Design And Simulation Of A Pipelined RISC Processor Using The Cadmium DEVS Framework

Ryan Rizk, Tala Nemeh, Belal AlObieda Prof. Gabriel Wainer

Department of Systems and Computer Engineering
Carleton University
Ottawa, ON K1S 5B6, CANADA
{TalaNemeh,RyanRizk,BelalAlObieda}@cmail.carleton.ca

ABSTRACT

The Discrete Event System Specification (DEVS) Formalism offers a modular hierarchical framework for modeling complex systems driven by discrete events. This framework allows to break down the system into reusable atomic and coupled models. Utilizing these properties, we designed and simulated a four-stage pipelined Reduced Instruction Set Computer (RISC) processor in the Cadmium DEVS environment. The pipeline stages include Fetch, Decide, Execute and Memory Access/ Write Back. To implement this pipeline in DEVS we designed six core models: Instruction Memory, Data Memory, Instruction Unit, Decode Unit, Execution Unit, and Register Unit as DEVS atomic models which were interconnected to form the coupled pipeline model. Using Cadmium simulations we verify correct behaviour of each pipeline stage and the overall instruction flow.

KEYWORDS*: discrete-event modeling and simulation; DEVS; RISC processor*

I. INTRODUCTION

Pipelined processors dominate modern embedded and general-purpose computing because they can overlap instruction activities and compute one instruction per clock once the pipeline is full[3]. Modeling such data paths early in the design cycle helps explore timing, hazards, and control strategies before committing to hardware.

The Discrete Event System Specification (DEVS) formalism, implemented in the Cadmium toolkit, offers a modular, hierarchical way to build and simulate these systems [2]. Each functional unit is an atomic model, and their connections form higher-level coupled models.

The goal of this project was to design and simulate a pipelined RISC (Reduced Instruction Set Computer) processor using Cadmium DEVS framework. This RISC processor includes a 13 bit instruction set, eight general purpose registers, and a four stage pipeline: Fetch, decode, execute and memory access/write back. In this implementation of the RISC processor, six main components are included in the architecture: Instruction Memory and Data Memory, Instruction Unit, Decode Unit, Execution Unit, and Register Unit.

The main idea for this project is to utilize the DEVS formalisim's modular structure to design each pipeline component independently and to simulate the interactions between the components to illustrate the pipeline behaviour.

Simulation results showed successful instruction processing across all pipeline stages. The designed processor was able to correctly fetch, decode, and execute instructions, as well as perform memory and write back operations. After the pipeline was filled with instructions, the simulated processor successfully executed instructions continuously and processed one instruction per clock cycle to ensure optimal throughput.

II.BACKGROUND

This section positions our work by outlining (i) the rationale for RISC pipelines, (ii) the DEVS/Cadmium modeling framework we employ, (iii) the advantages of a fully modular DEVS approach for pipeline prototyping, and iv) our RISC Processor Design using DEVS.

i) RISC Processors: A RISC processor is characterized as simple and efficient because it uses a fixed-length instruction set, fewer instructions format, and simplified decoding logic[4]. The simplified architecture speeds up

the pipeline execution. In computer architecture, a pipeline is a technique that allows the overlapping of execution stages for different instructions. This enables the processing of different instructions at the same time in different stages, increasing throughput and improving performance and throughput[4].

ii)DEVS and Cadmium Toolkit: The DEVS formalism models systems that react to discrete input events as a hierarchy of atomic models (with input/output ports, state variables, external / internal transition functions, output function λ, and time-advance function ta) and coupled models that connect those atomics together[2]. Cadmium is a tool based on the DEVS formalism for modeling and simulation Discrete-Event systems. It is a header-only library implemented in C++ [1].

iii)Advantages of a Modular DEVS Approach for Pipeline Design: Modeling each component independently using DEVS atomic models allows for more flexibility, easier debugging, and faster testing [2]. Changes to one part of the pipeline, such as modifying the execution stage or adjusting memory access timing, can be made without having to redesign the entire processor model. This modularity also enables individual testing of each unit before full system integration. As a result, the modular DEVS approach is ideal for building, validating, and extending pipelined processor architectures in a scalable way [2].

iv) Our Design: In this project, each component in the four-stage pipeline was designed as a DEVS atomic model with input and output ports, internal state, state transitions, and timing behaviour. By using the DEVS framework, we were able to break down the RISC processor into basic independent components. The pipeline implemented in this project consists of four main stages:

A. Instruction Fetch: Retrieves the next instruction from Instruction memory based on the program counter (PC) and passes it to the decoder unit.

B. Instruction Decode: Breaks down the fetched instruction to determine the operation type (opcode), source registers, destination register, and memory addresses.

C. Execution: Carries out the arithmetic, logical, and shift operations.

D. Memory Read and Write-Back: Accesses data memory for load/store operations, or writes operation results into the register file.

Separating the pipeline into distinct DEVS atomic models allowed each component to be individually tested, making the design modular and easier to debug. Additionally, the modular approach makes it straightforward to add new features or handle more complex scenarios, such as hazard detection and pipeline stalls, in the future.

III.MODELS

In this project, each stage in the RISC pipeline was implemented as an individual DEVS atomic model. The coupling of these 4 models is the RISC Processor model. The RISC Processor coupled model connects four pipeline stages—Instruction Unit, Decode Unit, Execution Unit, and Register Unit where:

Instruction Unit holds and increments the PC, sends it to Instruction Memory, and forwards the fetched 13-bit instruction; Decode Unit parses opcode, source/destination register indices, and memory addresses; Execution Unit performs ALU operations; and Register Unit carries out both write back and register reads, storing results into one of eight general-purpose registers.

Additionally, there are the Instruction Memory (a read-only memory of 32 × 13-bit instructions) and DataMemory (a 16-word read/write array of 8-bit data) atomic models coupled in the Main Memory model. It accepts a program counter (PC) input to fetch instructions and load/store addresses plus data for memory operations, then outputs either the fetched instruction or loaded data. Finally, there is the top RISC System model which includes the RISC Processor model and the Main Memory Model coupled together.

We assume an initial PC of 0, and register file, and instructions and data memories pre-loaded according to the scenario.

To synchronize the system, each model's ta function returns its sigma delay when an input arrives, triggering its output and the next stage's external transition.

A. Atomic Models

1. Instruction Memory:

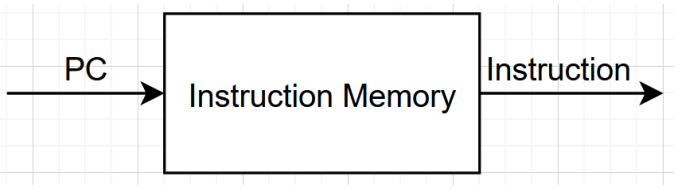

Figure 1: Instruction Memory Atomic Model

This is a read only memory to store the list of available 13-bit instructions. When it receives a PC value, it outputs the instruction located at that address.

DEVS Formalism:
InstructionMemory = ⟨ X, S, Y, δ_{ext}, δ_{int}, λ, ta ⟩
X: {pc: integer }
S: { instructions: array of 32 13-bit binary strings,
  pc: integer program counter,
  instruction: 13-bit binary string (current instruction),
  sigma: time until next internal event }
Y: {instruction: 13-bit binary string}
δ_{ext}:
  if pc arrives:
    pc = pc
    instruction = instructions[pc]
    sigma = 0.1
δ_{int}:
  Sigma = infinity
λ:
  send instruction = instruction
ta:
  Returns sigma

2. Instruction Unit (Instruction Fetch Stage):

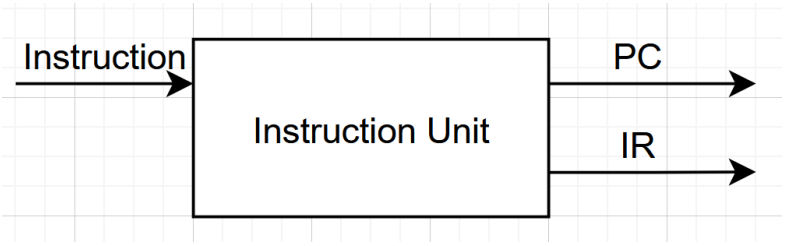

Figure 2: Instruction Unit Atomic Model

The instruction unit is primarily used to contain the PC. This register is responsible for keeping track of the next instruction to be processed. It sends the PC address to the instruction memory and is returned the corresponding instruction. It then forwards the instruction to the decode unit using the instruction register to be handled. Everytime a new instruction is received, the PC is incremented to point to the next instruction.

DEVS Formalism:
InstructionUnit = ⟨ X, S, Y, δ_{ext}, δ_{int}, λ, ta ⟩

X: { instruction: 13-bit binary string }
S: { pc: integer program counter,
  ir: 13-bit binary string instruction register,
sigma: time until next internal event}
Y: {pc: integer program counter, ir: 13-bit binary string instruction register}
δ_{ext}:
  if instruction arrives:
    ir   = instruction
    pc   = pc + 1
    if pc == 31:
      pc = 0
    Sigma = 0
δ_{int}:
  Sigma = infinity
λ:
  send pc = pc
  send ir = ir
ta: Returns sigma

3. Decode Unit (Decode Stage):

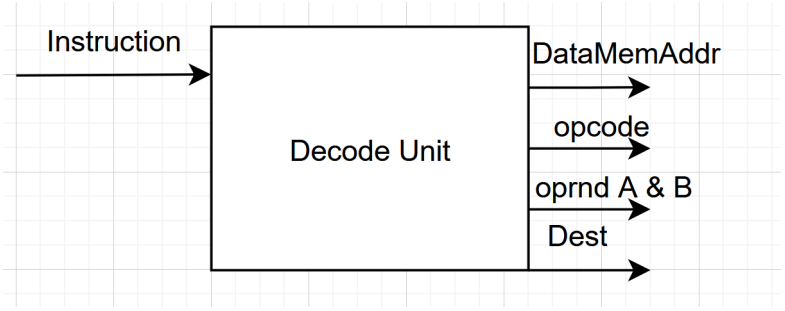

Figure 3: Decode Unit Atomic Model

The decode unit receives a 13-bit instruction and parses it to extract the opcode, the source and destination registers, and memory addresses. The opcode is contained in the leftmost 4 bits 13-10. This allows the processor to utilize up to 16 instructions. For a memory load operation, bits 8-5 are used as the data memory location and bits 3-1 are used as the destination register. For a memory store operation, bits 4-1 are used as the data memory location and bits 7-5 are used as the register location to be stored. For any other operation, bits 9-7 represent the register location of operand A, bits 6-4 represent the register location of operand B and bits 3-1 represent the destination address.

DEVS Formalism:

DecodeUnit = ⟨X, S, Y, δ_{ex}□, δ_{i}□□,, λ, ta⟩

X: { instruction: 13-bit binary string}
S:{ instruction: current 13-bit binary string,
dataMemAddr: 4-bit binary string,
opcode: 4-bit binary string,
opnda: 3-bit binary string,
opndb: 3-bit binary string,
dest: 3-bit binary string,
sigma: time until next internal event}
Y: { dataMemAddr: 4-bit binary string,
opcode: 4-bit binary string,
opnda: 3-bit binary string,
opndb: 3-bit binary string
dest: 3-bit binary string }
δ_{ex}□:

```
if instruction arrives:
   instruction = instruction
   opcode =  bits 0–3 of instruction
   if opcode == "1110":        // load
     dataMemAddr = bits 5–8 of instruction
     dest = bits 10–12 of instruction
     opnda = "000";  opndb = "000"
   else if opcode == "1111":    // store
     dataMemAddr = bits 9–12 of instruction
     dest = bits 6–8 of instruction
     opnda = "000";  opndb = "000"
   else:          // ALU or NOP
     dataMemAddr = "0000"
     opnda = bits 4–6 of instruction
     opndb  =  bits 7–9 of instruction
     dest = bits 10–12 of instruction
Sigma = 0
```

δ_{i}□□:

```
Sigma = infinity
```

λ:

```
send dataMemAddr = dataMemAddr
send opcode = opcode
```

send opnda = opnda
send opndb = opndb
send dest = dest
ta: Returns sigma

4. Execution Unit (Execution Stage):

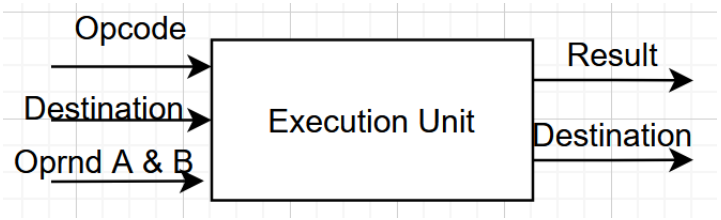

Figure 4: Execution Unit Atomic Model

This unit performs the actual arithmetic and logical operations based on the opcode decided by the decode unit. It receives its input from the decode unit and outputs the execution result to the register file unit.

DEVS Formalism
ExecutionUnit = ⟨X, S, Y, δ_{ex}□, δ_{i}□□,, λ, ta⟩

X: { instruction: 13-bit binary string }
S: {instruction: current 13-bit binary string, opcode: 4-bit binary string, opnda: 3-bit binary string, opndb: 3-bit binary string, dest: 3-bit binary string, sigma: time until next internal event}
Y:{result: 8-bit operation result, dest: 3-bit binary string}
δ_{ex}□:
if instruction arrives:
  instruction = instruction
  opcode = bits 0–3 of instruction
 // ALU or NOP only
   opnda = bits 4–6 of instruction
   opndb = bits 7–9 of instruction
   dest = bits 10–12 of instruction
 Sigma = 0.02
δ_{i}□□:
Sigma = infinity
λ: send result->result
send dest ->dest
ta:
Returns sigma

5. Data Memory (Memory Access Stage):

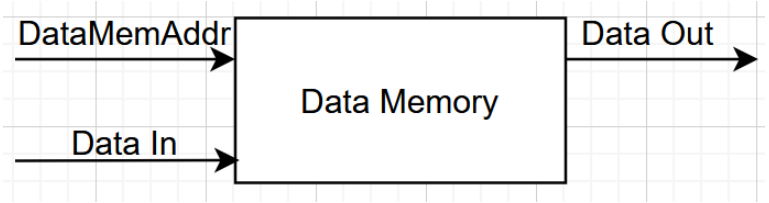

Figure 5: Data Memory Atomic Model

The data memory model received memory addresses for load and store instructions. Based on the operation, it either reads from or writes to memory. The result of a load instruction is passed to the write-back stage to be written in the register file.

DEVS Formalism:
DataMemory = ⟨ X, S, Y, δ_{ext}, δ_{int}, λ, ta ⟩

X: {dataMemAddrIn: 4-bit binary string (store address) ,
  dataMemAddrOut: 4-bit binary string (load address),
  dataIn: 8-bit binary string (write data)}

S: {data: array of sixteen 8-bit binary strings (memory contents),
  stAddr: 4-bit binary string (last store address),
  ldAddr: 4-bit binary string (last load address),
  dataOut: 8-bit binary string (last read data),
  sigma:time until next internal event}
Y: {dataOut: 8-bit binary string}
δ_{ext}:
  if dataMemAddrIn arrives:
    stAddr = dataMemAddrIn
    if dataIn arrives:
      data[integer value of stAddr] = dataIn
  else if dataMemAddrOut arrives:
    ldAddr = dataMemAddrOut
    dataOut = data[integer value of ldAddr]
δ_{int}:
  Sigma = infinity
λ:
  send dataOut = dataOut
ta:
  Returns sigma

6. Register Unit (Write-back Stage):

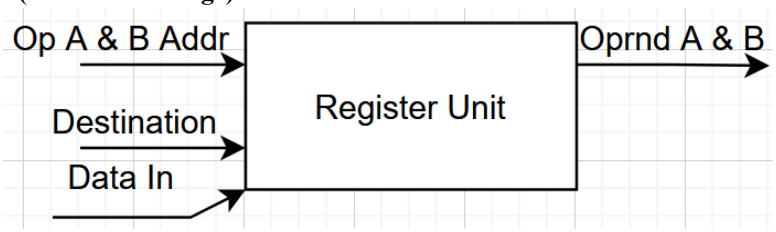

Figure 6: Register Unit Atomic Model

The register file unit stores the results from the execution unit or values loaded from the data memory during a load operation. It writes these values to the destination register specified by the instruction.

DEVS Formalism:

MregisterUnit = ⟨ X, S, Y, δ_{ex}□, δ_{i}□□, λ, ta ⟩

X: {result: 8-bit binary string, dst: 3-bit binary string, opnda_addr: 3-bit binary string, opndb_addr: 3-bit binary string}
S: { regs: array of eight 8-bit binary strings (register contents),
  readA: integer (index 0-7 of source A),
  readB: integer (index 0-7 of source B),
  sigma: time until next internal transition}
Y: { oprnd_a: 8-bit binary string, oprnd_b: 8-bit binary string, stAddrOut: 4-bit binary string, stData: 8-bit binary string }
δ_{ex}□:
  hasUpdate = false
  if result & dst arrive:
    d = integer value of dst
    regs[d] = result
    hasUpdate = true
  if opnda_addr arrives:
    readA = integer value of opnda_addr
    hasUpdate = true
  if opndb_addr arrives:
    readB = integer value of opndb_addr
    hasUpdate = true
  σ = (hasUpdate ? 0 : ∞)
δ_{i}□□:

Sigma = infinity
λ:
  send oprnd_a = regs[readA]
  send oprnd_b = regs[readB]
ta:
Returns sigma

B. Coupled Models

1. Main Memory:

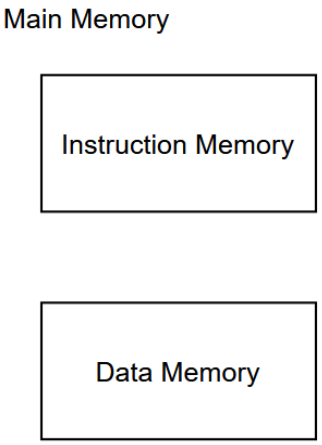

Figure 7: Main Memory Coupled Model

This coupled model includes both the Instruction Memory and the Data Memory atomic models. External inputs include the PC for instruction fetch and the data address to be read from or written into. During the fetch stage, the Main Memory model forwards the PC input to the Instruction Memory model then outputs the fetched instruction. For a Load/Store instruction, it takes in the data address and the data to be written (for a store instruction) and passes them to the data memory. If it is a Load instruction, it outputs the requested stored value.

DEVS Formalism:

main_memory = ⟨ X, Y, { M_i }, EIC, IC, EOC ⟩

X: { pc:int, dataMemAddrIn:string, dataMemAddrOut: string, dataIn:string, ldRegIn:string}
Y: {instruction:string, dataOut:string, ldRegOut:string}
M_i: {instruction_mem:InstructionMemory, data:DataMemory }
EIC: (pc -> instruction_mem.pc,
dataMemAddrIn -> data.dataMemAddrIn,
dataMemAddrOut -> data.dataMemAddrOut,
dataIn -> data.dataIn,
ldRegIn -> data.ldRegIn)
IC: ∅
EOC: instruction_mem.instruction -> instruction, data.dataOut -> dataOut, data.ldRegOut -> ldRegOut)

2. RISC Processor:

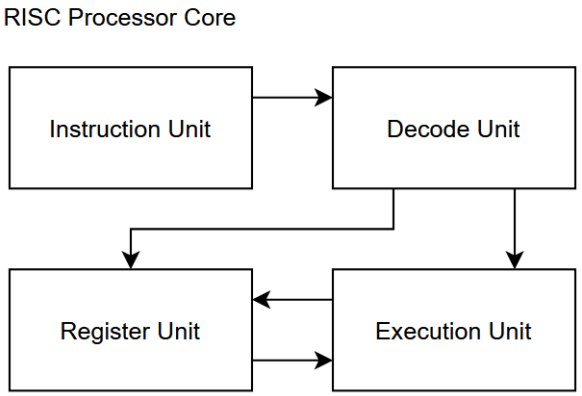

Figure 8: RISC Processor Core Coupled Model

This coupled model implements the core of the 4-stage RISC pipeline, excluding memory. On each cycle, an instruction flows Fetch→Decode→Execute→Write-Back.
For LOADs, the decode stage emits a memory address and target register; when memory returns the word, it is written into the register file. For STOREs, the register unit emits the data and target address for memory. All other opcodes drive the ALU in the execute stage and write back the result.

This modular DEVS design clearly separates pipeline stages and memory interface logic, making it straightforward to replace any different memory or pipeline implementations without affecting the core processor coupling.

DEVS Formalism

risc_processor = ⟨ X, Y, {M_i}, EIC, IC, EOC ⟩

X: { instrIn: 13-bit instruction word, ldData: 13-bit load-return data, ldRegIn: 3-bit load-dest register index }
Y = { pc: integer program counter, dataMemAddrLd: 4-bit load address , ldRegOut: 3-bit load-dest index, stAddrOut: 4-bit store address, stData: 13-bit store data }
{ M_i } = { i_unit: InstructionUnit, d_unit: DecodeUnit, e_unit: ExecutionUnit,r_unit: RegisterUnit}
EIC: (instrIn -> i_unit.instruction, ldData -> r_unit.ldData, ldRegIn -> r_unit.ldReg)
IC: (i_unit.ir-> d_unit.instruction
d_unit.opcode -> e_unit.opcode
d_unit.dest -> e_unit.destAddr
d_unit.opnda -> r_unit.opnda_addr
d_unit.opndb -> r_unit.opndb_addr
r_unit.oprnd_a -> e_unit.opnda
r_unit.oprnd_b -> e_unit.opndb
e_unit.result -> r_unit.result
e_unit.destReg -> r_unit.dst
d_unit.dataMemAddrSt -> r_unit.stAddrIn
d_unit.stReg -> r_unit.stReg)
EOC:
(e_unit.result -> aluResult
e_unit.destReg -> writeReg
i_unit.pc -> pc
d_unit.dataMemAddrLd -> dataMemAddrLd
d_unit.ldReg-> ldRegOut
r_unit.stAddrOut -> stAddrOut
r_unit.stData -> stData)

C. RISC System:

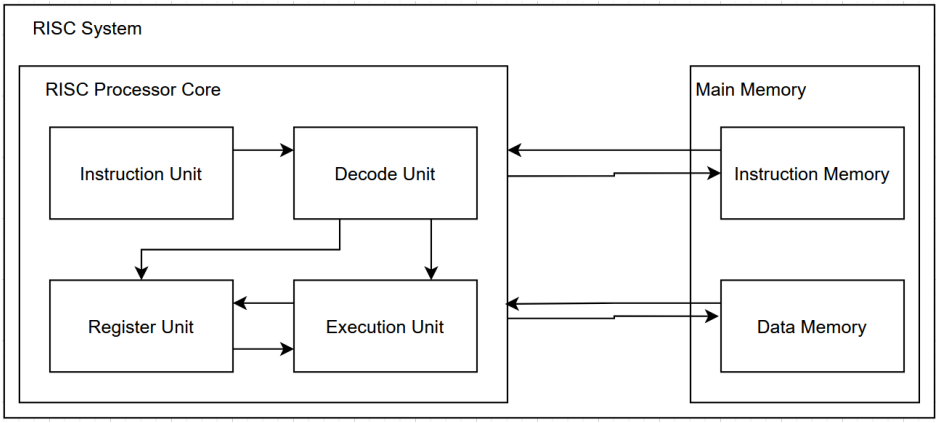

Figure 9: RISC System Coupled Model

The FullSystem top-level coupled model integrates the RISC Processor and Main Memory

Devs Formalism

FullSystem = ⟨ X, Y, {M_i}, EIC, IC, EOC ⟩

X: ∅
Y: ∅
EIC: ∅
IC: (risc.pc –> mm.pc,
mm.instruction -> risc.instrIn,
risc.dataMemAddrLd -> mm.dataMemAddrIn,
risc.ldRegOut -> mm.ldRegIn,
mm.dataOut-> risc.ldData,
mm.ldRegOut-> risc.ldRegIn,
risc.stAddrOut-> mm.dataMemAddrIn,
risc.stData-> mm.dataIn)
EOC: ∅

IV. SIMULATION RESULTS

This section demonstrates the behaviour of our pipelined RISC processor models. Before the full integration of the processor components, we performed unit testing of the four main atomic models: The Instruction Unit, the Decode Unit, the Execution Unit, and the Register Unit. Each unit was tested individually using a generator to verify timing behaviour and expected outputs.

A. Instruction Unit

The simulation begins with the instruction unit generator (instrUnit_gen) outputting the first instruction to the instruction unit. As seen in the logs in Figure 10:

Cycle 1: The instruction unit generator sends a load instruction (1110000000000) from PC=0. The instruction unit receives it and forwards it to the decode unit.

Cycle 2: The next instruction is an ALU operation (0001000100010). It is received and forwarded to the decode unit with updated PC=1.

Cycle 3: The final store instruction (1111001000011) is processed and sent to the decode unit successfully.

Each instruction was successfully received and saved in the Instruction Register (IR) before being forwarded to the decoding unit. The program counter was updated with each instruction as expected.

```
Tala@devssim:~/RISC$ ./bin/sample_project
time;model_id;model_name;port_name;data
0;1;instrUnit_gen;;{σ=0.1, cycle=0}
0;2;d_unit;;{sigma: inf, instruction: 0000000000000000000}
0;3;i_unit;;{sigma: 0.1, pc: 0, ir: 0000000000000}
0;4;i_mem;;{sigma: inf, pc: -641017870}
0;5;clock;;{0.05}
0.05;5;clock;clk;1
0.05;5;clock;;{0.05}
[i_unit]: instruction sent to d_unit: 0000000000000
0.1;1;instrUnit_gen;clk;1
0.1;1;instrUnit_gen;rst;1
0.1;1;instrUnit_gen;;{σ=0.1, cycle=1}
0.1;3;i_unit;pc;0
0.1;3;i_unit;ir;0000000000000
0.1;3;i_unit;;{sigma: 0.1, pc: 0, ir: 0000000000000000000}
0.1;5;clock;clk;1
0.1;5;clock;;{0.05}
0.15;5;clock;clk;1
0.15;5;clock;;{0.05}
[InstrUnitGen] cycle 1: rst=0, instr=1110000000000
[i_unit]: instruction sent to d_unit: 0000000000000000000
0.2;1;instrUnit_gen;clk;1
0.2;1;instrUnit_gen;rst;0
0.2;1;instrUnit_gen;instruction;1110000000000
0.2;1;instrUnit_gen;;{σ=0.1, cycle=2}
```

```
0.2;1;instrUnit_gen;rst;0
0.2;1;instrUnit_gen;instruction;1110000000000
0.2;1;instrUnit_gen;;{σ=0.1, cycle=2}
[i_unit]: instruction received from i_mem: 1110000000000
0.2;3;i_unit;pc;0
0.2;3;i_unit;ir;0000000000000000000
0.2;3;i_unit;;{sigma: 0.1, pc: 1, ir: 1110000000000}
0.2;5;clock;clk;1
0.2;5;clock;;{0.05}
0.25;5;clock;clk;1
0.25;5;clock;;{0.05}
0.3;5;clock;clk;1
0.3;5;clock;;{0.05}
[InstrUnitGen] cycle 2: rst=0, instr=0001000100010
[i_unit]: instruction sent to d_unit: 1110000000000
0.3;1;instrUnit_gen;clk;1
0.3;1;instrUnit_gen;rst;0
0.3;1;instrUnit_gen;instruction;0001000100010
0.3;1;instrUnit_gen;;{σ=0.1, cycle=3}
[i_unit]: instruction received from i_mem: 0001000100010
0.3;3;i_unit;pc;1
0.3;3;i_unit;ir;1110000000000
0.3;3;i_unit;;{sigma: 0.1, pc: 1, ir: 0001000100010}
0.35;5;clock;clk;1
0.35;5;clock;;{0.05}
0.4;5;clock;clk;1
```

```
0.3;3;i_unit;;{sigma: 0.1, pc: 1, ir: 0001000100010}
0.35;5;clock;clk;1
0.35;5;clock;;{0.05}
0.4;5;clock;clk;1
0.4;5;clock;;{0.05}
[InstrUnitGen] cycle 3: rst=0, instr=1111001000011
[i_unit]: instruction sent to d_unit: 0001000100010
0.4;1;instrUnit_gen;clk;1
0.4;1;instrUnit_gen;rst;0
0.4;1;instrUnit_gen;instruction;1111001000011
0.4;1;instrUnit_gen;;{σ=inf, cycle=4}
[i_unit]: instruction received from i_mem: 1111001000011
0.4;3;i_unit;pc;1
0.4;3;i_unit;ir;0001000100010
0.4;3;i_unit;;{sigma: 0.1, pc: 1, ir: 1111001000011}
0.45;5;clock;clk;1
0.45;5;clock;;{0.05}
0.5;5;clock;clk;1
0.5;5;clock;;{0.05}
[i_unit]: instruction sent to d_unit: 1111001000011
0.5;3;i_unit;pc;1
0.5;3;i_unit;ir;1111001000011
0.5;3;i_unit;;{sigma: inf, pc: 1, ir: 1111001000011}
0.55;5;clock;clk;1
0.55;5;clock;;{0.05}
0.6;5;clock;clk;1
```

Figure 10: Instruction Unit Atomic Model Testing

B. Decode Unit

The decode unit was responsible for parsing the fetched instructions and outputs the opcode, operands addresses, destination register, and a memory address if applicable. In this test, we used the same instruction generator as above to provide the decode unit with an instruction. As shown in the logs below (Figure 11):
Cycle 1 – Load Instruction (1110000000000), the decode unit correctly recognized the instruction as a load operation (opcode = 1110) and correctly parsed memory address and destination register.
Cycle 2 – ALU Instruction (0001000100010), the decode unit correctly identified the instruction as an ALU operation (opcode = 0001) as well as Operand A = 000, Operand B = 100 and Dest = 010.
Cycle 3 – Store Instruction (1111001000011), the decode unit correctly identified it as a store instruction (opcode = 1111). Memory address and source register were extracted.

Each instruction was decoded correctly immediately after it was received, and the output signals matched the expected fields based on the instruction format.

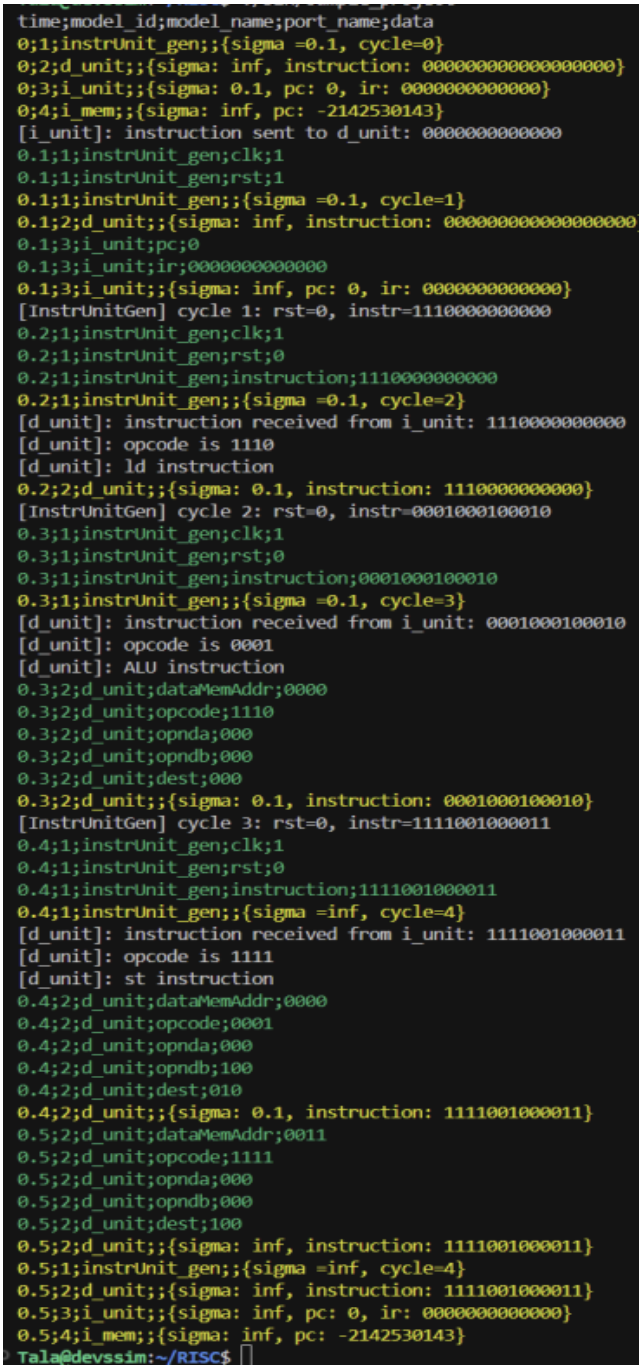

Figure 11: Decode Unit Atomic Model Testing

C. Register Unit

Figure 12 shows the unit test for the register unit. A simple generator model (RegGen) was used to validate the basic functionalities of write-back, operand reading, and operand forwarding. The goal of the test was to ensure that the register file correctly handled simultaneous read and write operations, saved data across cycles, and reflected updates on output ports in a timely manner. The generator cycled through three simple instructions. In each cycle, it drived different behaviour of the register file. The three cycles are summarized below.

Cycle 1: The generator issued a write to destination register R2 with a 13-bit value of 0000000000101. The register file correctly logged the write operation as [reg_unit] write R2 = 0000000000101. Following the write, operand A output was updated to reflect the new value of R2, while operand B remained zero since it pointed to R0. This demonstrates the immediate forwarding of written data to the output port.

Cycle 2: In the second cycle, RegGen performed another write, this time targeting register R3 with a value of 0000000000111. The generator issued a read request for R2 (operand A) and R3 (operand B). The log entries confirmed successful updates: [reg_unit] write R3 = 0000000000111 and [reg_unit] out A=0000000000101 B=0000000000111.

Cycle 3: During the final cycle, the generator simulated a read-only operation. The generator issued a read instruction to retrieve values from R2 and R3 without writing new data. The register file output remained consistent with the values recorded in Cycle 2, showing proper data updates when no write operation occurred.

This unit test demonstrates that the register-file functions as expected for read and write operations.

```
time;model_id;model_name;port_name;data
0;1;reg_gen;;{sigma=0.1, cycle=0}
0;2;r_unit;;{sigma=inf, readA=0, readB=0}
[RegGen] cycle 1: WRITE R2 <- 0000000000101
0.1;1;reg_gen;reg_wr_vld;1
0.1;1;reg_gen;result;0000000000101
0.1;1;reg_gen;dst;010
0.1;1;reg_gen;opnda_addr;010
0.1;1;reg_gen;opndb_addr;000
0.1;1;reg_gen;;{sigma=0.1, cycle=1}
[reg_unit] write R2 = 0000000000101
0.1;2;r_unit;;{sigma=0, readA=2, readB=0}
[reg_unit] out A=0000000000101  B=00000000
0.1;2;r_unit;oprnd_a;0000000000101
0.1;2;r_unit;oprnd_b;00000000
0.1;2;r_unit;;{sigma=inf, readA=2, readB=0}
[RegGen] cycle 2: WRITE R3 <- 0000000000111
0.2;1;reg_gen;reg_wr_vld;1
0.2;1;reg_gen;result;0000000000111
0.2;1;reg_gen;dst;011
0.2;1;reg_gen;opnda_addr;010
0.2;1;reg_gen;opndb_addr;011
0.2;1;reg_gen;;{sigma=0.1, cycle=2}
[reg_unit] write R3 = 0000000000111
0.2;2;r_unit;;{sigma=0, readA=2, readB=3}
[reg_unit] out A=0000000000101  B=0000000000111
0.2;2;r_unit;oprnd_a;0000000000101
0.2;2;r_unit;oprnd_b;0000000000111
0.2;2;r_unit;;{sigma=inf, readA=2, readB=3}
[RegGen] cycle 3: READ R2, R3
0.3;1;reg_gen;reg_wr_vld;0
0.3;1;reg_gen;opnda_addr;010
0.3;1;reg_gen;opndb_addr;011
0.3;1;reg_gen;;{sigma=0.1, cycle=3}
0.3;2;r_unit;;{sigma=0, readA=2, readB=3}
[reg_unit] out A=0000000000101  B=0000000000111
0.3;2;r_unit;oprnd_a;0000000000101
0.3;2;r_unit;oprnd_b;0000000000111
0.3;2;r_unit;;{sigma=inf, readA=2, readB=3}
0.4;1;reg_gen;;{sigma=0.1, cycle=4}
0.5;1;reg_gen;;{sigma=inf, cycle=5}
0.5;1;reg_gen;;{sigma=inf, cycle=5}
0.5;2;r_unit;;{sigma=inf, readA=2, readB=3}
```

Figure 12: Register Unit Atomic Model Testing

D. Execution Unit

This unit was tested using a generator for the Execution Unit, which drove three ALU tests into the Execution Unit. The instructions are ADD, SUB, and AND, and in each case the unit computed and returned the correct 13-bit result and destination index.

For Test 0 (ADD R1+R2->R3), at t=0 the generator emitted:
opcode=0001, A=00100010, B=01000100, dest=011
And the execution unit responded by sending:
destReg=011, result=01100110 (0x22+0x44=0x66)

For Test 1 (SUB R3–R1->R4), at t=0.1 the inputs to the Execution Unit are:
opcode=0010, A=01100110, B=00100010, dest=100
And the outputs:
destReg=100, result=01000100 (0x66–0x22=0x44)

Finally, for Test 2 (AND R2 AND R3->R5) at t=0.2 the unit received:
opcode=0011, A=01000100, B=01100110, dest=101
And outputted:
destReg=101, result=01000100 (0x44&0x66=0x44)

These results, show in Figure 13, confirm that the ExecutionUnit correctly latches its inputs, performs the ALU operation, and schedules the correct output exactly once per test.

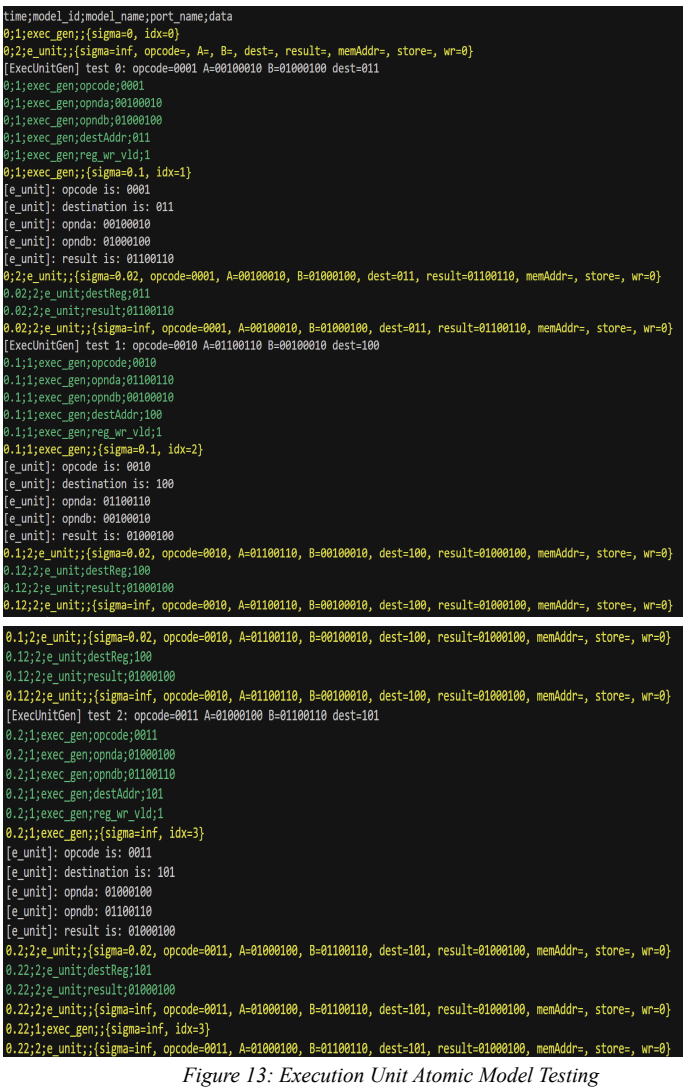

Figure 13: Execution Unit Atomic Model Testing

E. RISC Processor

The instructions driving the RISC core model, coming from the RiscGenerator are:

```
, instructions{
    "0001001010011",  // ADD  R1 + R2 -> R3
    "0010011001100",  // SUB  R3 - R1 -> R4
    "0011010011101"   // AND  R2 & R3 -> R5
}
```

Figure 14: Instructions from RiscGenerator Model

The test results show all four components (the Instruction Unit, the Decode Unit, the Register Unit, and the Execution Unit) exchanging data through their output and input port correctly. Each unit forwards its results to the next stage in the pipeline. Load and store instructions are excluded in this test and left for the full system test simulation, where the Main Memory is coupled with the RISC Processor model. The test also shows how all four models are synchronized and no model proceeds without the required input from another model.

Screenshots of the integration run are provided in Figure 15. The RiscGenerator outputs 3 instructions. The first instruction is sent at the start of the RiscGenerator model, while the subsequent ones are only sent after the instruction unit receives an instruction and increments the value of PC. For example, when the ADD instruction (0001001010011) is fetched at PC=1, the trace shows that the decode unit outputs: opcode is 0001. RegisterFile then outputs the contents of the two source operands at t=0.01:

r_unit;oprnd_a;00100010
r_unit;oprnd_b;01000100

The ExecutionUnit receives them and logs:

opnda: 00100010
opndb: 01000100
result is: 01100110

Successfully adding the two operands and outputting the correct result. It then sends the result back to the registeregister unit to be written back:

e_unit;destReg;011
e_unit;result;01100110

Similarly, for the SUB instruction (0010011001100) at PC = 2, we see:

[d_unit]: opcode is 0010
r_unit;oprnd_a;01100110
r_unit;oprnd_b;00100010
[e_unit]: opcode is 0010
[e_unit]: opnda: 01100110
[e_unit]: opndb: 00100010
[e_unit]: result is: 01000100
[e_unit]: destination is: 100

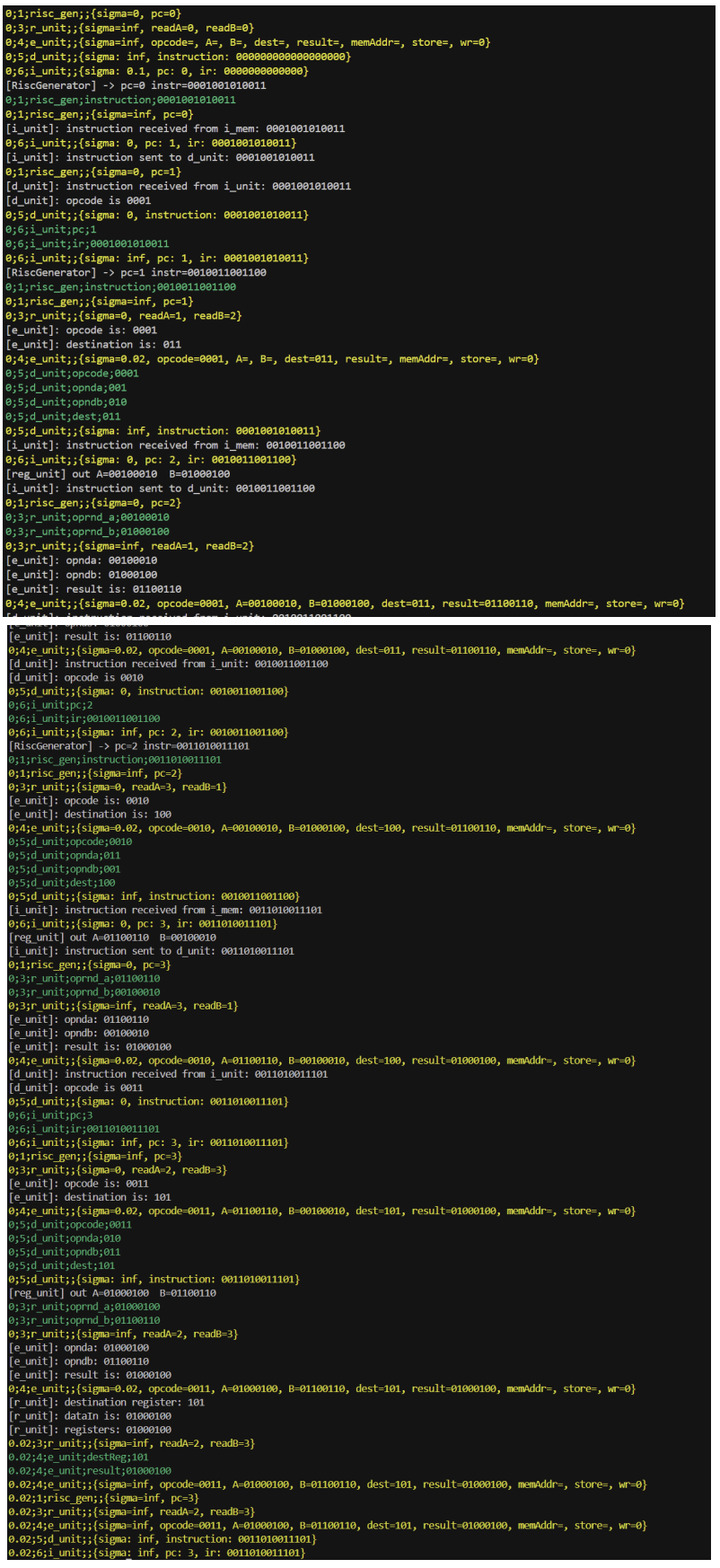

Figure 15: RISC Processor Core Coupled Model Testing

G. Full System

This simulation was conducted on the full system coupled model including the pipelined RISC processor and the Main Memory Model. The main memory included both Instruction Memory (for fetching program instructions) and Data Memory (for supporting load and store operations).

The processor successfully fetched, decoded, and executed a sequence of instructions, performing register operations and memory accesses as expected. The instruction included in the Instruction Memory are shown in Figure 16:

```
std::array<std::string, 32> instructions = {
    "0001001010011",
    "0010001010011",
    "0011010011100",
    "0100011100101",
    "0101100101110",
    "1110000010001",
    "1111000010010",
    "0001010101111",
    "0000000000000","0000000000000","0000000000000","0000000000000",
    "0000000000000","0000000000000","0000000000000","0000000000000",
    "0000000000000","0000000000000","0000000000000","0000000000000",
    "0000000000000","0000000000000","0000000000000","0000000000000",
    "0000000000000","0000000000000","0000000000000","0000000000000",
    "0000000000000","0000000000000","0000000000000","0000000000000"
};
```

Figure 16: Instruction set in Instruction Memory

The simulation ran from pc=0 to pc=31, with each instruction moving through all four pipeline stages: Fetch, Decode, Execute, and Memory Access/Write Back.

1. Instruction Fetch: The Instruction Unit retrieved instructions sequentially from Instruction Memory starting from pc= 0 and wrapping back to the first instruction when pc reached 31.
2. Decoding: The Decode Unit correctly parsed instruction fields. For instance, at pc=1, decoding the instruction 0001001010011 resulted in operands being sourced from registers R1 and R2, and the destination specified as R3. The opcode was correctly defined as 0001.
3. Execution and Write-Back: The Execution Unit performed ALU operations correctly. An example at pc=1 showed the addition of operands 00100010 and 01000100, resulting in 01100110, which was correctly written to R3.
4. Memory Operations: Memory load and store instructions were properly handled. For example, at pc=6, the load instruction 1110000010001 accessed address 1 in Data Memory and wrote the retrieved value into register R1. Subsequently, at pc=7, a store instruction 1111000010010 stored the value from R1 into Data Memory address 2.

```
time;model_id;model_name;port_name;data
[33m0;2;dataMemory;;{1}[0m
[33m0;3;instructionMemory;;{sigma: inf, pc: 1665503336}[0m
[33m0;5;r_unit;;{sigma=inf, readA=0, readB=0}[0m
[33m0;6;e_unit;;{sigma=inf, opcode=, A=, B=, dest=, result=, memAddr=, store=, wr=0}[0m
[33m0;7;d_unit;;{sigma: inf, instruction: 0000000000000000000}[0m
[33m0;8;i_unit;;{sigma: 0.1, pc: 0, ir: 0000000000000}[0m
[i_unit]: instruction sent to d_unit: 0000000000000
[33m0.1;3;instructionMemory;;{sigma: 0.1, pc: 0}[0m
[d_unit]: instruction received from i_unit: 0000000000000
[d_unit]: opcode is 0000
[33m0.1;7;d_unit;;{sigma: 0, instruction: 0000000000000}[0m
[32m0.1;8;i_unit;pc;0[0m
[32m0.1;8;i_unit;ir;0000000000000[0m
[33m0.1;8;i_unit;;{sigma: inf, pc: 0, ir: 0000000000000}[0m
[33m0.1;5;r_unit;;{sigma=0.01, readA=0, readB=0}[0m
[e_unit]: opcode is: 0000
[e_unit]: destination is: 000
[33m0.1;6;e_unit;;{sigma=0.02, opcode=0000, A=, B=, dest=000, result=, memAddr=, store=, wr=0}[0m
[32m0.1;7;d_unit;opcode;0000[0m
[32m0.1;7;d_unit;opnda;000[0m
[32m0.1;7;d_unit;opndb;000[0m
[32m0.1;7;d_unit;dest;000[0m
[33m0.1;7;d_unit;;{sigma: inf, instruction: 0000000000000}[0m
[33m0.11;5;r_unit;;{sigma=inf, readA=0, readB=0}[0m
[r_unit]: destination register: 000
[33m0.12;5;r_unit;;{sigma=inf, readA=0, readB=0}[0m
[32m0.12;6;e_unit;destReg;000[0m
[33m0.12;6;e_unit;;{sigma=inf, opcode=0000, A=, B=, dest=000, result=, memAddr=, store=, wr=0}[0m
[i_mem]: instruction sent to i_unit: 0001001010011
[32m0.2;3;instructionMemory;instruction;0001001010011[0m
[33m0.2;3;instructionMemory;;{sigma: inf, pc: 0}[0m
[i_unit]: instruction received from i_mem: 0001001010011
[33m0.2;8;i_unit;;{sigma: 0, pc: 1, ir: 0001001010011}[0m
[i_unit]: instruction sent to d_unit: 0001001010011
[33m0.2;3;instructionMemory;;{sigma: 0.1, pc: 1}[0m
[d_unit]: instruction received from i_unit: 0001001010011
[d_unit]: opcode is 0001
[33m0.2;7;d_unit;;{sigma: 0, instruction: 0001001010011}[0m
[32m0.2;8;i_unit;pc;1[0m
[32m0.2;8;i_unit;ir;0001001010011[0m
[33m0.2;8;i_unit;;{sigma: inf, pc: 1, ir: 0001001010011}[0m
[33m0.2;5;r_unit;;{sigma=0.01, readA=1, readB=2}[0m
[e_unit]: opcode is: 0001
[e_unit]: destination is: 011
[33m0.2;6;e_unit;;{sigma=0.02, opcode=0001, A=, B=, dest=011, result=, memAddr=, store=, wr=0}[0m
[32m0.2;7;d_unit;opcode;0001[0m
[32m0.2;7;d_unit;opnda;001[0m
[32m0.2;7;d_unit;opndb;010[0m
[32m0.2;7;d_unit;dest;011[0m
[33m0.2;7;d_unit;;{sigma: inf, instruction: 0001001010011}[0m
[reg_unit] out A=00100010  B=01000100
[32m0.21;5;r_unit;oprnd_a;00100010[0m
[32m0.21;5;r_unit;oprnd_b;01000100[0m
```

```
[i_mem]: instruction sent to i_unit: 0010001010011
[32m0.3;3;instructionMemory;instruction;0010001010011[0m
[33m0.3;3;instructionMemory;;{sigma: inf, pc: 1}[0m
[i_unit]: instruction received from i_mem: 0010001010011
[33m0.3;8;i_unit;;{sigma: 0, pc: 2, ir: 0010001010011}[0m
[i_unit]: instruction sent to d_unit: 0010001010011
[33m0.3;3;instructionMemory;;{sigma: 0.1, pc: 2}[0m
[d_unit]: instruction received from i_unit: 0010001010011
[d_unit]: opcode is 0010
[33m0.3;7;d_unit;;{sigma: 0, instruction: 0010001010011}[0m
[32m0.3;8;i_unit;pc;2[0m
[32m0.3;8;i_unit;ir;0010001010011[0m
[33m0.3;8;i_unit;;{sigma: inf, pc: 2, ir: 0010001010011}[0m
[33m0.3;5;r_unit;;{sigma=0.01, readA=1, readB=2}[0m
[e_unit]: opcode is: 0010
[e_unit]: destination is: 011
[33m0.3;6;e_unit;;{sigma=0.02, opcode=0010, A=00100010, B=01000100, dest=011, result=01100110, memAddr=, store=, wr=0}[0m
[32m0.3;7;d_unit;opcode;0010[0m
[32m0.3;7;d_unit;opnda;001[0m
[32m0.3;7;d_unit;opndb;010[0m
[32m0.3;7;d_unit;dest;011[0m
[33m0.3;7;d_unit;;{sigma: inf, instruction: 0010001010011}[0m
[reg_unit] out A=00100010  B=01000100
[32m0.31;5;r_unit;oprnd_a;00100010[0m
[32m0.31;5;r_unit;oprnd_b;01000100[0m
[33m0.31;5;r_unit;;{sigma=inf, readA=1, readB=2}[0m
[e_unit]: opnda: 00100010
[e_unit]: opndb: 01000100
[e_unit]: result is: 11011110
[33m0.31;6;e_unit;;{sigma=0.02, opcode=0010, A=00100010, B=01000100, dest=011, result=11011110, memAddr=, store=, wr=0}[0m
[r_unit]: destination register: 011
[r_unit]: dataIn is: 11011110
[r_unit]: registers: 11011110
[33m0.33;5;r_unit;;{sigma=inf, readA=1, readB=2}[0m
[32m0.33;6;e_unit;destReg;011[0m
[32m0.33;6;e_unit;result;11011110[0m
[33m0.33;6;e_unit;;{sigma=inf, opcode=0010, A=00100010, B=01000100, dest=011, result=11011110, memAddr=, store=, wr=0}[0m
[i_mem]: instruction sent to i_unit: 0011010011100
[32m0.4;3;instructionMemory;instruction;0011010011100[0m
[33m0.4;3;instructionMemory;;{sigma: inf, pc: 2}[0m
[i_unit]: instruction received from i_mem: 0011010011100
[33m0.4;8;i_unit;;{sigma: 0, pc: 3, ir: 0011010011100}[0m
[i_unit]: instruction sent to d_unit: 0011010011100
[33m0.4;3;instructionMemory;;{sigma: 0.1, pc: 3}[0m
[d_unit]: instruction received from i_unit: 0011010011100
[d_unit]: opcode is 0011
[33m0.4;7;d_unit;;{sigma: 0, instruction: 0011010011100}[0m
[32m0.4;8;i_unit;pc;3[0m
```

```
[i_mem]: instruction sent to i_unit: 1110000010001
[32m0.7;3;instructionMemory;instruction;1110000010001[0m
[33m0.7;3;instructionMemory;;{sigma: inf, pc: 5}[0m
[i_unit]: instruction received from i_mem: 1110000010001
[33m0.7;8;i_unit;;{sigma: 0, pc: 6, ir: 1110000010001}[0m
[i_unit]: instruction sent to d_unit: 1110000010001
[33m0.7;3;instructionMemory;;{sigma: 0.1, pc: 6}[0m
[d_unit]: instruction received from i_unit: 1110000010001
[d_unit]: opcode is 1110
[d_unit]: ld instruction
[d_unit]: dataMemAddr is 0001
[d_unit]: ldReg is 001
[33m0.7;7;d_unit;;{sigma: 0, instruction: 1110000010001}[0m
[32m0.7;8;i_unit;pc;6[0m
[32m0.7;8;i_unit;ir;1110000010001[0m
[33m0.7;8;i_unit;;{sigma: inf, pc: 6, ir: 1110000010001}[0m
[33m0.7;2;dataMemory;;{1}[0m
[e_unit]: opcode is: 1110
[33m0.7;6;e_unit;;{sigma=inf, opcode=1110, A=01000100, B=11011110, dest=110, result=10011010, memAddr=, store=, wr=0}[0m
[32m0.7;7;d_unit;dataMemAddrLd;0001[0m
[32m0.7;7;d_unit;opcode;1110[0m
[32m0.7;7;d_unit;ldReg;001[0m
[33m0.7;7;d_unit;;{sigma: inf, instruction: 1110000010001}[0m
[i_mem]: instruction sent to i_unit: 1111000010010
[32m0.8;3;instructionMemory;instruction;1111000010010[0m
[33m0.8;3;instructionMemory;;{sigma: inf, pc: 6}[0m
[i_unit]: instruction received from i_mem: 1111000010010
[33m0.8;8;i_unit;;{sigma: 0, pc: 7, ir: 1111000010010}[0m
[i_unit]: instruction sent to d_unit: 1111000010010
[33m0.8;3;instructionMemory;;{sigma: 0.1, pc: 7}[0m
[d_unit]: instruction received from i_unit: 1111000010010
[d_unit]: opcode is 1111
[d_unit]: st instruction
[33m0.8;7;d_unit;;{sigma: 0, instruction: 1111000010010}[0m
[32m0.8;8;i_unit;pc;7[0m
[32m0.8;8;i_unit;ir;1111000010010[0m
```

```
[33m0.8;8;i_unit;;{sigma: inf, pc: 7, ir: 1111000010010}[0m
[33m0.8;5;r_unit;;{sigma=0.01, readA=4, readB=5}[0m
[e_unit]: opcode is: 1111
[33m0.8;6;e_unit;;{sigma=inf, opcode=1111, A=01000100, B=11011110, dest=110, result=10011010, memAddr=, store=, wr=0}[0m
[32m0.8;7;d_unit;dataMemAddrSt;0010[0m
[32m0.8;7;d_unit;opcode;1111[0m
[32m0.8;7;d_unit;stReg;001[0m
[33m0.8;7;d_unit;;{sigma: inf, instruction: 1111000010010}[0m
[reg_unit] out A=01000100  B=11011110
[d_mem]: data stored: 00100010
[33m0.81;2;dataMemory;;{0}[0m
[32m0.81;5;r_unit;oprnd_a;01000100[0m
[32m0.81;5;r_unit;oprnd_b;11011110[0m
[32m0.81;5;r_unit;stAddrOut;0010[0m
[32m0.81;5;r_unit;stData;00100010[0m
[33m0.81;5;r_unit;;{sigma=inf, readA=4, readB=5}[0m
[33m0.81;6;e_unit;;{sigma=inf, opcode=1111, A=01000100, B=11011110, dest=110, result=10011010, memAddr=, store=, wr=0}[0m
[33m0.81;2;dataMemory;;{inf}[0m
[i_mem]: instruction sent to i_unit: 0001010101111
[32m0.9;3;instructionMemory;instruction;0001010101111[0m
[33m0.9;3;instructionMemory;;{sigma: inf, pc: 7}[0m
[i_unit]: instruction received from i_mem: 0001010101111
[33m0.9;8;i_unit;;{sigma: 0, pc: 8, ir: 0001010101111}[0m
[i_unit]: instruction sent to d_unit: 0001010101111
[33m0.9;3;instructionMemory;;{sigma: 0.1, pc: 8}[0m
[d_unit]: instruction received from i_unit: 0001010101111
[d_unit]: opcode is 0001
[33m0.9;7;d_unit;;{sigma: 0, instruction: 0001010101111}[0m
[32m0.9;8;i_unit;pc;8[0m
[32m0.9;8;i_unit;ir;0001010101111[0m
[33m0.9;8;i_unit;;{sigma: inf, pc: 8, ir: 0001010101111}[0m
[33m0.9;5;r_unit;;{sigma=0.01, readA=2, readB=5}[0m
[e_unit]: opcode is: 0001
[e_unit]: destination is: 111
```

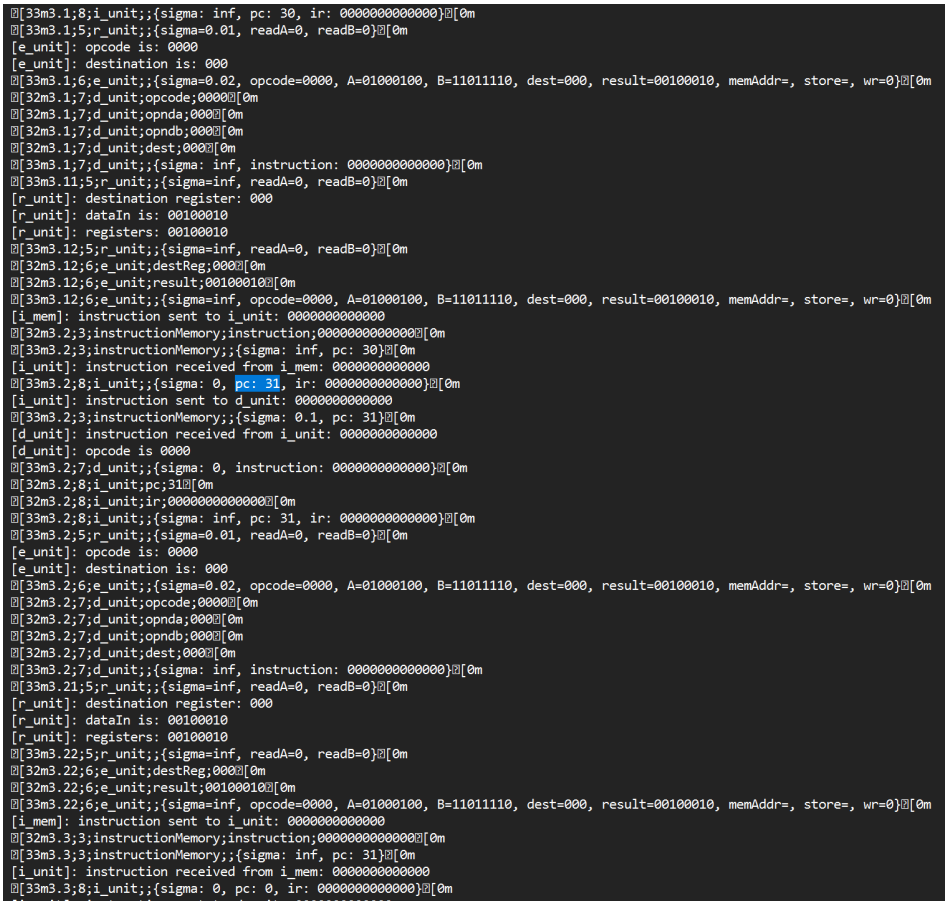

Figure 17: RISC System Coupled Model Testing

Throughout the simulation, instructions successfully moved through the four pipeline stages — Fetch, Decode, Execute, and Memory Access/Write Back. Registers and memory were updated at the appropriate cycles, and no data corruption or synchronization issues were observed.

V. CONCLUSION

In conclusion, the simulation of the pipelined RISC processor coupled with the Main Memory demonstrates the correct execution of a sequence of instructions within the DEVS-based modeling framework. The system successfully performed instruction fetch, decode, execute, and memory access/write-back operations while maintaining pipeline timing and register consistency. The simulation results showed that the processor behaved as expected across a variety of arithmetic and memory operations, with no synchronization issues.

This validates the effectivness of using DEVS modeling and simulation to design and test processor architectures before hardware implementation. Minor design changes, such as instruction set variations or memory configurations, could be easily implemented within this modular framework. In future work, more complex features such as branch handling, hazard detection, and forwarding mechanisms can be incorporated to observe their impact and enhance performance. Additionally, extending the simulation to cover larger instruction sequences and varying memory access patterns would help further validate processor behavior under a broader set of operational scenarios.

VI. REFERENCES

[1] “Cadmium – ARSLab – Simulation Tools,” *Carleton.ca*, 2025. Available: https://cell-devs.sce.carleton.ca/index.php/cadmium/.
[2] G. A. Wainer, *Discrete-event modeling and simulation : a practitioner’s approach*. Crc Press, 2017.
[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: a Quantitative Approach*. Cambridge, Ma: Morgan Kaufmann, 2019.
[4] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*. Elsevier, 2004.