UNIVERSITI TUNKU ABDUL RAHMAN

FACULTY OF INFORMATION & COMMUNICATION TECHNOLOGY

DEPARTMENT OF COMPUTER AND COMMUNICATION TECHNOLOGY

BACHELOR OF INFORMATION TECHNOLOGY (HONOURS)

COMPUTER ENGINEERING

YEAR TWO SEMESTER ONE

Session: 202506

UCEC2053 COMPUTER ORGANISATION AND ARCHITECTURE

ASSIGNMENT 1

PRACTICAL ASSIGNMENT

Pipeline Processor Design: Specification Development

Instructor: Mok Kai Ming

Date of Submission: 25/8/2025

Instruction Group: 1

Don't change the following grouping and sequence

Group Members		Tutorial Group	Task	Mobile Contact
1	All	1	Arch Spec, chip verification + boot program	
2	Ng Jing Xuan	1	Datapath, i-mem, d-mem	017-5759279
3	Chow Bin Lin	1	ALU, barrel shifter, register file	017-9916614
4	Ng Yu Heng	1	Main control, ALU control	011-55611695
5	Goh Xuan Hui	1	Multipler, hi-lo registers	011-10816801

Team name: nevermind

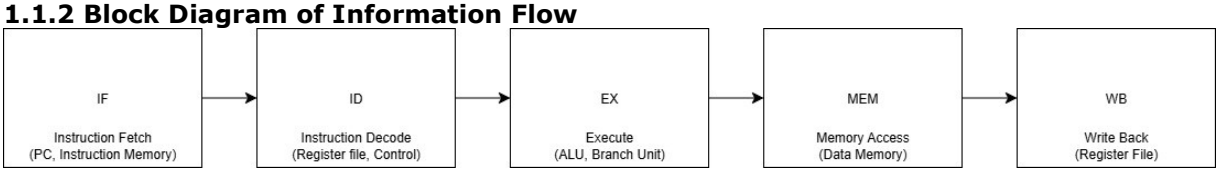
1 Architecture Specification: Written Spec

1.1 Functionality / feature

1.1.1 Key Features

1. Support 32-bit instructions
2. Contains a register file of 32 general-purpose registers (GPRs) and 6 special-purpose registers
3. 5-Stage pipeline (Instruction Fetch - IF, Instruction Decode - ID, Execute - EX, Memory Access - MEM, Write Back - WB)
4. Preform arithmetic and bitwise operations such as: Addition, Subtraction, Multiplication and also AND, OR, XOR, NOT.
   - **Arithmetic Operations:** add, addi, sub, addiu
   - **Logical Operations:** and, andi, or, ori, nor, xor
   - **Data Transfer:** lw, sw, lb, sb
   - **Control Flow:** beq, bne, j, jal
   - **Comparison:** slt, slti
   - **Shift Operations:** sll, srl, sra
   - **Multiplication:** multu, mflo, mfhi
5. Perform bit-level shift and rotate operations.
6. Supports loading and storing of words and bytes.
7. Supports conditional and unconditional control flow changes.
8. Perform set-on-less-than instructions for signed and immediate values.
9. Additional instruction group of sra and lui.

1.1.2 Block Diagram of Information Flow



1.2 Operating Procedures and Application

1.2.1 Programming Mode

1. Reset the processor (reset=1 for a few cycles, then reset=0).
2. Instruction memory is loaded with the program (in MIPS machine code).
3. This can be done by:
   - Preloading .text segment into instruction memory.
   - Setting program counter (PC) to start address (usually 0x00000000).
1. Data memory is initialized with required values.
2. HI and LO registers are cleared before multiplication operations.
3. Set control signals for testbench or external loader:
   - clk – driven by system clock.
   - reset – clears pipeline and registers.
   - start

1.2.2 Normal Mode

1. Processor fetches instructions sequentially from Instruction Memory.
2. Pipeline executes in 5 stages:

- IF – Fetch instruction from instruction memory.
- ID – Decode instruction, fetch operands from register file.
- EX – Execute ALU / Multiplier operation.
  - If multu, multiplier unit runs until completion, result stored in HI/LO.
- MEM – Perform memory read/write (for lw, sw, etc.).

- WB – Write results back to register file.

3. Program runs continuously until:
   - A branch (beq, bne, j, jal) changes PC.
   - The instruction sequence ends (simulation stop or HALT pseudo-instruction).

1.3 Naming convention

1.3.1 Port Name

<Direction><Level><Module Name><Function><Port Size>

1.3.2 Module Name

- ins_mem (memory unit)
- mctrl (control unit- main control block)
- reg (CPU unit- register file block)
- aluctrl (control unit- ALU control block)
- mult (CPU-unit- multiplier)
- alu (CPU unit- ALU block)
- data_mem (memory unit)

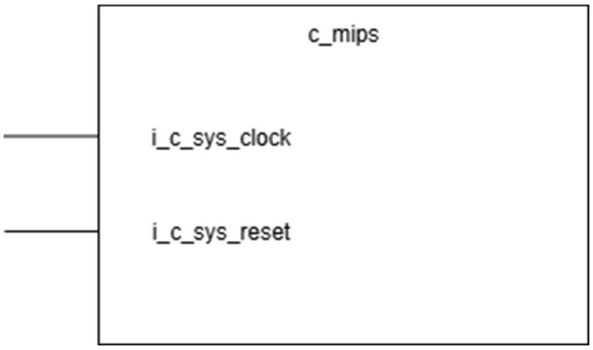
1.3.3 Internal Signal and Structure Name

Name	Name used
input	i
output	o
unit level	u
block level	b
system	sys
pipeline	p
processor	pcs
main	m
control	ctrl
arithmetic and logic unit	alu
enable	en
register	reg
opcode	op
program counter	pc
source	src
write	wr
memory	mem
extend	ext
jump	jmp
multiply	mult
function	funct
Jump and link	jal
instruction	ins

Table 1.3: Naming Convention of the pipeline processor chip

1.4 Pipeline Chip Interface and I/O Pin Description

1.4.1 Block Interface



1.4.2 I/O Pin Description

Pin name: **Pin class:** **Pin function:**	i_c_sys_clock control To provide a clock signal to the system	**Source** **→Destination:**	Clock Generator → CPU
Pin name: **Pin class:** **Pin function:**	i_c_sys_reset control To reset the clock	**Source** **→Destination:**	System reset → CPU

Table 1.4.2: I/O pin description

1.4.3 Timing diagram

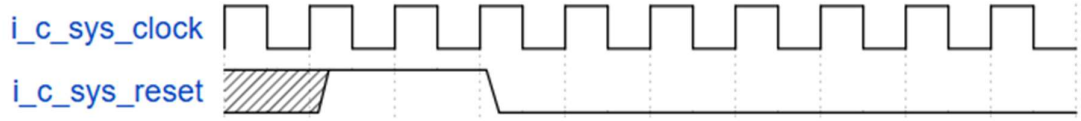
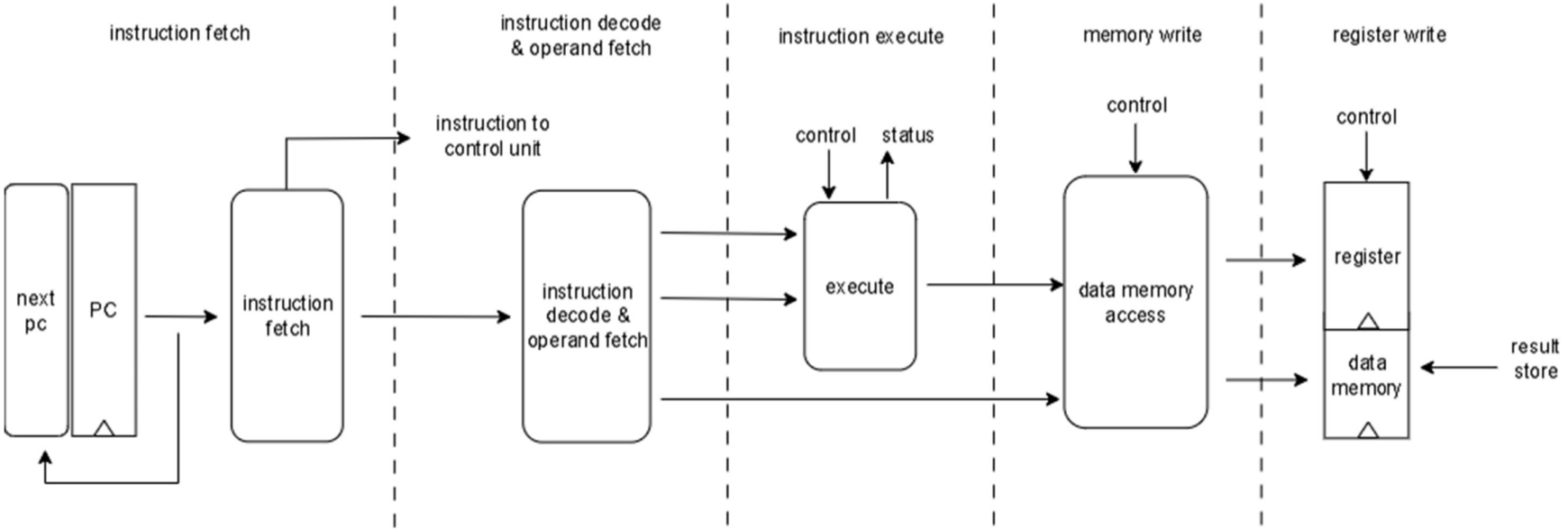


Diagram 1.4.3: Timing diagram

1.5 Internal Operation1.5.1 Logical View



1.5.2 5-Stages Description

1. Instructions Fetch

    - **Action:** The processor fetch the next instruction from the memory location pointed to by the Program Counter (PC).

    - **Goal:** To get the instruction from the Text Segment of memory into the processor.

2. Instruction Decode and Data Fetch

    - **Action:** The processor decodes the instruction to understand what operation to perform (e.g., add, load, branch) and identifies the source registers it needs. Simultaneously, it fetches the values from those registers.

    - **Goal:** To figure out what the instruction means and get the necessary data from the Register File.

3. Execute

    - **Action:** The Arithmetic Logic Unit (ALU) and Multiplier performs the actual calculation. This could be an arithmetic operation (like addition or subtraction), a logical operation (like AND or OR), or a memory address calculation for load/store instructions.

    - **Goal:** To perform the main operation of the instruction.

4. Memory Write

    - **Action:** This stage is only active for load and store instructions. For a store instruction (sw,sb), the processor writes data from a register to memory. For a load instruction (lw,lb), it reads data from memory.

    - **Goal:** To read data from or write data to the Data or Stack Segment of memory.

5. Register Write

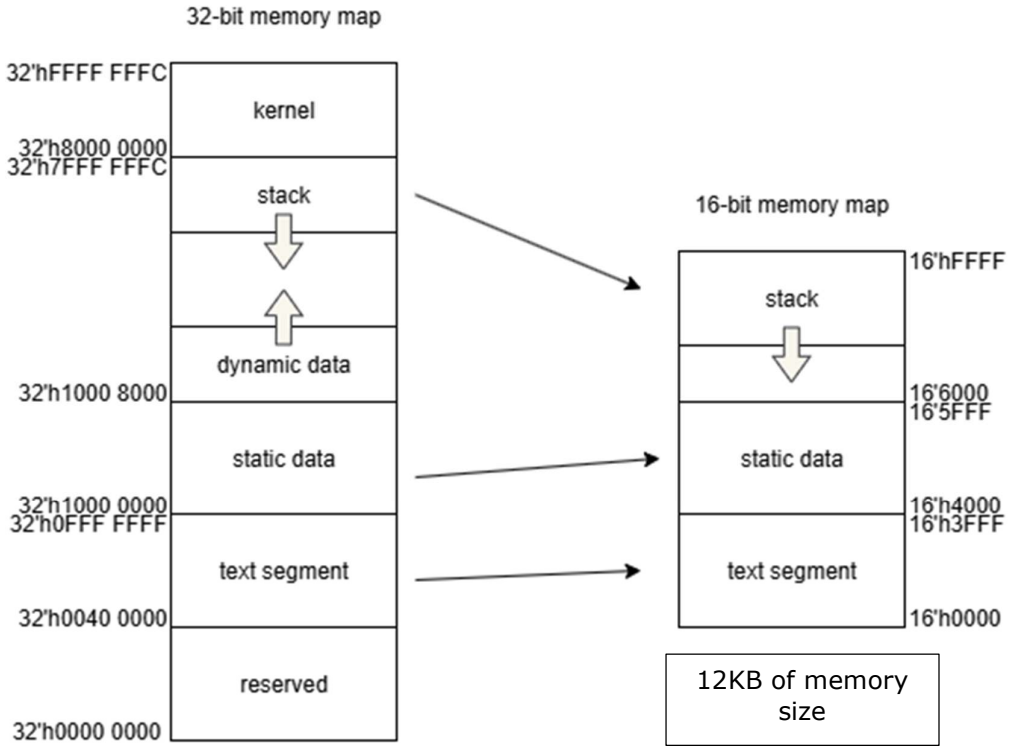
    - **Action:** The result of the operation is written back into the destination register.

    - **Goal:** To store the final result of the instruction in the Register File.

1.5.3 use of major component

Stage	Component
Instruction fetch	Instruction memory Program Counter
Instruction decode and data fetch	Main control Alu control Register
Excute	ALU Multiplier
Memory write	Data memory
Regsiter write	Register

1.6 memory map



Segment	Virtual Memory, v	Physical Memory,p[15:12] {v[29:28],v[15:14]}	v[11:0]	p[15:0]
Text segment	32'h0040_0000	0000 (4'h0)	3'h000	16'h0000
Static data	32'h1000_0000	0100 (4'h4)	3'h000	16'h4000
Stack (base)	32'h1000_8000	0110 (4'h6)	3'h000	16'h6000
Stack (top)	32'hFFFF_FFFC	1111 (4'hF)	3'hFFC	16'hFFFF

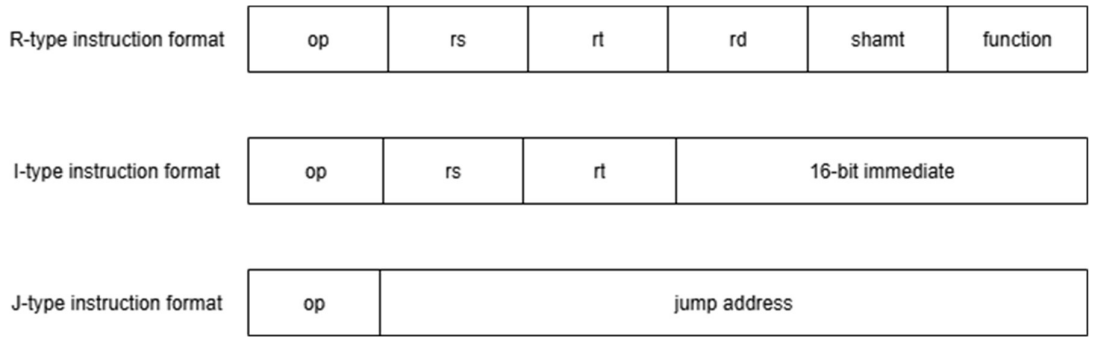
1.6 System register

Register name	Register No.	Width	Reset value	Function	Preserved on call?
$zero	$0	32-bit	32'h0000_0000	Constant zero	-
$at	$1			Reserved for assembler	-
$v0-$v1	$2-$3			Return values from function	No
$a0-$a3	$4-$7			Arguments	Yes
$t0-$t7	$8-$15			Hold temporary value	No
$s0-$s7	$16-$23			Hold long-lived value	yes
$t8-$t11	$24-$27			More temporaries	No
$gp	$28		32'h1000_4000	Point to the middle of memory	Yes
$sp	$29		32'h7FFF_FFFC	Stack pointer	yes
$fp	$30		32'h0000_0000	Frame pointer	Yes
$ra	$31		32'h0000_0000	Return memory	yes

Special Register

Register name	Width	Reset value	Function
HI	32-bit	0	store upper part of multiplier result
LO		0	store lower part of multiplier result
PC		0	pointer of intruction memory

1.7 Instruction Set, Machine Language,RTN, Instruction Format, Addressing mode



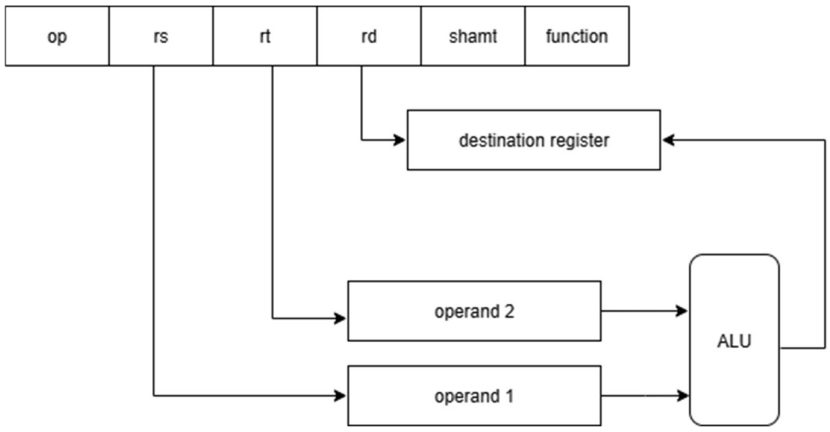
Instruction Set	Instruction Format	Machine Language	Register Transfer Node	Addressing Mode
add rd, rs, rt	R-Type	000000 [xxxxx] [xxxxx] [xxxxx] 00000 100000	reg[rd] ← reg[rs] + reg[rt] pc ← pc +4	Register addressing
sub rd, rs, rt	R-Type	000000 [xxxxx] [xxxxx] [xxxxx] 00000 100010	reg[rd] ← reg[rs] - reg[rt] pc ← pc +4	Register addressing
and rd, rs, rt	R-Type	000000 [xxxxx] [xxxxx] [xxxxx] 00000 100100	reg[rd] ← reg[rs] & reg[rt] pc ← pc +4	Register addressing
or rd, rs, rt	R-Type	000000 [xxxxx] [xxxxx] [xxxxx] 00000 100101	reg[rd] ← reg[rs] | reg[rt] pc ← pc +4	Register addressing
nor rd, rs, rt	R-Type	000000 [xxxxx] [xxxxx] [xxxxx] 00000 100111	reg[rd] ← ~(reg[rs] | reg[rt]) pc ← pc +4	Register addressing
xor rd, rs, rt	R-Type	000000 [xxxxx] [xxxxx] [xxxxx] 00000 100110	reg[rd] ← reg[rs] ^ reg[rt] pc ← pc +4	Register addressing
slt rd, rs, rt	R-Type	000000 [xxxxx] [xxxxx] [xxxxx] 00000 101010	reg[rd] ← (rs<rt)? 1: 0 pc ← pc +4	Register addressing

sll rd, rt, shamt	R-Type	000000 00000 [xxxxx] [xxxxx] [xxxxx] 000000	reg[rd] ← reg[rs] << shamt pc ← pc +4	Register addressing
srl rd, rt, shamt	R-Type	000000 00000 [xxxxx] [xxxxx] [xxxxx] 000010	reg[rd] ← reg[rs] >> shamt pc ← pc +4	Register addressing
multu rs, rt	R-Type	000000 [xxxxx] [xxxxx] 00000 00000 011001	temp[63:0] ← reg[rs] * reg[rt] hi ← temp[63:32] lo ←temp[31:0] pc ← pc +4	Register addressing
mfhi rd	R-Type	000000 00000 00000 [xxxxx] 00000 010000	reg[rd] ← hi pc ← pc +4	Register addressing
mflo rd	R-Type	000000 00000 00000 [xxxxx] 00000 010010	reg[rd] ← lo pc ← pc +4	Register addressing
jr rs	R-Type	000000 [xxxxx] 00000 00000 00000 001000	pc ← reg[rs]	Register indirect addressing
jalr rs, rd	R-Type	000000 [xxxxx] 00000 [xxxxx] 00000 001001	reg[rd] ← pc + 4 pc ← reg[rs]	Register indirect addressing
addi rt, rs, imm	I-Type	001000 [xxxxx] [xxxxx] [xxxx xxxx xxxx xxxx]	reg[rt] ← reg[rs] + sign-extended imm pc ← pc +4	Immediate addressing
andi rt, rs, imm	I-Type	001100 [xxxxx] [xxxxx] [xxxx xxxx xxxx xxxx]	reg[rt] ← reg[rs] & zero-extended imm	Immediate addressing
ori rt, rs, imm	I-Type	001101 [xxxxx] [xxxxx] [xxxx xxxx xxxx xxxx]	reg[rr] ← reg[rs] | zero-extended imm	Immediate addressing
slti rt, rs, imm	I-Type	001010 [xxxxx] [xxxxx] [xxxx xxxx xxxx xxxx]	reg[rd] ← (rs< sign-extended imm)? 1: 0	Immediate addressing
lui rt, imm	I-Type	001111 00000 [xxxxx] [xxxx xxxx xxxx xxxx]	reg[rt] ← imm << 16 pc ← pc +4	Immediate addressing
lw rt, imm(rs)	I-Type	100011 [xxxxx] [xxxxx] [xxxx xxxx xxxx xxxx]	reg[rt] ← mem[rs+ sign-extended imm] pc ← pc +4	Base addressing

sw rt, imm(rs)	I-Type	101011 [xxxxx] [xxxxx] [xxxx xxxx xxxx xxxx]	mem[rs+ sign-extended imm] ← reg[rt] pc ← pc +4	Base addressing
lb rt, imm(rs)	I-Type	100000 [xxxxx] [xxxxx] [xxxx xxxx xxxx xxxx]	reg[rt] ← mem[rs+ sign-extended imm] pc ← pc +4	Base addressing
sb rt, imm(rs)	I-Type	101000 [xxxxx] [xxxxx] [xxxx xxxx xxxx xxxx]	mem[rs+ sign-extended imm] ← reg[rt] pc ← pc +4	Base addressing
beq rt, rs, offset	I-Type	000100 [xxxxx] [xxxxx] [xxxx xxxx xxxx xxxx]	pc ← (rs == rt)? pc+4 + offset||00: pc +4	PC-Relative addressing
bne rt, rs, offset	I-Type	000101 [xxxxx] [xxxxx] [xxxx xxxx xxxx xxxx]	pc ← (rs != rt)? pc+4 + offset||00 : pc +4	PC-Relative addressing
j jump_addr	J-Type	000010 [xx xxxx xxxx xxxx xxxx xxxx xxxx]	pc ← (pc+4)[31:28]||jump_addr||00	Pseudo-direct addressing
jal jump_addr	J-Type	000011 [xx xxxx xxxx xxxx xxxx xxxx xxxx]	reg[31] ← pc + 4 pc ← (pc+4)[31:28]||jump_addr||00	Pseudo-direct addressing

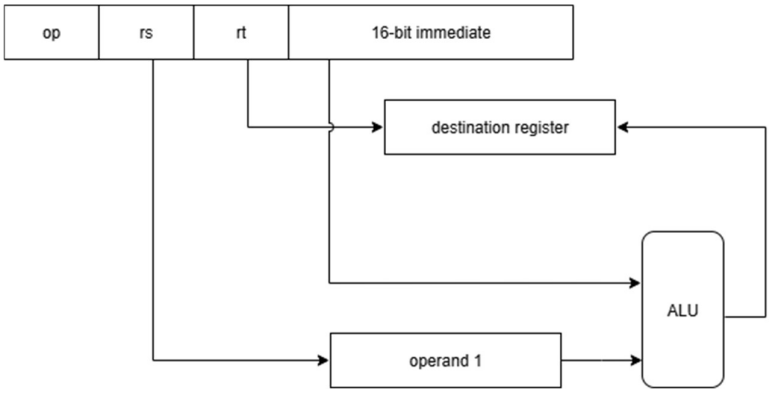
1.8 Addressing Modes

1.8.1 Register Addressing Mode



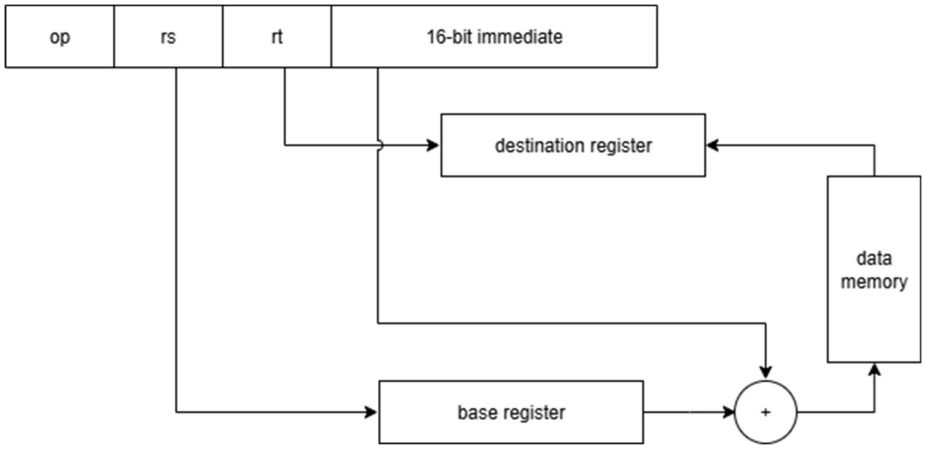
- The operand is located in register
- Eg. add $t0, $s1, $s2, operand is in $s1, $s2

1.8.2 Immediate Addressing Mode



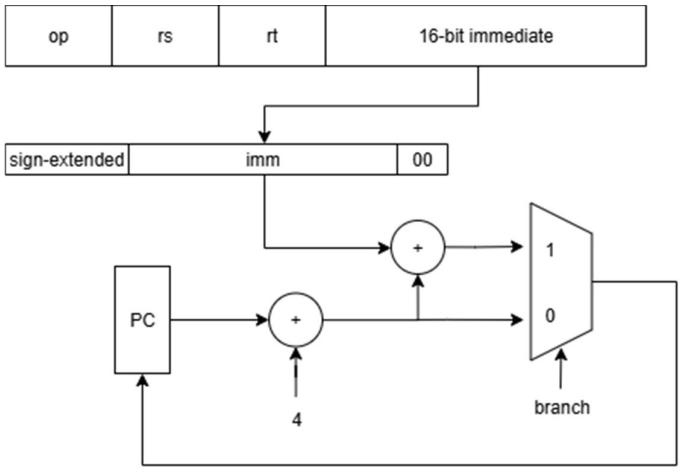
- The operand is a constant value contained within the instruction itself.
- Zero-extending when logical immediate instructions
- Sign-extending when arithmetic and program control immediate instructions

1.8.3 Base Addressing Mode



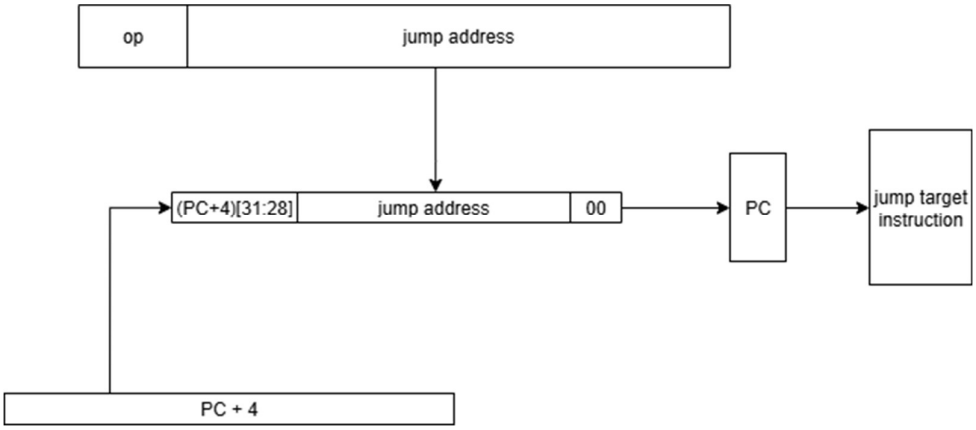
- The memory address is calculated by adding a sign-extended constand (from instruction) and the value from base register

1.8.4 PC-relative Addressing Mode



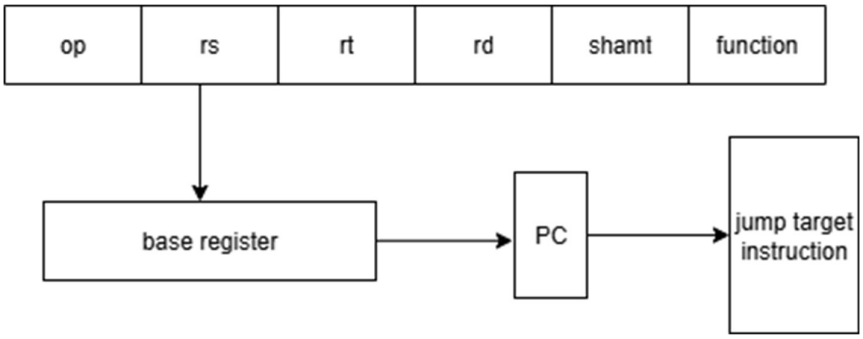
- The target address is calculated by adding an offset from the instruction and logic shifted left by 2
- If branch is taken PC<= PC+4 + {sign-extend imm,00}, else PC <= PC+4

1.8.5 Pseudo-direct Addressing Mode



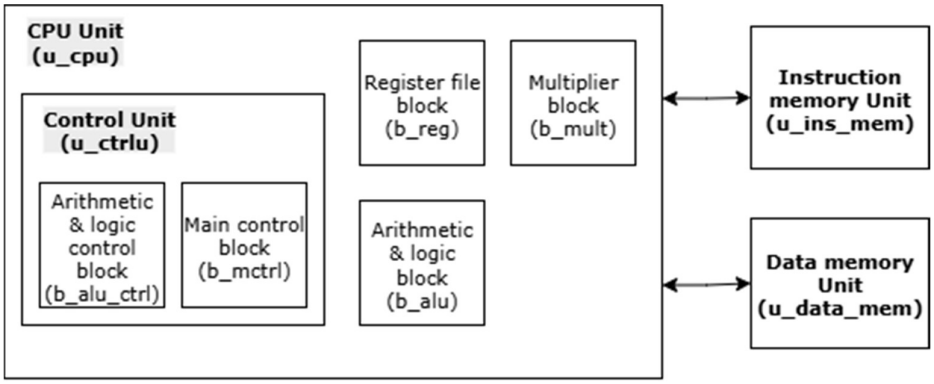
- The target address for a jump is formed by concatenating the upper bits of the PC with a 26-bit address from the instruction (shift left by 2)

1.8.6 Register Indirect Addressing Mode



- Targeted address is in the full 32-bit value is in the register

2 Micro-Architecture Specification



2.1 Design hierarchy

Diagram 2.1: Design hierarchy of the pipeline processor

Top Level at System Level (Chip Partitioning)	Micro-Architecture Level	
	Unit partitioning	Block partitioning
c_mips	u_ctrlu	b_mctrl
		b_aluctrl
	u_cpu	b_reg
		b_alu
		b_mult
	u_ins_mem	
	u_data_mem	

Table 2.1: Design hierarchy of the pipeline processor

2.2 Unit Level Brief Functional Description

Name	Function
u_cpu	• Decode and execute the instructions received • Receive data from memory and store in register file to be used
u_mem	• Fetch the instructions to be decode from instruction memory • Read data from data memory • Write data to data memory
u_ins_mem	• Decode opcode and function from instructions to several top-level control signals or opcode for the pipeline.
u_data_mem	• Read data from data memory • Write data to data memory

Table 2.1.1: Unit Level Brief Functional Description of the pipeline procesor

2.3 Pre-synthesis Unit-Level Schematic

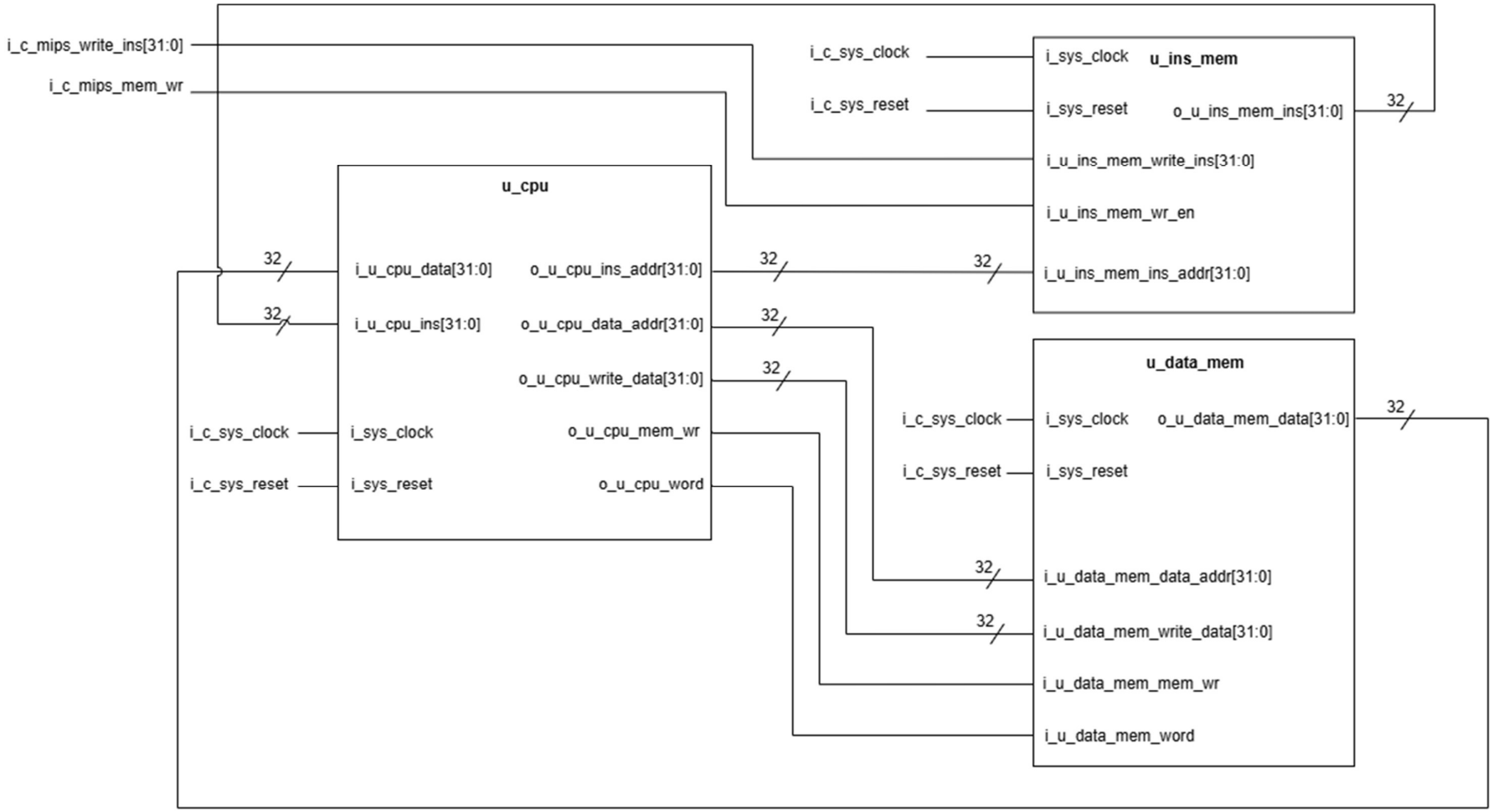


Diagram 2.3 Pre-synthesis Unit-Level Schematic diagrap of MIPS ISA 32 bit pipeline processor

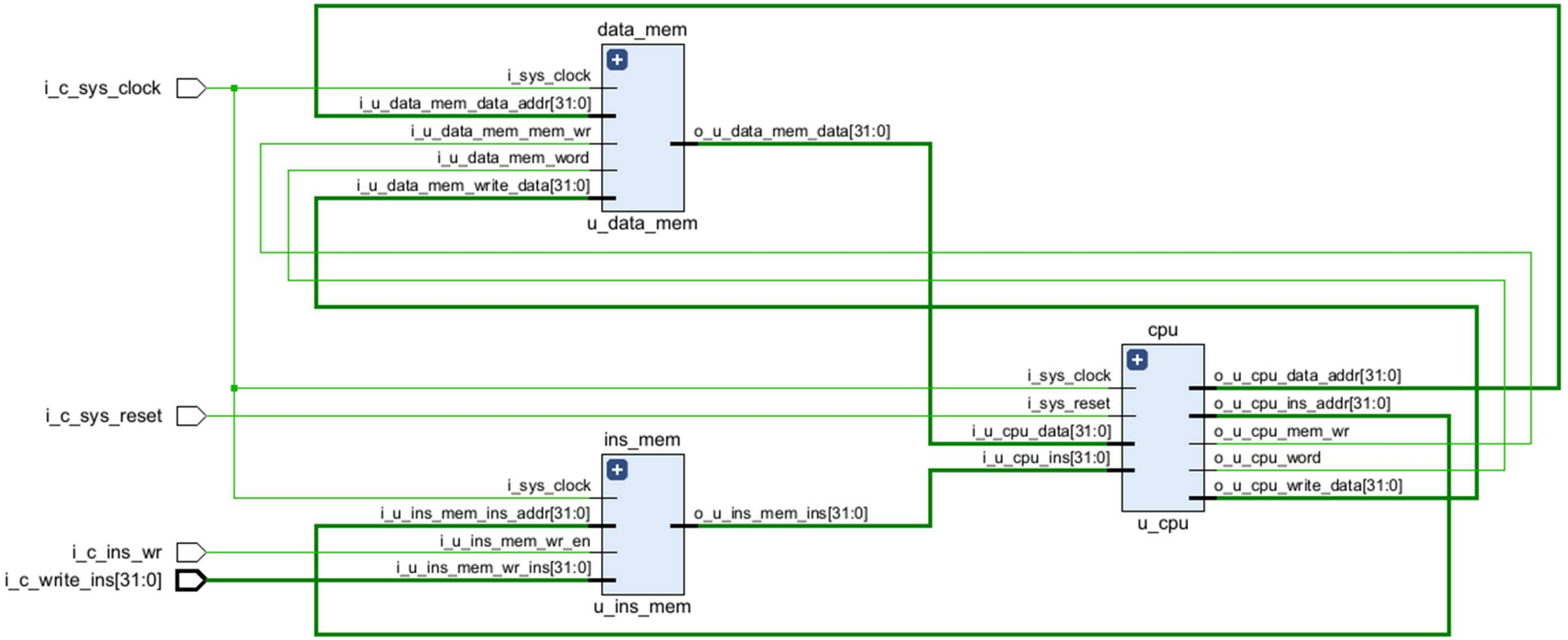


Diagram 2.3.1 Post-synthesis Unit-Level Schematic diagrap of MIPS ISA 32 bit pipeline processor

2.4 Full Chip Verilog Model

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Ng Yu Heng
//
// Create Date: 07.09.2025 13:45:34
// File Name: c_mips.sv
// Module Name: c_mips
// Project Name: MIPS ISA Pipeline processor
// Code Type: RTL level
// Description: Modeling of Mips pipeline processor chip block
//
//////////////////////////////////////////////////////////////////////////////////

module c_mips(
    input logic i_c_sys_reset,
    input logic i_c_sys_clock,
    input logic [31:0] i_c_write_ins,
    input logic i_c_ins_wr
    );

    // Internal wires connecting CPU & instruction memory
    logic [31:0] cpu_ins_addr;
    logic [31:0] ins_mem_ins;


    // Internal wires connecting CPU &* data memory
    logic [31:0] cpu_data_addr;
    logic [31:0] cpu_write_data;
    logic        cpu_mem_wr;
    logic        cpu_word;
    logic [31:0] data_mem_data;


    u_cpu cpu (
        .i_sys_clock          (i_c_sys_clock),
        .i_sys_reset          (i_c_sys_reset),
        .i_u_cpu_data           (data_mem_data),
        .i_u_cpu_ins           (ins_mem_ins),
        .o_u_cpu_ins_addr       (cpu_ins_addr),
        .o_u_cpu_data_addr       (cpu_data_addr),
        .o_u_cpu_write_data      (cpu_write_data),
        .o_u_cpu_mem_wr         (cpu_mem_wr),
        .o_u_cpu_word           (cpu_word)
    );

    u_ins_mem ins_mem (
        .i_sys_clock          (i_c_sys_clock),
        .o_u_ins_mem_ins      (ins_mem_ins),
        .i_u_ins_mem_ins_addr   (cpu_ins_addr),
        .i_u_ins_mem_wr_ins     (i_c_write_ins),
        .i_u_ins_mem_wr_en      (i_c_ins_wr)
    );

    u_data_mem data_mem (
        .i_sys_clock            (i_c_sys_clock),
```

```
        .o_u_data_mem_data       (data_mem_data),
        .i_u_data_mem_data_addr  (cpu_data_addr),
        .i_u_data_mem_write_data (cpu_write_data),
        .i_u_data_mem_mem_wr     (cpu_mem_wr),
        .i_u_data_mem_word       (cpu_word)
    );

endmodule
```

3 Architecture Specification: Verification Spec

3.1 Test plan

No.	Test Case	Description of test vector Generation	Expected Output	Status
1	**add (corner case)**	**ADD to Max value (2^{31})** li $t2, 0x7FFFFFFE li $t3, 1 nop nop nop add $t4, $t2, $t3	$t4 = 0x7FFFFFF	PASS

Assembly code		Hex Code
lui $1, 32767	# li $t2, 0x7FFFFFFE	3c017fff
nop x3		00000000(3)
ori $10, $1, -2		342afffe
ori $11, $0, 1	# li $t3, 1	340b0001
nop x3		00000000(3)
add $12, $10, $11	# add $t4, $t2, $t3	014b6020

No.	Test Case	Description of test vector Generation	Expected Output	Status
2	**addi (corner case)**	**Add max immediate value** addi $t1, $zero, 32767 nop nop nop	$t1 = 32767(7fff)	PASS

Assembly code		Hex Code
addi $9, $0, 32767	# addi $t1, $zero, 32767	20097fff
nop		00000000
nop		00000000
nop		00000000

No.	Test Case	Description of test vector Generation	Expected Output	Status
3	**multu (corner case)**	**Max unsigned × max unsigned** li $t0, 0xFFFFFFFF li $t1, 0xFFFFFFFF nop nop nop multu $t0, $t1 nop x 32	HI= 0xFFFFFFFE LO= 0x00000001	PASS

Assembly code		Hex Code
lui $1, -1	# li $t0, 0xFFFFFFFF	3c01ffff
nop		00000000
nop		00000000
nop		00000000
ori $8, $1, -1		3428ffff

lui $1, -1	# li $t1, 0xFFFFFFFF	3c01ffff
nop		00000000
nop		00000000
nop		00000000
ori $9, $1, -1		3429ffff
nop		00000000
nop		00000000
nop		00000000
multu $8, $9	# multu $t0, $t1	01090019
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000
nop		00000000

No.	Test Case	Description of test vector Generation	Expected Output	Status
4	**mfhi (corner case)**	**Move from HI after large mult** (continue from test plan 3) mfhi $t2	$t2 = 0xFFFFFFFE	PASS

Assembly code	Hex Code
(continue from test plan 3) mfhi $10 # mfhi $t2	 00005010

No.	Test Case	Description of test vector Generation	Expected Output	Status
5	**sw (corner case)**	**Store large word** .data store_word: .space 0 .text li $t1, 0xFEDCBA98 la $t0, store_byte nop nop nop sw $t1, 12($t0)	Memory[3] = 0xFEDCBA98	PASS

Assembly code		Hex Code
lui $1, -292	# li $t1, 0xFEDCBA98	3c01fedc
nop		00000000
nop		00000000
nop		00000000
ori $9, $1, -17768		3429ba98
lui $1, 0 [store_byte]	# la $t0, store_byte	3c010000
nop		00000000
nop		00000000
nop		00000000
ori $8, $1, 0 [store_byte]		34280000
nop		00000000
nop		00000000
nop		00000000
sw $9, 12($8)	# sw $t1, 12($t0)	ad09000c

No.	Test Case	Description of test vector Generation	Expected Output	Status
6	**lb (corner case)**	**Load byte [1] from a memory space ([15:8] lower bits)** (continue from previous test case) lb $t2, 13($t0)	$t2 = 0xBA = 8'b10111010	PASS

Assembly code		Hex Code
(continue from previous test case) lb $10, 13($8)	# lb $t2, 13($t0)	810a000d

No.	Test Case	Description of test vector Generation	Expected Output	Status
7	**beq (corner case)**	**Branch if -1 == -1** li $t3, 0xFFFFFFFF nop nop nop beq $t3, $t4, equal2 nop nop	$t5 not set $t6 = 1	PASS

		nop li $t5, 9 equal2: li $t5, 1		

Assembly code		Hex Code
lui $1, -1	# li $t3, 0xFFFFFFFF	3c01ffff
nop		00000000
nop		00000000
nop		00000000
ori $11, $1, -1		342bffff
lui $1, -1	# li $t4, 0xFFFFFFFF	3c01ffff
nop		00000000
nop		00000000
nop		00000000
ori $12, $1, -1		342cffff
nop		00000000
nop		00000000
nop		00000000
beq $11, $12, 12 [equal2-0x0040003c]	# beq $t3, $t4, equal	116c0005
nop	# solve ctrl hazard	00000000
nop		00000000
nop		00000000
ori $13, $0, 9	# li $t5, 9	340d0009
ori $14, $0, 1	# equal2: li $t6, 1	340e0001

No.	Test Case	Description of test vector Generation	Expected Output	Status
8	**j**	**Jump over instruction** j jump_target li $t0, 9 jump_target: li $t1, 1 #(0x00400012)	$t0 not set, $t1 = 1	PASS

Assembly code		Hex Code
j 0x00400012 [jump_target]	# j jump_target	08040004
ori $8, $0, 9	# li $t0, 9	34080009
ori $9, $0, 1	# li $t1, 1 (jump_target)	34090001

No.	Test Case	Description of test vector Generation	Expected Output	Status
9	**jr**	**Jump to register address** la $t0, target_label nop nop nop jr $t0 nop li $t1, 9 target_label: li $t2, 1	$t1 not set, $t2 = 1	PASS

Assembly code		Hex Code
lui $1, 64 [target_label]	# la $t0, target_label	3c010040

nop		00000000
nop		00000000
nop		00000000
ori $8, $1, 64 [target_label]		34280040
nop		00000000
nop		00000000
nop		00000000
jr $8	# jr $t0	01000008
nop		00000000
ori $9, $0, 9	# li $t1, 9	34090009
ori $10, $0, 1	# li $t2, 1	340a0001

No.	Test Case	Description of test vector Generation	Expected Output	Status

10	**Basic Conditional Branching (if-else)**	**If a>b,then c=100, else c=200**
		addi $s0, $zero, 15
		addi $s1, $zero, 10
		
		slt $t0, $s1, $s0
		beq $t0, $zero, ELSE
		addi $s2, $zero, 100
		j EXIT
		
		ELSE:
		addi $s2, $zero, 200
		
		EXIT:
			$s2=100	PASS

Assembly code		Hex Code
addi $16, $0, 15	# addi $s0, $zero, 15	2010000f
addi $17, $0, 10	# addi $s1, $zero, 10	2011000a
nop		00000000
nop		00000000
nop		00000000
slt $8, $17, $16	# slt $t0, $s1, $s0	0230402a
nop		00000000
nop		00000000
nop		00000000
beq $8, $0, 12 [ELSE-0x00400030]		11000003
addi $18, $0, 100	# addi $s2, $zero, 100	20120064
j 0x00400040 [EXIT]	# j EXIT	08100010
nop		00000000
addi $18, $0, 200	# addi $s2, $zero, 200	201200c8
	#EXIT:	

No.	Test Case	Description of test vector Generation	Expected Output	Status

11	**Loop Structure (beq,slt,j)**	**Sum of integers from 1 to 5**
		addi $s0, $zero, 0
		addi $s1, $zero, 5
		nop
		nop
		nop
		
		LOOP:
		slti $t0, $s1, 1
		nop
		nop
		nop
		slti $t0, $s1, 1
		nop
		nop
		nop
		beq $t0, $zero, DO_ADD
		nop
		nop
		j END
		nop
		
		DO_ADD:
		add $s0, $s0, $s1
		addi $s1, $s1, -1
		nop
		nop
		nop
		j LOOP
		nop
		
		END:
			$s0=15	PASS

Assembly code		**Hex Code**
addi $16, $0, 0	# addi $s0, $zero, 0	20100000
addi $17, $0, 5	# addi $s1, $zero, 5	20110005
nop	# nop	00000000
nop	# nop	00000000
nop	# nop	00000000
slti $8, $17, 1	# slti $t0, $s1, 1	2a280001
nop	# nop	00000000
nop	# nop	00000000
nop	# nop	00000000
slti $8, $17, 1	# slti $t0, $s1, 1	2a280001
nop	# nop	00000000
nop	# nop	00000000
nop	# nop	00000000
beq $8, $0, 20 [DO_ADD-current address]		11000005
nop	# nop	00000000
nop	# nop	00000000
j 0x00400060 [END]	# j END	08100018
nop	# nop	00000000
add $16, $16, $17	# add $s0, $s0, $s1	02118020
addi $17, $17, -1	# addi $s1, $s1, -1	2231ffff
nop	# nop	00000000
nop	# nop	00000000

nop # nop j 0x00400010 [LOOP] # j LOOP nop	00000000 08100004 00000000

No.	Test Case	Description of test vector Generation	Expected Output	Status

12	**Nested Structure**	**Counts the total number of iterations**
		add $s0, $zero, $zero
		addi $s1, $zero, 0
		
		OUTER_LOOP:
		slti $t0, $s1, 2
		beq $t0, $zero, DONE
		addi $s2, $zero, 0
		
		INNER_LOOP:
		slti $t1, $s2, 3
		beq $t1, $zero, INCR_I
		addi $s0, $s0, 1
		addi $s2, $s2, 1
		j INNER_LOOP
		
		INCR_I:
		addi $s1, $s1, 1
		j OUTER_LOOP
		
		DONE:
			$s0=6	PASS

Assembly code	Hex Code

add $16, $0, $0	# add $s0, $zero, $zero	20100000
addi $17, $0, 0	# addi $s1, $zero, 0	20110000
nop		00000000
nop		00000000
nop		00000000
slti $8, $17, 2	# slti $t0, $s1, 2	2a280002
nop		00000000
nop		00000000
nop		00000000
beq $8, $0, 36 [DONE-0x00400030]		11000008
nop		00000000
nop		00000000
addi $18, $0, 0	# addi $s2, $zero, 0	20120000
slti $9, $18, 3	# slti $t1, $s2, 3	2a490003
nop		00000000
nop		00000000
nop		00000000
beq $9, $0, 16 [INCR_I-0x0040003c]		11200005
nop		00000000
nop		00000000
addi $16, $16, 1	# addi $s0, $s0, 1	22100001
addi $18, $18, 1	# addi $s2, $s2, 1	22520001
nop		00000000
nop		00000000
nop		00000000
j 0x00400038 [INNER_LOOP]	# j INNER_LOOP	08100009
nop		00000000
addi $17, $17, 1	# addi $s1, $s1, 1	22310001
j 0x0040002c [OUTER_LOOP]	# j OUTER_LOOP	08100002
nop		00000000

No.	Test Case	Description of test vector Generation	Expected Output	Status

13	**Procedure Call**	**Double the value**
		addi $a0, $zero, 21
		jal double
		add $s0, $v0, $zero
		j end
		
		double:
		add $v0, $a0, $a0
		jr $ra
		
		end:
		j end
			$s0=42	PASS

Assembly code	Hex Code

addi $4, $0, 21	# addi $a0, $zero, 21	20040015
nop		00000000
nop		00000000
nop		00000000
jal 0x00400028 [double]	# jal double	0C00000A
nop		00000000
add $16, $2, $0	# add $s0, $v0, $zero	00508020
nop		00000000
j 0x00400044 [end]	# j end	08100011
nop		00000000
add $2, $4, $4	# add $v0, $a0, $a0	00841020
nop		00000000
nop		00000000
jr $31	# jr $ra	03e00008
nop		00000000
j 0x00400044 [end]	# j end	08100011
nop		00000000
	#end:	

No.	Test Case	Description of test vector Generation	Expected Output	Status
14	**Integration test program**	**findMax (int x, int y, int z){** **int max = x;** **if (y > max)** **max = y;** **if (z > max)** **max = z;** **return max;** **}** //to test jal and jr $t0 = findMax(4,5,6)	$t0 = 6	PASS

Assembly code	Hex Code
main:	
# $t0 = findMax(4, 5, 6)	
addi $a0, $zero, 4 # $a0 = 4	20040004
addi $a1, $zero, 5 # $a1 = 5	20050005
addi $a2, $zero, 6 # $a2 = 6	20060006
jal findMax # Jump and link to the findMax function	0c10000b
nop	00000000
add $t0, $v0, $zero	00404020
nop	00000000

nop	00000000
nop	00000000
nop	00000000
nop #wait to store	00000000
# findMax (int x, int y, int z)	
# Arguments: $a0 (x), $a1 (y), $a2 (z)	
# Return value: $v0	
findMax:	
# int max = x;	
add $v0, $a0, $zero # 'move $v0, $a0'	00801020
nop	00000000
nop	00000000
nop	00000000
# if (y > max)	
slt $t0, $v0, $a1 # Set less than: $t0 = 1 if $v0 < $a1 (y > max)	
nop	0045402a
nop	00000000
nop	00000000
	00000000
beq $t0, $zero, check_z # Branch to check_z if $t0 is not equal to zero	
nop	11000007
nop	00000000
nop	00000000
	00000000
# max = y;	
add $v0, $a1, $zero # 'move $v0, $a1'	
nop	00a01020
nop	00000000
nop	00000000
	00000000
check_z:	
# if (z > max)	
slt $t0, $v0, $a2 # Set less than: $t0 = 1 if $v0 < $a2 (z > max)	
nop	0046402a
nop	00000000
nop	00000000
	00000000
beq $t0, $zero, end_func # Branch to end_func if $t0 is not equal to zero	11000004
nop	
nop	00000000
nop	00000000
	00000000
# max = z;	
add $v0, $a2, $zero # 'move $v0, $a2'	
	00c01020
end_func:	
jr $ra # Jump register to the return address	
nop	03e00008
	00000000

3.2 Testbench and Simulation result

3.2.1 Testbench

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Ng Yu Heng, Goh Xuan Hui, Chow Bin Lin
//
// Create Date: 07.09.2025 16:51:54
// File Name: tb_c_mips.sv
// Module Name: tb_c_mips
// Project Name: MIPS ISA Pipeline processor
// Code Type: Behavioural
// Description: Testbench for MIPS ISA Pipeline processor Chip
//////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////
//Please insert the hex file into "\projectName.sim\sim_1\behav\xsim\testcases"
//before running the tb
//and
//change the simulation hex file when running the tb by changing
//"$readmemh(testcaseX,mem);"
//////////////////////////////////////////////////////////////////////////////////


module tb_c_mips();

    // Inputs
    logic tb_i_c_sys_reset;
    logic tb_i_c_sys_clock;
    logic [31:0] tb_i_c_write_ins;
    logic tb_i_c_ins_wr;
    logic [6:0] i;


    // Clock and cycle definition
    parameter CC = 10; // 10ns per clock cycle

    // for memory write
    parameter testcase1 = "testcases/testcase1.hex";
    parameter testcase2 = "testcases/testcase2.hex";
    parameter testcase3 = "testcases/testcase3.hex";
    parameter testcase4 = "testcases/testcase4.hex";
    parameter testcase5 = "testcases/testcase5.hex";
    parameter testcase6 = "testcases/testcase6.hex";
    parameter testcase7 = "testcases/testcase7.hex";
    parameter testcase8 = "testcases/testcase8.hex";
    parameter testcase9 = "testcases/testcase9.hex";
    parameter testcase10 = "testcases/testcase10.hex";
    parameter testcase11 = "testcases/testcase11.hex";
    parameter testcase12 = "testcases/testcase12.hex";
    parameter testcase13 = "testcases/testcase13.hex";
    parameter testcase14 = "testcases/testcase14.hex";
    logic [31:0] mem[63:0]; // 64 lines instruction might enough

    // Instantiate DUT
    c_mips dut (
        .i_c_sys_reset(tb_i_c_sys_reset),
```

```
        .i_c_sys_clock(tb_i_c_sys_clock),
        .i_c_write_ins(tb_i_c_write_ins),
        .i_c_ins_wr(tb_i_c_ins_wr)
    );

    // Clock generation
    always #(CC/2) tb_i_c_sys_clock = ~tb_i_c_sys_clock;

    initial begin
        tb_i_c_sys_clock =1'b1;
        tb_i_c_sys_reset = 1;
        @(posedge tb_i_c_sys_clock); // reset the processor
        tb_i_c_sys_reset = 0;
        //---------------------------------------
        // Test Case X: X is the current test case number
        //---------------------------------------
        tb_i_c_sys_reset = 1;
        @(posedge tb_i_c_sys_clock); // reset the processor
        tb_i_c_sys_reset = 0;
        //---------------------------------------
        //Program Mode
        //---------------------------------------
        $readmemh(testcase4,mem);
        tb_i_c_ins_wr = 1;
        for(i=0;i<60;i++)@(posedge tb_i_c_sys_clock)
        tb_i_c_write_ins = mem[i];
        tb_i_c_ins_wr = 0;
        repeat(1)@(posedge tb_i_c_sys_clock);
        tb_i_c_sys_reset = 1;                //reset the processor
        repeat(1)@(posedge tb_i_c_sys_clock);
        tb_i_c_sys_reset = 0;
        //---------------------------------------
        //Normal Mode
        //---------------------------------------
        repeat(100)@(posedge tb_i_c_sys_clock);//wait for result


        $finish;
    end

endmodule
```

3.2.2 Simulation result

Test Case 1: ADD to Max value (2^{31})

Assembly code			Hex Code
lui $1, 32767		# li $t2, 0x7FFFFFFE	3c017fff
nop	x3		00000000(3)
ori $10, $1, -2			342afffe
ori $11, $0, 1		# li $t3, 1	340b0001
nop	x3		00000000(3)
add $12, $10, $11		# add $t4, $t2, $t3	014b6020

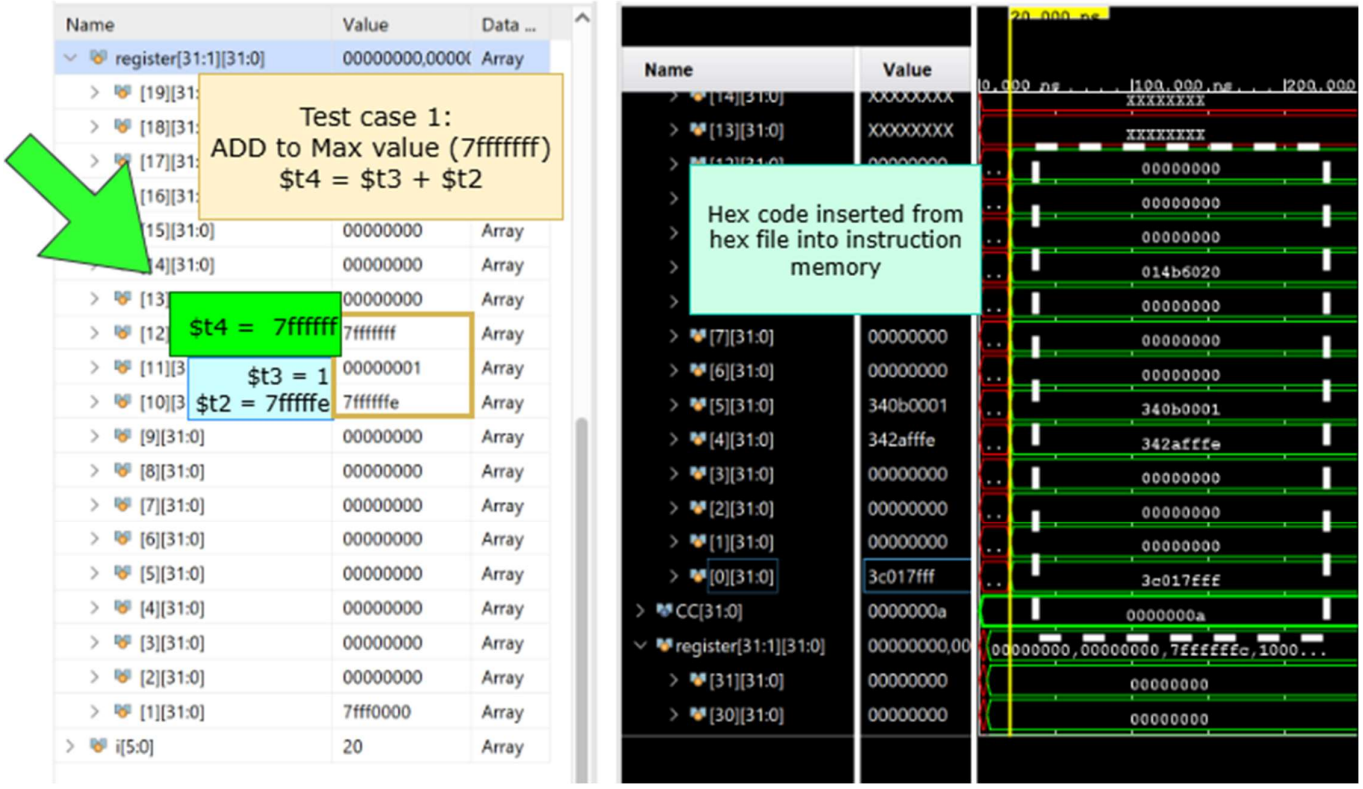


Diagram 3.2.2.1 Test case 1

Test Case 2: ADDI: Add max immediate value

Assembly code		Hex Code
addi $9, $0, 32767	# addi $t1, $zero, 32767	20097fff
nop		00000000
nop		00000000
nop		00000000

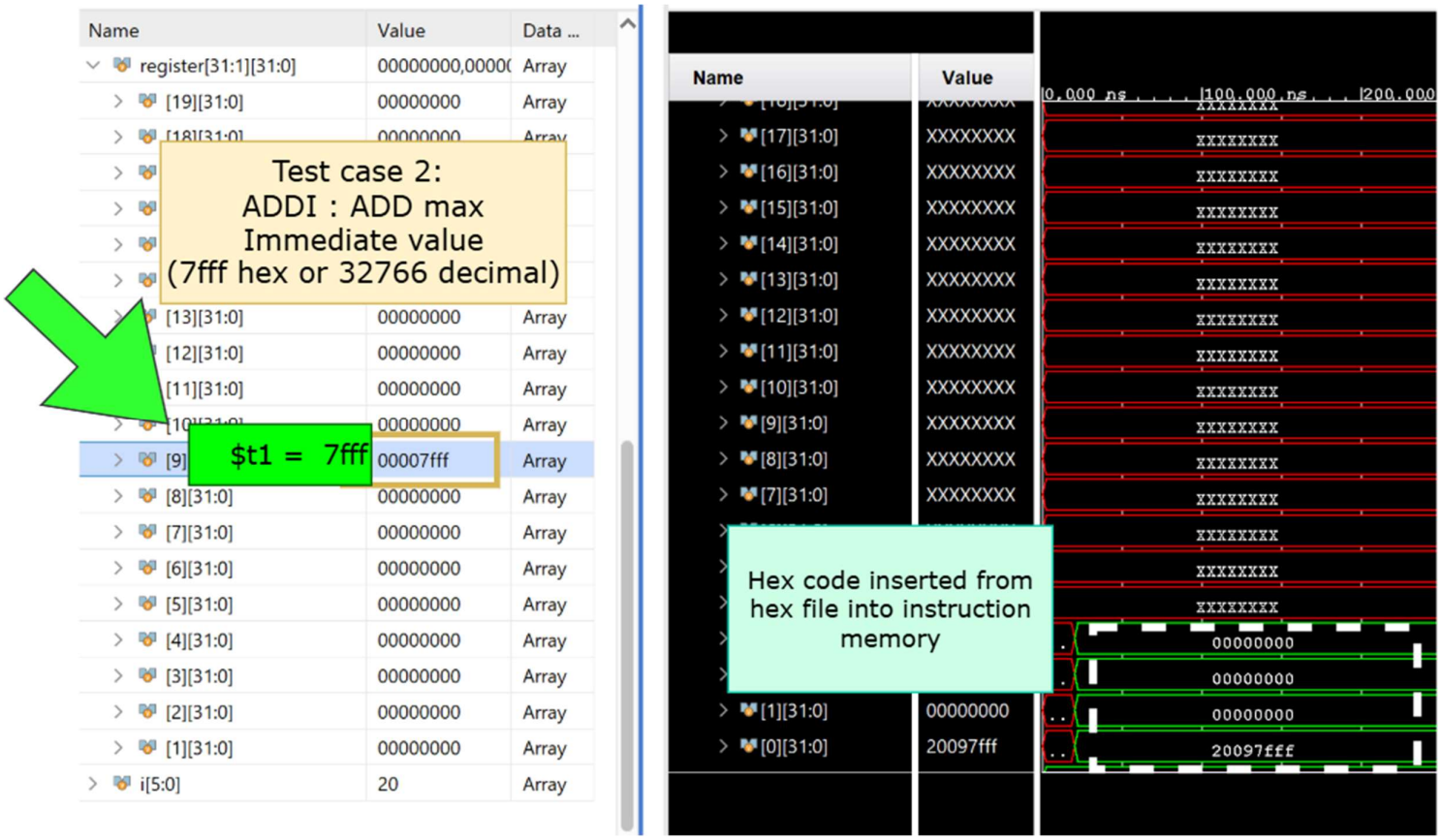


Diagram 3.2.2.2 Test case 2

Test Case 3: Multu (Max unsigned × max unsigned)

Assembly code			Hex Code
lui $1, -1		# li $t0, 0xFFFFFFFF	3c01ffff
nop	x3		00000000 (3)
ori $8, $1, -1			3428ffff
lui $1, -1		# li $t1, 0xFFFFFFFF	3c01ffff
nop	x3		00000000 (3)
ori $9, $1, -1			3429ffff
nop	x3		00000000 (3)
multu $8, $9		# multu $t0, $t1	01090019
nop	x35		00000000 (35)

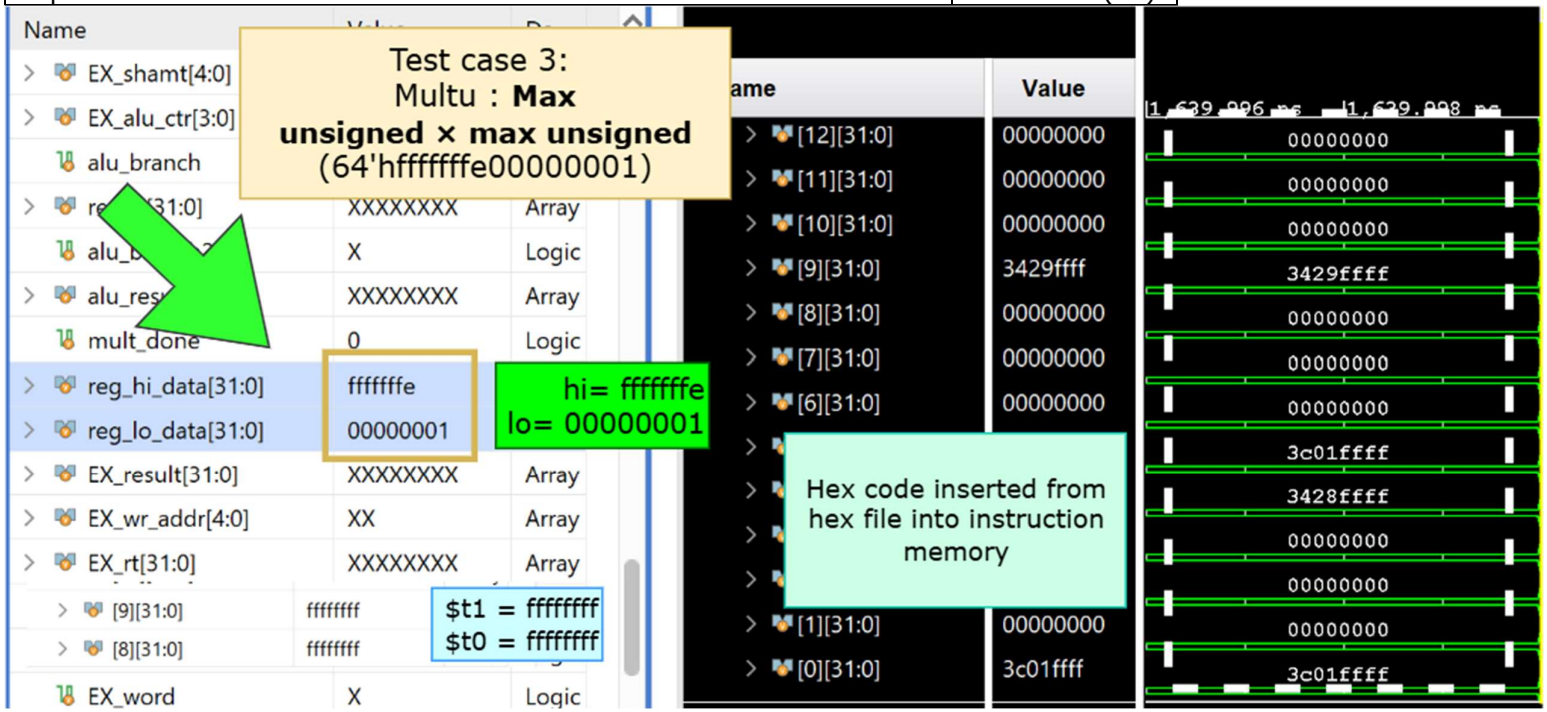


Diagram 3.2.2.3 Test case 3

Test Case 4: Mfhi: Move from HI after large mult

Assembly code	Hex Code
(continue from test plan 3) mfhi $10 # mfhi $t2	00005010

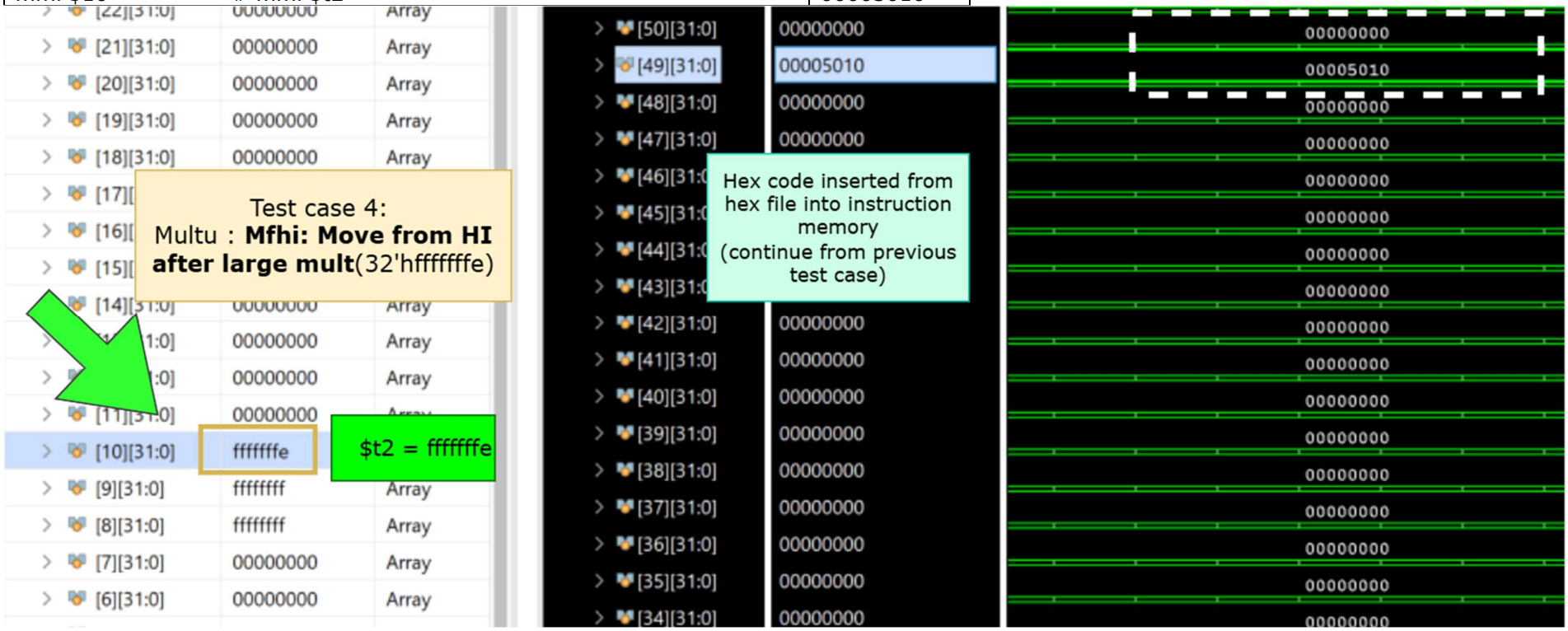


Diagram 3.2.2.4 Test case 4

Test Case 5: sw: Store large word

Assembly code			Hex Code
lui $1, -292		# li $t1, 0xFEDCBA98	3c01fedc
nop	x3		00000000(3)
ori $9, $1, -17768			3429ba98
lui $1, 0 [store_byte]		# la $t0, store_byte	3c010000
nop	x3		00000000(3)
ori $8, $1, 0 [store_byte]			34280000
nop	x3		00000000(3)
sw $9, 12($8)		# sw $t1, 12($t0)	ad09000c

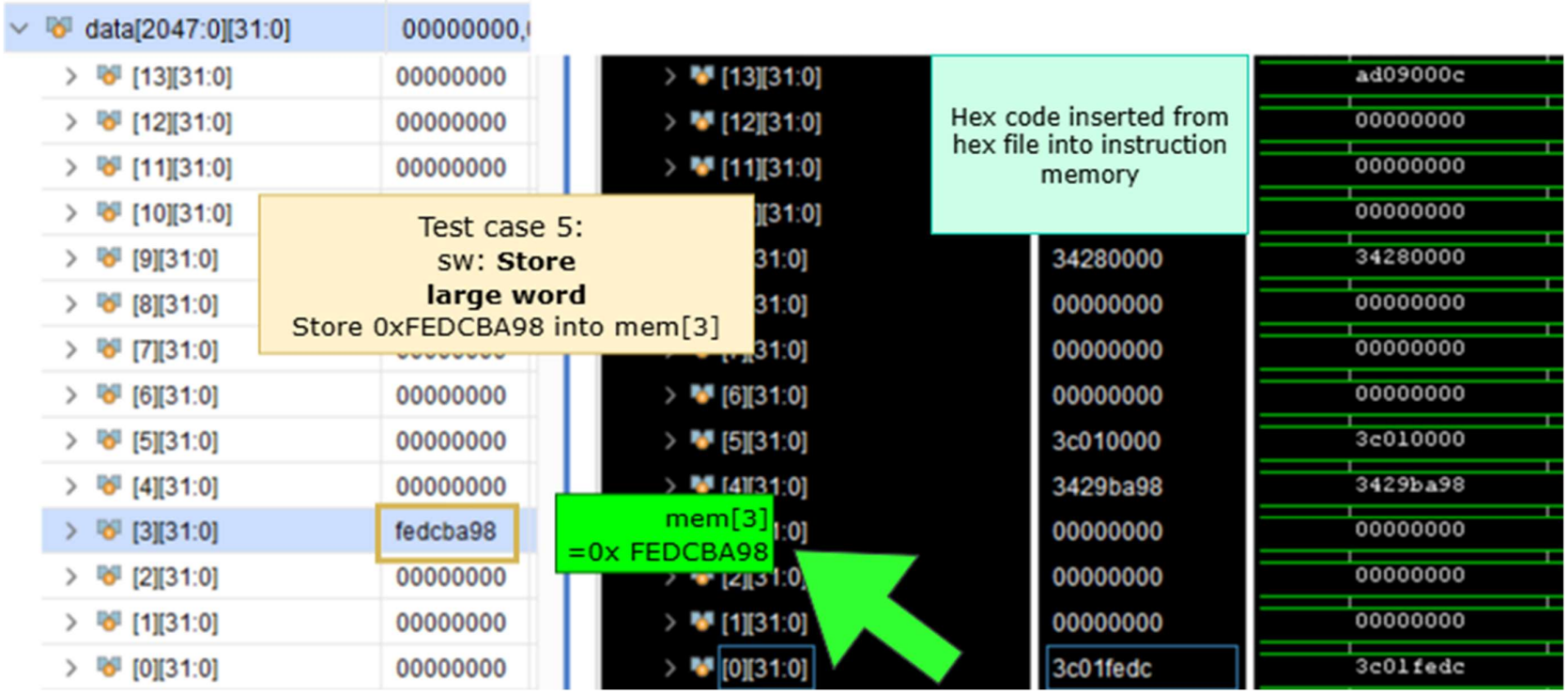


Diagram 3.2.2.4 Test case 5

Test Case 6: lb: Load byte [1] from a memory space ([15:8] lower bits)

Assembly code	Hex Code
(continue from previous test case) lb $10, 13($8) # lb $t2, 13($t0)	810a000d

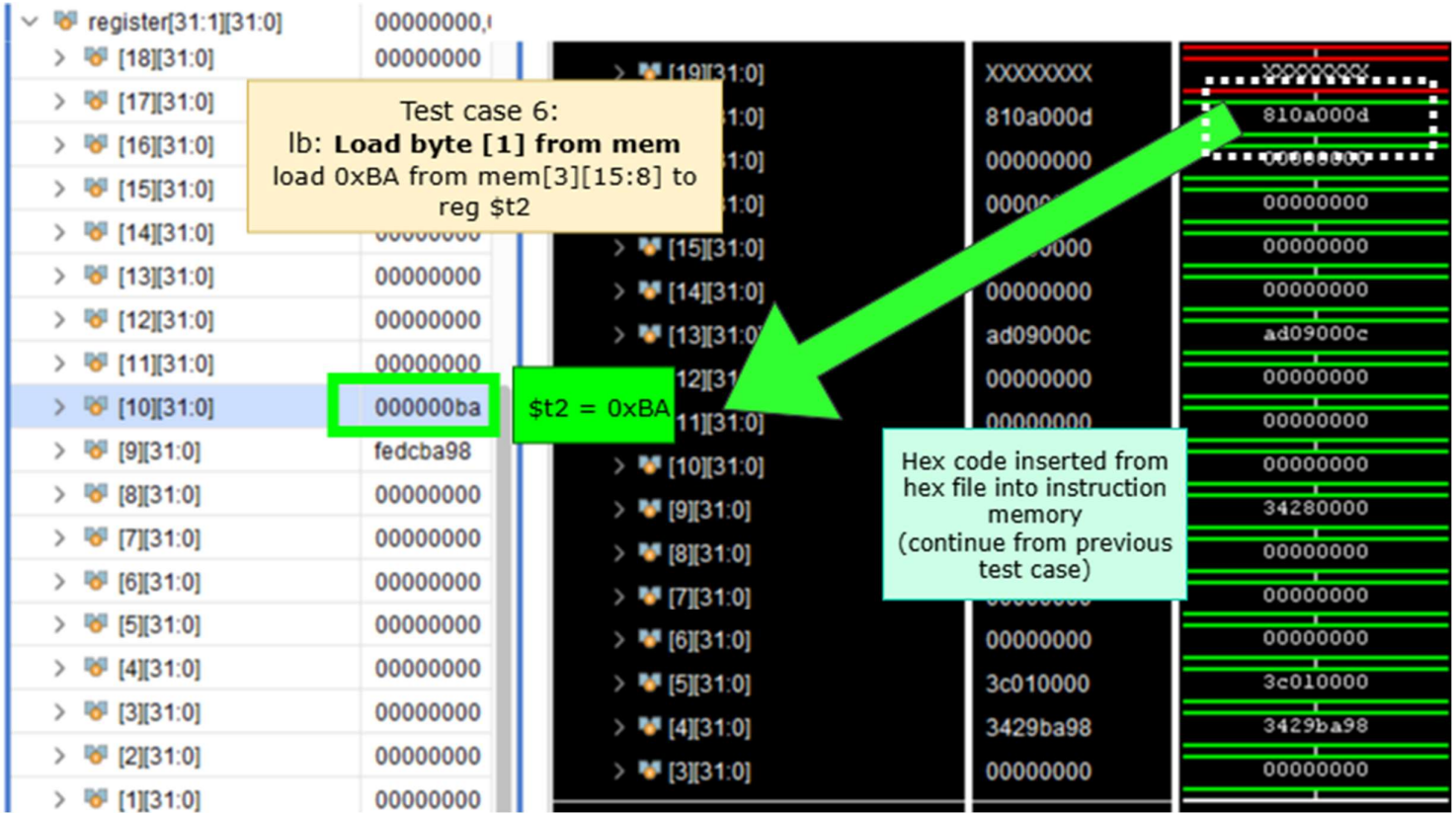


Diagram 3.2.2.6 Test case 6

Test Case 7: beq: Branch if -1 == -1

Assembly code			Hex Code
lui $1, -1		# li $t3, 0xFFFFFFFF	3c01ffff
nop	x3		00000000(3)
ori $11, $1, -1			342bffff
lui $1, -1		# li $t4, 0xFFFFFFFF	3c01ffff
nop	x3		00000000(3)
ori $12, $1, -1			342cffff
nop	x3		00000000(3)
beq $11, $12, 12 [equal2-0x0040003c]		# beq $t3, $t4, equal	116c0005
nop	x3	# solve ctrl hazard	00000000(3)
ori $13, $0, 9		# li $t5, 9	340d0009
ori $14, $0, 1		# equal2: li $t6, 1	340e0001

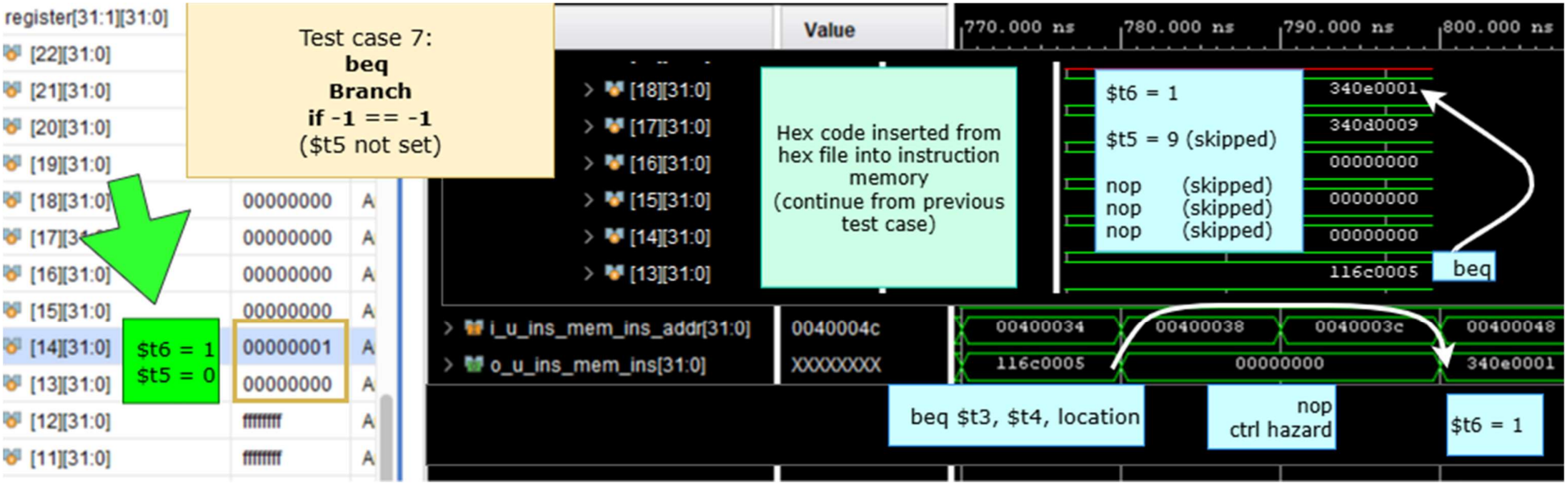


Diagram 3.2.2.7 Test case 7

Test Case 8: j: Jump over instruction

Assembly code		Hex Code
j 0x00400012 [jump_target]	# j jump_target	08040004
ori $8, $0, 9	# li $t0, 9	34080009
ori $9, $0, 1	# li $t1, 1 (jump_target)	34090001

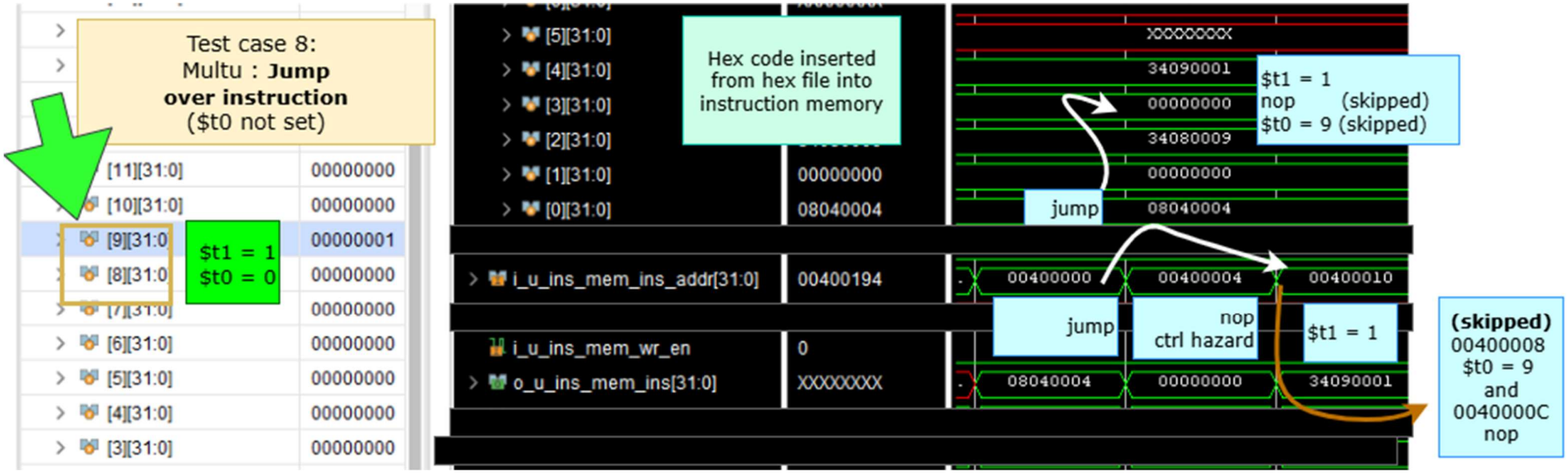


Diagram 3.2.2.8 Test case 8

Test Case 9: jr: Jump to register address

Assembly code			Hex Code
lui $1, 64 [target_label]		# la $t0, target_label	3c010040
nop	x3		00000000(3)
ori $8, $1, 64 [target_label]			34280040
nop	x3		00000000(3)
jr $8		# jr $t0	01000008
nop			00000000
ori $9, $0, 9		# li $t1, 9	34090009
ori $10, $0, 1		# li $t2, 1	340a0001

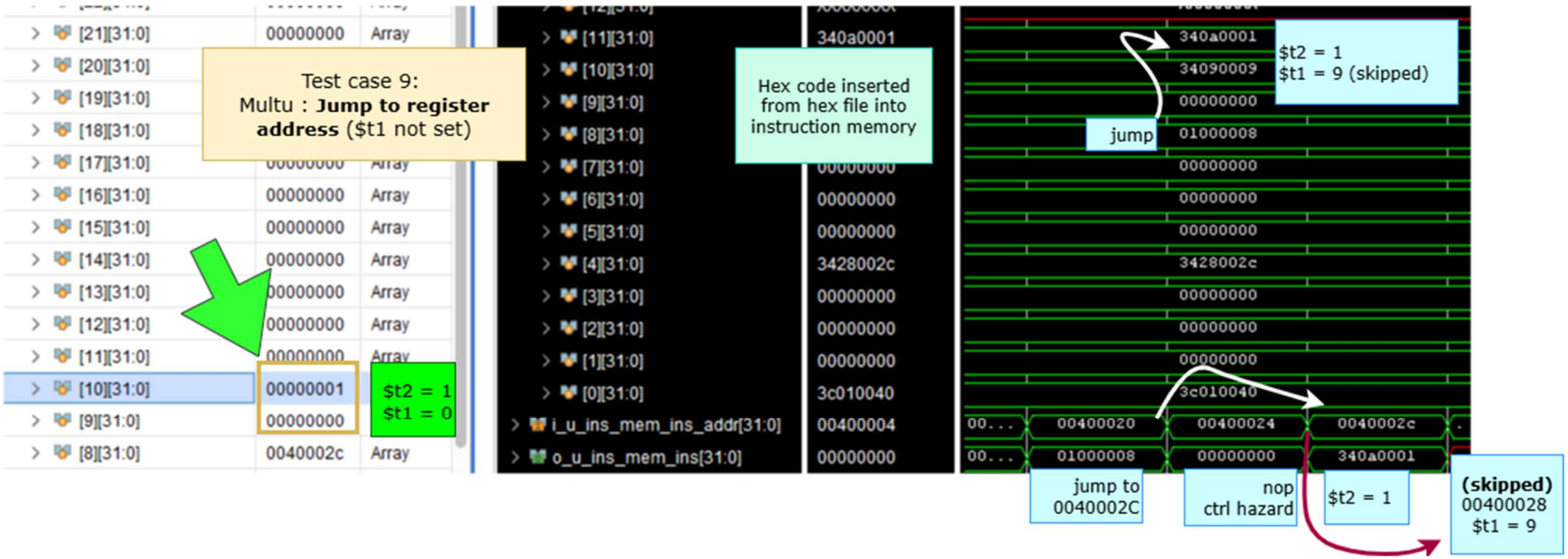


Diagram 3.2.2.9 Test case 9

Test Case 10: Basic Conditional Branching (if-else): If a>b,then c=100, else c=200

Assembly code			Hex Code
addi $16, $0, 15		# addi $s0, $zero, 15	2010000f
addi $17, $0, 10		# addi $s1, $zero, 10	2011000a
nop	x3		00000000(3)
slt $8, $17, $16		# slt $t0, $s1, $s0	0230402a
nop	x3		00000000(3)
beq $8, $0, 12 [ELSE-0x00400030]			11000003
addi $18, $0, 100		# addi $s2, $zero, 100	20120064
j 0x00400040 [EXIT]		# j EXIT	08100010
nop	x3		00000000(3)
addi $18, $0, 200		# addi $s2, $zero, 200	201200c8
		#EXIT:	

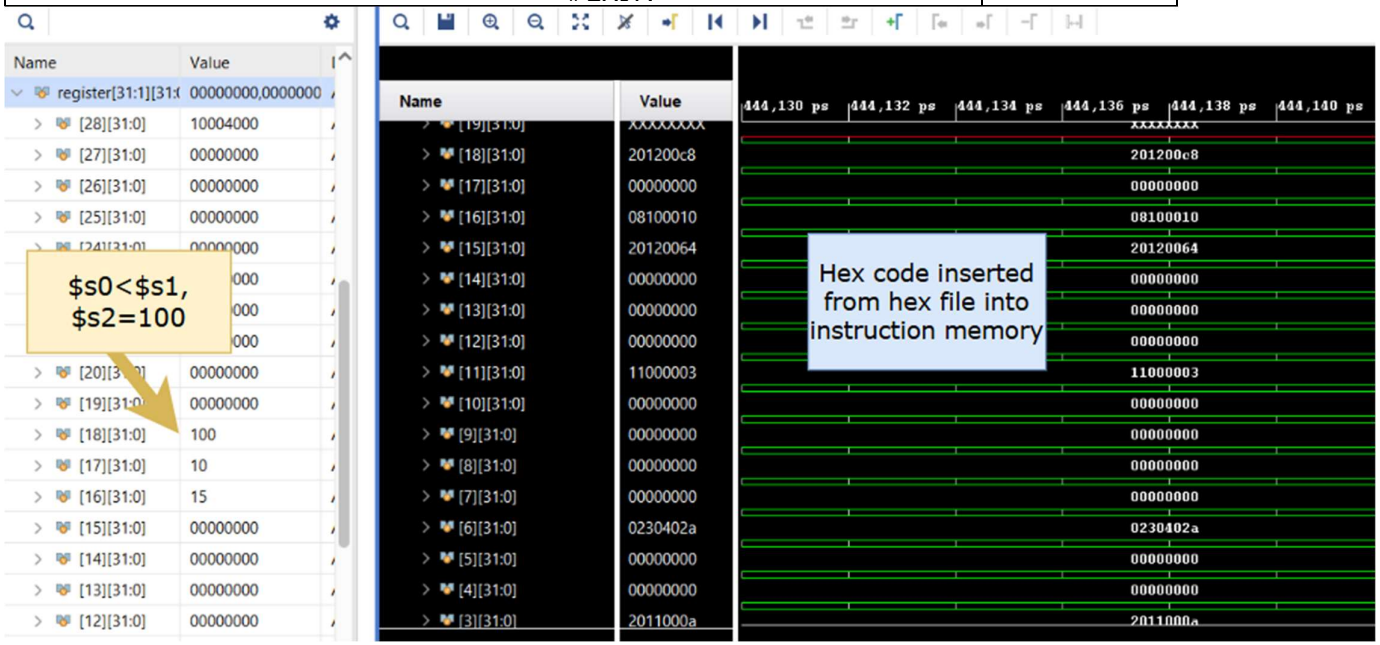


Diagram 3.2.2.10 Test case 10

Test Case 11: Loop Structure (beq,slt,j): Sum of integers from 1 to 5

Assembly code	Hex Code
addi $16, $0, 0 # addi $s0, $zero, 0	20100000
addi $17, $0, 5 # addi $s1, $zero, 5	20110005
nop x3	00000000(3)
slti $8, $17, 1 # slti $t0, $s1, 1	2a280001
nop x3	00000000(3)
slti $8, $17, 1 # slti $t0, $s1, 1	2a280001
nop x3	00000000(3)
beq $8, $0, 20 [DO_ADD-current address]	11000005
nop x2	00000000(2)
j 0x00400060 [END] # j END	08100018
nop	00000000
add $16, $16, $17 # add $s0, $s0, $s1	02118020
addi $17, $17, -1 # addi $s1, $s1, -1	2231ffff
nop x3	00000000(3)
j 0x00400010 [LOOP] # j LOOP	08100004
nop	00000000

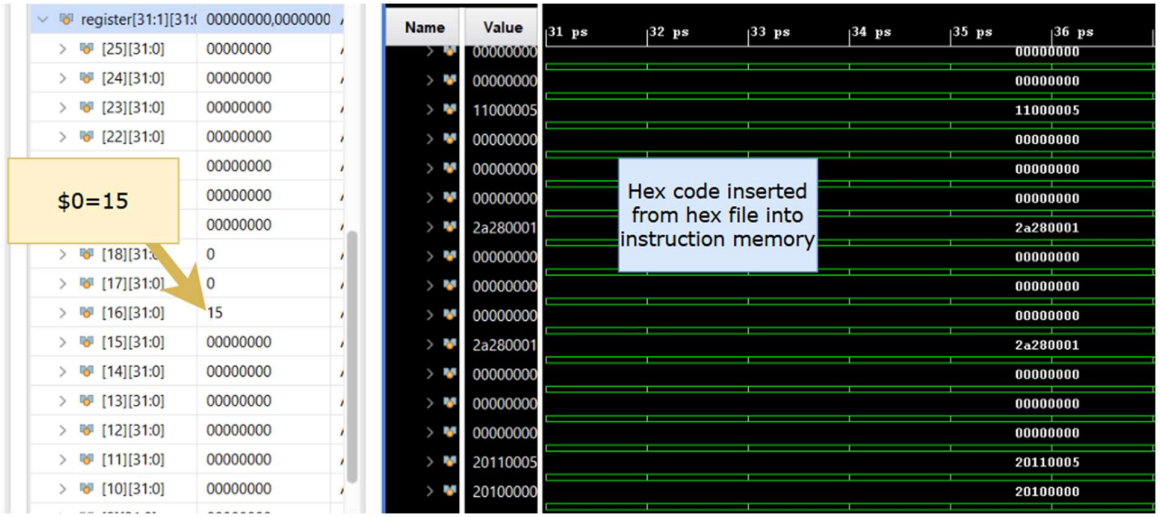


Diagram 3.2.2.11 Test case 1

Test Case 12: Nested Structure: Counts the total number of iterations

Assembly code			Hex Code
add $16, $0, $0		# add $s0, $zero, $zero	20100000
addi $17, $0, 0		# addi $s1, $zero, 0	20110000
nop	x3		00000000(3)
slti $8, $17, 2		# slti $t0, $s1, 2	2a280002
nop	x3		00000000(3)
beq $8, $0, 36 [DONE-0x00400030]			11000008
nop	x2		00000000(2)
addi $18, $0, 0		# addi $s2, $zero, 0	20120000
slti $9, $18, 3		# slti $t1, $s2, 3	2a490003
nop	x3		00000000(3)
beq $9, $0, 16 [INCR_I-0x0040003c]			11200005
nop	x2		00000000(2)
addi $16, $16, 1		# addi $s0, $s0, 1	22100001
addi $18, $18, 1		# addi $s2, $s2, 1	22520001
nop	x3		00000000(3)
j 0x00400038 [INNER_LOOP]		# j INNER_LOOP	08100009
nop			00000000
addi $17, $17, 1		# addi $s1, $s1, 1	22310001
j 0x0040002c [OUTER_LOOP]		# j OUTER_LOOP	08100002
nop			00000000

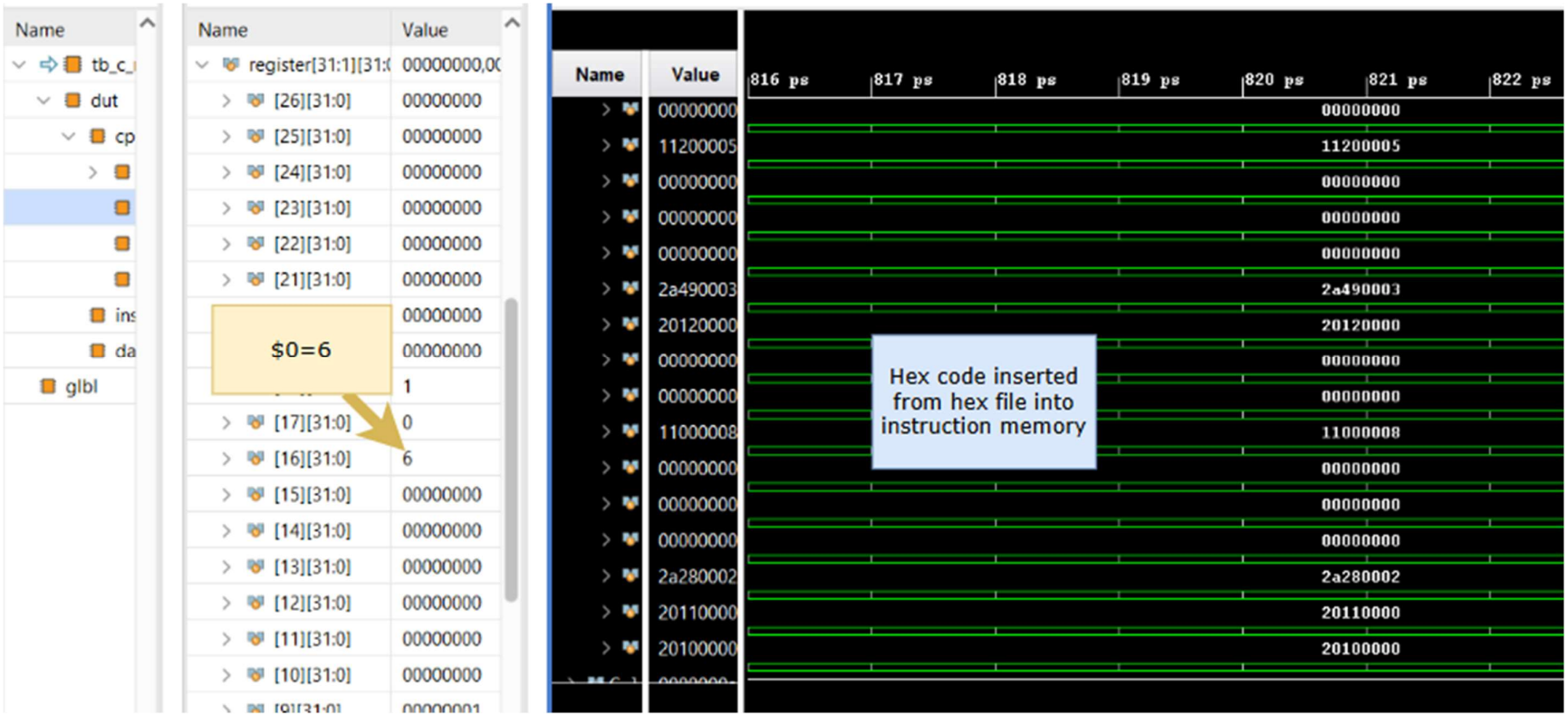


Diagram 3.2.2.12 Test case 12

Test Case 13: Procedure Call: Double the value

Assembly code	Hex Code
addi $4, $0, 21 # addi $a0, $zero, 21 nop	20040015
nop x3	00000000(3)
jal 0x00400028 [double] # jal double	0C00000A
nop	00000000
add $16, $2, $0 # add $s0, $v0, $zero	00508020
nop	00000000
j 0x00400044 [end] # j end	08100011
nop	00000000
add $2, $4, $4 # add $v0, $a0, $a0	00841020
nop x2	00000000(2)
jr $31 # jr $ra	03e00008
nop	00000000
j 0x00400044 [end] # j end	08100011
nop	00000000
#end:	

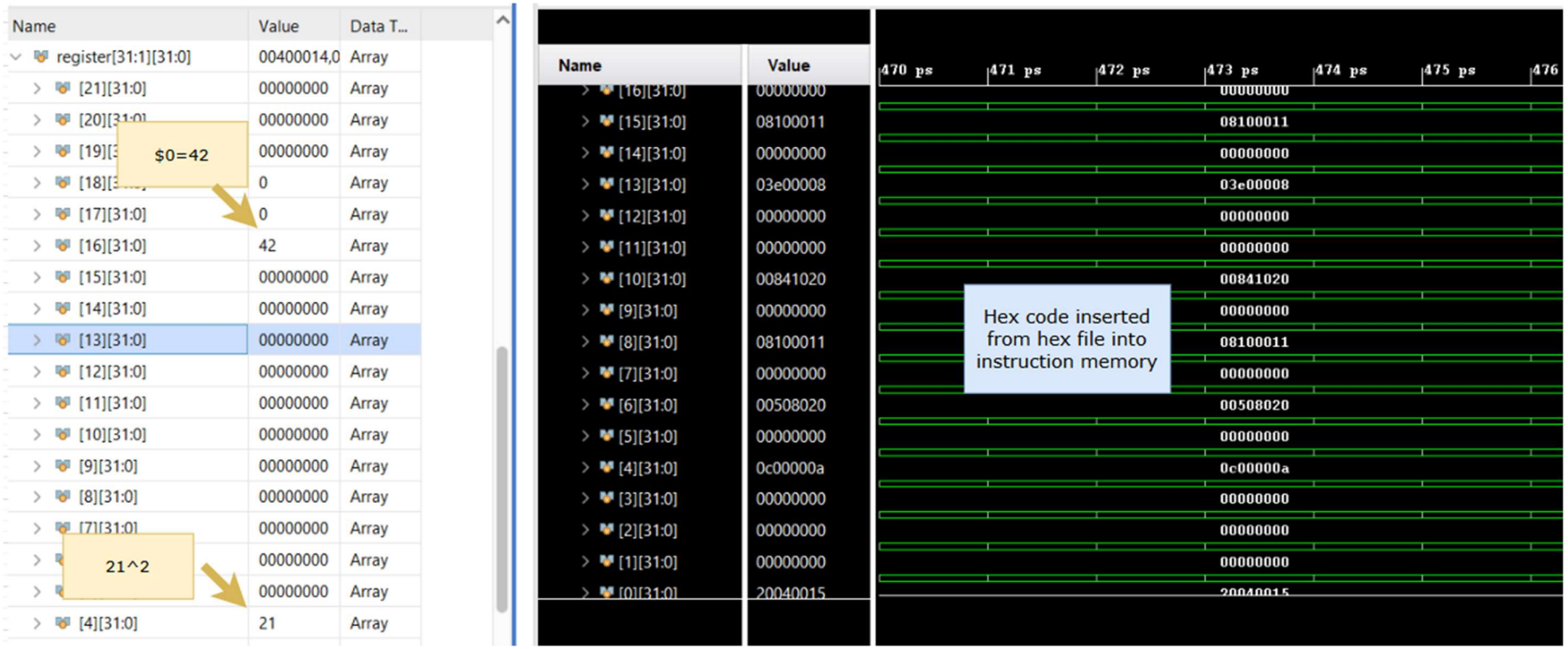


Diagram 3.2.2.13 Test case 13

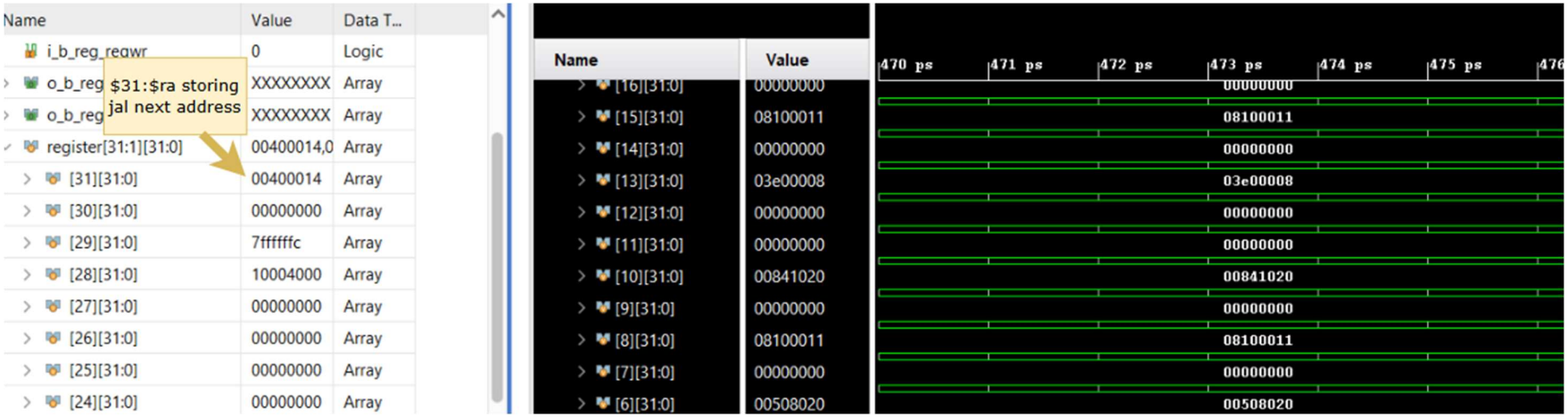
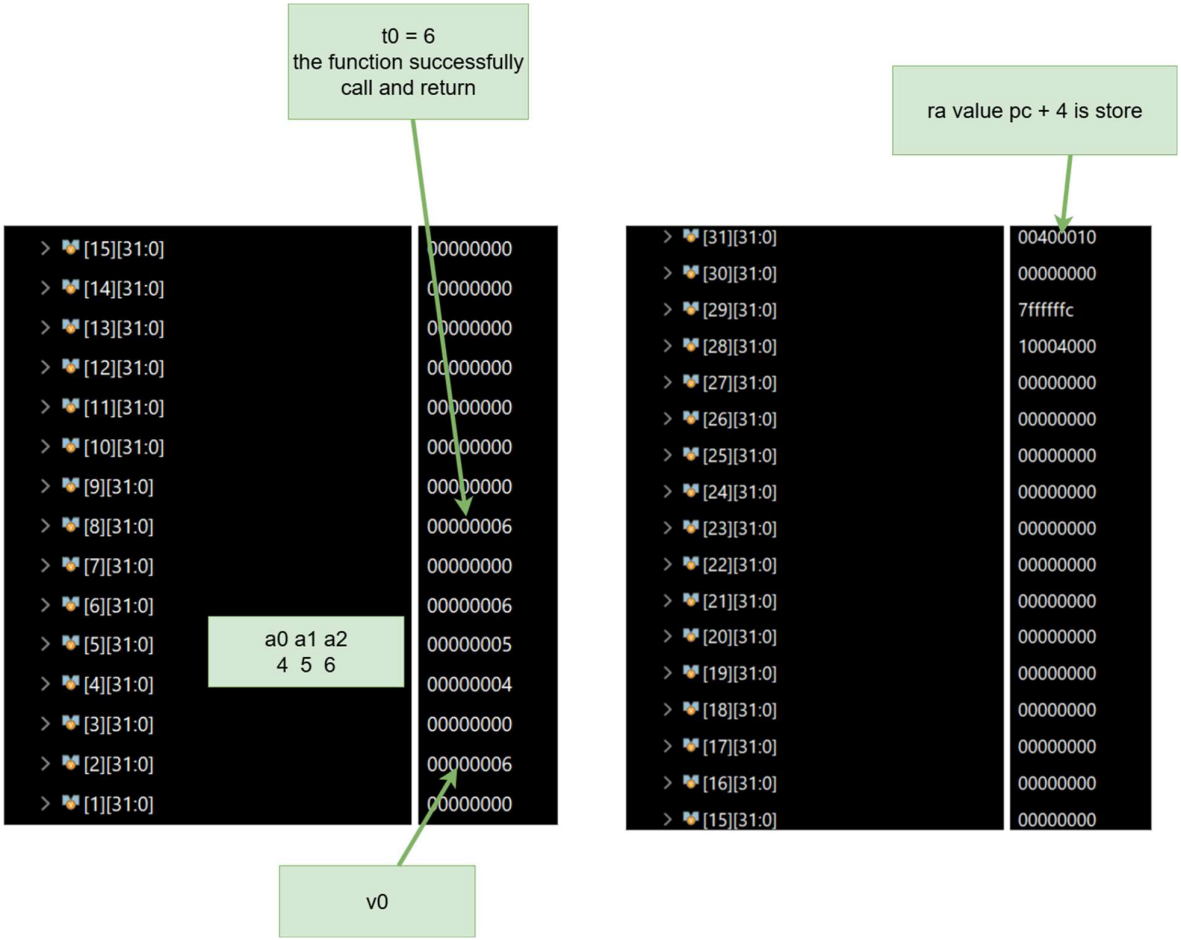


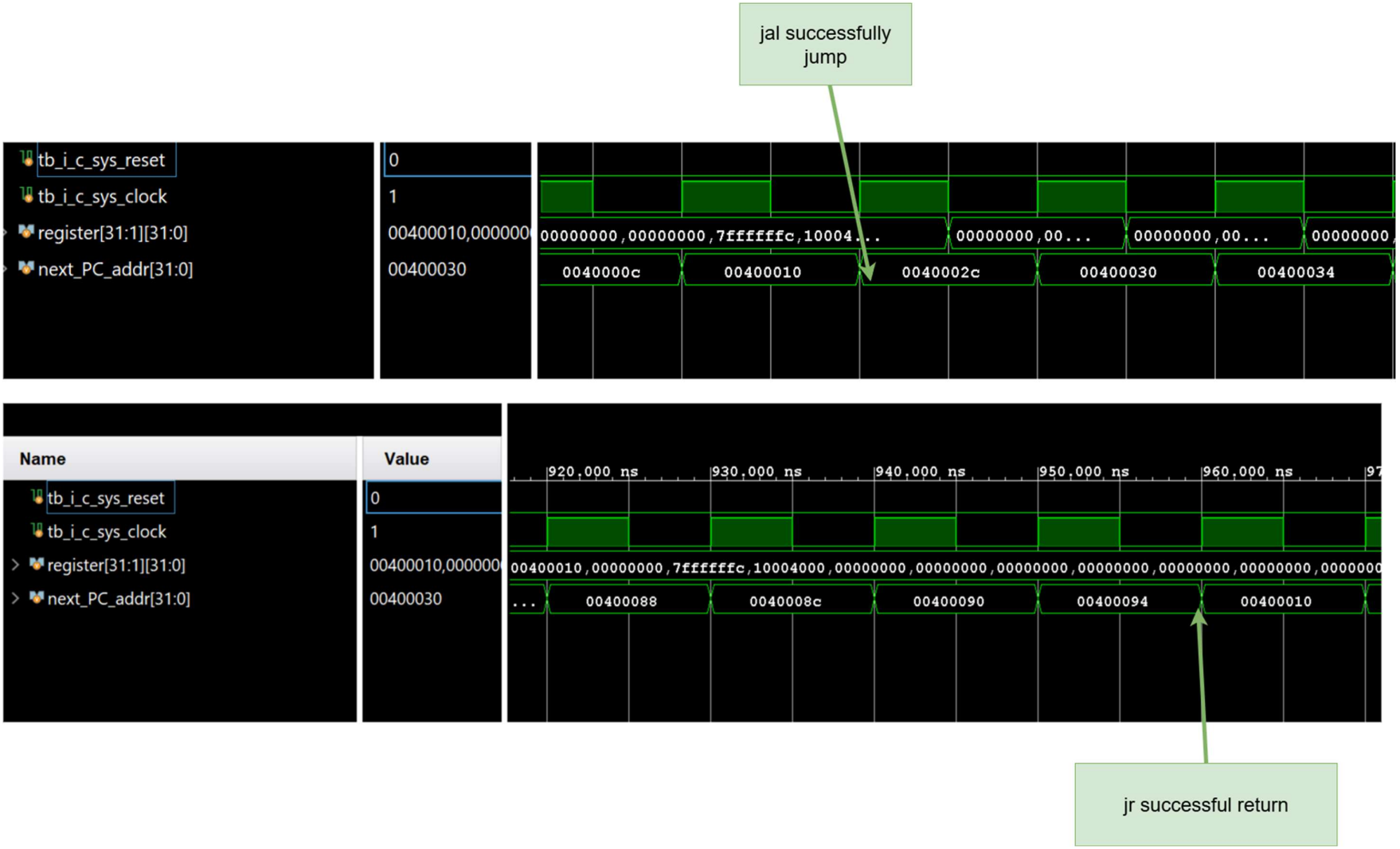
Diagram 3.2.2.14 Test case 13

Test Case 14: Integration test program : findmax

Assembly code	Hex Code
main:	
# $t0 = findMax(4, 5, 6)	
addi $a0, $zero, 4 # $a0 = 4	20040004
addi $a1, $zero, 5 # $a1 = 5	20050005
addi $a2, $zero, 6 # $a2 = 6	20060006
jal findMax # Jump and link to the findMax function	0c10000b
nop	00000000
add $t0, $v0, $zero	00404020
nop	00000000
nop	00000000
nop	00000000
nop	00000000
nop #wait to store	00000000
# findMax (int x, int y, int z)	
# Arguments: $a0 (x), $a1 (y), $a2 (z)	
# Return value: $v0	
findMax:	
# int max = x;	
add $v0, $a0, $zero # 'move $v0, $a0'	00801020
nop	00000000
nop	00000000
nop	00000000
# if (y > max)	

slt $t0, $v0, $a1 # Set less than: $t0 = 1 if $v0 < $a1 (y > max)	
nop	0045402a
nop	00000000
nop	00000000
	00000000
beq $t0, $zero, check_z # Branch to check_z if $t0 is not equal to zero	
nop	11000007
nop	00000000
nop	00000000
	00000000
# max = y;	
add $v0, $a1, $zero # 'move $v0, $a1'	
nop	00a01020
nop	00000000
nop	00000000
	00000000
check_z:	
# if (z > max)	
slt $t0, $v0, $a2 # Set less than: $t0 = 1 if $v0 < $a2 (z > max)	
nop	0046402a
nop	00000000
nop	00000000
	00000000
beq $t0, $zero, end_func # Branch to end_func if $t0 is not equal to zero	11000004
nop	
nop	00000000
nop	00000000
	00000000
# max = z;	
add $v0, $a2, $zero # 'move $v0, $a2'	
	00c01020
end_func:	
jr $ra # Jump register to the return address	
nop	03e00008
	00000000


t0 = 6
the function successfully
call and return
ra value pc + 4 is store
[15][31:0] 00000000
[14][31:0] 00000000
[13][31:0] 00000000
[12][31:0] 00000000
[11][31:0] 00000000
[10][31:0] 00000000
[9][31:0] 00000000
[8][31:0] 00000006
[7][31:0] 00000000
[6][31:0] 00000006
[5][31:0] 00000005
[4][31:0] 00000004
[3][31:0] 00000000
[2][31:0] 00000006
[1][31:0] 00000000
a0 a1 a2
4 5 6
[31][31:0] 00400010
[30][31:0] 00000000
[29][31:0] 7ffffffc
[28][31:0] 10004000
[27][31:0] 00000000
[26][31:0] 00000000
[25][31:0] 00000000
[24][31:0] 00000000
[23][31:0] 00000000
[22][31:0] 00000000
[21][31:0] 00000000
[20][31:0] 00000000
[19][31:0] 00000000
[18][31:0] 00000000
[17][31:0] 00000000
[16][31:0] 00000000
[15][31:0] 00000000
v0


jal successfully jump
tb_i_c_sys_reset
tb_i_c_sys_clock
register[31:1][31:0]
next_PC_addr[31:0]
0
1
00400010,000000
00400030
00000000,00000000,7ffffffc,10004...
00000000,00...
00000000,00...
00000000
0040000c
00400010
0040002c
00400030
00400034
Name
Value
tb_i_c_sys_reset
tb_i_c_sys_clock
register[31:1][31:0]
next_PC_addr[31:0]
0
1
00400010,000000
00400030
920,000 ns
930,000 ns
940,000 ns
950,000 ns
960,000 ns
00400010,00000000,7ffffffc,10004000,00000000,00000000,00000000,00000000,00000000,00000000,0000000
...
00400088
0040008c
00400090
00400094
00400010
jr successful return

4 Micro-Architecture Specification (Block level)

4.1 CPU Unit

4.1.0.1 Functionality/Feature

- Decode and execute the instructions received
- Receive data from memory and store in register file to be used

4.1.0.2 CPU Unit interface and I/O pin description

4.1.0.2.1 CPU block interface

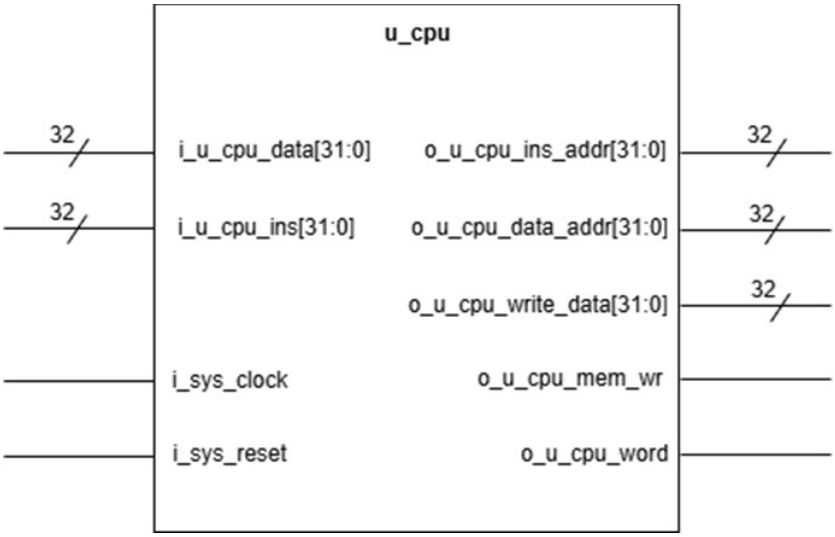


Diagram 4.1.2.1.1 Block interface of CPU unit

4.1.0.2.2 I/O pin description

Pin name: **Pin class:** **Pin function:**	i_u_cpu_data[31:0] data to receive the data read from data memory	**Source** **→Destination:**	Data Memory Unit → CPU Unit
Pin name: **Pin class:** **Pin function:**	i_u_cpu_ins[31:0] data to receive instruction for decoding	**Source** **→Destination:**	Instruction Memory Unit → CPU Unit
Pin name: **Pin class:** **Pin function:**	i_sys_clock control To provide a system clock to the block	**Source** **→Destination:**	Clock Generator → CPU Unit

Pin name: **Pin class:** **Pin function:**	i_sys_reset control To reset the clock	**Source** **→Destination:**	ON/Reset button → CPU Unit
Pin name: **Pin class:** **Pin function:**	o_u_cpu_ins_addr[31:0] data to give address for instruction fetching	**Source** **→Destination:**	CPU Unit → Instruction Memory Unit
Pin name: **Pin class:** **Pin function:**	o_u_cpu_data_addr[31:0] data to give address of data to read or write	**Source** **→Destination:**	CPU Unit → Data Memory Unit
Pin name: **Pin class:** **Pin function:**	o_u_cpu_write_data[31:0] data Provide data to be written into data memory	**Source** **→Destination:**	CPU Unit → Data Memory Unit
Pin name: **Pin class:** **Pin function:**	o_u_cpu_mem_wr control Provide write enable signal	**Source** **→Destination:**	CPU Unit → Data Memory Unit
Pin name: **Pin class:** **Pin function:**	o_u_cpu_word control Provide signal to indicate whether to store/load data in word or byte	**Source** **→Destination:**	CPU Unit → Data Memory Unit

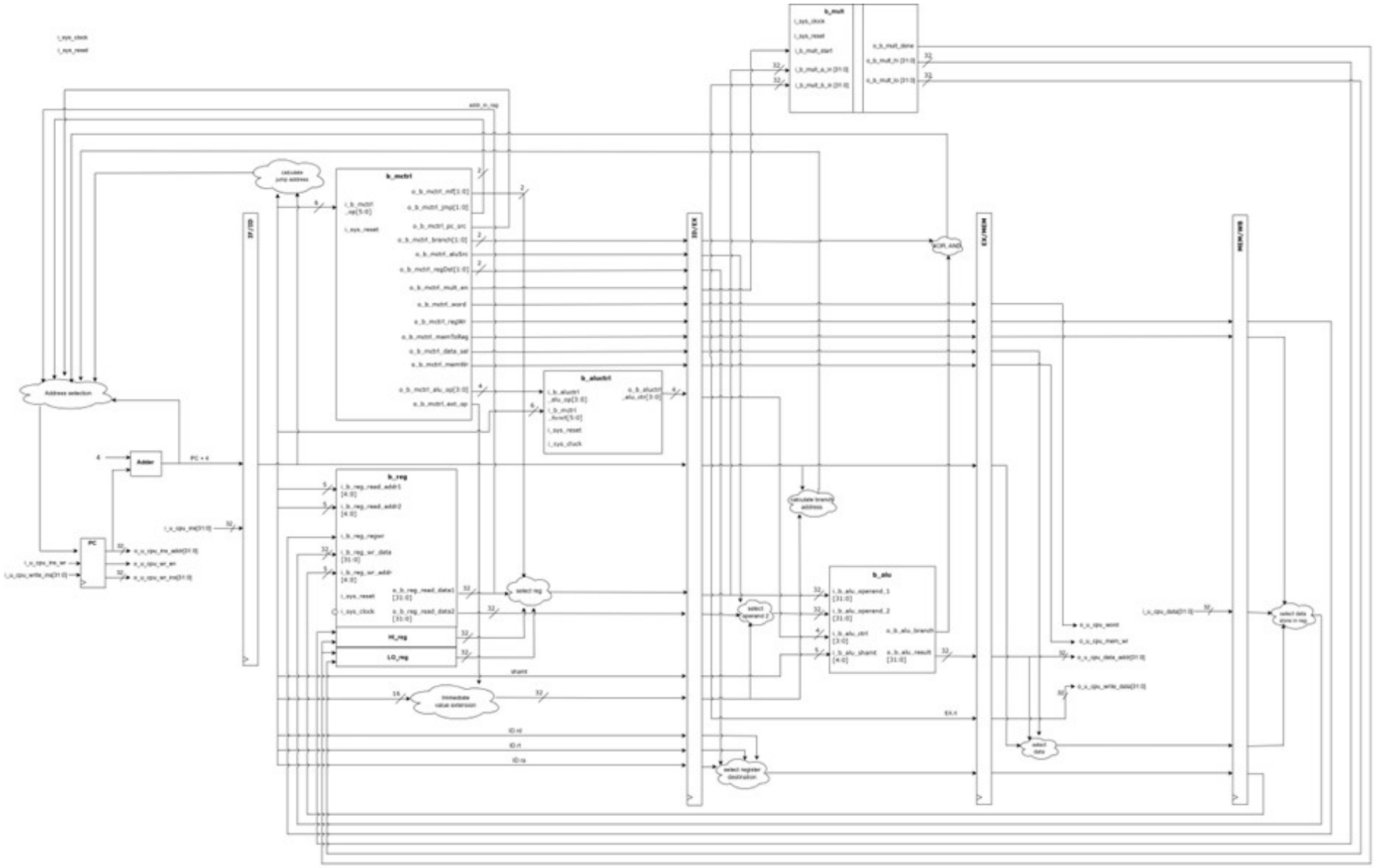
4.1.0.3 Internal Operation

Input	**Stages**				
Instruction	**IF**	**ID**	**EX**	**MEM**	**WB**
Add	- Fetch instruction - PC+4	- Decode instruction - read data from reg file	- Add both data	x	- Write back data to rd
Sub	- Fetch instruction - PC+4	- Decode instruction - read data from reg file	- Sub both data	x	- Write back data to rd
And	- Fetch instruction - PC+4	- Decode instruction - read data from reg file	- And both data	x	- Write back data to rd

Or	- Fetch instruction - PC+4	- Decode instruction - read data from reg file	- Or both data	x	- Write back data to rd
Xor	- Fetch instruction - PC+4	- Decode instruction - read data from reg file	- Xor both data	x	- Write back data to rd
Nor	- Fetch instruction - PC+4	- Decode instruction - read data from reg file	- Nor both data	x	- Write back data to rd
Slt	- Fetch instruction - PC+4	- Decode instruction - read data from reg file	- compare both data and set on less than	x	- Write back data to rd
Sll	- Fetch instruction - PC+4	- Decode instruction - read data from reg file	- shift left logical	x	- Write back data to rd
Srl	- Fetch instruction - PC+4	- Decode instruction - read data from reg file	- shift right logical	x	- Write back data to rd
Multu - need to wait (nop) 32 cycles	- Fetch instruction - PC+4	- Decode instruction - read data from reg file	- multiply both data	x	- Write back data to reg hi and reg lo
Mfhi	- Fetch instruction - PC+4	- Decode instruction - read data from reg hi	- add both data	x	- Write back data to rd
Mflo	- Fetch instruction - PC+4	- Decode instruction - read data from reg lo	- add both data	x	- Write back data to rd
Jr	- Fetch instruction - jump to address in register	- Decode instruction - new PC = address store in reg	x	x	x
Jalr	- Fetch instruction - jump to address	- Decode instruction - new PC = address store in reg	x	x	- Write back PC+4 to rd
Lw	- Fetch instruction - PC+4	Decode instruction - sign/ unsign extend of imm	- address = imm + rs	- Get data according to address	- Write back data to rt
Sw	- Fetch instruction - PC+4	Decode instruction - sign/ unsign extend of imm	- address = imm + rs	- Write data to address in memory	x

Lb	- Fetch instruction - PC+4	Decode instruction - sign/ unsign extend of imm	- address = imm + rs	- Get data according to address	- Write back data to rt
Sb	- Fetch instruction - PC+4	Decode instruction - sign/ unsign extend of imm	- address = imm + rs	- Write data to address in memory	x
Addi	- Fetch instruction - PC+4	- Decode instruction - read data from reg file - sign/ unsign extend of imm	- Add data with imm	x	- Write back data to rd
Andi	- Fetch instruction - PC+4	- Decode instruction - read data from reg file - sign/ unsign extend of imm	- And data with imm	x	- Write back data to rd
Ori	- Fetch instruction - PC+4	- Decode instruction - read data from reg file- sign/ unsign extend of imm	- Or data with imm	x	- Write back data to rd
Slti	- Fetch instruction - PC+4	- Decode instruction - read data from reg file - sign/ unsign extend of imm	- compare data with imm and set on less than	x	- Write back data to rd
Beq	- Fetch instruction	- Decode instruction - read data from reg file - sign/ unsign extend of imm	- sub both data and compare - new PC = PC+4 + imm - branch if equal	x	x
Bne	- Fetch instruction	- Decode instruction - read data from reg file - sign/ unsign extend of imm	- sub both data and compare - new PC = PC+4 + imm - branch if not equal	x	x
J	- Fetch instruction - jump to address	- Decode instruction - new PC = {(PC+4) [31:28], jump_ addr_offset, 00}	x	x	x
jal	- Fetch instruction - jump to address	- Decode instruction - new PC = {(PC+4) [31:28], jump_ addr_offset, 00}	x	x	- Write back PC+4 to ra

4.1.0.4 Pre-synthesis Schematic



4.1.0.5 SV model

```
 `timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Ng Jing Xuan
// Create Date: 04.09.2025 22:59:17
// File Name: u_cpu.sv
// Module Name: u_cpu
// Project Name: MIPS ISA Pipeline processor
// Code Type: RTL level
// Description: Modeling of CPU unit block
//
//////////////////////////////////////////////////////////////////////////////////
module u_cpu(
input logic i_sys_clock,
input logic i_sys_reset,
input logic [31:0] i_u_cpu_data,
input logic [31:0] i_u_cpu_ins,
output logic [31:0] o_u_cpu_ins_addr,
output logic [31:0] o_u_cpu_data_addr,
output logic [31:0] o_u_cpu_write_data,
output logic o_u_cpu_word,
output logic o_u_cpu_mem_wr);



//Instruction Fetch (IF) Phase
logic [31:0] instruction;
logic [31:0] IF_PC_plus4;
logic [31:0] next_PC_addr = 32'h00400000-4;
logic [31:0] instruction_addr1;
logic [31:0] instruction_addr2;


logic branch_en;
logic [31:0] branch_addr;
logic [1:0] jump;
logic [31:0] jump_addr;
logic [31:0] data1;

assign instruction = i_u_cpu_ins;
assign instruction_addr1 = branch_en ? branch_addr : (next_PC_addr+4);
assign instruction_addr2 = jump[1] ? data1 : (jump[0] ? jump_addr :
instruction_addr1);
assign o_u_cpu_ins_addr = next_PC_addr;

// o_u_cpu_wr_en = i_u_cpu_ins_wr;
//assign o_u_cpu_wr_ins = i_u_cpu_write_ins;

// IF/ID pipeline
logic [31:0] IF_instruction;
always_ff @(posedge i_sys_clock)begin
    if (i_sys_reset) begin
        IF_instruction <= 32'b0;
        IF_PC_plus4 <= 32'h00400000;
        next_PC_addr <= 32'h00400000-4;
    end
```

```
    else begin
        IF_instruction <= instruction;
        next_PC_addr <= instruction_addr2;
        IF_PC_plus4 <= next_PC_addr + 4;
    end
end


// Instruction Decode (ID) Phase
logic [5:0] opcode;
logic [5:0] funct;
logic [4:0] shamt;
logic [1:0] mf;
logic pc_src;
logic ext_op;
logic mult_en;
logic alu_src;
logic [3:0] alu_ctr;
logic [1:0] reg_dst;
logic data_sel;
logic mem_wr;
logic [1:0] branch;
logic word;
logic reg_wr;
logic mem_to_reg;

assign opcode = IF_instruction [31:26];
assign funct = IF_instruction[5:0];
assign shamt = IF_instruction[10:6];


u_ctrlu control_unit(
.i_u_ctrlu_op(opcode),
.i_u_ctrlu_funct(funct),
.o_u_ctrlu_pc_src(pc_src),
.o_u_ctrlu_mf(mf),
.o_u_ctrlu_ext_op(ext_op),
.o_u_ctrlu_jmp(jump),
.o_u_ctrlu_mult_en(mult_en),
.o_u_ctrlu_reg_dst(reg_dst),
.o_u_ctrlu_alu_src(alu_src),
.o_u_ctrlu_alu_ctr(alu_ctr),
.o_u_ctrlu_data_sel(data_sel),
.o_u_ctrlu_mem_wr(mem_wr),
.o_u_ctrlu_branch(branch),
.o_u_ctrlu_word(word),
.o_u_ctrlu_reg_wr(reg_wr),
.o_u_ctrlu_mem_to_reg(mem_to_reg));


logic [4:0] address1;
logic [4:0] address2;
logic [31:0] read_data1;
logic [31:0] read_data2;
logic [31:0] reg_hi = 32'b0;
logic [31:0] reg_lo = 32'b0;
logic [31:0] WB_wr_data;
```

```
logic [31:0] WB_wr_addr;
logic WB_reg_wr;

assign address1 = IF_instruction[25:21];
assign address2 = IF_instruction[20:16];

b_reg register_block(
.i_sys_clock(i_sys_clock),
.i_sys_reset(i_sys_reset),
.i_b_reg_read_addr1(address1),
.i_b_reg_read_addr2(address2),
.i_b_reg_wr_addr(WB_wr_addr),
.i_b_reg_wr_data(WB_wr_data),
.i_b_reg_regwr(WB_reg_wr),
.o_b_reg_read_data1(read_data1),
.o_b_reg_read_data2(read_data2));


logic [31:0] data2;
assign data2 = read_data2;
assign data1 = mf[1] ? reg_hi : (mf[0] ? reg_lo : read_data1);


logic [31:0] extended_imm;
assign extended_imm = (ext_op & IF_instruction[15]) ?
{{16{IF_instruction[15]}},IF_instruction[15:0]}: {16'b0,IF_instruction[15:0]};

logic [4:0] wr_addr;
assign wr_addr = reg_dst[1] ? 5'b11111 : (reg_dst[0] ? IF_instruction[15:11] :
IF_instruction[20:16]);

assign jump_addr = {IF_PC_plus4[31:20],IF_instruction[17:0],2'b00};
//assign jump_addr = {IF_PC_plus4[31:28],IF_instruction[25:0],2'b00};

// ID/EX pipeline
logic ID_mult_en;
logic ID_alu_src;
logic [3:0] ID_alu_ctr;
logic [1:0] ID_reg_dst;
logic ID_data_sel;
logic ID_mem_wr;
logic ID_word;
logic [1:0] ID_branch;
logic ID_reg_wr;
logic ID_mem_to_reg;
logic [31:0] ID_data1;
logic [31:0] ID_data2;
logic [31:0] ID_extended_imm;
logic [4:0] ID_wr_addr;
logic [4:0] ID_shamt;
logic [31:0] ID_PC_plus4;

always_ff @(posedge i_sys_clock)begin
    if (i_sys_reset) begin
        ID_mult_en <= 1'b0;
        ID_alu_src <= 1'b0;
        ID_alu_ctr <= 4'b0;
```

```
        ID_reg_dst <= 2'b0;
        ID_data_sel <= 1'b0;
        ID_mem_wr <= 1'b0;
        ID_word <= 1'b0;
        ID_branch <= 2'b0;
        ID_reg_wr <= 1'b0;
        ID_mem_to_reg <= 1'b0;
        ID_data1 <= 32'b0;
        ID_data2 <= 32'b0;
        ID_extended_imm <= 32'b0;
        ID_wr_addr <= 5'b0;
        ID_shamt <= 5'b0;
        ID_PC_plus4 <= 32'b0;
    end
    else begin
        ID_mult_en <= mult_en;
        ID_alu_src <= alu_src;
        ID_alu_ctr <= alu_ctr;
        ID_reg_dst <= reg_dst;
        ID_data_sel <= data_sel;
        ID_mem_wr <= mem_wr;
        ID_word <= word;
        ID_branch <= branch;
        ID_reg_wr <= reg_wr;
        ID_mem_to_reg <= mem_to_reg;
        ID_data1 <= data1;
        ID_data2 <= data2;
        ID_extended_imm <= extended_imm;
        ID_wr_addr <= wr_addr;
        ID_shamt <= shamt;
        ID_PC_plus4 <= IF_PC_plus4;
    end
end



// Execution (EX) Phase
logic [31:0] operand_1;
logic [31:0] operand_2;
logic [4:0] EX_shamt;
logic [3:0] EX_alu_ctr;
logic alu_branch;
logic [31:0] result;

assign operand_1 = ID_data1;
assign operand_2 = ID_alu_src ? ID_extended_imm : ID_data2;
assign EX_shamt = ID_shamt;
assign EX_alu_ctr = ID_alu_ctr;


b_alu alu_block(
.i_b_alu_operand_1(operand_1),
.i_b_alu_operand_2(operand_2),
.i_b_alu_shamt(EX_shamt),
.i_b_alu_ctrl(EX_alu_ctr),
.o_b_alu_branch(alu_branch),
.o_b_alu_result(result));
```

```
logic [31:0] alu_result;
assign alu_result = result;

logic mult_done;
logic [31:0] reg_hi_data;
logic [31:0] reg_lo_data;

b_mult multiplier_block(
.i_sys_clock(i_sys_clock),
.i_sys_reset(i_sys_reset),
.i_b_mult_start(ID_mult_en),
.i_b_mult_a_in(operand_1),
.i_b_mult_b_in(operand_2),
.o_b_mult_done(mult_done),
.o_b_mult_hi(reg_hi_data),
.o_b_mult_lo(reg_lo_data));



assign branch_en = (ID_branch[0] ~^ alu_branch) & ID_branch[1];
assign branch_addr = {ID_extended_imm << 2} +  ID_PC_plus4 - 4;



// EX/MEM pipeline
logic [31:0] EX_result;
logic [4:0] EX_wr_addr;
logic [31:0] EX_rt;
logic EX_data_sel;
logic EX_mem_wr;
logic EX_word;
logic EX_reg_wr;
logic EX_mem_to_reg;
logic [31:0] EX_PC_plus4;
logic EX_mult_done;
assign EX_mult_done = mult_done;

always_ff @(posedge i_sys_clock)begin
   if (i_sys_reset) begin
      EX_result <= 32'b0;
      EX_wr_addr <= 32'b0;
      EX_rt <= 32'b0;
      EX_data_sel <= 1'b0;
      EX_mem_wr <= 1'b0;
      EX_word <= 1'b0;
      EX_reg_wr <= 1'b0;
      EX_mem_to_reg <= 1'b0;
      EX_PC_plus4 <= 32'b0;
   end
   else begin
      if (EX_mult_done) begin
         reg_hi <= reg_hi_data;
         reg_lo <= reg_lo_data;
      end

      EX_result <= alu_result;
```

```
        EX_wr_addr <= ID_wr_addr;
        EX_rt <= ID_data2;
        EX_data_sel <= ID_data_sel;
        EX_mem_wr <= ID_mem_wr;
        EX_word <= ID_word;
        EX_reg_wr <= ID_reg_wr;
        EX_mem_to_reg <= ID_mem_to_reg;
        EX_PC_plus4 <= ID_PC_plus4;
    end
end


// Memory (MEM) Phase
logic [31:0] data;
logic [31:0] data3;
assign data = i_u_cpu_data;
assign data3 = EX_data_sel ? EX_PC_plus4 : EX_result;


// MEM/WB pipeline
logic [31:0] MEM_data2;
logic [4:0] MEM_wr_addr;
logic MEM_reg_wr;
logic MEM_mem_to_reg;


always_ff @(posedge i_sys_clock)begin
    if (i_sys_reset) begin
        o_u_cpu_data_addr <= 32'b0;
        o_u_cpu_write_data <= 32'b0;
        o_u_cpu_mem_wr <= 1'b0;
        o_u_cpu_word <= 1'b0;
        MEM_data2 <= 32'b0;
        MEM_wr_addr <= 5'b0;
        MEM_reg_wr <= 1'b0;
        MEM_mem_to_reg <= 1'b0;
    end
    else begin
        o_u_cpu_data_addr <= EX_result;
        o_u_cpu_write_data <= EX_rt;
        o_u_cpu_mem_wr <= EX_mem_wr;
        o_u_cpu_word <= EX_word;
        MEM_data2 <= data3;
        MEM_wr_addr <= EX_wr_addr;
        MEM_reg_wr <= EX_reg_wr;
        MEM_mem_to_reg <= EX_mem_to_reg;
    end
end


// Write Back (WB) Phase
logic [31:0] wr_data;
assign wr_data = MEM_mem_to_reg ? data : MEM_data2;

always_ff @(posedge i_sys_clock)begin
    if (i_sys_reset) begin
        WB_wr_data <= 32'b0;
```

```
      WB_wr_addr <= 5'b0;
      WB_reg_wr <= 1'b0;
   end
   else begin
      WB_wr_data <= wr_data;
      WB_wr_addr <= MEM_wr_addr;
      WB_reg_wr <= MEM_reg_wr;
   end
end
Endmodule
```

4.1.0.6 Post-synthesis Schematic

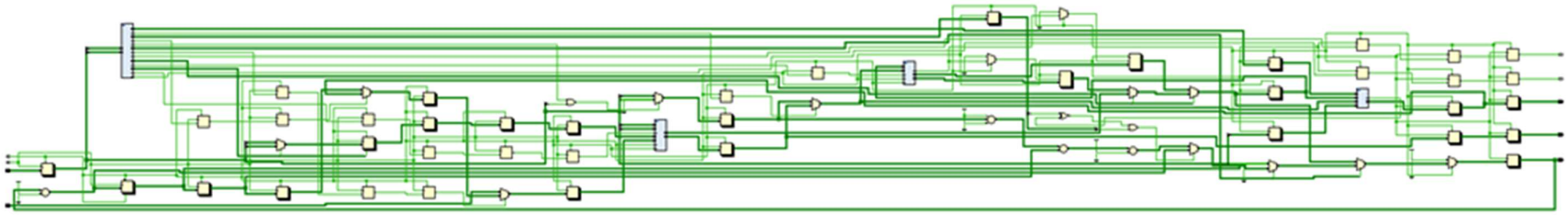


Diagram 4.1.6.1 CPU unit post-synthesis schematic

4.1.0.7 Test plan

No.	Test Case	Description of Test Vector Generation	Expected Output	Status
1	Reset	1. Set i_sys_reset = 1	1. o_u_cpu_mem_wr = 0 2. o_u_cpu_ins_addr[31:0] = 0x00400000 - 4 3. o_u_cpu_data_addr[31:0] = 0 4. o_u_cpu_write_data[31:0] = 0 5. o_u_cpu_word = 0	PASS
2	Immediate addressing mode	1. Set i_sys_reset = 0 2. Set i_u_cpu_ins[31:0] = 0x340804D2 (ori $t0, $zero, 1234)	1. o_u_cpu_mem_wr = 0 2. o_u_cpu_ins_addr[31:0] = PC+4 3. o_u_cpu_data_addr[31:0] = 32'h000004D2 4. o_u_cpu_write_data[31:0] = 0 5. o_u_cpu_word = x	PASS
3	Register addressing mode	1. Set i_u_cpu_ins[31:0] = 0x01004822 (sub $t1, $t0, $zero)	1. o_u_cpu_mem_wr = 0 2. o_u_cpu_ins_addr[31:0] = PC+4 3. o_u_cpu_data_addr[31:0] = 32'h000004D2 4. o_u_cpu_write_data[31:0] = 0 5. o_u_cpu_word = x	PASS
4	Base addressing mode (Store word)	1. Set i_u_cpu_ins[31:0] = 0xADEA0000 (sw $t2, 0($t7))	1. o_u_cpu_word = 1 2. o_u_cpu_mem_wr = 1 3. o_u_cpu_ins_addr[31:0] = PC+4 4. o_u_cpu_data_addr[31:0] = 03E80000 5. o_u_cpu_write_data[31:0] = 12341E61	PASS
5	Base addressing mode (Store byte)	1. Set i_u_cpu_ins[31:0] = 0xA1EB0001 (sb $t3, 1($t7))	1. o_u_cpu_word = 0 2. o_u_cpu_mem_wr = 1 3. o_u_cpu_ins_addr[31:0] = PC+4 4. o_u_cpu_data_addr[31:0] = 03E80001 5. o_u_cpu_write_data[31:0] = 0000003a	PASS

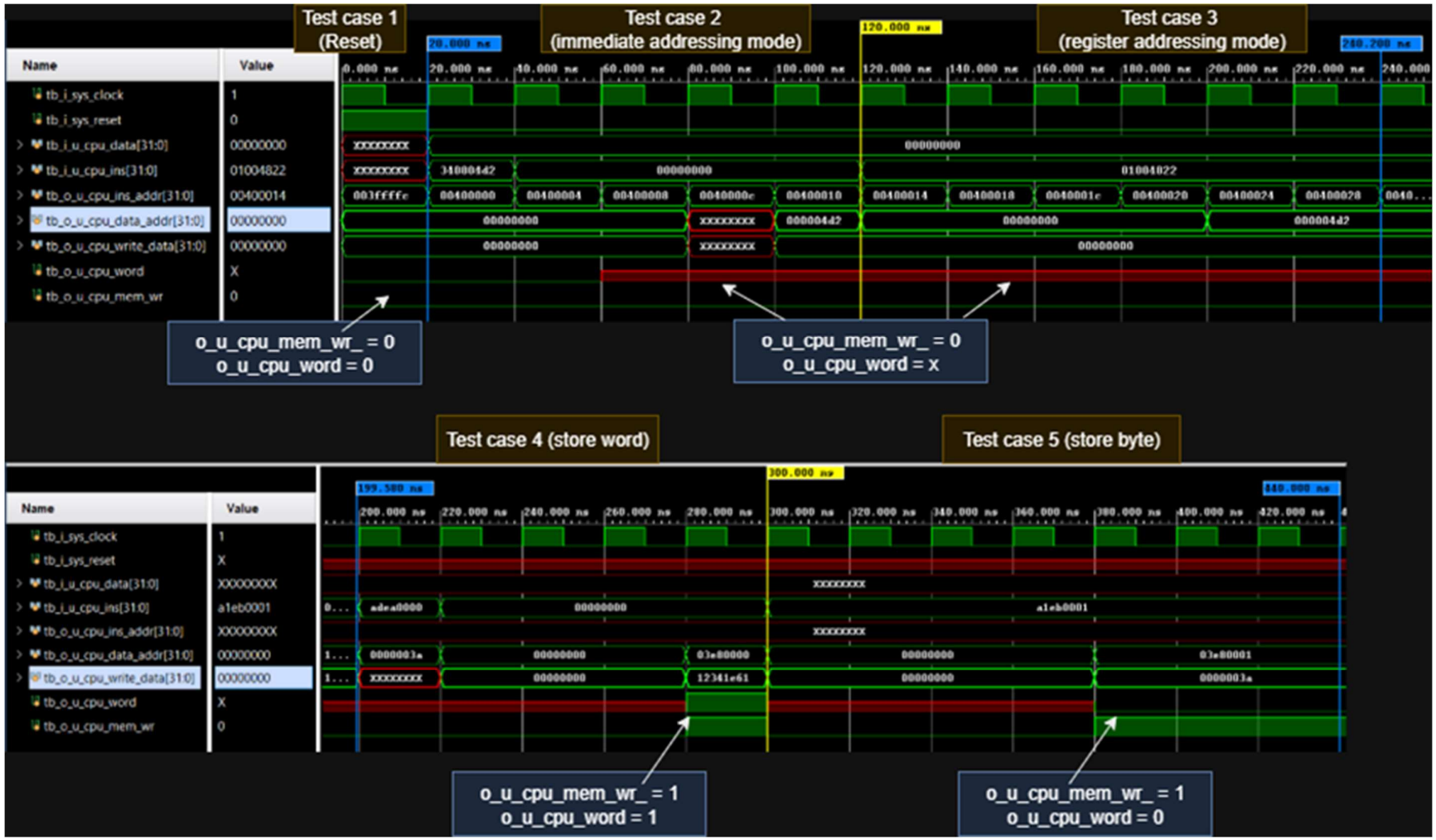
4.1.0.8 Testbench

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Ng Jing Xuan
//
// Create Date: 08.09.2025 23:18:48
```

```
// File Name: tb_u_cpu.sv
// Module Name: tb_u_cpu
// Project Name: MIPS ISA Pipeline processor
// Code Type: Behavioural
// Description: Testbench for CPU Unit
//
//////////////////////////////////////////////////////////////////////////////////
module tb_u_cpu();
parameter CC = 20;
logic tb_i_sys_clock;
logic tb_i_sys_reset;
logic [31:0] tb_i_u_cpu_data;
logic [31:0] tb_i_u_cpu_ins;
logic [31:0] tb_o_u_cpu_ins_addr;
logic [31:0] tb_o_u_cpu_data_addr;
logic [31:0] tb_o_u_cpu_write_data;
logic tb_o_u_cpu_word;
logic tb_o_u_cpu_mem_wr;


u_cpu dut_u_cpu
(.i_sys_clock(tb_i_sys_clock),
.i_sys_reset(tb_i_sys_reset),
.i_u_cpu_data(tb_i_u_cpu_data),
.i_u_cpu_ins(tb_i_u_cpu_ins),
.o_u_cpu_ins_addr(tb_o_u_cpu_ins_addr),
.o_u_cpu_data_addr(tb_o_u_cpu_data_addr),
.o_u_cpu_write_data(tb_o_u_cpu_write_data),
.o_u_cpu_word(tb_o_u_cpu_word),
.o_u_cpu_mem_wr(tb_o_u_cpu_mem_wr));


initial begin
tb_i_sys_clock = 1'b1;
forever #(CC/2) tb_i_sys_clock = ~tb_i_sys_clock ;
end

initial begin
tb_i_sys_reset = 1'b1;
repeat (1) @(posedge tb_i_sys_clock);

tb_i_sys_reset = 1'b0;
tb_i_u_cpu_data = 32'h0;
tb_i_u_cpu_ins = 32'h340804D2;      //ori $t0, $zero, 1234
repeat (1) @(posedge tb_i_sys_clock);

tb_i_u_cpu_ins = 32'h0;
repeat (4) @(posedge tb_i_sys_clock); //nop x3

tb_i_u_cpu_ins = 32'h01004822;       //sub $t1, $t0, $zero
```

```
repeat (1) @(posedge tb_i_sys_clock);
////////////////////////////////////////////////////////////////////////////
tb_i_u_cpu_ins = 32'h3C0F03E8;       //lui $t7, 1000
repeat (1) @(posedge tb_i_sys_clock);

tb_i_u_cpu_ins = 32'h3C0A1234;       //lui $t2, 0x1234
repeat (1) @(posedge tb_i_sys_clock);

tb_i_u_cpu_ins = 32'h0;
repeat (3) @(posedge tb_i_sys_clock); //nop x3

tb_i_u_cpu_ins = 32'h214A1E61;       //addi $t2, $t2, 7777
repeat (1) @(posedge tb_i_sys_clock);

tb_i_u_cpu_ins = 32'h200B003A;       //addi $t3, $zero, 58
repeat (1) @(posedge tb_i_sys_clock);

tb_i_u_cpu_ins = 32'h0; //nop x3
repeat (3) @(posedge tb_i_sys_clock);

tb_i_u_cpu_ins = 32'hADEA0000;       //sw $t2, 0($t7)
repeat (1) @(posedge tb_i_sys_clock);

tb_i_u_cpu_ins = 32'h0; //nop x3
repeat (4) @(posedge tb_i_sys_clock);

tb_i_u_cpu_ins = 32'hA1EB0001;       //sb $t3, 1($t7)
end
endmodule
```

4.1.0.9 Simulation Result



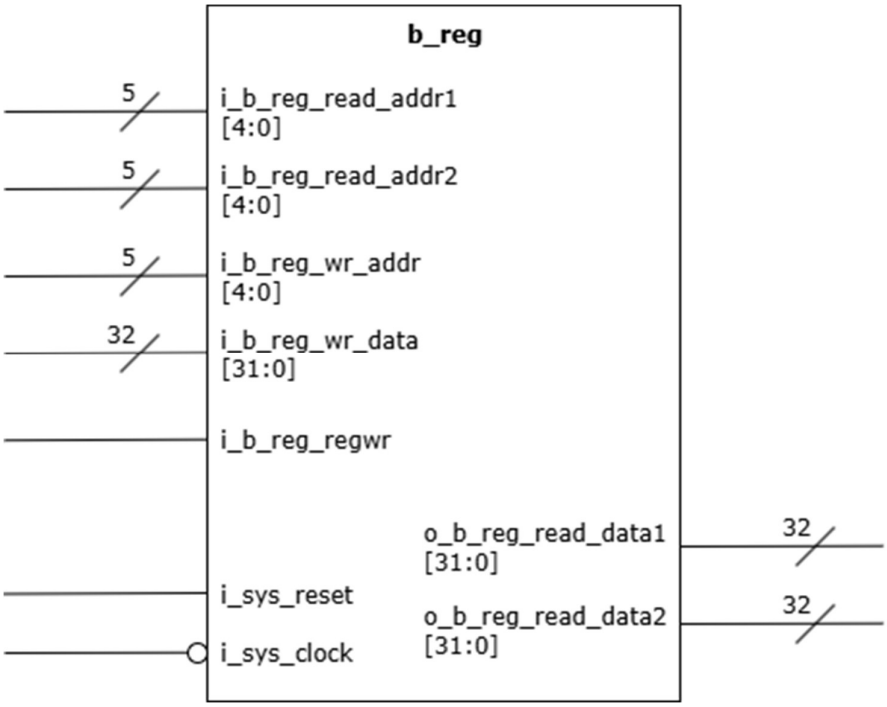
4.1.1 Register File Block

4.1.1.1 Functionality/Feature

- Two read ports to read two different register simultaneously
- Write one register at a time
- 5-bit register address (0-31)
- Every register is 32-bit
- Has 31 register ($1-$31) since $0 is a hard wire to ground

4.1.1.2 Block interface and I/O pin description

4.1.1.2.1 Register File Block Interface



4.1.1.2.2 I/O pin description

Pin name: **Pin class:** **Pin function:**	i_sys_reset Control To reset the memory	**Source** **→Destination:**	On/Reset button → b_reg
Pin name: **Pin class:** **Pin function:**	i_sys_clock Control To provide a system clock to block	**Source** **→Destination:**	Clock generater → b_reg
Pin name: **Pin class:** **Pin function:**	i_b_reg_read_addr1 control to select which register to read	**Source** **→Destination:**	IFID_reg → b_reg
Pin name: **Pin class:** **Pin function:**	i_b_reg_read_addr2 control to select which register to read	**Source** **→Destination:**	IFID_reg → b_reg
Pin name: **Pin class:** **Pin function:**	i_b_reg_wr_addr control to select which register to write	**Source** **→Destination:**	MEMWR_reg → b_reg

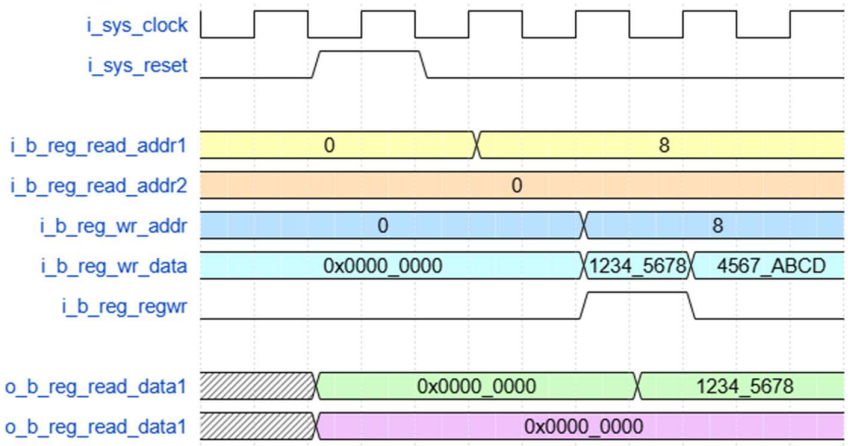
Pin name: **Pin class:** **Pin function:**	i_b_reg_wr_data data the data to write to the register	**Source →Destination:**	MEMWR_reg → b_reg
Pin name: **Pin class:** **Pin function:**	i_b_reg_regwr control enable write to the register	**Source →Destination:**	MEMWR_reg → b_reg
Pin name: **Pin class:** **Pin function:**	o_b_reg_read_data1 data the value get from the register	**Source →Destination:**	b_reg → IDEX_reg
Pin name: **Pin class:** **Pin function:**	o_b_reg_read_data2 data the value get from the register	**Source →Destination:**	b_reg → IDEX_reg

4.1.1.3 Internal Operation

Operation	**Condition Trigger**	**Action**	**Description**
Reset	i_sys_reset = 1	All register reset	
Read port1	Always active	o_b_reg_read_data1 = reg[i_b_reg_read_addr1]	There is no read enable and both are combinational operation.
Read port2	Always active	o_b_reg_read_data2 = reg[i_b_reg_read_addr2]	
Write operation	Falling edge of i_sys_clock AND i_b_reg_regwr = 1	Reg[i_b_reg_wr_addr] ← i_b_reg_wr_data	
Write to $0	Falling edge of i_sys_clock AND i_b_reg_regwr = 1 AND i_b_reg_wr_addr = 0	No operation	Since address 0 is not connected to any register therefore noting will happen

4.1.1.4 Timing Requirement

Operation	**Time requirement**	**Description**
read	Within a cycle (propagation delay)	Read is a conbinational operation
write	1 cycle	It active when falling edge of the clock


i_sys_clock
i_sys_reset
i_b_reg_read_addr1
0
8
i_b_reg_read_addr2
0
i_b_reg_wr_addr
0
8
i_b_reg_wr_data
0x0000_0000
1234_5678
4567_ABCD
i_b_reg_regwr
o_b_reg_read_data1
0x0000_0000
1234_5678
o_b_reg_read_data1
0x0000_0000

4.1.1.5 Pre-synthesis Schematic

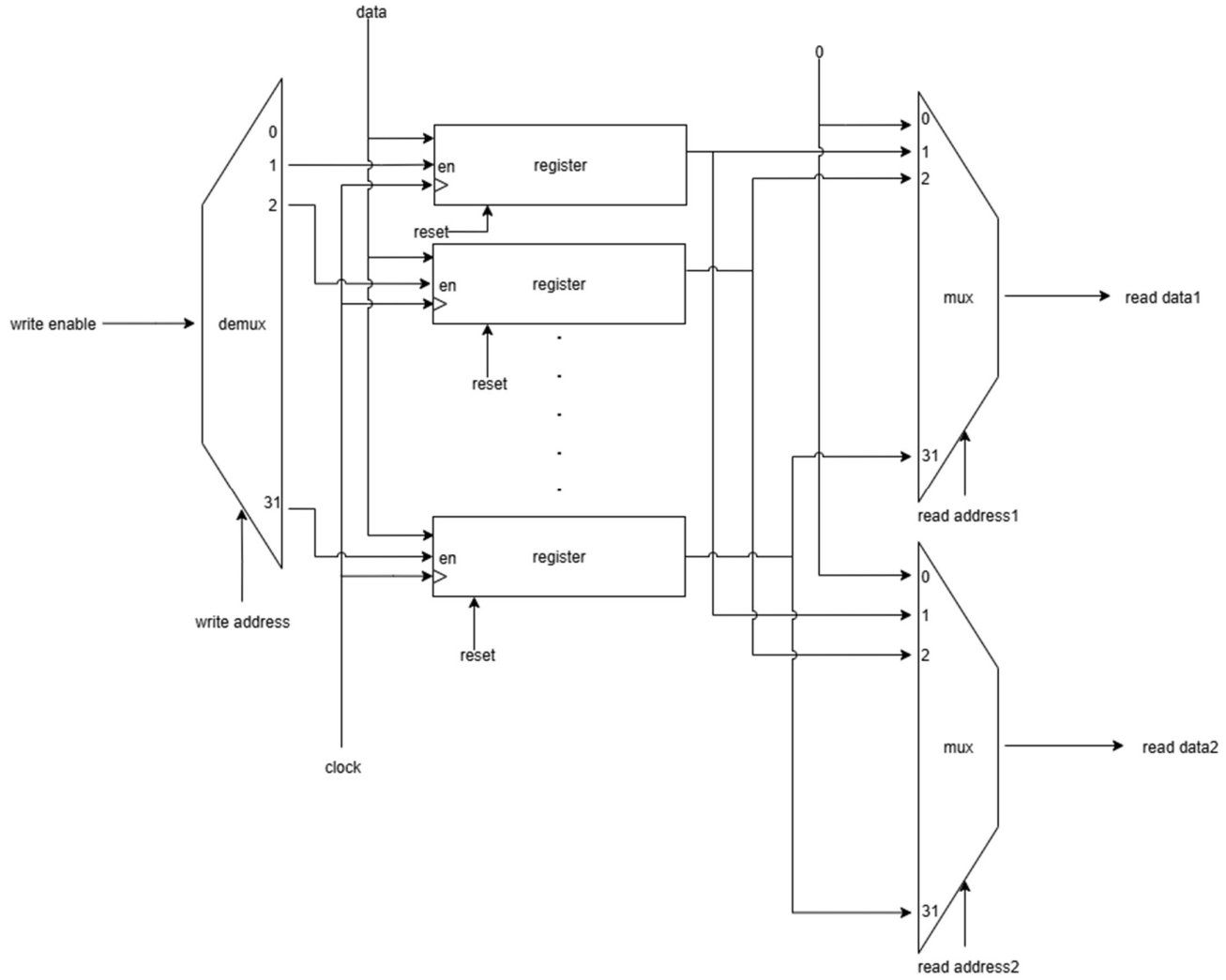


Diagram 4.1.1.5.: register file pre-schematic

4.1.1.6 SV Model

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Chow Bin Lin
//
// Create Date: 02.09.2025 16:35:18
// File Name: b_reg.sv
// Module Name: b_reg
// Project Name: MIPS ISA Pipeline processor
// Code Type: RTL level
// Description: Modeling of register block
//
//////////////////////////////////////////////////////////////////////////////////

module b_reg(
input logic i_sys_clock,
input logic i_sys_reset,

input logic[4:0] i_b_reg_read_addr1,
input logic[4:0] i_b_reg_read_addr2,
input logic[4:0] i_b_reg_wr_addr,
input logic[31:0] i_b_reg_wr_data,
input logic i_b_reg_regwr,

output logic[31:0] o_b_reg_read_data1,
output logic[31:0] o_b_reg_read_data2
);

logic [31:0] register[31:1];
logic [5:0] i;
// read data1
assign o_b_reg_read_data1 = (i_b_reg_read_addr1 == 5'b0)? 32'b0:
register[i_b_reg_read_addr1];
// read data2
assign o_b_reg_read_data2 = (i_b_reg_read_addr2 == 5'b0)? 32'b0:
register[i_b_reg_read_addr2];


always_ff @(negedge i_sys_clock)begin // write
   if(i_sys_reset)
      for(i=0;i<32;i++)
         case(i)
         0:;
         28: register[28]<=32'h1000_4000;
         29: register[29]<=32'h7fff_fffc;
         default:register[i] <= 32'b0;
         endcase
   else begin
      if(i_b_reg_regwr && i_b_reg_wr_addr != 5'b0 )
         register[i_b_reg_wr_addr] <= i_b_reg_wr_data;
   end
end
endmodule
```

4.1.1.7 post-synthesis Schematic

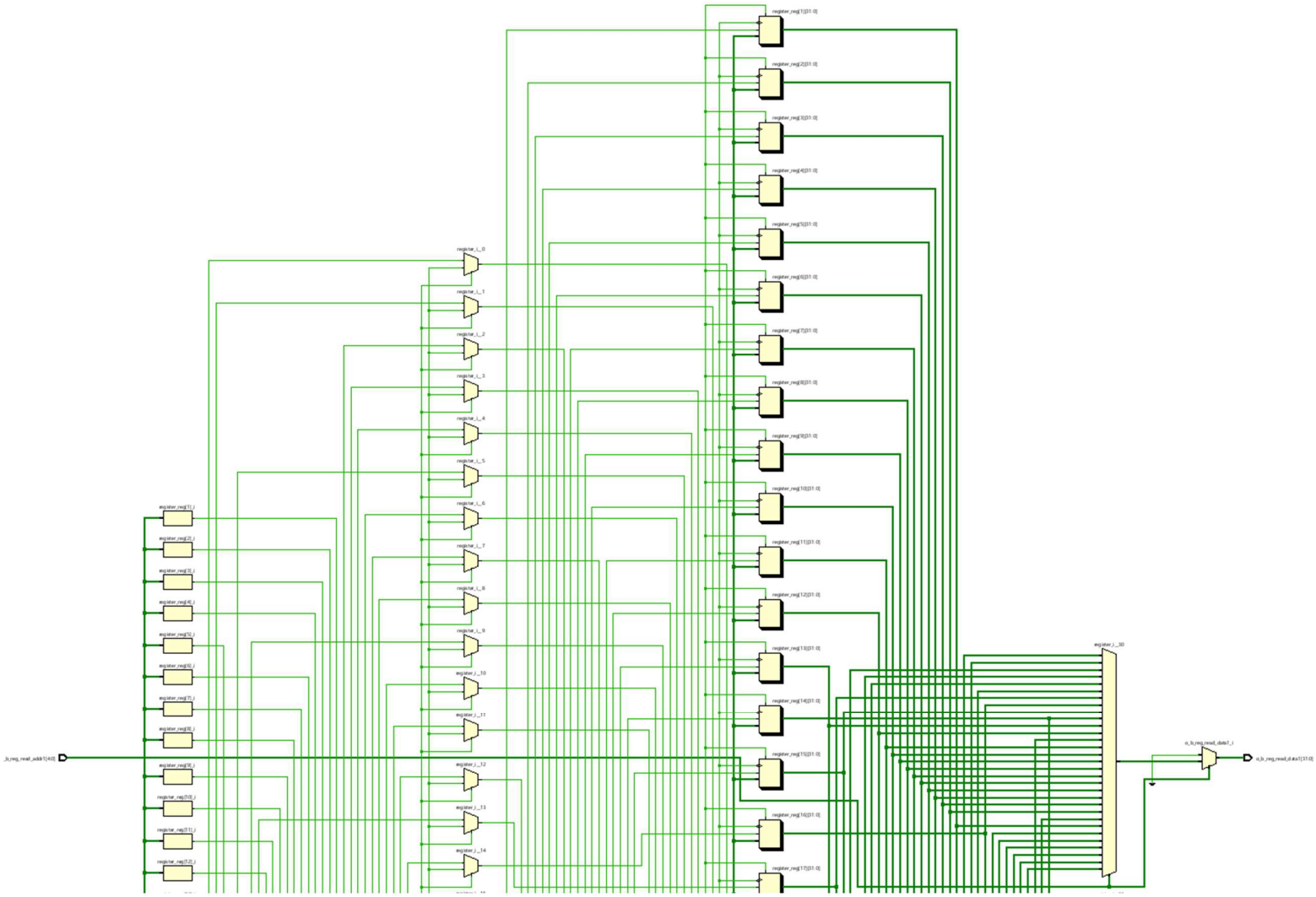


Diagram 4.1.1.7.1: register file schematic

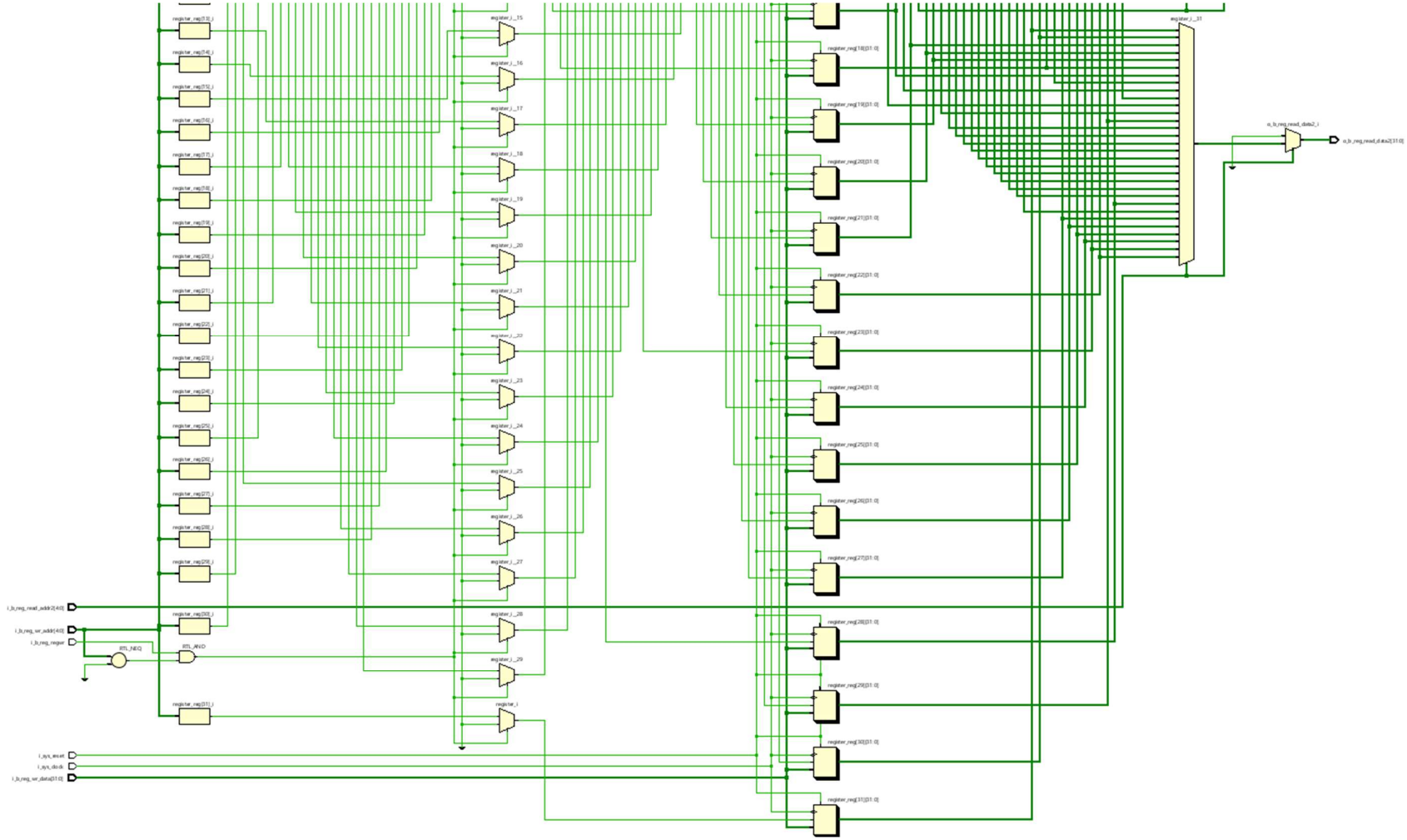


Diagram 4.1.1.7.2: register file schematic

4.1.1.8 Test Plan

No.	Test Case	Description of test vector Generation	Expected Output	Pass/ Fail
1	Reset	i_sys_reset = 1'b1 i_b_reg_read_addr1 <= 5'b0_0000($zero) i_b_reg_read_addr2 <= 5'b0_0000($zero) i_b_reg_wr_addr <= 5'b0_0000 i_b_reg_wr_data <= 32'h0_0000 i_b_reg_regwr <= 1'b0	$gp = 32'h1000_4000 $sp = 32'h7FFF_FFFC else = 32'b0000_0000	PASS
2	Write/ read	1. i_sys_reset = 1'b0 i_b_reg_wr_addr <= 5'b01000 ($t0) i_b_reg_wr_data <= 32'h0000_1234 i_b_reg_regwr <= 1'b1 i_b_reg_read_addr1 <= 5'b01000 ($t0) i_b_reg_read_addr2 <= 5'b00000 ($zero) 2. wait one cycle	i_b_reg_read_data1 = 32'h0000_1234 i_b_reg_read_data2 = 32'h0000_0000	PASS
3	Write without enable	1. i_sys_reset = 1'b0 i_b_reg_wr_addr <= 5'b01001 ($t1) i_b_reg_wr_data <= 32'h0000_5678 i_b_reg_regwr <= 1'b0 i_b_reg_read_addr1 <= 5'b01000 ($t0) i_b_reg_read_addr2 <= 5'b01001 ($t1) 2. wait one cycle	i_b_reg_read_data1 = 32'h0000_1234 i_b_reg_read_data2 = 32'h0000_0000	PASS

4	Write to $0	1. i_sys_reset = 1'b0 i_b_reg_wr_addr <= 5'b00000 ($zero) i_b_reg_wr_data <= 32'h0000_9876 i_b_reg_regwr <= 1'b1 i_b_reg_read_addr1 <= 5'b00000 ($zero) i_b_reg_read_addr2 <= 5'b01001 ($t1) 2. wait one cycle	i_b_reg_read_data1 = 32'h0000_0000 i_b_reg_read_data2 = 32'h0000_0000	PASS

Table 4.1.1.8: Register File Test Plan

4.1.1.8 Testbench and Simulation result

4.1.1.8.1 Testbench

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Chow Bin Lin
//
// Create Date: 02.09.2025 16:35:18
// File Name: tb_b_reg.sv
// Module Name: tb_b_reg
// Project Name: MIPS ISA Pipeline processor
// Code Type: Behavioural
// Description: Testbench for Register block
//
//////////////////////////////////////////////////////////////////////////////////


module tb_b_reg(
    );
    parameter CC = 10;
    logic clk;
    logic[3:0] i;
    logic[3:0] test_case;


    logic tb_i_sys_clock;
    logic tb_i_sys_reset;
    logic[4:0] tb_i_b_reg_read_addr1;
    logic[4:0] tb_i_b_reg_read_addr2;
    logic[4:0] tb_i_b_reg_wr_addr;
    logic[31:0] tb_i_b_reg_wr_data;
    logic tb_i_b_reg_regwr;
    logic[31:0] tb_o_b_reg_read_data1;
    logic[31:0] tb_o_b_reg_read_data2;

    b_reg reg_file(
    .i_sys_clock(tb_i_sys_clock),
    .i_sys_reset(tb_i_sys_reset),

    .i_b_reg_read_addr1(tb_i_b_reg_read_addr1),
    .i_b_reg_read_addr2(tb_i_b_reg_read_addr2),
    .i_b_reg_wr_addr(tb_i_b_reg_wr_addr),
    .i_b_reg_wr_data(tb_i_b_reg_wr_data),
    .i_b_reg_regwr(tb_i_b_reg_regwr),

    .o_b_reg_read_data1(tb_o_b_reg_read_data1),
    .o_b_reg_read_data2(tb_o_b_reg_read_data2)
    );

    always #(CC/2) clk = ~clk;
    assign tb_i_sys_clock = clk;

    initial begin
        clk = 1;

        for( i = 1; i <5 ; i++)@(posedge clk)
```

```
        test_case = i;
    end

    always_comb begin

    case(test_case)
    1:begin
    tb_i_sys_reset =1'b1;
    tb_i_b_reg_read_addr1 = 5'b0_0000;
    tb_i_b_reg_read_addr2 = 5'b0_0000;
    tb_i_b_reg_wr_addr = 5'b0_0000;
    tb_i_b_reg_wr_data = 32'h0000_0000;
    tb_i_b_reg_regwr = 1'b0;
    end
    2:begin
    tb_i_sys_reset =1'b0;
    tb_i_b_reg_read_addr1 = 5'b0_1000;
    tb_i_b_reg_read_addr2 = 5'b0_0000;
    tb_i_b_reg_wr_addr = 5'b0_1000;
    tb_i_b_reg_wr_data = 32'h0000_1234;
    tb_i_b_reg_regwr = 1'b1;
    end
    3:
    begin
    tb_i_sys_reset =1'b0;
    tb_i_b_reg_read_addr1 = 5'b0_1000;
    tb_i_b_reg_read_addr2 = 5'b0_1001;
    tb_i_b_reg_wr_addr = 5'b0_1001;
    tb_i_b_reg_wr_data = 32'h0000_5678;
    tb_i_b_reg_regwr = 1'b0;
    end
    4:
    begin
    tb_i_sys_reset =1'b0;
    tb_i_b_reg_read_addr1 = 5'b0_0000;
    tb_i_b_reg_read_addr2 = 5'b0_0000;
    tb_i_b_reg_wr_addr = 5'b0_0000;
    tb_i_b_reg_wr_data = 32'h0000_9876;
    tb_i_b_reg_regwr = 1'b1;
    end
    endcase
    end

endmodule
```

4.1.1.8.2 Simulation result

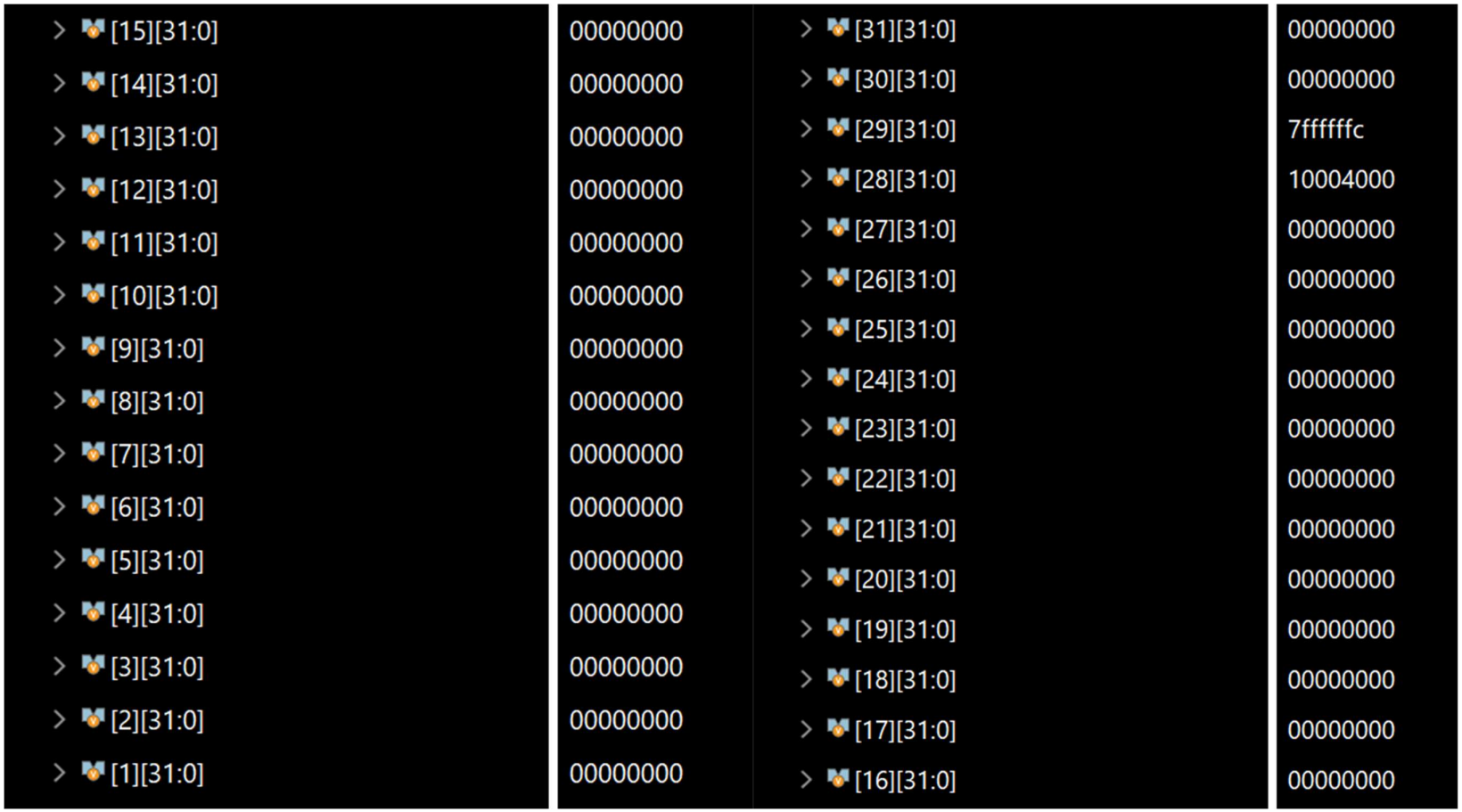


Diagram 4.1.1.8.2.1: register file simulation results after reset

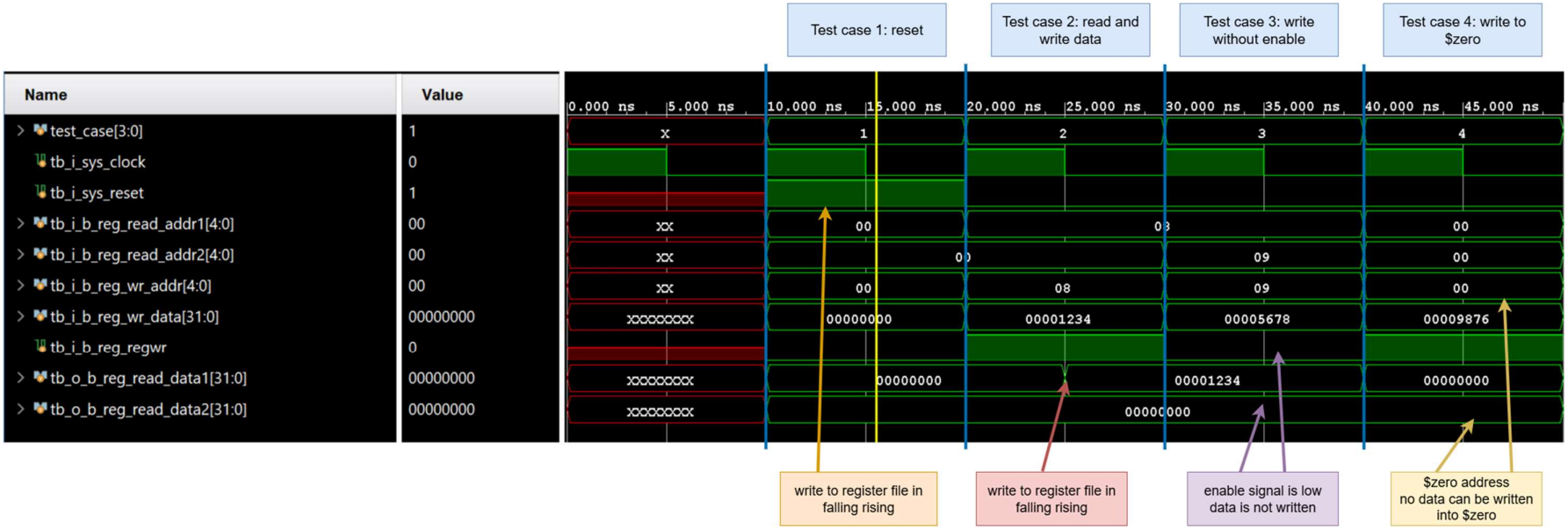


Diagram 4.1.1.8.2.2: register file simulation results

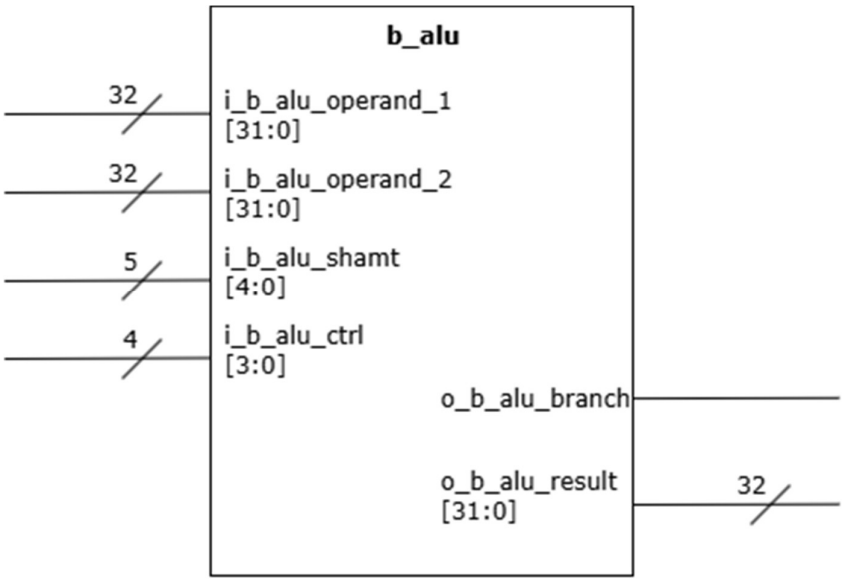
4.1.2 ALU Block

4.1.2.1 Functionality/Feature

- A full combinational curcuit block
- Allow to do some simple arithmetic (+,-, <<,>>) and logical (and, or, nor, xor)

4.1.2.2 Block interface and I/O pin description

4.1.2.2.1 ALU block interface



4.1.2.2.2 I/O pin description

Pin name: **Pin class:** **Pin function:**	i_b_alu_operand_1 data The value used to calculate result	**Source** **→Destination:**	IDEX_reg → b_alu
Pin name: **Pin class:** **Pin function:**	i_b_alu_operand_2 data The value used to calculate result	**Source** **→Destination:**	IDEX_reg → b_alu
Pin name: **Pin class:** **Pin function:**	i_b_alu_shamt control To decide shift amount	**Source** **→Destination:**	IDEX_reg → b_reg
Pin name: **Pin class:** **Pin function:**	i_b_alu_ctrl Control To decode the operator	**Source** **→Destination:**	b_alu_ctrl → b_alu
Pin name: **Pin class:** **Pin function:**	o_b_alu_branch data To check is the result is zero	**Source** **→Destination:**	b_alu → EXMEM_reg
Pin name: **Pin class:** **Pin function:**	o_b_alu_result data the result after the calculation	**Source** **→Destination:**	b_alu → EXMEM_reg

4.1.2.3 Internal Operation

Operation	**Condition Trigger**	**Action**
Add	i_b_alu_ctrl = 0000	Result = operand1 + operand2
Subtract	i_b_alu_ctrl = 0010	Result = operand1 - operand2
Bitwise AND	i_b_alu_ctrl = 0100	Result = operand1 & operand2
Bitwise OR	i_b_alu_ctrl = 0101	Result = operand1 | operand2
Bitwise XOR	i_b_alu_ctrl = 0110	Result = operand1 ^ operand2
Bitwise NOR	i_b_alu_ctrl = 0111	Result = ~(operand1 | operand2)
Set on less than	i_b_alu_ctrl = 1010	Result = {31{0},(operand1 + operand2)[31]}
Swift left logical	i_b_alu_ctrl = 0001	Result = operand2 << shamt
Swift right logical	i_b_alu_ctrl = 0011	Result = operand2 >> shamt
load upper immdiate	i_b_alu_ctrl = 1111	Result = operand2 << 16

4.1.2.4 Pre-synthesis Schematic

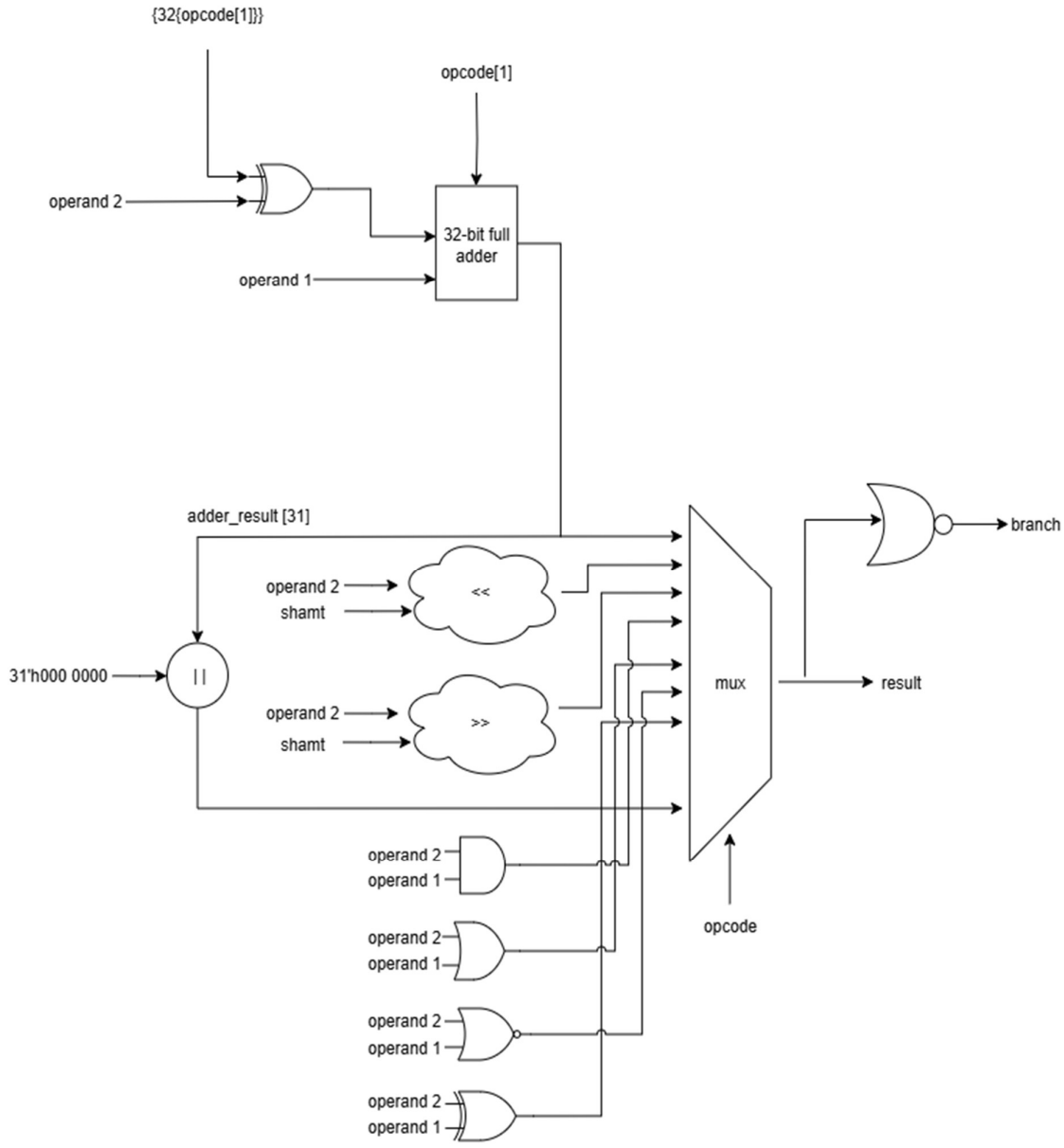


Diagram 4.1.2.4.1: ALU pre-schematic

4.1.2.5 System Verilog Model

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Chow Bin Lin
//
// Create Date: 02.09.2025 16:35:18
// File Name: b_alu.sv
// Module Name: b_alu
// Project Name: MIPS ISA Pipeline processor
// Code Type: RTL level
// Description: Modeling of ALU block
//
//////////////////////////////////////////////////////////////////////////////////


module b_alu(
input logic[31:0] i_b_alu_operand_1,
input logic[31:0] i_b_alu_operand_2,
input logic[4:0] i_b_alu_shamt,
input logic[3:0] i_b_alu_ctrl,

output logic o_b_alu_branch,
output logic[31:0] o_b_alu_result
   );

logic[31:0] adder_temp;
logic[31:0] shift_reg[5:0];
logic[4:0] shift_bit;
logic slt_result;

assign o_b_alu_branch = ~(|o_b_alu_result);
assign shift_bit = &i_b_alu_ctrl? 5'b10000: i_b_alu_shamt;
always_comb begin
   adder_temp = i_b_alu_operand_1 + (i_b_alu_operand_2 ^
{32{i_b_alu_ctrl[1]}})+i_b_alu_ctrl[1];
   shift_reg[0] = i_b_alu_operand_2;

   // since in slt case the overflow will only happen when both operand sign are different
   // and neg(msb = 1)<pos(msb = 0) is true(1)
   // pos(msb = 1)<neg(msb = 0) is false(0)
   // it show the relationship of the result and msb of operand1
   slt_result =
(i_b_alu_operand_1[31]^i_b_alu_operand_2[31])?i_b_alu_operand_1[31]:adder_temp[
31];

   case(i_b_alu_ctrl)
   4'b0000,4'b0010: o_b_alu_result = adder_temp; // add and substract
   4'b0100: o_b_alu_result = i_b_alu_operand_1 & i_b_alu_operand_2;//AND
   4'b0101: o_b_alu_result = i_b_alu_operand_1 | i_b_alu_operand_2;//OR
   4'b0110: o_b_alu_result = i_b_alu_operand_1 ^ i_b_alu_operand_2;//XOR
   4'b0111: o_b_alu_result = ~(i_b_alu_operand_1 | i_b_alu_operand_2);//NOR
   4'b1010: o_b_alu_result = {31'b0,slt_result};//slt

   4'b0001,4'b1111: begin // <<
            for (int col = 0; col < 5; col++) begin
               for (int row = 0; row < 32; row++)begin
```

```
                    if (row < 2**col)

                        shift_reg[col+1][row] = shift_bit[col] ? 1'b0 : shift_reg[col][row];
                    else

                        shift_reg[col+1][row] = shift_bit[col] ? shift_reg[col][row-2**col] :
shift_reg[col][row];
                  end
               end
               o_b_alu_result = shift_reg[5];
            end
   4'b0011: begin // >>
             for (int col = 0; col < 5; col++) begin
                for (int row = 0; row < 32; row++)begin
                    if (row >= 32 - 2**col)

                        shift_reg[col+1][row] = shift_bit[col] ? 1'b0 : shift_reg[col][row];
                    else

                        shift_reg[col+1][row] = shift_bit[col] ? shift_reg[col][row+2**col] :
shift_reg[col][row];
                  end
               end
               o_b_alu_result = shift_reg[5];
            end
   default: o_b_alu_result = 'x;
   endcase
end
endmodule
```

4.1.2.6 Post-synthesis Schematic Diagram

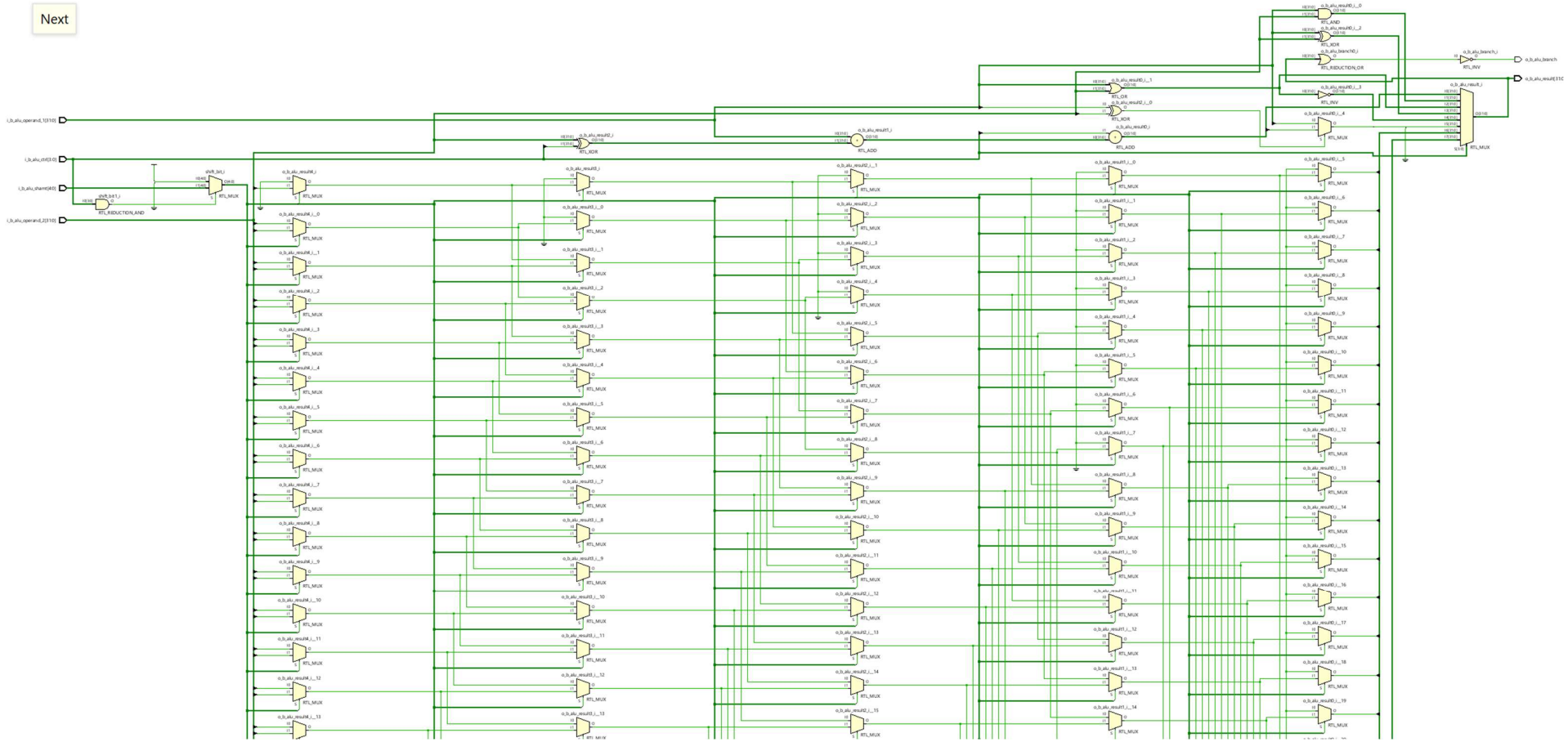


Diagram 4.1.2.6.1: ALU schematic

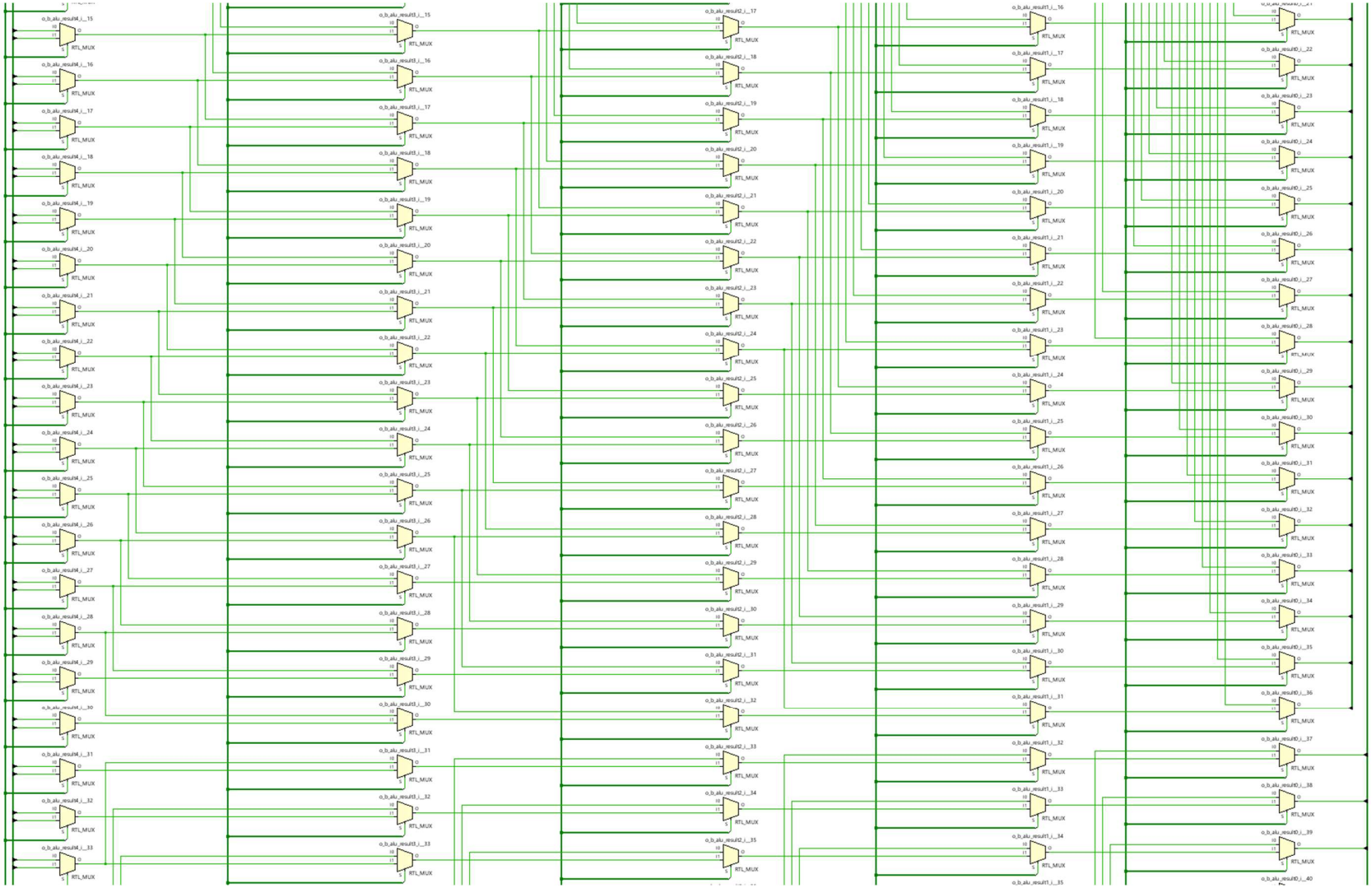


Diagram 4.1.2.6.2: ALU schematic

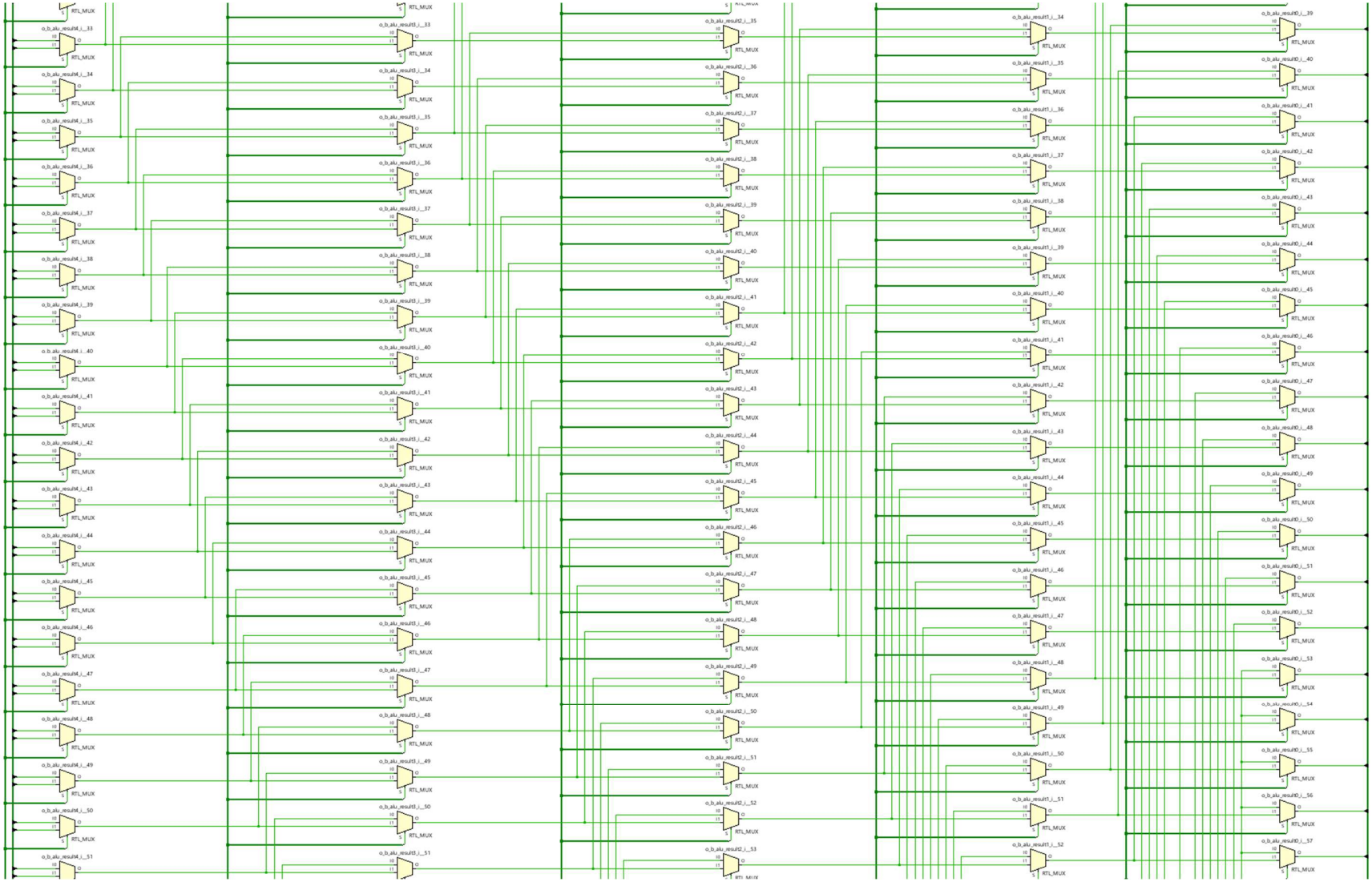


Diagram 4.1.2.6.3: ALU schematic

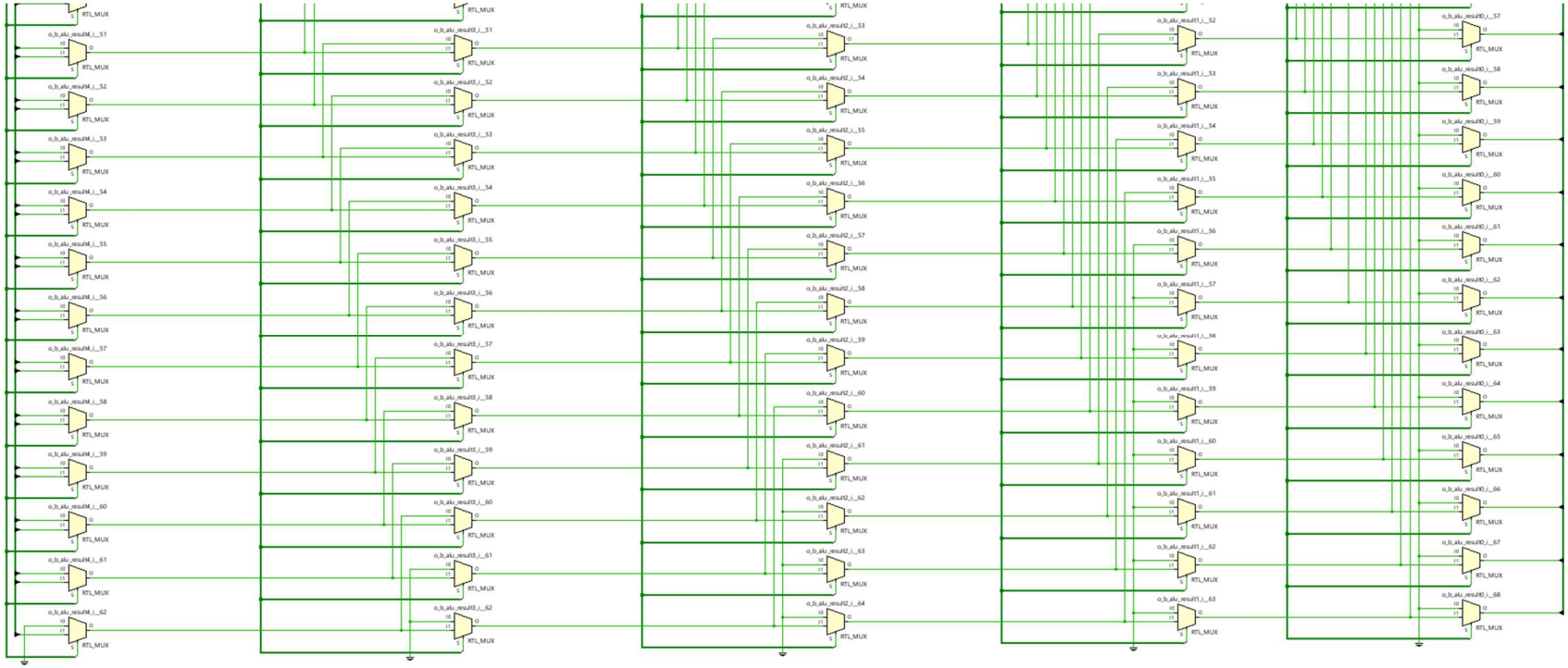


Diagram 4.1.2.6.4: ALU schematic

4.1.2.7 Test Plan

No.	Test Case	Description of test vector Generation	Expected Output	Pass/ Fail
1	Add	i_b_alu_operand_1 = 32'h0000 0001 i_b_alu_operand_2 = 32'h0000 0002 i_b_alu_ctrl = 4'b0000	o_b_alu_result =32'h0000 0003 o_b_alu_branch =0	PASS
2	Subtract	i_b_alu_operand_1 = 32'h0000 0001 i_b_alu_operand_2 = 32'h0000 0002 i_b_alu_ctrl = 4'b0010	o_b_alu_result =32'hFFFF FFFF o_b_alu_branch =0	PASS
3	Bitwise AND	i_b_alu_operand_1 = 32'h0000 0001 i_b_alu_operand_2 = 32'h0000 0002 i_b_alu_ctrl <= 4'b0100	o_b_alu_result =32'h0000 0000 o_b_alu_branch =1	PASS
4	Bitwise OR	i_b_alu_operand_1 = 32'h0000 0001 i_b_alu_operand_2 = 32'h0000 0002 i_b_alu_ctrl = 4'b0101	o_b_alu_result =32'h0000 0003 o_b_alu_branch =0	PASS
5	Bitwise XOR	i_b_alu_operand_1 = 32'h0000 0001 i_b_alu_operand_2 = 32'h0000 0002 i_b_alu_ctrl = 4'b0110	o_b_alu_result =32'h0000 0003 o_b_alu_branch =0	PASS
6	Bitwise NOR	i_b_alu_operand_1 = 32'h0000 0001 i_b_alu_operand_2 = 32'h0000 0002 i_b_alu_ctrl <= 4'b0111	o_b_alu_result =32'hFFFF FFFC o_b_alu_branch =0	PASS
7	Set on less than	i_b_alu_operand_1 = 32'h0000 0001 i_b_alu_operand_2 = 32'h0000 0002 i_b_alu_ctrl = 4'b1010	o_b_alu_result =32'h0000 0001 o_b_alu_branch =0	PASS

8	Swift left logical	i_b_alu_shamt = 5’b0010 i_b_alu_operand_2 = 32’h0000 0002 i_b_alu_ctrl = 4’b0010	o_b_alu_result =32’h0000 0008 o_b_alu_branch =0	PASS
9	Swift right logical	i_b_alu_shamt = 5’b0010 i_b_alu_operand_2 = 32’h0000 0200 i_b_alu_ctrl = 4’b0011	o_b_alu_result =32’h0000 0080 o_b_alu_branch =0	PASS
10	load upper immediate	i_b_alu_operand_2 = 32’h0000 FFFF i_b_alu_ctrl = 4’b1111	o_b_alu_result =32’hFFFF 0000 o_b_alu_branch =0	PASS

Table 4.1.2.7: ALU Test Plan

4.1.2.8 Testbench and Simulation results

4.1.2.8.1 Testbench

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Chow Bin Lin
//
// Create Date: 02.09.2025 16:35:18
// File Name: tb_b_alu.sv
// Module Name: tb_b_alu
// Project Name: MIPS ISA Pipeline processor
// Code Type: Behavioural
// Description: Testbench for ALU block
//
//////////////////////////////////////////////////////////////////////////////////


module tb_b_alu();

    parameter CC = 10;
    logic clk;
    logic[3:0] test_case;
    logic[3:0] i;

    logic[31:0] tb_i_b_alu_operand_1, tb_i_b_alu_operand_2;
    logic[4:0] tb_i_b_alu_shamt;
    logic[3:0] tb_i_b_alu_ctrl;
    logic tb_o_b_alu_branch;
    logic[31:0] tb_o_b_alu_result;

    b_alu alu(
    .i_b_alu_operand_1(tb_i_b_alu_operand_1),
    .i_b_alu_operand_2(tb_i_b_alu_operand_2),
    .i_b_alu_shamt(tb_i_b_alu_shamt),
    .i_b_alu_ctrl(tb_i_b_alu_ctrl),
    .o_b_alu_branch(tb_o_b_alu_branch),
    .o_b_alu_result(tb_o_b_alu_result));

    always #(CC/2) clk = ~clk;

    //to chooose the input
    initial begin
        clk = 1;

        for( i = 1; i <11 ; i++)@(posedge clk)
        test_case = i;
    end

    always begin
    case(test_case)
    //----------------------------------------
    // Test Case 1: Add
    //----------------------------------------
    1:begin
    tb_i_b_alu_operand_1 = 32'h0000_0001;
```

```
tb_i_b_alu_operand_2 = 32'h0000_0002;

tb_i_b_alu_shamt = 5'b0_0000;

tb_i_b_alu_ctrl = 4'b0000;

end
//----------------------------------------
// Test Case 2: Subtract
//----------------------------------------
2:begin
tb_i_b_alu_operand_1 = 32'h0000_0001;

tb_i_b_alu_operand_2 = 32'h0000_0002;

tb_i_b_alu_shamt = 5'b0_0000;

tb_i_b_alu_ctrl = 4'b0010;

end

//----------------------------------------
// Test Case 3: Bitwise AND
//----------------------------------------
3:begin
tb_i_b_alu_operand_1 = 32'h0000_0001;

tb_i_b_alu_operand_2 = 32'h0000_0002;

tb_i_b_alu_shamt = 5'b0_0000;

tb_i_b_alu_ctrl = 4'b0100;

end


//----------------------------------------
// Test Case 4: Bitwise OR
//----------------------------------------
4:begin
tb_i_b_alu_operand_1 = 32'h0000_0001;

tb_i_b_alu_operand_2 = 32'h0000_0002;

tb_i_b_alu_shamt = 5'b0_0000;

tb_i_b_alu_ctrl = 4'b0101;

end


//----------------------------------------
// Test Case 5: Bitwise XOR
//----------------------------------------
5:begin
tb_i_b_alu_operand_1 = 32'h0000_0001;
```

```
tb_i_b_alu_operand_2 = 32'h0000_0002;

tb_i_b_alu_shamt = 5'b0_0000;

tb_i_b_alu_ctrl = 4'b0110;

end


//----------------------------------------
// Test Case 6: Bitwise NOR
//----------------------------------------
6:begin
tb_i_b_alu_operand_1 = 32'h0000_0001;

tb_i_b_alu_operand_2 = 32'h0000_0002;

tb_i_b_alu_shamt = 5'b0_0000;

tb_i_b_alu_ctrl = 4'b0111;

end


//----------------------------------------
// Test Case 7: Set on less than
//----------------------------------------
7:begin
tb_i_b_alu_operand_1 = 32'h0000_0001;

tb_i_b_alu_operand_2 = 32'h0000_0002;

tb_i_b_alu_shamt = 5'b0_0000;

tb_i_b_alu_ctrl = 4'b1010;

end


//----------------------------------------
// Test Case 8: Swift left logical
//----------------------------------------
8:begin
tb_i_b_alu_operand_1 = 32'h0000_0000;

tb_i_b_alu_operand_2 = 32'h0000_0002;

tb_i_b_alu_shamt = 5'b0_0010;

tb_i_b_alu_ctrl = 4'b0001;

end


//----------------------------------------
// Test Case 9: Swift right logical
//----------
```

```
    9:begin
    tb_i_b_alu_operand_1 = 32'h0000_0000;

    tb_i_b_alu_operand_2 = 32'h0000_0200;

    tb_i_b_alu_shamt = 5'b0_0010;

    tb_i_b_alu_ctrl = 4'b0011;

    end


    //----------------------------------------
    // Test Case 10: load upper immediate
    //----------------------------------------
    10:begin
    tb_i_b_alu_operand_1 = 32'h0000_0000;

    tb_i_b_alu_operand_2 = 32'h0000_0020;

    tb_i_b_alu_shamt = 5'b0_0000;

    tb_i_b_alu_ctrl = 4'b1111;
    end
    default:;
    endcase
    end

endmodule
```

4.1.2.8.2 Simulation Results

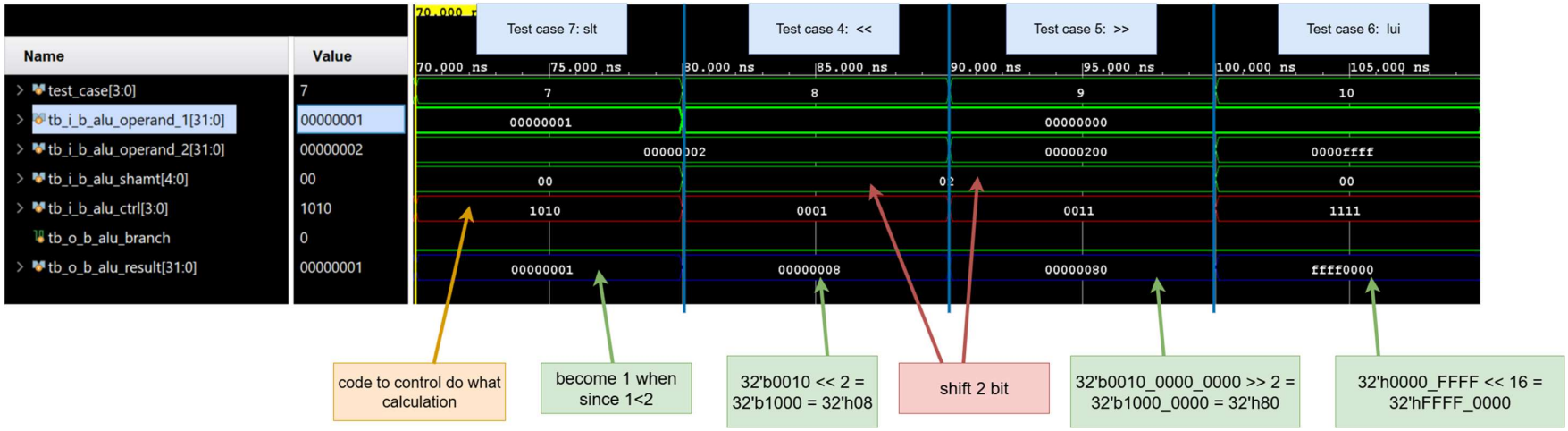


Diagram 4.1.2.8.2.1: ALU Simulation Results

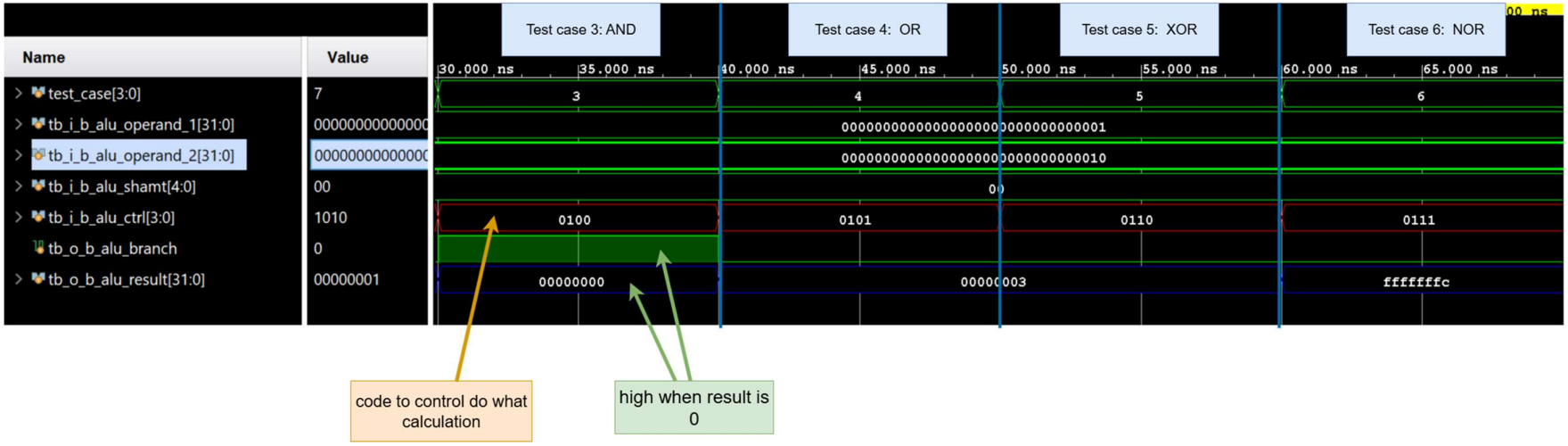


Diagram 4.1.2.8.2.2: ALU Simulation Results

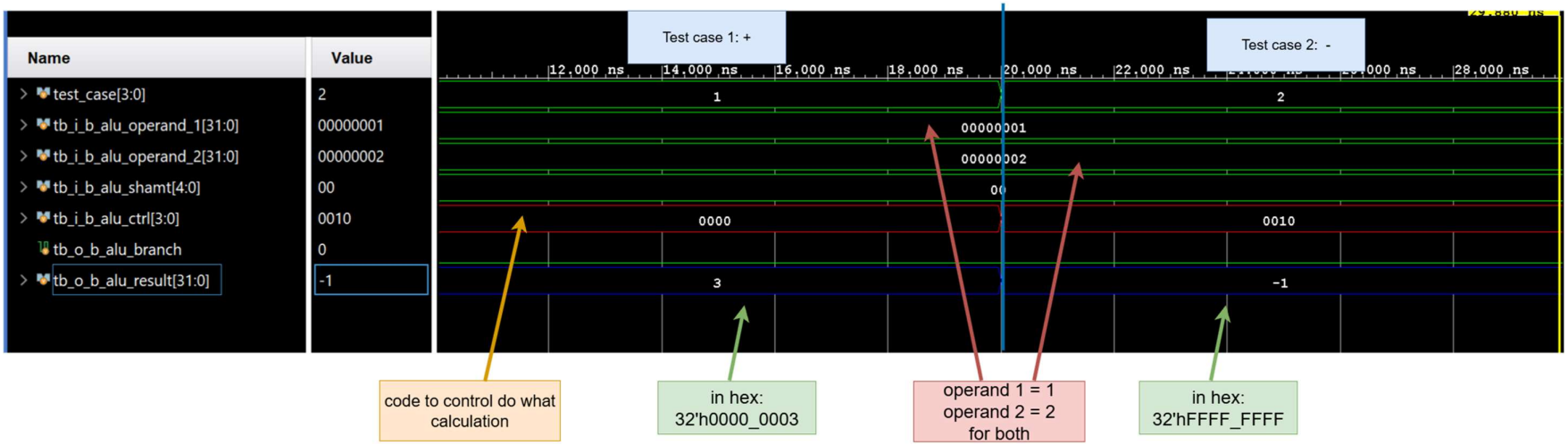


Diagram 4.1.2.8.2.3: ALU Simulation Result

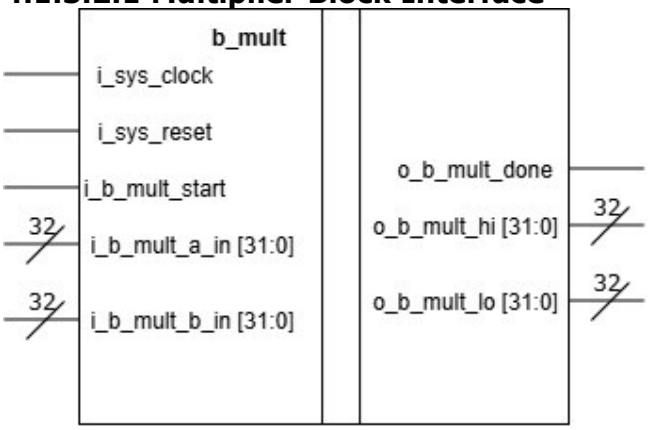
4.1.3 Sequential Multiplier

4.1.3.1 Functionality/Feature

- Performs a 32-bit × 32-bit unsigned multiplication.
- Implemented with sequential shift-and-add algorithm.
- HI stores the upper 32 bits of the product.
- LO stores the lower 32 bits of the product.
- Control signal during multiplication operation.

4.1.3.2 Block Interface and I/O Pin Description

4.1.3.2.1 Multiplier Block Interface



4.1.3.2.2 I/O Pin Description

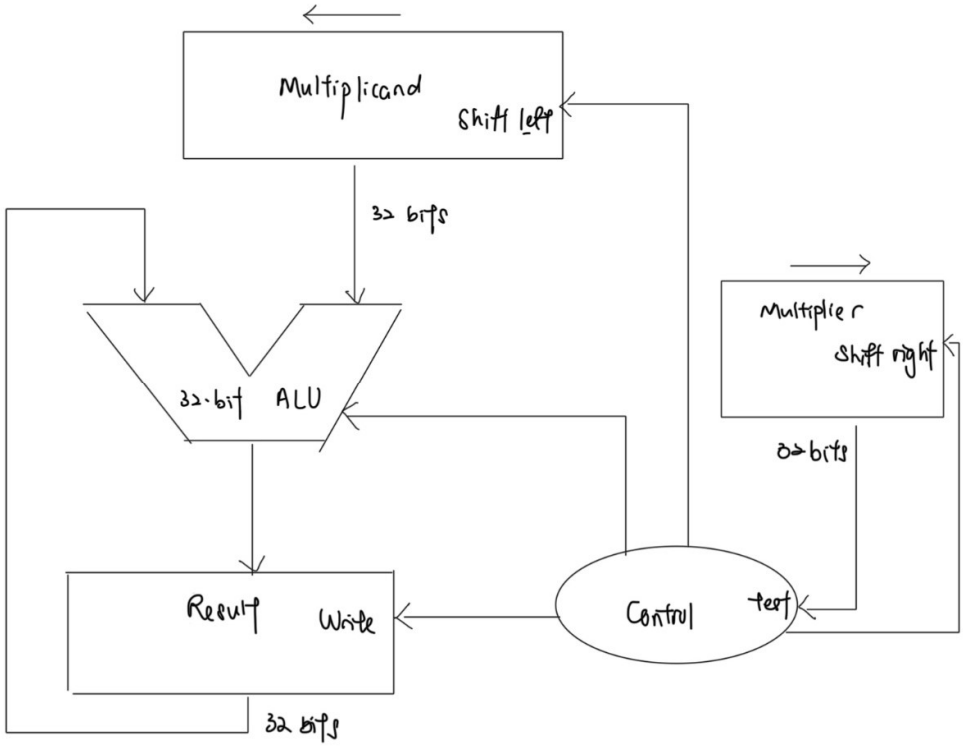
Pin name: **Pin class:** **Pin function:**	i_sys_clock control provide a clock signal to the system	**Source →Destination:**	Clock generator→Sequential Multiplier
Pin name: **Pin class:** **Pin function:**	i_sys_reset Control to reset the multiplier block	**Source →Destination:**	ON/Reset button→ Sequential Multiplier
Pin name: **Pin class:** **Pin function:**	i_b_mult_start control to trigger the multiplier to start	**Source →Destination:**	CPU→ Sequential Multiplier
Pin name: **Pin class:** **Pin function:**	i_b_mult_a_in [31:0] data operand a used to doing multiple calculation	**Source →Destination:**	CPU→ Sequential Multiplier
Pin name: **Pin class:** **Pin function:**	i_b_mult_b_in [31:0] data operand b used to doing multiple calculation	**Source →Destination:**	CPU→ Sequential Multiplier

Pin name: **Pin class:** **Pin function:**	o_b_mult_done control indicates that the multiplier has completed multiplication operation	**Source →Destination:**	Sequential Multiplier → Control Unit
Pin name: **Pin class:** **Pin function:**	o_b_mult_hi [31:0] data the result of the multiplier and store to high register	**Source →Destination:**	Sequential Multiplier → Register File
Pin name: **Pin class:** **Pin function:**	o_b_mult_lo [31:0] data the result of the multiplier and store to high register	**Source →Destination:**	Sequential Multiplier → Register File

4.1.3.3 Internal Operation

i_b_sys_reset	i_start	active(Current State)	i_done	Description
1	X	X	0	Reset
0	1	0	0	Start Operation
0	X	1	0	Busy Calculating
0	X	1	1	Operation Complete
0	0	0	0	IDLE

4.1.3.4 Pre-Synthesis Schematic

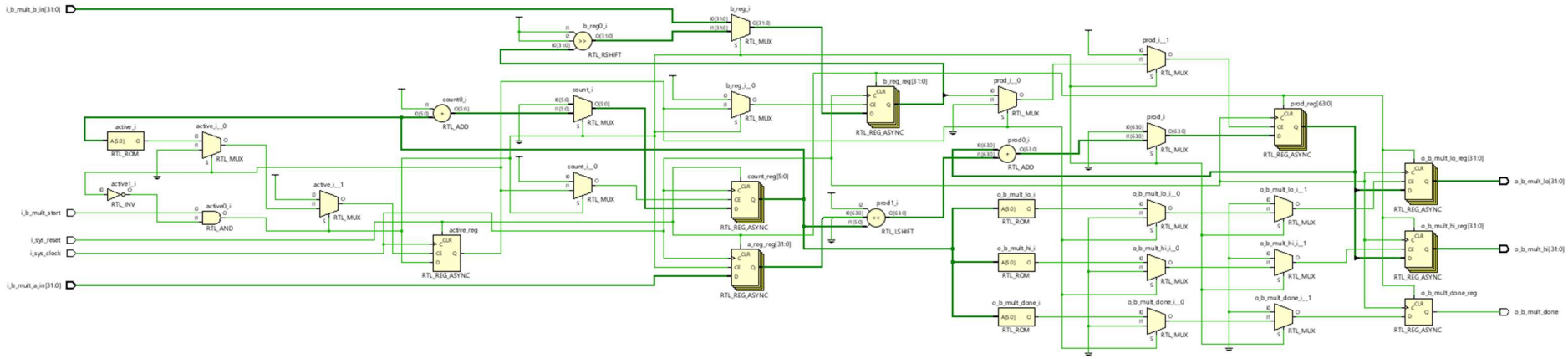


4.1.3.5 SV Model

```
`timescale 1s / 1ps

//////////////////////////////////////////////////////////////////////////////////
// Author: Goh Xuan Hui
//
// Create Date: 06.09.2025 01:22:23
// File Name: b_mult.sv
// Module Name: b_mult
// Project Name: MIPS ISA Pipeline processor
// Code Type: RTL level
// Description: Modeling of sequential multiplier block
//
//////////////////////////////////////////////////////////////////////////////////


 module b_mult(
  input  logic        i_sys_clock,
  input  logic        i_sys_reset,
  input  logic        i_b_mult_start,
  input  logic [31:0] i_b_mult_a_in,
  input  logic [31:0] i_b_mult_b_in,
  output logic        o_b_mult_done,
  output logic [31:0] o_b_mult_hi,
  output logic [31:0] o_b_mult_lo
);

  logic active;
  logic [5:0] count;   // 0-31 step
  logic [31:0] a_reg;
  logic [31:0] b_reg;
  logic [63:0] prod;

  // Sequential process
  always_ff @(posedge i_sys_clock or posedge i_sys_reset) begin
    if (i_sys_reset) begin
      active <= 1'b0;
      count  <= 6'd0;
      a_reg  <= 32'd0;
      b_reg  <= 32'd0;
      prod   <= 64'd0;
      o_b_mult_hi     <= 32'd0;
      o_b_mult_lo     <= 32'd0;
      o_b_mult_done   <= 1'b0;
    end else begin
      o_b_mult_done <= 1'b0; // default

      if (i_b_mult_start && !active) begin
        // Latch inputs and start
        active <= 1'b1;
        count  <= 6'd0;
        a_reg  <= i_b_mult_a_in;
        b_reg  <= i_b_mult_b_in;
        prod   <= 64'd0;
      end else if (active) begin

        if (b_reg[0] == 1'b1) begin
```

```
        prod <= prod + ({32'd0, a_reg} << count);
      end
      //
=============================================================

      // Shift left multiplicand, right multiplier
      b_reg <= b_reg >> 1;
      count <= count + 6'd1;

      // After 32 steps, finish
      if (count == 6'd32) begin
        active <= 1'b0;
        o_b_mult_done   <= 1'b1;
        o_b_mult_hi     <= prod[63:32];
        o_b_mult_lo     <= prod[31:0];
      end
    end
  end
end

endmodule
```

4.1.3.6 Post- Synthesis Schematic



4.1.3.7 Test Plan

No	Test Case	Description of Test Vector Generation	Expected Output	Status
1.	R-type: multu (Small numbers)	1.Set i_b_mult_a_in [31:0]= 32'd3 2.Set i_b_mult_b_in [31:0] =32'd5 3. Set i_start=1	o_done=1 o_b_mult_hi[31:0]=0 o_b_mult_lo[31:0]=32'd	PASS
2.	R-type: multu (Zero Multiply)	1.Set i_b_mult_a_in [31:0]= 0 2.Set i_b_mult_b_in [31:0] =32'd9999 3. Set i_start=1	o_done=1 o_b_mult_hi[31:0]=0 o_b_mult_lo[31:0]=0	PASS
3.	R-type: multu (Large numbers)	1.Set i_b_mult_a_in [31:0]= 12345 2.Set i_b_mult_b_in [31:0] =6789 3. Set i_start=1	o_done=1 o_b_mult_hi[31:0]= 0 o_b_mult_lo[31:0]= 32'd83810205	PASS
4.	R-type: multu (Multiply by one)	1.Set i_b_mult_a_in [31:0]= 32'd1 2.Set i_b_mult_b_in [31:0] =32'd8765 3. Set i_start=1	o_done=1 o_b_mult_hi[31:0]= 0 o_b_mult_lo[31:0]= 32'd8765	PASS
5.	R-type: multu (Overflow Check)	1.Set i_b_mult_a_in [31:0]= 32'hFFFFFFFF 2.Set i_b_mult_b_in [31:0] =32'd2 3. Set i_start=1	o_done=1 o_b_mult_hi[31:0]= 32'h00000001 o_b_mult_lo[31:0]= 32'hFFFFFFFE	PASS
6.	R-type: multu (Maximum multiply)	1.Set i_b_mult_a_in [31:0]= 32'hFFFFFFFF 2.Set i_b_mult_b_in [31:0] = 32'hFFFFFFFF 3. Set i_start=1	o_done=1 o_b_mult_hi[31:0]= 32'hFFFFFFFE o_b_mult_lo[31:0]= 32'h00000001	PASS

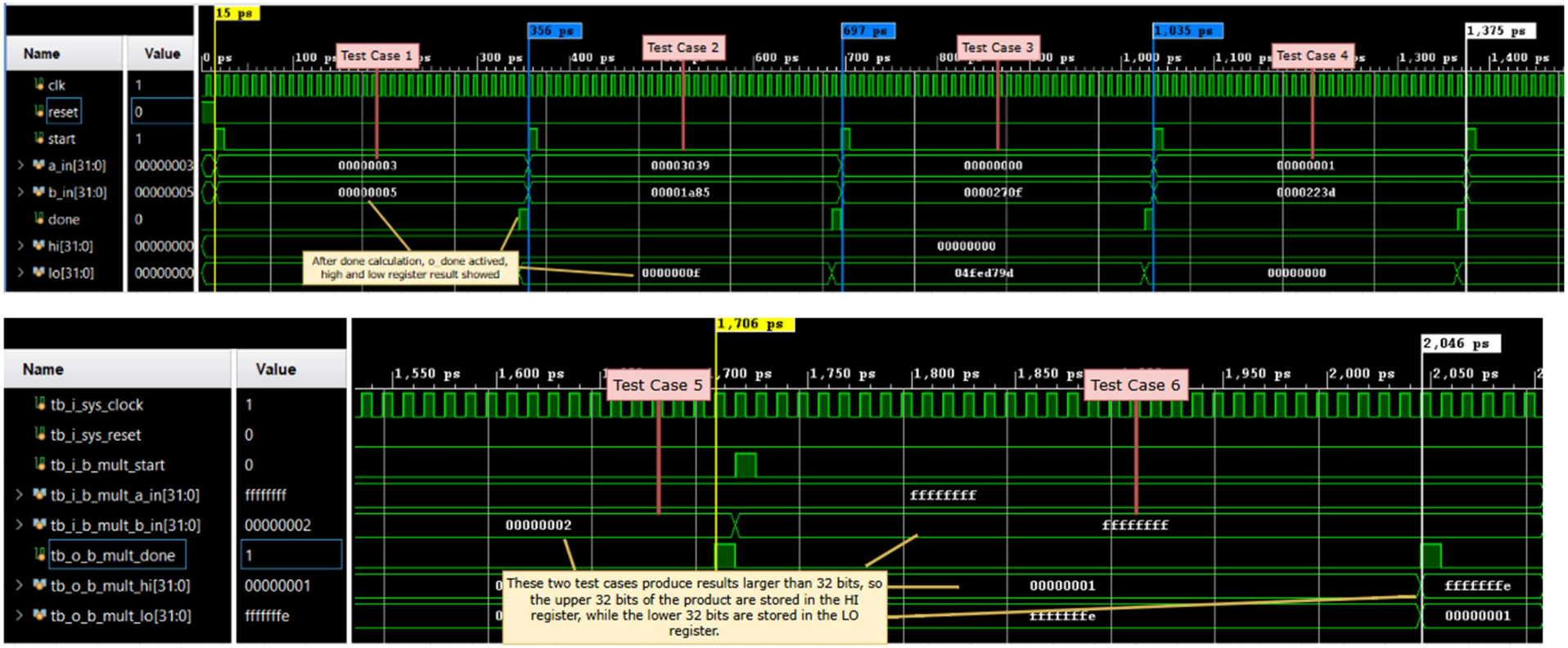
4.1.3.8 Testbench and Simulation Result

4.1.3.8.1 Testbench

```
//////////////////////////////////////////////////////////////////////////////////
// Author: Goh Xuan Hui
// Create Date: 06.09.2025 01:33:10
// File Name: tb_b_mult.sv
// Module Name: tb_b_mult
// Project Name: MIPS ISA Pipeline processor
// Code Type: Behavioural
// Description: Testbench for sequential multiplier block
//////////////////////////////////////////////////////////////////////////////////

module tb_b_mult;
logic tb_i_sys_clock;
 logic tb_i_sys_reset;
 logic tb_i_b_mult_start;
 logic [31:0] tb_i_b_mult_a_in;
 logic [31:0] tb_i_b_mult_b_in;
 logic tb_o_b_mult_done;
 logic [31:0] tb_o_b_mult_hi;
 logic [31:0] tb_o_b_mult_lo;
b_mult uut (
 .i_sys_clock(tb_i_sys_clock),
 .i_sys_reset(tb_i_sys_reset),
 .i_b_mult_start(tb_i_b_mult_start),
 .i_b_mult_a_in(tb_i_b_mult_a_in),
 .i_b_mult_b_in(tb_i_b_mult_b_in),
 .o_b_mult_done(tb_o_b_mult_done),
 .o_b_mult_hi(tb_o_b_mult_hi),
 .o_b_mult_lo(tb_o_b_mult_lo)
 );
// Clock generation
 initial tb_i_sys_clock = 0;
 always #5 tb_i_sys_clock = ~tb_i_sys_clock;
initial begin
 // Initialize & reset
 tb_i_sys_reset = 1;
 tb_i_b_mult_start = 0;
 tb_i_b_mult_a_in = 0;
 tb_i_b_mult_b_in = 0;
repeat(2) @(posedge tb_i_sys_clock); // Hold reset for 2 cycles
tb_i_sys_reset = 0;

// -------------------------
// Test 1: Small numbers (3 * 5 = 15)
tb_i_b_mult_a_in = 32'd3;
tb_i_b_mult_b_in = 32'd5;
tb_i_b_mult_start = 1;
@(posedge tb_i_sys_clock);
tb_i_b_mult_start = 0;
wait(tb_o_b_mult_done == 1);
@(posedge tb_i_sys_clock);

// -------------------------
// Test 2: Large numbers (12345 * 6789 = 83810205)
tb_i_b_mult_a_in = 32'd12345;
```

```
tb_i_b_mult_b_in = 32'd6789;
tb_i_b_mult_start = 1;
@(posedge tb_i_sys_clock);
tb_i_b_mult_start = 0;
wait(tb_o_b_mult_done == 1);
@(posedge tb_i_sys_clock);

// -------------------------
// Test 3: Zero multiply (0 * 9999 = 0)
tb_i_b_mult_a_in = 32'd0;
tb_i_b_mult_b_in = 32'd9999;
tb_i_b_mult_start = 1;
@(posedge tb_i_sys_clock);
tb_i_b_mult_start= 0;
wait(tb_o_b_mult_done== 1);
@(posedge tb_i_sys_clock);

// -------------------------
// Test 4: Multiply by one (1 * 8765 = 8765)
tb_i_b_mult_a_in = 32'd1;
tb_i_b_mult_b_in = 32'd8765;
tb_i_b_mult_start = 1;
@(posedge tb_i_sys_clock);
tb_i_b_mult_start = 0;
wait(tb_o_b_mult_done == 1);
@(posedge tb_i_sys_clock);

// -------------------------
// Test 5: Overflow check (0xFFFFFFFF * 2)
tb_i_b_mult_a_in = 32'hFFFFFFFF;
tb_i_b_mult_b_in = 32'd2;
tb_i_b_mult_start = 1;
@(posedge tb_i_sys_clock);
tb_i_b_mult_start = 0;
wait(tb_o_b_mult_done == 1);
@(posedge tb_i_sys_clock);

// -------------------------
// Test 6: Max * Max (0xFFFFFFFF * 0xFFFFFFFF)
tb_i_b_mult_a_in = 32'hFFFFFFFF;
tb_i_b_mult_b_in = 32'hFFFFFFFF;
tb_i_b_mult_start = 1;
@(posedge tb_i_sys_clock);
tb_i_b_mult_start = 0;
wait(tb_o_b_mult_done == 1);
@(posedge tb_i_sys_clock);

// -------------------------
repeat(5) @(posedge tb_i_sys_clock); // Extra cycles before finish
$finish;

end
endmodule
```

4.1.3.8.2 Simulation Result



4.2 Memory Unit

4.2.1 Instruction Memory Unit

4.2.1.1 Functionality/Feature

- Fetch the instructions to be decode from instruction memory

4.2.1.2 Instruction Memory Unit interface and I/O pin description

4.2.1.2.1 Instruction memory unit interface

Diagram 4.2.1.2.1 Block interface of instruction memory unit

4.2.1.2.2 I/O pin description

Pin name: **Pin class:** **Pin function:**	i_u_ins_mem_ins_addr[31:0] data receive address for instruction fetching	**Source → Destination:**	CPU Unit → Instruction Memory Unit
Pin name: **Pin class:** **Pin function:**	i_sys_clock control To provide a system clock to the block	**Source → Destination:**	Clock Generator → Instruction Memory Unit
Pin name: **Pin class:** **Pin function:**	i_u_ins_mem_wr_en control To provide signal to write instruction	**Source → Destination:**	CPU Unit → Instruction Memory Unit
Pin name: **Pin class:** **Pin function:**	i_u_ins_mem_wr_ins[31:0] data instruction to be write	**Source → Destination:**	CPU Unit → Instruction Memory Unit
Pin name: **Pin class:**	o_u_ins_mem_ins[31:0] data	**Source → Destination:**	Instruction Memory Unit → CPU Unit

Pin function:	to pass the instruction fetch for decoding		

4.2.1.3 Internal Operation: Function Table

Operation	Input/Condition	Output
Write instruction	i_u_ins_mem_wr_en = 1 i_u_ins_mem_ins_addr[31:0] = address of instruction to write i_u_ins_mem_wr_ins = instruction to be write	o_u_ins_mem_ins[31:0] = 0
Fetch instruction	i_u_ins_mem_wr_en = 0 i_u_ins_mem_ins_addr[31:0] = address of instruction to read	o_u_ins_mem_ins[31:0] = instruction fetched

4.2.1.4 Timing Requirement

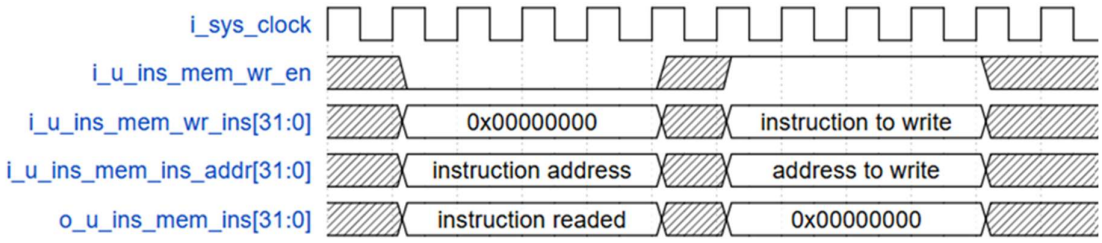


Diagram 4.2.4.1 Timing Requirement of Instruction Memory Unit

4.2.1.5 Pre-synthesis Schematic

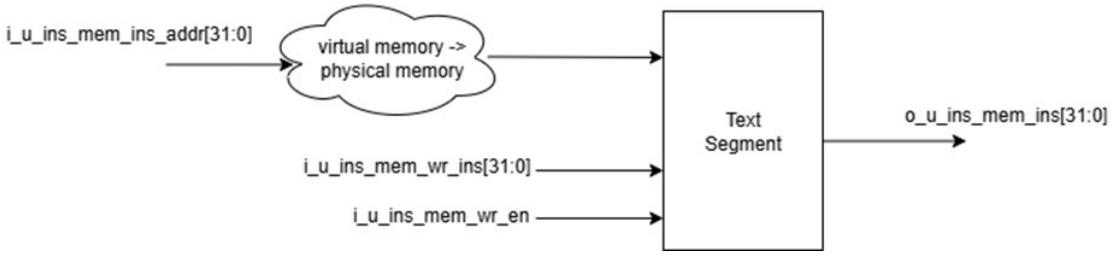


Diagram 4.2.1.5.1 Instruction Memory Unit Pre-synthesis schematic

4.2.1.6 System Verilog model

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Ng Jing Xuan
// Create Date: 04.09.2025 22:59:17
// File Name: b_ins_mem.sv
```

```
// Module Name: b_ins_mem
// Project Name: MIPS ISA Pipeline processor
// Code Type: RTL level
// Description: Modeling of instruction memory block
//
//////////////////////////////////////////////////////////////////////////////////
module u_ins_mem(
input logic i_sys_clock,
input logic [31:0] i_u_ins_mem_ins_addr, //instruction address
input logic [31:0] i_u_ins_mem_wr_ins, //data to write
input logic i_u_ins_mem_wr_en, // write enable
output logic [31:0] o_u_ins_mem_ins);

logic [31:0] instructions [4095:0];
logic [16:0] physical_mem_addr;
logic [11:0] physical_mem_loc;


initial begin  //initialize alll instruction memory to 0
   for (int i=0;i<4096;i++)
      instructions [i] = 32'b0;
end

assign physical_mem_addr =
{i_u_ins_mem_ins_addr[29:28],i_u_ins_mem_ins_addr[15:14],i_u_ins_mem_ins_addr[
11:0]};
assign physical_mem_loc = physical_mem_addr[13:2];
assign o_u_ins_mem_ins = i_u_ins_mem_wr_en ? 32'b0 : instructions
[physical_mem_loc];

always_ff @(posedge i_sys_clock)begin

   if(i_u_ins_mem_wr_en)
   instructions[physical_mem_loc] <= i_u_ins_mem_wr_ins;
end
endmodule
```

4.2.1.7 Test plan

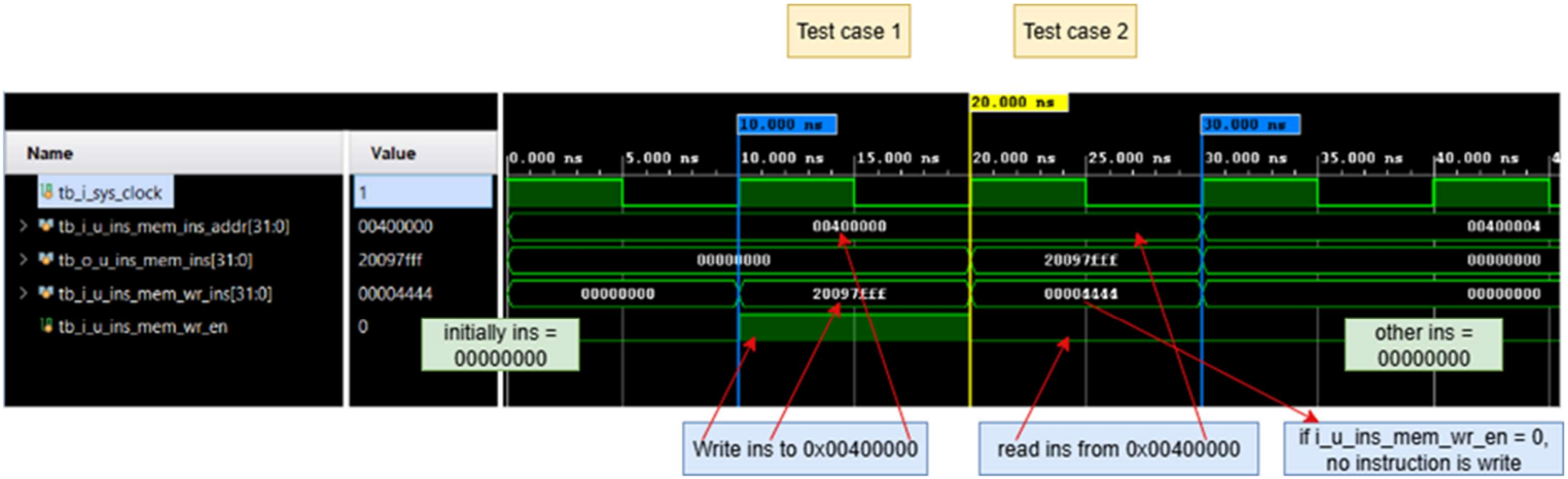
No.	Test Case	Description of Test Vector Generation	Expected Output	Status
1	Write instruction	1. Set i_u_ins_mem_wr_en = 1 2. Set i_u_ins_mem_ins_addr[31:0] = 0x00400000 3. Set i_u_ins_mem_wr_ins = 0x20097FFF	1. o_u_ins_mem_ins[31:0] = 0x00000000	PASS
2	Fetch instruction (continue from test case 1)	1. Set i_u_ins_mem_wr_en = 0 2. Set i_u_ins_mem_ins_addr[31:0] = 0x00400000	1. o_u_ins_mem_ins[31:0] = 0x20097FFF	PASS

4.2.1.8 Testbench

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Ng Jing Xuan
//
// Create Date: 08.09.2025 23:18:48
// File Name: tb_u_ins_mem.sv
// Module Name: tb_u_ins_mem
// Project Name: MIPS ISA Pipeline processor
// Code Type: Behavioural
// Description: Testbench for Instruction Memory Unit
//
//////////////////////////////////////////////////////////////////////////////////
module tb_i_u_ins_mem();
parameter CC = 10;
logic tb_i_sys_clock;
logic [31:0] tb_i_u_ins_mem_ins_addr;
logic [31:0] tb_o_u_ins_mem_ins;
logic [31:0] tb_i_u_ins_mem_wr_ins;
logic tb_i_u_ins_mem_wr_en;

u_ins_mem dut_u_ins_mem
(.i_sys_clock(tb_i_sys_clock),
.i_u_ins_mem_ins_addr(tb_i_u_ins_mem_ins_addr),
.i_u_ins_mem_wr_ins(tb_i_u_ins_mem_wr_ins),
.i_u_ins_mem_wr_en(tb_i_u_ins_mem_wr_en),
.o_u_ins_mem_ins(tb_o_u_ins_mem_ins));

initial begin
tb_i_sys_clock = 1'b1;
forever #(CC/2) tb_i_sys_clock = ~tb_i_sys_clock ;
end

initial begin
tb_i_u_ins_mem_wr_en = 1'b0;  //read ins[0]
tb_i_u_ins_mem_ins_addr = 32'h00400000;
tb_i_u_ins_mem_wr_ins = 32'h0;

repeat (1) @(posedge tb_i_sys_clock);
tb_i_u_ins_mem_wr_en = 1'b1;  //write ins[0]
tb_i_u_ins_mem_ins_addr = 32'h00400000;
tb_i_u_ins_mem_wr_ins = 32'h20097fff;

repeat (1) @(posedge tb_i_sys_clock);
tb_i_u_ins_mem_wr_en = 1'b0;  //read ins[0] again
tb_i_u_ins_mem_ins_addr = 32'h00400000;
tb_i_u_ins_mem_wr_ins = 32'h00004444;  //ins will not write if wr_en is 0

repeat (1) @(posedge tb_i_sys_clock);
tb_i_u_ins_mem_wr_en = 1'b0;  //read ins[1]
```

```
tb_i_u_ins_mem_ins_addr = 32'h00400004;
tb_i_u_ins_mem_wr_ins = 32'h0;
end
endmodule
```

4.2.1.9 Simulation Result



4.2.2 Data Memory Unit

4.2.2.1 Functionality/Feature

- Read data from data memory
- Write data to data memory

4.2.2.2 Data Memory Unit interface and I/O pin description

4.2.2.2.1 Data memory unit interface

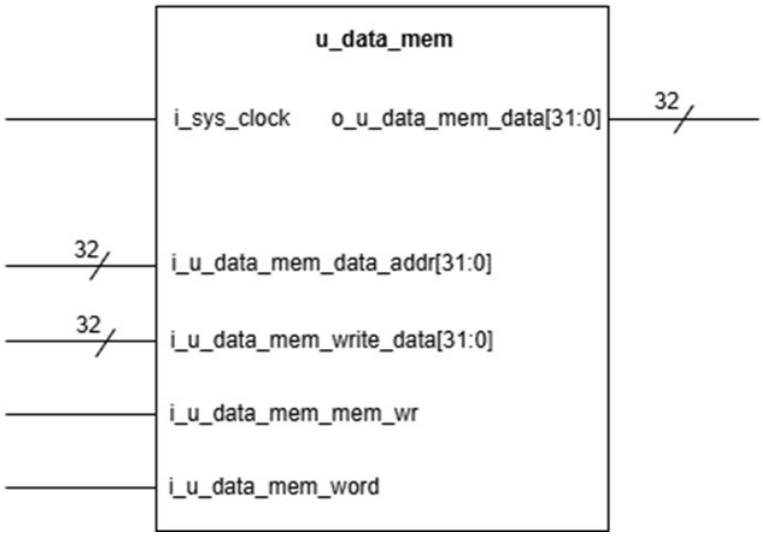


Diagram 4.2.2.2.1 Block interface of data memory unit

4.2.2.2.2 I/O pin description

Pin name: **Pin class:** **Pin function:**	i_u_data_mem_data_addr[31:0] data receive address of data to read or write	**Source → Destination:**	CPU Unit → Data Memory Unit
Pin name: **Pin class:** **Pin function:**	i_u_data_mem_write_data[31:0] data receive data to be written into data memory	**Source → Destination:**	CPU Unit → Data Memory Unit
Pin name: **Pin class:** **Pin function:**	i_u_data_mem_mem_wr control To enable write data to data memory	**Source → Destination:**	CPU Unit → Data Memory Unit
Pin name: **Pin class:** **Pin function:**	i_u_data_mem_word control	**Source → Destination:**	CPU Unit → Data Memory Unit

	to determine whether to load/store data in word or byte		
Pin name: **Pin class:** **Pin function:**	i_sys_clock control To provide a system clock to the block	**Source → Destination:**	Clock Generator → Data Memory Unit
Pin name: **Pin class:** **Pin function:**	o_u_data_mem_data[31:0] data to pass the data read from data memory	**Source → Destination:**	Data Memory Unit → CPU Unit

4.2.2.3 Internal Operation: Function Table

Operation	Input/Condition	Output
Read data from memory (load word)	i_u_data_mem_data_addr[31:0] = address to read i_u_data_mem_word = 1	o_u_data_mem_data[31:0] = data read in word
Read data from memory (load byte)	i_u_data_mem_data_addr[31:0] = address to read i_u_data_mem_word = 0	o_u_data_mem_data[31:0] = data read in byte
Write data to memory (store word)	i_u_data_mem_word = 1 i_u_data_mem_mem_wr = 1 i_u_data_mem_write_data[31:0] = data to write i_u_data_mem_data_addr[31:0] = address to write	o_u_data_mem_data[31:0] = 0
Write data to memory (store byte)	i_u_data_mem_word = 0 i_u_data_mem_mem_wr = 1 i_u_data_mem_write_data[31:0] = data to write i_u_data_mem_data_addr[31:0] = address to write	o_u_data_mem_data[31:0] = 0

4.2.2.4 Timing Requirement

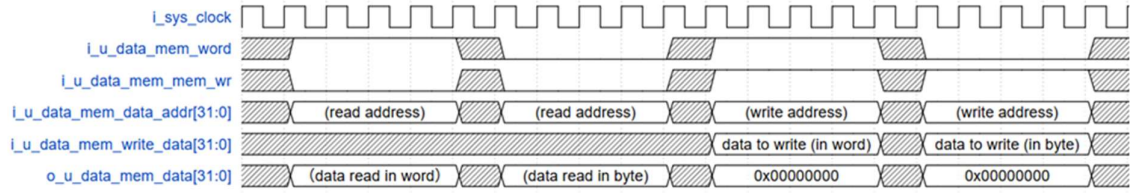


Diagram 4.2.2.4.1 Timing Requirement of Data Memory Unit

4.2.2.5 Pre-synthesis Schematic

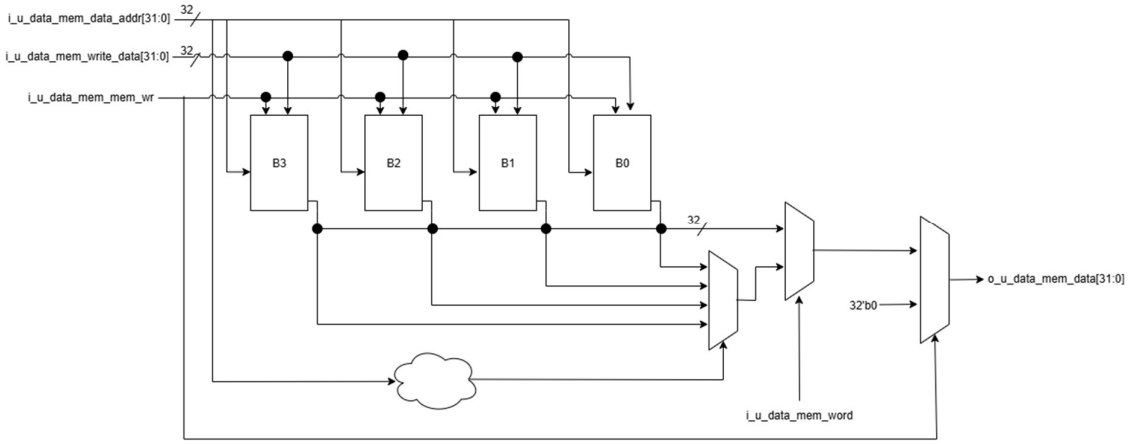


Diagram 4.2.2.5.1 Data Memory Unit Pre-synthesis schematic

4.2.2.6 System Verilog model

```
  `timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Ng Jing Xuan
// Create Date: 04.09.2025 22:59:17
// File Name: b_data_mem.sv
// Module Name: b_data_mem
// Project Name: MIPS ISA Pipeline processor
// Code Type: RTL level
// Description: Modeling of data memory block
//
//////////////////////////////////////////////////////////////////////////////////
module u_data_mem(
input logic i_sys_clock,
input logic [31:0] i_u_data_mem_data_addr,
input logic [31:0] i_u_data_mem_write_data,
input logic i_u_data_mem_mem_wr,
input logic i_u_data_mem_word,
output logic [31:0] o_u_data_mem_data
);

logic [31:0] data [2047:0];
logic [15:0] physical_data_addr;
logic [10:0] physical_data_loc;
```

```
logic [1:0] byte_no;
logic [31:0] byte_read;


initial begin  //initialize alll data memory to 0
   for (int i=0;i<2048;i++)
      data [i] = 32'b0;
end

assign physical_data_addr =
{i_u_data_mem_data_addr[29:28],i_u_data_mem_data_addr[15:14],i_u_data_mem_d
ata_addr[11:0]};
assign physical_data_loc = physical_data_addr[12:2];
assign byte_no = i_u_data_mem_data_addr[1:0];
assign byte_read = byte_no[1] ? (byte_no[0]? {24'b0,data
[physical_data_loc][31:24]}: {24'b0,data [physical_data_loc][23:16]}) : (byte_no[0]?
{24'b0,data [physical_data_loc][15:8]}: {24'b0,data [physical_data_loc][7:0]}) ;
assign o_u_data_mem_data = i_u_data_mem_mem_wr ? 32'b0 :
(i_u_data_mem_word ? data [physical_data_loc] : byte_read) ;


always_ff @(posedge i_sys_clock)begin
   if (i_u_data_mem_mem_wr)begin
      if (i_u_data_mem_word) //write word
         data [physical_data_loc] <= i_u_data_mem_write_data;

      else begin //write byte
         case (byte_no)
         2'b00: data [physical_data_loc][7:0] <= i_u_data_mem_write_data[7:0];
         2'b01: data [physical_data_loc][15:8] <= i_u_data_mem_write_data[7:0];
         2'b10: data [physical_data_loc][23:16] <= i_u_data_mem_write_data[7:0];
         2'b11: data [physical_data_loc][31:24] <= i_u_data_mem_write_data[7:0];
         default: data [physical_data_loc] <= data [physical_data_loc];
         endcase
      end
   end
end
endmodule
```

4.2.2.7 Test plan

No.	Test Case	Description of Test Vector Generation	Expected Output	Status
1	Write data (store word)	2. Set i_u_data_mem_word = 1 3. Set i_u_data_mem_mem_wr = 1 4. Set i_u_data_mem_data_addr[31:0] = 0x10000000 5. Set i_u_data_mem_write_data[31:0] = 0x87654321	1. o_u_data_mem_data[31:0] = 0	PASS
2	Read data in word	2. Set i_u_data_mem_word = 1 3. Set i_u_data_mem_mem_wr = 0	1 o_u_data_mem	PASS

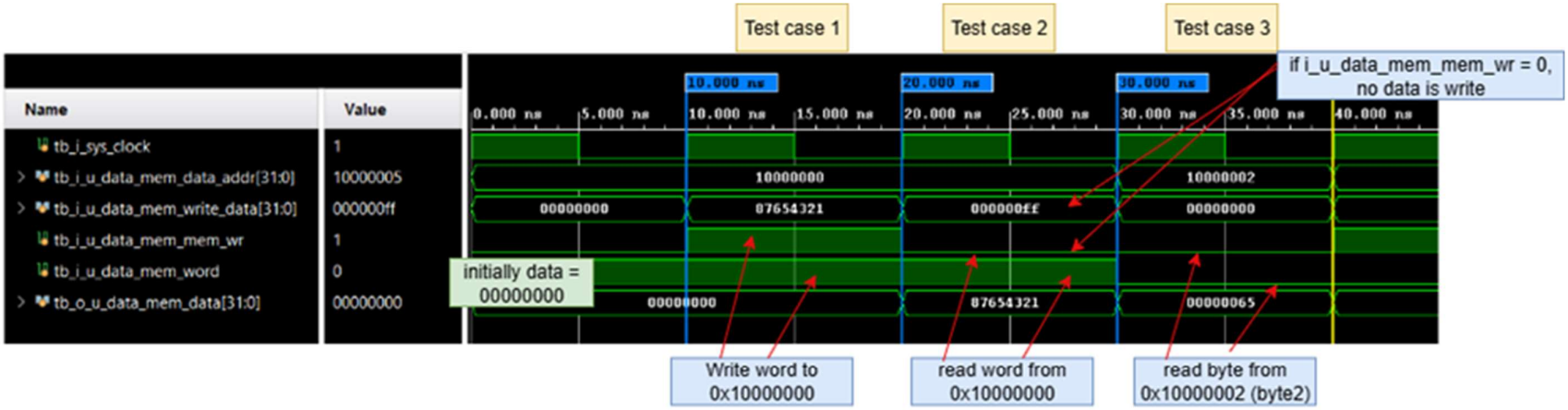
	(continue from test case 1)	4. Set i_u_data_mem_data_addr[31:0] = 0x10000000	_data[31:0] = 0x87654321	
3	Read data in byte (continue from test case 2)	2. Set i_u_data_mem_word = 0 3. Set i_u_data_mem_mem_wr = 0 4. Set i_u_data_mem_data_addr[31:0] = 0x10000002	1 o_u_data_mem_data[31:0] = 0x00000065	PASS
4	Write data (store byte)	2. Set i_u_data_mem_word = 0 3. Set i_u_data_mem_mem_wr = 1 4. Set i_u_data_mem_data_addr[31:0] = 0x10000005 5. Set i_u_data_mem_write_data[31:0] = 0x000000FF	1. o_u_data_mem_data[31:0] = 0	PASS

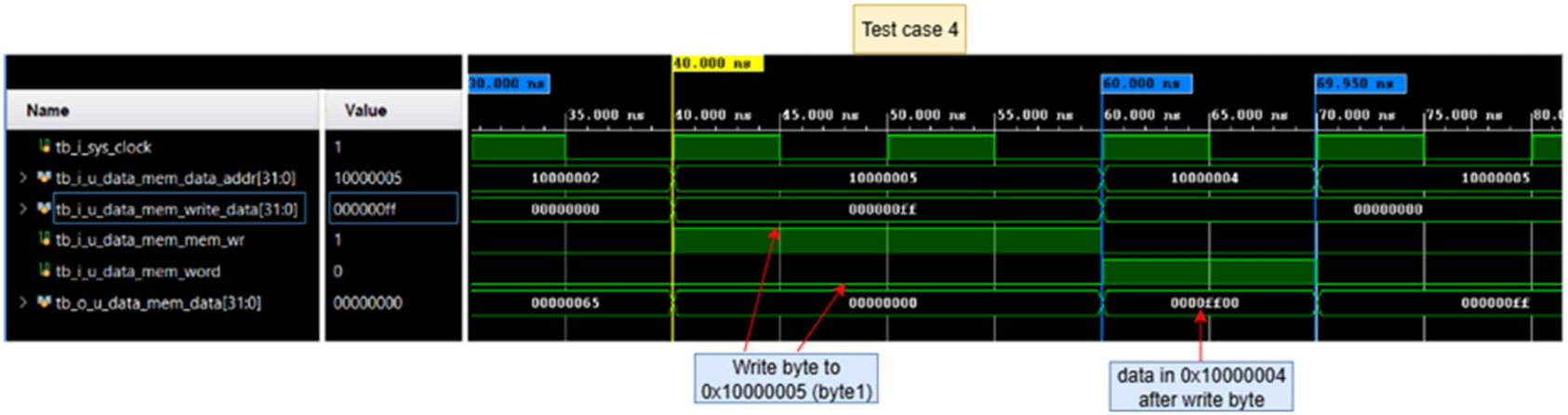
4.2.2.8 Testbench

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Ng Jing Xuan
//
// Create Date: 08.09.2025 23:18:48
// File Name: tb_u_data_mem.sv
// Module Name: tb_u_data_mem
// Project Name: MIPS ISA Pipeline processor
// Code Type: Behavioural
// Description: Testbench for Data Memory Unit
//
//////////////////////////////////////////////////////////////////////////////////
module tb_u_data_mem();
parameter CC = 10;
logic tb_i_sys_clock;
logic [31:0] tb_i_u_data_mem_data_addr;
logic [31:0] tb_i_u_data_mem_write_data;
logic tb_i_u_data_mem_mem_wr;
logic tb_i_u_data_mem_word;
logic [31:0] tb_o_u_data_mem_data;


u_data_mem dut_u_data_mem
(.i_sys_clock(tb_i_sys_clock),
.i_u_data_mem_data_addr(tb_i_u_data_mem_data_addr),
```

```
.i_u_data_mem_write_data(tb_i_u_data_mem_write_data),
.i_u_data_mem_mem_wr(tb_i_u_data_mem_mem_wr),
.i_u_data_mem_word(tb_i_u_data_mem_word),
.o_u_data_mem_data(tb_o_u_data_mem_data));


initial begin
tb_i_sys_clock = 1'b1;
forever #(CC/2) tb_i_sys_clock = ~tb_i_sys_clock ;
end

initial begin
tb_i_u_data_mem_mem_wr = 1'b0; //read word data [0] initially =0
tb_i_u_data_mem_word = 1'b1;
tb_i_u_data_mem_data_addr = 32'h10000000;
tb_i_u_data_mem_write_data = 32'h00000000;

repeat (1) @(posedge tb_i_sys_clock);
tb_i_u_data_mem_mem_wr = 1'b1; //write word data [0]
tb_i_u_data_mem_word = 1'b1;
tb_i_u_data_mem_data_addr = 32'h10000000;
tb_i_u_data_mem_write_data = 32'h87654321;

repeat (2) @(posedge tb_i_sys_clock);
tb_i_u_data_mem_mem_wr = 1'b0; //read word data [0]
tb_i_u_data_mem_word = 1'b1;
tb_i_u_data_mem_data_addr = 32'h10000000;
tb_i_u_data_mem_write_data = 32'h000000FF;

repeat (1) @(posedge tb_i_sys_clock);
tb_i_u_data_mem_mem_wr = 1'b0; //read data [0] byte 2
tb_i_u_data_mem_word = 1'b0;
tb_i_u_data_mem_data_addr = 32'h10000002;
tb_i_u_data_mem_write_data = 32'h0;


repeat (1) @(posedge tb_i_sys_clock);
tb_i_u_data_mem_mem_wr = 1'b1; //write data [1] byte 0
tb_i_u_data_mem_word = 1'b0;
tb_i_u_data_mem_data_addr = 32'h10000005;
tb_i_u_data_mem_write_data = 32'h000000FF;

repeat (2) @(posedge tb_i_sys_clock);
tb_i_u_data_mem_mem_wr = 1'b0;  //read data [1]
tb_i_u_data_mem_word = 1'b1;
tb_i_u_data_mem_data_addr = 32'h10000004;
tb_i_u_data_mem_write_data = 32'h0;
end
endmodule
```

4.2.2.9 Simulation Result





4.3 Control Unit

4.3.0 Control Unit

4.3.0.1 Functionality/Feature

- Decode opcode and function from instructions to several top-level control signals or opcode for the pipeline.

4.3.0.2 Block interface and I/O pin description

4.3.0.2.1 Main control block interface

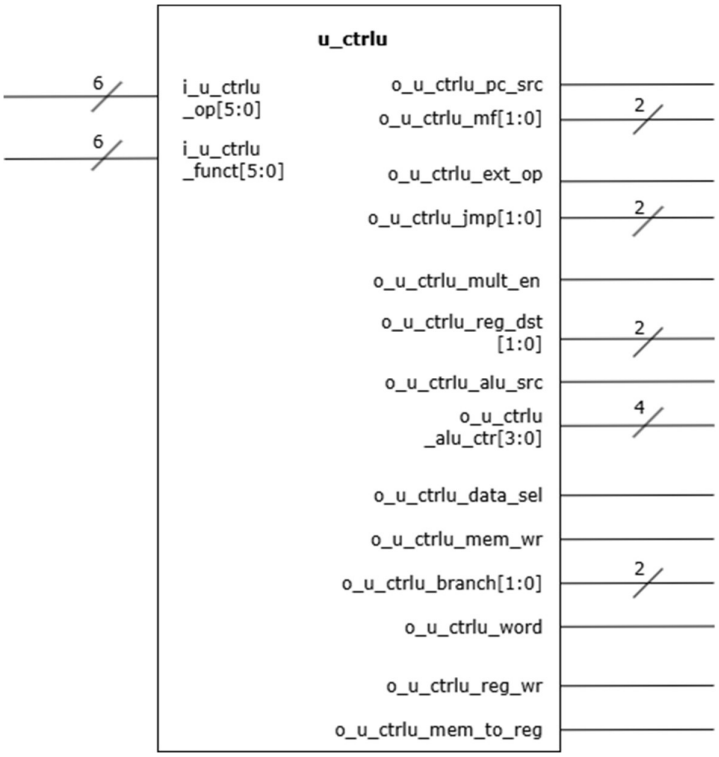


Diagram 4.3.0.2.1: Block interface of control unit

4.3.0.2.2 I/O pin description

Pin name: **Pin class:** **Pin function:**	i_u_ctrlu_op[5:0] control To receive instruction opcode for instruction decoding	**Source →Destination:**	CPU Unit → Control Unit
Pin name: **Pin class:** **Pin function:**	i_u_ctrlu_funct[5:0] control To receive R-type function field for detailed ALU decode	**Source →Destination:**	CPU Unit → Control Unit
Pin name: **Pin class:** **Pin function:**	o_u_ctrlu_pc_src control To select source of program counter	**Source →Destination:**	Control Unit → CPU Unit
Pin name: **Pin class:** **Pin function:**	o_u_ctrl_mf[1:0] control To select hi and lo register to receive data	**Source →Destination:**	Control Unit → CPU Unit

Pin name: **Pin class:** **Pin function:**	o_u_ctrlu_ext_op control To select zero or sign extend	**Source** **→Destination:**	Control Unit → CPU Unit
Pin name: **Pin class:** **Pin function:**	o_u_ctrlu_jmp [1:0] control to indicate jump instructions	**Source** **→Destination:**	Control Unit → CPU Unit
Pin name: **Pin class:** **Pin function:**	o_u_ctrlu_mult_en control To enable multiplier for multiplication process	**Source** **→Destination:**	Control Unit → CPU Unit
Pin name: **Pin class:** **Pin function:**	o_u_ctrlu_reg_dst [1:0] control To select destination register ($rt/ $rd/$ra)	**Source** **→Destination:**	Control Unit → CPU Unit
Pin name: **Pin class:** **Pin function:**	o_u_ctrlu_alu_src control To select ALU input source (register or immediate)	**Source** **→Destination:**	Control Unit → CPU Unit
Pin name: **Pin class:** **Pin function:**	o_u_ctrlu_alu_ctr[3:0] control To select specific ALU/shifter operation	**Source** **→Destination:**	Control Unit → CPU Unit
Pin name: **Pin class:** **Pin function:**	o_u_ctrlu_data_sel control To select pc+4	**Source** **→Destination:**	Control Unit → Memory Unit
Pin name: **Pin class:** **Pin function:**	o_u_ctrlu_mem_wr control To enable memory write	**Source** **→Destination:**	Control Unit → Memory Unit
Pin name: **Pin class:** **Pin function:**	o_u_ctrlu_branch[1:0] control To indicate branch instruction	**Source** **→Destination:**	Control Unit → CPU Unit
Pin name: **Pin class:** **Pin function:**	o_u_ctrlu_word control To differentiate word and byte	**Source** **→Destination:**	Control Unit → CPU Unit
Pin name: **Pin class:** **Pin function:**	o_u_ctrlu_reg_wr control To enable register file write	**Source** **→Destination:**	Control Unit → CPU Unit
Pin name: **Pin class:** **Pin function:**	o_u_ctrlu_mem_to_reg control To select value to write back to register (ALU/memory)	**Source** **→Destination:**	Control Unit → CPU Unit

Table 4.3.0.2.2: I/O pin description of Control Unit

4.3.0.3 Internal Operation: Function Table

Inputs			Outputs														
Instr			**IF**		**ID**		**EX**					**MEM**				**WB**	
	i_u_ctrlu_op[5:0]	**i_u_ctrlu_funct[5:0]**	**o_u_ctrlu_pc_src**	**o_u_mctrlu_mf[1:0]**	**o_u_ctrlu_ext_op**	**o_u_ctrlu_jmp[1:0]**	**o_u_ctrlu_mult_en**	**o_u_ctrlu_reg_dst [1:0]**	**o_u_ctrlu_alu_src**	**Required ALU action**	**o_b_mctrl_alu_op [3:0]**	**o_u_ctrlu_data_sel**	**o_u_ctrlu_mem_wr**	**o_u_ctrlu_branch [1:0]**	**o_u_ctrlu_word**	**o_u_ctrlu_reg_wr**	**o_u_ctrlu_mem_to_reg**
add	000000	100000	0	00	x	00	0	01($rd)	0	Add	0000	0	0	00	x	1	0
sub	000000	100010	0	00	x	00	0	01	0	Subtract	0010	0	0	00	x	1	0
and	000000	100100	0	00	x	00	0	01	0	Bitwise AND	0100	0	0	00	x	1	0
or	000000	100101	0	00	x	00	0	01	0	Bitwise OR	0101	0	0	00	x	1	0
xor	000000	100110	0	00	x	00	0	01	0	Bitwise XOR	0110	0	0	00	x	1	0
nor	000000	100111	0	00	x	00	0	01	0	Bitwise NOR	0111	0	0	00	x	1	0
slt	000000	101010	0	00	x	00	0	01	0	Set on Less Than	1010	0	0	00	x	1	0
sll	000000	000000	0	00	x	00	0	01	0	Shift Left Logical	0001	0	0	00	x	1	0
srl	000000	000010	0	00	x	00	0	01	0	Shift Right Logical	0011	0	0	00	x	1	0
multu	000000	011001	0	00	x	00	1	01	0	x	xxxx	0	0	00	x	1	0
mfhi	000000	010000	0	11	x	00	0	01	0	Add	0000	0	0	00	x	1	0
mflo	000000	010010	0	01	x	00	0	01	0	Add	0000	0	0	00	x	1	0
jr	000000	001000	0	00	x	10	0	xx	0	x	xxxx	0	0	00	x	0	0
jalr	000000	001001	0	00	x	10	0	01	0	x	xxxx	1	0	00	x	1	0
lw	100011	xxxxxx	0	00	1	00	0	00($rt)	1	Add (address)	0000	0	0	00	1	1	1

sw	101011	xxxxxx	0	00	1	00	0	xx	1	Add (address)	0000	0	1	00	1	0	x
lb	100000	xxxxxx	0	00	1	00	0	00	1	Add (address)	0000	0	0	00	0	1	1
sb	101000	xxxxxx	0	00	1	00	0	xx	1	Add (address)	0000	0	1	00	0	0	x
addi	001000	xxxxxx	0	00	1	00	0	00	1	Add	0000	0	0	00	x	1	0
andi	001100	xxxxxx	0	00	0	00	0	00	1	Bitwise AND	0100	0	0	00	x	1	0
ori	001101	xxxxxx	0	00	0	00	0	00	1	Bitwise OR	0101	0	0	00	x	1	0
slti	001010	xxxxxx	0	00	1	00	0	00	1	Set on Less Than	1010	0	0	00	x	1	0
beq	000100	xxxxxx	1	00	x	00	0	xx	0	Sub	0010	0	0	11	x	0	x
bne	000101	xxxxxx	1	00	x	00	0	xx	0	Sub	0010	0	0	10	x	0	x
j	000010	xxxxxx	1	00	x	01	0	xx	x	x	xxxx	0	0	00	x	0	x
jal	000011	xxxxxx	1	00	x	01	0	10	x	x	xxxx	1	0	00	x	1	0
lui	001111	xxxxxx	0	00	x	00	0	00	1	Shift left 16 bit	1111	0	0	00	x	1	0

Table 4.3.0.3: Function table of control unit

4.3.0.4 Pre-synthesis Schematic Diagram

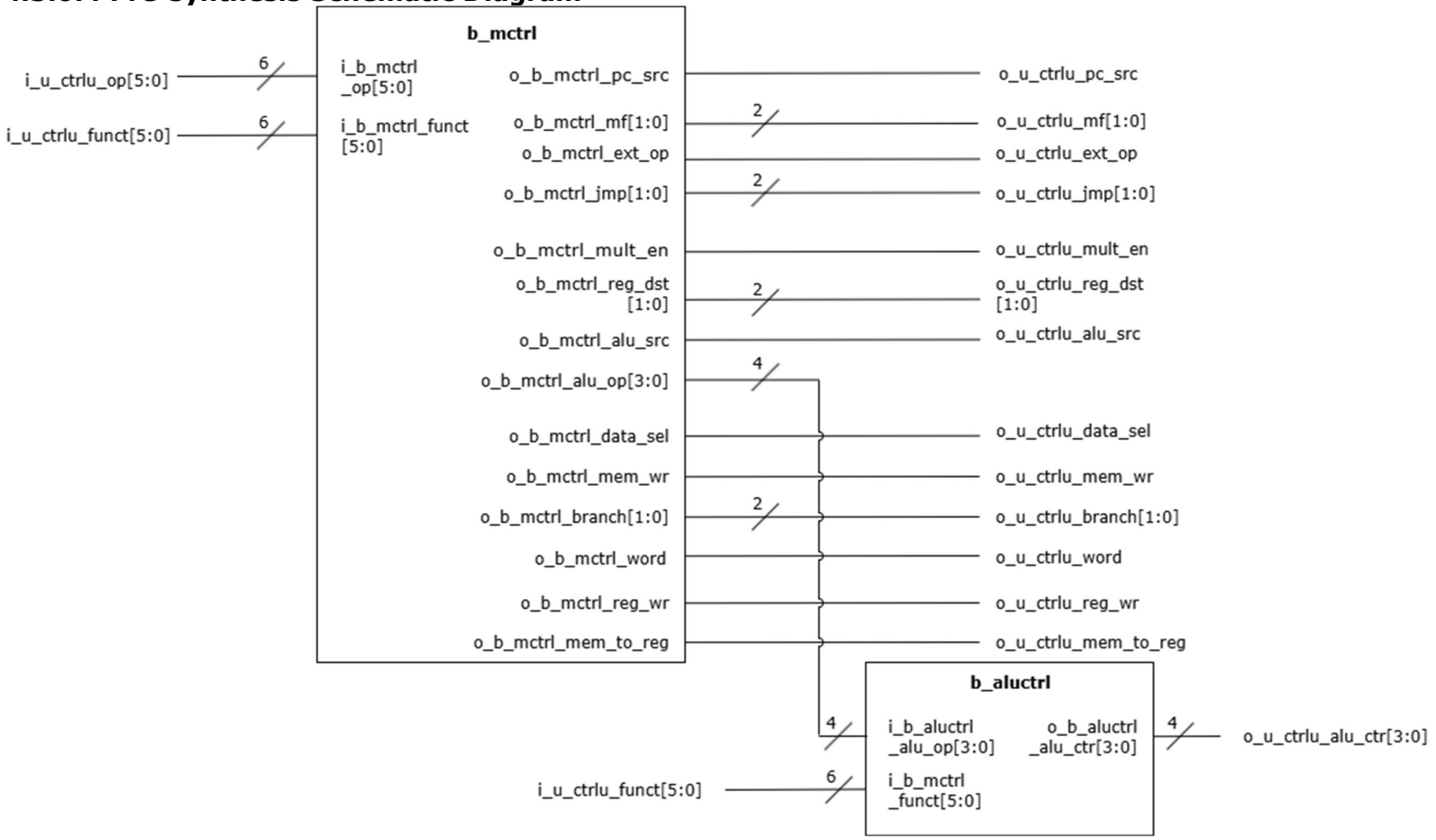


Diagram 4.3.0.2.1: pre-synthesis schematic diagram of control unit

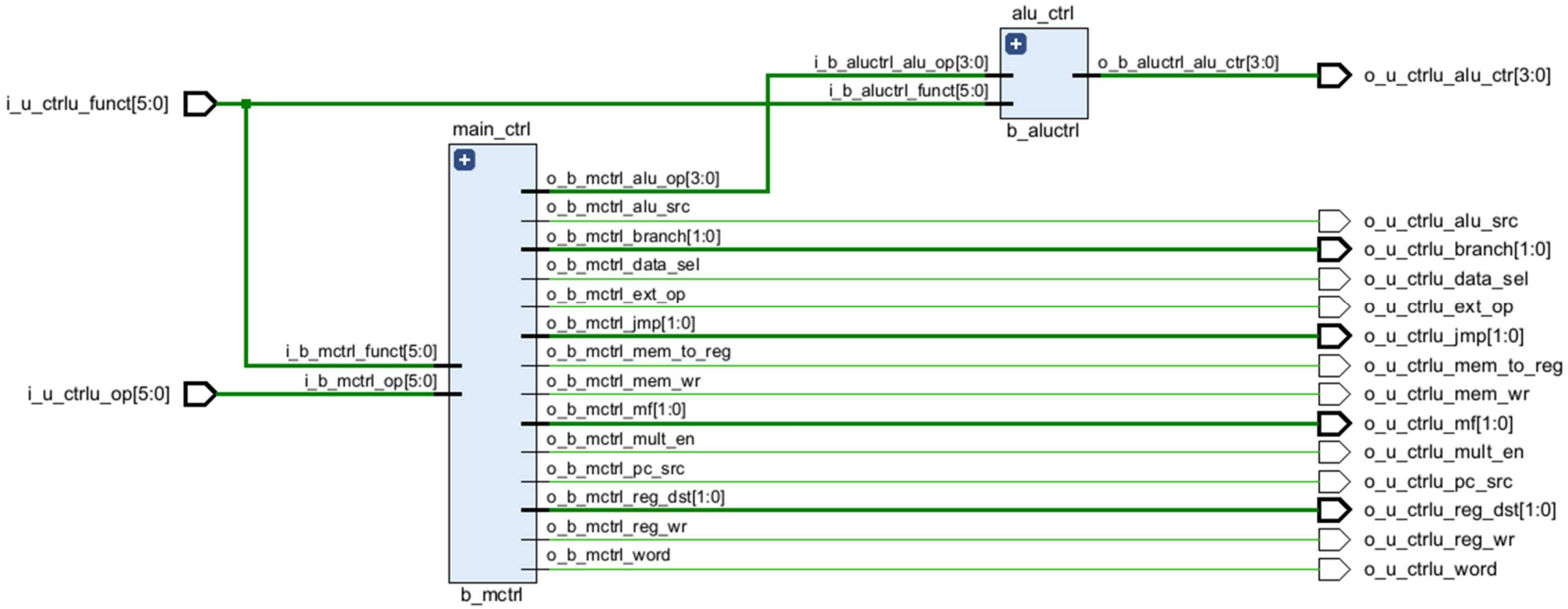


Diagram 4.3.0.2.2: Post-synthesis schematic diagram of control unit

4.3.0.5 System Verilog Model

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Ng Yu Heng
//
// Create Date: 31.08.2025 19:55:10
// File Name: u_ctrlu.sv
// Module Name: u_ctrlu
// Project Name: MIPS ISA Pipeline processor
// Code Type: RTL level
// Description: Modeling of Control Unit block
//
//////////////////////////////////////////////////////////////////////////////////
module u_ctrlu(
    // Inputs
    input  logic [5:0] i_u_ctrlu_op,      // Instruction opcode
    input  logic [5:0] i_u_ctrlu_funct,   // R-type function field

    // Outputs
    output logic       o_u_ctrlu_pc_src,   // PC source select
    output logic [1:0] o_u_ctrlu_mf,       // Hi/Lo register select
    output logic       o_u_ctrlu_ext_op,  // Zero/Sign extend select
    output logic [1:0] o_u_ctrlu_jmp,      // Jump instruction indicator
    output logic       o_u_ctrlu_mult_en, // Multiplier enable
    output logic [1:0] o_u_ctrlu_reg_dst, // Destination register select
    output logic       o_u_ctrlu_alu_src, // ALU input source select
    output logic [3:0] o_u_ctrlu_alu_ctr, // ALU operation control
    output logic       o_u_ctrlu_data_sel,// PC+4 select
    output logic       o_u_ctrlu_mem_wr,  // Memory write enable
    output logic [1:0] o_u_ctrlu_branch,  // Branch instruction indicator
    output logic       o_u_ctrlu_word,     // Word or Byte indicator
    output logic       o_u_ctrlu_reg_wr,  // Register file write enable
    output logic       o_u_ctrlu_mem_to_reg // Writeback source select
    );

    // Internal signals between main control and ALU control
    logic [3:0] main_ctrl_alu_op;  // ALU operation code from main control to ALU control

    // Instantiate Main Control Block
    b_mctrl main_ctrl(
        // Inputs
        .i_b_mctrl_op(i_u_ctrlu_op),
        .i_b_mctrl_funct(i_u_ctrlu_funct),

        // Outputs
        .o_b_mctrl_pc_src(o_u_ctrlu_pc_src),
        .o_b_mctrl_mf(o_u_ctrlu_mf),
        .o_b_mctrl_ext_op(o_u_ctrlu_ext_op),
        .o_b_mctrl_jmp(o_u_ctrlu_jmp),
        .o_b_mctrl_mult_en(o_u_ctrlu_mult_en),
        .o_b_mctrl_reg_dst(o_u_ctrlu_reg_dst),
        .o_b_mctrl_alu_src(o_u_ctrlu_alu_src),
        .o_b_mctrl_alu_op(main_ctrl_alu_op),  // Connected to ALU control
        .o_b_mctrl_data_sel(o_u_ctrlu_data_sel),
        .o_b_mctrl_mem_wr(o_u_ctrlu_mem_wr),
        .o_b_mctrl_branch(o_u_ctrlu_branch),
        .o_b_mctrl_word(o_u_ctrlu_word),
```

```
        .o_b_mctrl_reg_wr(o_u_ctrlu_reg_wr),
        .o_b_mctrl_mem_to_reg(o_u_ctrlu_mem_to_reg)
    );

    // Instantiate ALU Control Block
    b_aluctrl alu_ctrl (
        // Inputs
        .i_b_aluctrl_alu_op(main_ctrl_alu_op),  // From main control
        .i_b_aluctrl_funct(i_u_ctrlu_funct),
        // Outputs
        .o_b_aluctrl_alu_ctr(o_u_ctrlu_alu_ctr)
    );

endmodule
```

4.3.0.6 Test Plan

No.	Test Case	Description of test vector Generation	Expected Output	Status
1	R-type: add	1. Set i_u_ctrlu_op[5:0] = 000000. 2. Set i_u_ctrlu_funct[5:0] = 100000.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=01, o_u_ctrlu_alu_src=0, o_u_ctrlu_alu_ctr[3:0]=0000, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	PASS
2	R-type: sub	1. Set i_u_ctrlu_op[5:0] = 000000. 2. Set i_u_ctrlu_funct[5:0] = 100010.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=01, o_u_ctrlu_alu_src=0, o_u_ctrlu_alu_ctr[3:0]=0010, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	PASS
3	R-type: and	1. Set i_u_ctrlu_op[5:0] = 000000. 2. Set i_u_ctrlu_funct[5:0] = 100100.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=01, o_u_ctrlu_alu_src=0, o_u_ctrlu_alu_ctr[3:0]=0100, o_u_ctrlu_data_sel=0,	PASS

			o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	
4	R-type: or	1. Set i_u_ctrlu_op[5:0] = 000000. 2. Set i_u_ctrlu_funct[5:0] = 100101.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=01, o_u_ctrlu_alu_src=0, o_u_ctrlu_alu_ctr[3:0]=0101, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	PASS
5	R-type: xor	1. Set i_u_ctrlu_op[5:0] = 000000. 2. Set i_u_ctrlu_funct[5:0] = 100110.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=01, o_u_ctrlu_alu_src=0, o_u_ctrlu_alu_ctr[3:0]=0110, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	PASS
6	R-type: nor	1. Set i_u_ctrlu_op[5:0] = 000000. 2. Set i_u_ctrlu_funct[5:0] = 100111.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=01, o_u_ctrlu_alu_src=0, o_u_ctrlu_alu_ctr[3:0]=0111, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	PASS
7	R-type: slt	1. Set i_u_ctrlu_op[5:0] = 000000. 2. Set i_u_ctrlu_funct[5:0] = 101010.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=01, o_u_ctrlu_alu_src=0, o_u_ctrlu_alu_ctr[3:0]=1010, o_u_ctrlu_data_sel=0,	PASS

			o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	
8	R-type: sll	1. Set i_u_ctrlu_op[5:0] = 000000. 2. Set i_u_ctrlu_funct[5:0] = 000000.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=01, o_u_ctrlu_alu_src=0, o_u_ctrlu_alu_ctr[3:0]=0001, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	PASS
9	R-type: srl	1. Set i_u_ctrlu_op[5:0] = 000000. 2. Set i_u_ctrlu_funct[5:0] = 000010.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=01, o_u_ctrlu_alu_src=0, o_u_ctrlu_alu_ctr[3:0]=0011, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	PASS
10	R-type: multu	1. Set i_u_ctrlu_op[5:0] = 000000. 2. Set i_u_ctrlu_funct[5:0] = 011001.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=1, o_u_ctrlu_reg_dst[1:0]=01, o_u_ctrlu_alu_src=0, o_u_ctrlu_alu_ctr[3:0]=xxxx, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	PASS
11	R-type: mfhi	1. Set i_u_ctrlu_op[5:0] = 000000. 2. Set i_u_ctrlu_funct[5:0] = 010000.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=11, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=01, o_u_ctrlu_alu_src=0, o_u_ctrlu_alu_ctr[3:0]=0000, o_u_ctrlu_data_sel=0,	PASS

			o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	
12	R-type: mflo	1. Set i_u_ctrlu_op[5:0] = 000000. 2. Set i_u_ctrlu_funct[5:0] = 010010.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=01, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=01, o_u_ctrlu_alu_src=0, o_u_ctrlu_alu_ctr[3:0]=0000, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	PASS
13	R-type: jr	1. Set i_u_ctrlu_op[5:0] = 000000. 2. Set i_u_ctrlu_funct[5:0] = 001000.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=10, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=10, o_u_ctrlu_alu_src=0, o_u_ctrlu_alu_ctr[3:0]=xxxx, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	PASS
14	R-type: jalr	1. Set i_u_ctrlu_op[5:0] = 000000. 2. Set i_u_ctrlu_funct[5:0] = 001001.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=10, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=01, o_u_ctrlu_alu_src=0, o_u_ctrlu_alu_ctr[3:0]=xxxx, o_u_ctrlu_data_sel=1, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	PASS
15	I-type: lw	1. Set i_u_ctrlu_op[5:0] = 100011. 2. Set i_u_ctrlu_funct[5:0] = xxxxxx.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=1, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=00, o_u_ctrlu_alu_src=1, o_u_ctrlu_alu_ctr[3:0]=0000, o_u_ctrlu_data_sel=0,	PASS

			o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=1, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=1	
16	I-type: sw	1. Set i_u_ctrlu_op[5:0] = 101011. 2. Set i_u_ctrlu_funct[5:0] = xxxxxx.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=1, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=xx, o_u_ctrlu_alu_src=1, o_u_ctrlu_alu_ctr[3:0]=0000, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=1, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=1, o_u_ctrlu_reg_wr=0, o_u_ctrlu_mem_to_reg=x	PASS
17	I-type: lb	1. Set i_u_ctrlu_op[5:0] = 100000. 2. Set i_u_ctrlu_funct[5:0] = xxxxxx.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=1, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=00, o_u_ctrlu_alu_src=1, o_u_ctrlu_alu_ctr[3:0]=0000, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=0, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=1	PASS
18	I-type: sb	1. Set i_u_ctrlu_op[5:0] = 101000. 2. Set i_u_ctrlu_funct[5:0] = xxxxxx.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=1, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=xx, o_u_ctrlu_alu_src=1, o_u_ctrlu_alu_ctr[3:0]=0000, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=1, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=0, o_u_ctrlu_reg_wr=0, o_u_ctrlu_mem_to_reg=x	PASS
19	I-type: addi	1. Set i_u_ctrlu_op[5:0] = 001000. 2. Set i_u_ctrlu_funct[5:0] = xxxxxx.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=1, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=00, o_u_ctrlu_alu_src=1, o_u_ctrlu_alu_ctr[3:0]=0000, o_u_ctrlu_data_sel=0,	PASS

			o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	
20	I-type: andi	1. Set i_u_ctrlu_op[5:0] = 001100. 2. Set i_u_ctrlu_funct[5:0] = xxxxxx.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=0, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=00, o_u_ctrlu_alu_src=1, o_u_ctrlu_alu_ctr[3:0]=0100, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	PASS
21	I-type: ori	1. Set i_u_ctrlu_op[5:0] = 001101. 2. Set i_u_ctrlu_funct[5:0] = xxxxxx.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=0, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=00, o_u_ctrlu_alu_src=1, o_u_ctrlu_alu_ctr[3:0]=0101, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	PASS
22	I-type: slti	1. Set i_u_ctrlu_op[5:0] = 001010. 2. Set i_u_ctrlu_funct[5:0] = xxxxxx.	o_u_ctrlu_pc_src=0, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=1, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=00, o_u_ctrlu_alu_src=1, o_u_ctrlu_alu_ctr[3:0]=1010, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	PASS
23	I-type: beq	1. Set i_u_ctrlu_op[5:0] = 000100. 2. Set i_u_ctrlu_funct[5:0] = xxxxxx.	o_u_ctrlu_pc_src=1, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=xx, o_u_ctrlu_alu_src=0, o_u_ctrlu_alu_ctr[3:0]=0010, o_u_ctrlu_data_sel=0,	PASS

			o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=11, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=0, o_u_ctrlu_mem_to_reg=x	
24	I-type: bne	1. Set i_u_ctrlu_op[5:0] = 000101. 2. Set i_u_ctrlu_funct[5:0] = xxxxxx	o_u_ctrlu_pc_src=1, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=xx, o_u_ctrlu_alu_src=0, o_u_ctrlu_alu_ctr[3:0]=0010, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=01, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=0, o_u_ctrlu_mem_to_reg=x	PASS
25	J-type: j	1. Set i_u_ctrlu_op[5:0] = 000010. 2. Set i_u_ctrlu_funct[5:0] = xxxxxx.	o_u_ctrlu_pc_src=1, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=01, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=xx, o_u_ctrlu_alu_src=x, o_u_ctrlu_alu_ctr[3:0]=xxxx, o_u_ctrlu_data_sel=0, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=0, o_u_ctrlu_mem_to_reg=x	PASS
26	J-type: jal	1. Set i_u_ctrlu_op[5:0] = 000011. 2. Set i_u_ctrlu_funct[5:0] = xxxxxx	o_u_ctrlu_pc_src=1, o_u_ctrlu_mf[1:0]=00, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=10, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=01, o_u_ctrlu_alu_src=x, o_u_ctrlu_alu_ctr[3:0]=xxxx, o_u_ctrlu_data_sel=1, o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	PASS
27	I-type: lui	1. Set i_u_ctrlu_op[5:0] = 001111. 2. Set i_u_ctrlu_funct[5:0] = xxxxxx.	o_u_ctrl_pc_src=0, o_b_mctrl_mf[1:0]=00, o_u_ctrlu_ext_op=x, o_u_ctrlu_jmp[1:0]=00, o_u_ctrlu_mult_en=0, o_u_ctrlu_reg_dst[1:0]=00, o_u_ctrlu_alu_src=1, o_u_ctrlu_alu_ctr[3:0]=0111, o_u_ctrlu_data_sel=0,	PASS

			o_u_ctrlu_mem_wr=0, o_u_ctrlu_branch[1:0]=00, o_u_ctrlu_word=x, o_u_ctrlu_reg_wr=1, o_u_ctrlu_mem_to_reg=0	

4.3.0.7 Testbench and Simulation results

4.3.0.7.1 Testbench

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Ng Yu Heng
//
// Create Date: 31.08.2025 23:58:27
// File Name: tb_u_ctrlu.sv
// Module Name: tb_u_ctrlu
// Project Name: MIPS ISA Pipeline processor
// Code Type: Behavioural
// Description: Testbench for Control unit block
//
//////////////////////////////////////////////////////////////////////////////////

module tb_u_ctrlu();
    // Inputs
    logic [5:0] tb_i_u_ctrlu_op;
    logic [5:0] tb_i_u_ctrlu_funct;

    // Outputs
    logic      tb_o_u_ctrlu_pc_src;
    logic [1:0] tb_o_u_ctrlu_mf;
    logic      tb_o_u_ctrlu_ext_op;
    logic [1:0] tb_o_u_ctrlu_jmp;
    logic      tb_o_u_ctrlu_mult_en;
    logic [1:0] tb_o_u_ctrlu_reg_dst;
    logic      tb_o_u_ctrlu_alu_src;
    logic [3:0] tb_o_u_ctrlu_alu_ctr;
    logic      tb_o_u_ctrlu_data_sel;
    logic      tb_o_u_ctrlu_mem_wr;
    logic [1:0] tb_o_u_ctrlu_branch;
    logic      tb_o_u_ctrlu_word;
    logic      tb_o_u_ctrlu_reg_wr;
    logic      tb_o_u_ctrlu_mem_to_reg;

    // Clock and cycle definition
    parameter CC = 10; // 10ns per clock cycle
    logic clk;

    // Instantiate DUT
    u_ctrlu dut (
        .i_u_ctrlu_op(tb_i_u_ctrlu_op),
        .i_u_ctrlu_funct(tb_i_u_ctrlu_funct),
        .o_u_ctrlu_pc_src(tb_o_u_ctrlu_pc_src),
        .o_u_ctrlu_mf(tb_o_u_ctrlu_mf),
        .o_u_ctrlu_ext_op(tb_o_u_ctrlu_ext_op),
        .o_u_ctrlu_jmp(tb_o_u_ctrlu_jmp),
        .o_u_ctrlu_mult_en(tb_o_u_ctrlu_mult_en),
        .o_u_ctrlu_reg_dst(tb_o_u_ctrlu_reg_dst),
```

```
    .o_u_ctrlu_alu_src(tb_o_u_ctrlu_alu_src),
    .o_u_ctrlu_alu_ctr(tb_o_u_ctrlu_alu_ctr),
    .o_u_ctrlu_data_sel(tb_o_u_ctrlu_data_sel),
    .o_u_ctrlu_mem_wr(tb_o_u_ctrlu_mem_wr),
    .o_u_ctrlu_branch(tb_o_u_ctrlu_branch),
    .o_u_ctrlu_word(tb_o_u_ctrlu_word),
    .o_u_ctrlu_reg_wr(tb_o_u_ctrlu_reg_wr),
    .o_u_ctrlu_mem_to_reg(tb_o_u_ctrlu_mem_to_reg)
);

// Clock generation
always #(CC/2) clk = ~clk;

initial begin
    // Initialize
    clk = 0;
    tb_i_u_ctrlu_op = 6'bx;
    tb_i_u_ctrlu_funct = 6'bx;

    // Wait 2 cycles
    repeat(2) @(negedge clk);

    //---------------------------------------
    // Test Case 1: R-type add
    //---------------------------------------
    tb_i_u_ctrlu_op = 6'b000000;
    tb_i_u_ctrlu_funct = 6'b100000;
    repeat(2) @(negedge clk);

    //---------------------------------------
    // Test Case 2: R-type sub
    //---------------------------------------
    tb_i_u_ctrlu_op = 6'b000000;
    tb_i_u_ctrlu_funct = 6'b100010;
    repeat(2) @(negedge clk);

    //---------------------------------------
    // Test Case 3: R-type and
    //---------------------------------------
    tb_i_u_ctrlu_op = 6'b000000;
    tb_i_u_ctrlu_funct = 6'b100100;
    repeat(2) @(negedge clk);

    //---------------------------------------
    // Test Case 4: R-type or
    //---------------------------------------
    tb_i_u_ctrlu_op = 6'b000000;
    tb_i_u_ctrlu_funct = 6'b100101;
    repeat(2) @(negedge clk);

    //---------------------------------------
    // Test Case 5: R-type xor
    //---------------------------------------
    tb_i_u_ctrlu_op = 6'b000000;
    tb_i_u_ctrlu_funct = 6'b100110;
    repeat(2) @(negedge clk);
```

```
//----------------------------------------
// Test Case 6: R-type nor
//----------------------------------------
tb_i_u_ctrlu_op = 6'b000000;
tb_i_u_ctrlu_funct = 6'b100111;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 7: R-type slt
//----------------------------------------
tb_i_u_ctrlu_op = 6'b000000;
tb_i_u_ctrlu_funct = 6'b101010;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 8: R-type sll
//----------------------------------------
tb_i_u_ctrlu_op = 6'b000000;
tb_i_u_ctrlu_funct = 6'b000000;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 9: R-type srl
//----------------------------------------
tb_i_u_ctrlu_op = 6'b000000;
tb_i_u_ctrlu_funct = 6'b000010;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 10: R-type multu
//----------------------------------------
tb_i_u_ctrlu_op = 6'b000000;
tb_i_u_ctrlu_funct = 6'b011001;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 11: R-type mfhi
//----------------------------------------
tb_i_u_ctrlu_op = 6'b000000;
tb_i_u_ctrlu_funct = 6'b010000;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 12: R-type mflo
//----------------------------------------
tb_i_u_ctrlu_op = 6'b000000;
tb_i_u_ctrlu_funct = 6'b010010;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 13: R-type jr
//----------------------------------------
tb_i_u_ctrlu_op = 6'b000000;
tb_i_u_ctrlu_funct = 6'b001000;
repeat(2) @(negedge clk);

//----------------------------------------
```

```
// Test Case 14: R-type jalr
//----------------------------------------
tb_i_u_ctrlu_op = 6'b000000;
tb_i_u_ctrlu_funct = 6'b001001;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 15: I-type lw
//----------------------------------------
tb_i_u_ctrlu_op = 6'b100011;
tb_i_u_ctrlu_funct = 6'bx;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 16: I-type sw
//----------------------------------------
tb_i_u_ctrlu_op = 6'b101011;
tb_i_u_ctrlu_funct = 6'bx;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 17: I-type lb
//----------------------------------------
tb_i_u_ctrlu_op = 6'b100000;
tb_i_u_ctrlu_funct = 6'bx;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 18: I-type sb
//----------------------------------------
tb_i_u_ctrlu_op = 6'b101000;
tb_i_u_ctrlu_funct = 6'bx;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 19: I-type addi
//----------------------------------------
tb_i_u_ctrlu_op = 6'b001000;
tb_i_u_ctrlu_funct = 6'bx;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 20: I-type andi
//----------------------------------------
tb_i_u_ctrlu_op = 6'b001100;
tb_i_u_ctrlu_funct = 6'bx;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 21: I-type ori
//----------------------------------------
tb_i_u_ctrlu_op = 6'b001101;
tb_i_u_ctrlu_funct = 6'bx;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 22: I-type slti
```

```
        //---------------------------------------
        tb_i_u_ctrlu_op = 6'b001010;
        tb_i_u_ctrlu_funct = 6'bx;
        repeat(2) @(negedge clk);

        //---------------------------------------
        // Test Case 23: I-type beq
        //---------------------------------------
        tb_i_u_ctrlu_op = 6'b000100;
        tb_i_u_ctrlu_funct = 6'bx;
        repeat(2) @(negedge clk);

        //---------------------------------------
        // Test Case 24: I-type bne
        //---------------------------------------
        tb_i_u_ctrlu_op = 6'b000101;
        tb_i_u_ctrlu_funct = 6'bx;
        repeat(2) @(negedge clk);

        //---------------------------------------
        // Test Case 25: J-type j
        //---------------------------------------
        tb_i_u_ctrlu_op = 6'b000010;
        tb_i_u_ctrlu_funct = 6'bx;
        repeat(2) @(negedge clk);

        //---------------------------------------
        // Test Case 26: J-type jal
        //---------------------------------------
        tb_i_u_ctrlu_op = 6'b000011;
        tb_i_u_ctrlu_funct = 6'bx;
        repeat(2) @(negedge clk);

        //---------------------------------------
        // Test Case 27: I-type lui
        //---------------------------------------
        tb_i_u_ctrlu_op = 6'b001111;
        tb_i_u_ctrlu_funct = 6'bx;
        repeat(2) @(negedge clk);

        // Finish simulation
        repeat(2) @(negedge clk);
        $finish;
    end

endmodule
```

4.3.0.7.2 Simulation result

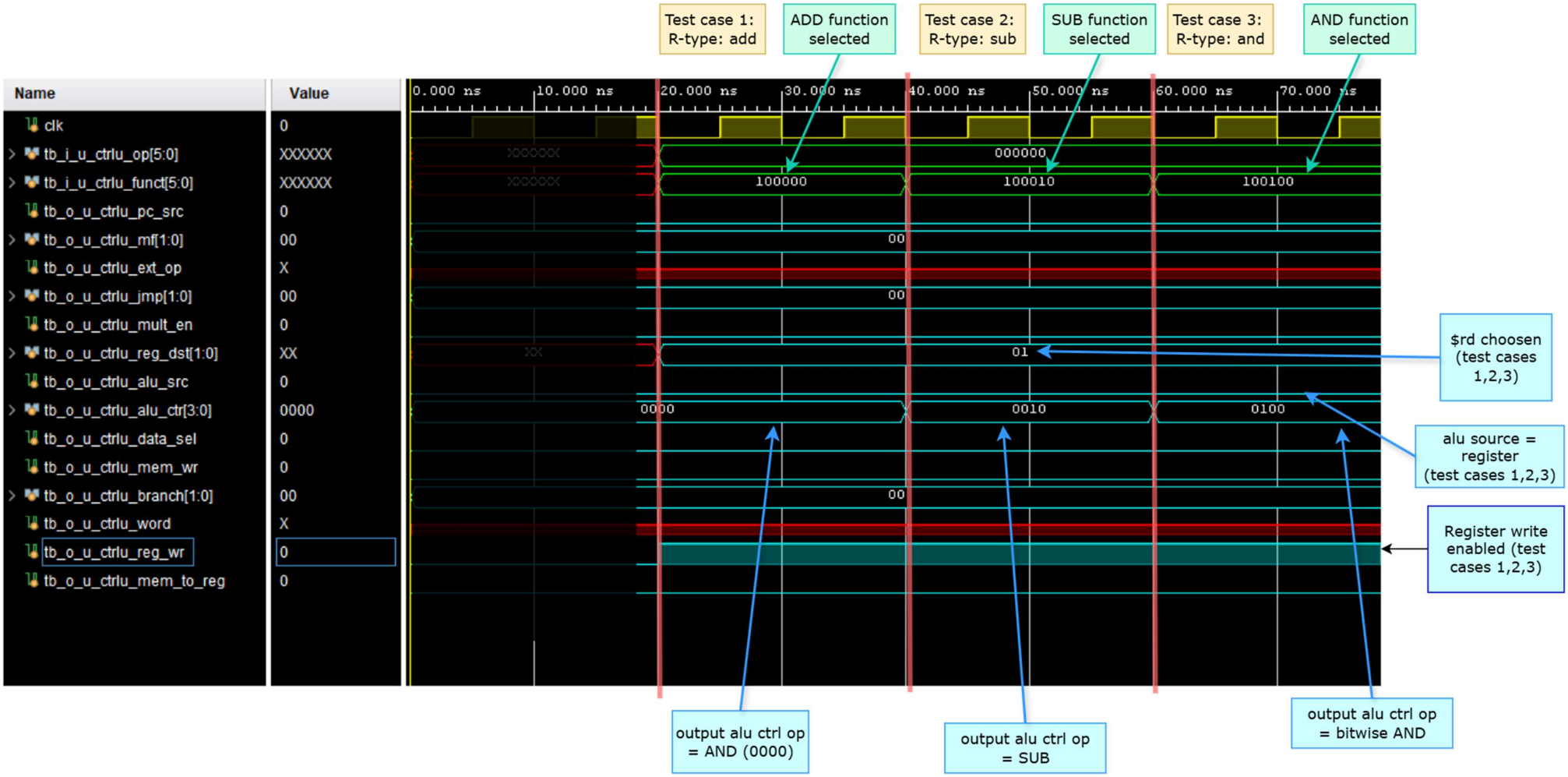


Diagram 4.3.0.7.2.1 Test case 1, 2, 3

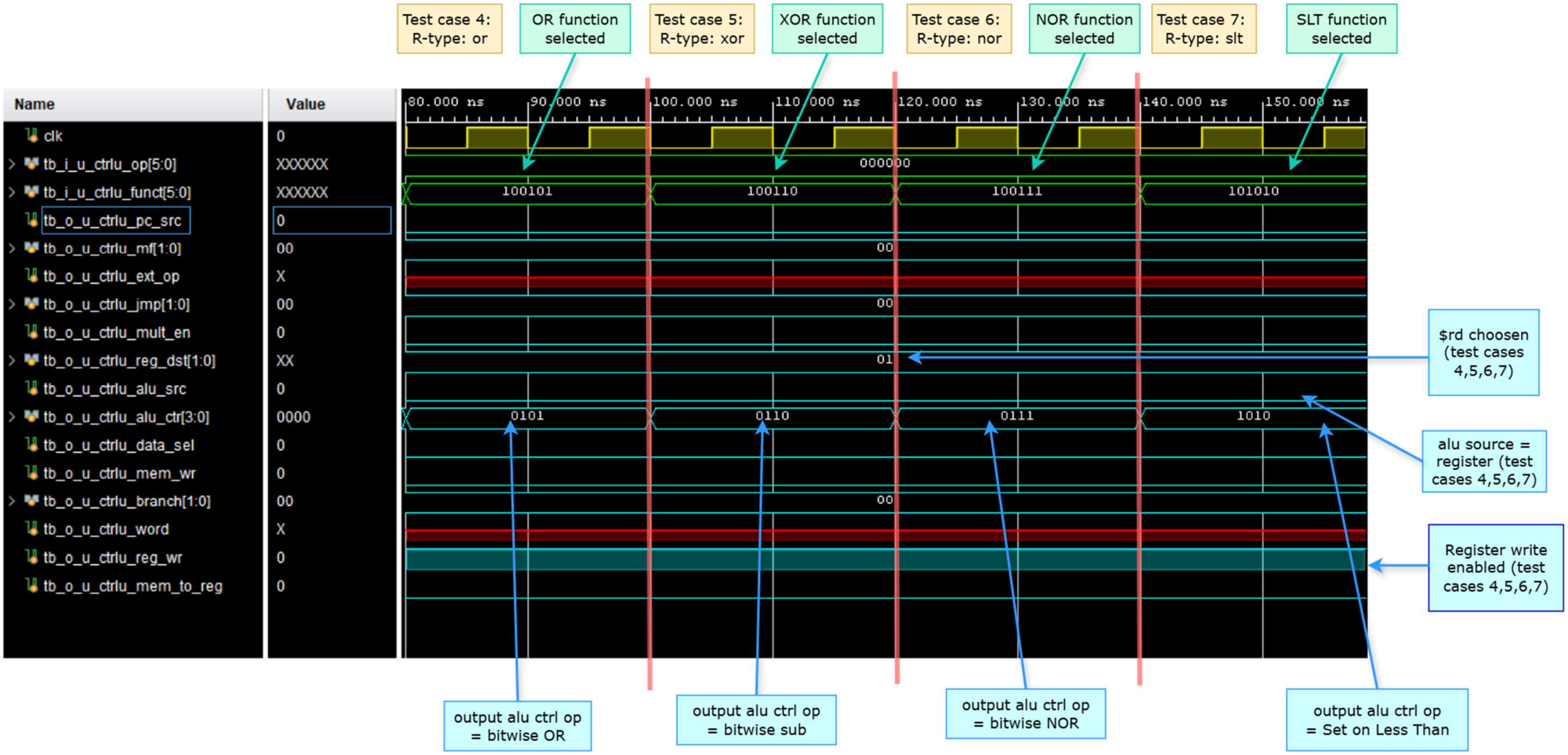


Diagram 4.3.0.7.2.2 Test case 4, 5, 6, 7

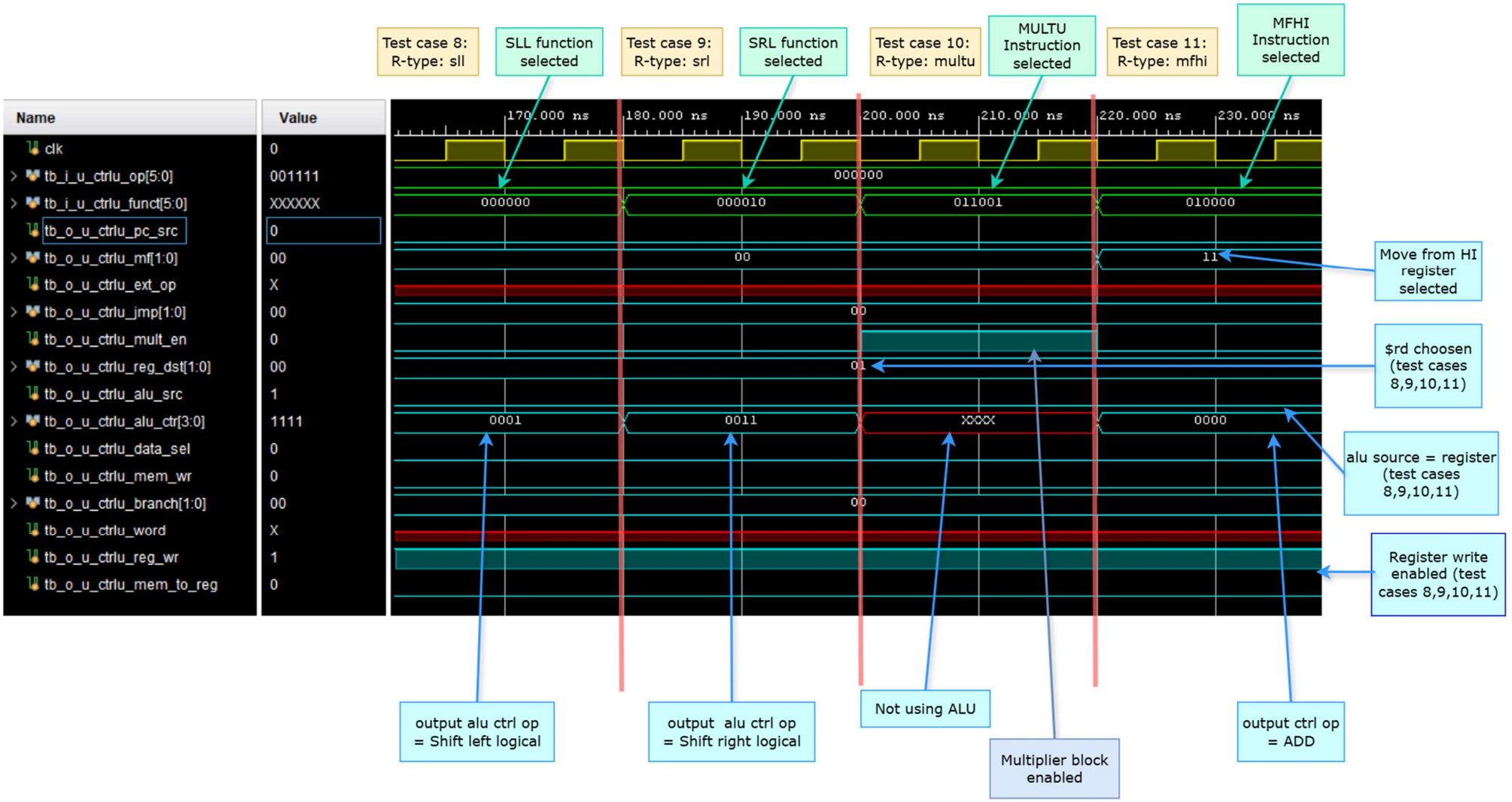


Diagram 4.3.0.7.2.3 Test case 8, 9, 10, 11

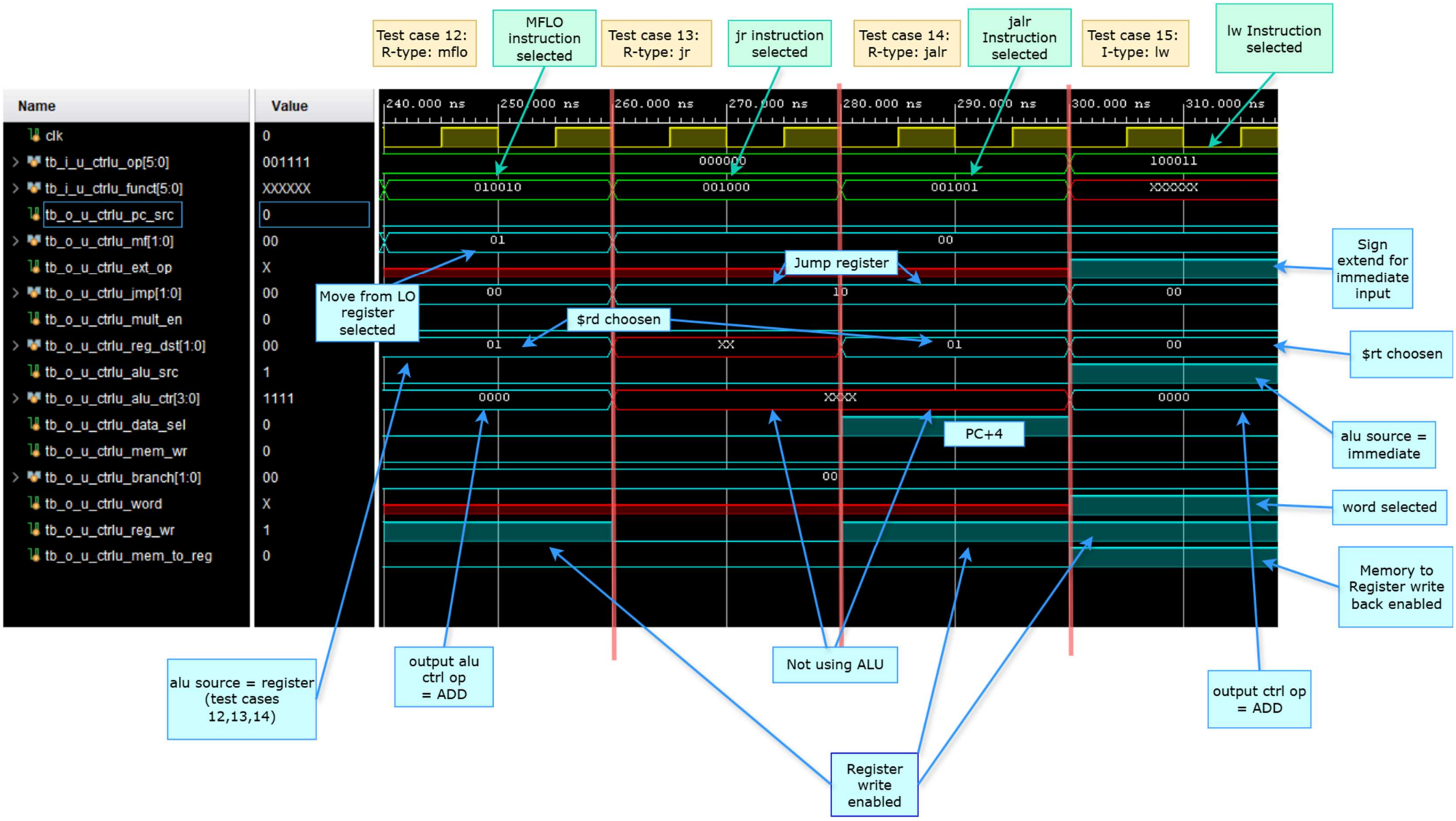


Diagram 4.3.0.7.2.4 Test case 12, 13, 14, 15

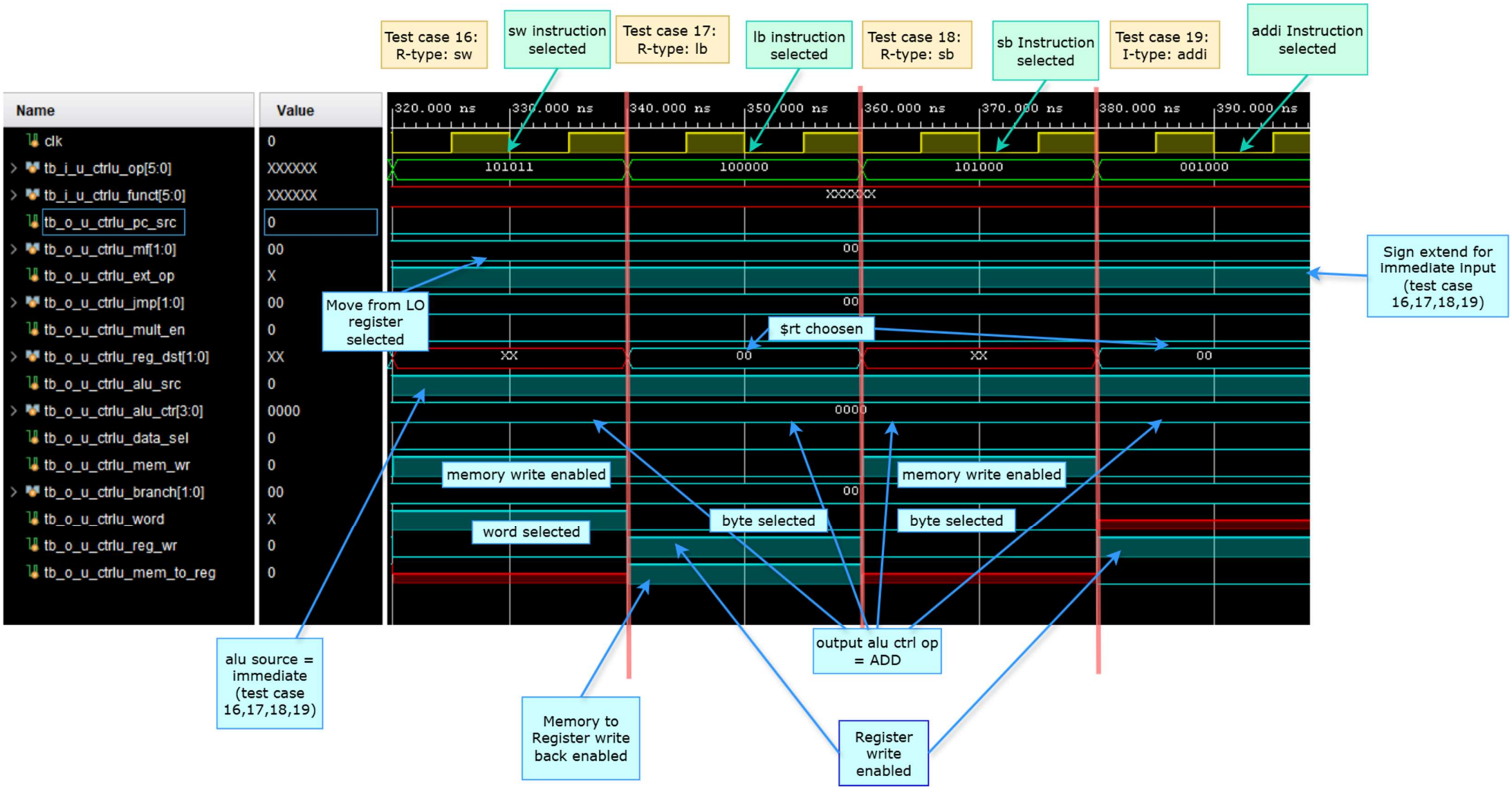


Diagram 4.3.0.7.2.5 Test case 16, 17, 18, 19

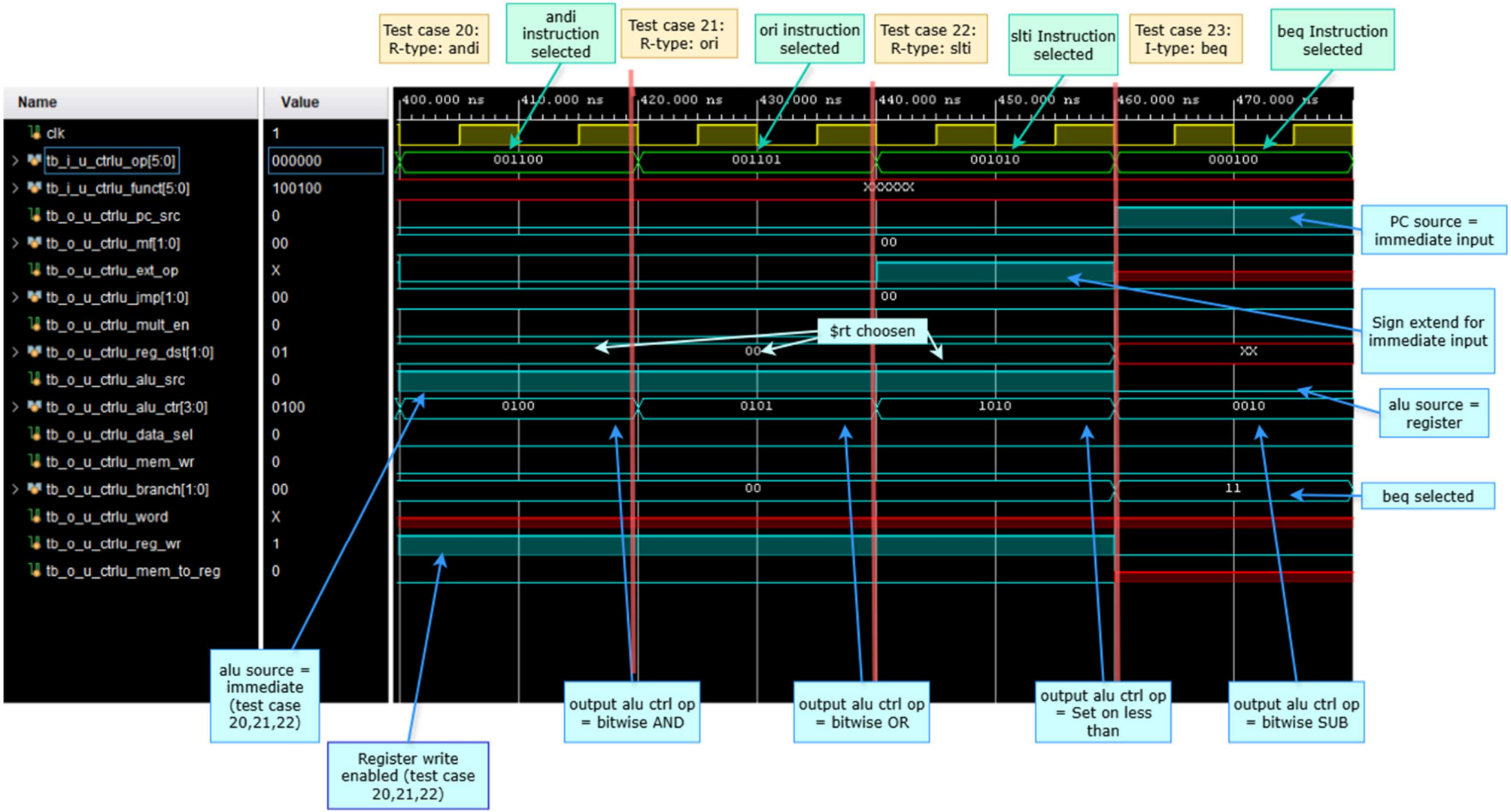


Diagram 4.3.0.7.2.6 Test case 20, 21, 22, 23

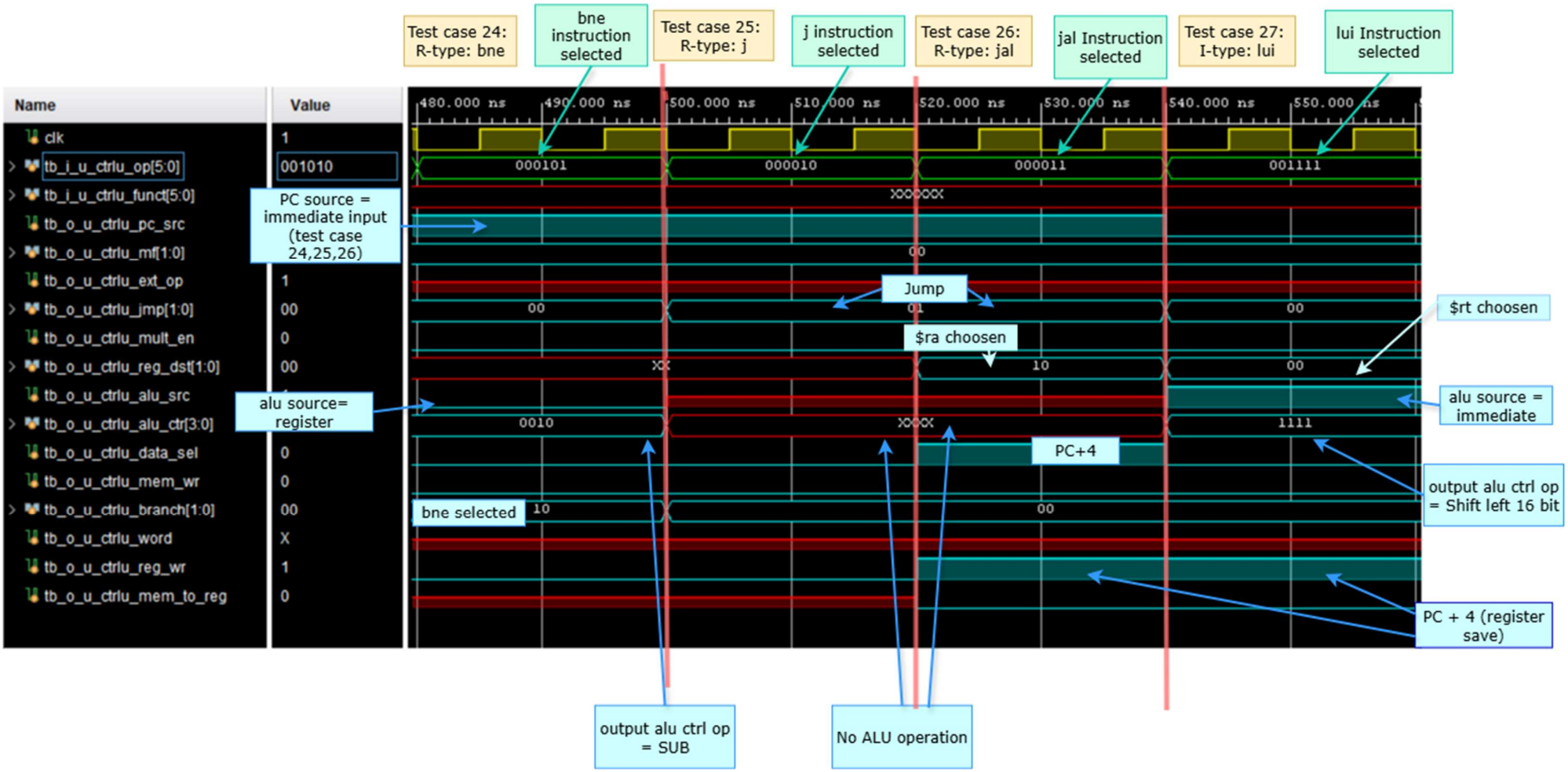


Diagram 4.3.0.7.2.7 Test case 24, 25, 26, 27

4.3.1 Main Control Block

4.3.1.1 Functionality/Feature

- Generates all top-level control signals for the pipeline based on the opcode
- Selects the destination register (RegDst), writeback source (MemToReg), and immediate sign extension mode (ExtOp).
- Chooses ALU input source (ALUSrc) and ALU behavior (ALUop).
- Control drives memory (MemWr), register file (RegWr), and PC flow (Branch, Jump).

4.3.1.2 Block interface and I/O pin description

4.3.1.2.1 Main control block interface

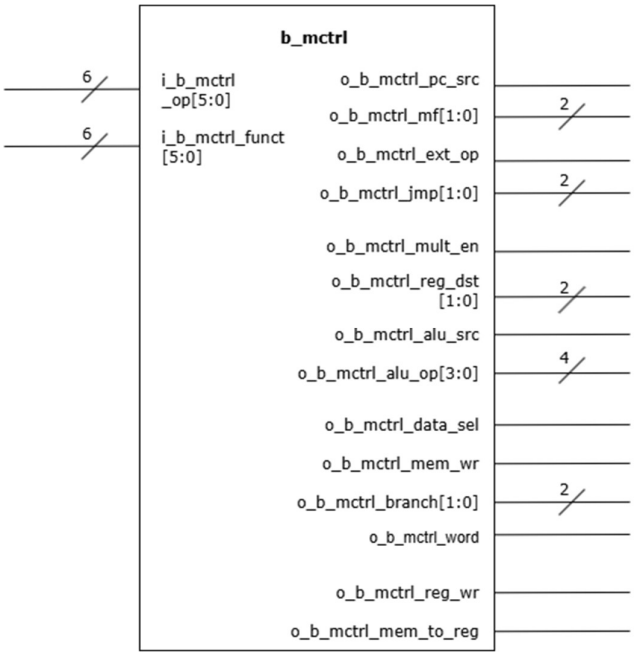


Diagram 4.3.1.2.1: Block interface of main control block

4.3.1.2.2 I/O pin description

Pin name: **Pin class:** **Pin function:**	i_b_mctrl_op[5:0] control To receive instruction opcode for instruction decoding	**Source →Destination:**	Instruction [31:26] → Main Control
Pin name: **Pin class:** **Pin function:**	i_b_mctrl_funct [5:0] control To receive R-type function field for detailed ALU decode (multu, mflo, mfhi)	**Source →Destination:**	Instruction [5:0] → Main Control
Pin name: **Pin class:** **Pin function:**	o_b_mctrl_pc_src control To select source of program counter	**Source →Destination:**	Main Control → PC Source MUX
Pin name: **Pin class:** **Pin function:**	o_b_mctrl_mf[1:0] control To select hi and lo register to receive data	**Source →Destination:**	Main Control → Register select

Pin name: **Pin class:** **Pin function:**	o_b_mctrl_ext_op control To select zero or sign extend	**Source** **→Destination:**	Main Control → Zero/Sign Extender
Pin name: **Pin class:** **Pin function:**	o_b_mctrl_jmp[1:0] control to indicate jump instruction	**Source** **→Destination:**	Main Control → PC Selection MUX
Pin name: **Pin class:** **Pin function:**	o_b_mctrl_mult_en control To enable multiplier for multiplication process	**Source** **→Destination:**	Main Control → Multiplier
Pin name: **Pin class:** **Pin function:**	o_b_mctrl_reg_dst [1:0] control To select destination register ($rt/ $rd/$ra)	**Source** **→Destination:**	Main Control → Register write MUX
Pin name: **Pin class:** **Pin function:**	o_b_mctrl_alu_src control To select ALU input source (register or immediate)	**Source** **→Destination:**	Main Control → ALU
Pin name: **Pin class:** **Pin function:**	o_b_mctrl_alu_op[3:0] control To specify ALU operation class (add/sub/logical/R-type)	**Source** **→Destination:**	Main Control → ALU Control
Pin name: **Pin class:** **Pin function:**	o_b_mctrl_data_sel control To select data to write into data memory	**Source** **→Destination:**	Main Control → Data Memory
Pin name: **Pin class:** **Pin function:**	o_b_mctrl_mem_wr control To enable memory write	**Source** **→Destination:**	Main Control → Data Memory
Pin name: **Pin class:** **Pin function:**	o_b_mctrl_branch[1:0] control To indicate branch instruction	**Source** **→Destination:**	Main Control → Branch Decision Unit
Pin name: **Pin class:** **Pin function:**	o_b_mctrl_word control To differentiate word and byte	**Source** **→Destination:**	Main Control → Word selection MUX
Pin name: **Pin class:** **Pin function:**	o_b_mctrl_reg_wr control To enable register file write	**Source** **→Destination:**	Main Control → Register File
Pin name: **Pin class:** **Pin function:**	o_b_mctrl_mem_to_reg control To select value to write back to register (ALU/memory)	**Source** **→Destination:**	Main Control → Data Memory

Table 4.3.1.2.2: I/O pin description of Main Control Block

4.3.1.3 Internal Operation: Function Table

Inputs			Outputs														
Instr	**i_b_mctrl_op [5:0]**	**i_b_mctr_funct [5:0]**	**IF**		**ID**		**EX**					**MEM**				**WB**	
			o_b_mctrl_pc_src	**o_b_mctrl_mf[1:0]**	**o_b_mctrl_ext_op**	**o_b_mctr_jmp[1:0]**	**o_b_mctrl_mult_en**	**o_b_mctrl_reg_dst [1:0]**	**o_b_mctrl_alu_src**	**Required ALU action**	**o_b_mctrl_alu_op [3:0]**	**o_b_mctrl_data_sel**	**o_b_mctrl_memWr**	**o_b_mctr_branch[1:0]**	**o_b_mctrl_word**	**o_b_mctrl_reg_wr**	**o_b_mctrl_mem_to_reg**
R-type (except multu, mfhi, mflo, jr, jalr)	000000	xxxxxx	0	00	x	00	0	01	0	R-type	1000	0	0	00	x	1	0
multu	000000	011001	0	00	x	00	1	xx	x	x	xxxx	0	0	00	x	1	0
mfhi	000000	010000	0	11	x	00	0	01	0	add	0000	0	0	00	x	1	0
mflo	000000	010010	0	01	x	00	0	01	0	add	0000	0	0	00	x	1	0
jr	000000	xxxxxx	1	00	x	10	0	xx	0	x	xxxx	0	0	00	x	0	0
jalr	000000	xxxxxx	1	00	x	10	0	01	0	x	xxxx	1	0	00	x	1	0
lw	100011	xxxxxx	0	00	1	00	0	00	1	add	0000	0	0	00	x	1	1
sw	101011	xxxxxx	0	00	1	00	0	xx	1	add	0000	0	1	00	1	0	x
lb	100000	xxxxxx	0	00	1	00	0	00	1	add	0000	0	0	00	1	1	1
sb	101000	xxxxxx	0	00	1	00	0	xx	1	add	0000	0	1	00	0	0	x
addi	001000	xxxxxx	0	00	1	00	0	00	1	add	0000	0	0	00	0	1	0
andi	001100	xxxxxx	0	00	0	00	0	00	1	and	0100	0	0	00	x	1	0

ori	001101	xxxxxx	0	00	0	00	0	00	1	or	0101	0	0	00	x	1	0
slti	001010	xxxxxx	0	00	1	00	0	00	1	Set on le	1010	0	0	00	x	1	0
beq	000100	xxxxxx	1	00	x	00	0	xx	0	sub	0010	0	0	11	x	0	x
bne	000101	xxxxxx	1	00	x	00	0	xx	0	sub	0010	0	0	01	x	0	x
j	000010	xxxxxx	1	00	x	01	0	xx	x	x	xxxx	0	0	00	x	0	x
jal	000011	xxxxxx	1	00	x	01	0	10	x	x	xxxx	1	0	00	x	1	0
lui	001111	xxxxxx	0	00	x	00	0	00	1	Shift left 16 bit	0111	0	0	00	x	1	0

Table 4.3.1.3: Function table of main control block

4.3.1.4 System Verilog Model

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Ng Yu Heng
//
// Create Date: 30.08.2025 00:49:53
// File Name: b_mctrl.sv
// Module Name: b_mctrl
// Project Name: MIPS ISA Pipeline processor
// Code Type: RTL level
// Description: Modeling of Main control block
//
//////////////////////////////////////////////////////////////////////////////////

module b_mctrl(
    // Inputs
    input  logic [5:0] i_b_mctrl_op,      // Instruction opcode
    input  logic [5:0] i_b_mctrl_funct,   // R-type function field

    // Outputs
    output logic       o_b_mctrl_pc_src,   // PC source select
    output logic [1:0] o_b_mctrl_mf,       // Hi/Lo register select
    output logic       o_b_mctrl_ext_op,   // Zero/Sign extend select
    output logic [1:0] o_b_mctrl_jmp,      // Jump instruction indicator
    output logic       o_b_mctrl_mult_en,  // Multiplier enable
    output logic [1:0] o_b_mctrl_reg_dst,  // Destination register select
    output logic       o_b_mctrl_alu_src,  // ALU input source select
    output logic [3:0] o_b_mctrl_alu_op,   // ALU operation class
    output logic       o_b_mctrl_data_sel, // Data memory write select
    output logic       o_b_mctrl_mem_wr,   // Memory write enable
    output logic [1:0] o_b_mctrl_branch,   // Branch instruction indicator
    output logic       o_b_mctrl_word,      // Word or Byte indicator
    output logic       o_b_mctrl_reg_wr,   // Register file write enable
    output logic       o_b_mctrl_mem_to_reg // Writeback source select

    );

    always_comb begin
        // Default values
        o_b_mctrl_pc_src   = 1'b0;
        o_b_mctrl_mf       = 2'b00;
        o_b_mctrl_ext_op   = 1'bx;
        o_b_mctrl_jmp      = 2'b00; // 10 = jump register, 01 = jump
        o_b_mctrl_mult_en  = 1'b0;
        o_b_mctrl_reg_dst  = 2'bx; // 00 = $rt, 01 = $rd, 10 = $ra
        o_b_mctrl_alu_src  = 1'b0;
        o_b_mctrl_alu_op   = 4'bx;
        o_b_mctrl_data_sel = 1'b0;
        o_b_mctrl_mem_wr   = 1'b0;
        o_b_mctrl_branch   = 2'b00; //1st bit = branch, 2nd bit choose bne or beq
        o_b_mctrl_word     = 1'bx;
        o_b_mctrl_reg_wr   = 1'b0;
        o_b_mctrl_mem_to_reg = 1'b0;

        case (i_b_mctrl_op)
            // R-type instructions
            6'b000000: begin
```

```
   case (i_b_mctrl_funct)
      6'b011001: begin // multu
         o_b_mctrl_mult_en = 1'b1;
         o_b_mctrl_reg_dst = 2'b01;
         o_b_mctrl_reg_wr  = 1'b1;
      end

      6'b010000: begin // mfhi
         o_b_mctrl_mf      = 2'b11;
         o_b_mctrl_reg_dst = 2'b01;
         o_b_mctrl_alu_op  = 4'b0000; // add
         o_b_mctrl_reg_wr  = 1'b1;
      end

      6'b010010: begin // mflo
         o_b_mctrl_mf      = 2'b01;
         o_b_mctrl_reg_dst = 2'b01;
         o_b_mctrl_alu_op  = 4'b0000; // add
         o_b_mctrl_reg_wr  = 1'b1;
      end

      6'b001000: begin // jr
         o_b_mctrl_jmp     = 2'b10;
         o_b_mctrl_reg_dst = 2'bxx;
      end

      6'b001001: begin // jalr
         o_b_mctrl_jmp     = 2'b10;
         o_b_mctrl_reg_dst = 2'b01;
         o_b_mctrl_data_sel= 1'b1;
         o_b_mctrl_reg_wr  = 1'b1;
      end

      // jr and jalr would need specific funct codes
      default: begin // Other R-type
         o_b_mctrl_reg_dst = 2'b01;
         o_b_mctrl_alu_op  = 4'b1000; // R-type
         o_b_mctrl_reg_wr  = 1'b1;
      end
   endcase
end

// Load/Store instructions
6'b100011: begin // lw
   o_b_mctrl_ext_op   = 1'b1;
   o_b_mctrl_reg_dst  = 2'b00;
   o_b_mctrl_alu_src  = 1'b1;
   o_b_mctrl_alu_op   = 4'b0000; // add
   o_b_mctrl_reg_wr   = 1'b1;
   o_b_mctrl_mem_to_reg = 1'b1;
   o_b_mctrl_word     = 1'b1;
end

6'b101011: begin // sw
   o_b_mctrl_ext_op   = 1'b1;
   o_b_mctrl_alu_src  = 1'b1;
```

```
   o_b_mctrl_alu_op   = 4'b0000; // add
   o_b_mctrl_mem_wr   = 1'b1;
   o_b_mctrl_word     = 1'b1;
   o_b_mctrl_mem_to_reg = 1'bx;
end

6'b100000: begin // lb
   o_b_mctrl_ext_op   = 1'b1;
   o_b_mctrl_reg_dst  = 2'b00;
   o_b_mctrl_alu_src  = 1'b1;
   o_b_mctrl_alu_op   = 4'b0000; // add
   o_b_mctrl_reg_wr   = 1'b1;
   o_b_mctrl_mem_to_reg = 1'b1;
   o_b_mctrl_word     = 1'b0;
end

6'b101000: begin // sb
   o_b_mctrl_ext_op   = 1'b1;
   o_b_mctrl_alu_src  = 1'b1;
   o_b_mctrl_alu_op   = 4'b0000; // add
   o_b_mctrl_mem_wr   = 1'b1;
   o_b_mctrl_word     = 1'b0;
   o_b_mctrl_mem_to_reg = 1'bx;
end

// I-type
6'b001000: begin // addi
   o_b_mctrl_ext_op   = 1'b1;
   o_b_mctrl_reg_dst  = 2'b00;
   o_b_mctrl_alu_src  = 1'b1;
   o_b_mctrl_alu_op   = 4'b0000; // add
   o_b_mctrl_reg_wr   = 1'b1;
end

6'b001100: begin // andi
   o_b_mctrl_ext_op   = 1'b0; // zero extend
   o_b_mctrl_reg_dst  = 2'b00;
   o_b_mctrl_alu_src  = 1'b1;
   o_b_mctrl_alu_op   = 4'b0100; // and
   o_b_mctrl_reg_wr   = 1'b1;
end

6'b001101: begin // ori
   o_b_mctrl_ext_op   = 1'b0; // zero extend
   o_b_mctrl_reg_dst  = 2'b00;
   o_b_mctrl_alu_src  = 1'b1;
   o_b_mctrl_alu_op   = 4'b0101; // or
   o_b_mctrl_reg_wr   = 1'b1;
end

6'b001010: begin // slti
   o_b_mctrl_ext_op   = 1'b1;
   o_b_mctrl_reg_dst  = 2'b00;
   o_b_mctrl_alu_src  = 1'b1;
   o_b_mctrl_alu_op   = 4'b0001; // sub (for comparison)
   o_b_mctrl_reg_wr   = 1'b1;
end
```

```
        // Branch instructions
        6'b000100: begin // beq
           o_b_mctrl_pc_src   = 1'b1;
           o_b_mctrl_alu_op   = 4'b0010; // sub for comparison
           o_b_mctrl_branch   = 2'b11; //beq
           o_b_mctrl_mem_to_reg = 1'bx;
        end

        6'b000101: begin // bne
           o_b_mctrl_pc_src   = 1'b1;
           o_b_mctrl_alu_op   = 4'b0010; // sub for comparison
           o_b_mctrl_branch   = 2'b10; //bne
           o_b_mctrl_mem_to_reg = 1'bx;
        end

        // Jump instructions
        6'b000010: begin // j
           o_b_mctrl_pc_src   = 1'b1;
           o_b_mctrl_jmp      = 2'b01;
           o_b_mctrl_alu_src  = 1'bx;
           o_b_mctrl_mem_to_reg = 1'bx;
        end

        6'b000011: begin // jal
           o_b_mctrl_pc_src   = 1'b1;
           o_b_mctrl_jmp      = 2'b01;
           o_b_mctrl_reg_dst  = 2'b10; // $ra
           o_b_mctrl_alu_src  = 1'bx;
           o_b_mctrl_data_sel= 1'b1;
           o_b_mctrl_reg_wr   = 1'b1;
        end

        6'b001111: begin // lui
           o_b_mctrl_alu_src  = 1'b1;
           o_b_mctrl_alu_op   = 4'b0111; // shift left 16
           o_b_mctrl_reg_dst  = 2'b00;
           o_b_mctrl_reg_wr   = 1'b1;
           o_b_mctrl_ext_op   = 1'bx;   // Don't care
        end

        default: begin
           // NOP
           o_b_mctrl_alu_op = 4'b0000;
        end
      endcase
   end

endmodule
```

4.3.1.5 Post-synthesis Schematic Diagram

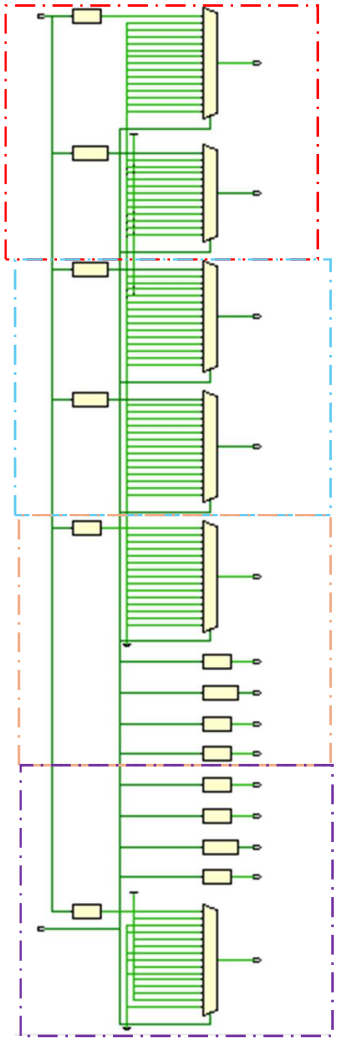


Diagram 4.3.1.5.1 Post-synthesis Schematic Diagram of Main control block

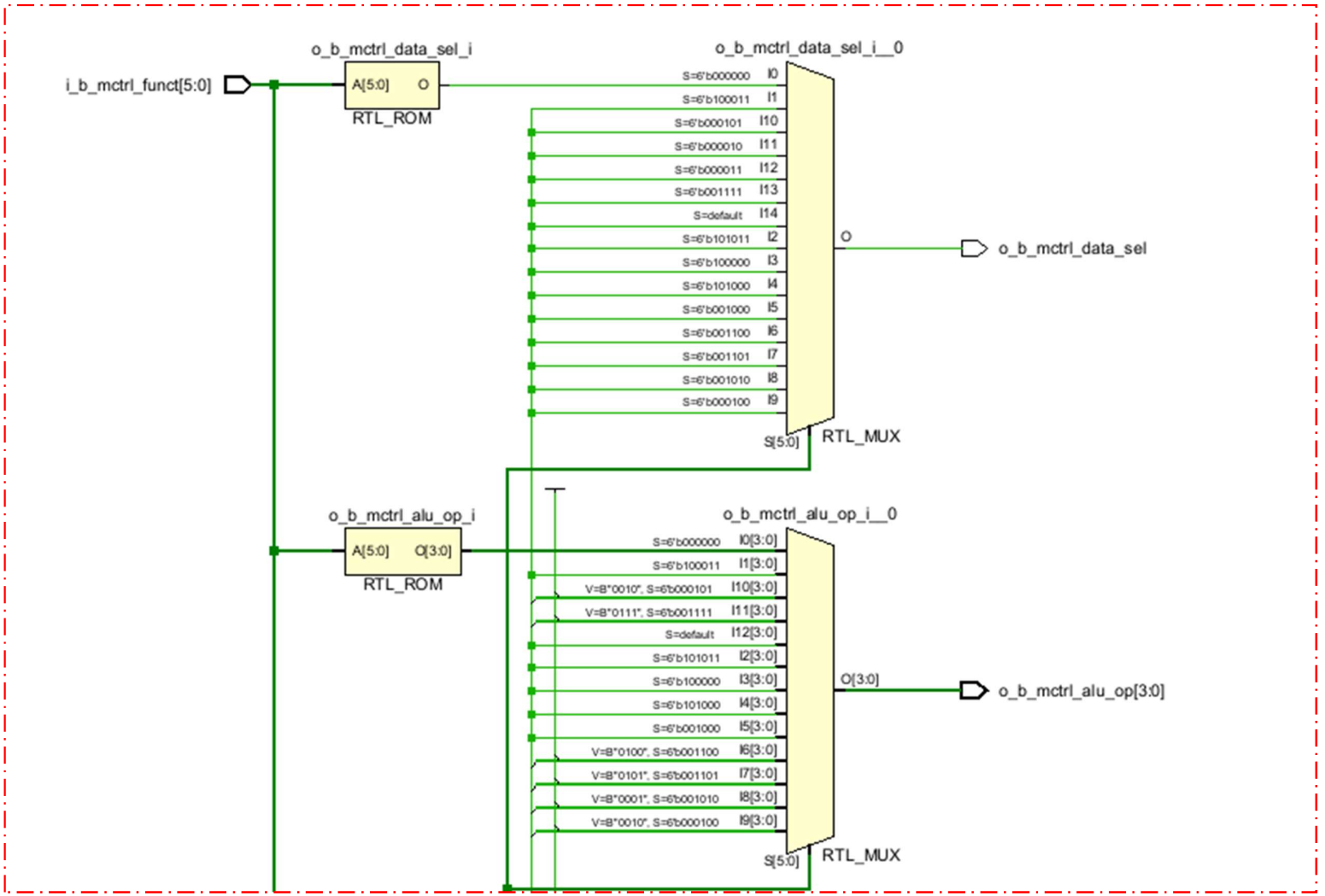


Diagram 4.3.1.5.2 Part 1 of post-synthesis Schematic Diagram of Main control block

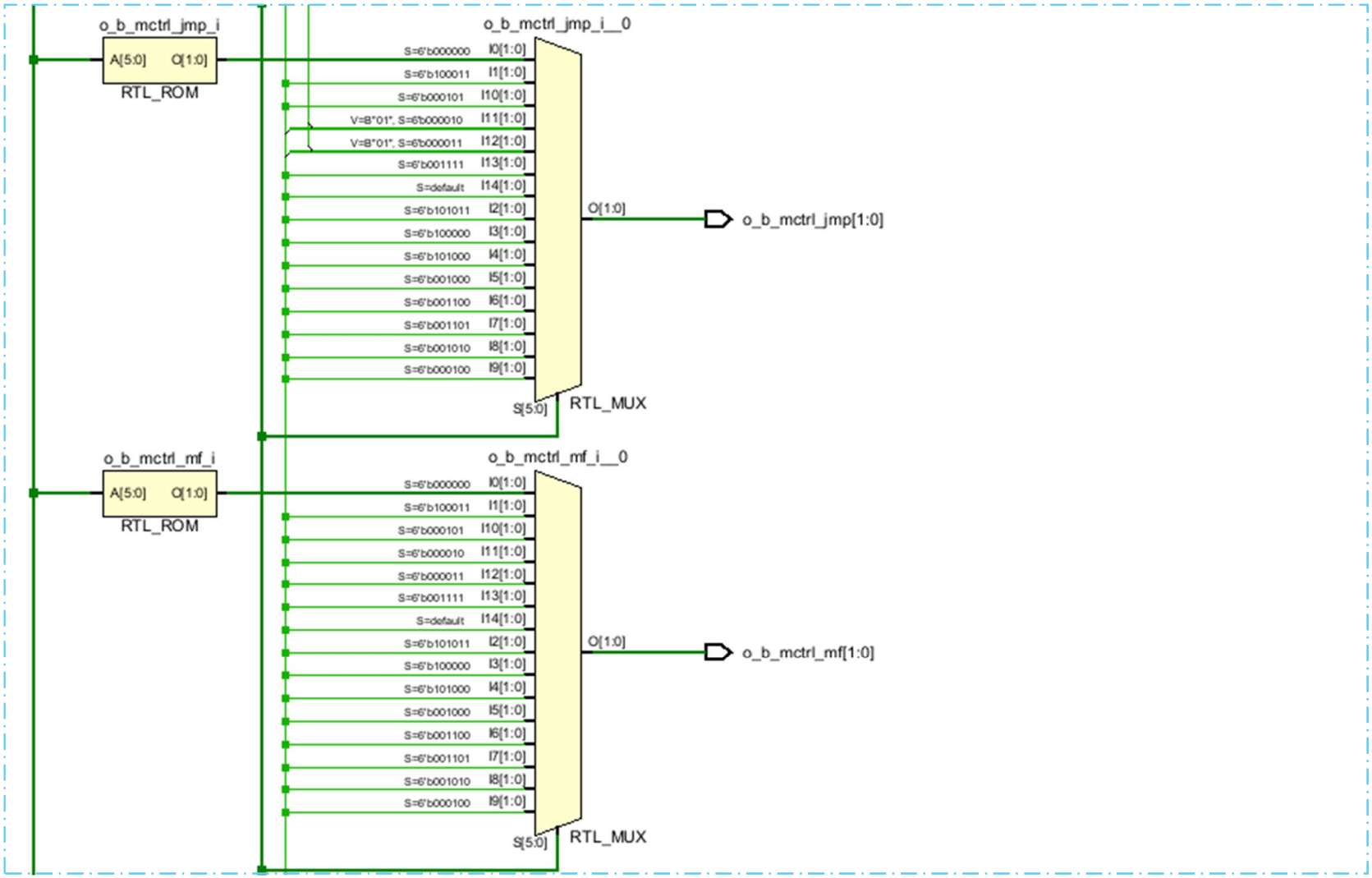


Diagram 4.3.1.5.3 Part 2 of post-synthesis Schematic Diagram of Main control block

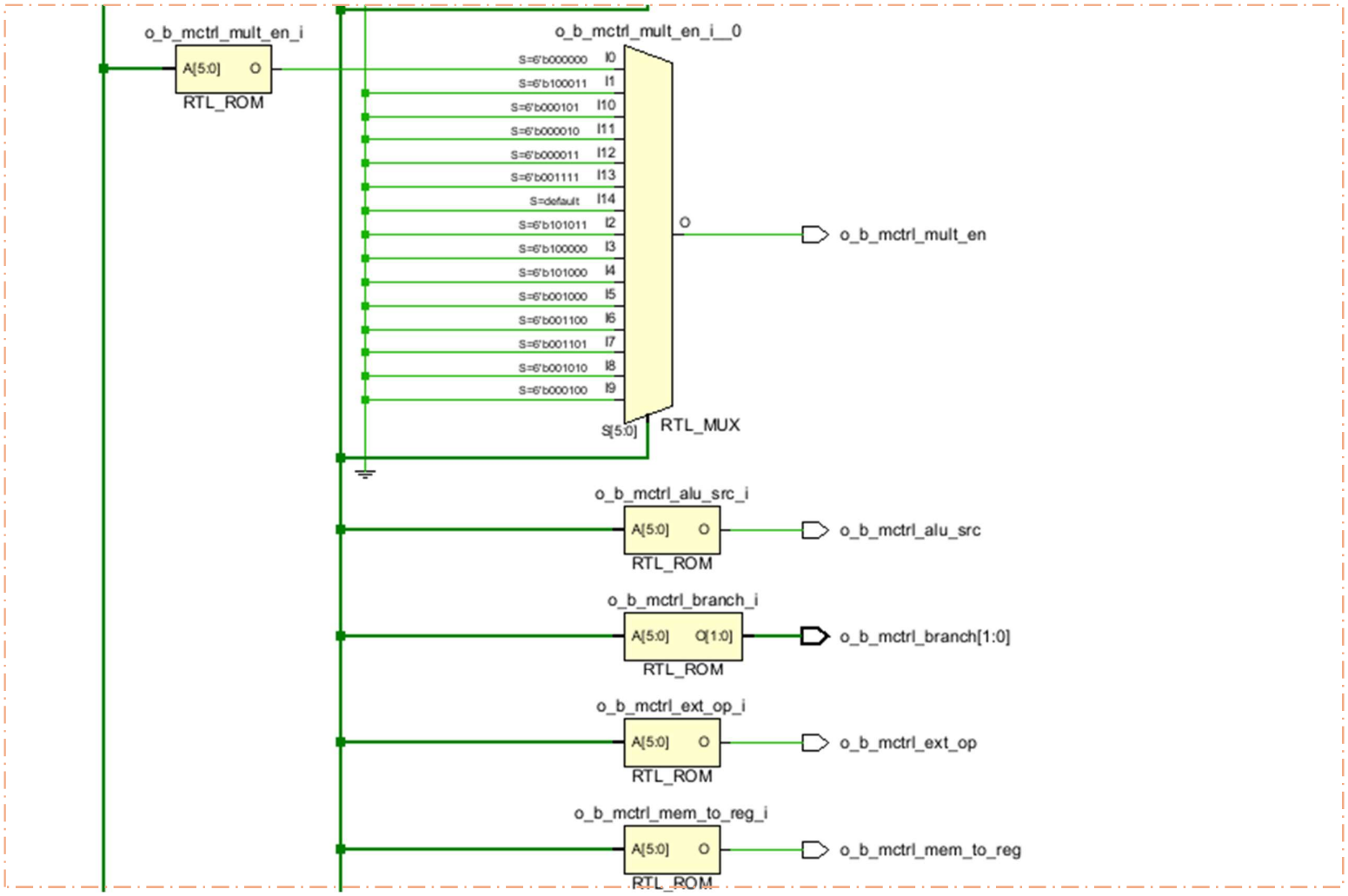


Diagram 4.3.1.5.4 Part 3 of post-synthesis Schematic Diagram of Main control block

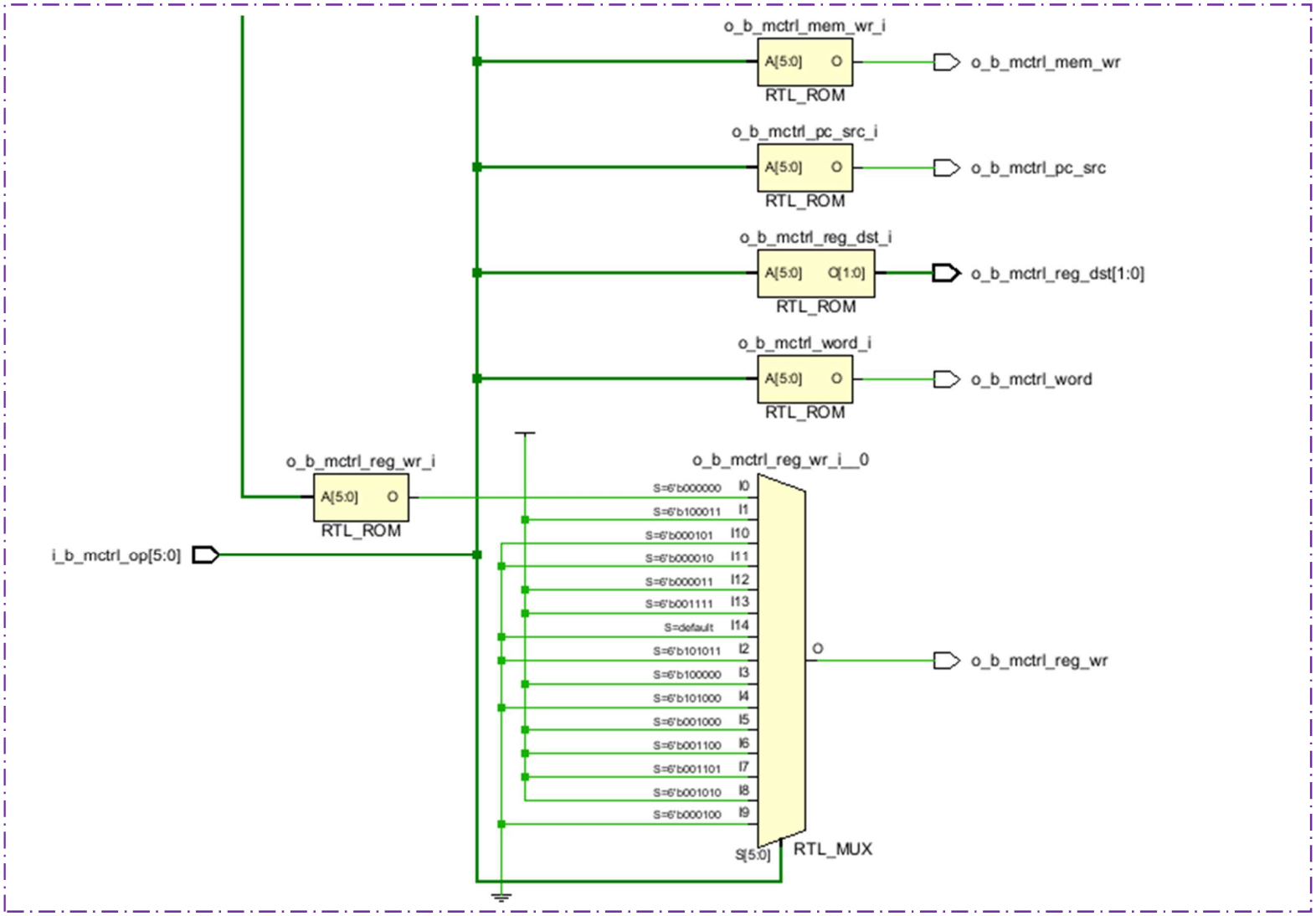


Diagram 4.3.1.5.5 Part 4 of post-synthesis Schematic Diagram of Main control block

4.3.2 ALU Control Block

4.3.2.1 Functionality/Feature

- Decode ALU opcode and instruction function into precise ALU/shifter selects control code.

4.3.2.2 Block interface and I/O pin description

4.3.2.2.1 ALU control block interface

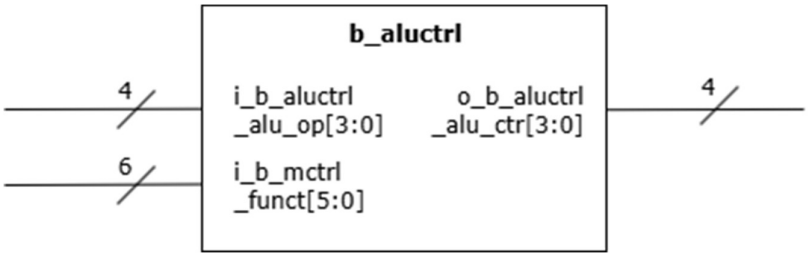


Diagram 4.3.2.2.1: Block interface of ALU control block

4.3.2.2.2 I/O pin description

Pin name: **Pin class:** **Pin function:**	i_b_aluctrl_alu_op[3:0] control To receive ALU operation code from Main Control	**Source →Destination:**	Main Control → ALU Control
Pin name: **Pin class:** **Pin function:**	i_b_aluctrl_funct[5:0] control To receive R-type function field for detailed ALU decode	**Source →Destination:**	Instruction [5:0] → ALU Control
Pin name: **Pin class:** **Pin function:**	o_b_aluctrl_alu_ctr[3:0] control To select specific ALU/shifter operation	**Source →Destination:**	ALU Control → ALU

Table 4.3.2.2.2: I/O pin description of ALU Control Block

4.3.2.3 Internal Operation: Function Table

Input				Output	
Instruction	**Instr Type**	**i_b_aluctrl_alu_op [3:0]**	**i_b_aluctrl_funct [5:0]**	**Required ALU Action**	**o_b_aluctrl_alu_ctr[3:0]**
add	R-type	1xxx	100000	Add	0000
sub	R-type	1xxx	100010	Subtract	0010
and	R-type	1xxx	100100	Bitwise AND	0100
or	R-type	1xxx	100101	Bitwise OR	0101
xor	R-type	1xxx	100110	Bitwise XOR	0110
nor	R-type	1xxx	100111	Bitwise NOR	0111
slt	R-type	1xxx	101010	Set on Less Than	1010
sll	R-type	1xxx	000000	Shift Left Logical	0001
srl	R-type	1xxx	000010	Shift Right Logical	0011
multu	R-type	1xxx	011001	x	xxxx
mfhi	R-type	0000	xxxxxx	Add	0000
mflo	R-type	0000	xxxxxx	Add	0000
jr	R-type	1xxx	001000	x	xxxx
jalr	R-type	1xxx	001001	x	xxxx
lw	I-type	0000	xxxxxx	Add (address)	0000
sw	I-type	0000	xxxxxx	Add (address)	0000
lb	I-type	0000	xxxxxx	Add (address)	0000
sb	I-type	0000	xxxxxx	Add (address)	0000
addi	I-type	0000	xxxxxx	Add	0000
andi	I-type	0100	xxxxxx	Bitwise AND	0100
ori	I-type	0101	xxxxxx	Bitwise OR	0101
slti	I-type	0001	xxxxxx	Set on Less Than	1010
beq	I-type	0010	xxxxxx	Subtract	0010
bne	I-type	0010	xxxxxx	Subtract	0010
j	J-type	xxxx	xxxxxx	x	xxxx
jal	J-type	xxxx	xxxxxx	x	xxxx
lui	I-type	0111	xxxxxx	Shift left 16 bit	1111

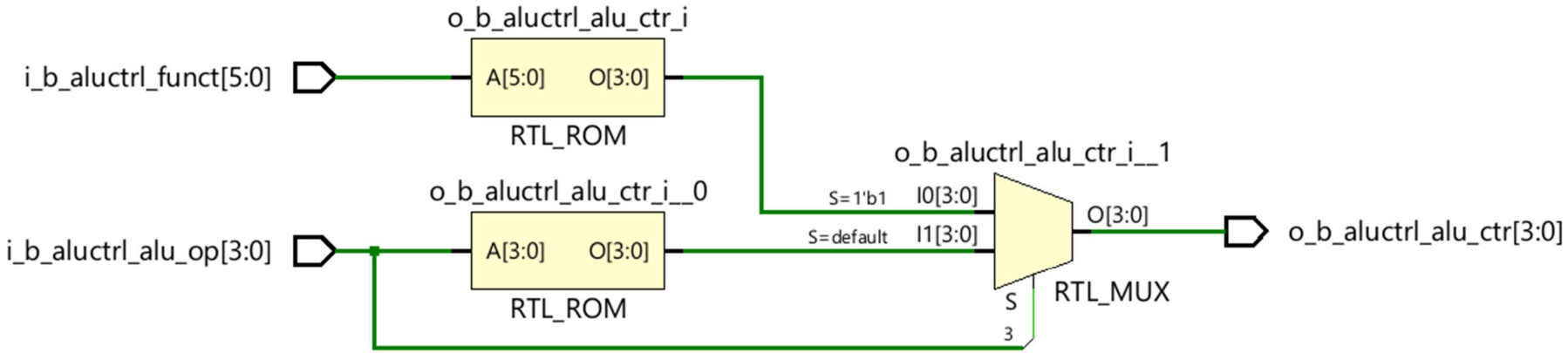
Table 4.3.1.3: Function table of ALU control block

4.3.2.4 System Verilog Model

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Ng Yu Heng
//
// Create Date: 31.08.2025 00:41:40
// File Name: b_aluctrl.sv
// Module Name: b_aluctrl
// Project Name: MIPS ISA Pipeline processor
// Code Type: RTL level
// Description: Modeling of ALU control block
//
//////////////////////////////////////////////////////////////////////////////////

module b_aluctrl(
    // Inputs
    input  logic [3:0] i_b_aluctrl_alu_op,    // ALU operation code from Main Control
    input  logic [5:0] i_b_aluctrl_funct,     // R-type function field

    // Outputs
```

```
    output logic [3:0] o_b_aluctrl_alu_ctr    // Specific ALU/shifter operation
);

always_comb begin
        // Default output is xxxx for instructions that don't need ALU
        o_b_aluctrl_alu_ctr = 4'bxxxx;

        // R-type instructions (when ALU op starts with 1)
        if (i_b_aluctrl_alu_op[3] == 1'b1) begin
            case (i_b_aluctrl_funct)
                6'b100000: o_b_aluctrl_alu_ctr = 4'b0000; // add
                6'b100010: o_b_aluctrl_alu_ctr = 4'b0010; // sub
                6'b100100: o_b_aluctrl_alu_ctr = 4'b0100; // and
                6'b100101: o_b_aluctrl_alu_ctr = 4'b0101; // or
                6'b100110: o_b_aluctrl_alu_ctr = 4'b0110; // xor
                6'b100111: o_b_aluctrl_alu_ctr = 4'b0111; // nor
                6'b101010: o_b_aluctrl_alu_ctr = 4'b1010; // slt
                6'b000000: o_b_aluctrl_alu_ctr = 4'b0001; // sll
                6'b000010: o_b_aluctrl_alu_ctr = 4'b0011; // srl
                // multu, jr, jalr remain xxxx (default)
            endcase
        end
        // I-type and other instructions
        else begin
            case (i_b_aluctrl_alu_op)
                4'b0000: o_b_aluctrl_alu_ctr = 4'b0000; // lw, sw, lb, sb, addi, mfhi, mflo
                4'b0100: o_b_aluctrl_alu_ctr = 4'b0100; // andi
                4'b0101: o_b_aluctrl_alu_ctr = 4'b0101; // ori
                4'b0001: o_b_aluctrl_alu_ctr = 4'b1010; // slti
                4'b0010: o_b_aluctrl_alu_ctr = 4'b0010; // beq, bne
                4'b0111: o_b_aluctrl_alu_ctr = 4'b1111; // lui
                // default remains xxxx for j, jal
            endcase
        end
    end

endmodule
```

4.3.2.5 Post-synthesis Schematic Diagram



4.3.2.5 Post-synthesis Schematic Diagram of ALU control block

4.3.2.6 Test Plan

No.	Test Case	Description of Test Vector Generation	Expected Output	Status
1	R-type: add	1. Set i_b_aluctrl_alu_op = 1xxx. 2. Set i_b_aluctrl_funct = 100000.	o_b_aluctrl_alu_ctrl[3:0]=0000	PASS
2	R-type: sub	1. Set i_b_aluctrl_alu_op = 1xxx. 2. Set i_b_aluctrl_funct = 100010.	o_b_aluctrl_alu_ctrl[3:0]=0010	PASS
3	R-type: and	1. Set i_b_aluctrl_alu_op = 1xxx. 2. Set i_b_aluctrl_funct = 100100.	o_b_aluctrl_alu_ctrl[3:0]=0100	PASS
4	R-type: or	1. Set i_b_aluctrl_alu_op = 1xxx. 2. Set i_b_aluctrl_funct = 100101.	o_b_aluctrl_alu_ctrl[3:0]=0101	PASS
5	R-type: xor	1. Set i_b_aluctrl_alu_op = 1xxx. 2. Set i_b_aluctrl_funct = 100110.	o_b_aluctrl_alu_ctrl[3:0]=0110	PASS
6	R-type: nor	1. Set i_b_aluctrl_alu_op = 1xxx. 2. Set i_b_aluctrl_funct = 100111.	o_b_aluctrl_alu_ctrl[3:0]=0111	PASS
7	R-type: slt	1. Set i_b_aluctrl_alu_op = 1xxx. 2. Set i_b_aluctrl_funct = 101010.	o_b_aluctrl_alu_ctrl[3:0]=1010	PASS
8	R-type: sll	1. Set i_b_aluctrl_alu_op = 1xxx. 2. Set i_b_aluctrl_funct = 000000.	o_b_aluctrl_alu_ctrl[3:0]=0001	PASS
9	R-type: srl	1. Set i_b_aluctrl_alu_op = 1xxx. 2. Set i_b_aluctrl_funct = 000010.	o_b_aluctrl_alu_ctrl[3:0]=0011	PASS
10	R-type: mfhi, mflo	1. Set i_b_aluctrl_alu_op = 0000 . 2. Set i_b_aluctrl_funct = xxxxxx.	o_b_aluctrl_alu_ctrl[3:0]=0000	PASS
11	I-type: lw, sw, lb, sb, addi	1. Set i_b_aluctrl_alu_op = 0000.	o_b_aluctrl_alu_ctrl[3:0]=0000	PASS

		2. Set i_b_aluctrl_funct = xxxxxx		
12	I-type: andi	1. Set i_b_aluctrl_alu_op = 0100. 2. Set i_b_aluctrl_funct = xxxxxx	o_b_aluctrl_alu_ctrl[3:0]=0100	PASS
13	I-type: ori	1. Set i_b_aluctrl_alu_op = 0101. 2. Set i_b_aluctrl_funct = xxxxxx	o_b_aluctrl_alu_ctrl[3:0]=0101	PASS
14	I-type: slti	1. Set i_b_aluctrl_alu_op = 0001. 2. Set i_b_aluctrl_funct = xxxxxx	o_b_aluctrl_alu_ctrl[3:0]=1010	PASS
15	beq, bne,	1. Set i_b_aluctrl_alu_op = 0010 2. Set i_b_aluctrl_funct = xxxxxx	o_b_aluctrl_alu_ctrl[3:0]=0010	PASS
16	jr	1. Set i_b_aluctrl_alu_op = 1xxx. 2. Set i_b_aluctrl_funct = 001000.	o_b_aluctrl_alu_ctrl[3:0]=xxxx	PASS
17	jalr	1. Set i_b_aluctrl_alu_op = 1xxx. 2. Set i_b_aluctrl_funct = 001000.	o_b_aluctrl_alu_ctrl[3:0]=xxxx	PASS
18	multu	1. Set i_b_aluctrl_alu_op = 1xxx 2. Set i_b_aluctrl_funct = 011001	o_b_aluctrl_alu_ctrl[3:0]=xxxx	PASS
19	j, jal	1. Set i_b_aluctrl_alu_op = xxxx. 2. Set i_b_aluctrl_funct = xxxxxx.	o_b_aluctrl_alu_ctrl[3:0]=xxxx	PASS
20	lui	1. Set i_b_aluctrl_alu_op = 0111. 2. Set i_b_aluctrl_funct = xxxxxx.	o_b_aluctrl_alu_ctrl[3:0]=1111	PASS

4.3.2.6 Test Plan of ALU control block

4.3.2.7 Testbench and Simulation results

4.3.2.7.1 Testbench

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Author: Ng Yu Heng
//
// Create Date: 31.08.2025 00:49:43
// File Name: tb_b_aluctrl.sv
// Module Name: tb_b_aluctrl
// Project Name: MIPS ISA Pipeline processor
// Code Type: Behavioural
// Description: Testbench for ALU control block
//
//////////////////////////////////////////////////////////////////////////////////

module tb_b_aluctrl();
    // Inputs
    logic [3:0] tb_i_b_aluctrl_alu_op;
    logic [5:0] tb_i_b_aluctrl_funct;

    // Outputs
    logic [3:0] tb_o_b_aluctrl_alu_ctrl;

    // Clock and cycle definition
    parameter CC = 10; // 10ns per clock cycle
    logic clk;

    // Instantiate DUT
    b_aluctrl dut (
        .i_b_aluctrl_alu_op(tb_i_b_aluctrl_alu_op),
        .i_b_aluctrl_funct(tb_i_b_aluctrl_funct),
        .o_b_aluctrl_alu_ctr(tb_o_b_aluctrl_alu_ctrl)
    );

    // Clock generation
    always #(CC/2) clk = ~clk;

    initial begin
        // Initialize
        clk = 0;
        tb_i_b_aluctrl_alu_op = 4'b0;
        tb_i_b_aluctrl_funct = 6'b0;

        // Wait 2 cycles
        repeat(2) @(negedge clk);

        //----------------------------------------
        // Test Case 1: R-type add
        //----------------------------------------
        tb_i_b_aluctrl_alu_op = 4'b1xxx;
        tb_i_b_aluctrl_funct = 6'b100000;
        repeat(2) @(negedge clk);

        //----------------------------------------
        // Test Case 2: R-type sub
        //----------------------------------------
```

```
tb_i_b_aluctrl_alu_op = 4'b1xxx;
tb_i_b_aluctrl_funct = 6'b100010;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 3: R-type and
//----------------------------------------
tb_i_b_aluctrl_alu_op = 4'b1xxx;
tb_i_b_aluctrl_funct = 6'b100100;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 4: R-type or
//----------------------------------------
tb_i_b_aluctrl_alu_op = 4'b1xxx;
tb_i_b_aluctrl_funct = 6'b100101;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 5: R-type xor
//----------------------------------------
tb_i_b_aluctrl_alu_op = 4'b1xxx;
tb_i_b_aluctrl_funct = 6'b100110;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 6: R-type nor
//----------------------------------------
tb_i_b_aluctrl_alu_op = 4'b1xxx;
tb_i_b_aluctrl_funct = 6'b100111;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 7: R-type slt
//----------------------------------------
tb_i_b_aluctrl_alu_op = 4'b1xxx;
tb_i_b_aluctrl_funct = 6'b101010;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 8: R-type sll
//----------------------------------------
tb_i_b_aluctrl_alu_op = 4'b1xxx;
tb_i_b_aluctrl_funct = 6'b000000;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 9: R-type srl
//----------------------------------------
tb_i_b_aluctrl_alu_op = 4'b1xxx;
tb_i_b_aluctrl_funct = 6'b000010;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 10: R-type mfhi, mflo
//----------------------------------------
tb_i_b_aluctrl_alu_op = 4'b0000;
```

```
tb_i_b_aluctrl_funct = 6'bx;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 11: I-type lw, sw, lb, sb, addi
//----------------------------------------
tb_i_b_aluctrl_alu_op = 4'b0000;
tb_i_b_aluctrl_funct = 6'bx;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 12: I-type andi
//----------------------------------------
tb_i_b_aluctrl_alu_op = 4'b0100;
tb_i_b_aluctrl_funct = 6'bx;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 13: I-type ori
//----------------------------------------
tb_i_b_aluctrl_alu_op = 4'b0101;
tb_i_b_aluctrl_funct = 6'bx;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 14: I-type slti
//----------------------------------------
tb_i_b_aluctrl_alu_op = 4'b0001;
tb_i_b_aluctrl_funct = 6'bx;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 15: beq, bne
//----------------------------------------
tb_i_b_aluctrl_alu_op = 4'b0010;
tb_i_b_aluctrl_funct = 6'bx;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 16: jr
//----------------------------------------
tb_i_b_aluctrl_alu_op = 4'b1xxx;
tb_i_b_aluctrl_funct = 6'b001000;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 17: jalr
//----------------------------------------
tb_i_b_aluctrl_alu_op = 4'b1xxx;
tb_i_b_aluctrl_funct = 6'b001001;
repeat(2) @(negedge clk);

//----------------------------------------
// Test Case 18: multu
//----------------------------------------
tb_i_b_aluctrl_alu_op = 4'b1xxx;
tb_i_b_aluctrl_funct = 6'b011001;
```

```
        repeat(2) @(negedge clk);

        //----------------------------------------
        // Test Case 19: j
        //----------------------------------------
        tb_i_b_aluctrl_alu_op = 4'bx;
        tb_i_b_aluctrl_funct = 6'bx;
        repeat(2) @(negedge clk);

        //----------------------------------------
        // Test Case 20: lui
        //----------------------------------------
        tb_i_b_aluctrl_alu_op = 4'b0111;
        tb_i_b_aluctrl_funct = 6'bx;
        repeat(2) @(negedge clk);


        // Finish simulation
        repeat(2) @(negedge clk);
        $finish;
    end

endmodule
```

4.3.2.7.2 Simulation result

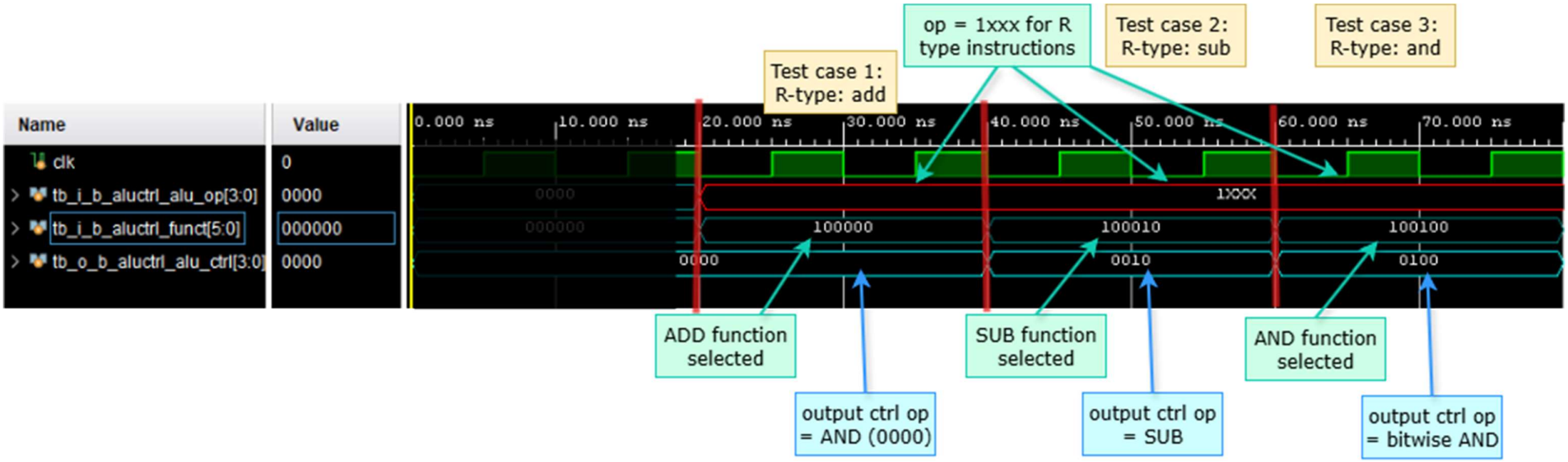


Diagram 4.3.2.7.2.1 Test case 1, 2, 3

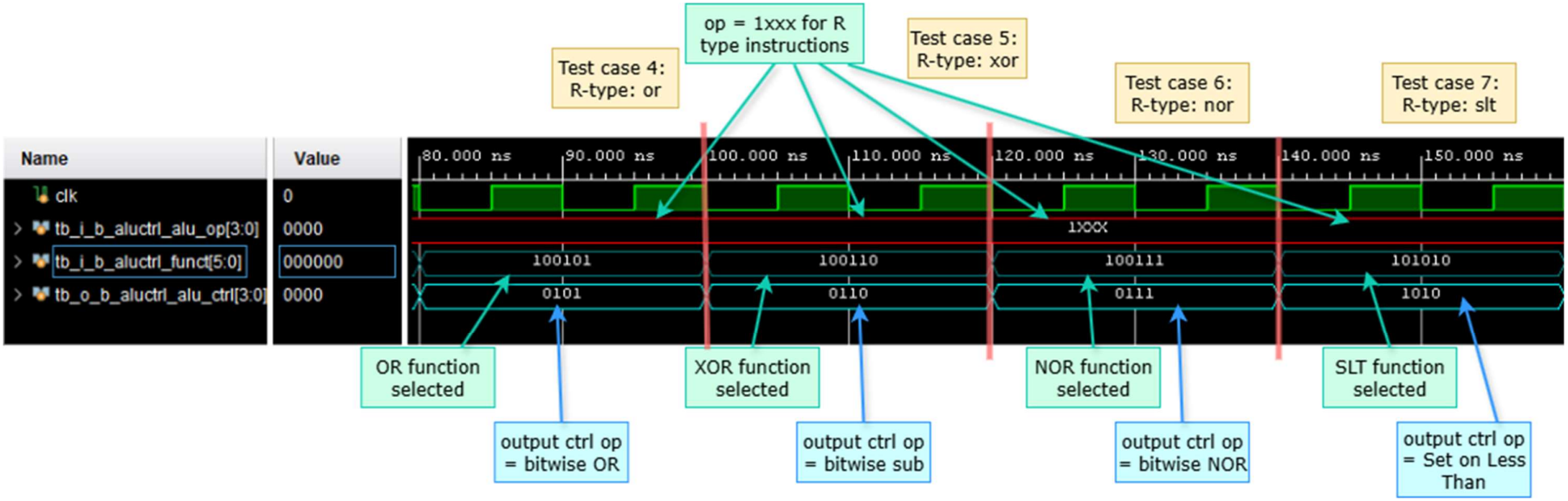


Diagram 4.3.2.7.2.2 Test case 4, 5, 6, 7

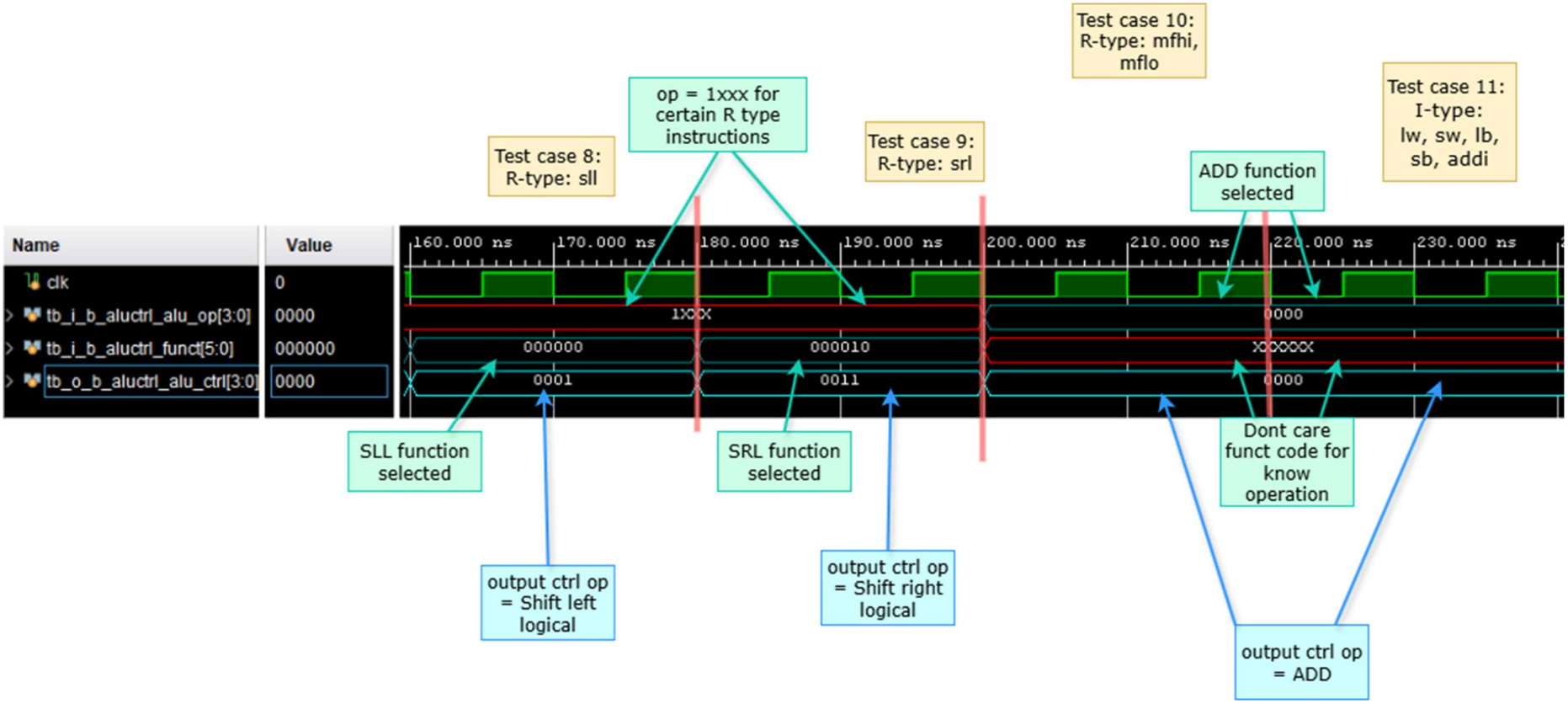


Diagram 4.3.2.7.2.3 Test case 8, 9, 10, 11

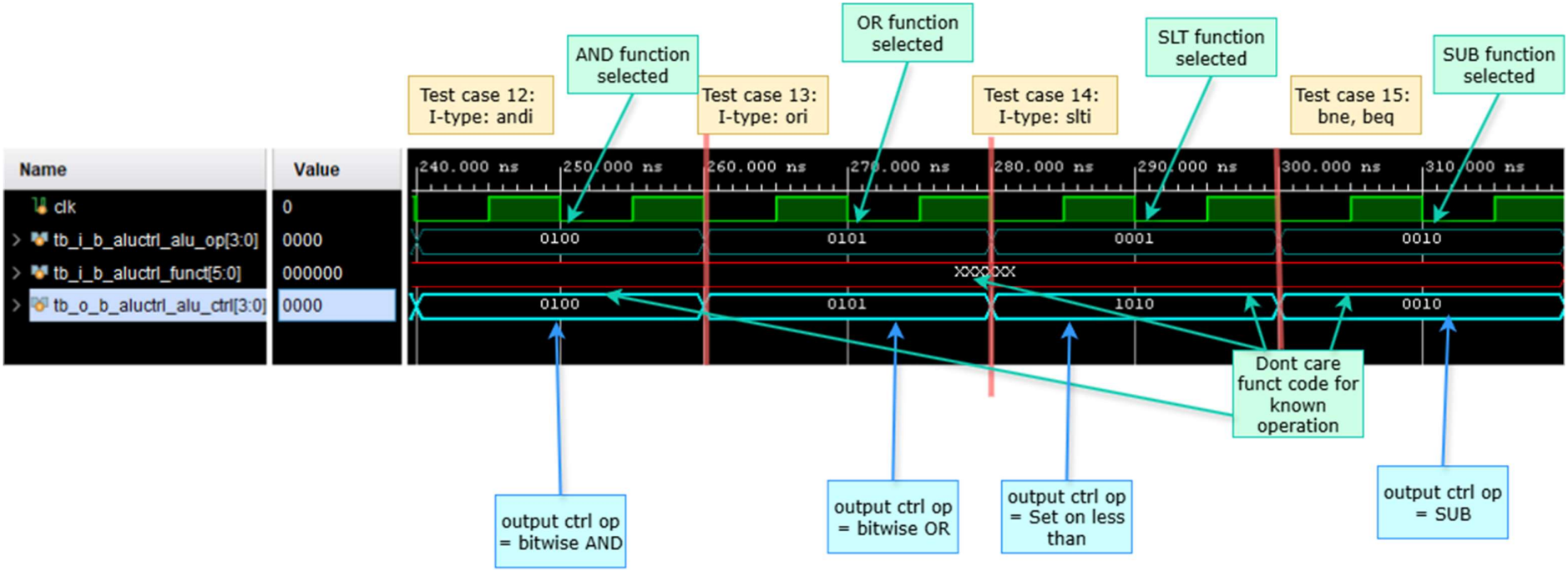


Diagram 4.3.2.7.2.4 Test case 12, 13, 14, 15

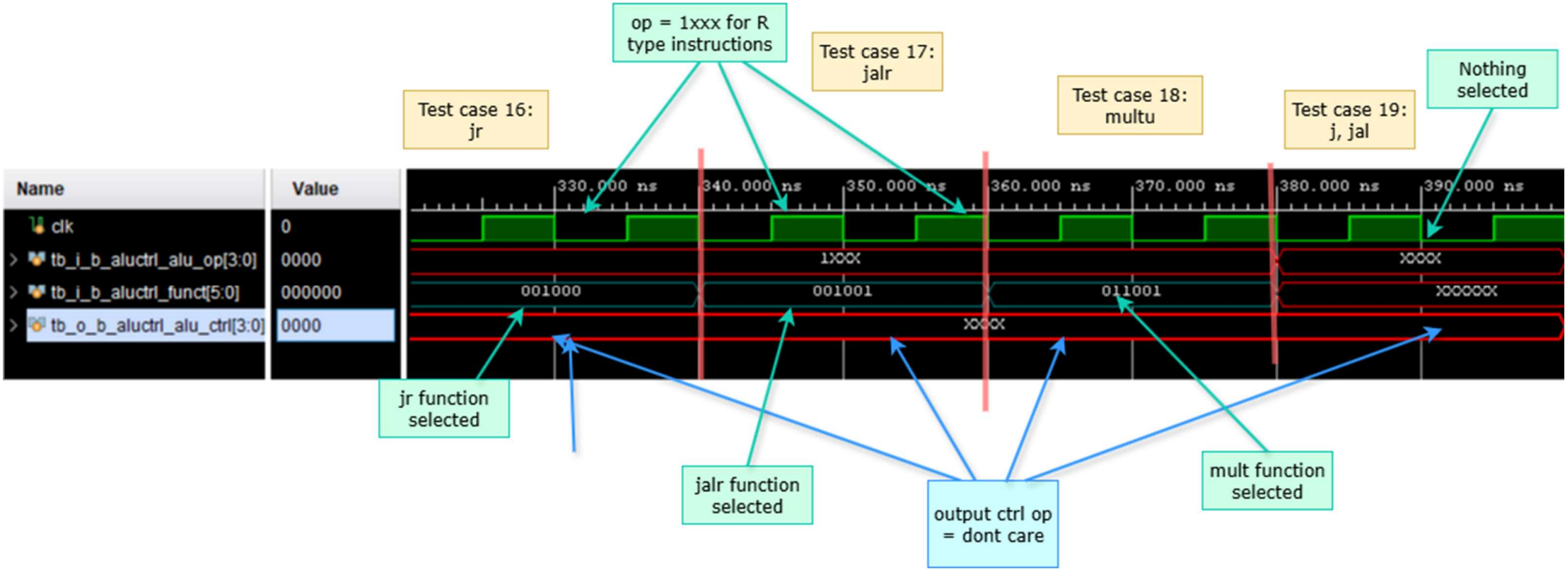


Diagram 4.3.2.7.2.5 Test case 16, 17 ,18, 19

Diagram 4.3.2.7.2.6 Test case 20, 21, 22