A logo of a company

Description automatically generated

**California Polytechnic State University, Pomona**

Department of Electrical and Computer Engineering

**MIPS Code**

Computer Architecture: ECE 4300

December 12, 2024

Professor Beverly Quon

Ethan McKendell

**MIPS Code**

**Code and Screenshots:** <https://github.com/emmckendell/MIPS_Processor>

**Instruction Fetch (IF) stage:**

The Instruction Fetch is the first stage of the MIPS pipeline where its purpose is to fetch the next instruction one of the 127 32-bit instruction from instruction memory based on the address within the program counter. If a r-type, load (lw), or store (sw) instruction was executed then the next instruction is current program count + 1 for a word addressable. However, if a branch not equal (beq) instruction and the ALU result = 0, then any address from 0 to 127 can be jumped to. If the program counter is above 127 then instructions will stop executing until its looped back to program count 1024.

Components: program counter, instruction memory, 32-bit mux, incrementor (ALU: b = 1 and ALUControl = add), and IF/ID latch.

Inputs: clock, reset, PCSrc (MEM stage) and next program count (MEM stage)

Outputs: next program count (ID stage) and instruction (ID stage)

A diagram of a computer component

Description automatically generated

**A diagram of a computer component

Description automatically generatedProgram Counter:**

The program counter contains the current address

Inputs: clock, reset, next program counter

Output: program count

A diagram of instructions and address

Description automatically generated**Instruction Memory:**

The instruction memory contains 32-bit of

127 instructions (word addressable)

Inputs: clock, reset, address  
Outputs: data (instruction)

**32-bit mux (IF stage):**

A diagram of a cylinder with arrows and letters

Description automatically generatedA 32-bit mux (IF stage) that determines the next

program count depending on PCSrc (sel)

If PCSrc = 0 then y = b (next program count + 1)  
If PCSrc = 1 then y = a (next program count (MEM stage))

Inputs: a, b

Control Input: sel  
Output: y

A diagram of a diagram

Description automatically generated**Incrementor:**

The incrementor that outputs next program count + 1  
if PCSrc = 0, can be implemented with an ALU by tying values

Inputs: program count, 1 (optional)

Control Input: add (optional)  
Output: program count + 1

A diagram of a long rectangular object with arrows

Description automatically generated**IF/ID latch:**  
The IF/ID stage latch that contains next program count

and instruction for ID stage

Inputs: clock, reset, next program count, instruction

Outputs: next program count out, instruction out

**Instruction Decode (ID) stage:**

The Instruction Fetch is the second stage of the MIPS pipeline where its purpose is to decode the 32-bit instruction from 5-bit opcode (instruction[31:26]) to set control bits. Contains 32 32-bit registers (register 0 ($zero) is locked to value 0). Read two registers: register source (rs) (instruction[25:21]) and register target (rt) (instruction[20:16]). Writes data determined in WB stage if RegWrite = 1 to a register determined from WB stage. Sign extends (instruction[15:0]) with the MSB to 32-bits.

Components: control, register, sign extend, ID/EX latch

Inputs: clock, reset, next program count (IF stage), and instruction (IF stage)

Control outputs: write back control, memory control, reg destination, ALU operation, ALU source

Outputs: next program count (EX stage), register source (rs) data, register target (rt) data, instruction sign extend, instruction[20:16], and instruction[15:11]

A diagram of a computer

Description automatically generated

**A diagram of a sign

Description automatically generatedSign Extend:**The sign extend is combinational logic that extends

the MSB (instruction[15]) of (instruction[15:0]) to 32 bits

Input: sign unextended (instruction[15:0])

Output: sign extended

**A diagram of a control system

Description automatically generatedControl:**

The control unit is combinational logic that takes 5-bit

opcode (instruction[31:26]) to set 9 control bits

Input: opcode (instruction[31:26])

Output: {control, memory, and execute} control

A diagram of a computer program

Description automatically generated with medium confidenceOpcodes:  
R-format = 0

lw (load) = 35  
sw (store) = 43

beq = 4  
nop = 32 (control bits = 0)

default = nop

A diagram of a rectangular object with text

Description automatically generated

**Registers:**The registers contain 32 registers of 32-bits of data,

however, register0 ($zero) is locked to value 0, meaning   
it is not possible to overwrite register 0.

Reads: register source (rs) and register (rt)

Write: register destination (rd) with 32-bit of data if regwrite = 1

Inputs: clock, reset, rs, rt, rd, write data

Control: reg write

Outputs: rs data, rt data

**A diagram of a computer

Description automatically generated**

**ID/EX latch:**The ID/EX stage latch that contains control bits, next program count,

two registers data, sign extend, and part of instruction for EX stage

Inputs: clock, reset, {write back, memory, execute} control bits, next

program count, {rs, rt} data, sign unextend, {instr} [20:16], and [15:11]

Outputs: {write back, memory, execute} control bits out, next

program count out, {rs, rt} data out, sign unextend out,

{instr} [20:16], and [15:11] out

**Execute (EX) stage:**

The Execute is the third stage of the MIPS pipeline where its purpose is to execute the instruction that determines the next program with offset for beq instructions, from an ALU result determines the data to write to either to data memory or registers unless beq instruction is executed, and if r-format or lw instruction is executed then a register destination (rd) would be selected through a mux. Four control bits are used within the stage with register destination (control[3]), ALU operation (control[1:0]), and ALU source (control[0]).

Components: ALU, ALU (adder), 32-bit mux, 5-bit mux, ALU control, EX/MEM latch

Inputs: clock, reset, next program count (EX stage), register source (rs), register target (rt), instruction sign extend, instruction[20:16], and instruction[15:11]

Inputs control: write back control, memory control, reg destination (execute[3]), ALU operation (execute[2:1]), ALU source (execute[0])

Outputs: ALU add result (next program count with offset (MEM stage)), ALU result (address for data memory/data for registers), register target (write data for data memory), and write register

Outputs control: write back control, memory read, memory write, memory branch, zero flag (ALU result = 0) for beq instruction

A diagram of a component

Description automatically generated

A diagram of a cylinder with arrows and letters

Description automatically generated**32-bit mux (EX stage):**

A 32-bit mux (EX stage) that determines operand b for ALU

depending on ALUSrc (sel)

If ALUSrc = 0 then y = b (register target(rt))  
If ALUSrc = 1 then y = a (sign extend)

Inputs: a, b

Control Input: sel  
Output: y

A diagram of a cylinder with arrows and letters

Description automatically generated**5-bit mux (EX stage):**

A 5-bit mux (EX stage) that determines operand b for ALU

depending on ALUSrc (sel)

If ALUSrc = 0 then y = b (register target(rt))  
If ALUSrc = 1 then y = a (sign extend)

Inputs: a, b

Control Input: sel  
Output: y

A diagram of a diagram of a person's control

Description automatically generated with medium confidence

**ALU Control:**

The ALU control is combinational logic that determines

the ALU operation first checks ALU opcode (execute[2:1])

for lw/sw and beq then funct (instruction[5:0]) for r-type

Input: funct (instruction[5:0])

Input control: ALU opcode (execute[2:1])

Output control: ALU control bits

A diagram of a program

Description automatically generated with medium confidenceALU control:  
lw/sw = add

beq = sub

r-type (add) = add

r-type (sub) = sub  
r-type (and) = and

r-type (or) = or

r-type (slt) = slt

**32-bit ALU:**The arithmetic logic unit (ALU) performs arithmetic

and bitwise operations (addition, subtraction, bitwise AND,

**A diagram of a diagram

Description automatically generated**bitwise OR, and set less than) based on ALU control bits

with two operands that outputs a result with zero flag as

a signal bit for ALU result = 0.

The result for this ALU is the address for data memory or

data for a register destination

Inputs: operand A, operand B

Input control: ALU control bits

Output: result

Output control: zero flag

If ALU control = 3’b010, then result = A + B

If ALU control = 3’b110, then result = A – B

If ALU control = 3’b000, then result = A & B

If ALU control = 3’b001, then result = A | B

If ALU control = 3’b111, then result = (A < B)? 1 for TRUE : 0 for FALSE, (1 bit)

A diagram of a mathematical equation

Description automatically generated**32-bit ALU (Adder):**The arithmetic logic unit (ALU) adder is typed to ALU control

of addition with unused zero flag

The result for this ALU adder is the next program counter  
with an offset for beq instructions if PCSrc = 1

Inputs: operand A, operand B

Input control: ALU control bits (typed to add)

Output: result

ALU control = 3’b010, result = A + B

A diagram of a computer program

Description automatically generated with medium confidence**EX/MEM latch:**The EX/MEM stage latch that contains control bits, ALU (adder) result

ALU result, zero flag, rt data, write register for MEM stage

Inputs: clock, reset, {write back, memory} control bits, add result,

zero, ALU result, rt data, and write register

Outputs: {write back, memory} control bits out, add result out,

zero out, ALU result out, rt data out, and write register out

**Memory (MEM) stage:**

The Execute is the fourth stage of the MIPS pipeline where its purpose is to store/load to the data memory that contains 256 data memory of 32-bits of data (256 \* 32-bits = 1kB) for store (sw) (MemRead = 1) and load (lw) (MemWrite = 1) instructions respectively. For r-type instructions will get pass through without reading or writing to data memory. For beq instruction will jump a new address for instruction memory if PCSrc = 1 when ALU result = 0. Three control bits are used within the stage with branch (memory[2]), memory read (memory[1]), and memory write (memory[0]).

Components: Branch (AND gate), data memory, and MEM/WB latch

Inputs: clock, reset, next program count (MEM stage), address data memory, register target (write data for data memory), and write register

Inputs control: write back control, branch (memory[2]), memory read (memory[1]), and memory write (memory[0]).

Outputs: next program count (used in IF stage), memory data, address data memory (write data for register destination), write register

Outputs control: PCSrc (used in IF stage), write back register write, write back memory to register

A diagram of a computer component

Description automatically generated

**Branch (AND gate):**The branch is an AND gate

A yellow and white square with arrows

Description automatically generated with medium confidencePCSrc = memory banch & zero flag

The output of the AND gate controls the next

program count and jumps to new address

instruction when a memory branch = 1, only

for beq instructions

Inputs control: a (memory branch), b (zero flag)

Output control: y (PCSrc)

A diagram of data processing

Description automatically generated

**Data memory:**

The data memory contain 256 registers of 32-bits of data (1kB).

Reads: memory data if MemRead = 1

Write: data to an memory address if MemWrite = 1

sw instructions store registers data to data memory

lw instructions loads data memory to a register

Inputs: clock, reset, memory address, and write memory data

Control: MemWrite and MemRead

Outputs: memory data

**MEM/WB latch:**The MEM/WB stage latch that contains control bits,

A diagram of a column with text and numbers

Description automatically generatedmemory data, ALU result (write register data), write register

for WB stage

Inputs: clock, reset, memory control bits, memory data,

ALU result (write register data), write register

Outputs: memory control bits out, memory data out,

ALU result out (write register data), write register out

**Write Back (WB) stage:**

The Write Back is the fifth and final stage of the MIPS pipeline where its purpose is to write back data for r-type instructions for ALU result of two registers or load (lw) instructions to load data memory from a memory address to a register. Two control bits are used within the stage with register write (write\_back[1]) and memory to register (write\_back[0]). No latch is used within the WB stage.

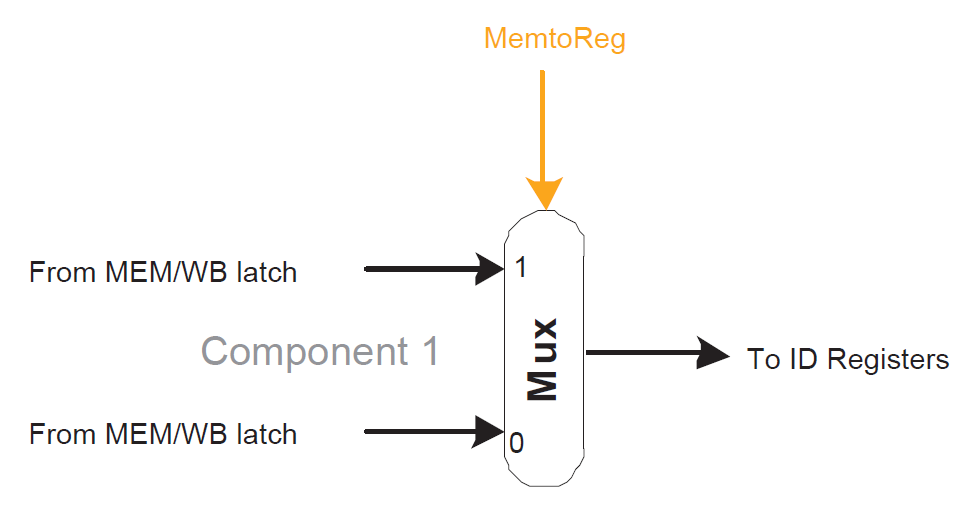
Components: 32-bit mux

Inputs: clk (unused), rst (unused), memory data, ALU result (write register data), write register

Inputs control: register write (write\_back[1]) and memory to register (write\_back[0])

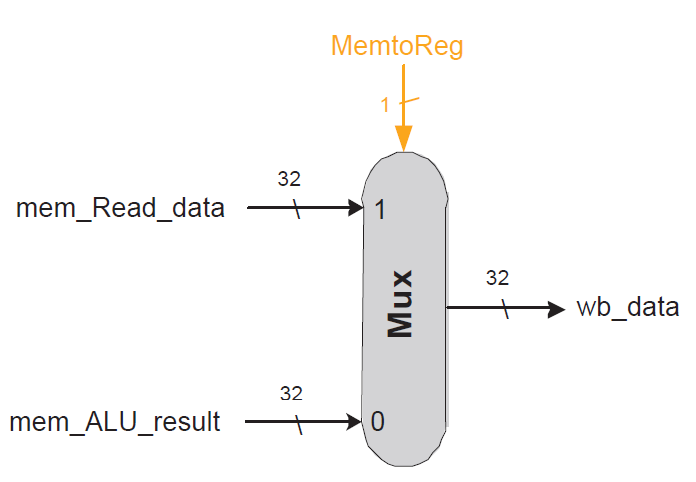
Outputs: write back register write (used in ID stage), write back memory to register (used in ID stage)

Outputs control: write register (write\_back[1]) (used in ID stage)



**32-bit mux (WB stage):**

A 32-bit mux (WB stage) that determines whether to

read the ALU result for r-type instructions or read memory

data for load (lw) instructions depending on MemToReg (sel)

If MemToReg = 0 then y = b (ALU result)  
If MemToReg = 1 then y = a (data memory)

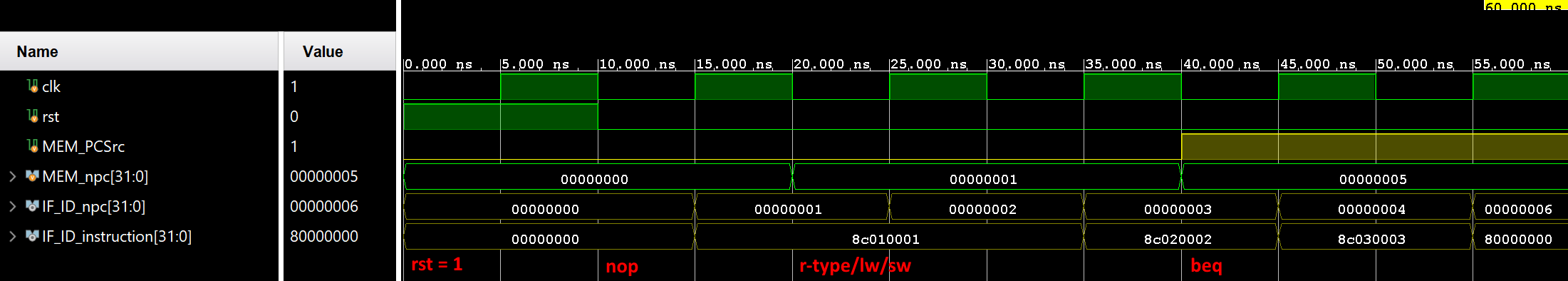
Inputs: a, b

Control Input: sel  
Output: y

**Screenshots:**

Key: control inputs, control outputs, inputs, outputs

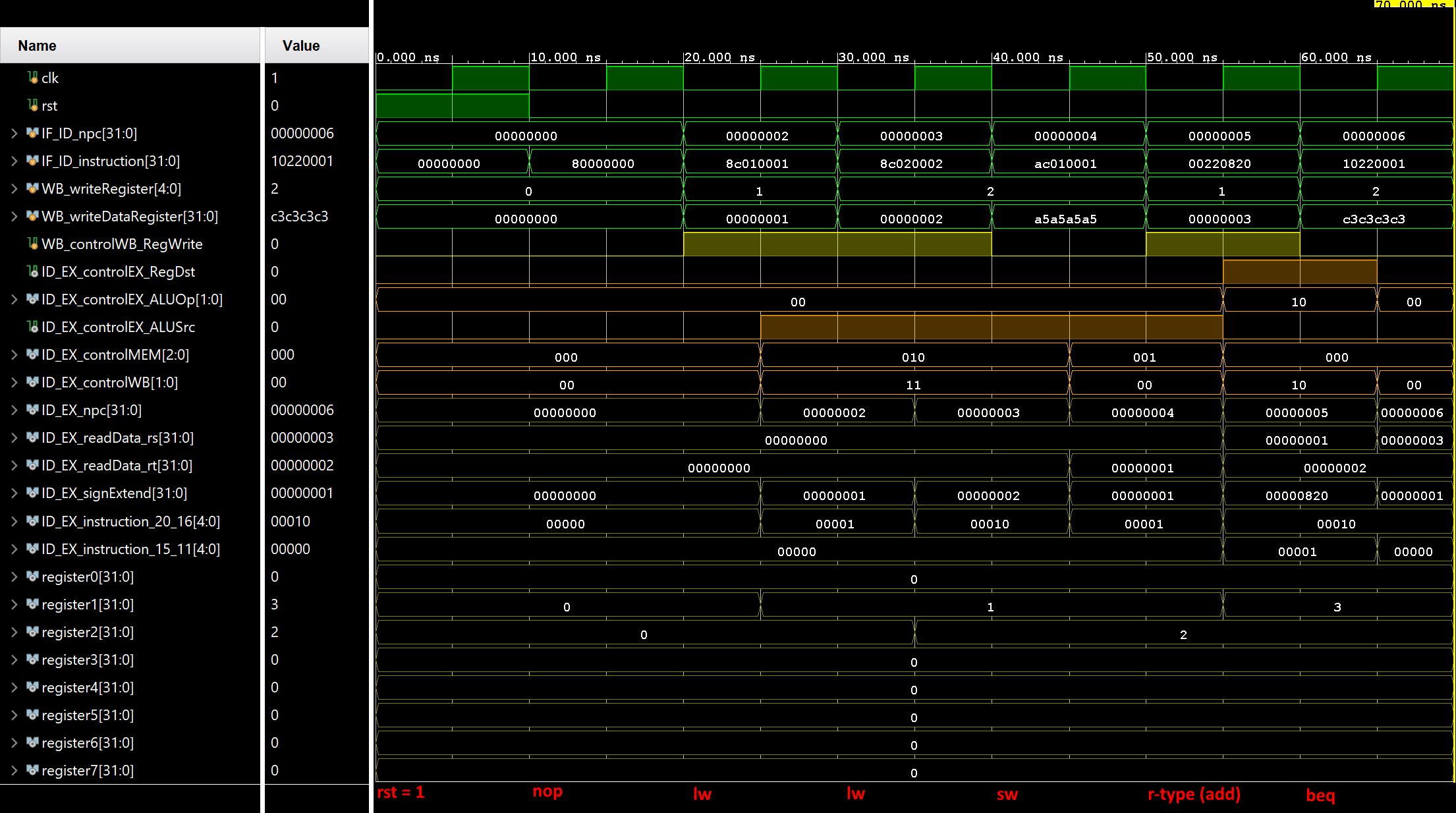
* Refer to final program of register1 = 12 and loaded values of data memory

****

INSTRUCTION\_FETCH\_tb\_wf.png

r-type/lw/sw: fetches next instruction in instruction memory

beq: jumps to address from MEM stage (next program count + instruction [15:0] sign extend)

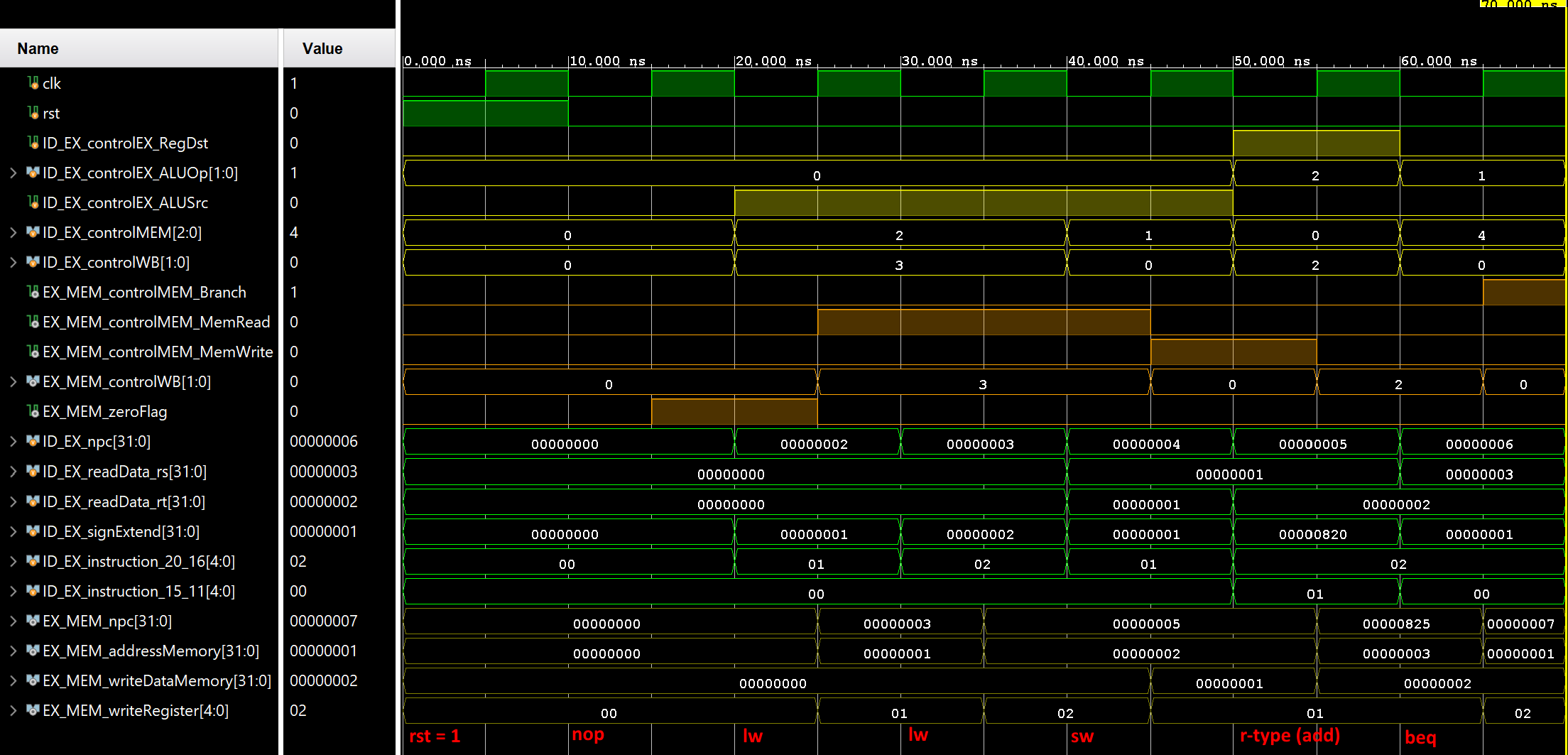


INSTRUCTION\_DECODE\_tb\_wf.png

r-type: writes data to a register. RegDst, ALUOp[1], RegWrite = 1

lw: loads data to register from data memory. ALUSrc, MemRead, RegWrite, MemToReg = 1  
sw: store data to data memory. ALUSrc, MemWrite = 1

beq: branches to address in instruction memory if ALU result = 0. ALUOp[0], Branch = 1

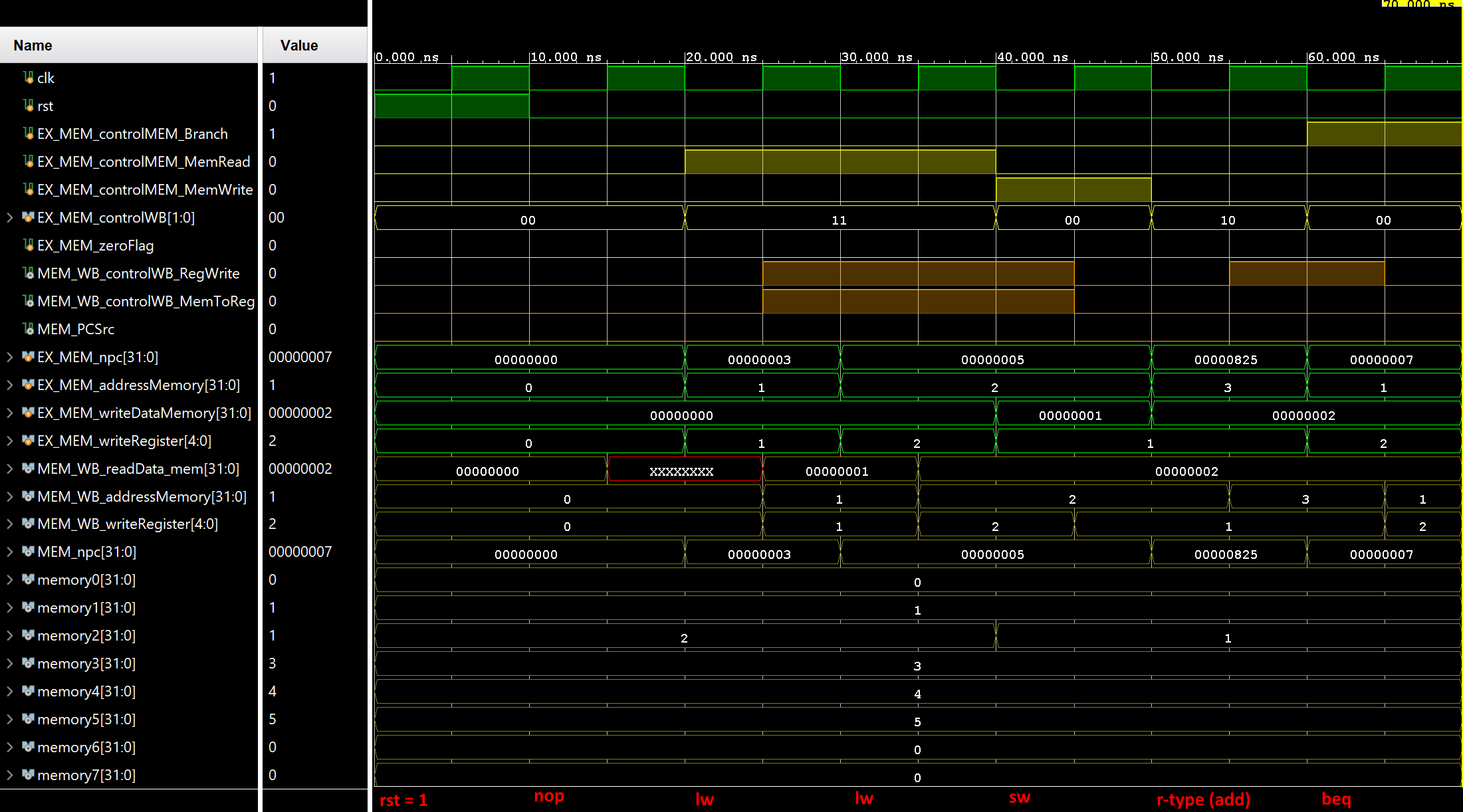


EXECUTE\_tb\_wf.png

r-type: does ALU operation of two registers according to instruction[6:0]

lw/sw: adds register source (rs) with instruction[15:0] sign extend for either address for data memory or data for register destination

beq: next program count jumps with an offset of instruction[15:0] sign extend if ALU result = 0



MEMORY\_tb\_wf.png

r-type: passes ALU result, data memory is not read or written from

lw: loads value to register target (rt) at address (register source + instruction [15:0] sign extend)

sw: stores of value of register target at address (register source + instruction [15:0] sign extend)

beq: next program count, if PCSrc = 1 then sets program counter to that

**A screenshot of a computer

Description automatically generated**

WRITE\_BACK\_tb\_wf.png

r-type: Reads from address memory (ALU result) to write to a register

lw: Reads from data from data memory to write to a register

sw / beq: Don’t care from reading data as register write is disable

Other screenshots of individual modules can be found on Github

**Conclusion / MIPS Pipeline:**

The MIPS pipeline can be created by first instantiating the Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Write Back (WB) stage with its own modules (components). For example: The IF stage has the 32-bit mux, incrementor, program counter, instruction memory, and IF/ID latch. The ID stage would not need the IF/ID stage, but, control unit, registers, sign extend, and ID/EX stage. So on and so forth for EX, MEM, and WB stages with the except of WB stage does not need a latch. Making modules are helpful as they become reuseable similar to function calls in software programming languages such the 32-bit mux that is used three times throughout the MIPS pipeline. These instantiations can be instantiated again as these stages become modules within themselves to have cleaner readability, otherwise known as abstraction, even if the stages are not used multiple times.

A diagram of a machine

Description automatically generated

The MIPS pipeline runs instructions that could either be r-type, load, store, branch not equal, and no operation instructions. The R-type (add, sub, and, etc.) instructions reads two values of registers to performs a ALU control and write register destination. The load (lw) instructions loads data from address in data memory to a register. The store (sw) instructions stores a register value to an address in data memory. The branch not equal (beq) instructions jumps to a new address using an offset in instruction memory when the ALU result = 0. The no operation (nop) instructions perform nothing on program count, register values, or data memories but acts like stalls to let other instruction propagate through the MIPS pipeline without changing the current state.

A white and blue text with black text

Description automatically generated with medium confidence

To verify the behavioral model of the MIPS pipeline, a .mem/.txt file will be placed in instruction to perform various lw and add instructions. This is machine code contrary to assembly or high-level language like C as these are typical to program a processor. This program will simply load values from data memory then perform additions to eventually have register1 contain value 12 in decimal. No operations are inserted to prevent data hazards, more specifically read after write (RAW) and write after read (WAR) with register1 values are needed as dependencies. Without these no operations, it is likely that pipeline will produce incorrect/unexpected data as the contents of register1 are not saved in time for the next instruction.

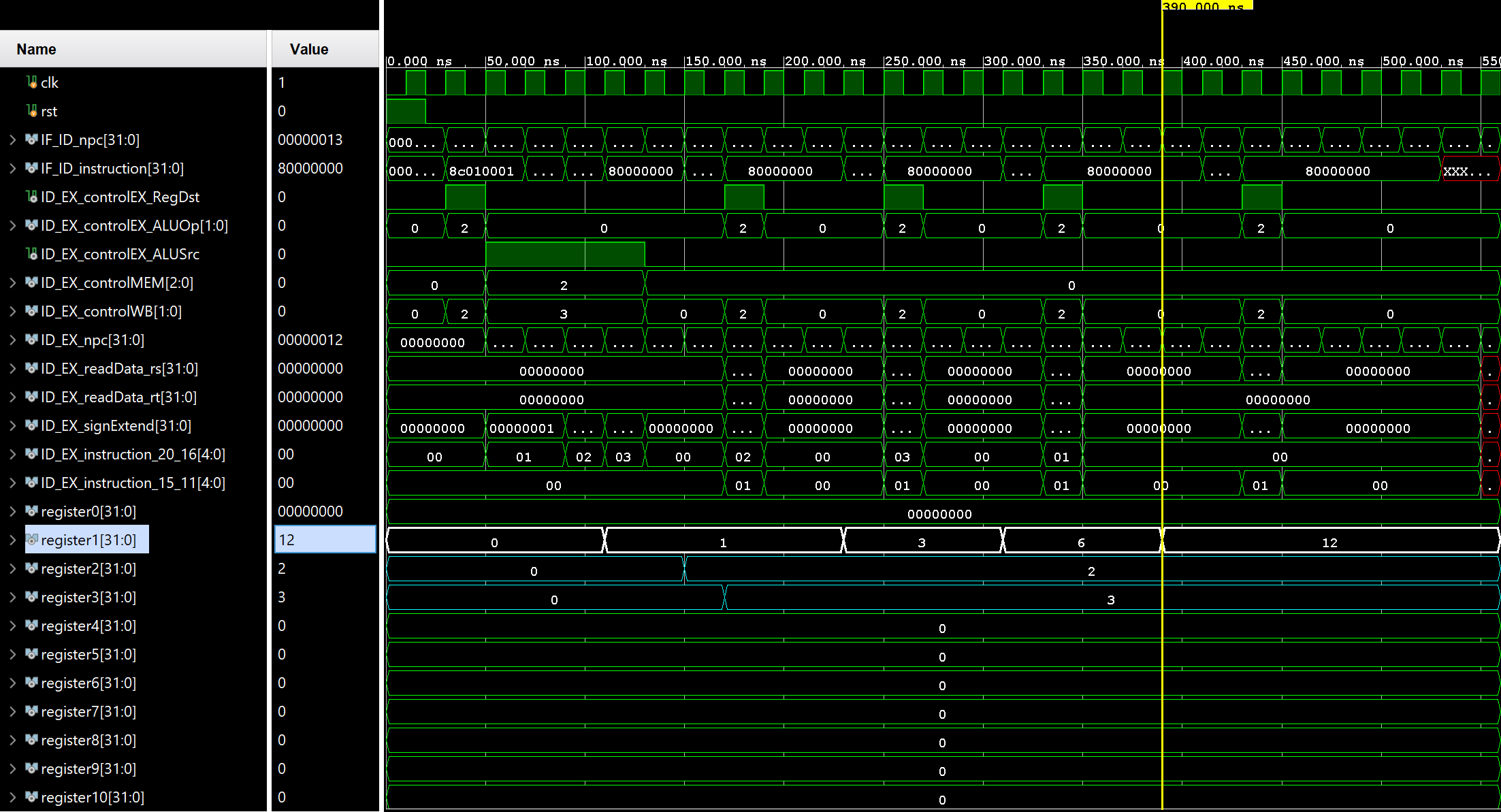
A screenshot of a computer program

Description automatically generated

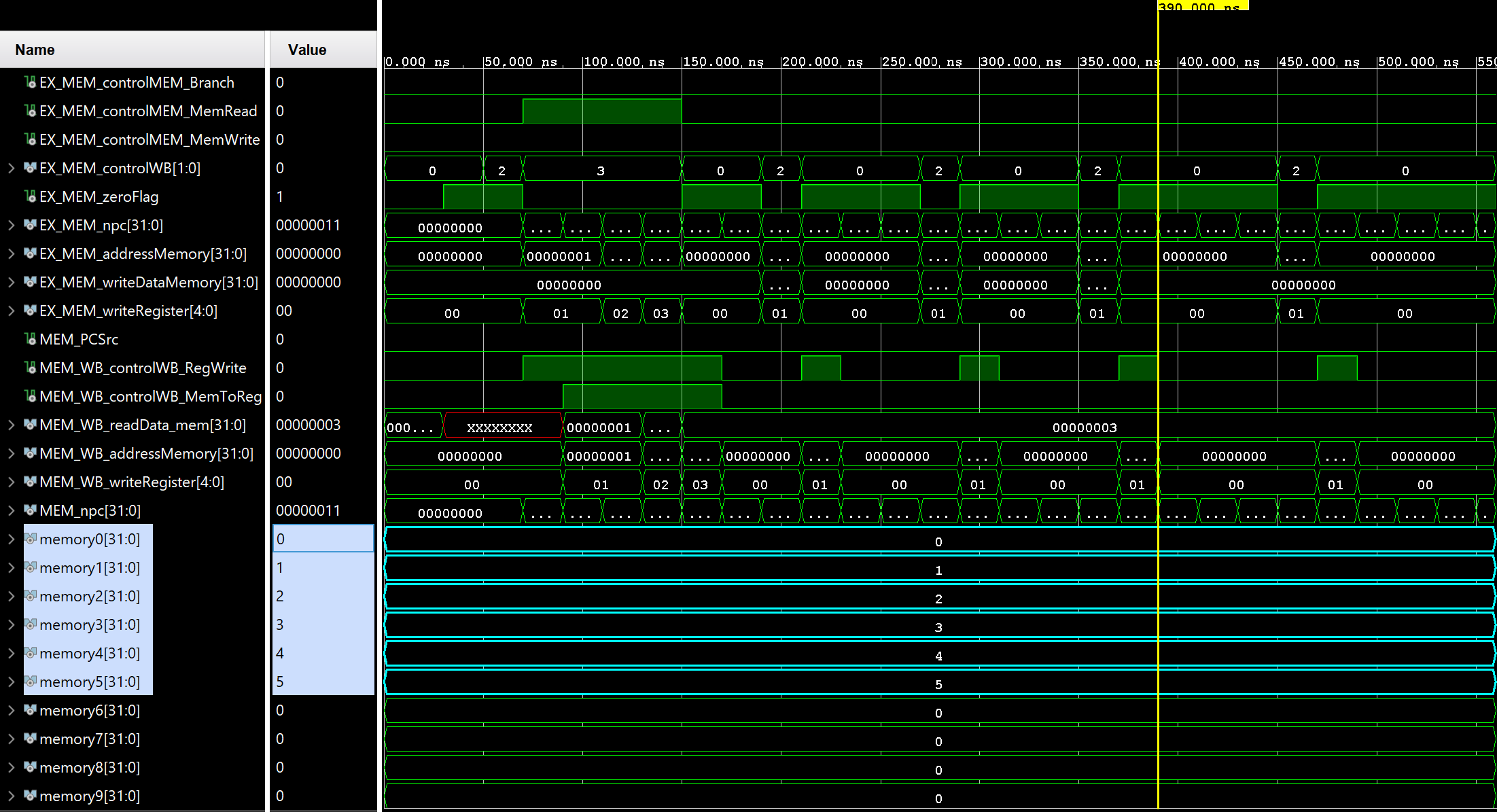
Data memory will be pre-loaded with some values bypassing the store (sw) instructions. These will get loaded towards the register with the load (lw) instructions.

**A yellow card with numbers

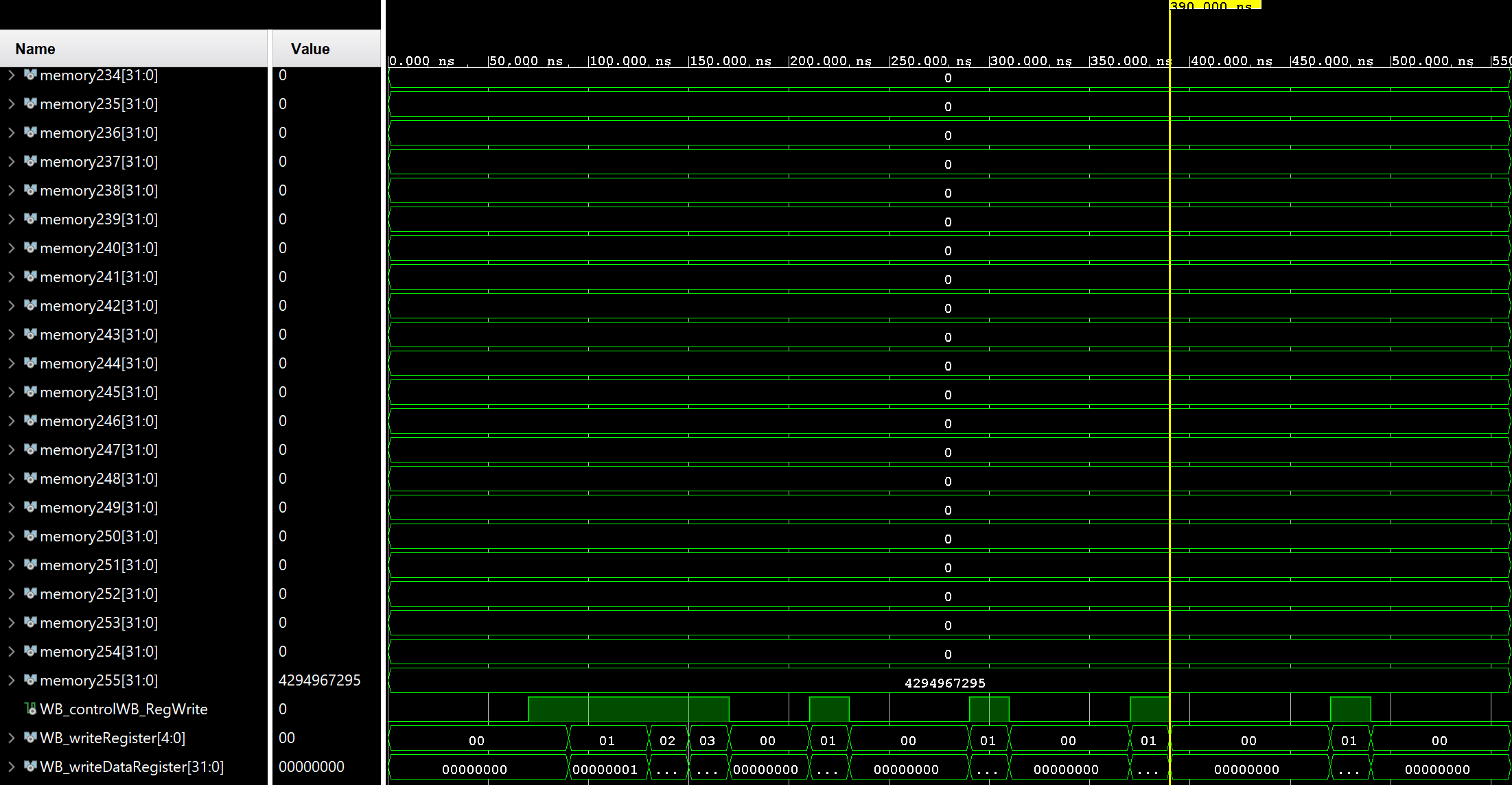
Description automatically generated**



MIPS\_PIPELINE\_tb\_wf\_register1.png



MIPS\_PIPELINE\_tb\_wf\_memories.png

  
MIPS\_PIPELINE\_tb\_wf\_extras.png

The hazards that can occur are structural, data, and control.

Structural hazards occur when a conflict for use of a resource is busy as there is insufficient hardware to support a computation in given pipeline segment, in other words frequently used components such as: muxs or ALUs can be used for the entire pipeline however if parallelism is introduced to overlay these instructions to improve performance, that two stages might need that component are the same time. Adding separate hardware for muxs or ALUs can solve this problem. Structural hazards are hardware dependent.

Data hazards is the most common hazard as instructions that require the values of a preceding instruction (dependencies) that cannot not write back the data in time that a value is read or write incorrectly by the next instruction. The possible data hazards are read after write (RAW), write after read (WAR), and write after write (WAW). Data hazards remedies are to forwarding aka bypassing to write back values faster skipping segment under certain conditions, code re-ordering can re-order instruction through either source code or assembler that instructions that have no dependencies are executed first before instructions with dependencies, the most simple fix is the stall insertion that inserts stalls (no operation instructions) to delay the next instructions execute to have properly read/write dependencies but decreases efficiency overall used only if forwarding and re-ordering fails, register renaming to create logical register is another fix. Data hazards are software dependent

Control hazards are to do with branch instructions that as the next instruction fetched is dependent on a branch outcome and must wait, pipelines could make wrong decisions on branch predictions. Control hazards are software dependent.

The current MIPS pipeline can be improved upon by adding hardware components such as: hazard detection unit to detect and handle hazards as needed and forwarding unit to improve performance.

A diagram of a computer system

Description automatically generated