

# Behavioral Modeling of DaVinci v1.0 with CS147DV Instruction Set

Chihkuo Hsu

San Jose State University

[chih-kuo.hsu@sjsu.edu](mailto:chih-kuo.hsu@sjsu.edu)

**Abstract**— This report outlines the design, implementation, and testing of DaVinci v1.0, a minimal computer system supporting the CS147DV instruction set. The system integrates a 32-bit processor with 256 MB of memory, including an Arithmetic Logic Unit (ALU), Register File (RF), and Control Unit (CU). Using Verilog, the components were modeled and validated through testbenches. The system successfully executed small programs in the CS147DV instruction set, demonstrating its functionality and efficiency.

## 1. INTRODUCTION

The HDL simulator allows for the design and simulation of digital circuits using Hardware Description Language (HDL). In this project, Verilog is utilized to develop the DaVinci v1.0 computer system, which supports the CS147DV instruction set. ModelSim serves as the simulation tool for testing and verification. The project involves setting up the simulation environment, completing the core functionalities of the ALU, Register File, and Control Unit, implementing R-type, I-type, and J-type instructions, and performing comprehensive testing of the entire system to ensure functionality and accuracy.

1. Setting up the provided project files, configuring the simulation tool (ModelSim), and preparing the environment.
2. Implementing the core functionalities of `alu.v`, `register_file.v`, and `control_unit.v`.
3. Implementing and verifying the functionality of R-type, I-type, and J-type instructions.
4. Performing comprehensive testing of the entire DaVinci v1.0 system.

## 2. MODELSIM INSTALLATION AND SETUP

### 1. Installation

Go to the following website

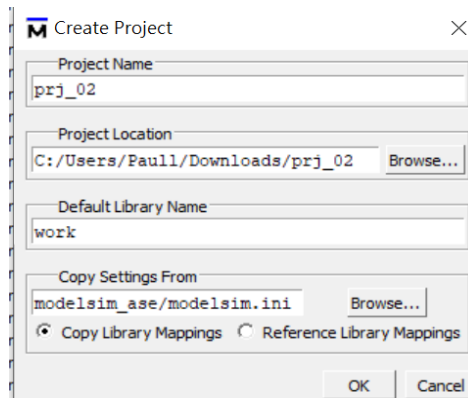
URL: <https://www.intel.com/content/www/us/en/software-kit/750368/modelsim-intel-fpgas-standard-edition-software-version-18-1.html>

Download ModelSim-Intel® FPGA Edition (includes Starter Edition)

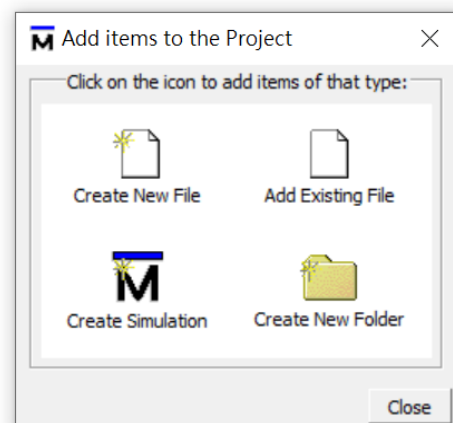
Once the download is complete, open the file, agree to all the agreements, and install.

### 2. Setup

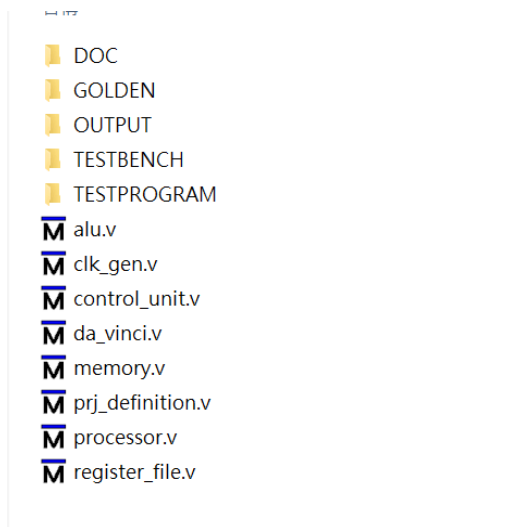
After opening ModelSim, click "New", then select "Project" and choose "Create Project". Enter your project name, click "Browse" to select your project location, and finally, click "OK".



The following screen will appear, select "Add Existing Files".



Select your source code and add all of them.



### 3. REQUIREMENTS FOR DAVINCI v1.0

The DaVinci v1.0 system requires a processor and memory module to operate effectively. The processor is composed of three key components: the Arithmetic Logic Unit (ALU), Register File (RF), and Control Unit (CU). The system functions by transferring input operands from the register file to the ALU for computation. After the operation is completed, the output is returned to the register file and subsequently written into memory. When data retrieval is necessary, the memory provides the requested data back to the register file for processing.

#### A. Arithmetic and Logic Unit

The ALU serves as the computational core of the processor, executing essential arithmetic and logical operations required by the system. In this project, the ALU supports nine distinct operations specified in the CS147DV instruction set, as well as a ZERO flag that indicates whether the output result equals zero. The ALU is a critical component that facilitates the system's computational functionality and will undergo thorough testing for correctness.

#### B. Register File

The register file is a 32x32 array of processor registers, enabling efficient data storage and retrieval. It features a single input data bus and two output data buses dedicated to read and write operations. The register file supports 32-bit

data processing and includes a reset mechanism triggered on the negative edge of the clock signal. During the positive edge of the clock, read operations are initiated, while write operations occur based on specific conditions. Invalid states (00 OR 11) ensure previously read data remains unchanged, providing stability. Proper functionality is verified through a dedicated testbench.

#### C. Memory

The memory module shares similar characteristics with the register file but utilizes a bidirectional data bus for its operations. This project implements a 64MB memory capable of handling 32-bit words. Unlike the register file, memory operations are executed during the positive clock edge, while reset functions occur on the negative edge. Invalid states (00 or 11) transition the memory to a high-impedance state, ensuring no unintended operations are performed. The memory is integral to the system's data storage and retrieval capabilities.

#### D. Control Unit

The Control Unit manages signal flow within the processor and coordinates operations between the ALU, register file, and memory. It operates as a state machine with five key states:

1. **Instruction Fetch (IF):** Retrieves instructions from memory.
2. **Instruction Decode (ID):** Decodes the fetched instruction into meaningful components.
3. **Execution (EXE):** Performs the designated operation using the ALU.
4. **Memory (MEM):** Handles read/write operations for memory access.
5. **Write Back (WB):** Writes results back to the register file or updates the program counter.

#### E. Instruction Set

The CS147DV instruction set consists of three types:

1. **R-Type (Register Type):** Involves two source registers (rs,rt) for operations and a destination register (rd) for storing results. Each instruction contains 6 bits for the opcode, shift amount, and function.
2. **I-Type (Immediate Type):** Contains two registers (rs,rt) and a 16-bit immediate value. The result is stored in one of the registers.
3. **J-Type (Jump Type):** Includes a 26-bit address used for jump operations.

This instruction set is fully supported by the DaVinci v1.0 system and enables it to handle a variety of computational tasks.

#### 4. THE DESIGN AND IMPLEMENTATION OF ARITHMETIC AND LOGIC UNIT (ALU)

The Arithmetic and Logic Unit (ALU) is a fundamental component of the DaVinci v1.0 processor, designed to execute both arithmetic and logical operations. Its primary role in the system is to process computations based on the CS147DV instruction set, providing essential functionality for the processor's overall operation. In this project, the ALU was developed using Verilog to support a total of nine operations alongside an extended ZERO flag, which determines whether the result of an operation equals zero.

##### A. ALU Functional Requirements

The ALU's design was guided by the following functional requirements:

1. Supported Operations:
  1. Arithmetic operations such as addition, subtraction, and multiplication.
  2. Logical operations such as AND, OR, NOR, and SLT (set less than).
  3. Bitwise operations like shift left logical (SLL) and shift right logical (SRL).
  4. Each operation is mapped to an opcode defined in the CS147DV instruction set
2. Zero Flag:
  1. Arithmetic operations such as addition, subtraction, and multiplication.

##### B. ALU Implementation

The ALU was implemented with careful consideration of its inputs, outputs, and integration with the processor. It accepts two input operands (operand1 and operand2), an opcode specifying the operation, and control signals from the Control Unit. The output consists of the computed result and a ZERO flag, which indicates whether the result is zero. Registers were included to store intermediate values for the output and the ZERO flag, ensuring consistent performance and seamless integration with other processor components. The ALU decodes the opcode to determine the operation to execute, using conditional statements or case structures for efficiency. It is tightly integrated with the processor, connecting to the Control Unit, Register File, and Memory to process data according to the instruction cycle. Operating synchronously with the processor's clock signal, the ALU maintains reliable and consistent functionality across all instructions.

```
module ALU(
    OUT,
    ZERO,
    OP1,
    OP2,
    OPRN
);
// Input list
input [DATA_INDEX_LIMIT:0] OP1; // operand 1
input [DATA_INDEX_LIMIT:0] OP2; // operand 2
input [ALU_OPRN_INDEX_LIMIT:0] OPRN; // operation code

// Output list
output reg [DATA_INDEX_LIMIT:0] OUT; // result of the operation
output reg ZERO; // Zero flag: 1 if OUT is 0, other

// Always block for combinational ALU operations
always @(OP1 or OP2 or OPRN) begin
    case (OPRN)
        'ALU_OP_ADD : OUT = OP1 + OP2; // addition
        'ALU_OP_SUB : OUT = OP1 - OP2; // subtraction
        'ALU_OP_MUL : OUT = OP1 * OP2; // multiplication
        'ALU_OP_SHR : OUT = OP1 >> OP2; // shift right
        'ALU_OP_SHL : OUT = OP1 << OP2; // shift left
        'ALU_OP_AND : OUT = OP1 & OP2; // bitwise AND
        'ALU_OP_OR : OUT = OP1 | OP2; // bitwise OR
        'ALU_OP_NOR : OUT = ~(OP1 | OP2); // bitwise NOR
        'ALU_OP_SLT : OUT = (OP1 < OP2) ? 1 : 0; // set less than
        default : OUT = {DATA_WIDTH[1'bx]}; // undefined operation
    endcase

    // Set the ZERO flag
    ZERO = (OUT == 0) ? 1'b1 : 1'b0;
end
endmodule
```

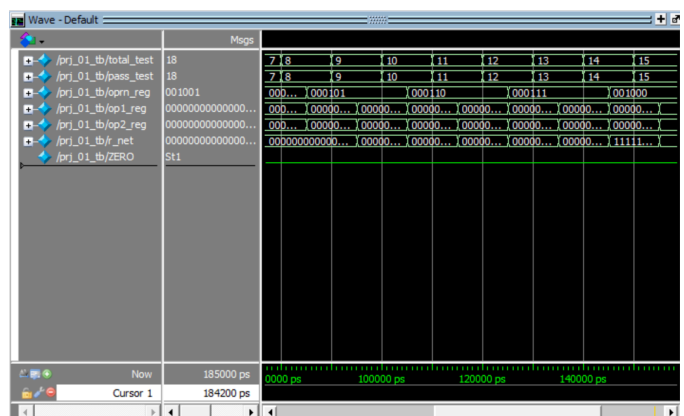
##### C. ALU testing

To verify the functionality of the implementation, a testbench file prj\_01\_tb.v from project 1 will be used.

```
# [TEST] 10 << 1 = 20 , got 20 ... [PASSED]
# [TEST] 15 & 3 = 3 , got 3 ... [PASSED]
# [TEST] 12 & 5 = 4 , got 4 ... [PASSED]
# [TEST] 15 | 3 = 15 , got 15 ... [PASSED]
# [TEST] 12 | 5 = 13 , got 13 ... [PASSED]
# [TEST] 15 NOR 3 = 4294967280 , got 4294967280 ... [PASSED]
# [TEST] 10 NOR 2 = 4294967285 , got 4294967285 ... [PASSED]
# [TEST] 5 < 10 = 1 , got 1 ... [PASSED]
# [TEST] 10 < 5 = 0 , got 0 ... [PASSED]

# Total number of tests: 18
# Total number of pass: 18

** Note: $stop : C:/Users/Paul1/Downloads/prj_02/prj_01_tb.v(189)
Time: 185 ns Iteration: 0 Instance: /prj_01_tb
```



## 5. THE DESIGN AND IMPLEMENTATION OF REGISTER FILE

### A. Register File Design

The Register File (RF) is a critical component in the processor, providing a small, high-speed storage area that facilitates data exchange between the processor's registers and instructions. In this project, a 32x32 register file is implemented, supporting dual-read and single-write operations for 32-bit data. The RF includes 32 registers, with two dedicated output buses, DATA\_R1 and DATA\_R2, for read operations, and one input bus, DATA\_W, for write operations. The addresses for these operations are managed using ADDR\_R1, ADDR\_R2, and ADDR\_W.

The RF's functionality is governed by control signals READ and WRITE. When READ is set to 1 and WRITE to 0, the data at the addresses specified by ADDR\_R1 and ADDR\_R2 are output through DATA\_R1 and DATA\_R2. Conversely, when READ is 0 and WRITE is 1, data from DATA\_W is written to the address specified by ADDR\_W. If both READ and WRITE are either 0 or 1, the outputs remain idle, with values set to X. To ensure consistency and a clean state, a reset mechanism initializes all registers to zero when the RST signal transitions to its negative edge. All read and write operations are synchronized to the positive edge of the clock signal (CLK).

Unlike bidirectional bus designs, this RF employs separate input and output buses, eliminating data contention and removing the need for Hi-Z states. Logic statements control the read and write operations, ensuring only valid actions are executed. Additionally, an initial block sets all registers to zero at the beginning of the simulation. This design ensures seamless integration of the Register File into the processor, supporting fast and efficient data access for computation and instruction execution.

### B. Register File Testing

The testing of the Register File (RF) is carried out using the RF\_TB testbench to validate its functionality, ensuring that the read and write operations are performed correctly.

Initialization:

The testbench initializes all control signals, including READ, WRITE, and RST, as well as the data signal DATA\_REG and address signals ADDR\_R1 and ADDR\_R2. A reset sequence is applied to guarantee the RF begins in a consistent state.

Write Cycle:

Data is written into the 32 registers using a loop. The DATA\_REG is updated sequentially for each address specified by ADDR\_W, and the WRITE signal is activated. This verifies that each register can store the intended value.

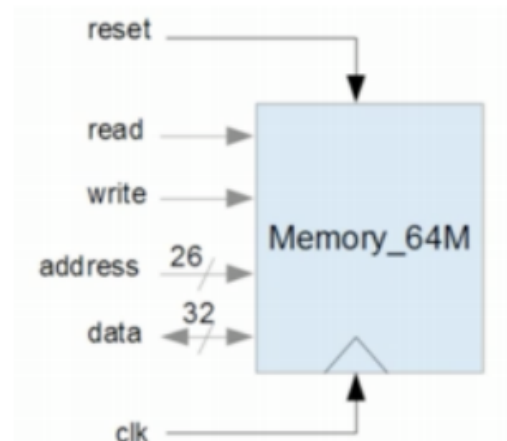
Read Cycle:

After completing the write operations, the testbench enters a read phase. Data is read back from the registers using addresses ADDR\_R1 and ADDR\_R2 with the READ signal enabled. The values retrieved (DATA\_R1 and DATA\_R2) are compared with the expected data to validate accuracy.

```
sim:/RF_TB/no_of_pass \
sim:/RF_TB/load_data \
sim:/RF_TB/CLK \
sim:/RF_TB/DATA_R1 \
sim:/RF_TB/DATA_R2
VSI9>run -all
#
#      Total number of tests      32
#      Total number of pass      32
#
# ** Note: Stop    : C:/Users/Paul1/Downloads/prj_02/TESTBENCH/register_file_tb.v(88)
# Time: 680 ns Iteration: 0 Instance: /RF_TB
```

## 6. THE DESIGN AND IMPLEMENTATION OF MEMORY

### A. Memory Design



The memory module, as part of the processor system, is implemented using Verilog to simulate a 64MB memory model with 32-bit word accessibility. This module facilitates data storage and retrieval during the execution of instructions within the processor.

The memory module includes a parameterized memory bank `sram_32x64m` for storing data. Its functionality is determined by the `READ` and `WRITE` control signals. A read operation occurs when `READ` is asserted, and `WRITE` is deasserted, allowing data to be fetched from the memory at the specified address (`ADDR`). Conversely, a write operation happens when `WRITE` is asserted, and `READ` is deasserted, enabling data to be written to the memory. If both signals are inactive (`READ=0`, `WRITE=0`), or both are active (`READ=1`, `WRITE=1`), the `DATA` bus is set to a high-impedance state (`X`).

The memory module operates synchronously with the clock signal (`CLK`), while reset functionality is triggered by the negative edge of the `RST` signal. Upon reset, the memory is cleared, and its contents are initialized using an external data file (`mem_content_01.dat`). This ensures consistency and facilitates testing with preloaded data.

To test the memory module's operations, the `mem_64MB_tb.v` testbench simulates read and write cycles under various scenarios, verifying the correct behavior of the module.

#### B. Memory Testing

The memory testing framework validates the functionality of the `MEMORY_64MB` module, focusing on reset, write, read, and high-impedance behaviors.

##### 1. Objectives:

1. Reset: Ensures memory initializes correctly and loads data from the initialization file.

2. Write/Read Operations: Confirms data integrity during write and read cycles.

3. Hi-Z State: Verifies proper behavior when no operations are active.

##### 2. Implementation:

The testbench uses control signals (`READ`, `WRITE`, `RST`, `CLK`) to perform operations on the memory. Data and address buses (`DATA`, `ADDR`) manage data flow.

1. Reset Phase: Clears memory and loads predefined content.

2. Write Phase: Sequentially writes data to memory addresses

3. Read Phase: Retrieves and validates data from the memory.

4. Initialization Test: Checks preloaded memory contents.

```

Transcript
sim:/MEM_64MB_TB/01_01.v
sim:/MEM_64MB_TB/no_of_test \
sim:/MEM_64MB_TB/no_of_pass \
sim:/MEM_64MB_TB/load_data \
sim:/MEM_64MB_TB/CLK \
sim:/MEM_64MB_TB/DATA
V$IM 12> run -all
#
#      Total number of tests      27
#      Total number of pass      27
#
# ** Note: $stop      : C:/Users/Paul/Downloads/prj_02/TESTBENCH/mem_64MB_tb.v(107)
#      Time: 405 ns  Iteration: 0  Instance: /MEM_64MB_TB

```

## 7. THE DESIGN AND IMPLEMENTATION OF PROCESSOR

### A. Core Modules of the Processor.

#### 1. ALU (Arithmetic Logic Unit):

The ALU handles arithmetic operations such as addition (`add`), subtraction (`sub`), and multiplication (`mul`), as well as logical operations like `AND`, `OR`, `NOR`, and `SLT` (set less than).

It includes a `ZERO` flag that identifies whether the result of an operation is zero. This flag is crucial for conditional instructions such as `beq` (branch if equal) and `bne` (branch if not equal).

The ALU was adapted from a prior implementation, with changes to support additional operations and to ensure compatibility with the updated instruction set.

Example: For an R-Type `add` instruction, the ALU takes inputs from the Register File and produces a result that is written back to the destination register.

#### 2. Register File (RF):

The Register File is a 32x32 dual-read, single-write register bank. It supports three address ports: `ADDR_R1`, `ADDR_R2` (for reading) and `ADDR_W` (for writing).

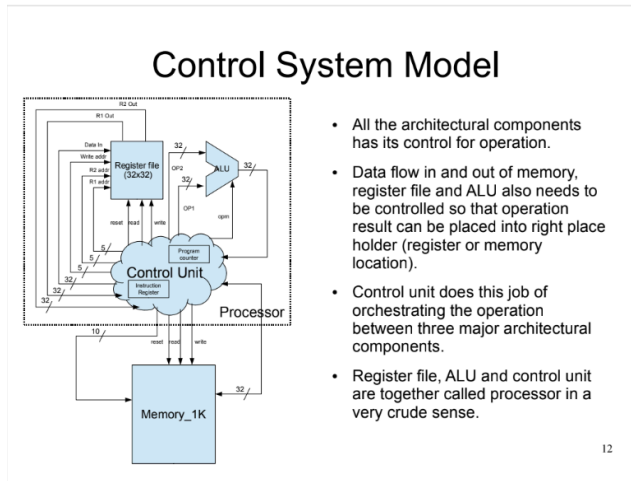
Initial challenges included variable naming conflicts when assigning output values to `DATA_R1` and `DATA_R2`. This was resolved by using unique identifiers for return values.



The RF is reset on the negative edge of the RST signal and initializes all registers to zero.

Example: For an I-Type addi instruction, the RF provides source register data (rs), and the ALU calculates the result which is written to the destination register (rt).

### 3. Control Unit (CU):



The Control Unit (CU) coordinates data flow within the processor, managing interactions among the ALU, the Register File (RF), and Memory. It generates control signals to ensure data is transferred correctly based on the instruction requirements.

### 1. Instruction Fetch (IF):

- Fetches the instruction from the address specified by the Program Counter (PC).
- Stores the fetched instruction in INST\_REG for decoding in the next stage.

### 2. Decode (ID):

Decodes the fetched instruction into components: opcode, operands, and immediate values.

Handles:

R-Type instructions (e.g., add, sub): Utilize three register fields (rs, rt, rd) and perform operations using the ALU.

I-Type instructions (e.g., addi, lw, sw): Use two register fields (rs, rt) and an immediate value for operations.

J-Type instructions (e.g., jmp, jal): Rely on a target address for program flow changes.

### 3. Execute (EXE):

- Sends operands from the RF to the ALU.
- ALU performs the computation based on the instruction (e.g., arithmetic, logical, or comparison operations).
- For branch or jump instructions, the CU evaluates conditions (e.g., the ZERO flag) to determine the next instruction address.

### 4. Memory Access (MEM):

Executes memory-related instructions:

- lw (load word): Reads data from memory into a register.
- sw (store word): Writes data from a register into memory.
- push/pop: Manages stack operations.

### 5. Write Back (WB):

- Writes the computation result or memory data back to the RF.
- Updates the PC for the next instruction.

```
// output assignment
assign STATE = current_state;
// initialization block: Sets default values for state registers at simulation start
initial begin
    current_state = 3'bxxx;
    next_state = 'PROC_FETCH;
end

// reset block: Resets state machine when RST goes low
always @(negedge RST) begin
    current_state = 3'bxxx; // Reset to undefined state
    next_state = 'PROC_FETCH; // Reset to FETCH state
end

// handles state updates on the rising edge of CLK
always @(posedge CLK) begin
    current_state = next_state; // Update current state

    // transition to the next state based on the current state
    case (current_state)
        'PROC_FETCH: next_state = 'PROC_DECODE; // Fetch to Decode
        'PROC_DECODE: next_state = 'PROC_EXE; // Decode to Execute
        'PROC_EXE: next_state = 'PROC_MEM; // Execute to Memory
        'PROC_MEM: next_state = 'PROC_WB; // Memory to Write-back
        'PROC_WB: next_state = 'PROC_FETCH; // Write-back to Fetch
        default: next_state = 'PROC_FETCH; // Default to Fetch for safety
    endcase
end
endmodule
```

Example:For a beq (branch if equal) instruction:

The CU examines the ZERO flag from the ALU.If the ZERO flag is set, it updates the PC to branch to the target address specified in the instruction. Otherwise, it proceeds to the next sequential instruction.

B. Processor Testing

The Processor Testing phase ensures that all integrated components of the processor work together as expected. This includes verifying the Control Unit (CU), Register File (RF), Arithmetic Logic Unit (ALU), and memory interactions.

Key Testing Steps:

1. Integration Testing:

- 1. Test the functionality of the processor using multiple test programs stored in memory (e.g., Fibonacci sequence, Reverse Fibonacci).
- 2. The processor executes the instructions, and results are validated against expected outputs.

2. State Transitions:

- 1. Validate the state machine transitions in the control unit (PROC\_FETCH, PROC\_DECODE, PROC\_EXE, PROC\_MEM, PROC\_WB) using specific test instructions.
- 2. Ensure proper behavior for each type of instruction (R-Type, I-Type, J-Type).
- 3.

3. ALU Testing:

- 1. Verify that the ALU correctly performs arithmetic and logical operations, including edge cases where the ZERO flag is activated.

4. Register File Testing:

- 1. Test dual-read and single-write operations.
- 2. Confirm that the RF handles simultaneous reads and writes correctly without conflicts.

5. Memory Testing:

- 1. Validate memory read and write operations for instructions like lw (load word) and sw (store word).
- 2. Ensure stack operations (e.g., push, pop) function as specified.

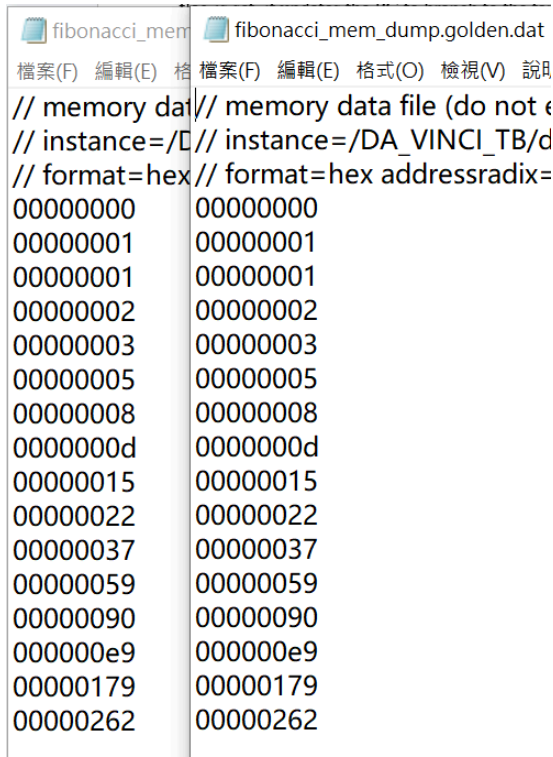
Test Programs:

The testbench (da\_vinci\_tb.v) runs multiple test programs, each designed to cover a specific aspect of the processor’s functionality.

1. fibonacci\_mem\_dump.dat

Purpose: Captures the memory state after running a Fibonacci sequence program.

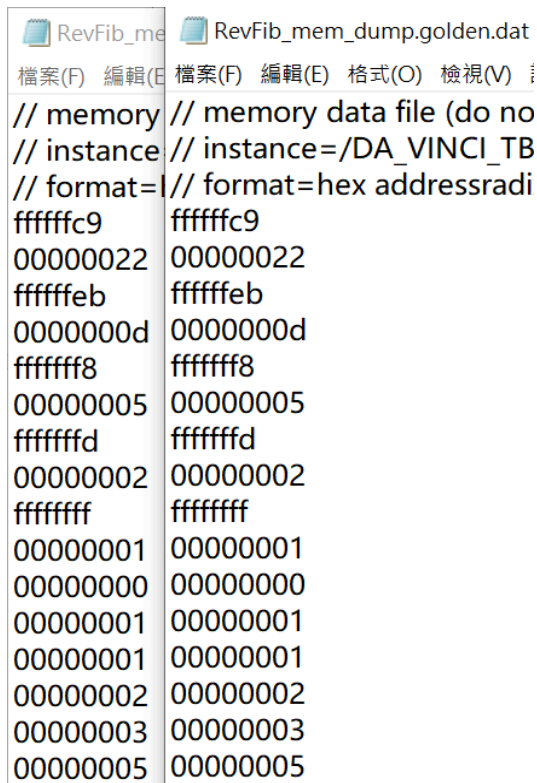
Usage: Verifies arithmetic operations (e.g., addi) and memory instructions (e.g., sw, lw). Ensures correct results are written to memory.



2. RevFib\_mem\_dump.dat

Purpose: Stores memory results from a program that computes the reverse Fibonacci sequence.

Usage: Confirms the processor's capability to handle reverse computations and write correct data to memory.



```

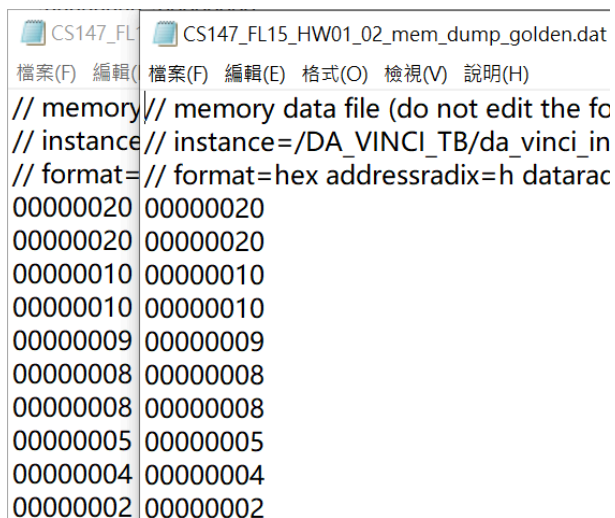
RevFib_mem_dump.golden.dat
檔案(F) 編輯(E) 格式(O) 檢視(V)
// memory // memory data file (do not edit the following)
// instance // instance=/DA_VINCI_TB/da_vinci_inst/
// format=hex addressradix=h dataradix=h
fffffc9 fffffc9
00000022 00000022
fffffeb fffffeb
0000000d 0000000d
fffff8 fffff8
00000005 00000005
fffffd fffffd
00000002 00000002
ffffff ffffff
00000001 00000001
00000000 00000000
00000001 00000001
00000001 00000001
00000002 00000002
00000003 00000003
00000005 00000005

```

### 3. CS147\_FL15\_HW01\_02\_mem\_dump.dat

Purpose: Contains memory data after executing a program designed to test stack operations (push, pop).

Usage: Evaluates stack pointer functionality (SP\_REF) and proper handling of stack-based memory instructions.



```

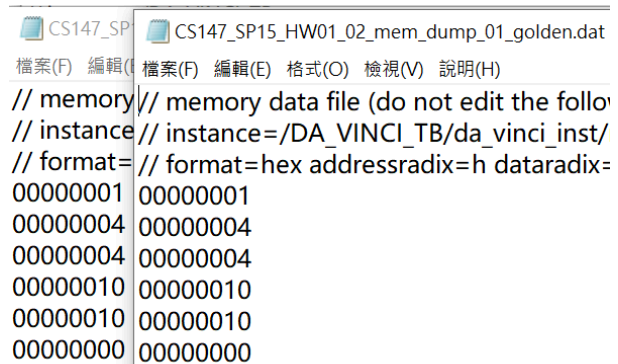
CS147_FL15_HW01_02_mem_dump.golden.dat
檔案(F) 編輯(E) 格式(O) 檢視(V) 說明(H)
// memory // memory data file (do not edit the following)
// instance // instance=/DA_VINCI_TB/da_vinci_inst/
// format=hex addressradix=h dataradix=h
00000020 00000020
00000020 00000020
00000010 00000010
00000010 00000010
00000009 00000009
00000008 00000008
00000008 00000008
00000005 00000005
00000004 00000004
00000002 00000002

```

### 4. CS147\_SP15\_HW01\_02\_mem\_dump\_01.dat

Purpose: Saves partial memory state from a Spring 2015 program, focusing on control flow (beq, bne).

Usage: Verifies that branch instructions modify the program counter (PC\_REG) as expected based on ALU flags (e.g., ZERO).



```

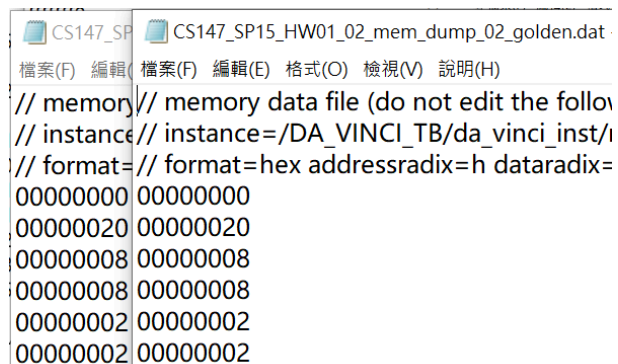
CS147_SP15_HW01_02_mem_dump_01.golden.dat
檔案(F) 編輯(E) 格式(O) 檢視(V) 說明(H)
// memory // memory data file (do not edit the following)
// instance // instance=/DA_VINCI_TB/da_vinci_inst/
// format=hex addressradix=h dataradix=h
00000001 00000001
00000004 00000004
00000004 00000004
00000010 00000010
00000010 00000010
00000000 00000000

```

### 5. CS147\_SP15\_HW01\_02\_mem\_dump\_02.dat

Purpose: Captures additional memory results for the same Spring 2015 program, emphasizing logical instructions (and, or).

Usage: Ensures logical operations are executed correctly and results are written to memory.



```

CS147_SP15_HW01_02_mem_dump_02.golden.dat
檔案(F) 編輯(E) 格式(O) 檢視(V) 說明(H)
// memory // memory data file (do not edit the following)
// instance // instance=/DA_VINCI_TB/da_vinci_inst/
// format=hex addressradix=h dataradix=h
00000000 00000000
00000020 00000020
00000008 00000008
00000008 00000008
00000002 00000002
00000002 00000002

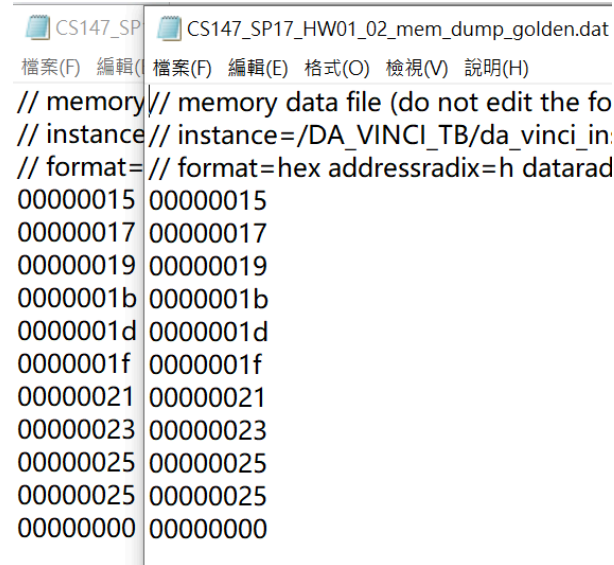
```

### 6. CS147\_SP17\_HW01\_02\_mem\_dump.dat

Purpose: Records memory state after executing a program that tests a wide range of arithmetic and control instructions.



Usage: Used for comprehensive testing of the processor, confirming correct interaction between the ALU, Register File, and Control Unit.



```
// memory data file (do not edit the fo
// instance=/DA_VINCI_TB/da_vinci_in
// format=hex addressradix=h data
00000015 00000015
00000017 00000017
00000019 00000019
0000001b 0000001b
0000001d 0000001d
0000001f 0000001f
00000021 00000021
00000023 00000023
00000025 00000025
00000025 00000025
00000000 00000000
```

Steps for Validation:

Compare each file in OUTPUT with its corresponding file in the GOLDEN directory. If discrepancies arise, create new test programs or modify existing ones to ensure all functionalities are validated.

## 6. CONCLUSION

The DaVinci v1.0 processor successfully integrates a 32-bit Arithmetic Logic Unit, Register File, Control Unit, and memory to support the CS147DV instruction set.

Through rigorous testing using various programs, including Fibonacci and stack operations, the system demonstrated accurate execution of R-Type, I-Type, and J-Type instructions. Completing this project provided invaluable experience in designing and implementing processor components, debugging complex systems, and validating functionality through comprehensive testing.

This work not only strengthened our understanding of processor design but also highlighted the importance of modular and scalable architectures for efficient computation, laying a solid foundation for future advancements in digital system development.

## REFERENCES

- [1] K. Patra, (2024, November). *Module Ten - Verilog Tutorial*, Canvas, San Jose State University.
- [2] K. Patra, (2024, November). *Lab 06: Discussion on Behavioral Modeling of a Processor*, Canvas, San Jose State University.

