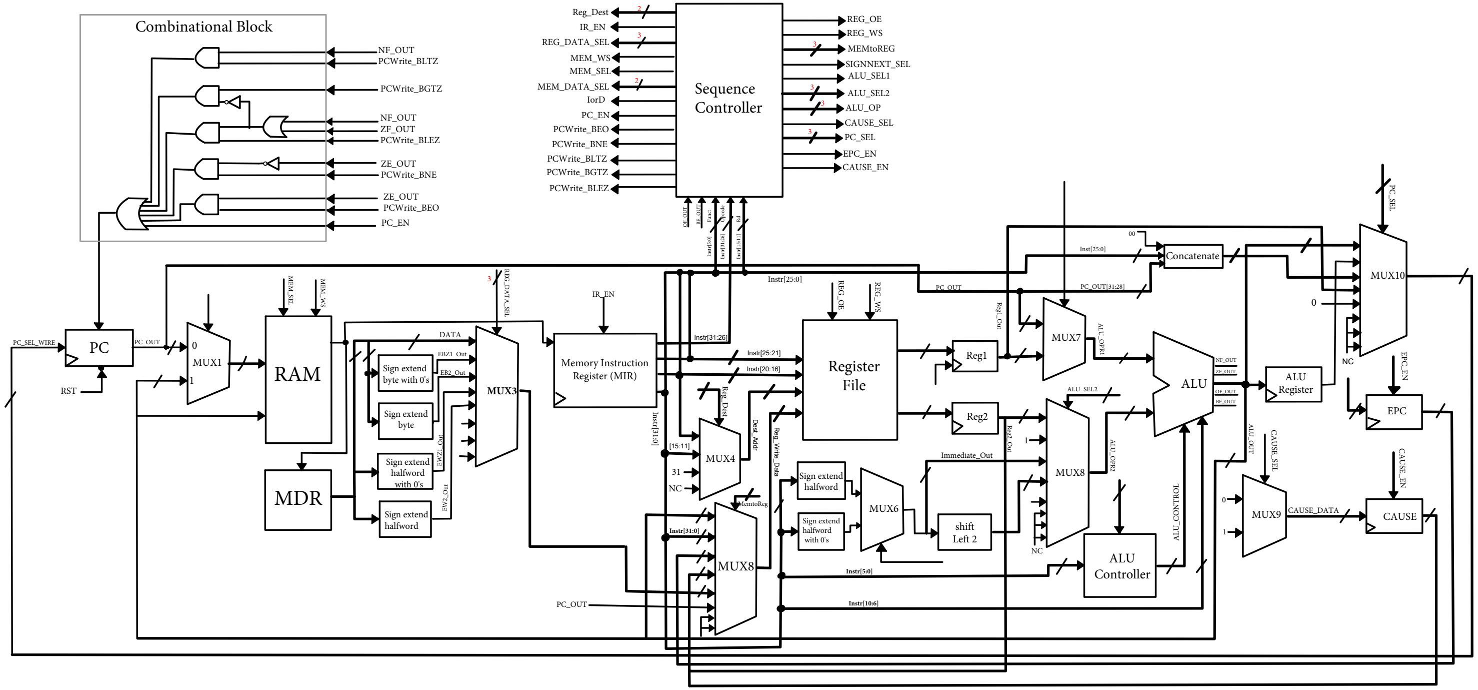

Combinational Block
NF_OUT
PCWrite_BLTZ
PCWrite_BGTZ
NF_OUT
ZF_OUT
PCWrite_BLEZ
ZE_OUT
PCWrite_BNE
ZE_OUT
PCWrite_BEO
PC_EN
Sequence Controller
Reg_Dest
IR_EN
REG_DATA_SEL
MEM_WS
MEM_SEL
MEM_DATA_SEL
IorD
PC_EN
PCWrite_BEO
PCWrite_BNE
PCWrite_BLTZ
PCWrite_BGTZ
PCWrite_BLEZ
REG_OE
REG_WS
MEMtoREG
SIGNNEXT_SEL
ALU_SEL1
ALU_SEL2
ALU_OP
CAUSE_SEL
PC_SEL
EPC_EN
CAUSE_EN
OF_OUT
BF_OUT
funct
Opcode
Rd
Instr[5:0]
Instr[31:26]
Instr[15:11]
PC_SEL_WIRE
PC
PC_OUT
RST
MUX1
RAM
MEM_SEL
MEM_WS
MDR
DATA
Sign extend byte with 0's
EBZ1_Out
EB2_Out
Sign extend byte
Sign extend halfword with 0's
EWZ1_Out
Sign extend halfword
EW2_Out
MUX3
REG_DATA_SEL
IR_EN
Memory Instruction Register (MIR)
Instr[25:0]
Instr[31:26]
Instr[25:21]
Instr[20:16]
Instr[31:0]
Reg_Dest
Dest_Addr
Reg_Write_Data
[15:11]
31
NC
MUX4
MemtoReg
Instr[31:0]
MUX8
PC_OUT
REG_OE
REG_WS
Register File
Reg1
Reg1_Out
Reg2
Reg2_Out
Sign extend halfword
Sign extend halfword with 0's
MUX6
shift Left 2
Immediate_Out
Instr[5:0]
Instr[10:6]
PC_OUT
Inst[25:0]
00
Concatenate
PC_OUT[31:28]
MUX7
ALU_OPR1
ALU_SEL2
MUX8
ALU_OPR2
NC
ALU
NF_OUT
ZF_OUT
OF_OUT
BF_OUT
ALU_CONTROL
ALU Controller
ALU_OUT
ALU Register
PC_SEL
MUX10
NC
EPC_EN
EPC
CAUSE_SEL
MUX9
CAUSE_DATA
CAUSE_EN
CAUSE

3 MIPS INSTRUCTIONS

R-Type

opcode[31:26]	Rs[25:21]	Rt[20:16]	Rd[15:11]	Shamt[10:6]	Function[5:0]
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-Type

opcode[31:26]	Rs[25:21]	Rt[20:16]	Address/constant [15:0]
6 bits	5 bits	5 bits	16 bits

J-Type

opcode[31:26]	Address[25:0]
6 bits	26 bits

3.1 Supported J-type Instructions

1. jump – j instruction

J-Type

000010	Jump target (26 bits)

Operation: PC ← PC[31:28] || Inst[25:0] || 00
Number of Cycles: 3
Sequence Controller States: 0, 1, and 2.
Comments: Used to jump to the address derived by shifting the 26-bit Jump Target left by two bits and concatenating the result with the 4 most significant bits of the original program counter.

2. jump and link – jal instruction

J-Type

000011	Jump target (26 bits)

Operation: [$ra]← PC + 4
PC ← PC[31:28] || Inst[25:0] || 00
Number of Cycles: 3
Sequence Controller States: 0, 1 and 3.
Comments: The next PC is calculated using the same equation as in the jump instruction. The difference in this instruction is that the next PC (PC + 4) is stored in the return address register ($ra).

3.2 Supported R-type Instructions

1. Jump and link register – jalr instruction

R-Type

000000	Rs	Rt	Rd	00000	001001

Operation: PC ← [Rs]
[$ra] ← PC + 4
Number of Cycles: 3
Sequence Controller States: 0, 1 and 21.
Comments: None.

2. Jump register – jr instruction

R-Type

000000	Rs	Rt	Rd	00000	001000

Operation: PC← [Rs]
Number of Cycles: 3
Sequence Controller States: 0, 1, and 20.
Comments: ALU not used in the execution of this instruction.

3. Addition – add instruction

R-Type

000000	Rs	Rt	Rd	00000	100000

Operation: Rd ← Rs + Rt
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 4, and 5.
Comments: In the case of an overflow, an exception is created and the Sequence Controller will set the appropriate registers. This means the instead of going to state 5 the Sequence Controller will go to state 30 in the write back cycle.

4. Unsigned addition – addu instruction

R-Type

000000	Rs	Rt	Rd	00000	100001

Operation: Rd ← Rs + Rt
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 4, and 5.
Comments: Same as add instruction, but overflow is ignored.

5. Subtraction – sub instruction

R-Type

000000	Rs	Rt	Rd	00000	100010

Operation: Rd ← Rs - Rt
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 4, and 5.
Comments: In the case of an overflow, an exception is created and the Sequence Controller will set the appropriate registers. This means the instead of going to state 5 the Sequence Controller will go to state 30 in the write back cycle.

6. Unsigned subtraction – subu instruction

R-Type

000000	Rs	Rt	Rd	00000	100011

Operation: Rd ← Rs - Rt
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 4, and 5.
Comments: Same as sub instruction, but overflow is ignored.

7. Logical AND – AND instruction

R-Type

000000	Rs	Rt	Rd	00000	100100

Operation: Rd ← Rs AND Rt
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 4, and 5.
Comments: None.

8. Logical OR – OR instruction

R-Type

000000	Rs	Rt	Rd	00000	100101

Operation: Rd ← Rs OR Rt
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 4, and 5.
Comments: None.

9. <u>Logical XOR – XOR instruction</u>

R-Type

000000	Rs	Rt	Rd	00000	100110

<u>Operation</u>: Rd ← Rs XOR Rt
PC ← PC + 4
<u>Number of Cycles</u>: 4
<u>Sequence Controller States</u>: 0, 1, 4, and 5.
<u>Comments</u>: None.

10. <u>Logical NOR – NOR instruction</u>

R-Type

000000	Rs	Rt	Rd	00000	100111

<u>Operation</u>: Rd ← Rs NOR Rt
PC ← PC + 4
<u>Number of Cycles</u>: 4
<u>Sequence Controller States</u>: 0, 1, 4, and 5.
<u>Comments</u>: None.

11. <u>Shift Left Logic – sll instruction</u>

R-Type

000000	Rs	Rt	Rd	active	000000

<u>Operation</u>: Rd ← Rt shifted left by Shamt
PC ← PC + 4
<u>Number of Cycles</u>: 4
<u>Sequence Controller States</u>: 0, 1, 4, and 5.
<u>Comments</u>: Rt is shifted left by the number specified by the shamt field of the instruction and stored in Rd. The most significant bits fall off.

12. Shift Left Logic by variable – sllv instruction

R-Type

000000	Rs	Rt	Rd	00000	000100

Operation: Rd ← Rt shifted left by Rs
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 4, and 5.
Comments: Rt is shifted left by the number specified by the Rs field of the instruction and stored in Rd. The most significant bits fall off.

13. Shift Right Logic – srl instruction

R-Type

000000	Rs	Rt	Rd	active	000010

Operation: Rd ← Rt shifted right by shamt
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 4, and 5.
Comments: Rt is shifted right by the number specified by the shamt field of the instruction and stored in Rd. The most significant bits are filled with 0's.

14. Shift Right Logic variable – srlv instruction

R-Type

000000	Rs	Rt	Rd	00000	000110

Operation: Rd ← Rt shifted right by Rs
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 4, and 5.
Comments: Rt is shifted right by the number specified by the Rs field of the instruction and stored in Rd. The most significant bits are filled with 0's.

15. Shift Right Arithmetic – sra instruction

R-Type

000000	Rs	Rt	Rd	active	000011

Operation: Rd ← Rt shifted right by shamt
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 4, and 5.
Comments: Rt is shifted right by the number specified by the shamt field of the instruction and stored in Rd. The sign bit is shifted in from the most significant end while bits fall off the least significant end.

16. Shift Right Arithmetic Variable – srav instruction

R-Type

000000	Rs	Rt	Rd	00000	000111

Operation: Rd ← Rt shifted right by Rs
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 4, and 5.
Comments: Rt is shifted right by the number specified by the Rs field of the instruction and stored in Rd. The sign bit is shifted in from the most significant end while bits fall off the least significant end.

17. Set if less than unsigned: sltu instruction

R-Type

000000	Rs	Rt	Rd	00000	101001

Operation: if Rs < Rt then Rd ← 1
Else Rd ← 0
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 4, and 5.
Comments: Rs and Rt are unsigned numbers.

18. Set if less than: slt instruction

R-Type

000000	Rs	Rt	Rd	00000	101010

Operation: if Rs < Rt then Rd ← 1
Else Rd ← 0
PC ← PC + 4

Number of Cycles: 4

Sequence Controller States: 0, 1, 4, and 5.

Comments: Rs and Rt are signed numbers.

19. Move from C0: mfc0 instruction

R-Type

010000	000000	Rt	Rd	00000	000000

Operation: if Rd = 01110 then Rt ← EPC
Else if Rd = 01101 then Rt ← Cause

Number of Cycles: 3

Sequence Controller States: 0, 1 and 32.

Comments: This instruction is dependent on Rd and is used to transfer the content of either Exception Register to the Register File.

20. Move to C0: mtc0 instruction

R-Type

010000	000100	Rt	Rd	00000	000000

Operation: if Rd = 01110 then EPC ← Rt
Else if Rd = 01101 then Cause ← Rt

Number of Cycles: 3

Sequence Controller States: 0, 1 and 34.

Comments: This instruction is dependent on Rd and is used to transfer the content of the Register File to either Exception Register .

21. Multiply(32-b) – mul instruction

R-Type

011100	Rs	Rt	Rd	00000	000010

Operation: Rd ← Rt * Rs
Number of Cycles: theoretically 4 but practically more than that, according to mult algorithm.
Sequence Controller States: 0, 1, 33, and 35.
Comments: Only the first 32 bits of the result are considered.

22. Move from hi – mfhi instruction

R-Type

000000	Rs	Rt	Rd	00000	010000

Operation: Rd ← hi
Number of Cycles: 2.
Sequence Controller States: 0, 1.
Comments: none.

23. Move to hi – mthi instruction

R-Type

000000	Rs	Rt	Rd	00000	010001

Operation: hi ← Rs
Number of Cycles: 2.
Sequence Controller States: 0, 1.
Comments: none.

24. Move from lo – mflo instruction

R-Type

000000	Rs	Rt	Rd	00000	010010

Operation: Rd ← lo
Number of Cycles: 2.
Sequence Controller States: 0, 1.
Comments: none.

25. Move to lo – mtlo instruction

R-Type

000000	Rs	Rt	Rd	00000	010011

Operation: lo ← Rs

Number of Cycles: 2.
Sequence Controller States: 0, 1.
Comments: none.

26. Multiply – mult instruction

R-Type

000000	Rs	Rt	Rd	00000	011000

Operation: {hi,lo} ← Rt * Rs
Number of Cycles: theoretically 4 but practically more than that, according to mult algorithm.
Sequence Controller States: 0, 1, 36, and 37.
Comments: In the case of an overflow, an exception is created and the Sequence Controller will set the appropriate registers. This means the instead of going to state 37 theSequence Controller will go to state 30 in the write back cycle.

27. Divide – div instruction

R-Type

000000	Rs	Rt	Rd	00000	011010

Operation: lo ← Rs / Rt ,
hi ← Rs % Rt
Number of Cycles: theoretically 4 but practically more than that, according to DIV algorithm.
Sequence Controller States: 0, 1, 36, and 37.
Comments: In the case of an overflow, an exception is created and the Sequence Controller will set the appropriate registers. This means the instead of going to state 37 theSequence Controller will go to state 30 in the write back cycle.

3.3 Supported I-type Instructions

1. Branch on less than zero: bltz instruction

I-Type

000001	Rs	000000	Address/constant

Operation: if Rs < 0 then PC ← PC + 4 + ((sign extended I[15:0]) || 00)
else PC ← PC + 4
Number of Cycles: 3
Sequence Controller States: 0, 1, and 13.
Comments: If condition is met, branch to PC + 4 + 4*offset.

2. Branch on equal: beq instruction

I-Type

000100	Rs	Rt	Address/constant

Operation: if Rs = Rt then PC ← PC + 4 + ((sign extended I[15:0]) || 00)
else PC ← PC + 4
Number of Cycles: 3
Sequence Controller States: 0, 1, and 9.
Comments: If condition is met, branch to PC + 4 + 4*offset.

3. Branch on not equal: bne instruction

I-Type

000101	Rs	Rt	Address/constant

Operation: if Rs ≠ Rt then PC ← PC + 4 + ((sign extended I[15:0]) || 00)
else PC ← PC + 4
Number of Cycles: 3
Sequence Controller States: 0, 1, and 10.
Comments: If condition is met, branch to PC + 4 + 4*offset.

4. Branch on less than or equal zero: blez instruction

I-Type

000110	Rs	Rt	Address/constant

Operation: if Rs $\leq$ 0 then PC $\leftarrow$ PC + 4 + ((sign extended I[15:0]) || 00)
else PC $\leftarrow$ PC + 4

Number of Cycles: 3

Sequence Controller States: 0, 1, and 11.

Comments: If condition is met, branch to PC +4 + 4*offset.

5. Branch on greater than zero: bgtz instruction

I-Type

000111	Rs	Rt	Address/constant

Operation: if Rs > 0 then PC $\leftarrow$ PC + 4 + ((sign extended I[15:0]) || 00)
else PC $\leftarrow$ PC + 4

Number of Cycles: 3

Sequence Controller States: 0, 1, and 12.

Comments: If condition is met, branch to PC + 4 + 4*offset.

6. Immediate addition: addi instruction

I-Type

001000	Rs	Rt	Address/constant

Operation: Rt $\leftarrow$ Rs + (sign extended I[15:0])
PC $\leftarrow$ PC +4

Number of Cycles: 4

Sequence Controller States: 0, 1, 7, and 8.

Comments: In the case of an overflow, an exception is created.

7. Immediate addition unsigned: addiu instruction

I-Type

001001	Rs	Rt	Address/constant

Operation: Rt ← Rs + (sign extended I[15:0])
PC ← PC +4
Number of Cycles: 4
Sequence Controller States: 0, 1, 29, and 8.
Comments: Same as immediate add instruction, but overflow is ignored.

8. Immediate set-if-less-than unsigned: sltiu instruction

I-Type

001011	Rs	Rt	Address/constant

Operation: if Rs < (sign extended I[15:0]) then Rt ← 1
else Rt ← 0
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 28, and 8.
Comments: None.

9. Immediate set-if-less-than: slti instruction

I-Type

001010	Rs	Rt	Address/constant

Operation: if Rs < (sign extended I[15:0]) then Rt ← 1
else Rt ← 0
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 6, and 8.
Comments: None.

10. Immediate logic AND: andi instruction

I-Type

001100	Rs	Rt	Address/constant

Operation: Rt ← Rs AND (sign extended I[15:0])
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 14, and 8.
Comments: None.

11. Immediate logic OR: ori instruction

I-Type

001101	Rs	Rt	Address/constant

Operation: Rt ← Rs OR (sign extended I[15:0])
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 15, and 8.
Comments: None.

12. Immediate logic XOR: xori instruction

I-Type

001110	Rs	Rt	Address/constant

Operation: Rt ← Rs XOR (sign extended I[15:0])
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 16, and 8.
Comments: None.

13. Load Word: lw instruction

I-Type

100011	Rs	Rt	Address/constant

Operation: Rt ← M[Rs + (sign extended I[15:0])]
PC ← PC + 4
Number of Cycles: 5
Sequence Controller States: 0, 1, 7, 18 and 19.
Comments: ALU output (i.e. Rs + sign extended I[15:0]) determines the address of the word loaded from memory into the register file at the location specified by Rt.

14. Load Unsigned Byte: lbu instruction

I-Type

100100	Rs	Rt	Address/constant

Operation: Rt ← sign extended with 0's (M[Rs + (sign extended I[15:0])])
PC ← PC + 4
Number of Cycles: 5
Sequence Controller States: 0, 1, 7, 22 and 19.
Comments: The first byte is signed extended with 0's and loaded from memory into the Register File at the location specified by Rt.

15. Load Byte: lb instruction

I-Type

100000	Rs	Rt	Address/constant

Operation: Rt ← sign extended with the leftmost bit (M[Rs + (sign extended I[15:0])])
PC ← PC + 4
Number of Cycles: 5
Sequence Controller States: 0, 1, 7, 23 and 19.
Comments: The first byte is signed extended with the most significant bit and loaded from memory into the Register File at the location specified by Rt.

16. Load Unsigned Half Word: lhu instruction

I-Type

100101	Rs	Rt	Address/constant

Operation: Rt ← sign extended with 0's (M[Rs + (sign extended I[15:0])])
PC ← PC + 4
Number of Cycles: 5
Sequence Controller States: 0, 1, 7, 24 and 19.
Comments: The first half of the word is signed extended with 0's and loaded from memory into the Register File at the location specified by Rt.

17. Load Half Word: lh instruction

I-Type

100001	Rs	Rt	Address/constant

Operation: Rt ← sign extended with the leftmost bit (M[Rs + (sign extended I[15:0])])
PC ← PC + 4
Number of Cycles: 5
Sequence Controller States: 0, 1, 7, 25 and 19.
Comments: The first half word is signed extended with the most significant bit and loaded from memory into the Register File at the location specified by Rt.

18. Store Byte: sb instruction

I-Type

101000	Rs	Rt	Address/constant

Operation: M[Rs + (sign extended I[15:0])] ← Rt[7:0]
PC ← PC + 4
Number of Cycles: 4
Sequence Controller States: 0, 1, 7 and 26.
Comments: Rt[7:0] is sign extended with 0's and stored in memory at the address determined by the ALU output (i.e. Rs + sign extended I[15:0]).

19. Store Byte: sh instruction

I-Type

101001	Rs	Rt	Address/constant

Operation: M[Rs + (sign extended I[15:0])] ← Rt[15:0]
PC ← PC + 4

Number of Cycles: 4

Sequence Controller States: 0, 1, 7 and 27.

Comments: Rt[15:0] is sign extended with 0's and stored in memory at the address determined by the ALU output (i.e. Rs + sign extended I[15:0]).

20. Store Word: sw instruction

I-Type

101011	Rs	Rt	Address/constant

Operation: M[Rs + (sign extended I[15:0])] ← Rt
PC ← PC + 4

Number of Cycles: 4

Sequence Controller States: 0, 1, 7 and 17.

Comments: Rt[31:0] is stored in memory at the address determined by the ALU output (i.e. Rs + sign extended I[15:0]).

4 COMPONENTS AND OPERATION

4.1 ALU with Control Module

The Arithmetic Logic Unit (ALU) performs all of the major arithmetic and logical operations in the processor. Additionally, the ALU performs all of the shift operation with exception to a few 2-bit shift registers outside the ALU. The ALU operation is controlled by 4 control signals generated by the ALU controller. Figure 3 shows the ALU and ALU controller.

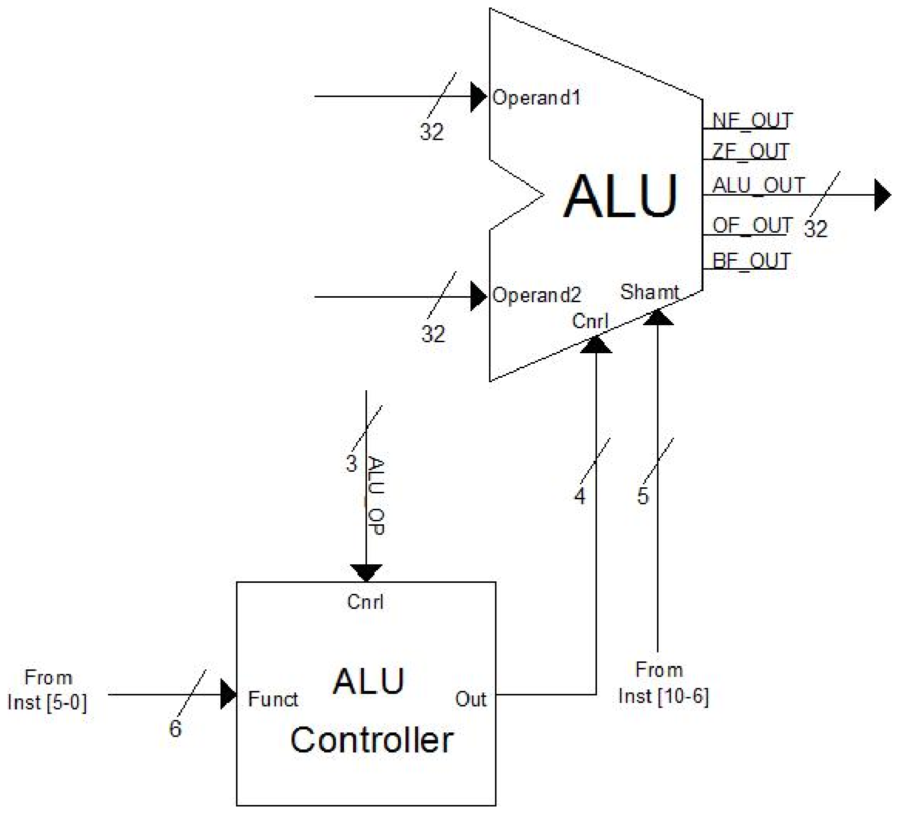


Figure 3. ALU with Controller

The 4 ALU input control signals dictate the ALU operation to be performed. This ALU design can perform a total of 14 operations. The ALU operations are shown in Table 2 along with their corresponding ALU control input combination.

Table 2. ALU Operations

ALU Control Input	ALU Operation
0000	AND
0001	OR
0010	add
0011	XOR
0100	NOR
0101	Set-if-less-than unsigned
0110	subtract
0111	Set-if-less-than
1000	Shift left logic
1001	Shift left logic by variable
1010	Shift right logic
1011	Shift right logic by variable
1100	Shift right arithmetic
1101	Shift right arith by variable
1110	Signed multiplication
1111	Signed division

The ALU control signals are dependent on the 6-bit Function and 3-bit ALU_OP inputs of the ALU controller. The Function input is connected to the function field of an R-type instruction (bits [5:0]) while the ALU_OP inputs are derived from the sequence controller. The ALU Controller first decodes the ALU_OP signals to determine the instruction opcode. If the instruction is an R-type instruction, the ALU controller uses the Function input to determine the ALU operation (i.e. ALU Control output) to be performed. The instruction is not an R-type instruction, the Function input is not used and the ALU Controller determines the ALU operation based on the ALU_OP inputs alone. Table 3 shows the correlation between all ALU control signals and the resulting ALU operation for all of the supported instruction. Note that x's signify 'don't cares' and used when the Function field is not used.

Table 3. ALU Controller Logic

Opcode	ALU_OP	Operation	Function	ALU Operation	ALU Control
lw	000	Load word	XXXXXX	add	0010
lb	000	Load byte	XXXXXX	add	0010
lbu	000	Load byte unsigned	XXXXXX	add	0010
lh	000	Load half word	XXXXXX	add	0010
lhu	000	Load half word unsigned	XXXXXX	add	0010
sw	000	Store word	XXXXXX	add	0010
sb	000	Store byte	XXXXXX	add	0010
sh	000	Store half word	XXXXXX	add	0010
addi	000	Add immediate	XXXXXX	add	0010
addiu	000	Add immediate unsigned	XXXXXX	add	0010
andi	100	AND immediate	XXXXXX	AND	0000
ori	101	OR immediate	XXXXXX	OR	0001
xori	110	XOR immediate	XXXXXX	XOR	0011
j	000	jump	XXXXXX	add	0010
jal	000	jump and link	XXXXXX	add	0010
beq	001	Branch equal	XXXXXX	subtract	0110
bne	001	Branch not equal	XXXXXX	subtract	0110
Blez	001	Branch if less than or equal to zero	XXXXXX	subtract	0110
bgtz	001	Branch if greater than zero	XXXXXX	subtract	0110
bltz	001	Branch if less than zero	XXXXXX	subtract	0110
slti	011	Set-if-less-than immediate	XXXXXX	Set-if-less-than	0111
sltiu	011	Set-if-less-than immediate unsigned	XXXXXX	Set-if-less-than	0111
mul	111	32-b multiplication	000010	multiply	1110
R-type	010	Jump register	001000	Don't Care	XXXX
		Jump and link reg	001001	add	0010
		add	100000	add	0010
		addu	100001	add unsigned	0010
		sub	100010	subtract	0110
		subu	100011	subtract unsigned	0110
		Logical AND	100100	AND	0000
		Logical OR	100101	OR	0001
		Logical XOR	100110	XOR	0011
		Logical NOR	100111	NOR	0100
		sll	000000	Shift left logic	1000

		sllv	000100	Shift left logic by variable	1001
		srl	000010	Shift right logic	1010
		srlv	000110	Shift right logic by variable	1011
		sra	000011	Shift right arithmetic	1100
		srav	000111	Shift right arith by variable	1101
		Set-if-less-than unsigned	101001	Set-if-less-than unsigned	0101
		Set-if-less-than	101010	Set-if-less-than	0111
		mult	011000	Signed_mult	1110
		div	011010	Signed_div	1111

The ALU also has a 5-bit Shamt input, which is derived from bits [10:6] of the instruction being executed, and two 32-bit operands. The Shamt input determines the shift amount for each shift operation and is not used for any other operation.

The ALU has 5 output signals including a 32-bit ALU-OUT signal and 4 single bit outputs, namely the Negative Flag, Zero Flag, Overflow Flag, and Bad Instruction Flag. The ALU-OUT signal simply provides the ALU result when performing the specific ALU operation on the two operands.

The Negative Flag, or NF_OUT as shown in Figure 4, is set true when the ALU output is a negative number meaning the most significant bit is one. The Negative Flag is updated in both arithmetic and Boolean operations and is used as a trigger for various branch operations.

The Zero Flag, or ZF_OUT, is set true when the ALU output is zero. Like the Negative Flag, the Zero Flag is used to facilitate various branch instructions as described in section 3.

The Overflow Flag, or OF_OUT, is set true in the case where the ALU results produce an arithmetic overflow. An arithmetic overflow occurs when the ALU result is too large to be correctly captured by the 32 bit output. As an example, adding two positive numbers and getting a negative result is an indication of an overflow condition as the most significant bit was carried out of the 32 bit output. Likewise, adding two negative numbers and getting a positive result will result in an overflow.

The Bad Instruction Flag, or BF_OUT, is set when either the opcode or function code of the instruction being executed is invalid. Both Overflow and Bad Instruction flags are used to detect processor exceptions. See section 5 for more information on processor exceptions.

Here is an explanation of each Sequence Controller FSM state :

- State 0 (Fetch Instruction) -
    - The instruction in the address location indicated by the current PC value is read from memory and set up to be written into the instruction register on the next clock cycle.
    - The current PC is incremented by 4 andloaded into the PC register on the next clock cycle.
- State 1 (Decode) –
    - ALU registers are loaded with operands from Register File.
    - Branch target is calculated by the ALU.
    - Jump address is derived by shifting Instruction [25-0] left by two bits and concatenating with PC[31-28] to generate 32 bits address.
    - Sequence Generator decodes instruction.
- State 2 (Execute Jump) –
    - PC is loaded with the jump address derived in State 1.
- State 3 (Execute Jump and link) –
    - PC + 4 is brought to the Register File awaiting to be stored at the return address register $Ra
- State 4 (Execute R-type) –
    - Operands from Reg1 and Reg2 are loaded into the ALU. Reg1 and Reg2 content is derived directly from the instruction.
    - The ALU executes the operation determined by the function (bits [5-0]) and shamt (bits[10:6]) fields of the instruction as the ALU_OP signal is set to '010'.
- State 5 (Completion of R-type) –
    - ALU results are stored in the Register File at the location determined by the register destination field Rd (bits [15-11]) of the instruction.
- State 6 (Execution of SLTi) –
    - Operands from Reg1 and Reg2 are loaded into the ALU. Reg1 is derived directly from the instruction while Reg2 has the sign extended immediate value.
    - The ALU executes a "set-if-less-than" operation.
- State 7 (Execution of various I-type) –
    - Operands from Reg1 and Reg2 are loaded into the ALU. Reg1 is derived directly from the instruction while Reg2 has the sign extended immediate value.
    - The ALU executes an "add" operation.
- State 8 (Completion of various I-type) –
    - ALU results are stored in the Register File at the location determined by the Rt field (bits [20-16]) of the instruction.

- State 9 (Completion of BEQ) –
    - ALU executes the branch condition
    - If branch condition is met (i.e. Rs = Rt), the PC is loaded with the branch target calculated in State 1.
- State 10 (Completion of BNE) –
    - ALU executes the branch condition
    - If branch condition is met (i.e. Rs ≠ Rt), the PC is loaded with the branch target calculated in State 1.
- State 11 (Completion of BLEZ) –
    - ALU executes the branch condition
    - If branch condition is met (i.e. Rs ≤ 0), the PC is loaded with the branch target calculated in State 1.
- State 12 (Completion of BGTZ) –
    - ALU executes the branch condition
    - If branch condition is met (i.e. Rs > 0), the PC is loaded with the branch target calculated in State 1.
- State 13 (Completion of BLTZ) –
    - ALU executes the branch condition
    - If branch condition is met (i.e. Rs < 0), the PC is loaded with the branch target calculated in State 1.
- State 14 (Execution of ANDi) –
    - Operands from Reg1 and Reg2 are loaded into the ALU. Reg1 is derived directly from the instruction while Reg2 has the sign extended immediate value.
    - The ALU executes an "AND" operation.
- State 15 (Execution of ORi) –
    - Operands from Reg1 and Reg2 are loaded into the ALU. Reg1 is derived directly from the instruction while Reg2 has the sign extended immediate value.
    - The ALU executes an "OR" operation.
- State 16 (Execution of XORi) –
    - Operands from Reg1 and Reg2 are loaded into the ALU. Reg1 is derived directly from the instruction while Reg2 has the sign extended immediate value.
    - The ALU executes an "XOR" operation.
- State 17 (Memory write for SW) –
    - Contents of Reg2 are brought to RAM_data input to be stored at the address specified by the ALU output calculated in State 7.
- State 18 (Memory access for LW) –
    - Memory location specified by the ALU output is read and written into the instruction register.

- State 19 (Memory write for LW) –
    - Memory data from instruction register is loaded into the register file at the location specified by the Rt field of the extracted instruction.
- State 20 (Execute jump register (JR)) –
    - PC is loaded with register file content at location specified by Rs field of the instruction.
- State 21 (Execute jump and link register (JALR)) –
    - Register file content at location specified by Rs field of the instruction is extracted.
    - PC + 4 is brought to the Register File Write Data input to be stored in the return address register $Ra specified by the Rd field of the instruction.
- State 22 (Memory access for LBu) –
    - Memory location specified by the ALU output is read, and the unsigned least significant byte is written into the register file.
- State 23 (Memory access for LB) –
    - Memory location specified by the ALU output is read, and the least significant byte is written into the register file.
- State 24 (Memory access for LHu) –
    - Memory location specified by the ALU output is read, and the unsigned least significant haft word is written into the register.
- State 25 (Memory access for LH) –
    - Memory location specified by the ALU output is read, and the least significant half word is written into the instruction register file.
- State 26 (Memory write for SB) –
    - Least significant byte of Register 2 is brought to theRAM Data input to be stored at the address specified by the ALU output calculated in State 7.
- State 27 (Memory write for SH) –
    - Least significant half word of Register 2 is brought tothe RAM Data input to be stored at the address specified by the ALU output calculated in State 7.
- State 28 (Execution of SLTiu) –
    - Operands from Reg1 and Reg2 are loaded into the ALU. Reg1 is derived directly from the instruction while Reg2 has the zero sign extended immediate value.
    - The ALU executes a “set-if-less-than” operation.
- State 29 (Execution of ADDiu) –
    - Operands from Reg1 and Reg2 are loaded into the ALU. Reg1 is derived directly from the instruction while Reg2 has the zero sign extended immediate value.
    - The ALU executes an “add” operation.

- State 30 (Exception handling for arithmetic overflow) –
    - The Cause register is loaded with ‘0’ to identify an arithmetic overflow exception.
    - The EPC register is loaded with the address of the next instruction (i.e. PC + 4).
    - The PC is loaded with the hardwired address of the exception handling routine, which is 0x0000.
- State 31 (Exception handling for invalid instruction) –
    - The Cause register is loaded with ‘1’ to identify an invalid instruction exception.
    - The EPC register is loaded with the address of the next instruction (i.e. PC + 4).
    - The PC is loaded with the hardwired address of the exception handling routine which is 0x0000.
- State 32 (MFC0 -EPC-CAUSE) –
    - Content of the EPC/CAUSE register are brought to the Register File to be stored in address specified by Rt .
- State 33 (MUL-32_bit Multiplication) –
    - The one where the execution of 32-bit multiplication operation occurs.
- State 34 (MTC0 -EPC-CAUSE) –
    - Content of the EPC/CAUSE register are updated by the values stored in the Register File at the address specified by Rt
- State 35 (MUL-32_bit Multiplication) –
    - The one where the write-back onto Rd after 32-bit multiplication operation gets executed occurs.
- State 36 (MULT/DIV) –
    - The one where the execution of multiplication/division operation occurs.
- State 37 (MULT/DIV) –
    - The one where the write-back onto hi&lo registers after multiplication/division operation gets executed occurs.

.

- **Test Bench Description:**

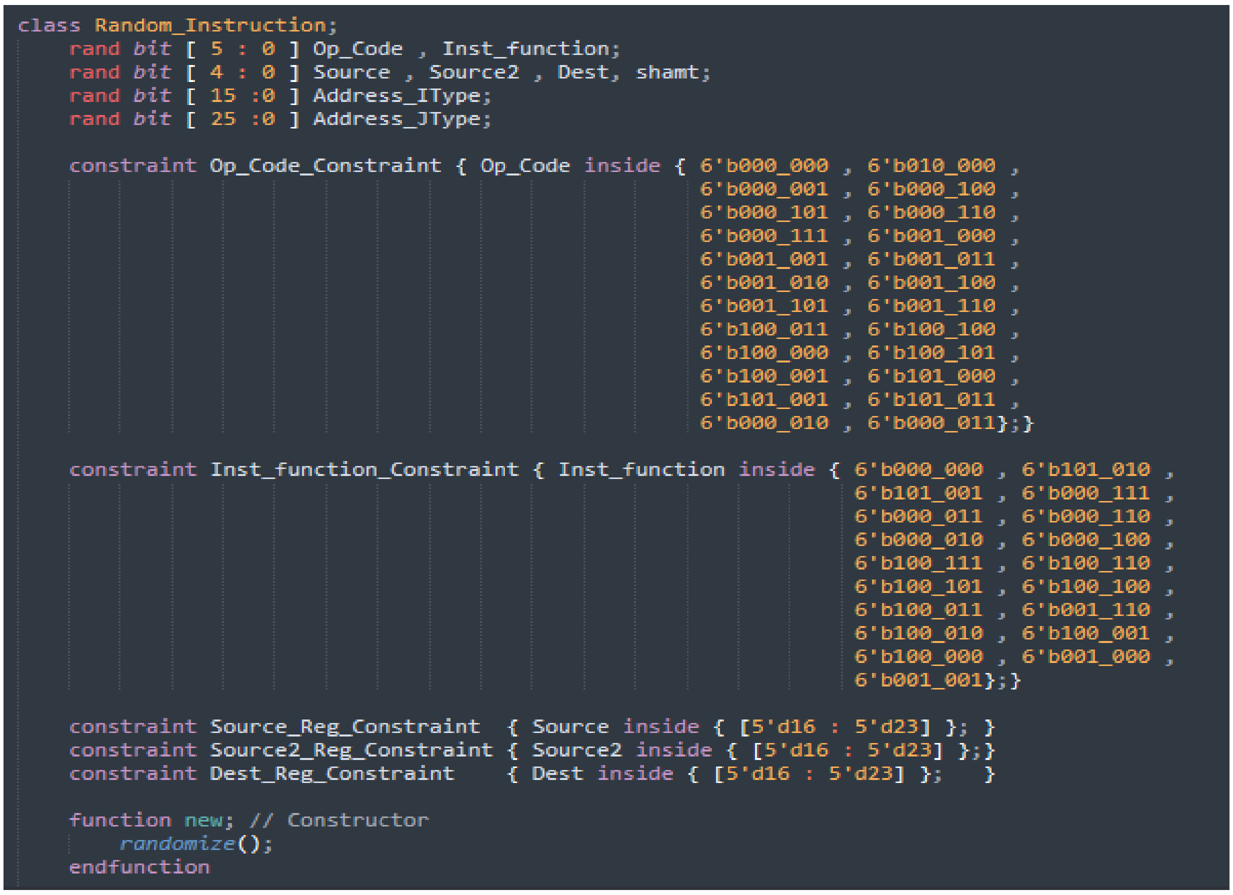
We used a generic testbench to generate random instruction with random sources, destinations, and types, in order to fully test our multi cycled MIPS.

Testbench uses randomization techniques in system Verilog , but the generated randomized function is constrained to our OP Codes and saved registers location of our MIPS Processor, in order to avoid making any undefined instructions that our processor donot support.

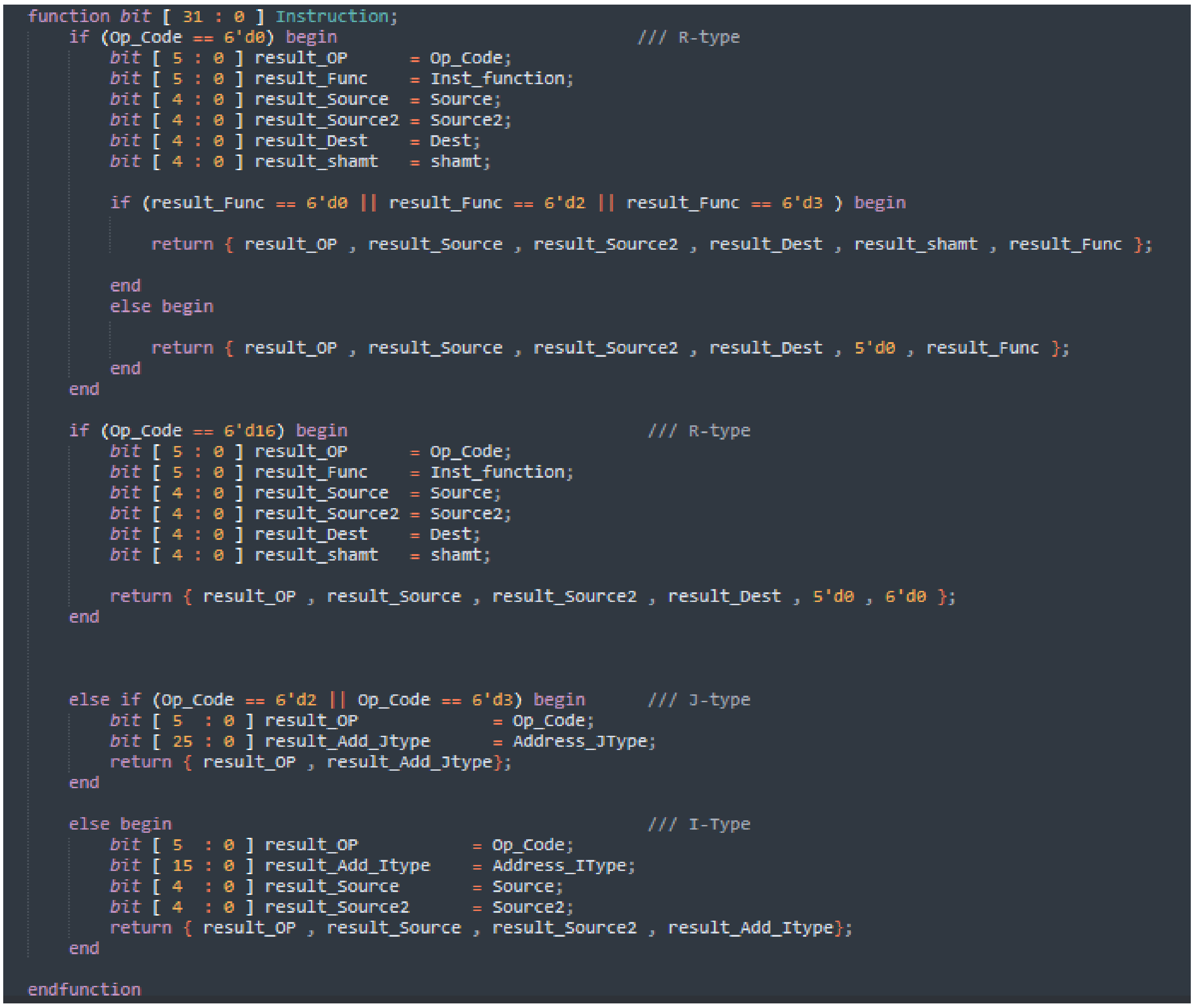
We also made a monitor function to give us a complete description of each randomized instruction in order to tell which instruction passes or fails within our design.

And we output both of results in two separate files, one containing the randomized instructions which are generated from random function, and the other contains the complete description of each generated instruction.

➢ Randomize Function:

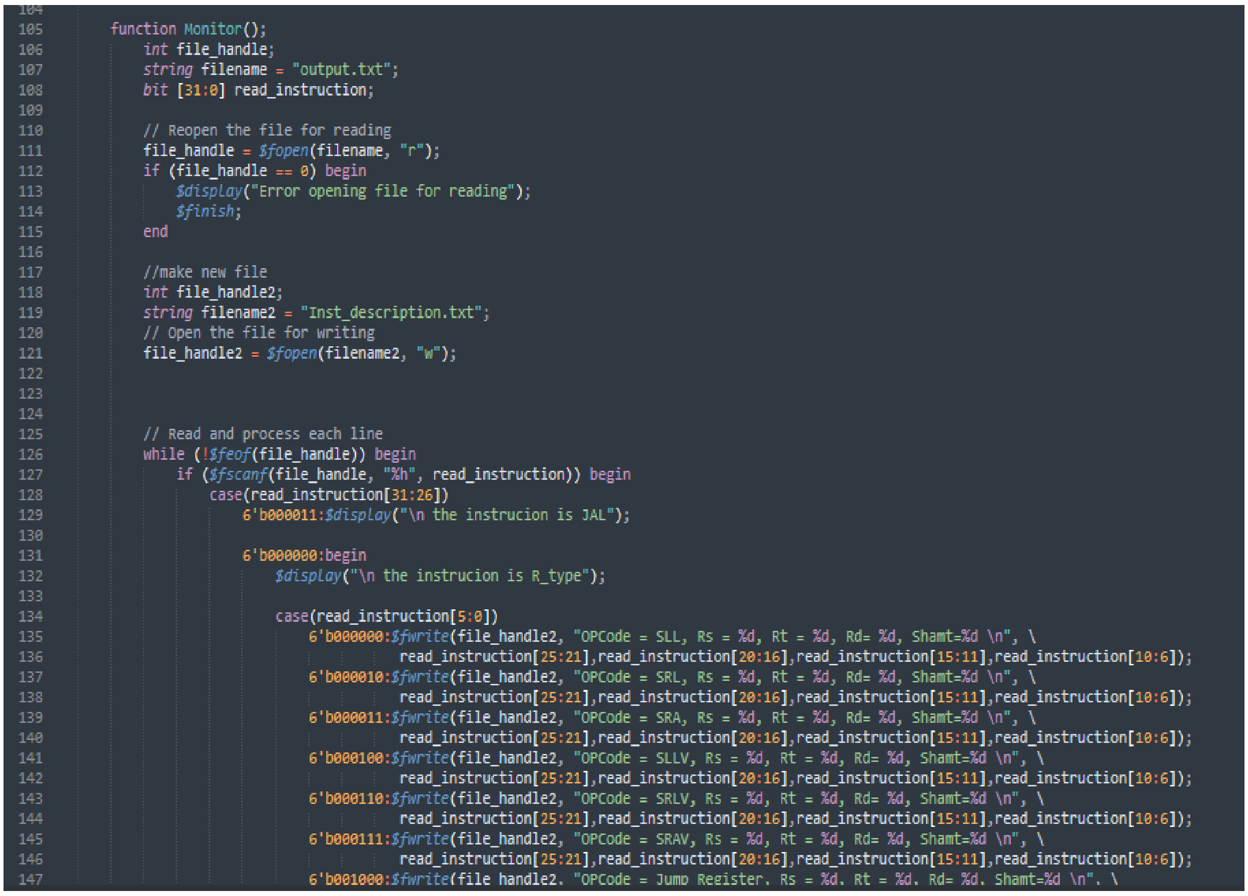


```
class Random_Instruction;
    rand bit [ 5 : 0 ] Op_Code , Inst_function;
    rand bit [ 4 : 0 ] Source , Source2 , Dest, shamt;
    rand bit [ 15 :0 ] Address_IType;
    rand bit [ 25 :0 ] Address_JType;

    constraint Op_Code_Constraint { Op_Code inside { 6'b000_000 , 6'b010_000 ,
                                                     6'b000_001 , 6'b000_100 ,
                                                     6'b000_101 , 6'b000_110 ,
                                                     6'b000_111 , 6'b001_000 ,
                                                     6'b001_001 , 6'b001_011 ,
                                                     6'b001_010 , 6'b001_100 ,
                                                     6'b001_101 , 6'b001_110 ,
                                                     6'b100_011 , 6'b100_100 ,
                                                     6'b100_000 , 6'b100_101 ,
                                                     6'b100_001 , 6'b101_000 ,
                                                     6'b101_001 , 6'b101_011 ,
                                                     6'b000_010 , 6'b000_011};}

    constraint Inst_function_Constraint { Inst_function inside { 6'b000_000 , 6'b101_010 ,
                                                                 6'b101_001 , 6'b000_111 ,
                                                                 6'b000_011 , 6'b000_110 ,
                                                                 6'b000_010 , 6'b000_100 ,
                                                                 6'b100_111 , 6'b100_110 ,
                                                                 6'b100_101 , 6'b100_100 ,
                                                                 6'b100_011 , 6'b001_110 ,
                                                                 6'b100_010 , 6'b100_001 ,
                                                                 6'b100_000 , 6'b001_000 ,
                                                                 6'b001_001};}

    constraint Source_Reg_Constraint  { Source inside { [5'd16 : 5'd23] }; }
    constraint Source2_Reg_Constraint { Source2 inside { [5'd16 : 5'd23] };}
    constraint Dest_Reg_Constraint    { Dest inside { [5'd16 : 5'd23] };   }

    function new; // Constructor
        randomize();
    endfunction
```

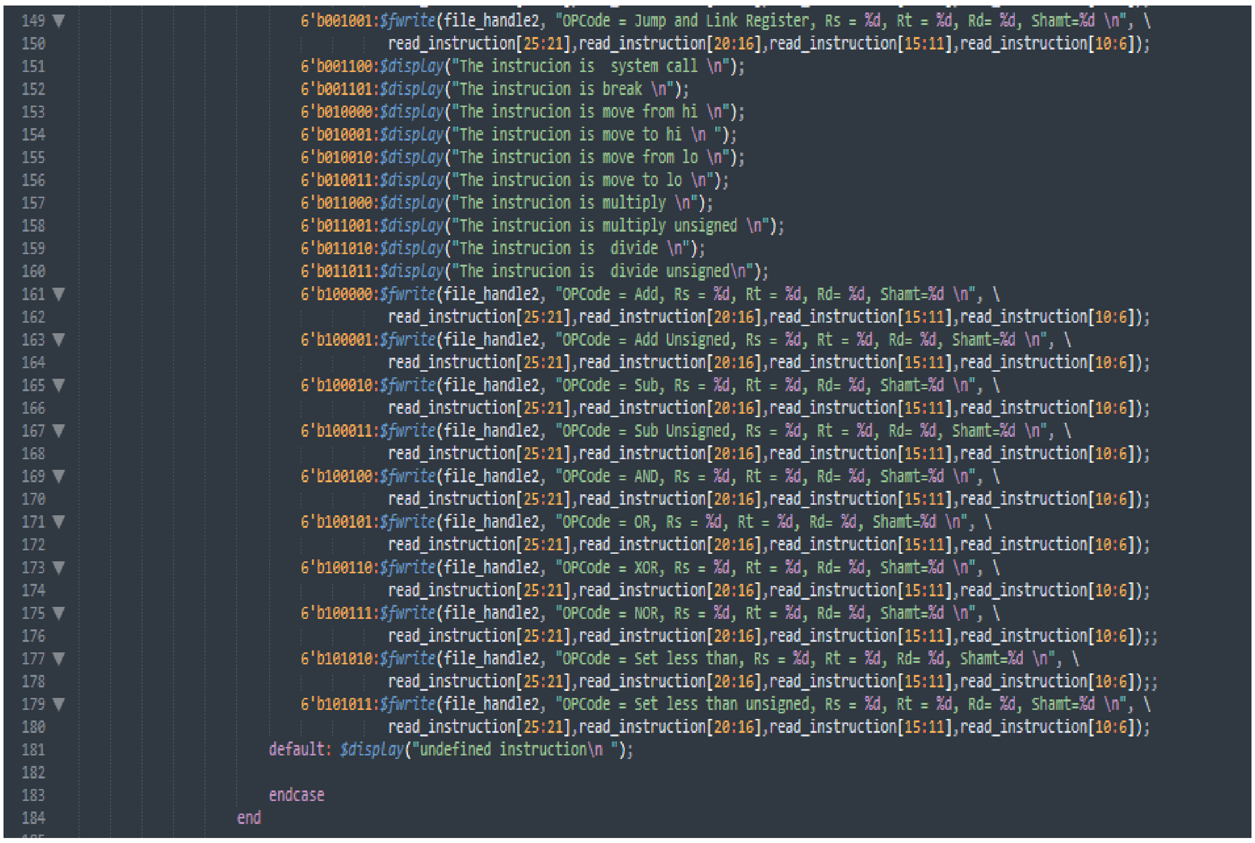


```
function bit [ 31 : 0 ] Instruction;
    if (Op_Code == 6'd0) begin                              /// R-type
        bit [ 5 : 0 ] result_OP      = Op_Code;
        bit [ 5 : 0 ] result_Func    = Inst_function;
        bit [ 4 : 0 ] result_Source  = Source;
        bit [ 4 : 0 ] result_Source2 = Source2;
        bit [ 4 : 0 ] result_Dest    = Dest;
        bit [ 4 : 0 ] result_shamt   = shamt;

        if (result_Func == 6'd0 || result_Func == 6'd2 || result_Func == 6'd3 ) begin

            return { result_OP , result_Source , result_Source2 , result_Dest , result_shamt , result_Func };

        end
        else begin

            return { result_OP , result_Source , result_Source2 , result_Dest , 5'd0 , result_Func };
        end
    end

    if (Op_Code == 6'd16) begin                              /// R-type
        bit [ 5 : 0 ] result_OP      = Op_Code;
        bit [ 5 : 0 ] result_Func    = Inst_function;
        bit [ 4 : 0 ] result_Source  = Source;
        bit [ 4 : 0 ] result_Source2 = Source2;
        bit [ 4 : 0 ] result_Dest    = Dest;
        bit [ 4 : 0 ] result_shamt   = shamt;

        return { result_OP , result_Source , result_Source2 , result_Dest , 5'd0 , 6'd0 };
    end



    else if (Op_Code == 6'd2 || Op_Code == 6'd3) begin      /// J-type
        bit [ 5  : 0 ] result_OP             = Op_Code;
        bit [ 25 : 0 ] result_Add_Jtype      = Address_JType;
        return { result_OP , result_Add_Jtype};
    end

    else begin                                               /// I-Type
        bit [ 5  : 0 ] result_OP           = Op_Code;
        bit [ 15 : 0 ] result_Add_Itype    = Address_IType;
        bit [ 4  : 0 ] result_Source       = Source;
        bit [ 4  : 0 ] result_Source2      = Source2;
        return { result_OP , result_Source , result_Source2 , result_Add_Itype};
    end

endfunction
```

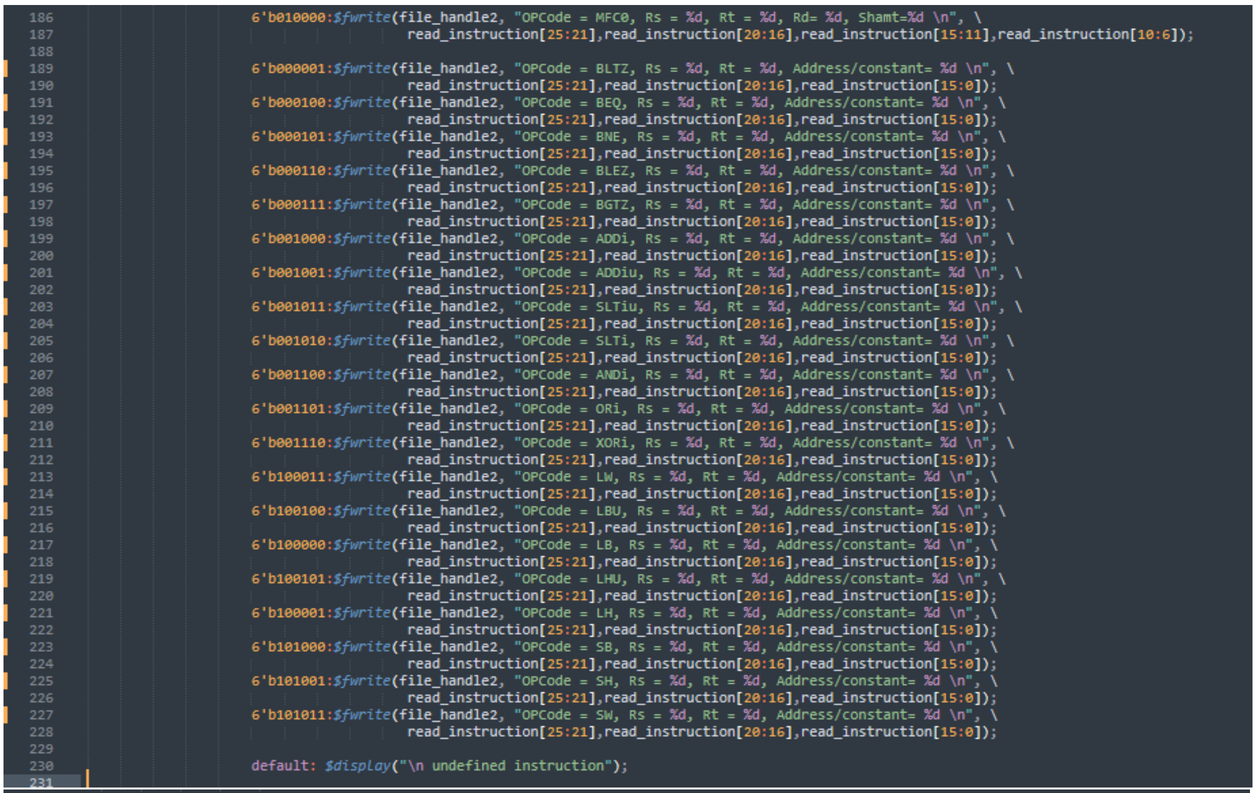
➢ Monitor Function:



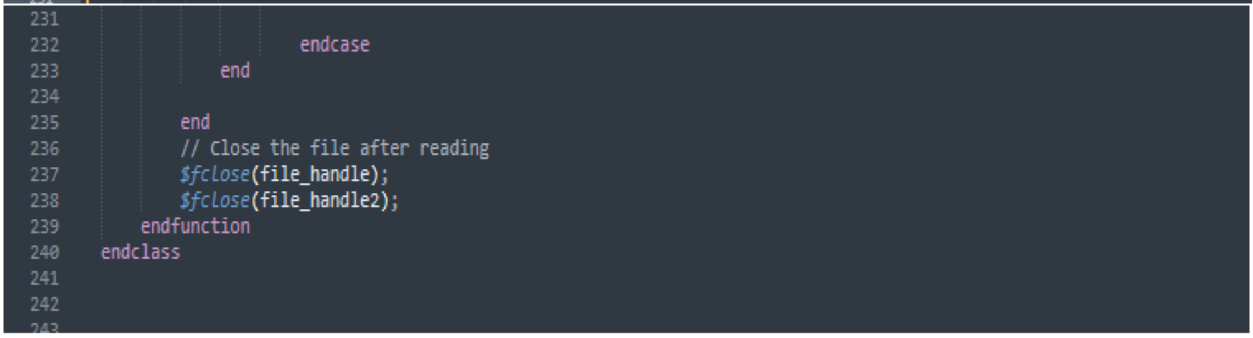
```
    function Monitor();
        int file_handle;
        string filename = "output.txt";
        bit [31:0] read_instruction;

        // Reopen the file for reading
        file_handle = $fopen(filename, "r");
        if (file_handle == 0) begin
            $display("Error opening file for reading");
            $finish;
        end

        //make new file
        int file_handle2;
        string filename2 = "Inst_description.txt";
        // Open the file for writing
        file_handle2 = $fopen(filename2, "w");



        // Read and process each line
        while (!$feof(file_handle)) begin
            if ($fscanf(file_handle, "%h", read_instruction)) begin
                case(read_instruction[31:26])
                    6'b000011:$display("\n the instrucion is JAL");

                    6'b000000:begin
                        $display("\n the instrucion is R_type");

                        case(read_instruction[5:0])
                            6'b000000:$fwrite(file_handle2, "OPCode = SLL, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                       read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);
                            6'b000010:$fwrite(file_handle2, "OPCode = SRL, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                       read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);
                            6'b000011:$fwrite(file_handle2, "OPCode = SRA, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                       read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);
                            6'b000100:$fwrite(file_handle2, "OPCode = SLLV, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                       read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);
                            6'b000110:$fwrite(file_handle2, "OPCode = SRLV, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                       read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);
                            6'b000111:$fwrite(file_handle2, "OPCode = SRAV, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                       read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);
                            6'b001000:$fwrite(file_handle2, "OPCode = Jump Register, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
```



```
                            6'b001001:$fwrite(file_handle2, "OPCode = Jump and Link Register, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                       read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);
                            6'b001100:$display("The instrucion is  system call \n");
                            6'b001101:$display("The instrucion is break \n");
                            6'b010000:$display("The instrucion is move from hi \n");
                            6'b010001:$display("The instrucion is move to hi \n ");
                            6'b010010:$display("The instrucion is move from lo \n");
                            6'b010011:$display("The instrucion is move to lo \n");
                            6'b011000:$display("The instrucion is multiply \n");
                            6'b011001:$display("The instrucion is multiply unsigned \n");
                            6'b011010:$display("The instrucion is  divide \n");
                            6'b011011:$display("The instrucion is  divide unsigned\n");
                            6'b100000:$fwrite(file_handle2, "OPCode = Add, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                       read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);
                            6'b100001:$fwrite(file_handle2, "OPCode = Add Unsigned, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                       read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);
                            6'b100010:$fwrite(file_handle2, "OPCode = Sub, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                       read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);
                            6'b100011:$fwrite(file_handle2, "OPCode = Sub Unsigned, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                       read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);
                            6'b100100:$fwrite(file_handle2, "OPCode = AND, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                       read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);
                            6'b100101:$fwrite(file_handle2, "OPCode = OR, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                       read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);
                            6'b100110:$fwrite(file_handle2, "OPCode = XOR, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                       read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);
                            6'b100111:$fwrite(file_handle2, "OPCode = NOR, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                       read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);;
                            6'b101010:$fwrite(file_handle2, "OPCode = Set less than, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                       read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);;
                            6'b101011:$fwrite(file_handle2, "OPCode = Set less than unsigned, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                       read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);
                        default: $display("undefined instruction\n ");

                        endcase
                    end
```

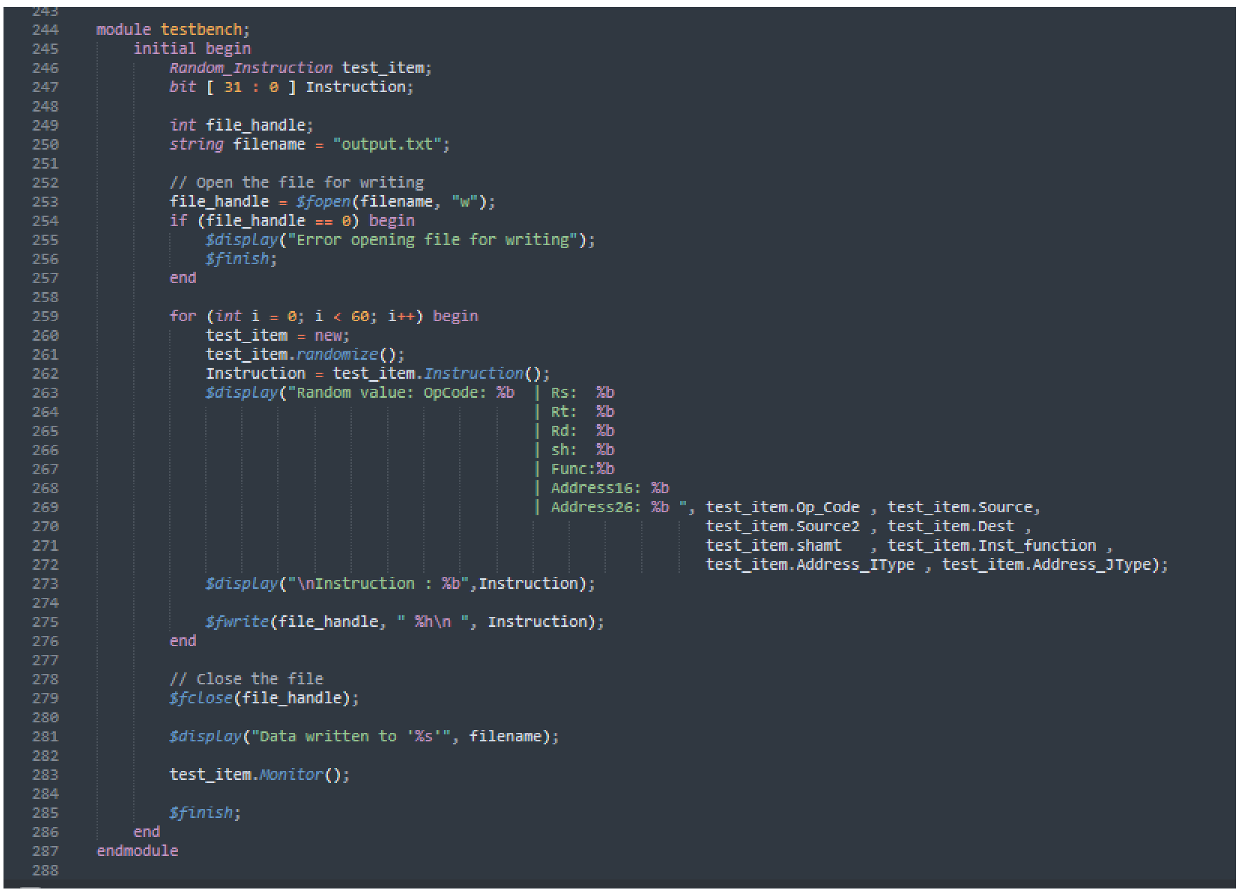


```
                    6'b010000:$fwrite(file_handle2, "OPCode = MFC0, Rs = %d, Rt = %d, Rd= %d, Shamt=%d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:11],read_instruction[10:6]);

                    6'b000001:$fwrite(file_handle2, "OPCode = BLTZ, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b000100:$fwrite(file_handle2, "OPCode = BEQ, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b000101:$fwrite(file_handle2, "OPCode = BNE, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b000110:$fwrite(file_handle2, "OPCode = BLEZ, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b000111:$fwrite(file_handle2, "OPCode = BGTZ, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b001000:$fwrite(file_handle2, "OPCode = ADDi, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b001001:$fwrite(file_handle2, "OPCode = ADDiu, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b001011:$fwrite(file_handle2, "OPCode = SLTiu, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b001010:$fwrite(file_handle2, "OPCode = SLTi, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b001100:$fwrite(file_handle2, "OPCode = ANDi, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b001101:$fwrite(file_handle2, "OPCode = ORi, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b001110:$fwrite(file_handle2, "OPCode = XORi, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b100011:$fwrite(file_handle2, "OPCode = LW, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b100100:$fwrite(file_handle2, "OPCode = LBU, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b100000:$fwrite(file_handle2, "OPCode = LB, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b100101:$fwrite(file_handle2, "OPCode = LHU, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b100001:$fwrite(file_handle2, "OPCode = LH, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b101000:$fwrite(file_handle2, "OPCode = SB, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b101001:$fwrite(file_handle2, "OPCode = SH, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);
                    6'b101011:$fwrite(file_handle2, "OPCode = SW, Rs = %d, Rt = %d, Address/constant= %d \n", \
                                        read_instruction[25:21],read_instruction[20:16],read_instruction[15:0]);

                    default: $display("\n undefined instruction");
```



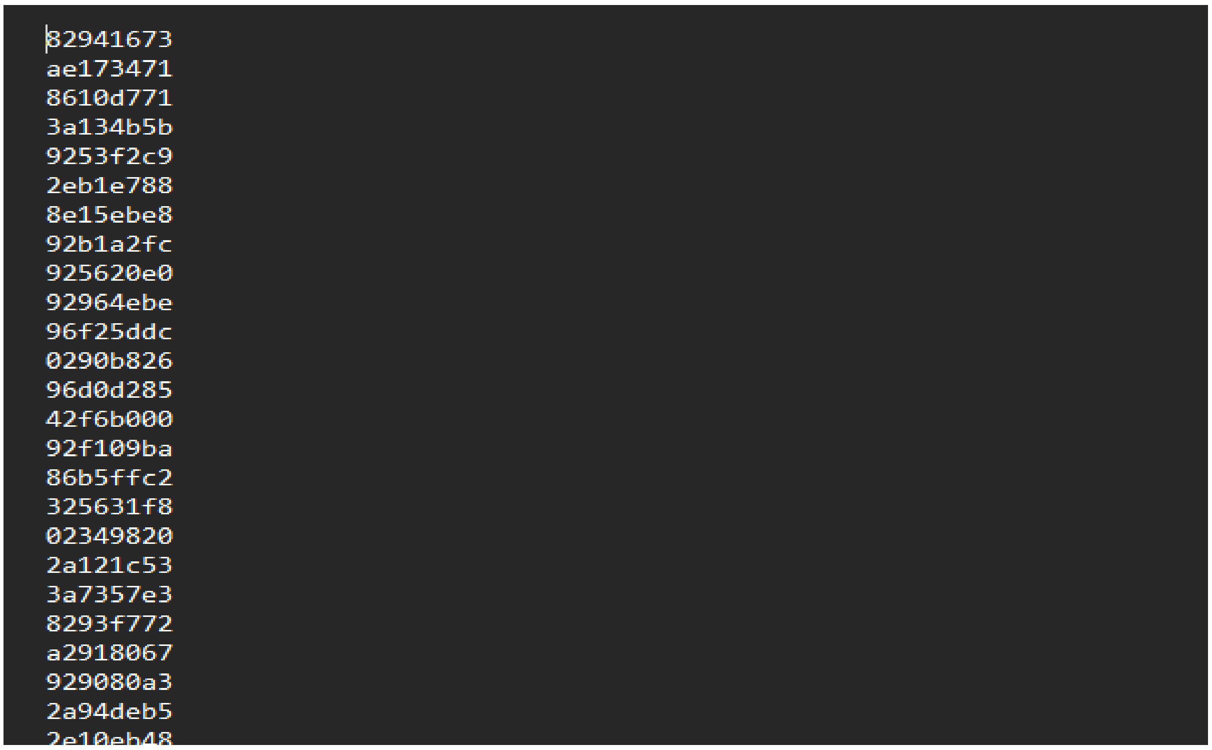
```

                    endcase
                end

            end
            // Close the file after reading
            $fclose(file_handle);
            $fclose(file_handle2);
        endfunction
    endclass


```

➢ Testbench Function:



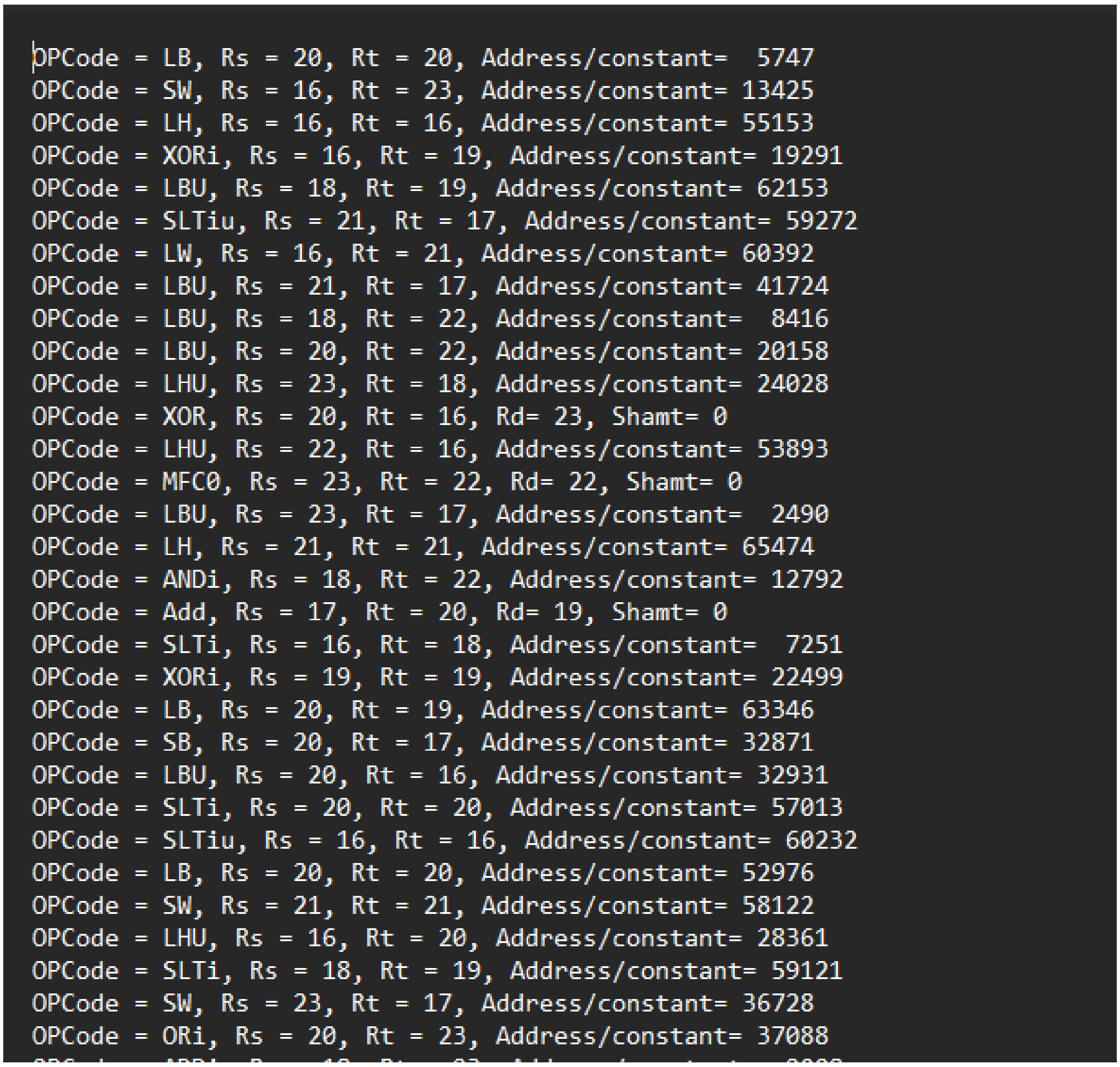
```
module testbench;
    initial begin
        Random_Instruction test_item;
        bit [ 31 : 0 ] Instruction;

        int file_handle;
        string filename = "output.txt";

        // Open the file for writing
        file_handle = $fopen(filename, "w");
        if (file_handle == 0) begin
            $display("Error opening file for writing");
            $finish;
        end

        for (int i = 0; i < 60; i++) begin
            test_item = new;
            test_item.randomize();
            Instruction = test_item.Instruction();
            $display("Random value: OpCode: %b  | Rs:  %b
                                                | Rt:  %b
                                                | Rd:  %b
                                                | sh:  %b
                                                | Func:%b
                                                | Address16: %b
                                                | Address26: %b ", test_item.Op_Code , test_item.Source,
                                                                   test_item.Source2 , test_item.Dest ,
                                                                   test_item.shamt   , test_item.Inst_function ,
                                                                   test_item.Address_IType , test_item.Address_JType);
            $display("\nInstruction : %b",Instruction);

            $fwrite(file_handle, " %h\n ", Instruction);
        end

        // Close the file
        $fclose(file_handle);

        $display("Data written to '%s'", filename);

        test_item.Monitor();

        $finish;
    end
endmodule
```

➢ Sample of Random Instruction File Produced:



```
82941673
ae173471
8610d771
3a134b5b
9253f2c9
2eb1e788
8e15ebe8
92b1a2fc
925620e0
92964ebe
96f25ddc
0290b826
96d0d285
42f6b000
92f109ba
86b5ffc2
325631f8
02349820
2a121c53
3a7357e3
8293f772
a2918067
929080a3
2a94deb5
2e10eb48
```

➢ Sample of Detailed Instruction Description File Produced:



```
OPCode = LB, Rs = 20, Rt = 20, Address/constant=  5747
OPCode = SW, Rs = 16, Rt = 23, Address/constant= 13425
OPCode = LH, Rs = 16, Rt = 16, Address/constant= 55153
OPCode = XORi, Rs = 16, Rt = 19, Address/constant= 19291
OPCode = LBU, Rs = 18, Rt = 19, Address/constant= 62153
OPCode = SLTiu, Rs = 21, Rt = 17, Address/constant= 59272
OPCode = LW, Rs = 16, Rt = 21, Address/constant= 60392
OPCode = LBU, Rs = 21, Rt = 17, Address/constant= 41724
OPCode = LBU, Rs = 18, Rt = 22, Address/constant=  8416
OPCode = LBU, Rs = 20, Rt = 22, Address/constant= 20158
OPCode = LHU, Rs = 23, Rt = 18, Address/constant= 24028
OPCode = XOR, Rs = 20, Rt = 16, Rd= 23, Shamt= 0
OPCode = LHU, Rs = 22, Rt = 16, Address/constant= 53893
OPCode = MFC0, Rs = 23, Rt = 22, Rd= 22, Shamt= 0
OPCode = LBU, Rs = 23, Rt = 17, Address/constant=  2490
OPCode = LH, Rs = 21, Rt = 21, Address/constant= 65474
OPCode = ANDi, Rs = 18, Rt = 22, Address/constant= 12792
OPCode = Add, Rs = 17, Rt = 20, Rd= 19, Shamt= 0
OPCode = SLTi, Rs = 16, Rt = 18, Address/constant=  7251
OPCode = XORi, Rs = 19, Rt = 19, Address/constant= 22499
OPCode = LB, Rs = 20, Rt = 19, Address/constant= 63346
OPCode = SB, Rs = 20, Rt = 17, Address/constant= 32871
OPCode = LBU, Rs = 20, Rt = 16, Address/constant= 32931
OPCode = SLTi, Rs = 20, Rt = 20, Address/constant= 57013
OPCode = SLTiu, Rs = 16, Rt = 16, Address/constant= 60232
OPCode = LB, Rs = 20, Rt = 20, Address/constant= 52976
OPCode = SW, Rs = 21, Rt = 21, Address/constant= 58122
OPCode = LHU, Rs = 16, Rt = 20, Address/constant= 28361
OPCode = SLTi, Rs = 18, Rt = 19, Address/constant= 59121
OPCode = SW, Rs = 23, Rt = 17, Address/constant= 36728
OPCode = ORi, Rs = 20, Rt = 23, Address/constant= 37088
```