CO502 - Advanced Computer Architecture
Part 02 (Hardware Units)
Group 04

Group Members

E/18/077 - Nipun Dharmarathne
E/18/397 - Shamod Wijerathne
E/18/402 - Chathura Wimalasiri

Instruction types and their encoding formats

There are 6 types of instructions in RISC-V architecture:

1. R-type (register-type) instructions: These instructions operate on registers and have three register operands. Examples include ADD, SUB, AND, OR, XOR, SLL, SRL, and SRA.

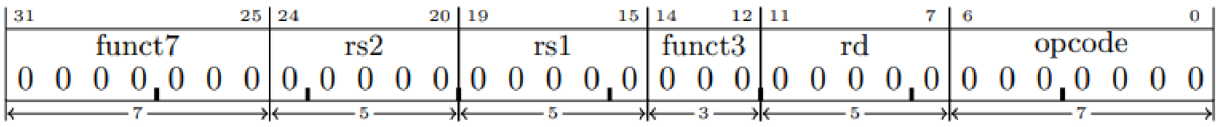


Figure 01: R-type instruction encoding format

Here, the funct7, funct3, and opcode fields identify the specific operation, while the rs1, rs2, and rd fields specify the source and destination registers.

The instruction performs the operation specified by funct3 on the values stored in the two source registers rs1 and rs2. The result of the operation is stored in the destination register rd. The specific operation performed by the instruction depends on the function code funct3.

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

2. I-type (immediate-type) instructions: These instructions have one immediate value and one register operand. Examples include ADDI, SLTI, ANDI, ORI, XORI, SLLI, SRLI, and SRAI.

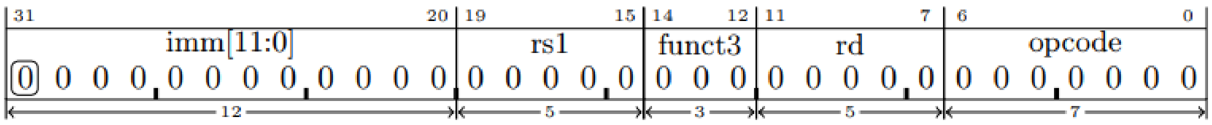


Figure 02: I-type instruction encoding format

Here, imm is a 12-bit immediate value, rs1 is the source register, funct3 is a function code that identifies the specific I-type instruction, rd is the destination register where the result is stored, and opcode is the operation code that identifies the type of instruction.

The instruction performs the operation specified by funct3 on the value stored in the source register rs1 and the immediate value imm. The result of the operation is stored in the destination register rd. The specific operation performed by the instruction depends on the function code funct3.

imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
imm[11:0]		rs1	000	rd	1100111	JALR

3. S-type (store-type) instructions: These instructions store a value from a register into memory with an offset. Examples include SB, SH, and SW.

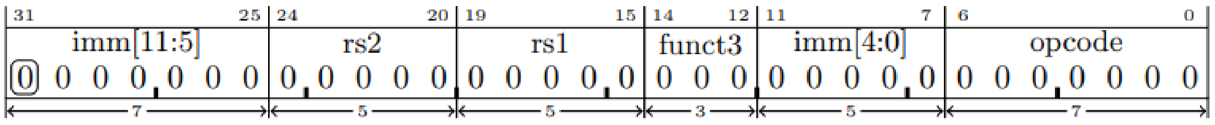


Figure 03: S-type instruction encoding format

Here, imm is a 12-bit immediate value used as an offset to access memory, rs1 is the base register used to calculate the memory address, rs2 is the source register whose value needs to be stored in memory, and funct3 identifies the specific store operation.

When executed, the S-type instruction calculates the memory address by adding the 12-bit immediate value imm to the value of register rs1. It then stores the value of register rs2 into the memory location specified by the calculated address. The specific store operation is determined by the funct3 field, which can be used to specify the byte size of the store operation, and can have values such as 000 for storing a byte, 001 for storing a halfword (2 bytes), or 010 for storing a word (4 bytes).

imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

4. U-type (upper immediate) instructions: These instructions load a 20-bit immediate value into a register. Examples include LUI and AUIPC.

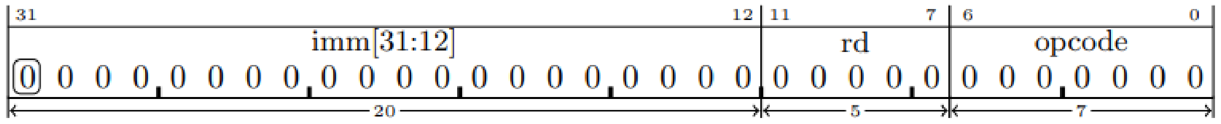


Figure 04: U-type instruction encoding format

Here, imm is a 20-bit immediate value, rd is the destination register where the immediate value is loaded, and opcode is the operation code that identifies the specific U-type instruction.

When executed, the U-type instruction loads the 20-bit immediate value imm into register rd. The immediate value is zero-extended to 32 bits before being stored in the register. Since the immediate value is 20 bits, it cannot represent the entire 32-bit address space. To load a full 32-bit address, the U-type instruction is typically combined with a lower 12-bit immediate value specified in an I-type instruction, using a special encoding called the UJ-type (unconditional jump-type) instruction.

imm[31:12]	rd	0110111	LUI
imm[31:12]	rd	0010111	AUIPC

5. B-type (branch-type) instructions: These instructions perform conditional branches based on the comparison of two register values. Examples include BEQ, BNE, BLT, BGE, BLTU, and BGEU.

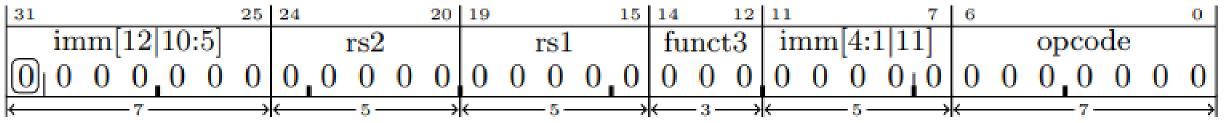
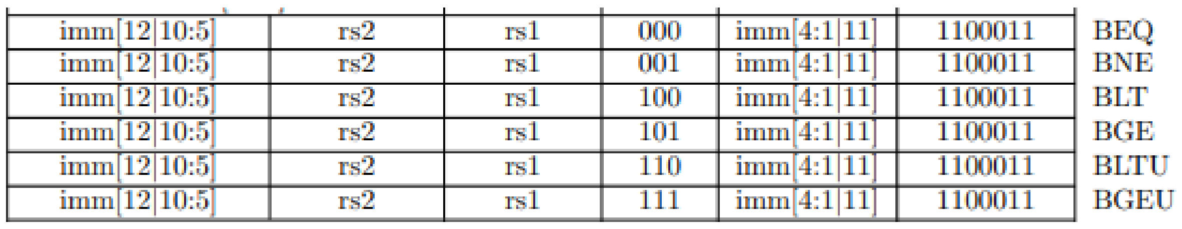


Figure 05: B-type instruction encoding format

Here, rs1 and rs2 are the two source registers, funct3 is a function code that identifies the specific B-type instruction, opcode is the operation code that identifies the type of instruction, and imm is a 13-bit immediate value that encodes the branch offset.

The immediate value is constructed from three fields: the 1-bit imm[12] field, which is used to construct the top bit of the 13-bit immediate value; the 6-bit imm[10:5] field, which is used to construct bits 4-9 of the 13-bit immediate value; and the 4-bit imm[4:1|11] field, which is used to construct bits 0-3 and 12 of the 13-bit immediate value.
The rs1 and rs2 registers are compared based on the specific function code funct3. If the comparison is true, then the program counter is updated to the target address computed from the relative branch offset. If the comparison is false, the program counter is incremented to the next sequential instruction.



imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

6. J-type (jump-type) instructions: These instructions perform unconditional jumps and jump with a link (i.e., save the address of the next instruction to be executed). Examples include JAL and JALR.

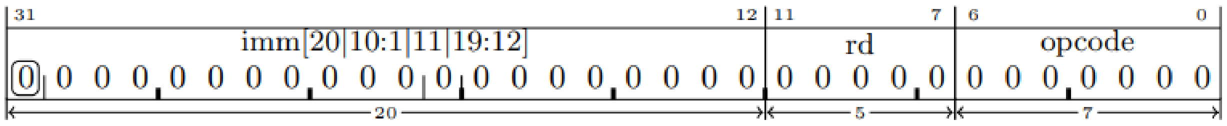
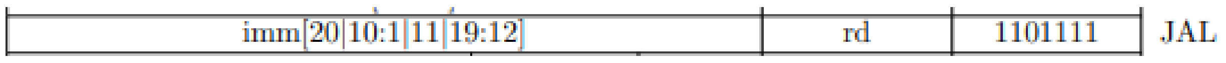


Figure 06: J-type instruction encoding format

Here, rd is the destination register, which is used to store the return address of the jump instruction. The immediate value is added to the current program counter (PC) value to compute the target address. The immediate value is then shifted left by 1 bit to adjust for the fact that RISC-V instructions are 32 bits long and the target address must be aligned to a 32-bit boundary.

After the jump instruction is executed, the PC is set to the target address computed from the immediate value, with the least-significant bit set to zero to ensure that the target address is aligned to a 32-bit boundary. The rd register is used to store the return address of the jump instruction, which is the address of the instruction following the jump instruction.



imm[20|10:1|11|19:12]	rd	1101111	JAL

Instructions and their opcodes

R-Type instructions

Instruction	Description	Opcode	func3	func7
ADD	ADD the value in rs1 and rs2 value and put it into rd	0110011	000	0000000
SUB	SUBTRACT the value in rs1 and rs2 value and put it into rd	0110011	000	0100000
SLL	Shift Left Logical (rs1 value by rs2 amount and put it to rd)	0110011	001	0000000
SLT	Set less than (if rs1 value less than rs2 value then put 1 to the rd register else 0 to the rd register)	0110011	010	0000000
SLTU	Set less than Unsigned (if rs1 value less than rs2 value then put 1 to the rd register else 0 to the rd register)	0110011	011	0000000
XOR	XOR the value in rs1 and rs2 value and put it into rd	0110011	100	0000000
SRL	Shift Right Logical (rs1 value by rs2 amount and put it to rd)	0110011	101	0000000
SRA	Shift Right Arithmetic (rs1 value by rs2 amount and put it to rd)	0110011	101	0100000
OR	OR the value in rs1 and rs2 value and put it into rd	0110011	110	0000000
AND	AND the value in rs1 and rs2 value and put it into rd	0110011	111	0000000

MUL	Multiplication	0110011	000	0000001
MULH	Returns upper 32-bits of signed x signed (use rs1 value and the rs2 value and put the answer to the rd register)	0110011	001	0000001
MULHSU	Returns upper 32-bits of signed x unsigned (use rs1 value and the rs2 value and put the answer to the rd register)	0110011	010	0000001
MULHU	Returns upper 32-bits of unsigned x unsigned (use rs1 value and the rs2 value and put the answer to the rd register)	0110011	011	0000001
DIV	Signed Integer division (use rs1 value and the rs2 value and put the answer to the rd register)	0110011	100	0000001
DIVU	Unsigned Integer division (use rs1 value and the rs2 value and put the answer to the rd register)	0110011	101	0000001
REM	Signed remainder of integer division (use rs1 value and the rs2 value and put the answer to the rd register)	0110011	110	0000001
REMU	Unsigned remainder of integer division (use rs1 value and the rs2 value and put the answer to the rd register)	0110011	111	0000001

I-Type instructions

Instruction	**Description**	**Opcode**	**func3**	**func7**
LB	Load Byte to rd register (signed extended)	0000011	000	
LH	Load 2 Bytes to rd register (signed extended)	0000011	001	
LW	Load Word to rd register (signed extended)	0000011	010	
LBU	Load Byte to rd register (zero extended)	0000011	100	
LHU	Load 2 Bytes to rd register (zero extended)	0000011	101	
ADDI	ADD immediate value and value of rs1 and put result to rd register	0010011	000	
SLLI	Shift Left Logical with Immediate (shift rs1 value by immediate amount)	0010011	001	0000000
SLTI	if immediate value is less than value of rs1 put 1 to rd register otherwise put 0	0010011	010	
SLTIU	if immediate value is less than value of rs1 put 1 to rd register otherwise put 0 (unsigned)	0010011	011	
XORI	XOR immediate value and value of rs1 and put result to rd register	0010011	100	
SRLI	Shift Right Logical with Immediate (shift rs1 value by immediate amount)	0010011	101	0000000
SRAI	Shift Right Arithmetic Immediate (shift rs1 value by immediate amount)	0010011	101	0100000
ORI	OR immediate value and value of rs1 and put result to rd register	0010011	110	
ANDI	AND immediate value and value of rs1 and put result to rd register	0010011	111	
JALR	Jump and Link Register	1100111		

S-Type instructions

Instruction	**Description**	**Opcode**	**func3**	**func7**
SB	Store Byte rs2 reg value, base is in rs1 address and the offset taken from the immediate value	0100011	000	
SH	rs2 reg value, base is in rs1 address and the offset taken from the immediate value	0100011	001	
SW	rs2 reg value, base is in rs1 address and the offset taken from the immediate value	0100011	010	
SBU	Store unsigned Byte	0100011	100	
SHU	Store unsigned half word	0100011	101	

U-Type instructions

Instruction	**Description**	**Opcode**	**func3**	**func7**
AUIPC	Add upper immediate to PC and put to rd register	0010011		
LUI	Load Upper Immediate (puts the immediate value with 12 zeros at the end and put it into rd register)	0110111		

B-Type instructions

Instruction	**Description**	**Opcode**	**func3**	**func7**
BEQ	Branch if equal (if values in rs1 and rs2 are equal jump to the offset)	1100011	000	
BNE	Branch if not equal (if values in rs1 and rs2 are not equal jump to the offset)	1100011	001	
BLT	Branch if lower than (if values in rs1 < rs2 jump to the offset)	1100011	100	
BGE	Branch if greater than or equal (if values in rs1 > rs2 jump to the offset)	1100011	101	
BLTU	Branch if lower than, unsigned (if values in rs1 < rs2 jump to the offset)	1100011	110	
BGEU	Branch greater than or equal, unsigned (if values in rs1 > rs2 jump to the offset)	1100011	111	

J-Type instructions and other instructions

Instruction	**Description**	**Opcode**	**func3**	**func7**
JAL	Jump and Link (Jumps to the address in rs1 and put the current PC to the rd register)	1101111		
FENCE	Fence - This to ensure all the operation before FENCE observed before operation after the Fence	0001111	000	
FENCE.I	Fence Instruction	0001111	001	

Pipeline diagram with datapath and control

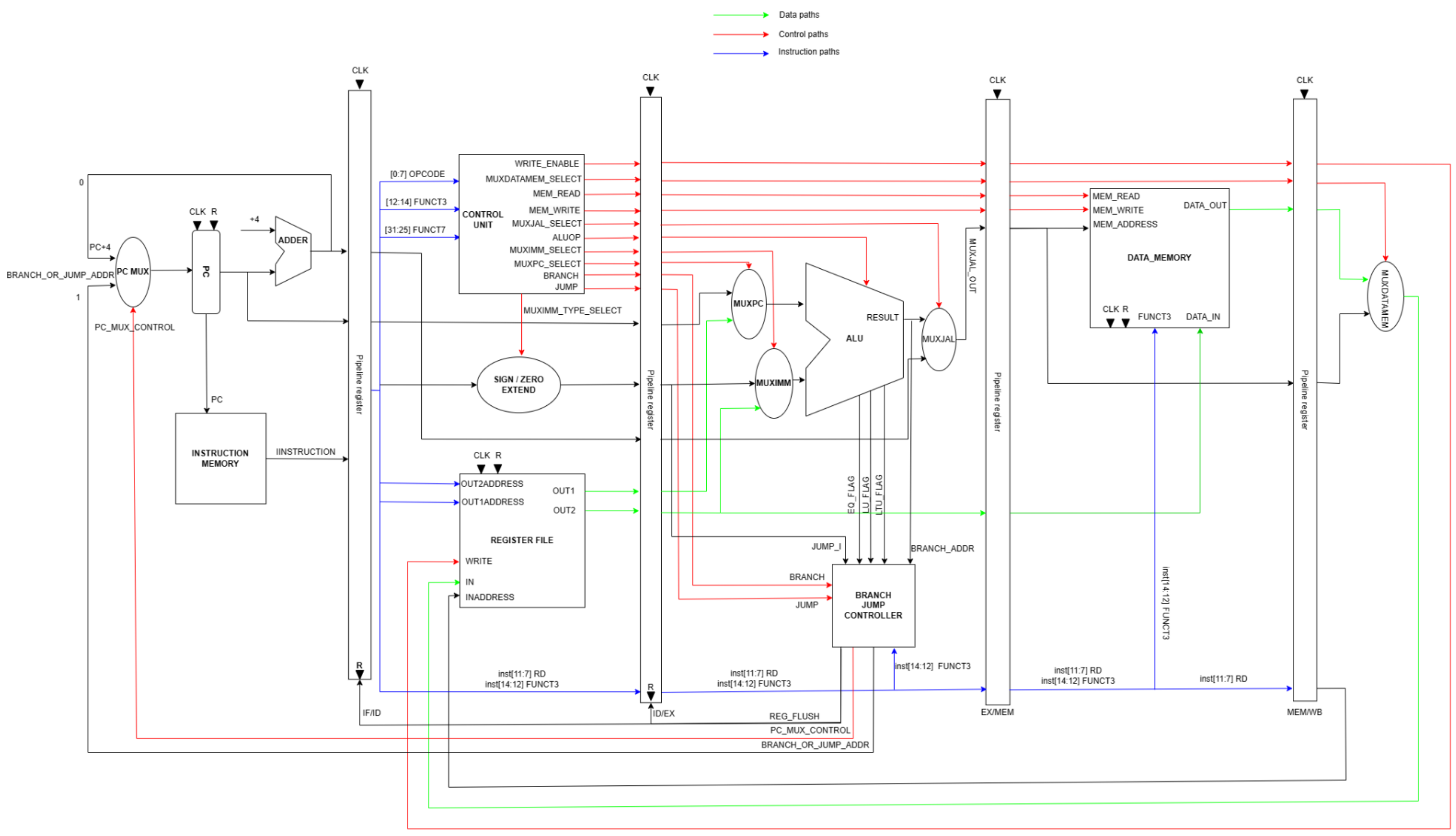


Figure 07: Pipeline diagram with datapath and control

In case of not clear enough to observe above FIGURE 07 - Pipeline diagram with datapath and control path

Main units of the processor

1. Instruction Memory

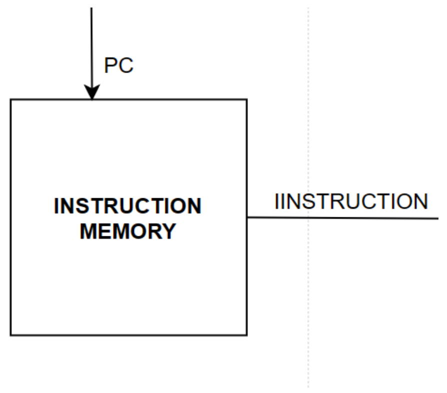


Figure 08: Instruction Memory

Inputs

1. PC - the memory address of the next instruction to be executed in a computer program.

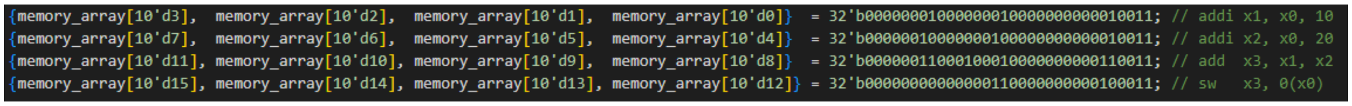
Outputs

1. INSTRUCTION - 32-bit Instruction

Design Decision

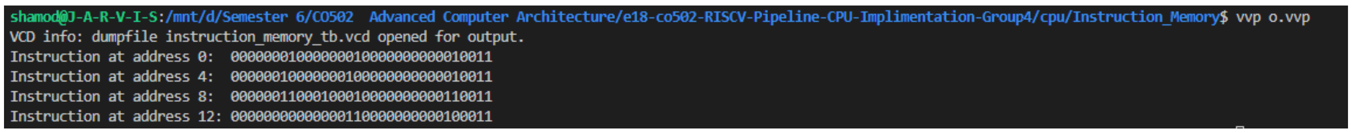
So far implemented 1024*8 bits memory. Which can hold 256 instructions. If necessary we can increase the size of the instruction memory.

Instructions were hardcoded in this instruction memory for the testing purpose,



```
{memory_array[10'd3],  memory_array[10'd2],  memory_array[10'd1],  memory_array[10'd0]}  = 32'b00000001000000001000000000010011; // addi x1, x0, 10
{memory_array[10'd7],  memory_array[10'd6],  memory_array[10'd5],  memory_array[10'd4]}  = 32'b00000010000000010000000000010011; // addi x2, x0, 20
{memory_array[10'd11], memory_array[10'd10], memory_array[10'd9],  memory_array[10'd8]}  = 32'b00000011000100010000000000110011; // add  x3, x1, x2
{memory_array[10'd15], memory_array[10'd14], memory_array[10'd13], memory_array[10'd12]} = 32'b00000000000000011000000000100011; // sw   x3, 0(x0)
```

Result of the instruction testbed,



```
shamod@J-A-R-V-I-S:/mnt/d/Semester 6/CO502  Advanced Computer Architecture/e18-co502-RISCV-Pipeline-CPU-Implimentation-Group4/cpu/Instruction_Memory$ vvp o.vvp
VCD info: dumpfile instruction_memory_tb.vcd opened for output.
Instruction at address 0:  00000001000000001000000000010011
Instruction at address 4:  00000010000000010000000000010011
Instruction at address 8:  00000011000100010000000000110011
Instruction at address 12: 00000000000000011000000000100011
```

2. Control Unit

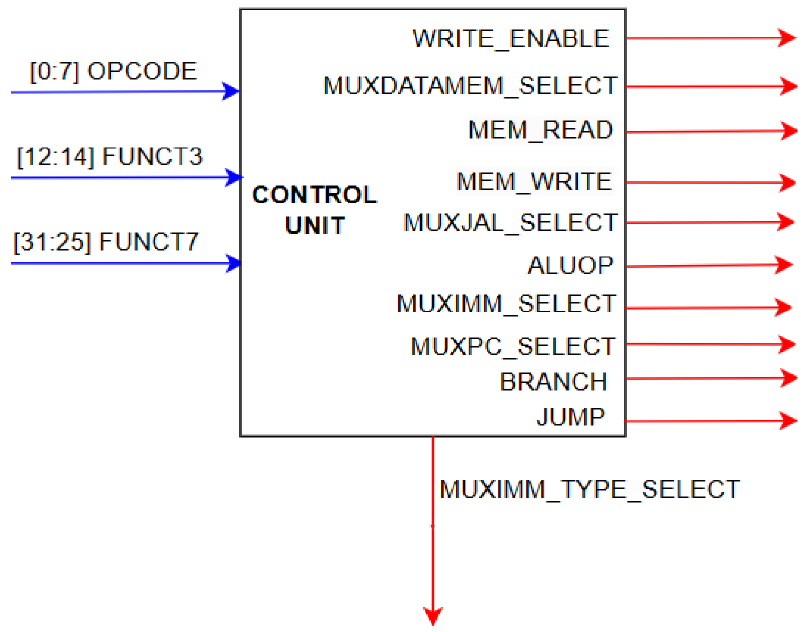


Figure 09: Control Unit

Inputs

1. OPCODE

   An OPCODE, short for "operation code," is a binary code that specifies the operation to be performed by a processor or microcontroller in order to execute a particular instruction.

2. FUNCT3

   FUNCT3 (short for function 3) is a field in the instruction format that specifies a subset of operations that can be performed by an instruction.

3. FUNCT7

   FUNCT7 (short for function 7) is a field in the instruction format that provides additional information about the operation to be performed by an instruction.

Outputs - Control signals

1. WRITE_ENABLE:

   1 bit signal. This is the control signal that determines whether data can be written into the register.

   When,

WRITE_ENABLE=1 → Enables the write operation to the register file
WRITE_ENABLE=0 → Disables the write operation to the register file

2. MUXDATAMEM_SELECT:

   1 bit signal. This is the select signal for MUXDATAMEM which is used to select between MUXJAL_OUT and data memory output (DATA_OUT). Then the selected data is directed to the register file.

   When,
   MUXDATAMEM_SELECT=1 → DATA_OUT
   MUXDATAMEM_SELECT=0 → MUXJAL_OUT

3. MEM_READ:

   1 bit signal. This is the control signal that enables the output data from the memory unit. When the MEM_READ signal is high, the memory unit is instructed to output data (DATA_OUT) from the specified address, which is provided on the address lines (MEM_ADDRESS).

   When,
   MEM_READ=1 → Enables the read operation from the data memory
   MEM_READ=0 → Disables the read operation from the data memory

4. MEM_WRITE:

   1 bit signal. This is the control signal that enables the input data to be written into the memory unit. When the MEM_WRITE signal is high, the memory unit is instructed to write the data provided on the data lines (DATA_IN) to the specified address on the address lines (MEM_ADDRESS).

   When,
   MEM_WRITE=1 → Enables the write operation to the data memory
   MEM_WRITE=0 → Disables the write operation to the data memory

5. MUXJAL_SELECT:

   1 bit signal. This is the select signal for the MUXJAL multiplexer which is used to select between ALU_RESULT and current PC+4 value.

   When,
   MUXJAL_SELECT=1 → PC+4
   MUXJAL_SELECT=0 → ALU_RESULT

6. ALUOP:

   5-bit signal for ALU. This signal specifies the arithmetic or logical operation to be performed by the ALU. It selects one of several operations that the ALU can perform, such as addition, subtraction, multiplication, bitwise AND, OR, XOR, and shift operations.

7. MUXPC_SELECT:

   1 bit signal. This is the select signal for MUXPC, which is used to select between register file output (REGOUT1) and PC value. Then the selected data is directed to the ALU for its operations.

   When,
   MUXPC_SELECT=1 → PC
   MUXPC_SELECT=0 → REGOUT1

8. MUXIMM_SELECT:

   1 bit signal. This is the select signal for MUXIMM, which is used to select between immediate value and register file output (REGOUT2). Then the selected data is directed to the ALU for its operations.

   When,
   MUXIMM_SELECT=1 → IMMEDIATE VALUE
   MUXIMM_SELECT=0 → REGOUT2

9. MUXIMMTYPE_SELECT:

   3 bit signal. This is the select signal for MUXIMMTYPE, which is used to select between I_IMM, S_IMM, B_IMM, U_IMM, J_IMM. Then the selected data is directed to the MUXIMM.

   When,
   MUXIMMTYPE_SELECT=000 → I_IMM
   MUXIMMTYPE_SELECT=001 → S_IMM
   MUXIMMTYPE_SELECT=010 → U_IMM
   MUXIMMTYPE_SELECT=011 → B_IMM
   MUXIMMTYPE_SELECT=100 → J_IMM

10. BRANCH and JUMP:

    1-bit signals. These signals are used in the BRANCH JUMP CONTROLLER in order to distinguish whether the instruction is a branch or a jump. The BRANCH signal is used to conditionally execute a set of instructions based on a particular condition. The JUMP signal is used to unconditionally jump to a specific instruction address.

When,

BRANCH=1 → Enables branch operation
BRANCH=0 → Disables branch operation
JUMP=1 → Enables jump operation
JUMP=0 → Disables jump operation

Note:

- Though the above diagram displays the parts of the decoded instruction (OPCODE, FUNCT3, FUNCT7) which are used in the control unit as inputs to determine the control signals; in our implementation, we have done the decoding inside the control unit module. Hence, we have used the whole 32-bit INSTRUCTION as input to the control unit.
- All the control signals of the control unit are indicated in red in the datapath and control diagram.

Design decisions

As the first step we have decoded the instruction and extracted the OPCODE, FUNCT3 and FUNCT7.

We can observe that, for the instruction set we have considered, we only need the 0th and 5th bits of FUNCT7. But considering further expansion of our instruction set, we have used all the 7-bits of FUNCT7 in our implementation.

When we consider the R type instructions their OPCODEs are the same. They are differentiated using FUNCT3 and FUNCT7. All the control signals are the same for these instructions except for the ALUOP. Since, we have used a nested case structure to implement this. We have defined all the control signals except ALUOP in the outer case as they are the same for all R type instructions and ALUOP was defined for each instruction in the inner case considering FUNCT3 and FUNCT7. This will make the code much simpler and easy to understand. The same thing was done for I type instructions as well.

For load and store instructions the control unit generates the same control signals, whether the instruction is a full word, half word or a byte. These instructions are controlled in the memory access stage using the FUNCT3 field.

The below table consists of all the control signals for each instruction,

Control logic truth table (Considering both control unit and branch/jump controller unit)

INSTRUCTION	ALUOP	EQ_FLAG	LT_FLAG	LTU_FLAG	PC MUX_SEL	MUX IMM_TYPE_SEL	MUX PC_SEL	MUX IMM_SEL	MUX JAL_SEL	MUXDATA MEM_SEL	WRITE_ENABLE	MEM_READ	MEM_WRITE	BRANCH	JUMP
ADD	ADD	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0

SUB	SUB	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0
SLL	SLL	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0
SLT	SLT	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0
SLTU	SLTU	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0
XOR	XOR	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0
SRL	SRL	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0
SRA	SRA	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0
OR	OR	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0
AND	AND	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0
MUL	MUL	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0
MULH	MULH	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0
MULHSU	MULHSU	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0
MULHU	MULHU	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0
DIV	DIV	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0
DIVU	DIVU	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0
REM	REM	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0
REMU	REMU	*	*	*	+4	*	Reg	Reg	ALU	MUXJAL	1	0	0	0	0
LB	ADD	*	*	*	+4	I_Imm	Reg	Imm	ALU	Mem	1	1	0	0	0
LH	ADD	*	*	*	+4	I_Imm	Reg	Imm	ALU	Mem	1	1	0	0	0
LW	ADD	*	*	*	+4	I_Imm	Reg	Imm	ALU	Mem	1	1	0	0	0
LBU	ADD	*	*	*	+4	I_Imm	Reg	Imm	ALU	Mem	1	1	0	0	0
LHU	ADD	*	*	*	+4	I_Imm	Reg	Imm	ALU	Mem	1	1	0	0	0
ADDI	ADD	*	*	*	+4	I_Imm	Reg	Imm	ALU	MUXJAL	1	0	0	0	0
SLLI	SLL	*	*	*	+4	I_Imm	Reg	Imm	ALU	MUXJAL	1	0	0	0	0
SLTI	SLT	*	*	*	+4	I_Imm	Reg	Imm	ALU	MUXJAL	1	0	0	0	0
SLTIU	SLTU	*	*	*	+4	I_Imm	Reg	Imm	ALU	MUXJAL	1	0	0	0	0
XORI	XOR	*	*	*	+4	I_Imm	Reg	Imm	ALU	MUXJAL	1	0	0	0	0
SRLI	SRL	*	*	*	+4	I_Imm	Reg	Imm	ALU	MUXJAL	1	0	0	0	0
ORI	OR	*	*	*	+4	I_Imm	Reg	Imm	ALU	MUXJAL	1	0	0	0	0
ANDI	AND	*	*	*	+4	I_Imm	Reg	Imm	ALU	MUXJAL	1	0	0	0	0
JALR	ADD	*	*	*	ALU	I_Imm	Reg	Imm	PC+4	MUXJAL	1	0	0	0	1
SB	ADD	*	*	*	+4	S_Imm	Reg	Imm	ALU	*	0	0	1	0	0
SH	ADD	*	*	*	+4	S_Imm	Reg	Imm	ALU	*	0	0	1	0	0
SW	ADD	*	*	*	+4	S_Imm	Reg	Imm	ALU	*	0	0	1	0	0

SBU	ADD	*	*	*	+4	S_Imm	Reg	Imm	ALU	*	0	0	1	0	0
SHU	ADD	*	*	*	+4	S_Imm	Reg	Imm	ALU	*	0	0	1	0	0
AUIPC	ADD	*	*	*	+4	U_Imm	PC	Imm	ALU	MUXJAL	1	0	0	0	0
LUI	FORWARD	*	*	*	+4	U_Imm	*	Imm	ALU	MUXJAL	1	0	0	0	0
BEQ	SUB	0	*	*	+4	B_Imm	PC	Imm	ALU	*	0	0	0	1	0
BEQ	SUB	1	*	*	ALU	B_Imm	PC	Imm	ALU	*	0	0	0	1	0
BNE	SUB	0	*	*	ALU	B_Imm	PC	Imm	ALU	*	0	0	0	1	0
BNE	SUB	1	*	*	+4	B_Imm	PC	Imm	ALU	*	0	0	0	1	0
BLT	SUB	*	1	*	ALU	B_Imm	PC	Imm	ALU	*	0	0	0	1	0
BGE	SUB	*	0	*	ALU	B_Imm	PC	Imm	ALU	*	0	0	0	1	0
BLTU	SUB	*	*	1	ALU	B_Imm	PC	Imm	ALU	*	0	0	0	1	0
BGEU	SUB	*	*	0	ALU	B_Imm	PC	Imm	ALU	*	0	0	0	1	0
JAL	ADD	*	*	*	ALU	J_Imm	PC	Imm	PC+4	MUXJAL	1	0	0	0	1

Note: Above table is colored by considering the type of instruction.

Timing

As the first step we have decoded the instruction and extracted the OPCODE, FUNCT3 and FUNCT7. For this we didn’t add a delay. As it is just an asynchronous wire operation, the time delay for this is very low.

For all instructions we have introduced #1 time unit delay for generating control signals to represent combinational logic delays.

Description	Delay
Combinational logic delays	#1

3. Register FIle

The purpose of the register file is to store output values generated by the ALU, and to supply the ALU's inputs with operands.

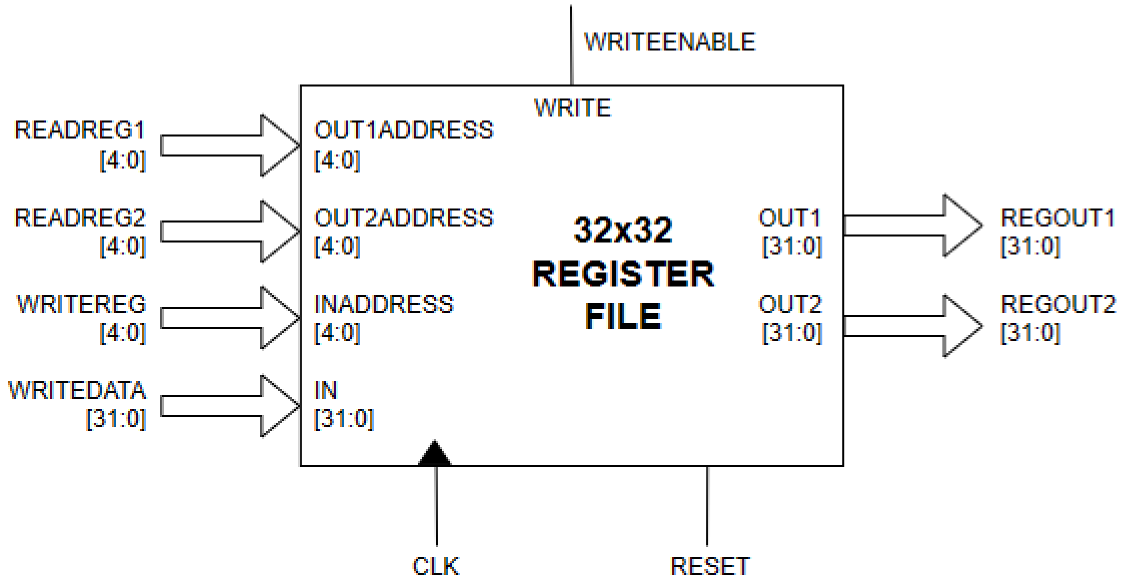


Figure 10: Register File

Inputs

1. CLK

   In a register file, the clock signal is used to trigger the read or write operations to the registers.

2. RESET

   The RESET signal is a read control signal that is used to enable a read operation from the register file.

3. OUT1ADDRESS and OUT2ADDRESS

   These are address signals that are used to specify the register addresses for a read operation.

4. WRITE

   This is used to enable a write operation to the register file.

5. IN

   This is used to specify the data value to be written to a register in the register file during a write operation.

6. INADDRESS

   This is used to specify the address of the register to be written during a write operation.

<u>Outputs</u>

1. OUT1 and OUT2

   OUT1 and OUT2 are typically used in the context of an arithmetic or logic operation in a computer system. They represent the two data operands that are being used in the operation.

<u>Design decisions</u>

- 32x32 register file (Can store thirty two 32-bit values, register0 - register31).
- Contains one 32-bit data input port (IN) and two 32-bit output ports (OUT1 and OUT2).
- Three 5-bit address ports (INADDRESS, OUT1ADDRESS, OUT2ADDRESS) are included to specify which register is reading or writing.
- Control port WRITE is used to accommodate the WRITEENABLE control signal.
- As the register file is a sequential unit, CLOCK and RESET signals are used for synchronization.
- The port IN represents the data input, with INADDRESS providing the register number to store data in. The ports OUT1 and OUT2 are parallel data outputs, where OUT1ADDRESS and OUT2ADDRESS respectively provide the register numbers where data is retrieved from.

Read operation

- Registers identified by OUT1ADDRESS and OUT2ADDRESS are used to read asynchronously and the values are loaded to OUT1 and OUT2 respectively.

Write operation

- Writing to the register file is done synchronously.
- When the WRITEENABLE signal at the write port is set high, the rising edge of CLOCK makes the data present on the IN port to be written to the input register specified by the INADDRESS.
- Register file reset happens synchronously at the positive edge of the clock if the RESET signal is high. In a reset event, all registers are cleared (written zero).

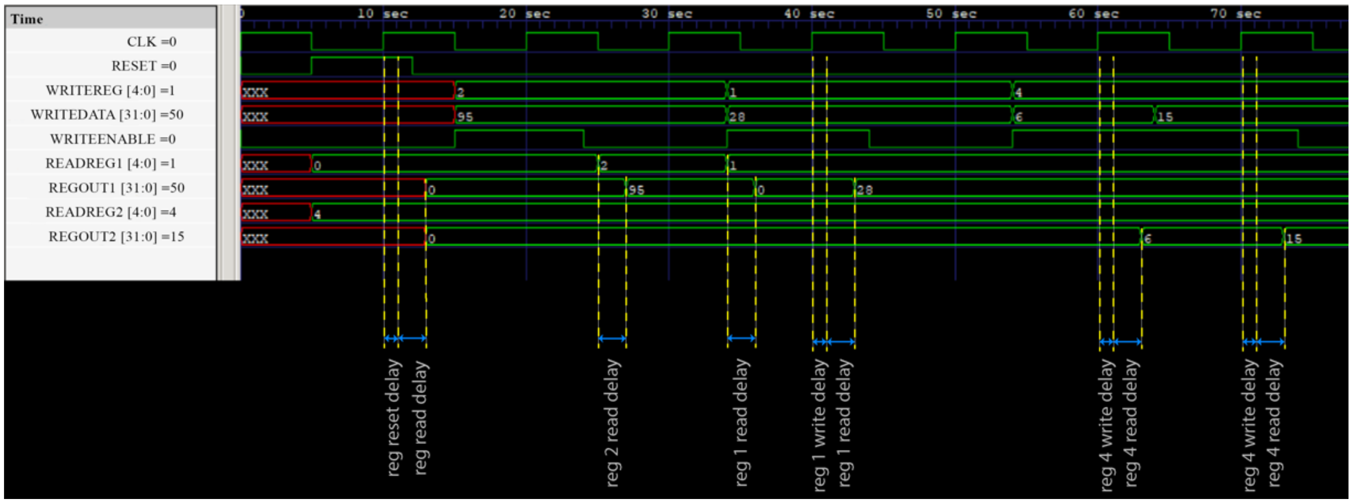
.

Timing

To simulate the register file read and write latencies realistically, artificial delays of two time units (#2) for register reading and one time unit (#1) for writing operations including reset were added.

Description	Delay
Read operation	#2
Write operation	#1
Reset	#1

Timing diagram for register file testbench generated using GTKWave,



4. ALU

This is the part of the computer processor which performs arithmetic and logic operations on numbers, e.g. addition, subtraction, etc.

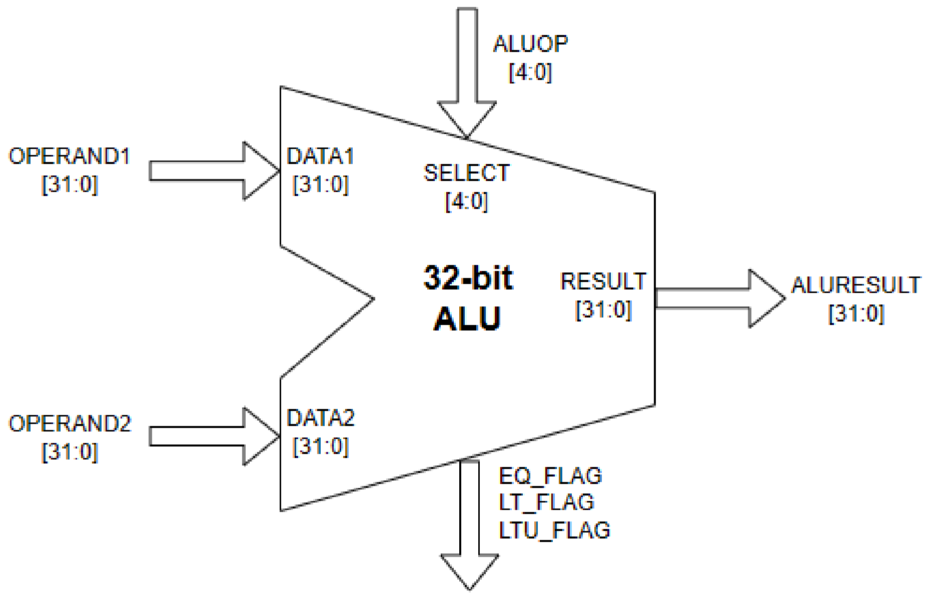


Figure 11: Interfaces of the ALU

Inputs

1. DATA1 and DATA2

   Two 32-bit input ports for operands

2. SELECT

   5-bit control input port which is used to pick the required function inside the ALU. As there are 18 functions in our ALU, 5-bit is the minimum required size for the SELECT input.

Outputs

1. RESULT

   32-bit output port

2. EQ_FLAG (Equal flag)

   1-bit output signal used to notify whether DATA1=DATA2 or not. Implemented using the help of ALU SUB operation.

   When,
   DATA1=DATA2 → EQ_FLAG = 1

DATA1≠DATA2 → EQ_FLAG = 0

3. LT_FLAG (Less than flag)

   1-bit output signal used to notify whether DATA1<DATA2 or not. Implemented using the help of ALU SUB operation.

   When,
   DATA1<DATA2 → LT_FLAG = 1
   DATA1>=DATA2 → LT_FLAG = 0

4. LTU_FLAG (Less than unsigned flag)

   1-bit output signal used to notify whether DATA1<DATA2 or not for unsigned data. Implemented using the help of ALU SLTU operation. Any bit of the RESULT of SLTU operation can be considered (In our implementation, we have considered the zeroth bit of the RESULT of SLTU operation).

   When,
   unsigned(DATA1) < unsigned(DATA2) → LTU_FLAG = 1
   unsigned(DATA1) >= unsigned(DATA2) → LTU_FLAG = 0

Design decisions

- 32-bit ALU. Hence it works with 32-bit operands. This means that all data inputs and outputs of the ALU are 32-bit values.
- Contains two 32-bit input ports for operands (DATA1 and DATA2), one 32-bit output port (RESULT) and one 5-bit control port (SELECT) which is used to pick the required function inside the ALU out of the available 19 functions based on the instructions.
- As there are 19 functions in our ALU, 5-bit is used for the SELECT input which is the minimum number of bits required.
- This is a fully asynchronous module.
- RISC-V uses a reduced instruction set compared to other architectures, which means that the ALU only needs to support a smaller set of instructions. This simplifies the design and reduces the hardware resources required.
- Support for multiplication and division operations, which are essential for many applications.
- The ALU generates EQ_FLAG, LT_FLAG and LTU_FLAG control signals. These flags are used by the branch and jump controller unit to determine the next instruction to be executed.
- The ALU works as follows for each type of instruction,
  - R, I type instructions: Performs the relevant operation for each instruction (Ex: ADD, SUB)
  - Load and store instructions: Calculates the data memory address using ADD operation
  - Branch instructions: Performs the SUB operation and sets the EQ_FLAG, LT_FLAG and LTU_FLAG accordingly.

    - Jump instructions: Calculates the jump address using ADD operation

Limitations

- The ALU only supports 32-bit inputs and outputs.
- Limited precision for multiplication and division operations.
- The ALU does not support floating point operations because it is designed to perform arithmetic and logical operations on binary data, which are typically represented using fixed-point or integer data types.

Timing

To simulate the ALU latencies realistically, artificial delays were added to each function as shown in the below table. For addition and subtraction, a delay of two-time units (#2) was added. As the delay for logic operations and shift operations is less compared to ADD and SUB, a delay of one-time unit (#1) was introduced. As multiplication and division are complex functions its delay is high compared to others. So, a delay of three-time units (#3) was added for multiplication and division operations.

The table below shows the 19 functions that our 32-bit ALU can perform with the relevant Unit's delay.

ALU functions

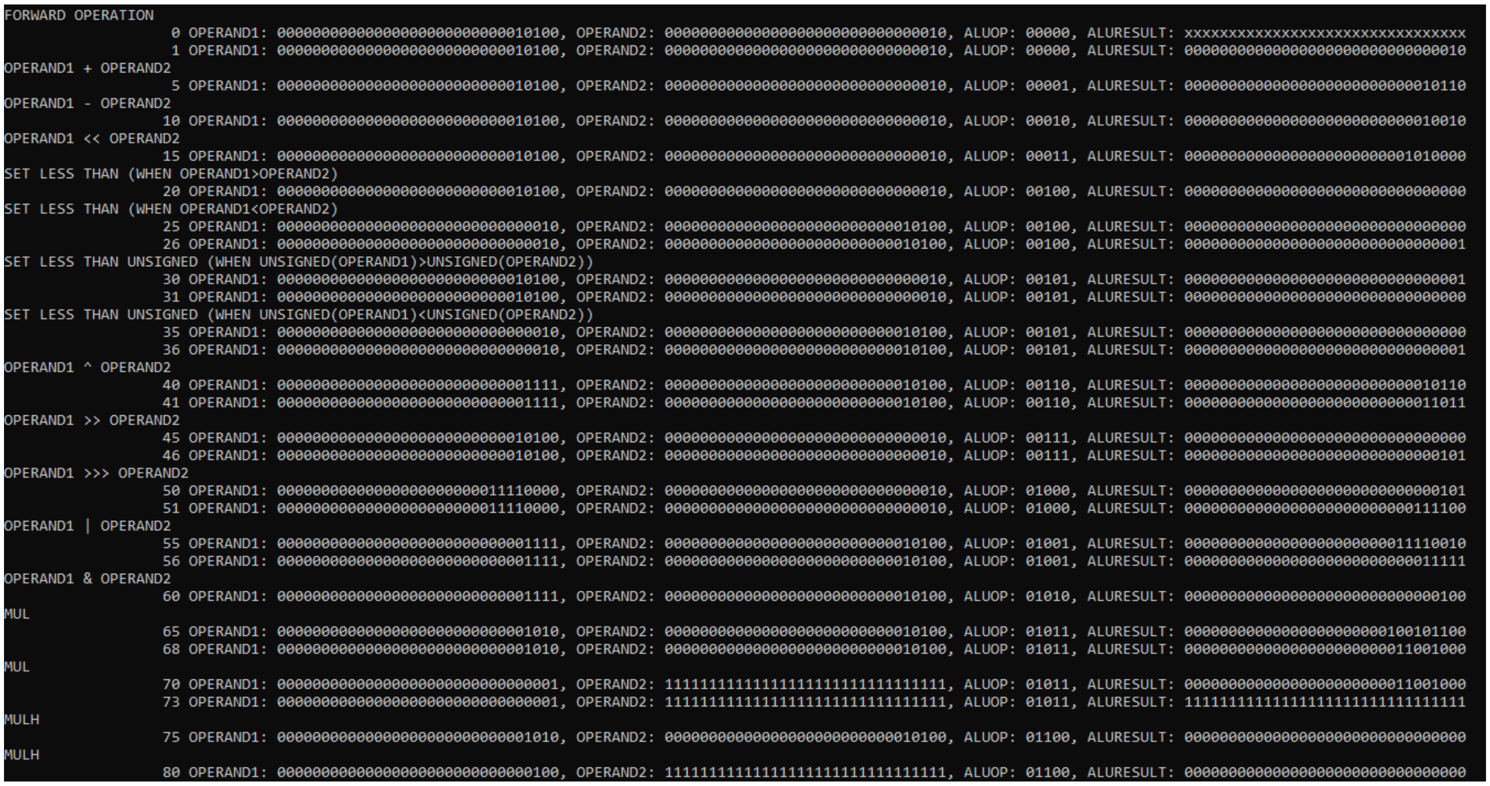
SELECT	Function	Description	Supported Instructions	Unit's Delay
00000	FORWARD	(forward DATA2 into RESULT) DATA2 → RESULT	LUI	#1
00001	ADD	(Add DATA1 and DATA2) DATA1 + DATA2 → RESULT	ADD, LB, LH, LW, LBU, LHU, ADDI, JAL, JALR, SB, SH, SW, SBU, SHU, AUIPC	#2
00010	SUB	(Subtract DATA2 from DATA1) DATA1 - DATA2 → RESULT	SUB, BEQ, BNE, BLT, BGE, BLTU, BGEU	#2
00011	SLL	(Shift left logical) DATA1 << DATA2 → RESULT	SLL, SLLI	#1

00100	SLT	(Set less than) If ($signed(DATA1) < $signed(DATA2)) → RESULT=32’d1 else → RESULT=32’d0	SLT, SLTI	#1
00101	SLTU	(Set less than unsigned) If ($unsigned(DATA1) < $unsigned(DATA2)) → RESULT=32’d1 else → RESULT=32’d0	SLTU, SLTIU	#1
00110	XOR	(Bitwise XOR on DATA1 with DATA2) DATA1 ^ DATA2 → RESULT	XOR, XORI	#1
00111	SRL	(Shift right logical) DATA1 >> DATA2 → RESULT	SRL, SRLI	#1
01000	SRA	(Shift right arithmetic) DATA1 >>> DATA2 → RESULT	SRA	#1
01001	OR	(Bitwise OR on DATA1 with DATA2) DATA1 | DATA2 → RESULT	OR, ORI	#1
01010	AND	(Bitwise AND on DATA1 with DATA2) DATA1 & DATA2 → RESULT	AND, ANDI	#1
01011	MUL	(Multiplication) DATA1 * DATA2 → RESULT	MUL	#3
01100	MULH	(Multiplication-Returns upper 32 bits) (DATA1 * DATA2)>>32 → RESULT	MULH	#3
01101	MULHSU	(Multiplication-Returns upper 32 bits (Signed x UnSigned)) ($signed(DATA1) * $unsigned(DATA2))>>32 → RESULT	MULHSU	#3
01110	MULHU	(Multiplication-Returns upper 32 bits (UnSigned)) ($unsigned(DATA1) * $unsigned(DATA2))>>32 → RESULT	MULHU	#3
01111	DIV	(Division) DATA1 / DATA2 → RESULT	DIV	#3
10000	DIVU	(Division Unsigned) $unsigned(DATA1) / $unsigned(DATA2) → RESULT	DIVU	#3

10001	REM	(Remainder) DATA1 % DATA2 → RESULT	REM	#3
10010	REMU	(Remainder Unsigned) DATA1 % DATA2 → RESULT	REMU	#3

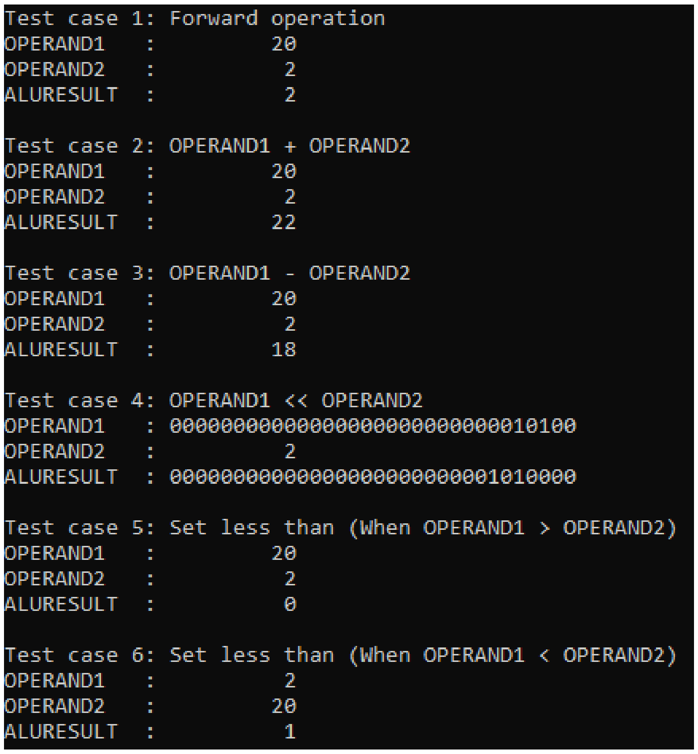
Testing

1. Testing using testbench 1



```
FORWARD OPERATION
                   0 OPERAND1: 00000000000000000000000000010100, OPERAND2: 00000000000000000000000000000010, ALUOP: 00000, ALURESULT: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                   1 OPERAND1: 00000000000000000000000000010100, OPERAND2: 00000000000000000000000000000010, ALUOP: 00000, ALURESULT: 00000000000000000000000000000010
OPERAND1 + OPERAND2
                   5 OPERAND1: 00000000000000000000000000010100, OPERAND2: 00000000000000000000000000000010, ALUOP: 00001, ALURESULT: 00000000000000000000000000010110
OPERAND1 - OPERAND2
                  10 OPERAND1: 00000000000000000000000000010100, OPERAND2: 00000000000000000000000000000010, ALUOP: 00010, ALURESULT: 00000000000000000000000000010010
OPERAND1 << OPERAND2
                  15 OPERAND1: 00000000000000000000000000010100, OPERAND2: 00000000000000000000000000000010, ALUOP: 00011, ALURESULT: 00000000000000000000000001010000
SET LESS THAN (WHEN OPERAND1>OPERAND2)
                  20 OPERAND1: 00000000000000000000000000010100, OPERAND2: 00000000000000000000000000000010, ALUOP: 00100, ALURESULT: 00000000000000000000000000000000
SET LESS THAN (WHEN OPERAND1<OPERAND2)
                  25 OPERAND1: 00000000000000000000000000000010, OPERAND2: 00000000000000000000000000010100, ALUOP: 00100, ALURESULT: 00000000000000000000000000000000
                  26 OPERAND1: 00000000000000000000000000000010, OPERAND2: 00000000000000000000000000010100, ALUOP: 00100, ALURESULT: 00000000000000000000000000000001
SET LESS THAN UNSIGNED (WHEN UNSIGNED(OPERAND1)>UNSIGNED(OPERAND2))
                  30 OPERAND1: 00000000000000000000000000010100, OPERAND2: 00000000000000000000000000000010, ALUOP: 00101, ALURESULT: 00000000000000000000000000000001
                  31 OPERAND1: 00000000000000000000000000010100, OPERAND2: 00000000000000000000000000000010, ALUOP: 00101, ALURESULT: 00000000000000000000000000000000
SET LESS THAN UNSIGNED (WHEN UNSIGNED(OPERAND1)<UNSIGNED(OPERAND2))
                  35 OPERAND1: 00000000000000000000000000000010, OPERAND2: 00000000000000000000000000010100, ALUOP: 00101, ALURESULT: 00000000000000000000000000000000
                  36 OPERAND1: 00000000000000000000000000000010, OPERAND2: 00000000000000000000000000010100, ALUOP: 00101, ALURESULT: 00000000000000000000000000000001
OPERAND1 ^ OPERAND2
                  40 OPERAND1: 00000000000000000000000000001111, OPERAND2: 00000000000000000000000000010100, ALUOP: 00110, ALURESULT: 00000000000000000000000000010110
                  41 OPERAND1: 00000000000000000000000000001111, OPERAND2: 00000000000000000000000000010100, ALUOP: 00110, ALURESULT: 00000000000000000000000000011011
OPERAND1 >> OPERAND2
                  45 OPERAND1: 00000000000000000000000000010100, OPERAND2: 00000000000000000000000000000010, ALUOP: 00111, ALURESULT: 00000000000000000000000000000000
                  46 OPERAND1: 00000000000000000000000000010100, OPERAND2: 00000000000000000000000000000010, ALUOP: 00111, ALURESULT: 00000000000000000000000000000101
OPERAND1 >>> OPERAND2
                  50 OPERAND1: 00000000000000000000000011110000, OPERAND2: 00000000000000000000000000000010, ALUOP: 01000, ALURESULT: 00000000000000000000000000000101
                  51 OPERAND1: 00000000000000000000000011110000, OPERAND2: 00000000000000000000000000000010, ALUOP: 01000, ALURESULT: 00000000000000000000000000111100
OPERAND1 | OPERAND2
                  55 OPERAND1: 00000000000000000000000000001111, OPERAND2: 00000000000000000000000000010100, ALUOP: 01001, ALURESULT: 00000000000000000000000011110010
                  56 OPERAND1: 00000000000000000000000000001111, OPERAND2: 00000000000000000000000000010100, ALUOP: 01001, ALURESULT: 00000000000000000000000000011111
OPERAND1 & OPERAND2
                  60 OPERAND1: 00000000000000000000000000001111, OPERAND2: 00000000000000000000000000010100, ALUOP: 01010, ALURESULT: 00000000000000000000000000000100
MUL
                  65 OPERAND1: 00000000000000000000000000001010, OPERAND2: 00000000000000000000000000010100, ALUOP: 01011, ALURESULT: 00000000000000000000000100101100
                  68 OPERAND1: 00000000000000000000000000001010, OPERAND2: 00000000000000000000000000010100, ALUOP: 01011, ALURESULT: 00000000000000000000000011001000
MUL
                  70 OPERAND1: 00000000000000000000000000000001, OPERAND2: 11111111111111111111111111111111, ALUOP: 01011, ALURESULT: 00000000000000000000000011001000
                  73 OPERAND1: 00000000000000000000000000000001, OPERAND2: 11111111111111111111111111111111, ALUOP: 01011, ALURESULT: 11111111111111111111111111111111
MULH
                  75 OPERAND1: 00000000000000000000000000001010, OPERAND2: 00000000000000000000000000010100, ALUOP: 01100, ALURESULT: 00000000000000000000000000000000
MULH
                  80 OPERAND1: 00000000000000000000000000000100, OPERAND2: 11111111111111111111111111111111, ALUOP: 01100, ALURESULT: 00000000000000000000000000000000
```

2. <u>Testing using testbench 2 (This is easy to understand)</u>



```
Test case 1: Forward operation
OPERAND1    :         20
OPERAND2    :          2
ALURESULT   :          2

Test case 2: OPERAND1 + OPERAND2
OPERAND1    :         20
OPERAND2    :          2
ALURESULT   :         22

Test case 3: OPERAND1 - OPERAND2
OPERAND1    :         20
OPERAND2    :          2
ALURESULT   :         18

Test case 4: OPERAND1 << OPERAND2
OPERAND1    : 00000000000000000000000000010100
OPERAND2    :          2
ALURESULT   : 00000000000000000000000001010000

Test case 5: Set less than (When OPERAND1 > OPERAND2)
OPERAND1    :         20
OPERAND2    :          2
ALURESULT   :          0

Test case 6: Set less than (When OPERAND1 < OPERAND2)
OPERAND1    :          2
OPERAND2    :         20
ALURESULT   :          1
```

Note: As there are many test cases, all the test case results are not shown in the report. But we have implemented them in the ALU test bench.

5. Branch-Jump Controller

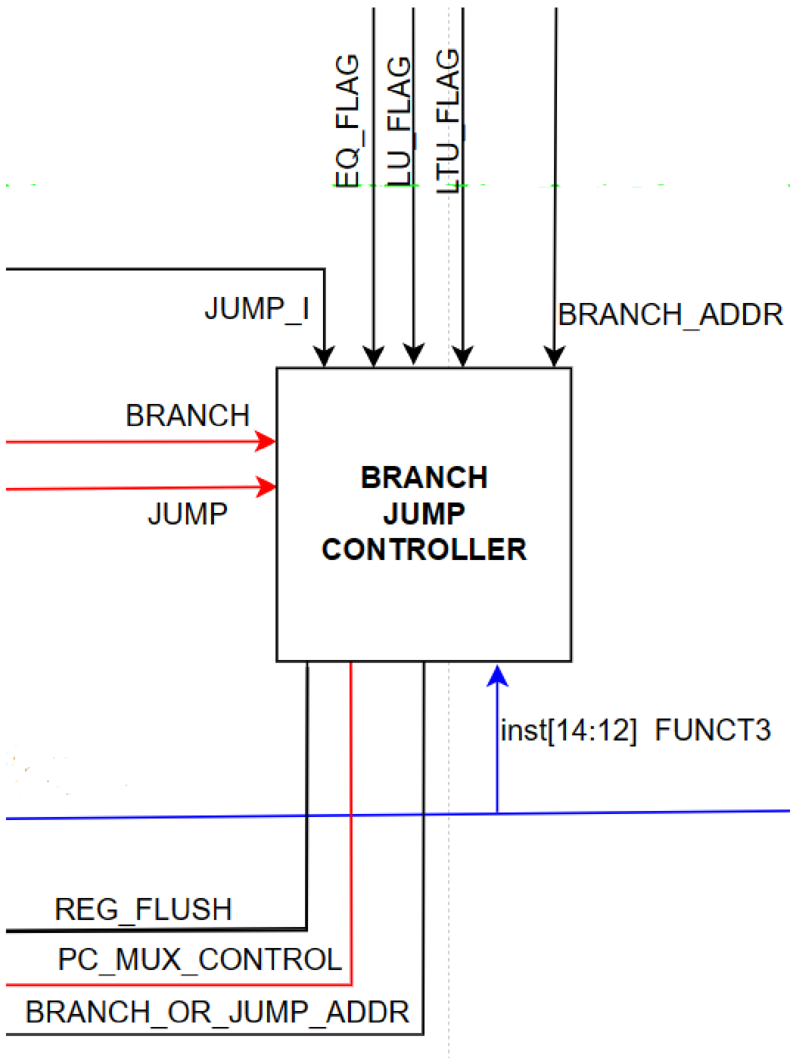


Figure 12: Branch Jump Controller

Inputs

1. BRANCH

   The BRANCH input is a signal that indicates whether a branch instruction has been executed.

2. JUMP

   The JUMP input is a signal that indicates whether a jump instruction has been executed.

3. FUNCT3

   Used to distinguish the condition of the branch instruction (Ex: Branch Less Than, Branch Greater Than or Equal, Branch Less Than Unsigned, etc)

4. EQ_FLAG (Equal flag)

   1-bit input signal used to notify whether DATA1=DATA2 or not from ALU.

5. LT_FLAG (Less than flag)

   1-bit input signal used to notify whether DATA1<DATA2 or not from ALU.

6. LTU_FLAG (Less than unsigned flag)

   1-bit input signal used to notify whether DATA1<DATA2 or not for unsigned data from ALU.

7. JUMP_I

   32 bit jump immediate value from sign and zero extend unit

8. BRANCH_ADDR

   32 bit branch address which was calculated using the adder in ALU.

Outputs

1. PC_MUX_CONTROL

   This is used in PC MUX as the select bit to choose the next PC value

2. BRANCH_OR_JUMP_ADDR

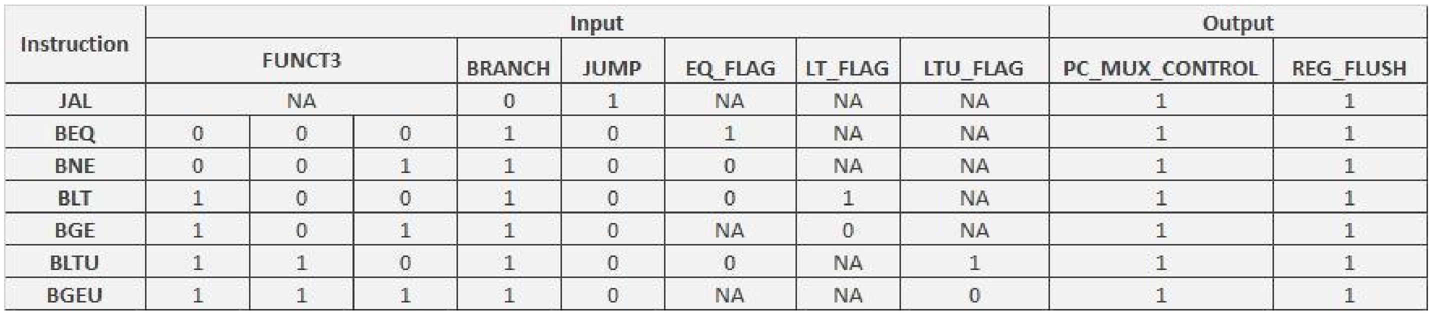
   This contains the target address of the Branch/ Jump instruction that has been executed.

3. REG_FLUSH

   This signal is used to clear fetched data from the pipeline registers (IF/ID Register an ID/EX Register) when branch or jump occurs.

Design Decisions

Delays- correct control signals will be available after a single time unit once the all necessary signals are received.



Instruction	Input								Output	
	FUNCT3			BRANCH	JUMP	EQ_FLAG	LT_FLAG	LTU_FLAG	PC_MUX_CONTROL	REG_FLUSH
JAL	NA			0	1	NA	NA	NA	1	1
BEQ	0	0	0	1	0	1	NA	NA	1	1
BNE	0	0	1	1	0	0	NA	NA	1	1
BLT	1	0	0	1	0	0	1	NA	1	1
BGE	1	0	1	1	0	NA	0	NA	1	1
BLTU	1	1	0	1	0	0	NA	1	1	1
BGEU	1	1	1	1	0	NA	NA	0	1	1

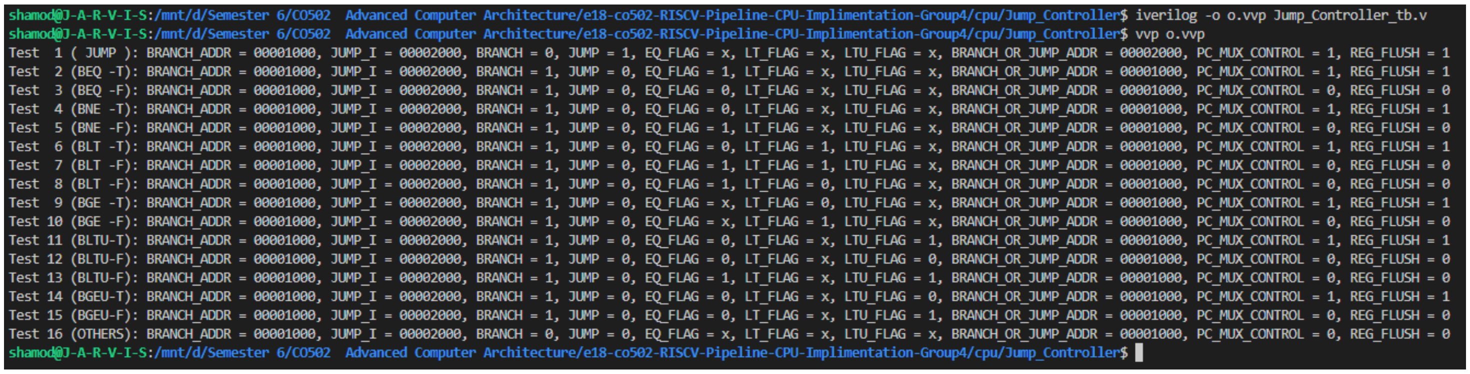
Output Signals According to Branch and Jump instructions to take the Branch or Jump

- Other Instructions and other combinations for above instructions (not affect if the necessary signal values are satisfied)

  PC_MUX_CONTROL = 0

  REG_FLUSH = 0

Testing Results for jump control unit:-



```
shamod@J-A-R-V-I-S:/mnt/d/Semester 6/CO502  Advanced Computer Architecture/e18-co502-RISCV-Pipeline-CPU-Implimentation-Group4/cpu/Jump_Controller$ iverilog -o o.vvp Jump_Controller_tb.v
shamod@J-A-R-V-I-S:/mnt/d/Semester 6/CO502  Advanced Computer Architecture/e18-co502-RISCV-Pipeline-CPU-Implimentation-Group4/cpu/Jump_Controller$ vvp o.vvp
Test  1 ( JUMP ): BRANCH_ADDR = 00001000, JUMP_I = 00002000, BRANCH = 0, JUMP = 1, EQ_FLAG = x, LT_FLAG = x, LTU_FLAG = x, BRANCH_OR_JUMP_ADDR = 00002000, PC_MUX_CONTROL = 1, REG_FLUSH = 1
Test  2 (BEQ -T): BRANCH_ADDR = 00001000, JUMP_I = 00002000, BRANCH = 1, JUMP = 0, EQ_FLAG = 1, LT_FLAG = x, LTU_FLAG = x, BRANCH_OR_JUMP_ADDR = 00001000, PC_MUX_CONTROL = 1, REG_FLUSH = 1
Test  3 (BEQ -F): BRANCH_ADDR = 00001000, JUMP_I = 00002000, BRANCH = 1, JUMP = 0, EQ_FLAG = 0, LT_FLAG = x, LTU_FLAG = x, BRANCH_OR_JUMP_ADDR = 00001000, PC_MUX_CONTROL = 0, REG_FLUSH = 0
Test  4 (BNE -T): BRANCH_ADDR = 00001000, JUMP_I = 00002000, BRANCH = 1, JUMP = 0, EQ_FLAG = 0, LT_FLAG = x, LTU_FLAG = x, BRANCH_OR_JUMP_ADDR = 00001000, PC_MUX_CONTROL = 1, REG_FLUSH = 1
Test  5 (BNE -F): BRANCH_ADDR = 00001000, JUMP_I = 00002000, BRANCH = 1, JUMP = 0, EQ_FLAG = 1, LT_FLAG = x, LTU_FLAG = x, BRANCH_OR_JUMP_ADDR = 00001000, PC_MUX_CONTROL = 0, REG_FLUSH = 0
Test  6 (BLT -T): BRANCH_ADDR = 00001000, JUMP_I = 00002000, BRANCH = 1, JUMP = 0, EQ_FLAG = 0, LT_FLAG = 1, LTU_FLAG = x, BRANCH_OR_JUMP_ADDR = 00001000, PC_MUX_CONTROL = 1, REG_FLUSH = 1
Test  7 (BLT -F): BRANCH_ADDR = 00001000, JUMP_I = 00002000, BRANCH = 1, JUMP = 0, EQ_FLAG = 1, LT_FLAG = 1, LTU_FLAG = x, BRANCH_OR_JUMP_ADDR = 00001000, PC_MUX_CONTROL = 0, REG_FLUSH = 0
Test  8 (BLT -F): BRANCH_ADDR = 00001000, JUMP_I = 00002000, BRANCH = 1, JUMP = 0, EQ_FLAG = 1, LT_FLAG = 0, LTU_FLAG = x, BRANCH_OR_JUMP_ADDR = 00001000, PC_MUX_CONTROL = 0, REG_FLUSH = 0
Test  9 (BGE -T): BRANCH_ADDR = 00001000, JUMP_I = 00002000, BRANCH = 1, JUMP = 0, EQ_FLAG = x, LT_FLAG = 0, LTU_FLAG = x, BRANCH_OR_JUMP_ADDR = 00001000, PC_MUX_CONTROL = 1, REG_FLUSH = 1
Test 10 (BGE -F): BRANCH_ADDR = 00001000, JUMP_I = 00002000, BRANCH = 1, JUMP = 0, EQ_FLAG = x, LT_FLAG = 1, LTU_FLAG = x, BRANCH_OR_JUMP_ADDR = 00001000, PC_MUX_CONTROL = 0, REG_FLUSH = 0
Test 11 (BLTU-T): BRANCH_ADDR = 00001000, JUMP_I = 00002000, BRANCH = 1, JUMP = 0, EQ_FLAG = 0, LT_FLAG = x, LTU_FLAG = 1, BRANCH_OR_JUMP_ADDR = 00001000, PC_MUX_CONTROL = 1, REG_FLUSH = 1
Test 12 (BLTU-F): BRANCH_ADDR = 00001000, JUMP_I = 00002000, BRANCH = 1, JUMP = 0, EQ_FLAG = 0, LT_FLAG = x, LTU_FLAG = 0, BRANCH_OR_JUMP_ADDR = 00001000, PC_MUX_CONTROL = 0, REG_FLUSH = 0
Test 13 (BLTU-F): BRANCH_ADDR = 00001000, JUMP_I = 00002000, BRANCH = 1, JUMP = 0, EQ_FLAG = 1, LT_FLAG = x, LTU_FLAG = 1, BRANCH_OR_JUMP_ADDR = 00001000, PC_MUX_CONTROL = 0, REG_FLUSH = 0
Test 14 (BGEU-T): BRANCH_ADDR = 00001000, JUMP_I = 00002000, BRANCH = 1, JUMP = 0, EQ_FLAG = 0, LT_FLAG = x, LTU_FLAG = 0, BRANCH_OR_JUMP_ADDR = 00001000, PC_MUX_CONTROL = 1, REG_FLUSH = 1
Test 15 (BGEU-F): BRANCH_ADDR = 00001000, JUMP_I = 00002000, BRANCH = 1, JUMP = 0, EQ_FLAG = 0, LT_FLAG = x, LTU_FLAG = 1, BRANCH_OR_JUMP_ADDR = 00001000, PC_MUX_CONTROL = 0, REG_FLUSH = 0
Test 16 (OTHERS): BRANCH_ADDR = 00001000, JUMP_I = 00002000, BRANCH = 0, JUMP = 0, EQ_FLAG = x, LT_FLAG = x, LTU_FLAG = x, BRANCH_OR_JUMP_ADDR = 00001000, PC_MUX_CONTROL = 0, REG_FLUSH = 0
shamod@J-A-R-V-I-S:/mnt/d/Semester 6/CO502  Advanced Computer Architecture/e18-co502-RISCV-Pipeline-CPU-Implimentation-Group4/cpu/Jump_Controller$
```

6. Memory

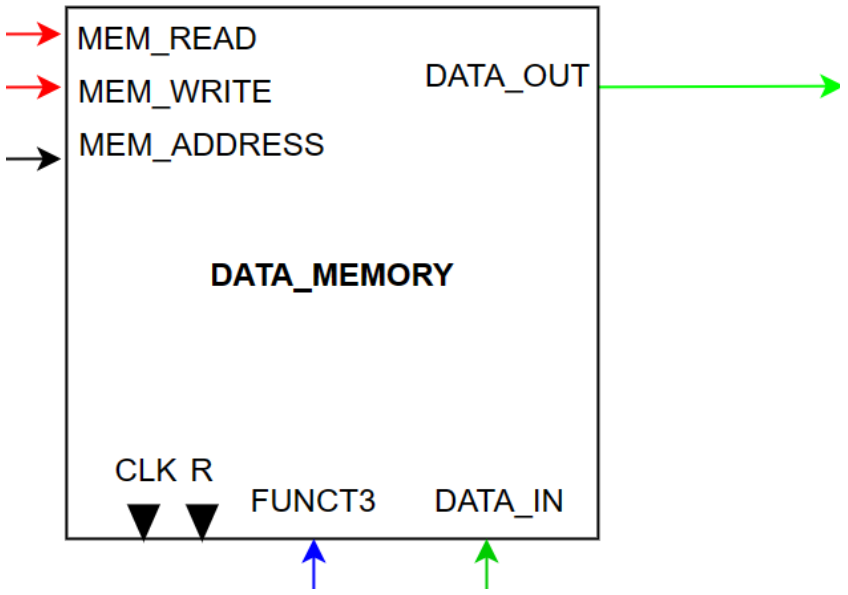


Figure 13: Memory

Inputs

1. CLK

   The clock signal is an input that synchronizes the operation of the memory with other components in the system.

2. MEM_READ

   The MEM_READ signal is an input that specifies whether the memory should read data from the specified address.

3. MEM_WRITE

   The MEM_WRITE signal is an input that specifies whether the memory should write data to the specified address.

4. MEM_ADDRESS

   The MEM_ADDRESS signal is an input that specifies the memory address to read from or write to.

5. funct3

   funct3 is an input that is used to decode the instruction and determine the operation to be performed on the memory.

6. DATA_IN

   The DATA_IN signal is an input that provides data to be written to the specified memory address.

<u>Outputs</u>

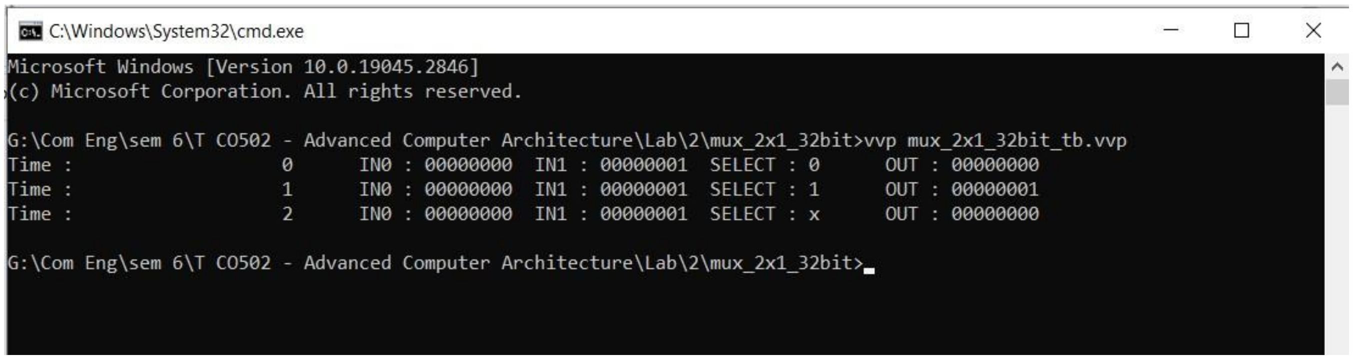
1. DATA_OUT

   The DATA_OUT signal is an output that provides the data read from the specified memory address.

7. <u>MUX 2 to 1</u>

In the pipeline diagram, MUX1, MUX2, MUX3, PC MUX, MUXDATAMEM and MUXJAL are 2 to 1 MUXes. Each mux has two 32-bit inputs and a 32-bit output. Delay is not added to mux. The delay is not added to the MUX because the impact of adding delay to the MUX may depend on other components in the system. For example, if other components in the system already introduce significant delays, adding delays to the MUX may not have a significant impact on the overall performance.

Testbench of the MUX 2 to 1,



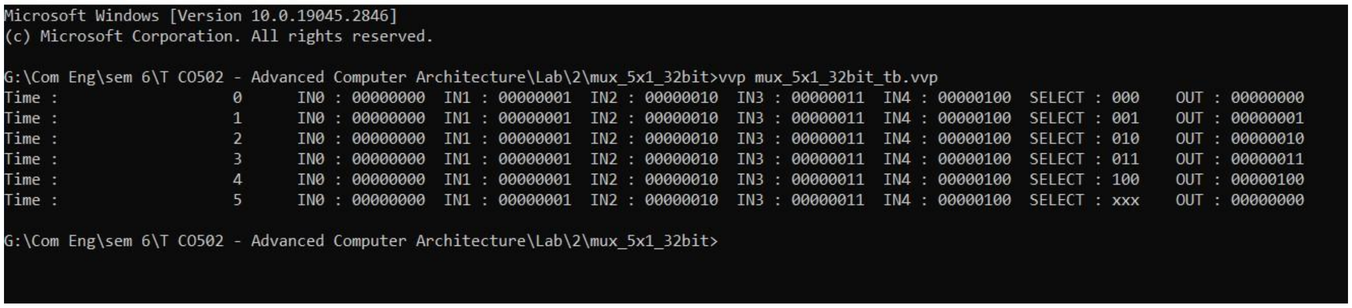
```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.2846]
(c) Microsoft Corporation. All rights reserved.

G:\Com Eng\sem 6\T CO502 - Advanced Computer Architecture\Lab\2\mux_2x1_32bit>vvp mux_2x1_32bit_tb.vvp
Time :                 0      IN0 : 00000000  IN1 : 00000001  SELECT : 0      OUT : 00000000
Time :                 1      IN0 : 00000000  IN1 : 00000001  SELECT : 1      OUT : 00000001
Time :                 2      IN0 : 00000000  IN1 : 00000001  SELECT : x      OUT : 00000000

G:\Com Eng\sem 6\T CO502 - Advanced Computer Architecture\Lab\2\mux_2x1_32bit>
```

8. <u>MUX 5 to 1</u>

Our design needs a 5 to 1 MUX to select the output of the SIGN/ZERO EXTEND unit. Our design can control five types of instructions, which are U type, I type, J type, S type, and B type. According to the MUXIMM_TYPE_SELECT signal, MUX selects one output. All inputs are 32-bit and all output is 32-bit.

The delay is not added to the MUX because the impact of adding delay to the MUX may depend on other components in the system. For example, if other components in the system already introduce significant delays, adding delays to the MUX may not have a significant impact on the overall performance.

Testbench of the MUX 5 to 1,



```
Microsoft Windows [Version 10.0.19045.2846]
(c) Microsoft Corporation. All rights reserved.

G:\Com Eng\sem 6\T CO502 - Advanced Computer Architecture\Lab\2\mux_5x1_32bit>vvp mux_5x1_32bit_tb.vvp
Time :                 0      IN0 : 00000000  IN1 : 00000001  IN2 : 00000010  IN3 : 00000011  IN4 : 00000100  SELECT : 000    OUT : 00000000
Time :                 1      IN0 : 00000000  IN1 : 00000001  IN2 : 00000010  IN3 : 00000011  IN4 : 00000100  SELECT : 001    OUT : 00000001
Time :                 2      IN0 : 00000000  IN1 : 00000001  IN2 : 00000010  IN3 : 00000011  IN4 : 00000100  SELECT : 010    OUT : 00000010
Time :                 3      IN0 : 00000000  IN1 : 00000001  IN2 : 00000010  IN3 : 00000011  IN4 : 00000100  SELECT : 011    OUT : 00000011
Time :                 4      IN0 : 00000000  IN1 : 00000001  IN2 : 00000010  IN3 : 00000011  IN4 : 00000100  SELECT : 100    OUT : 00000100
Time :                 5      IN0 : 00000000  IN1 : 00000001  IN2 : 00000010  IN3 : 00000011  IN4 : 00000100  SELECT : xxx    OUT : 00000000

G:\Com Eng\sem 6\T CO502 - Advanced Computer Architecture\Lab\2\mux_5x1_32bit>
```

9. <u>32-bit ADDER with PLUS 4</u>

   In the 32-bit adder, add an offset value to the PC in order to fetch the instruction at the correct address. The offset value is 4. PC is a register that holds the address of the next instruction to be fetched.

   One input is the PC value, the other is 4, and the output is PC+4. Adder has a 1-unit delay. The reason is that the adder does computational work.

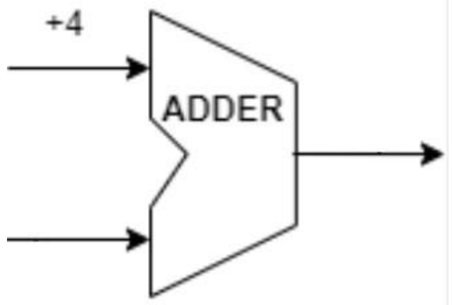
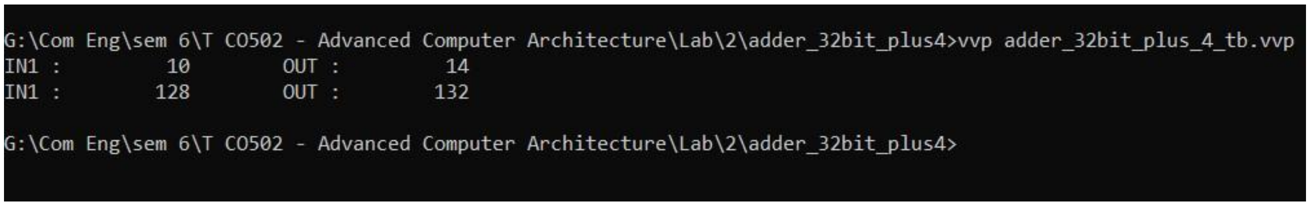


Figure 14: Adder with plus 4

Testbench of the adder,



```
G:\Com Eng\sem 6\T CO502 - Advanced Computer Architecture\Lab\2\adder_32bit_plus4>vvp adder_32bit_plus_4_tb.vvp
IN1 :         10        OUT :         14
IN1 :        128        OUT :        132

G:\Com Eng\sem 6\T CO502 - Advanced Computer Architecture\Lab\2\adder_32bit_plus4>
```

<u>Reason to skip implementation of FENCE AND FENCE.I instructions</u>**

fence instructions are an important feature of the RISC-V architecture that provide memory ordering guarantees in multi-threaded programs and ensure correct execution of memory operations.

Here we are only implementing a single pipelined CPU. Therefore this CPU couldn't support multithreading. Because of that we didn't implement support for FENCE instructions in our CPU.