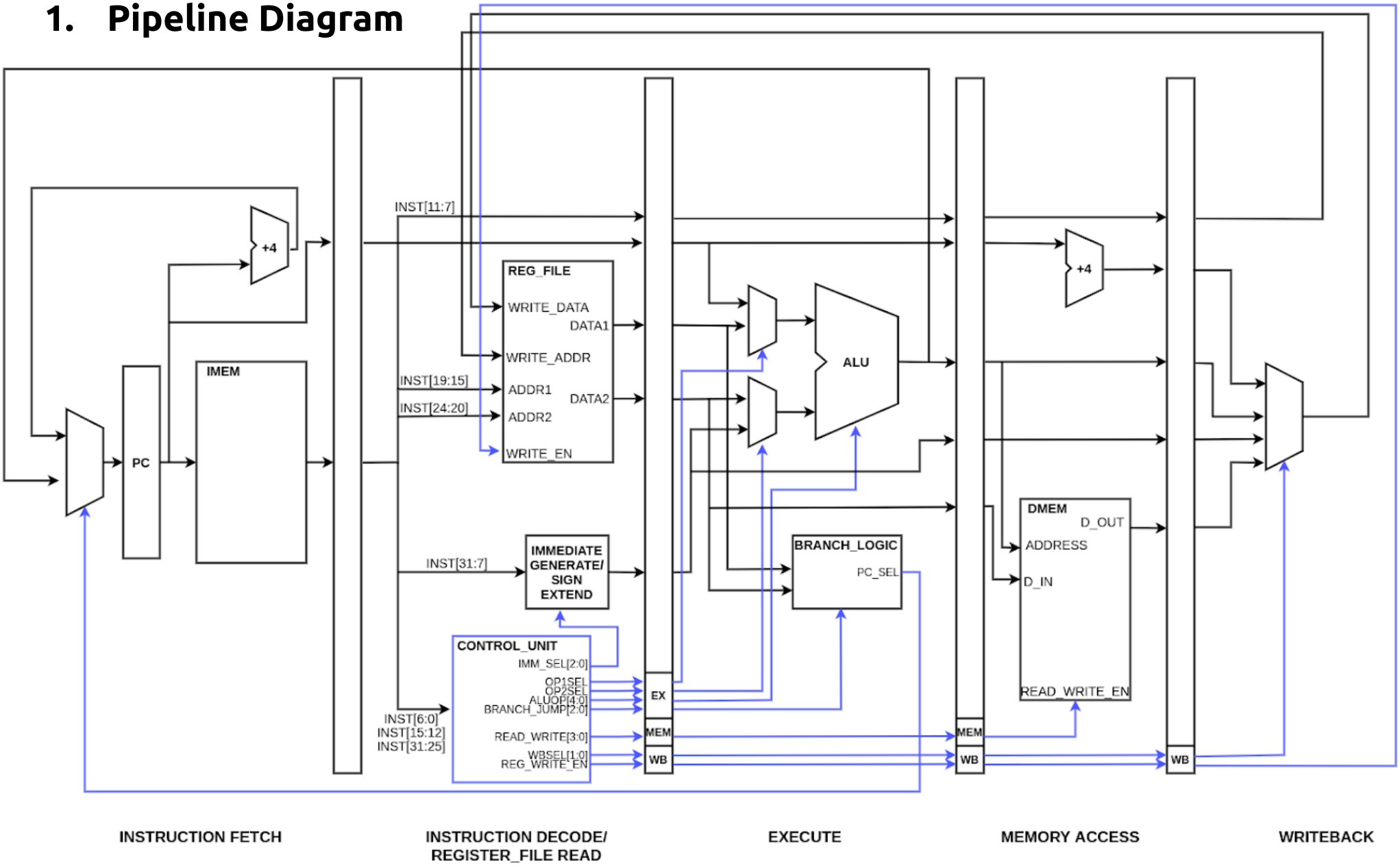


FIGURE 1 : Pipeline Diagram

2. Hardware Units

2.1. Control Unit

The control unit generates the necessary control signals to select the proper data path for the instruction.

Inputs to the control unit are,

- OPCODE[6:0]
- FUNCT3[2:0]
- FUNCT7[6:0]

Outputs generated from the control unit are,

- IMM_SEL[2:0]
- OP1_SEL
- OP2_SEL
- ALU_OP[4:0]
- BRANCH_JUMP[2:0]
- READ_WRITE[3:0]
- WB_SEL[1:0]
- REG_WRITE_EN

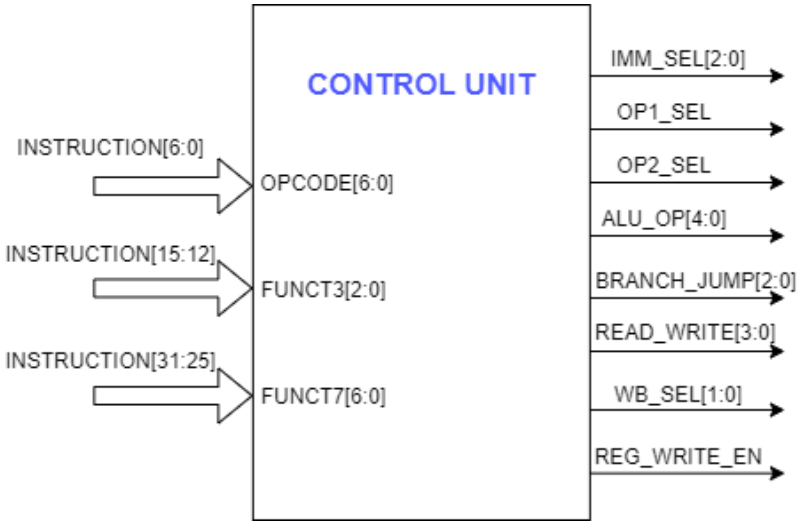


FIGURE 2 : Control Unit

2.1.1. Control Signals Generated by the Control Unit

2.1.1.1. IMM_SEL

This control signal is for the immediate value generation unit. In RISC-V ISA, according to the ordering of the immediate value bits there are 7 variants (See Immediate value generation unit for more details).

- U - Type
- J - Type
- S - Type
- B - Type
- I - Type signed
- I - Type containing shift amount
- I - Type unsigned

The immediate value generation unit will generate these 7 types of immediate values and the IMM_SEL control signal will select the relevant immediate value. The IMM_SEL control signal is a 3-bit signal and the encoding of the bits are as shown in Table 1.

Immediate Type	IMM_SEL[2]	IMM_SEL[1]	IMM_SEL[0]
U	0	0	0
J	0	0	1
S	0	1	0
B	0	1	1
I_signed	1	0	0
I_shift	1	0	1
I_unsigned	1	1	1

TABLE 1 : IMM_SEL Encodings

2.1.1.2. OP1_SEL

The input operand 1 of the ALU unit is of 2 values.

- PC value - For AUIPC, JAL, B - Type instructions
- DATA1 (value from the register file) - For all the other remaining instructions

This control signal will select between these two values. This is a 1 - bit control signal and the encoding is shown in Table 2.

Operand 1 Value	OP1_SEL
DATA1	0
PC	1

TABLE 2 : OP1_SEL Encoding

2.1.1.3. OP2_SEL

The input operand 2 of the ALU unit is of 2 values.

- DATA2 (value from the register file) - For R - Type instructions
- Immediate value - For all the other remaining instructions

This control signal will select between these two values. This is a 1 - bit control signal and the encoding is shown in Table 3.

Operand 2 Value	OP2_SEL
DATA2	0
Immediate Value	1

TABLE 3 : OP2_SEL Encoding

2.1.1.4. ALU_OP

This signal will select the relevant ALU operation out of the 18 ALU operations . This is a 5 - bit control signal and the encoding is shown in Table 4.

ALU operation	ALU_OP[4]	ALU_OP[3]	ALU_OP[2]	ALU_OP[1]	ALU_OP[0]
ADD	0	0	0	0	0
SUB	0	0	0	1	0
SLL	0	0	1	0	0
SLT	0	1	0	0	0
SLTU	0	1	1	0	0
XOR	1	0	0	0	0
SRL	1	0	1	0	0
SRA	1	0	1	1	0
OR	1	1	0	0	0
AND	1	1	1	0	0
MUL	0	0	0	0	1
MULH	0	0	1	0	1
MULHU	0	1	0	0	1
MULHSU	0	1	1	0	1
DIV	1	0	0	0	1
DIVU	1	0	1	0	1
REM	1	1	0	0	1
REMU	1	1	1	0	1

TABLE 4 : ALU_OP Encoding

2.1.1.5. BRANCH_JUMP

This control signal will select the type of branching to be considered by the Branching and Jump detection unit. Instructions in RV32IM can be categorized into 8 categories depending on their branching (See Branching and Jump Detection Unit for more details).

- BEQ - For BEQ instruction
- BNE - For BNE instruction
- J - For J - Type instruction
- BLT - For BLT instruction
- BGE - For BGE instruction
- BLTU - For BLTU instruction
- BGEU - For BGEU instruction
- NO - For all other remaining instructions

BRANCH_JUMP control signal is a 3 - bit control signal and encoding is shown in Table 5.

Branch Type	**BRANCH_JUMP [2]**	**BRANCH_JUMP [1]**	**BRANCH_JUMP [0]**
BEQ	0	0	0
BNE	0	0	1
NO	0	1	0
J	0	1	1
BLT	1	0	0
BGE	1	0	1
BLTU	1	1	0
BGEU	1	1	1

TABLE 5 : BRANCH_JUMP Encoding

2.1.1.6. READ_WRITE

In RISC-V ISA, there are 5 types of load instructions and 3 types of store instructions depending on the number of bits loaded/stored. This control signal is sent to the data cache memory and the data cache memory will load/store according to the READ_WRITE signal. Types of load/store instructions are,

- LB - Load byte (8 bits from given address)
- LH - Load halfword (16 bits from given address)
- LW - Load word (32 bits from given address)
- LBU - Load byte unsigned(8 bits from given address)
- LHU - Load halfword unsigned(16 bits from given address)
- SB - Store byte (least significant 8 bits from input data)
- SH - Store halfword (least significant 16 bits from input data)
- SW - Store word (32 bits from input data)

READ_WRITE control signal is a 4 bit control signal and the encoding is shown in Table 6.

Load/Store type	**READ_WRITE[3]**	**READ_WRITE[2]**	**READ_WRITE[1]**	**READ_WRITE[0]**
No load/store	0	0	0	0
LB	1	0	0	0
LH	1	0	0	1
LW	1	0	1	0
LBU	1	1	0	0
LHU	1	1	0	1
SB	1	0	1	1
SH	1	1	1	0
SW	1	1	1	1

TABLE 6 : READ_WRITE Encoding

2.1.1.7. WB_SEL

There are 4 sources for the write back value to be written to the register file.

- ALU result - For AUIPC, I - Type and R - Type
- Data from the data memory - For Load instructions
- Immediate value - For LUI instruction
- PC + 4 value - For J - Type instruction

This control signal will select between these 4 sources. The WB_SEL signal is a 2 - bit control signal and the encoding is shown in Table 6.

Writeback Source	**WB_SEL[1]**	**WB_SEL[0]**
ALU result	0	0
Data from data memory	0	1
Immediate value	1	0
PC + 4	1	1

TABLE 7 : WB_SEL Encoding

2.1.1.8. REG_WRITE_EN

This control signal will enable writing to the register file. When REG_WRITE_EN is set, the write back value is written to the register file and when REG_WRITE_EN is cleared, the write back value is not written to the register file.

2.1.2. Control Unit Design and Design Decisions

2.1.2.1. Control Signals Generated from the OPCODE

Control unit was implemented using combinational logic. Some control signals were generated by using the OPCODE bits in the instruction and for some control signals an intermediate signal was generated using the OPCODE bits and then the intermediate signal, FUNCT3 bits and FUNCT7 bits of the instruction were used to generate the control signals.

Control signals generated using the OPCODE,

- OP1_SEL
- OP2_SEL
- REG_WRITE_EN
- WB_SEL[1:0]

Control signals generated using the OPCODE, FUNCT3 and FUNCT7,

- IMM_SEL[2:0]
- ALU_OP[4:0]
- BRANCH_JUMP[2:0]
- READ_WRITE[3:0]

Figure 3 shows the combinational logic circuit designed to generate the control signals and the intermediate signals using the OPCODE bits. ALUOP_TYPE, BL and IMM_TYPE are the intermediate signals generated using the OPCODE bits which will be later used by separate combinational logic circuits to generate the ALU_OP, BRANCH_JUMP and IMM_SEL control signals respectively.

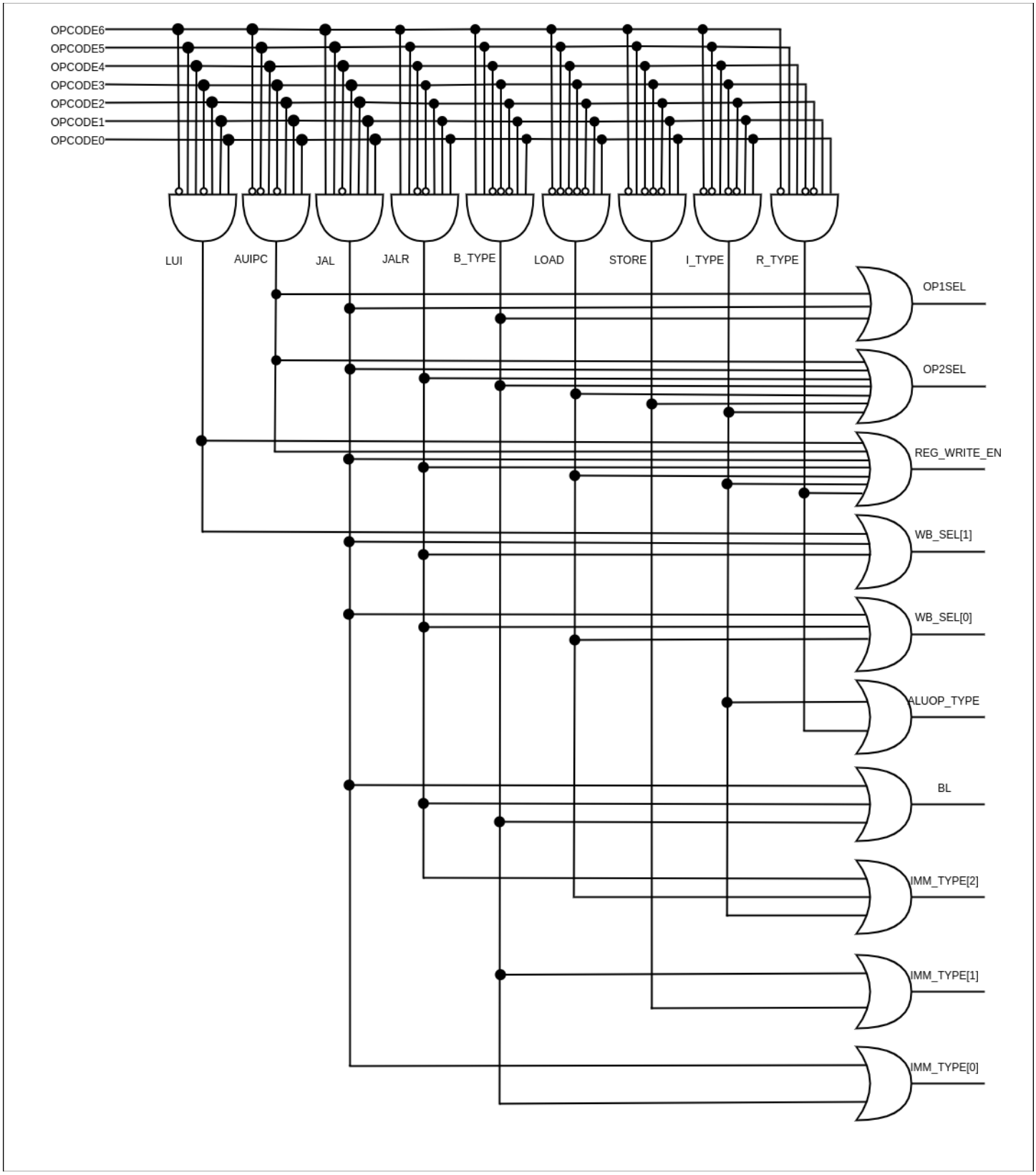


FIGURE 3 : Combinational circuit generating the control signals from OPCODE

2.1.2.2. Generating the IMM_SEL control signal

To generate the IMM_SEL control signal, the FUCNT3 bits of the instruction were used to obtain the proper immediate variant for the instructions given below.

- SLTIU - This is the only instruction that requires an unsigned immediate variant.
- SLLI, SRLI, SRAI - These instructions require the immediate variant that contains the shift amount.

FUNCT3 bits in SLTIU, SLLI, SRLI and SRAI instructions were used along with the IMM_TYPE intermediate signal to generate the IMM_SEL control signal.

Immediate Type	**INPUT**						**OUTPUT**		
	IMM_TYPE[2]	**IMM_TYPE[1]**	**IMM_TYPE[0]**	**FUNCT[2]**	**FUNCT[1]**	**FUNCT[0]**	**IMM_SEL[2]**	**IMM_SEL[1]**	**IMM_SEL[0]**
U	0	0	0	x	x	x	0	0	0
J	0	0	1	x	x	x	0	0	1
S	0	1	0	x	x	x	0	1	0
B	0	1	1	x	x	x	0	1	1
I_signed	1	0	0	x	x	x	1	0	0
I_shift	1	0	0	x	0	1	1	0	1
I_unigned	1	0	0	0	1	1	1	1	1

Table 8 : Truth table for generating the IMM_SEL control signal

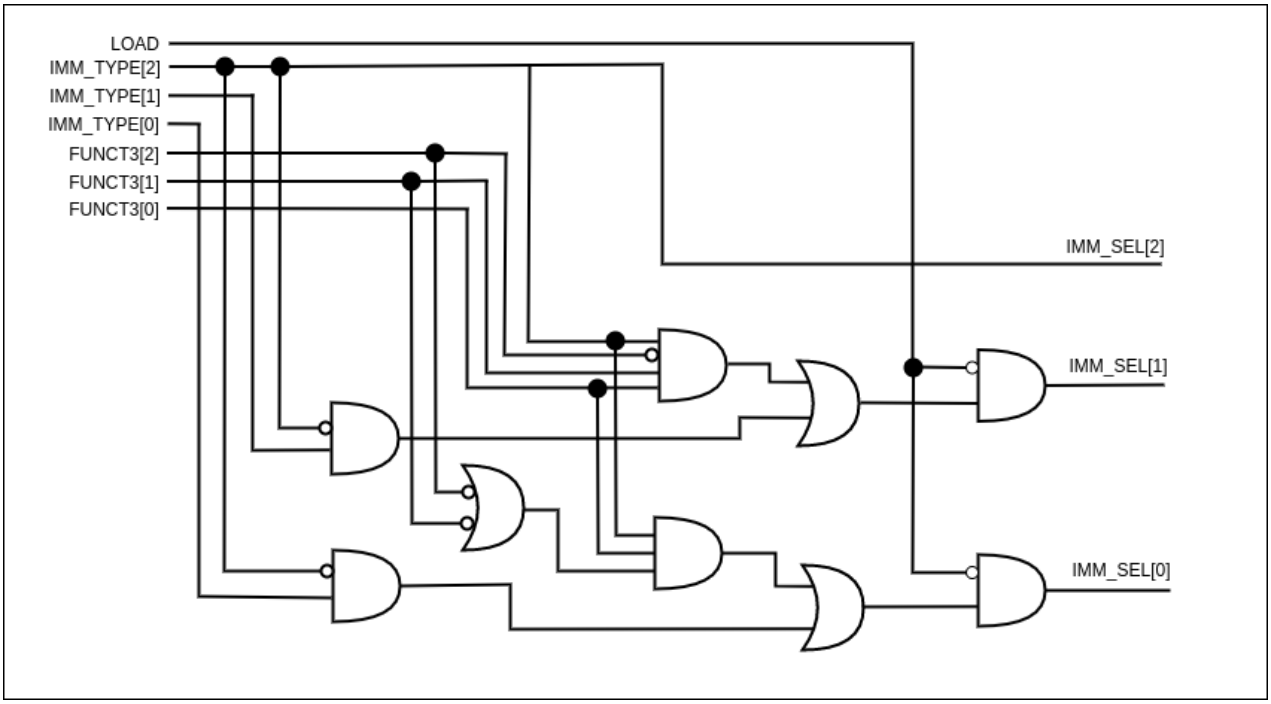


FIGURE 4 : Combinational circuit generating the IMM_SEL control signal

2.1.2.3. Generating the ALU_OP control signal

ALUOP_TYPE intermediate signal, FUNCT3, FUNCT7, IMM_SEL control signal, and R_TYPE(generated from the OPCODE) were used to generate the ALU_OP control signal. 4 variants can be considered when generating the control signal.

- R - Type, SLLI, SRLI and SRAI instructions

ALU_OP for these instructions were generated by concatenating the 3 bits in the FUNCT3, 5th bit in FUNCT7 and 0th bit in the FUNCT7.

- I - Type instructions without SLLI, SRLI and SRAI instructions

ALU_OP for these instructions were generated by concatenating the 3 bits in the FUNCT3, 2 zero signals.

- All other instructions that uses ALU

ALU_OP for these instructions was an ADD operation.

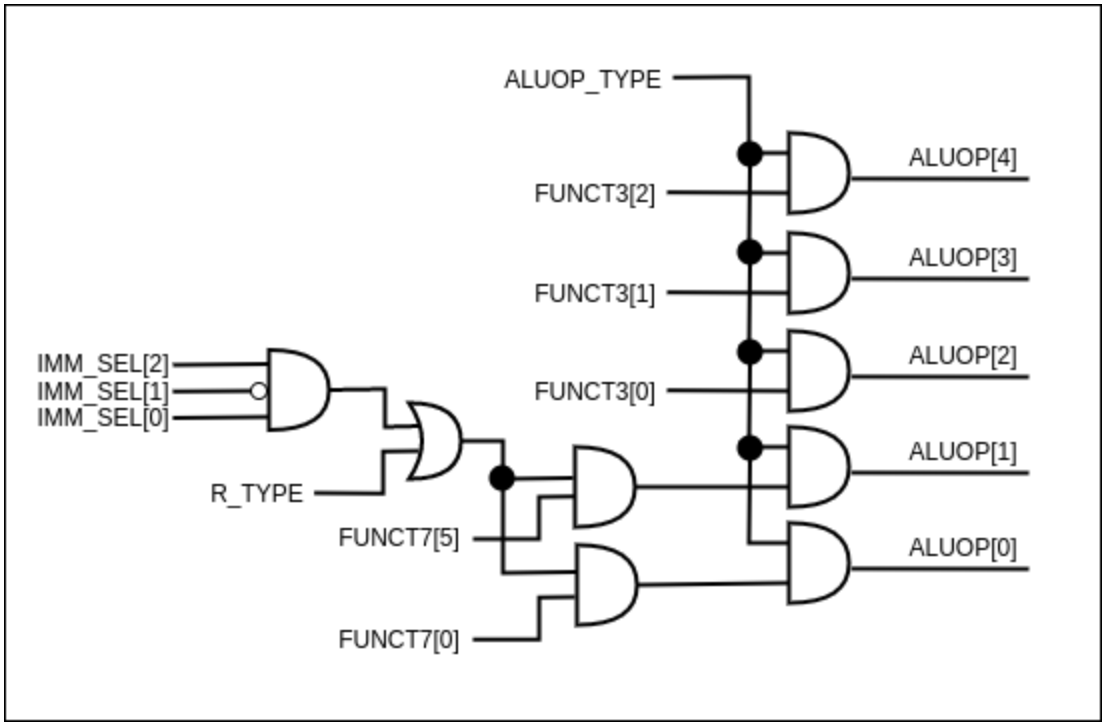


FIGURE 5 : Combinational circuit generating the ALU_OP control signal

2.1.2.4. Generating the BRANCH_JUMP control signal

BL intermediate signal, OPCODE[2] and FUNCT3 bits were used to generate the BRANCH_JUMP control signal. In B - Type instructions, the FUNCT3 bits define the type of branching. Therefore FUNCT3 bits were used to identify the branch type. OPCODE[2] bit was used to distinguish between the B - Type and the J - Type instructions.

Branch Type	INPUT					OUTPUT		
	OPCODE [2]	BL	FUCNT 3[2]	FUCNT 3[1]	FUCNT 3[0]	BRANCH_JUMP [2]	BRANCH_JUMP [1]	BRANCH_JUMP [0]
BEQ	0	1	0	0	0	0	0	0
BNE	0	1	0	0	1	0	0	1
NO	x	0	x	x	x	0	1	0
J	1	1	x	x	x	0	1	1
BLT	0	1	1	0	0	1	0	0
BGE	0	1	1	0	1	1	0	1
BLTU	0	1	1	1	0	1	1	0
BGEU	0	1	1	1	1	1	1	1

Table 9 : Truth table for generating the BRANCH_JUMP control signal

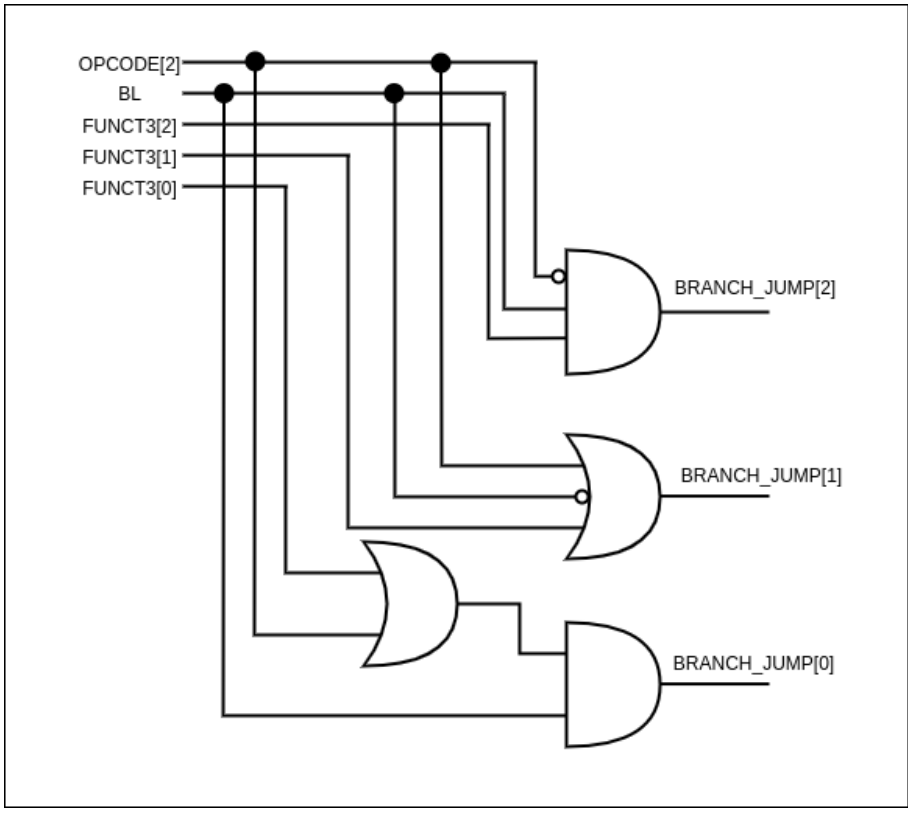


FIGURE 6 : Combinational circuit generating the BRANCH_JUMP control signal

2.1.2.5. Generating the READ_WRITE control signal

FUNCT3 bits, LOAD and STORE (LOAD and STORE by decoding the OPCODE) signals were used to generate the READ_WRITE control signal. LOAD and STORE signals are asserted when the instruction is a LOAD or a STORE instruction respectively. FUNCT3 bits were used to determine the number of bits (8 bit, 16 bit, 32 bit) loaded/stored.

When the instruction is a LOAD or a STORE, the READ_WRITE[3] bit will get asserted and by using only the READ_WRITE[3] bit, the data cache memory can assert the BUSYWAIT signal and the remaining bits of the READ_WRITE signal will determine the number of bits stored.. If the instruction is not a LOAD or a STORE the READ_WRITE[3] bit is cleared.

Load/Store type	**INPUT**					**OUTPUT**			
	STORE	**LOAD**	**FUNCT3[2]**	**FUNCT3[1]**	**FUNCT3[0]**	**READ_WRITE[3]**	**READ_WRITE[2]**	**READ_WRITE[1]**	**READ_WRITE[0]**
No load/store	0	0	x	x	x	0	0	0	0
	1	1	x	x	x				
LB	0	1	0	0	0	1	0	0	0
LH	0	1	0	0	1	1	0	0	1
LW	0	1	0	1	0	1	0	1	0
LBU	0	1	1	0	0	1	1	0	0
LHU	0	1	1	0	1	1	1	0	1
SB	1	0	0	0	0	1	0	1	1
SH	1	0	0	0	1	1	1	1	
SW	1	0	0	1	0	1	1	1	1

TABLE 10: Truth table for generation of the READ_WRITE control signal

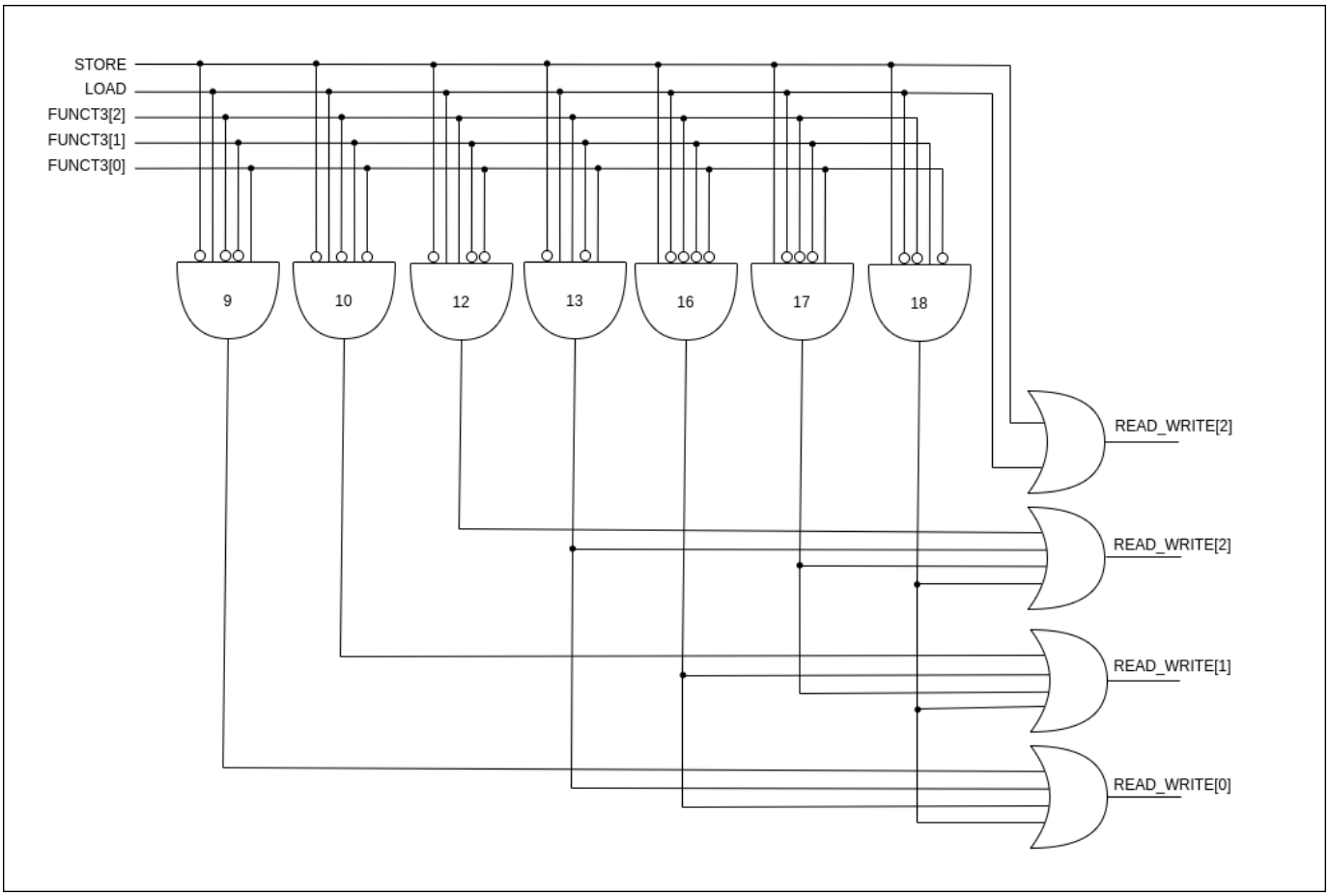


FIGURE 7 : Combinational Circuit Generating the READ_WRITE Control Signal

2.1.3. Instructions and the Control Signals

INSTRUCTION	PC_SEL	IMM_SEL	OP1SEL	OP2SEL	ALUOP	READ_WRITE	REG_WRITE_EN	WB_SEL	BRANCH_JUMP
LUI	PC + 4	U	*	*	*	NO_RW	1	IMM	NO
AUIPC	PC + 4	U	PC	IMM	ADD	NO_RW	1	ALU	NO
JAL	ALU	J	PC	IMM	ADD	NO_RW	1	PC + 4	J
JALR	ALU	I	DATA1	IMM	ADD	NO_RW	1	PC + 4	J
BEQ	PC + 4 / ALU	B	PC	IMM	ADD	NO_RW	0	*	BEQ
BNE	PC + 4 / ALU	B	PC	IMM	ADD	NO_RW	0	*	BNE
BLT	PC + 4 / ALU	B	PC	IMM	ADD	NO_RW	0	*	BLT
BGE	PC + 4 / ALU	B	PC	IMM	ADD	NO_RW	0	*	BGE
BLTU	PC + 4 / ALU	B	PC	IMM	ADD	NO_RW	0	*	BLTU

BGEU	PC + 4 / ALU	B	PC	IMM	ADD	NO_RW	0	*	BGEU
LB	PC + 4	I_signed	DATA1	IMM	ADD	LB	1	MEM	NO
LH	PC + 4	I_signed	DATA1	IMM	ADD	LH	1	MEM	NO
LW	PC + 4	I_signed	DATA1	IMM	ADD	LW	1	MEM	NO
LBU	PC + 4	I_signed	DATA1	IMM	ADD	LBU	1	MEM	NO
LHU	PC + 4	I_signed	DATA1	IMM	ADD	LHU	1	MEM	NO
SB	PC + 4	S	DATA1	IMM	ADD	SB	0	*	NO
SH	PC + 4	S	DATA1	IMM	ADD	SH	0	*	NO
SW	PC + 4	S	DATA1	IMM	ADD	SW	0	*	NO
ADDI	PC + 4	I_signed	DATA1	IMM	ADD	NO_RW	1	ALU	NO
SLTI	PC + 4	I_signed	DATA1	IMM	SLT	NO_RW	1	ALU	NO
SLTIU	PC + 4	I_unsigned	DATA1	IMM	SLTU	NO_RW	1	ALU	NO
XORI	PC + 4	I_signed	DATA1	IMM	XOR	NO_RW	1	ALU	NO
ORI	PC + 4	I_signed	DATA1	IMM	OR	NO_RW	1	ALU	NO
ANDI	PC + 4	I_signed	DATA1	IMM	AND	NO_RW	1	ALU	NO
SLLI	PC + 4	I_shift	DATA1	IMM	SLL	NO_RW	1	ALU	NO
SRLI	PC + 4	I_shift	DATA1	IMM	SRL	NO_RW	1	ALU	NO
SRAI	PC + 4	I_shift	DATA1	IMM	SRA	NO_RW	1	ALU	NO
ADD	PC + 4	*	DATA1	DATA2	ADD	NO_RW	1	ALU	NO
SUB	PC + 4	*	DATA1	DATA2	SUB	NO_RW	1	ALU	NO
SLL	PC + 4	*	DATA1	DATA2	SLL	NO_RW	1	ALU	NO
SLT	PC + 4	*	DATA1	DATA2	SLT	NO_RW	1	ALU	NO
SLTU	PC + 4	*	DATA1	DATA2	SLTU	NO_RW	1	ALU	NO
XOR	PC + 4	*	DATA1	DATA2	XOR	NO_RW	1	ALU	NO
SRL	PC + 4	*	DATA1	DATA2	SRL	NO_RW	1	ALU	NO
SRA	PC + 4	*	DATA1	DATA2	SRA	NO_RW	1	ALU	NO
OR	PC + 4	*	DATA1	DATA2	OR	NO_RW	1	ALU	NO
AND	PC + 4	*	DATA1	DATA2	AND	NO_RW	1	ALU	NO
MUL	PC + 4	*	DATA1	DATA2	MUL	NO_RW	1	ALU	NO
MULH	PC + 4	*	DATA1	DATA2	MULH	NO_RW	1	ALU	NO
MULHSU	PC + 4	*	DATA1	DATA2	MULHSU	NO_RW	1	ALU	NO
MULHU	PC + 4	*	DATA1	DATA2	MULHU	NO_RW	1	ALU	NO
DIV	PC + 4	*	DATA1	DATA2	DIV	NO_RW	1	ALU	NO
DIVU	PC + 4	*	DATA1	DATA2	DIVU	NO_RW	1	ALU	NO
REM	PC + 4	*	DATA1	DATA2	REM	NO_RW	1	ALU	NO
REMU	PC + 4	*	DATA1	DATA2	REMU	NO_RW	1	ALU	NO

Table 11 : Instructions and Control Signals

2.2. ALU

ALU is the main hardware unit that performs arithmetic and logic operations. This ALU supports 18 operations.

- ADD
- SUB
- SLL
- SLT
- SLTU
- XOR
- SRL
- SRA
- OR
- AND
- MUL
- MULH
- MULHU
- MULHSU
- DIV
- DIVU
- REM
- REMU

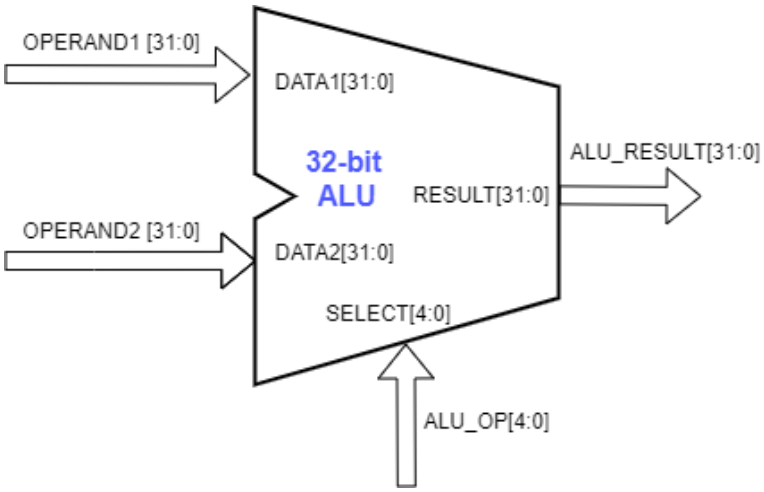


FIGURE 8 : ALU

Inputs to the ALU are,

- DATA1[31:0]
- DATA2[31:0]
- ALU_OP[4:0] - control signal
  (See ALU_OP for encoding and Generating the ALU_OP control signal for design)

Output of the ALU,

- ALU_RESULT[31:0]

2.3. Register File

Register file contains 32 registers and the size of a register is 32 bits. Register x0 is set to zero by making all the bits in register x0 to 0. Registers x1 to x31 can be used by the instructions.
In RISC-V ISA NOP instruction is encoded as an ADDI instruction. In this case the register x0 containing the value zero will be used.
NOP ⇒ ADDI x0, x0, 0

Inputs to the register file,

- WRITE_DATA[31:0]
- WRITE_ADDRESS[4:0]
- DATA1_ADDRESS[4:0]
- DATA2_ADDRESS[4:0]
- REG_WRITE_EN control signal
- RESET
- CLK

Outputs of the register file,

- DATA1[31:0]
- DATA2[31:0]

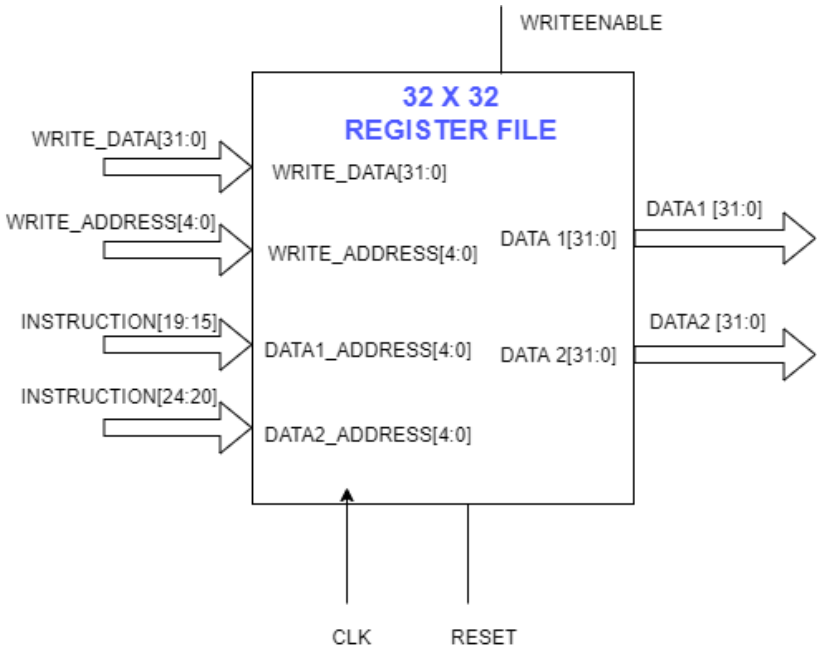


FIGURE 9 : Register File

2.4. Branching and Jump Detection Unit

This unit is for detecting whether the branch or the jump has to be taken or not. Inputs to this unit are,

- DATA1[31:0]
- DATA2[31:0]
- BRANCH_JUMP control signal
  (See BRANCH_JUMP for encoding)

Output of this unit is,

- PC_SELECT

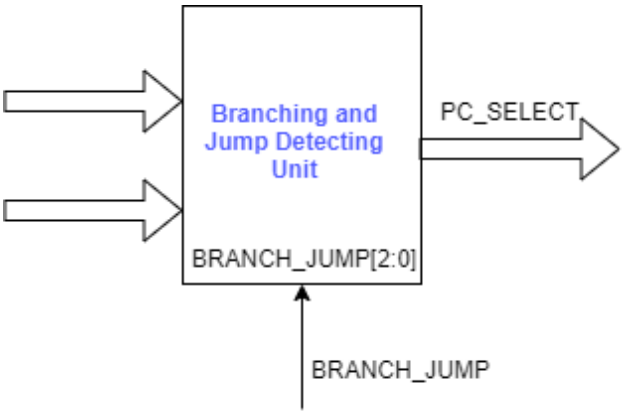


FIGURE 10 : Branch Jump Detection Unit

This unit will contain a comparator implemented using behavioural modeling and a combinational logic circuit to generate the PC_SELECT control signal. Branch and jump detection is done in 2 steps.

1. DATA1 and DATA2 values will be the inputs for the comparator. The comparator will output two 1 bit signals by comparing the input values,
   1.1. EQUAL - If DATA1 and DATA2 are equal this signal will be set. Else it will be cleared.
   1.2. LESS_THAN - If DATA1 is less than DATA2 this signal will be set. Else it will be cleared.

2. BRANCH_JUMP control signal, EQUAL and LESS_THAN intermediate signals will be the inputs to the combinational logic circuit. The combinational logic circuit will generate the PC_SELECT control signal depending on its inputs.

Branch Type	**INPUT**					**OUTPUT**
	BRANCH_JUMP[2]	**BRANCH_JUMP[1]**	**BRANCH_JUMP[0]**	**Equal**	**Less Than**	**PC_SELECT**
BEQ	0	0	0	0	x	0
				1	x	1
BNE	0	0	1	0	x	1
				1	x	0
NO	0	1	0	x	x	0
J	0	1	1	x	x	1
BLT	1	0	0	x	0	0
				x	1	1
BGE	1	0	1	0	0	1
				0	1	0
				1	x	1
BLTU	1	1	0	x	0	0
				x	1	1
BGEU	1	1	1	0	0	1
				0	1	0
				1	x	1

Table 12 : Truth Table of the combinational logic circuit

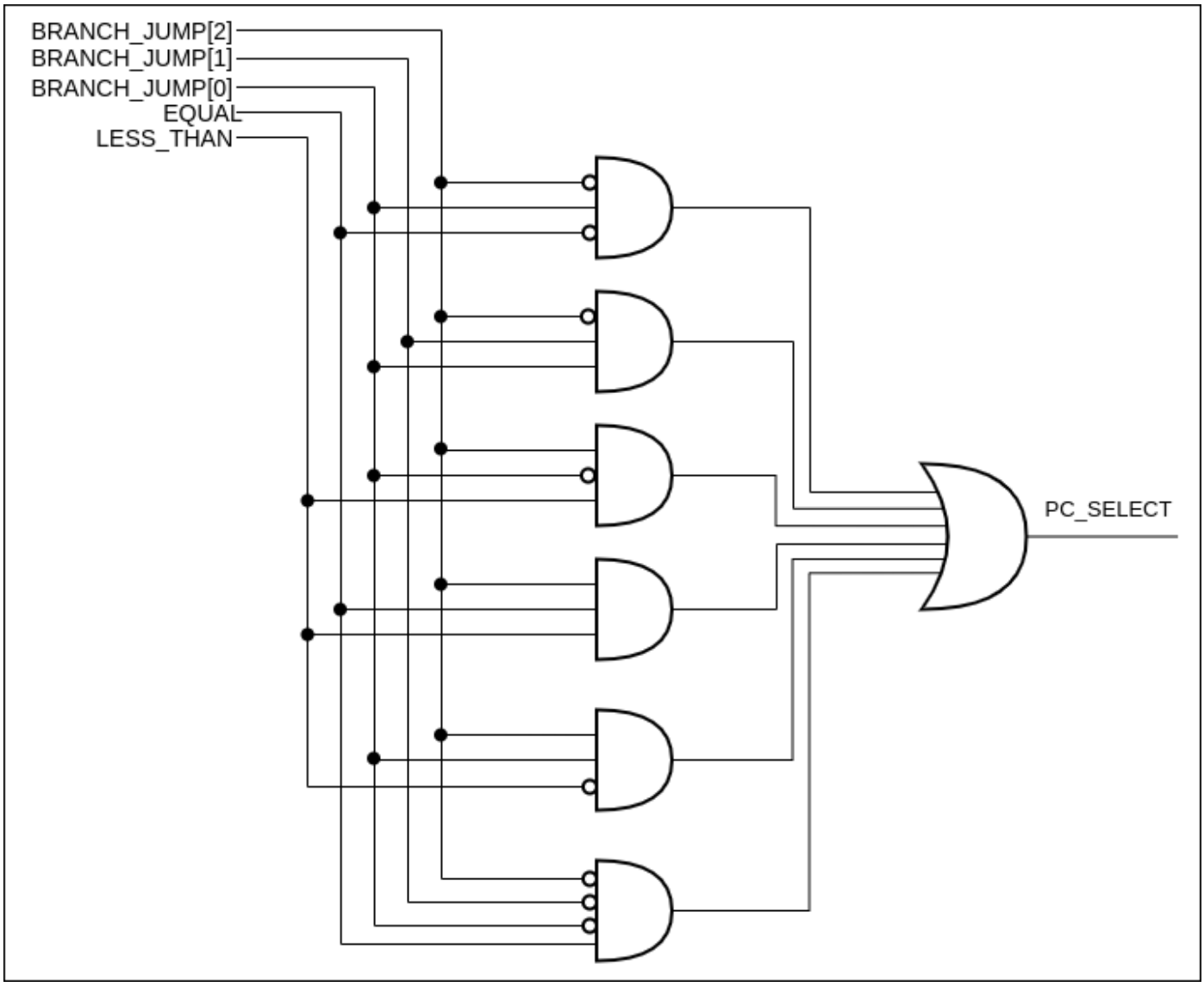


FIGURE 11 : Combinational Circuit

2.4.1. PC_SELECT Control Signal

PC_SELECT is a 1 bit control signal generated from the branch and jump detection unit. This signal will select the address source for the PC register. The PC register has 2 address sources.

- Address computed by the ALU
- Address computed by adding 4 to the current PC

Encoding of the PC_SELECT signal is shown in Table 10.

Address Source	PC_SELECT
PC + 4	0
Computer address from ALU	1

Table 13 : PC_SELECT Encoding

2.5. Immediate Value Generation Unit

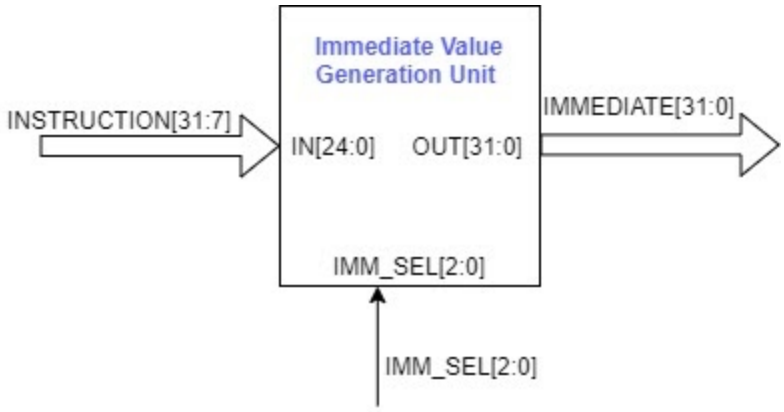
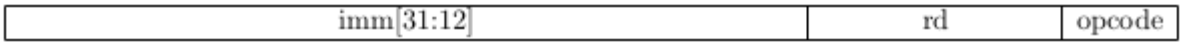


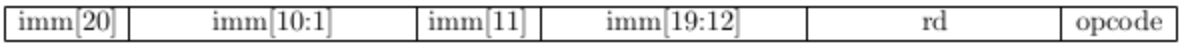
FIGURE 12 : Immediate Value Generation Unit

Instructions with immediate values can be categorized into 4 groups and I-Type instructions can be further categorized into 3 sub groups.

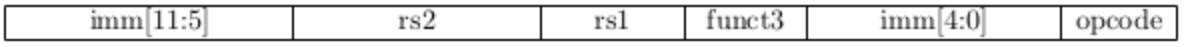
- U type Immediate



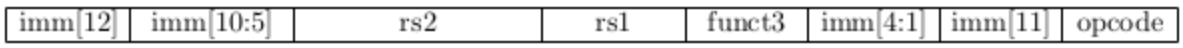
- J type Immediate



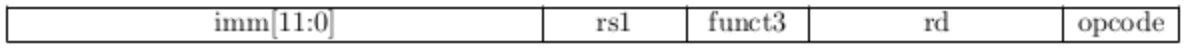
- S type Immediate



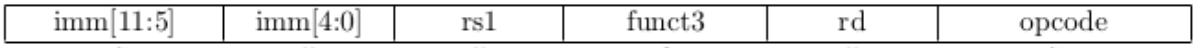
- B type Immediate



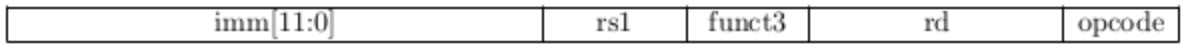
- I type Immediate with signed Extension (Extend with sign bit)



- I type immediate containing shift amount



- I type immediate with unsigned Extension (Extend with Zero)



Inputs to the immediate value generation unit are,

- INSTRUCTION[31:7]
- IMM_SEL control signal[2:0]

Output of the immediate value generation unit is,

- IMMEDIATE_VALUE[31:0]

Immediate value generation unit will generate the immediate values and sign extends by rerouting the wires. Depending on the IMM_SEL control signal, this unit will output the proper immediate value. Immediate value generation and sign extending of the 7 categories are shown in the figure below.

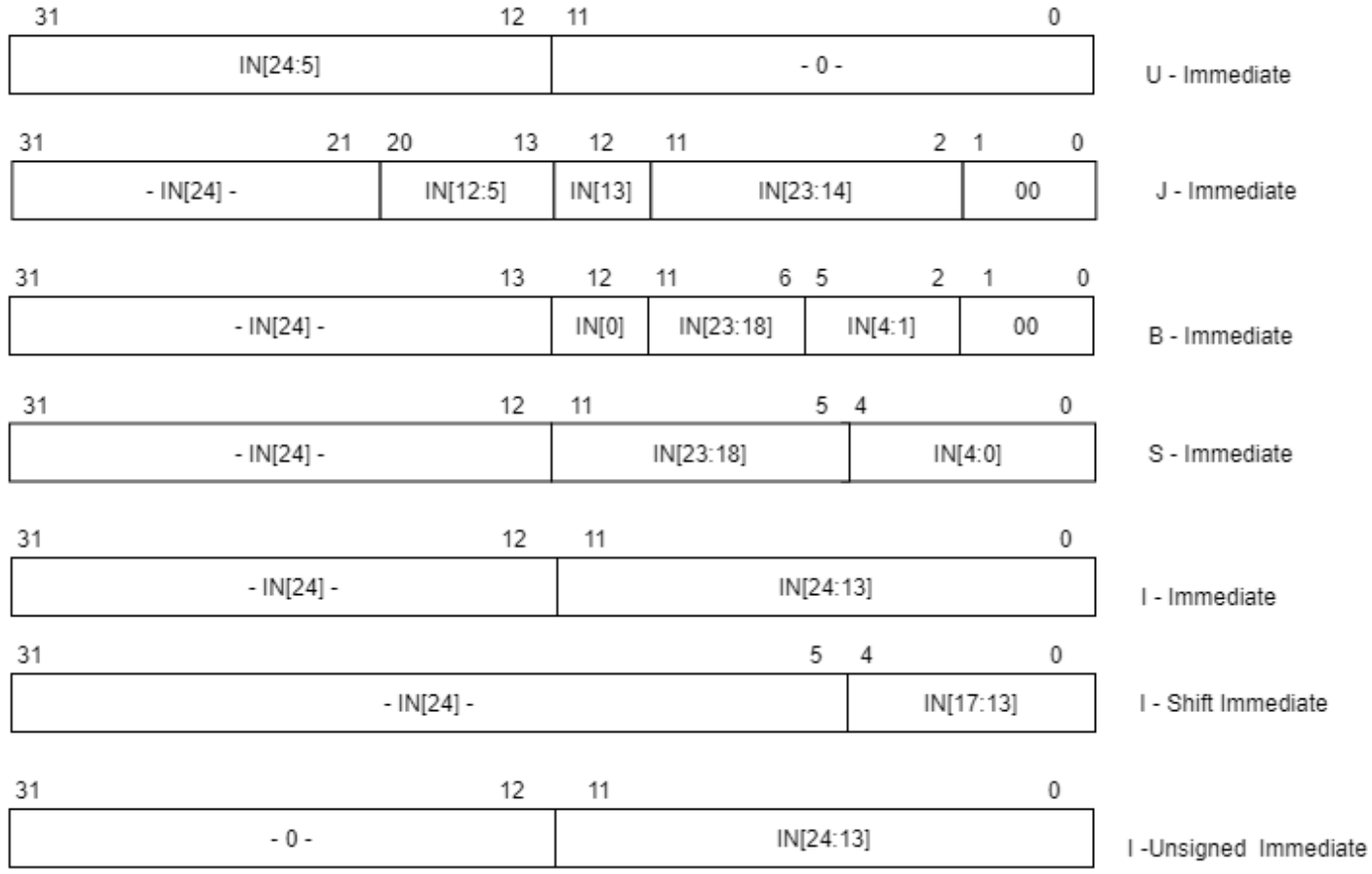


FIGURE 13 : Rerouting and Sign Extension of Immediate Values

2.6. Program counter register

Program counter register stores the address of the instruction. The value stored in this register is used when fetching the instruction from the instruction memory. Writing to the PC register is synchronous to the positive clock edge and the reading from the PC register is asynchronous. When the reset signal is set, the value in the PC register will be set to -4 and the program will restart from the next clock cycle.

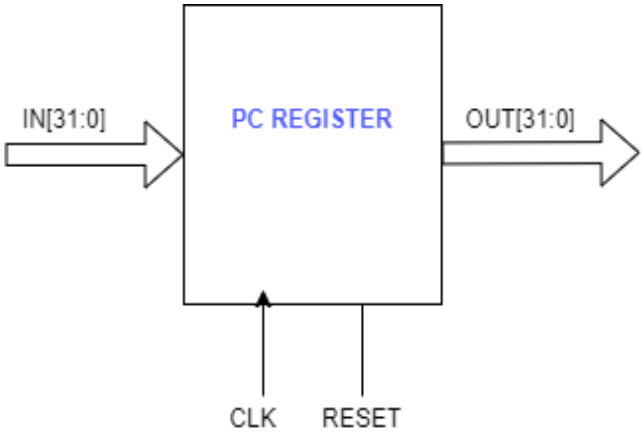


FIGURE 14 : Program Counter Register

2.7. Multiplexers

Three 2x1 and one 4x1 multiplexers are used in the design. Depending on the select signal, the multiplexer will output the corresponding value.

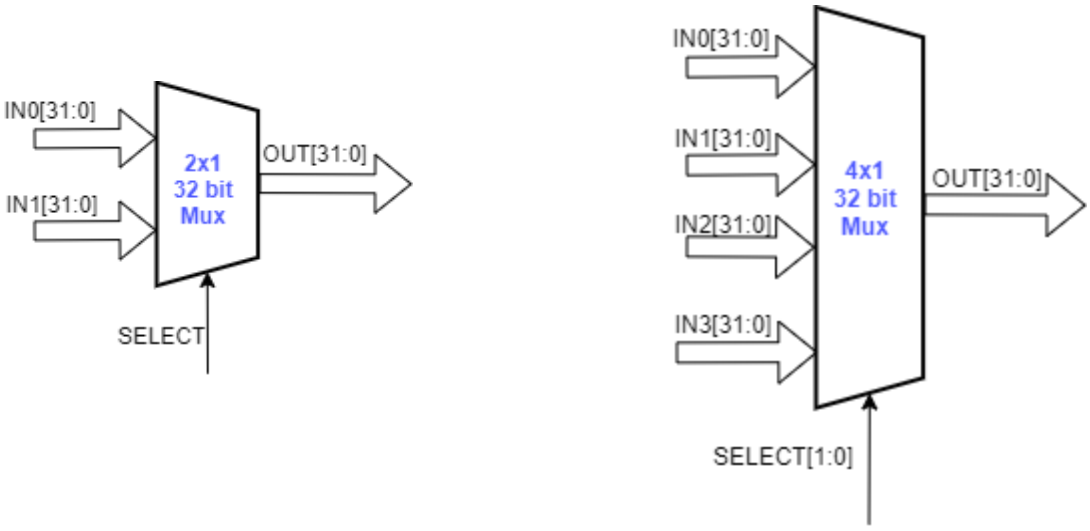


FIGURE 15 : Multiplexers

Table 11 shows the details about the multiplexers.

	Type of MUX	Inputs from	Select signal	Output to
Selecting the address source for PC register	2x1 MUX	PC+4, Address computer by the ALU	PC_SELECT control signal	PC register
Selecting the operand 1 for the ALU	2x1 MUX	DATA1 from register file, PC value	OP1_SEL	DATA1 input of the ALU
Selecting the operand 2 for the ALU	2x1 MUX	DATA2 from register file, Immediate value	OP2_SEL	DATA2 input of the ALU
Selecting the writeback source	4x1 MUX	ALU result, PC + 4 value, Immediate value, Read data from data memory	WB_SEL	Write data input of the register file

Table 14 : Multiplexer Details

2.8. Program counter incrementing adder

This hardware unit will add 4 to the input value. This unit is used for incrementing the PC and the PC + 4 value will be the output of this device.

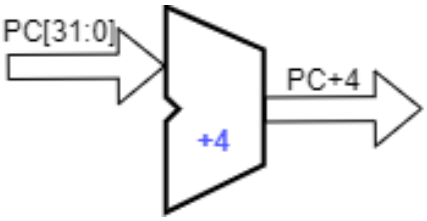


FIGURE 16 : PC incrementing adder

2.9. Pipeline Registers

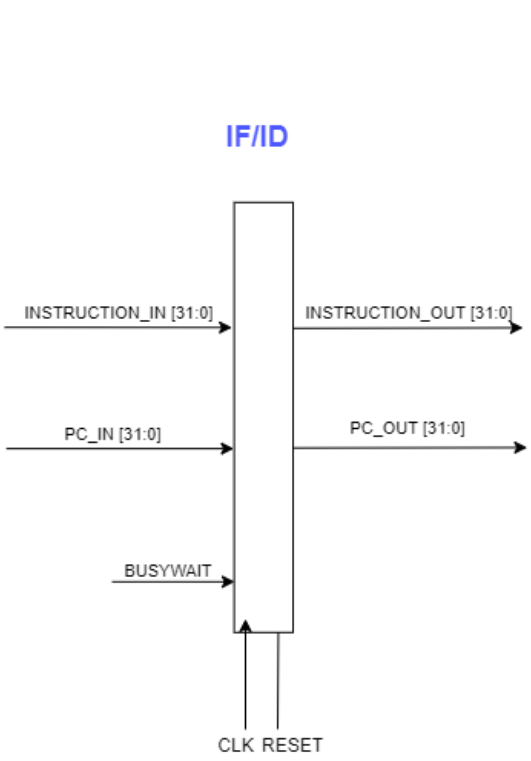
In CPU pipelines, pipeline registers are used to store the information and control signals related to the instruction. In this CPU design there are 5 stages,

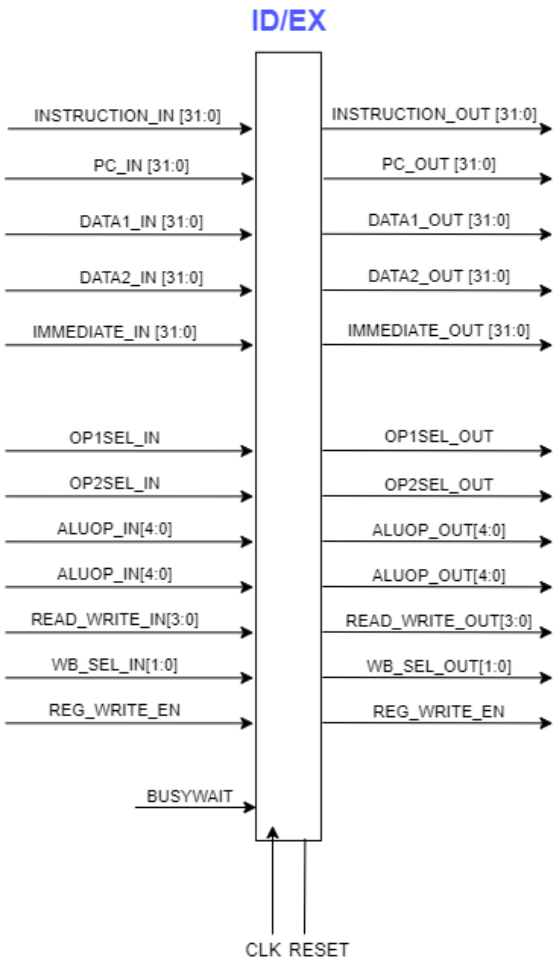
- Instruction Fetch stage
- Instruction Decode stage
- Execution stage
- Memory Access stage
- Writeback stage

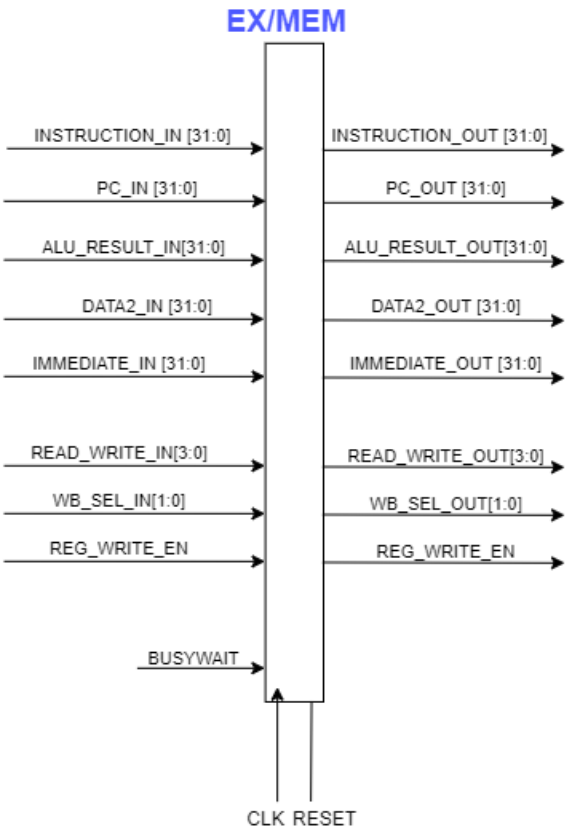
In between each stage there is a pipeline register.

- IF/ID pipeline register
- ID/EX pipeline register
- EX/MEM pipeline register
- MEM/WB pipeline register

Data and control signals are written to the pipeline registers at the positive edge of the clock cycle and when the reset signal is set, the pipeline registers will get reset.







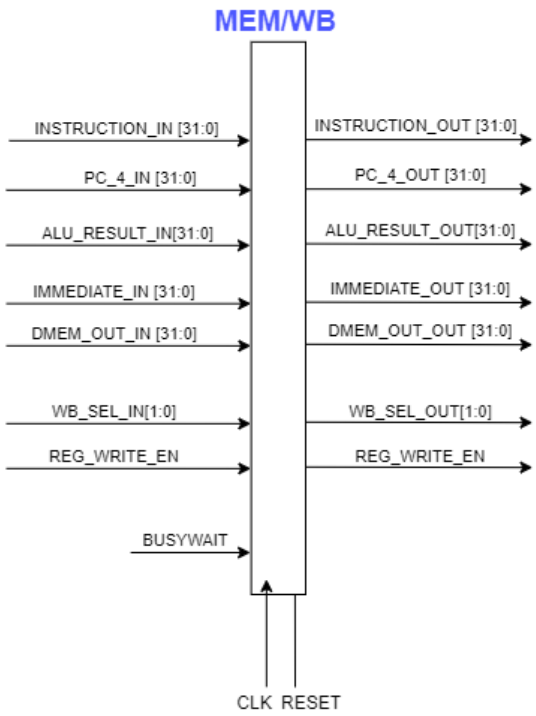


FIGURE 17 : Pipeline Registers

3. Memory Hierarchy

3.1. Overview

Memory is organized into a hierarchy so that from the point of view of the CPU it will have a large and fast memory. The CPU directly communicates with the caches which are faster than the primary memory and smaller in capacity. Memory is organized as shown in the following figure and it consists of,

- A data cache
- An instruction cache
- A data memory
- An instruction cache

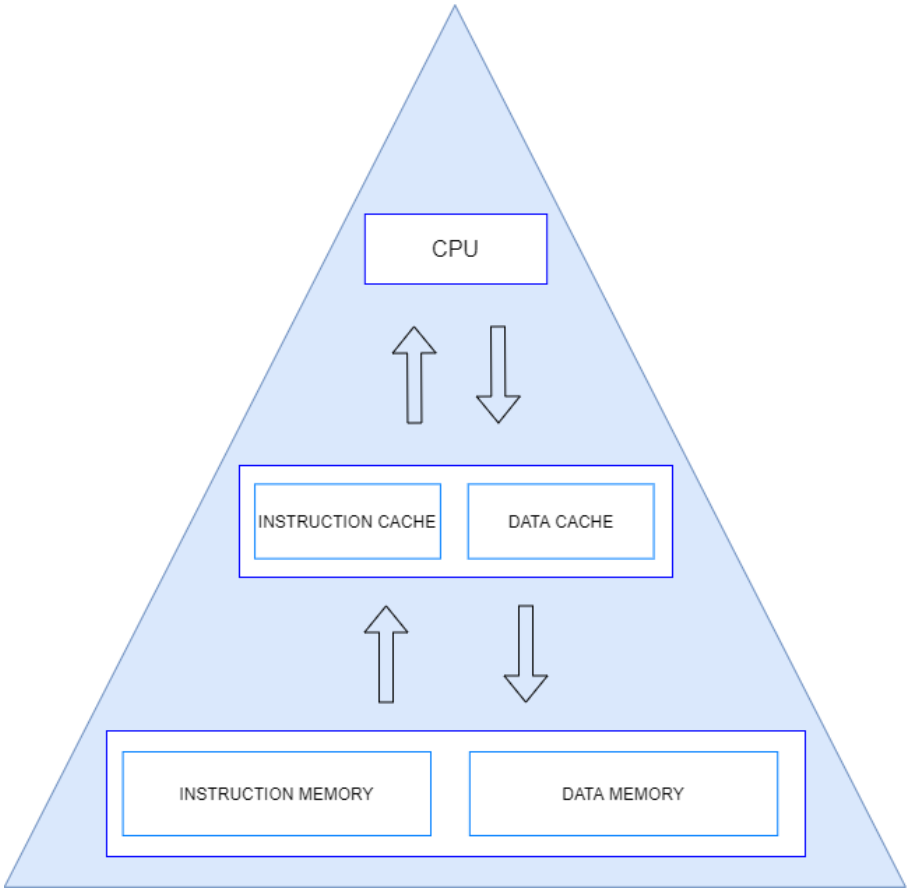
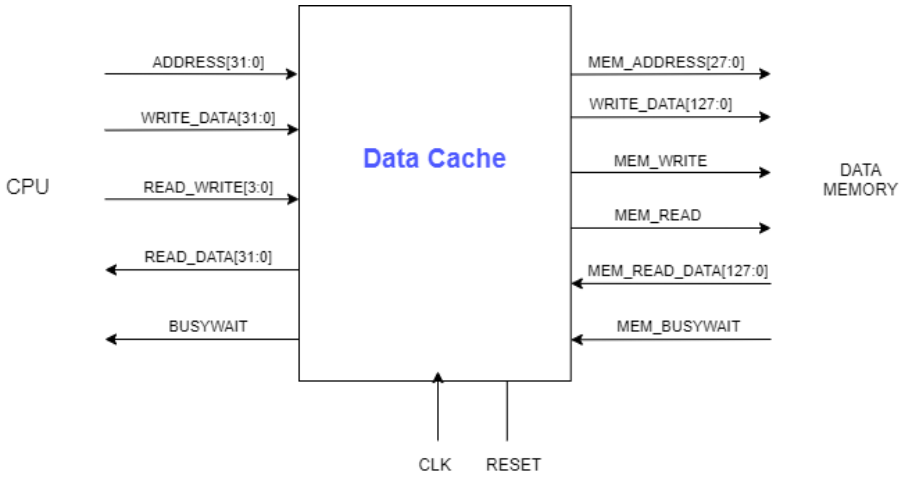


FIGURE 18 : Memory Hierarchy

3.2. Data cache

Data cache sits in between the CPU and the data memory. When the CPU loads/stores data from/to the data cache, depending on the validity of the data blocks stored in the cache it will undergo a hit or a miss.



In the case of a hit, the cache will read/write the data from/to the cache. This will not stall the CPU.

In the case of a miss, the cache will stall the CPU and fetch the correct block from the memory and update the cache. Once the cache is updated, the CPU can read/write the data from/to the cache.

FIGURE 19 : Data Cache

Inputs to Data Cache from CPU are,

- ADDRESS[31:0]
- WRITE_DATA[31:0]
- READ_WRITE[3:0]

Outputs from Data Cache to CPU are,

- READ_DATA[31:0]
- BUSYWAIT

Inputs to Data Cache from Data Memory are,

- MEM_READ_DATA[127:0]
- MEM_BUSYWAIT

Outputs from Data Cache to Data Memory are,

- MEM_ADDRESS[27:0]
- WRITE_DATA[127:0]
- MEM_WRITE
- MEM_READ

When the CPU sends an address to the cache and if the READ_WRITE[3] bit of the control signal is set, the cache will set the BUSYWAIT signal to stall the CPU. Then the cache will check if the address is a hit or a miss. Once the operations in the cache are completed and the CPU is served, the cache has to clear the BUSYWAIT. Clearing of the BUSYWAIT wait signal will happen at the next positive edge of the clock.

Data cache is implemented as a direct mapped cache and a cache block contains 16 bytes of data (4 words) and has 16 cache blocks. The implemented data cache can hold 256 bytes of data.

The input ADDRESS is separated into TAG, INDEX and OFFSET as the following figure.

FIGURE 20 : Address Separation

INDEX	V	D	TAG	DATA
0000				
0001				
0010				
0011				
0100				
0101				
0110				
0111				
1000				
1001				
1010				
1011				
1100				
1101				
1110				
1111				

FIGURE 21 : Data Cache Layout

In the case of a cache miss, until the data fetching from the data memory is completed, the cache controller will have to keep the signals sent to the memory in a stable state. A finite sate machine was used to implement the cache controller and the state diagram is shown in the following figure.

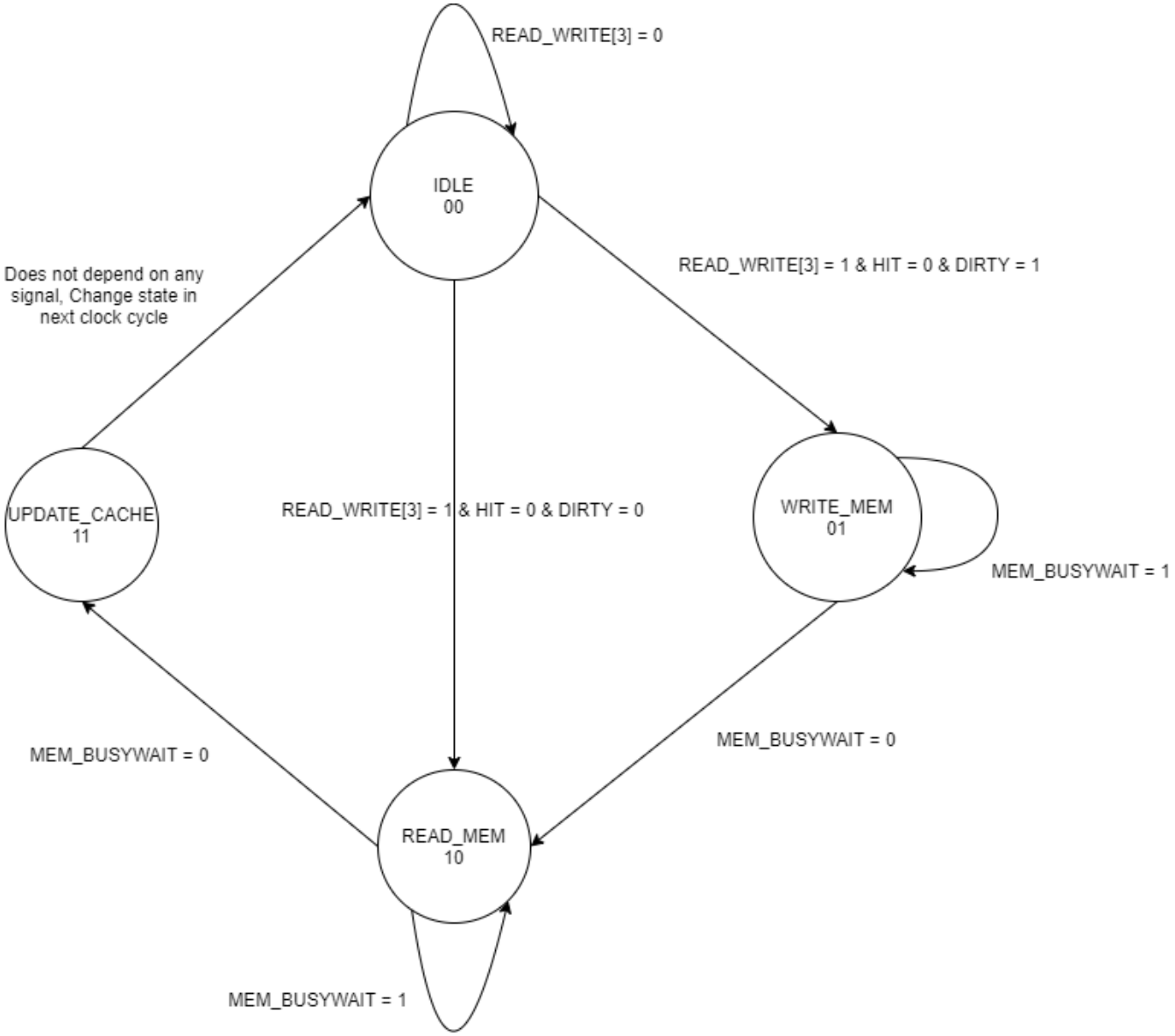


FIGURE 22 : Finite State Machine Diagram for Data Cache Controller

3.3. Data memory

Inputs to Data Memory from Data Cache are,

- ADDRESS[27:0]
- WRITE_DATA[127:0]
- READ
- WRITE

Outputs from Data Cache to CPU are,

- READ_DATA[127:0]
- BUSYWAIT

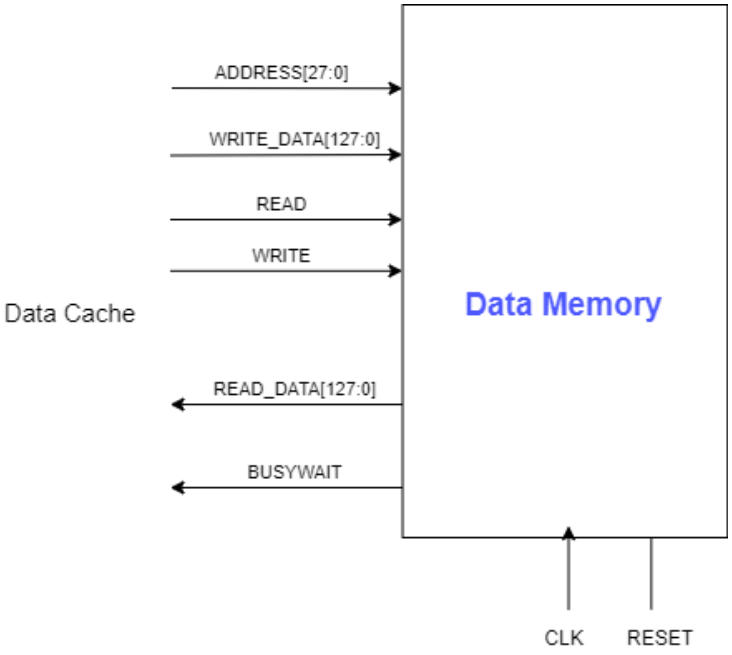


FIGURE 23 : Data Memory

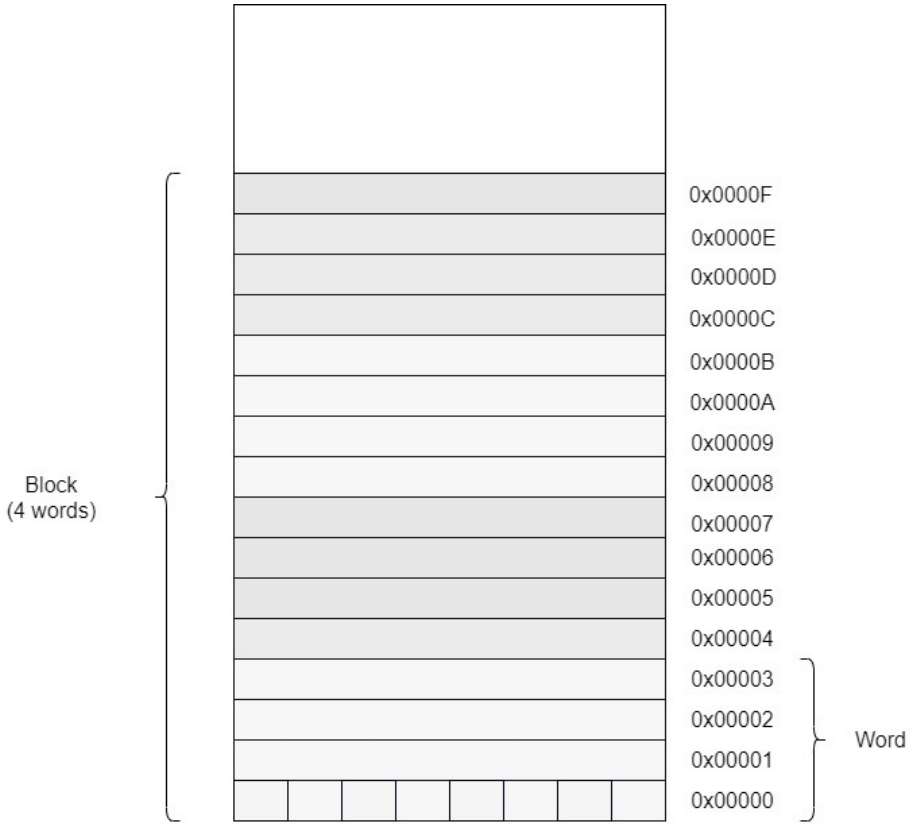
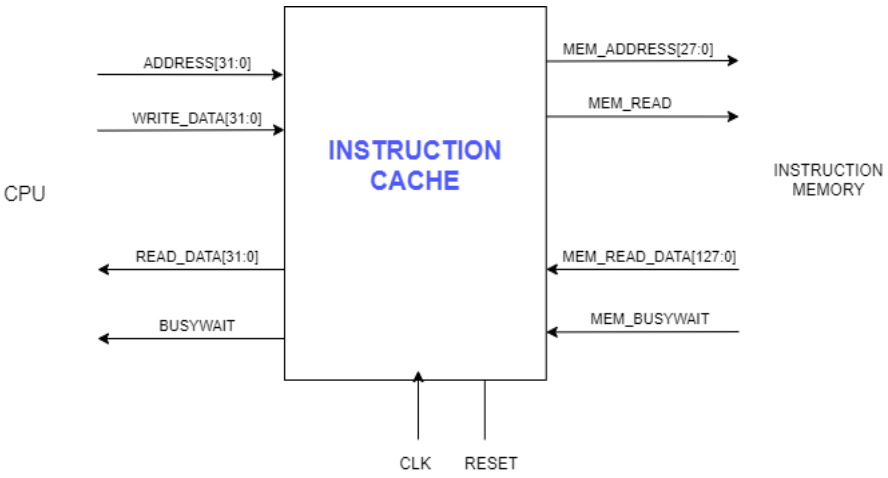


FIGURE 24 : Data Memory Layout

3.4. Instruction cache



Instruction cache operation is similar to the data cache and the only difference is that it does not undergo writing.

Inputs to Instruction Cache from CPU are,

- ADDRESS[31:0]
- WRITE_DATA[31:0]

Outputs from Instruction Cache to CPU are,

- READ_DATA[31:0]
- BUSYWAIT

FIGURE 25 : Instruction Cache

Inputs to Instruction Cache from Instruction Memory are,

- MEM_READ_DATA[127:0]
- MEM_BUSYWAIT

Outputs from Data Cache to Data Memory are,

- MEM_ADDRESS[27:0]
- MEM_READ

When considering the structure of the instruction cache, it is also implemented as a direct mapped cache. A cache block contains 16 bytes of data and has 8 blocks. This cache structure can store 128 bytes of data.

The input ADDRESS is separated into TAG, INDEX and OFFSET as the following figure.

FIGURE 26 : Address Separation

INDEX	V	TAG	DATA			
000						
001						
010						
011						
100						
101						
110						
111						

FIGURE 27 : Instruction Cache Layout

Similar to the data cache, instruction cache controller is also implemented using a finite state machine. In this state machine, there is no memory writing stage since instructions are not written to the memory.
The state diagram is shown in the following figure.

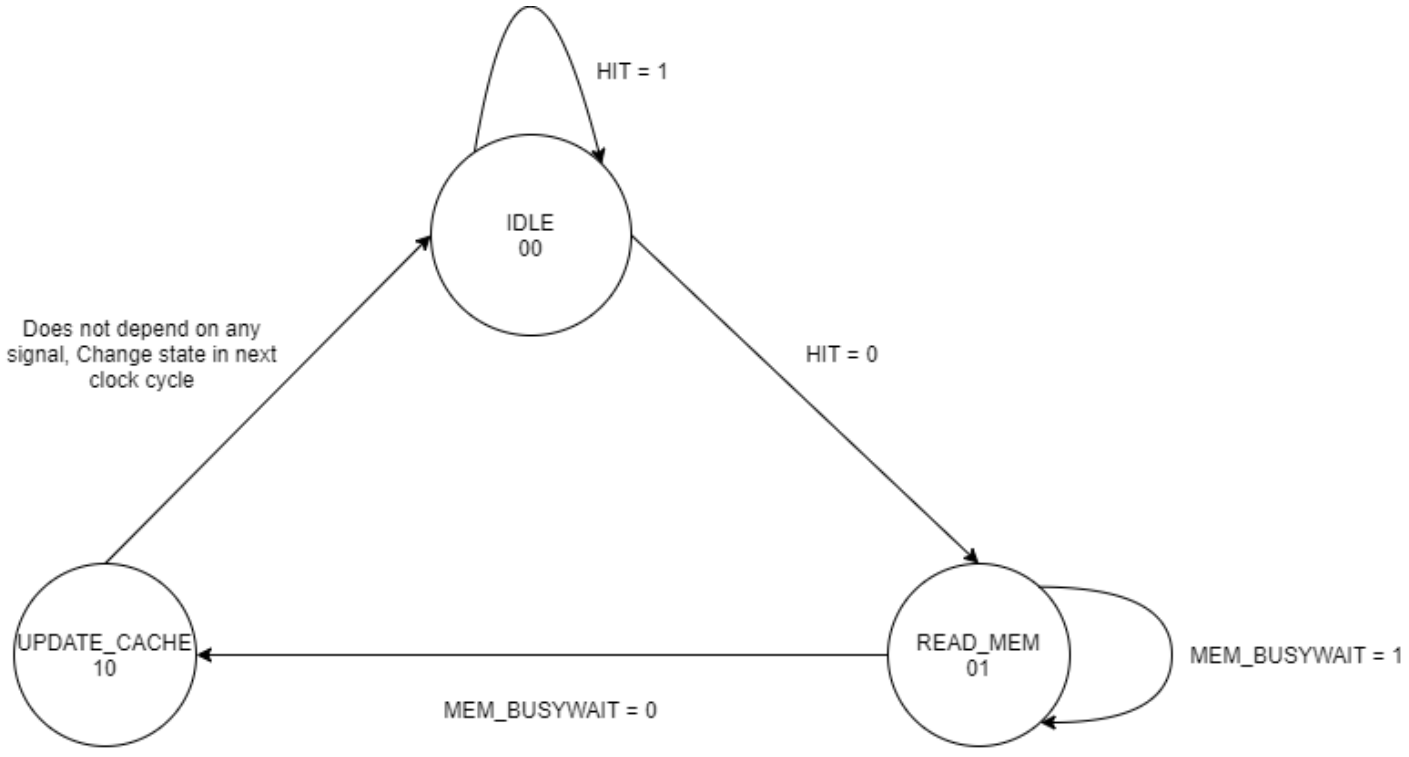


FIGURE 28 : Finite State Machine Diagram for Instruction Cache Controller

3.5. Instruction Memory

Inputs to Data Memory from Data Cache are,

- ADDRESS[27:0]
- WRITE_DATA[127:0]
- READ

Outputs from Data Cache to CPU are,

- READ_DATA[127:0]
- BUSYWAIT

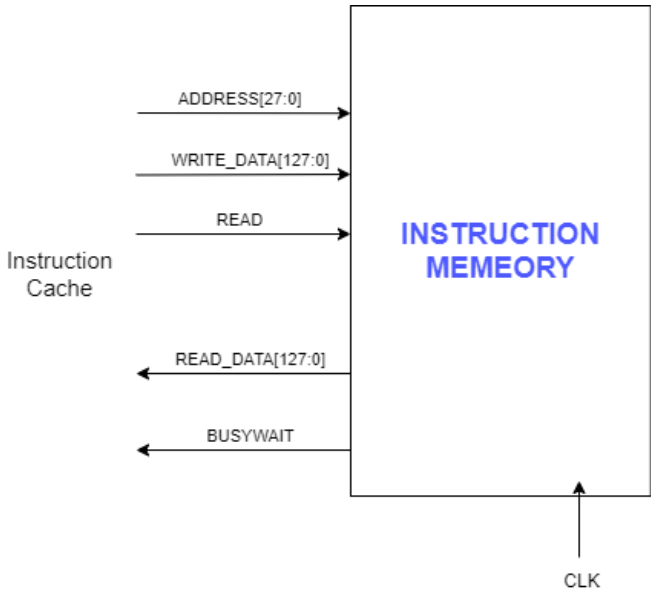


FIGURE 29 : Instruction Memory

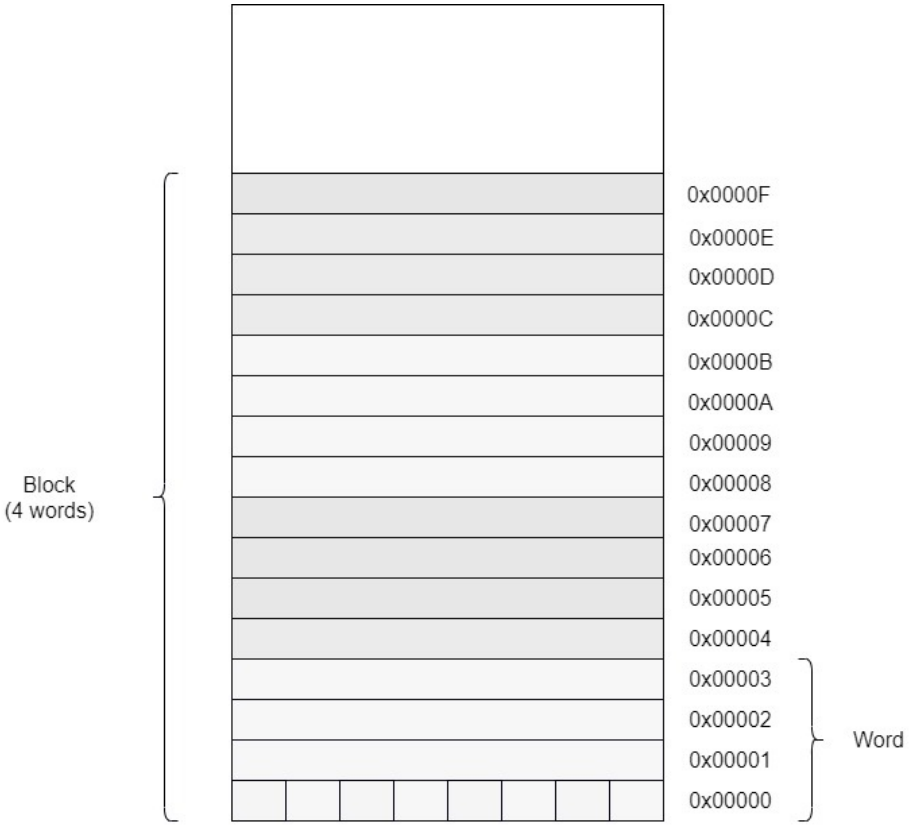


FIGURE 30 : Instruction Memory Layout

3. Timing and Simulation Delays

3.1. Simulation Delays of Hardware Units

3.1.1. Instruction Memory and Instruction Cache

Instruction memory is associated with the instruction cache. The memory access time will be determined by hits and misses. To determine the hit or miss it takes 2 time units and if it's a hit, it serves the data to the cpu after 1 time units. Altogether it takes 3 time units to access the instruction. And if it is a miss, then the cache will have to fetch the data from the instruction memory. To access the data from the instruction memory it takes 40 time units per byte. Altogether it takes 640 time units to fetch an entire block (16 bytes).

3.1.2. Register File

In the register file reading happens asynchronously. Register file read latency will be 2 time units. The writing to the register file happens synchronously to the positive edge of the clock. Register file write delay will be 1 time unit.

3.1.3. Data Memory and Data Cache

Data memory is associated with the data cache. The memory access time will be determined by hits and misses. To determine the hit or miss it takes 2 time units and if it's a hit, it serves the data to the cpu after 1 time units. Altogether it takes 3 time units to access the data from the cache memory. And if it is a miss, then the cache will have to fetch the data from the data memory. To access the data from the data memory it takes 40 time units per byte. Altogether it takes 640 time units to fetch an entire block (16 bytes).

3.1.4. Control Unit

Control unit generates the control signals of the instructions. Since the control signal generation was implemented using basic combinational logic circuits the delay was assumed as 1 time unit.

3.1.5. PC+4 adder

The PC+4 adder takes 1 time unit to calculate the PC+4 value.

3.1.6. PC Register

Writing to the PC register is done synchronously to the positive edge of the clock and it takes 1 time unit.

3.1.7. ALU

Depending on the complexity of the ALU operation following timing delays were used.

OPERATION	DELAY (time units)
AND	1
OR	1
XOR	1
ADD	2
SUB	2
SLT (Set less than)	2
SLTU (Set less than unsigned)	2
SLL (Shift left logical)	2
SRL (Shift right logical)	2
SRA (Shift right arithmetic)	2
MUL	3
MULH	3
MULHSU	3
DIV	3
DIVU	3
REM	3
REMU	3

Table 15 : ALU Timing Details

3.1.8. Pipeline Registers

Pipeline register writes happen synchronously to the clock at the positive edge of the clock. The writing latency is taken as 1 time units.

3.1.9. Branch Jump Detect Unit

This unit generates the PC_SELECT control signal depending on the BRANCH_JUMP signal. It generates the PC_SELECT signal asynchronously with the 2 time units of delay.

3.2. Clock Cycle Period and Overall Delays on the Pipeline Stages

Instruction fetch stage, execution stage and the memory access stage consumes the most number of time units and it is 4 time units. Therefore the clock cycle period was taken as 4 time units so that each and every stage will complete within a clock cycle.

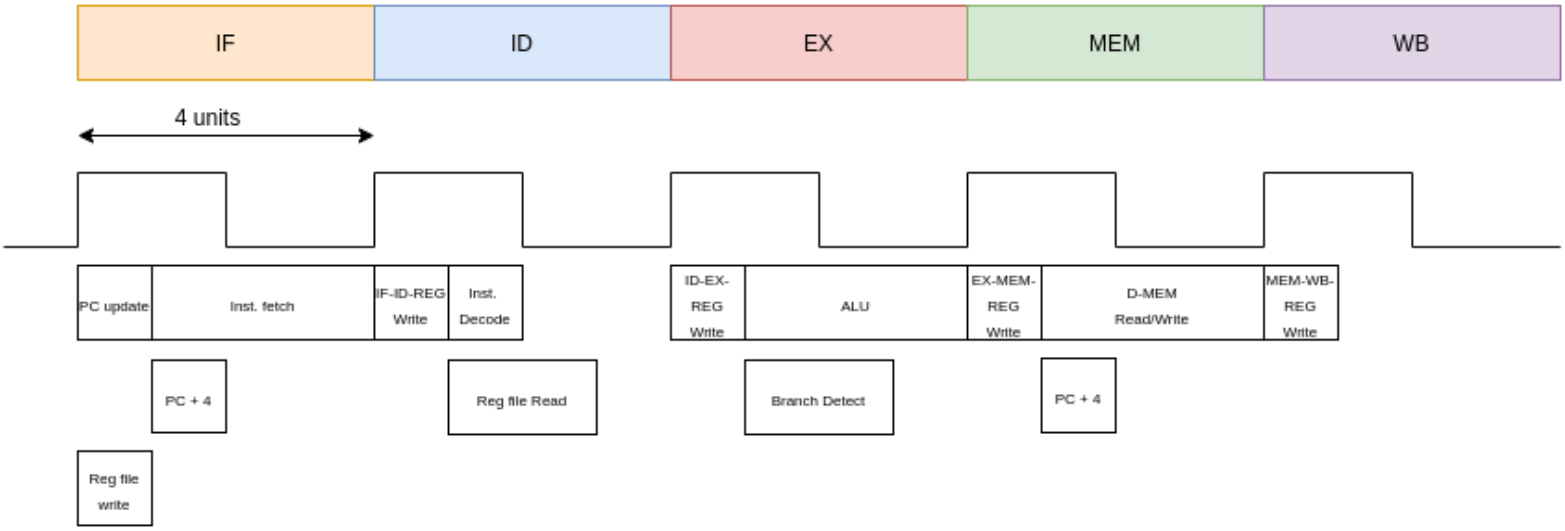


FIGURE 31 : Overall Delays of the Pipeline Stages

4. Handling Hazards

4.1. Data Hazards

To avoid data hazards nop instructions were used in the assembly program and assembly will encode nop instructions as addi x0 x0 0.

To avoid data hazard nop instructions were used as shown in the following figure.

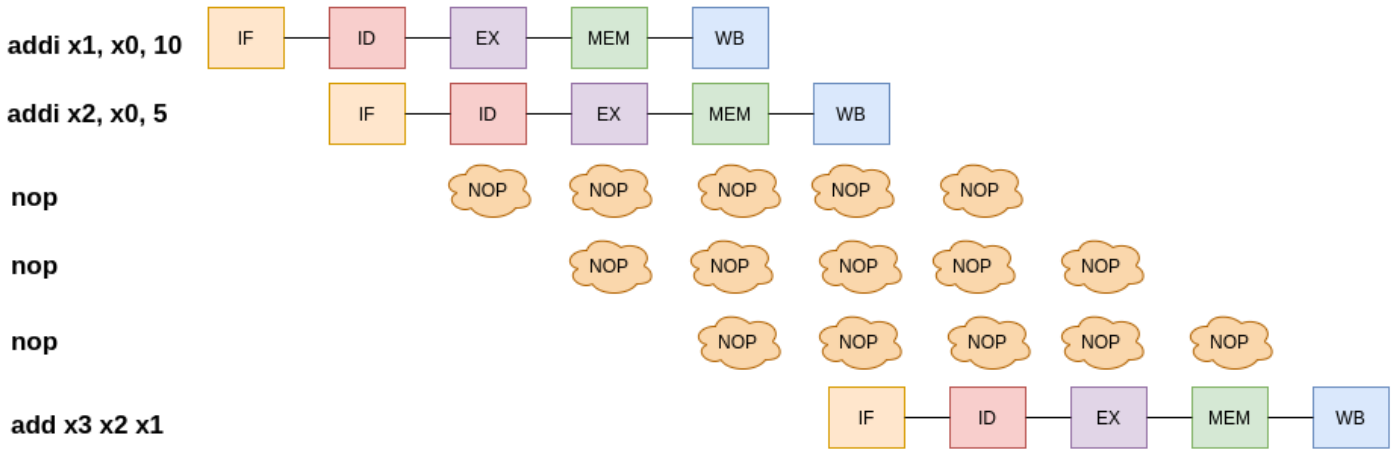


FIGURE 32 : Handling Data Hazards

4.2. Control Hazards

Control hazards occur when there are branches. The branch is decided in the execution stage and there will be 2 instructions fetched and if the branch is not taken then there will be no errors. But if the branch is taken then the fetched 2 instructions have to be discarded. Since branch prediction is not yet implemented, 2 bubbles have to be included in the assembly code to avoid these types of hazards. Following figure shows the addition of the nop instructions after the branch instruction.

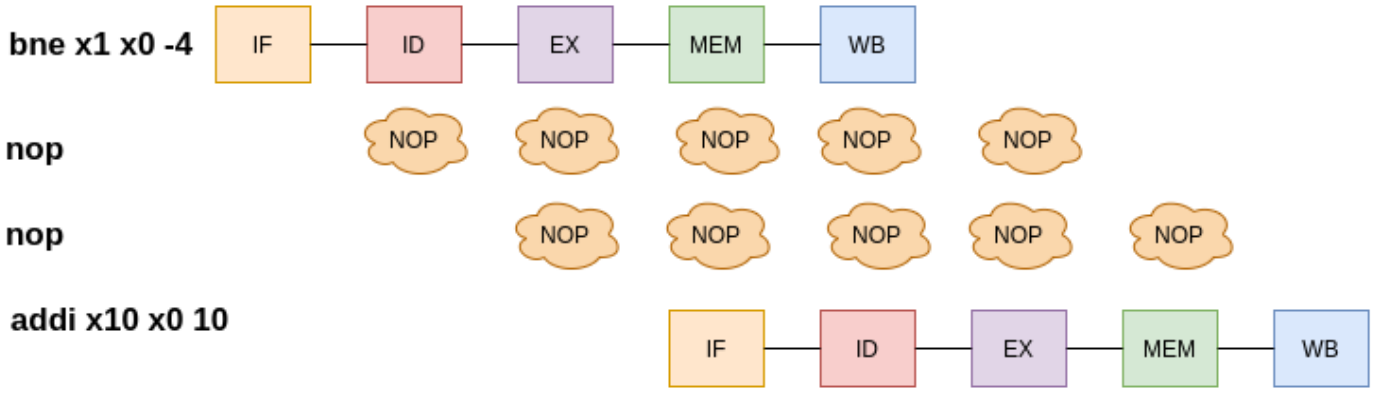


FIGURE 33 : Handling Control Hazards

5. Integrated CPU

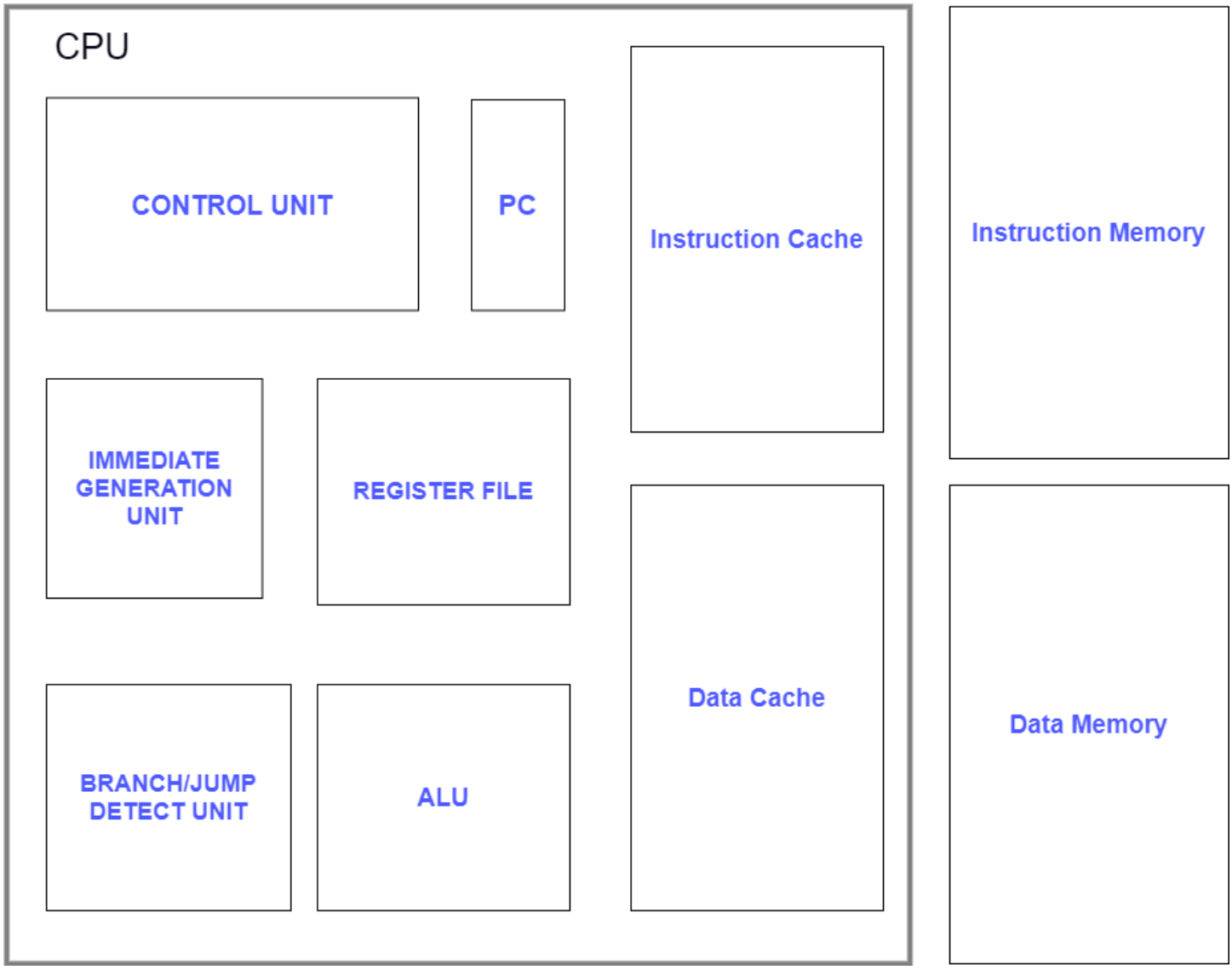


FIGURE 34 : Integrated CPU

6. Testing

6.1. Testing individual modules

Testbenches to test individual hardware units were implemented and all the tests were automated. Scripts to test the individual hardware units are in the /cpu/scripts directory.

NOTE : Make sure to set execute permission for test scripts if needed.

6.2. Testing overall CPU

There are several programs defined in the /cpu directory (eg: program.s, program1.s etc.) To test an assembly program, add the assembly program to the /cpu directory and run test-program.sh script with the program file as an argument.

Eg: ./test-program.sh program.s

The test script will create a file named cpu_pipeline_wavedata.vcd in /cpu/cpu_pipeline directory and this file can be opened in gtkwave to observe the signal changes.

IMPORTANT!
iverilog version 11 was used to implement the pipeline CPU. Using an iverilog version lower than 11 will cause unexpected results. To see the correct functioning of the CPU you have to have iverilog version 11 installed.
To check the iverilog version use the “iverilog -v” command.
If you don’t have verilog v11 installed, you can download it from here. (for linux users only)