CSE 332 - Computer Organization and Architecture

Project Report

Design Your Own MIPS CPU and OS

Course Instructor
Dr. Mohammad Abdul Qayum (MAQm)

Group 1

Team Members

Walidur Rahman	231 1712 642
Mridha Fahad Hossain	231 2199 042
Sabbir Ahmod	231 1501 042
Hasan Al Bannah	211 2290 642

Contents

1 Abstract

This project focuses on the design and implementation of a simplified 32-bit Single-Cycle MIPS CPU and a basic operating system in Verilog. We began with designing an Instruction Set Architecture (ISA), provided an assembler to translate assembly into machine code, wrote benchmark programs including an OS-like program for MIN, MAX, and MEAN, and finally enhanced the datapath and control logic of the CPU in Verilog. Simulation and benchmarking were performed to validate correctness and efficiency. The report discusses ISA design, assembler workflow, program testing, CPU implementation, simulation results, and challenges faced.

2 Introduction

2.1 Motivation

Due to time constraints, this project involved working with a nearly complete Single-Cycle MIPS CPU implementation provided by the faculty. Our responsibilities included understanding the existing system architecture, implementing missing JAL and JR instructions, and testing the overall functionality.

2.2 Objectives

- Understand the architecture and operation of the provided MIPS CPU implementation.
- Analyze existing datapath and control logic for **R-type**, **I-type**, and **J-type** instructions.
- Implement **JAL (Jump and Link)** and **JR (Jump Register)** instructions in the CPU.

2.3 Report Organization

The report follows a structured flow starting from ISA design, assembler, benchmark programs, Verilog CPU implementation, simulation, benchmarking, results, and concluding with discussions and future improvements.

3 Instruction Set Architecture (ISA)

3.1 Overview of ISA

The designed ISA is a 32-bit instruction set with R-type, I-type, and J-type formats. It supports arithmetic, logical, branch, memory, and jump operations.

3.2 Instruction Formats

- **R-type:** Register-Register operations with opcode, rs, rt, rd, shamt, funct fields.
- **I-type:** Immediate and memory operations with opcode, rs, rt, immediate.
- **J-type:** Jump operations with opcode and address.

3.3 Instruction Table

A full instruction table is prepared listing instructions, opcodes, formats, and functionality.

Table 1: Complete Instruction Set Architecture Table

Instruction	**Opcode**	**Format**	**Funct**	**Functionality**
		R-Type Instructions		
add	000000	R	100000	Add: rd = rs + rt
sub	000000	R	100010	Subtract: rd = rs - rt
and	000000	R	100100	Bitwise AND: rd = rs & rt
or	000000	R	100101	Bitwise OR: rd = rs | rt
xor	000000	R	100110	Bitwise XOR: rd = rs ⊕ rt
nor	000000	R	100111	Bitwise NOR: rd = ∼(rs | rt)
slt	000000	R	101010	Set less than: rd = (rs < rt) ? 1 : 0
sll	000000	R	000000	Shift left logical: rd = rt « shamt
srl	000000	R	000010	Shift right logical: rd = rt » shamt
jr	000000	R	001000	Jump register: PC = rs
		I-Type Instructions		
addi	001000	I	–	Add immediate: rt = rs + imm
andi	001100	I	–	AND immediate: rt = rs & imm
ori	001101	I	–	OR immediate: rt = rs | imm
lw	100011	I	–	Load word: rt = mem[rs + imm]
sw	101011	I	–	Store word: mem[rs + imm] = rt
beq	000100	I	–	Branch equal: if (rs == rt) PC += imm
bne	000101	I	–	Branch not equal: if (rs != rt) PC += imm
bgez	000001	I	–	Branch if greater or equal to zero: if (rs >= 0) PC += imm
		J-Type Instructions		
j	000010	J	–	Jump: PC = address
jal	000011	J	–	Jump and link: $ra = PC + 4, PC = address

3.4 Register File

The register file consists of 32 general-purpose registers. Registers include `$zero`, `$at`, `$v0-$v1`, `$a0-$a3`, `$t0-$t9`, `$s0-$s7`, `$k0-$k1`, `$gp`, `$sp`, `$fp`, and `$ra`.

3.5 I/O Convention

Since this is a simulation-based project without physical I/O devices, I/O is handled through memory-mapped addresses using `lw` and `sw`. Input data is pre-loaded into data memory, and output results are written to specific register or memory locations that can be monitored during simulation. Additionally, instruction and data memory are initialized directly through Verilog using:

```
initial $readmemb("instruction.mem", Imem);
initial $readmemb("data.mem", Dmem);
```

This approach allows for efficient testing and verification of CPU functionality without requiring external hardware interfaces.

4 Control Signals and Instruction Encoding

The control of different module and routing signals to different modules for different instructions are crucial for the correct operation of the CPU. These signals are implemented and defined in the `Control.v` module. The control unit generates appropriate signals based on instruction opcodes and function codes to coordinate datapath operations. The control signals for each instruction are summarized in the following table:

Table 2: Control Signals for Each Instruction

Instruction	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	Jump	JAL	JR	ALUControl
add	1	1	0	0	0	0	0	0	0	0000
addu	1	1	0	0	0	0	0	0	0	0000
sub	1	1	0	0	0	0	0	0	0	0001
subu	1	1	0	0	0	0	0	0	0	0001
and	1	1	0	0	0	0	0	0	0	0010
or	1	1	0	0	0	0	0	0	0	0011
xor	1	1	0	0	0	0	0	0	0	0100
nor	1	1	0	0	0	0	0	0	0	1010
slt	1	1	0	0	0	0	0	0	0	1000
sltu	1	1	0	0	0	0	0	0	0	1001
sll	1	1	0	0	0	0	0	0	0	0101
srl	1	1	0	0	0	0	0	0	0	0110
sra	1	1	0	0	0	0	0	0	0	0111
sllv	1	1	0	0	0	0	0	0	0	1011
srlv	1	1	0	0	0	0	0	0	0	1100
srav	1	1	0	0	0	0	0	0	0	1101
jr	0	0	0	0	0	0	0	0	1	1111
j	0	0	0	0	0	0	1	0	0	0010
jal	1	0	0	0	0	0	1	1	0	0000
addi	1	0	1	0	0	0	0	0	0	0000
addiu	1	0	1	0	0	0	0	0	0	0000
andi	1	0	1	0	0	0	0	0	0	0010
ori	1	0	1	0	0	0	0	0	0	0011
xori	1	0	1	0	0	0	0	0	0	0100
slti	1	0	1	0	0	0	0	0	0	1000
sltiu	1	0	1	0	0	0	0	0	0	1001
lui	1	0	1	0	0	0	0	0	0	1110
lw	1	0	1	0	0	1	0	0	0	0000
sw	0	0	1	0	1	0	0	0	0	0000
beq	0	0	0	1	0	0	0	0	0	0001
bne	0	0	0	1	0	0	0	0	0	0001
bgez	0	0	0	1	0	0	0	0	0	1000

5 Assembler

5.1 Implementation Details

A MIPS assembler was given to convert human-readable MIPS-assembly code into machine code compatible with our CPU implementation. The assembler supports the complete instruction set defined in our ISA and generates multiple output formats to facilitate different aspects of CPU testing and simulation.

The assembler generates three distinct files:

- `.no_address.text.bin` - Contains pure instruction machine code without memory addresses
- `.no_address.data.bin` - Contains initialized data segment values in binary format
- `.text` - Contains formatted instruction machine code with address information. And can be generated in multiple formats (binary and hexadecimal). Later discussed in Improvement & Bug Fixes

These output files are specifically designed for direct loading into Verilog memory modules using `$readmemb` system tasks.

5.2 Architecture and Components

The assembler is implemented in C++ using a modular architecture. The key components include:

`finalassembler.cpp` The main assembler engine that orchestrates the entire assembly process. It performs lexical analysis, syntax parsing, symbol table management, two-pass assembly, and binary code generation. The module handles both text and data segment processing with comprehensive error reporting.

`mips.h` A comprehensive header file that defines the complete MIPS instruction set architecture. It contains instruction format specifications, opcode mappings, function code definitions, register encodings, and instruction type classifications. This module serves as the central reference for all ISA-related constants and mappings.

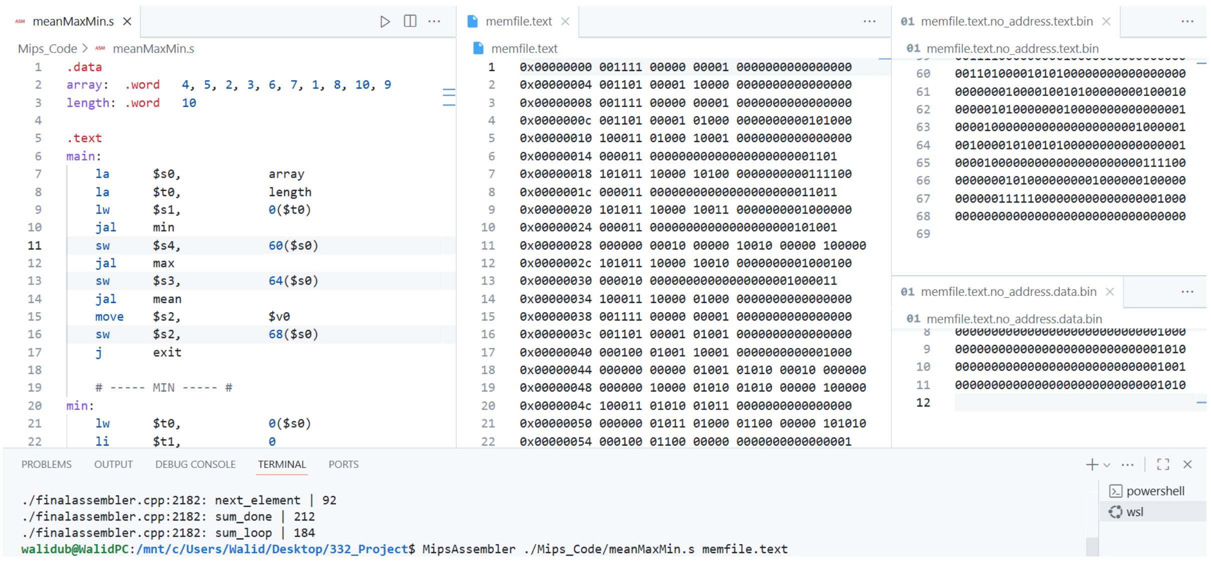
`data_symbol_table.h` A sophisticated symbol table implementation specifically designed for data segment management. It handles various data directives including `.word`, `.ascii`, `.asciiz`, `.space`, and others. The module uses visitor pattern for code generation and supports complex data structure initialization.

5.3 Assembly Process and Workflow

The compiled binary of the assembler takes a MIPS assembly file (`.s`) as input and produces `.text`, `no_address.text.bin`, and `no_address.data.bin` files. These outputs are suitable for direct loading into instruction and data memories using Verilog's `$readmemb`.

```
$ MipsAssembler input_file.s output_file.text
```

The screenshot below shows the assembler being executed in a terminal, along with the generated output filenames and basic assembler diagnostics. This provides a quick visual confirmation that the assembler completed successfully and produced the expected memory files for simulation.



```
.data
array:  .word   4, 5, 2, 3, 6, 7, 1, 8, 10, 9
length: .word   10

.text
main:
    la      $s0,            array
    la      $t0,            length
    lw      $s1,            0($t0)
    jal     min
    sw      $s4,            60($s0)
    jal     max
    sw      $s3,            64($s0)
    jal     mean
    move    $s2,            $v0
    sw      $s2,            68($s0)
    j       exit

    # ----- MIN ----- #
min:
    lw      $t0,            0($s0)
    li      $t1,            0
```

```
0x00000000 001111 00000 00001 0000000000000000
0x00000004 001101 00001 10000 0000000000000000
0x00000008 001111 00000 00001 0000000000000000
0x0000000c 001101 00001 01000 0000000000101000
0x00000010 100011 01000 10001 0000000000000000
0x00000014 000011 00000000000000000000001101
0x00000018 101011 10000 10100 0000000000111100
0x0000001c 000011 00000000000000000000011011
0x00000020 101011 10000 10011 0000000001000000
0x00000024 000011 00000000000000000000101001
0x00000028 000000 00010 00000 10010 00000 100000
0x0000002c 101011 10000 10010 0000000001000100
0x00000030 000010 00000000000000000001000011
0x00000034 100011 10000 01000 0000000000000000
0x00000038 001111 00000 00001 0000000000000000
0x0000003c 001101 00001 01001 0000000000000000
0x00000040 000100 01001 10001 0000000000001000
0x00000044 000000 00000 01001 01010 00010 000000
0x00000048 000000 10000 01010 01010 00000 100000
0x0000004c 100011 01010 01011 0000000000000000
0x00000050 000000 01011 01000 01100 00000 101010
0x00000054 000100 01100 00000 0000000000000001
```

```
00110100001010100000000000000000
00000001000010010100000000100010
00000101000000010000000000000001
00001000000000000000000001000001
00100001010010100000000000000001
00001000000000000000000000111100
00000001010000000001000000100000
00000011111000000000000000001000
00000000000000000000000000000000
```

```
00000000000000000000000000001000
00000000000000000000000000001010
00000000000000000000000000001001
00000000000000000000000000001010
```

```
./finalassembler.cpp:2182: next_element | 92
./finalassembler.cpp:2182: sum_done | 212
./finalassembler.cpp:2182: sum_loop | 184
walidub@WalidPC:/mnt/c/Users/Walid/Desktop/332_Project$ MipsAssembler ./Mips_Code/meanMaxMin.s memfile.text
```

Figure 1: Assembler compilation run: command invocation, assembler messages, and generated output files (`memfile.text`, `memfile.text.no_address.text.bin`, `memfile.text.no_address.data.bin`) ready for Verilog memory initialization.

5.4 Testing and Validation

The assembler was extensively tested with various assembly programs including:

- Simple arithmetic and logical operations
- Complex control flow programs with nested loops and conditionals
- Function call mechanisms using `jal` and `jr` instructions
- Conditional branching with `beq`, `bne`, and `bgez` instructions
- The complete **minMaxMean** implementation

All test programs were successfully assembled, and the generated machine code was verified against manual calculations. The binary outputs were validated by loading them into the Verilog CPU simulation and confirming correct execution behavior.

5.5 Testing and Sample Outputs

Example programs were assembled and tested successfully. Outputs matched expected binary instruction encodings. However, for conditional branching instructions, the assembler produces incorrect branch offsets. Discussed in Improvement & Bug Fixes.

5.6 Improvement & Bug Fixes

Several improvements and bug fixes were made to enhance the assembler's functionality and reliability, including:

- Generate the text file in a human-readable format that follows MIPS Convention with proper bit field separation for each instruction type:
  - **R-type:** 6-bit opcode | 5-bit rs | 5-bit rt | 5-bit rd | 5-bit shamt | 6-bit funct
  - **I-type:** 6-bit opcode | 5-bit rs | 5-bit rt | 16-bit immediate
  - **J-type:** 6-bit opcode | 26-bit address

This format replaces the previous 4-bit grouping with proper MIPS instruction field boundaries for easier debugging and verification.

- Added support for generating `.text` files in both binary and hexadecimal formats, depending on the compiler flags.

```
$ MipsAssembler --hex input_file.s output_file.text
```

This enhancement was done mainly for debugging purposes. Here's a comparative analysis showing the old format versus the new improved formats:

Table 3: Assembler Output Format Comparison

Instr.	**Old Format**	**New Binary**	**New Hex**
addi \$t0, \$zero, 10	0010 0000 0000 1000 0000 0000 0000 1010	001000 00000 01000 0000000000001010	8 0 8 A
addi \$t1, \$zero, 5	0010 0000 0000 1001 0000 0000 0000 0101	001000 00000 01001 0000000000000101	8 0 9 5
lw \$t2, 0(\$t1)	1000 1101 0010 1010 0000 0000 0000 0000	100011 01001 01010 0000000000000000	23 9 A 0
add \$t3, \$t0, \$t1	0000 0001 0000 1001 0101 1000 0010 0000	000000 01000 01001 01011 00000 100000	0 8 9 B 0 20
sw \$t3, 4(\$t1)	1010 1101 0010 1011 0000 0000 0000 0100	101011 01001 01011 0000000000000100	2B 9 B 4
jr \$ra	0000 0011 1110 0000 0000 0000 0000 1000	000000 11111 00000 00000 00000 001000	0 1F 0 0 0 8

The enhanced binary format distinctly delineates MIPS instruction field boundaries, substantially improving debugging and verification processes. The hexadecimal format offers a concise representation ideal for rapid visual inspection for verification and debugging purposes.

- **Fixed branch offset calculations for conditional instructions (Critical)**
  The old assembler incorrectly calculated branch offsets for `beq`, `bne`, and `bgez` instructions. It just calculated the jump address not the relative offset.

Table 4: Mistake Branch Offset Calculation Example

Address	**Instruction**	**Binary**	**Notes**
0x00000000	addi \$t0, \$zero, 4	0010 0000 0000 1000 0000 0000 0000 0100	put 4 to \$t0
0x00000004	addi \$t1, \$zero, 5	0010 0000 0000 1001 0000 0000 0000 0101	put 5 to \$t1
0x00000008	jal increment	0000 1100 0000 0000 0000 0000 0000 0101	Jump to 0101 « 2 = 0x14
0x0000000c	beq \$t0, \$t1, success	0001 0001 0000 1001 0000 0000 0000 0111	Branch offset = 0x7 Final address: PC + 4 + Offset « 2 = 0xC + 0x4 + 0x1C = 2C (expected 0x14)
0x00000010	j exit	0000 1000 0000 0000 0000 0000 0000 1000	Jump to 1000 « 2 = 0x20
0x00000014	addi \$t0, \$t0, 1	0010 0001 0000 1000 0000 0000 0000 0001	increment function
0x00000018	jr \$ra	0000 0011 1110 0000 0000 0000 0000 1000	Return from function
0x0000001c	addi \$v0, \$zero, 1	0010 0000 0000 0010 0000 0000 0000 0001	success label
0x00000020	nop	0000 0000 0000 0000 0000 0000 0000 0000	exit label

As shown in the example above, the branch instruction at address `0x0000000c` is intended to branch to the `success` label at address `0x00000014`. However, the old assembler calculated the offset incorrectly, resulting in a final branch target of `0x0000002c` instead of the expected `0x00000014`. which is not the instruction range, causing the program to malfunction. This is because the assembler used the absolute address of the target label instead of computing the relative offset from the current program counter (PC) value. The correct offset should be calculated as:

$$\begin{aligned}\text{Offset} &= \text{target\_address} - (\text{PC} + 4)\\ &= 0x00000014 - (0x0000000c + 4)\\ &= 0x00000014 - 0x00000010\\ &= 0x00000004\end{aligned}$$

The assembler was updated to correctly compute this relative offset, ensuring that branch instructions now function as intended. The updated assembler implementation is available in the reference section.

6 Operating System (Benchmark Programs)

6.1 Main Benchmark Program

A pseudo-OS program was written in MIPS assembly to test the CPU's functionality. This program, which calculates the minimum, maximum, and mean values of a 10-integer array, serves as a comprehensive test case. The implementation is structured with a `.data` section for data initialization in the data memory `(dmem)` and `.text` section containing the program logic. This logic incorporates a variety of core MIPS instructions, such as `lw`, `sw`, `addi`, `sll`, `slt`, `beq`, `j`, `jal`, and `jr`.

It performs both memory read and write operations, arithmetic computations, procedure calls and control flow management through loops and conditional branches, thoroughly testing the CPU implementation. **The full code is available in the Assembly Code section.**

MIPS Assembly Code for MIN, MAX

```
.data
array:  .word 4, 5, 2, 3, 6, 7, 1, 8, 10, 9
length:  .word 10

.text
main:             min:                           max:
 la $s0, array     lw $t0, 0($s0)                 lw $t0, 0($s0)
 la $t0, length    li $t1, 0                      li $t1, 0
 lw $s1, 0($t0)   min_loop:                      max_loop:
 jal min           beq $t1, $s1, min_done         beq $t1, $s1, max_done
 sw $s4, 60($s0)   sll $t2, $t1, 2                sll $t2, $t1, 2
 jal max           add $t2, $s0, $t2              add $t2, $s0, $t2
 sw $s4, 64($s0)   lw $t3, 0($t2)                 lw $t3, 0($t2)
 jal mean          slt $t4, $t3, $t0              slt $t4, $t0, $t3
 move $s2, $v0     beq $t4, $zero, next_element   beq $t4, $zero, next_element
 sw $s4, 68($s0)   move $t0, $t3                  move $t0, $t3
 j exit           next_element:                  next_element:
                   addi $t1, $t1, 1               addi $t1, $t1, 1
                   j min_loop                     j max_loop
                  min_done:                      max_done:
                   move $s4, $t0                  move $s4, $t0
                   jr $ra                         jr $ra
```

6.2 Use of JAL and JR

As we discussed earlier, we need to implement `jal` and `jr` instructions in Verilog. So, we are required to ensure that the program uses these instructions and works correctly.

The `jal` and `jr` instructions are among the core instructions of the MIPS architecture. The `jal` instruction is used to jump to a procedure and link the return address, or in other words, stores the return address in the `$ra` register, while the `jr` instruction is used to return from a procedure by jumping to the address stored in a register (typically in `$ra`).

6.3 Other Benchmark Programs

In addition to the **minMaxMean** program, several other benchmark programs were developed to thoroughly test the CPU and assembler. These programs contains a variety of instruction types, control flow constructs, and memory operations:

- **Fibonacci Sequence**: Computes the Fibonacci series using loops and register operations, testing arithmetic, branching, and memory access. Source Code.
- **Array Manipulation**: Demonstrates array traversal and element modification, validating correct implementation of load/store and indexed addressing. Source Code.
- **Branch Test**: Contains multiple conditional branches and jump instructions to verify the accuracy of branch offset calculations and control flow. Source Code.

6.4 CPUlator Simulation Results

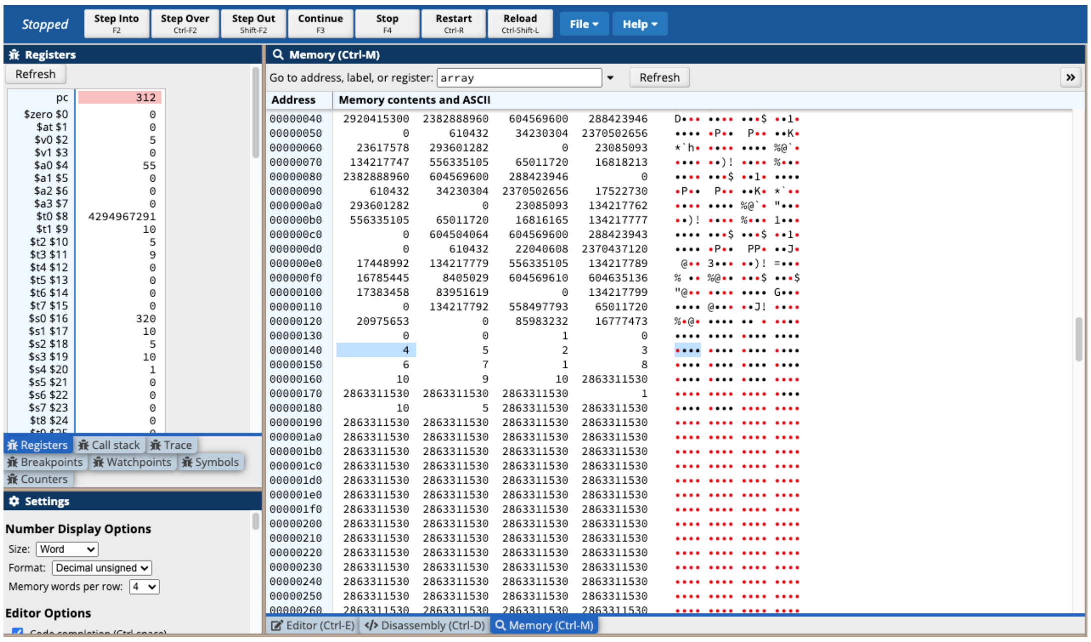
To further validate our program, we used the CPUlator MIPS simulator to run our programs. CPUlator provides a cycle-accurate simulation environment for MIPS CPUs, allowing us to observe instruction execution, register updates, and memory operations in detail.

6.4.1 Simulation Setup and Key Observations

Assembly code was loaded into CPUlator's code editor and compiled. Benchmark programs (minMaxMean, Fibonacci, Array Manipulation) were executed step-by-step, with register and memory contents monitored after each instruction to verify correctness.

All instructions, including `jal` and `jr`, executed as expected. Register file and memory contents matched the expected results after each benchmark.

6.4.2 Screenshots and Output



```
        pc         312
  $zero $0           0
    $at $1           0
    $v0 $2           5
    $v1 $3           0
    $a0 $4          55
    $a1 $5           0
    $a2 $6           0
    $a3 $7           0
    $t0 $8  4294967291
    $t1 $9          10
   $t2 $10           5
   $t3 $11           9
   $t4 $12           0
   $t5 $13           0
   $t6 $14           0
   $t7 $15           0
   $s0 $16         320
   $s1 $17          10
   $s2 $18           5
   $s3 $19          10
   $s4 $20           1
   $s5 $21           0
   $s6 $22           0
   $s7 $23           0
   $t8 $24           0
```

Number Display Options

Size: Word

Format: Decimal unsigned

Memory words per row: 4

Editor Options

Go to address, label, or register: array

```
Address   Memory contents and ASCII
00000040  2920415300 2382888960  604569600  288423946   D••• •••• •••$ ••1•
00000050           0     610432   34230304 2370502656   •••• •P••  P•• ••K•
00000060    23617578  293601282          0   23085093   *`h• •••• •••• %@`•
00000070   134217747  556335105   65011720   16818213   •••• ••)! •••• %•••
00000080  2382888960  604569600  288423946          0   •••• •••$ ••1• ••••
00000090      610432   34230304 2370502656   17522730   •P••  P•• ••K• *`••
000000a0   293601282          0   23085093  134217762   •••• •••• %@`• "•••
000000b0   556335105   65011720   16816165  134217777   ••)! •••• %••• 1•••
000000c0           0  604504064  604569600  288423943   •••• •••$ •••$ ••1•
000000d0           0     610432   22040608 2370437120   •••• •P••  PP• ••J•
000000e0    17448992  134217779  556335105  134217789    @•• 3••• ••)! =•••
000000f0    16785445    8405029  604569610  604635136   % •• %@•• •••$ •••$
00000100    17383458   83951619          0  134217799   "@•• •••• •••• G•••
00000110           0  134217792  558497793   65011720   •••• @••• ••J! ••••
00000120    20975653          0   85983232   16777473   %•@• •••• ••  • ••••
00000130           0          0          1          0   •••• •••• •••• ••••
00000140           4          5          2          3   •••• •••• •••• ••••
00000150           6          7          1          8   •••• •••• •••• ••••
00000160          10          9         10 2863311530   •••• •••• •••• ••••
00000170  2863311530 2863311530 2863311530          1   •••• •••• •••• ••••
00000180          10          5 2863311530 2863311530   •••• •••• •••• ••••
00000190  2863311530 2863311530 2863311530 2863311530   •••• •••• •••• ••••
000001a0  2863311530 2863311530 2863311530 2863311530   •••• •••• •••• ••••
000001b0  2863311530 2863311530 2863311530 2863311530   •••• •••• •••• ••••
000001c0  2863311530 2863311530 2863311530 2863311530   •••• •••• •••• ••••
000001d0  2863311530 2863311530 2863311530 2863311530   •••• •••• •••• ••••
000001e0  2863311530 2863311530 2863311530 2863311530   •••• •••• •••• ••••
000001f0  2863311530 2863311530 2863311530 2863311530   •••• •••• •••• ••••
00000200  2863311530 2863311530 2863311530 2863311530   •••• •••• •••• ••••
00000210  2863311530 2863311530 2863311530 2863311530   •••• •••• •••• ••••
00000220  2863311530 2863311530 2863311530 2863311530   •••• •••• •••• ••••
00000230  2863311530 2863311530 2863311530 2863311530   •••• •••• •••• ••••
00000240  2863311530 2863311530 2863311530 2863311530   •••• •••• •••• ••••
00000250  2863311530 2863311530 2863311530 2863311530   •••• •••• •••• ••••
00000260  2863311530 2863311530 2863311530 2863311530   •••• •••• •••• ••••
```

Figure 2: CPUlator simulation of the minMaxMean program.
Input array values from 1 to 10.

Screenshots from CPUlator showing register and memory states for each benchmark, Other screenshots are included in the appendix. The input was an array of integers from 1 to 10, and the output correctly showed Min = 1 at `$s4`, Max = 10 at `$s3`, Mean = 5 at `$s2`, as well as in the memory locations relative to the base address of the array.

7 CPU Design in Verilog

The CPU was implemented in Verilog HDL as a single-cycle MIPS processor. The design includes the following key components:

7.1 Datapath Architecture

The CPU datapath consists of interconnected functional units that execute instructions in a single clock cycle:

Arithmetic Logic Unit (ALU) A 32-bit ALU that performs arithmetic and logical operations based on 4-bit control signals. Supports 16 operations including addition, subtraction, bitwise operations (AND, OR, XOR, NOR), shift operations (SLL, SRL, SRA, SLLV, SRLV, SRAV), comparison (SLT, SLTU), and load upper immediate (LUI).

Register File A dual-port register file containing 32 general-purpose 32-bit registers. Provides simultaneous read access to two registers and write access to one register per cycle.

Memory Subsystem Consists of separate instruction and data memories. Instruction memory stores 32-bit program instructions loaded from assembler output, while data memory handles load/store operations or can be loaded with .data segments.

Program Counter (PC) holds the address of the current instruction. Updated each cycle based on control flow: sequential increment (PC+4), branch target calculation, or jump address selection.

7.2 Control Unit Architecture

The control unit implements a combinational logic design that decodes 32-bit instructions and generates appropriate control signals. It analyzes the 6-bit opcode and 6-bit function fields to determine instruction type and operation, then asserts specific control signals for datapath coordination. The unit supports all three MIPS instruction formats (R-type, I-type, J-type) and generates 10 primary control signals as detailed in Table 2.

7.3 Integration of JAL and JR Instructions

The datapath and control unit were enhanced to support `jal` (Jump and Link) and `jr` (Jump Register) instructions, which are essential for function calls and returns in MIPS assembly programs.

7.3.1 Control Unit Enhancements

Two new control signals were added to the control unit: `JAL` and `JR`. When `jal` is executed, the control unit sets `RegWrite = 1`, `Jump = 1`, and `JAL = 1` to enable register writing, jump operation, and return address selection respectively. When `jr` is executed, the control unit sets `JR = 1` to select the register value as the next PC address. The control signal table is shown in Table 2.

7.3.2 Datapath Enhancements

The datapath modifications (highlighted in red boxes in Figure 3) implement function call mechanisms through two key enhancements:

For the **JAL (Jump and Link)** instruction, we introduced two 2-to-1 multiplexers to handle its dual requirements of jumping and linking. The first multiplexer selects the write-back data to the register file, choosing between the standard `ALU or memory` output and the return address `(PC + 4)`. This is essential because JAL must store the return address in `$ra` (register 31) to enable the caller to resume execution after the subroutine, avoiding conflicts with other computational results. The second multiplexer overrides the normal destination register (rd) selection to force writing to register 31 (`$ra`).

For the **JR (Jump Register)** instruction, we added a single 2-to-1 multiplexer to the PC update logic. This multiplexer selects between the normal next-PC value (PC + 4) and the register file's read data `(data one)` when the JR signal is asserted. When `JR = 1`, this enables unconditional jumps to the address stored in a register (typically \$ra), enabling subroutine returns by jumping to the stored return address.

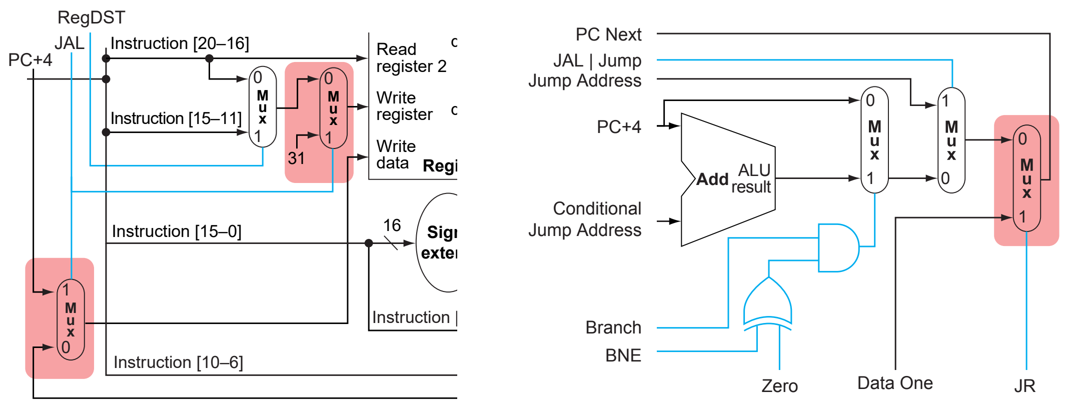


Figure 3: Changes made to the datapath to support `jal` and `jr` instructions. Red boxes indicate added components.

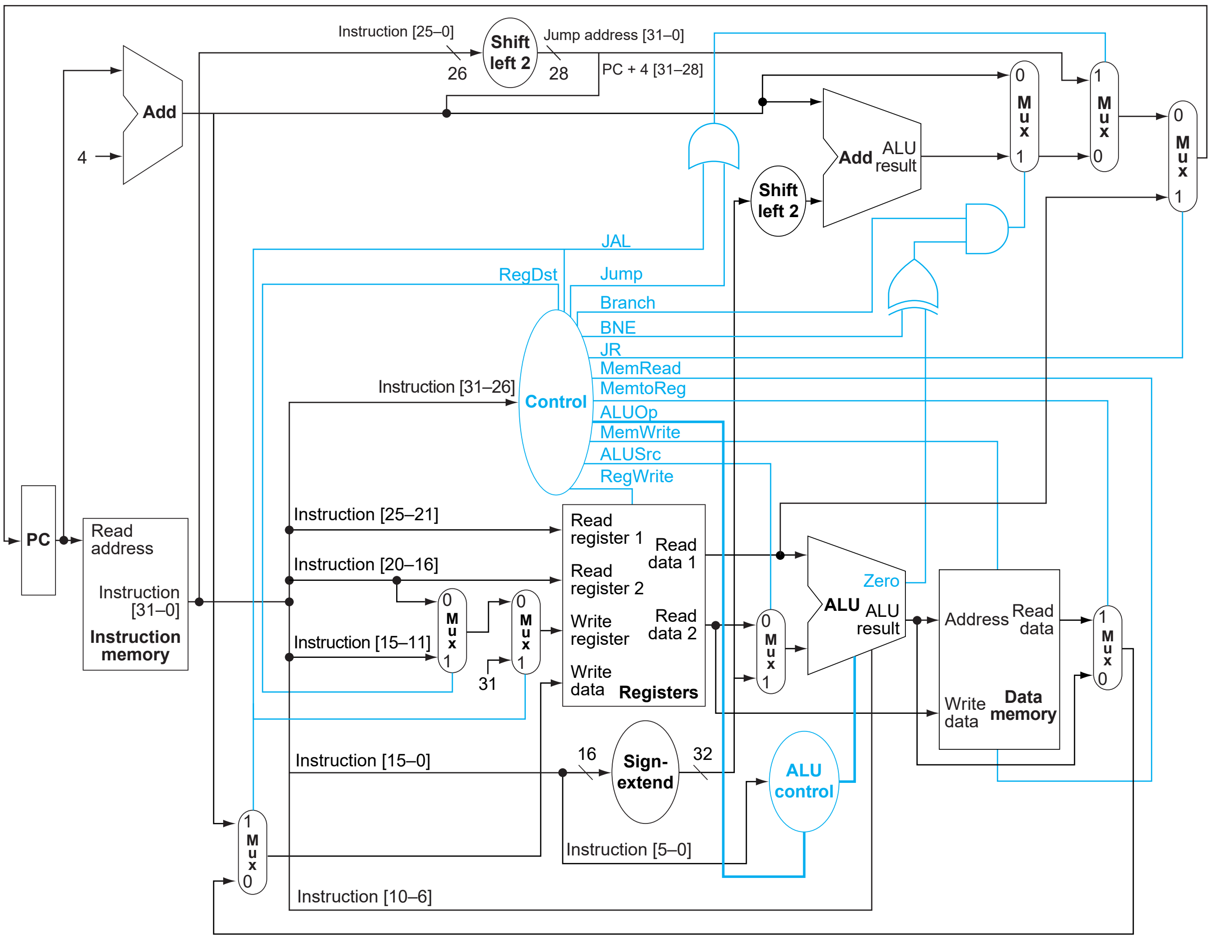


Figure 4: Complete datapath and control unit layout after enhancements for `jal` and `jr` instructions.

7.4 Block Diagrams and RTL Synthesis

The following diagram shows the synthesized RTL representation of our implemented MIPS CPU. This synthesized diagram illustrates the actual hardware structure generated from our Verilog code, where the control unit combines opcode and function code processing into a unified control logic block.

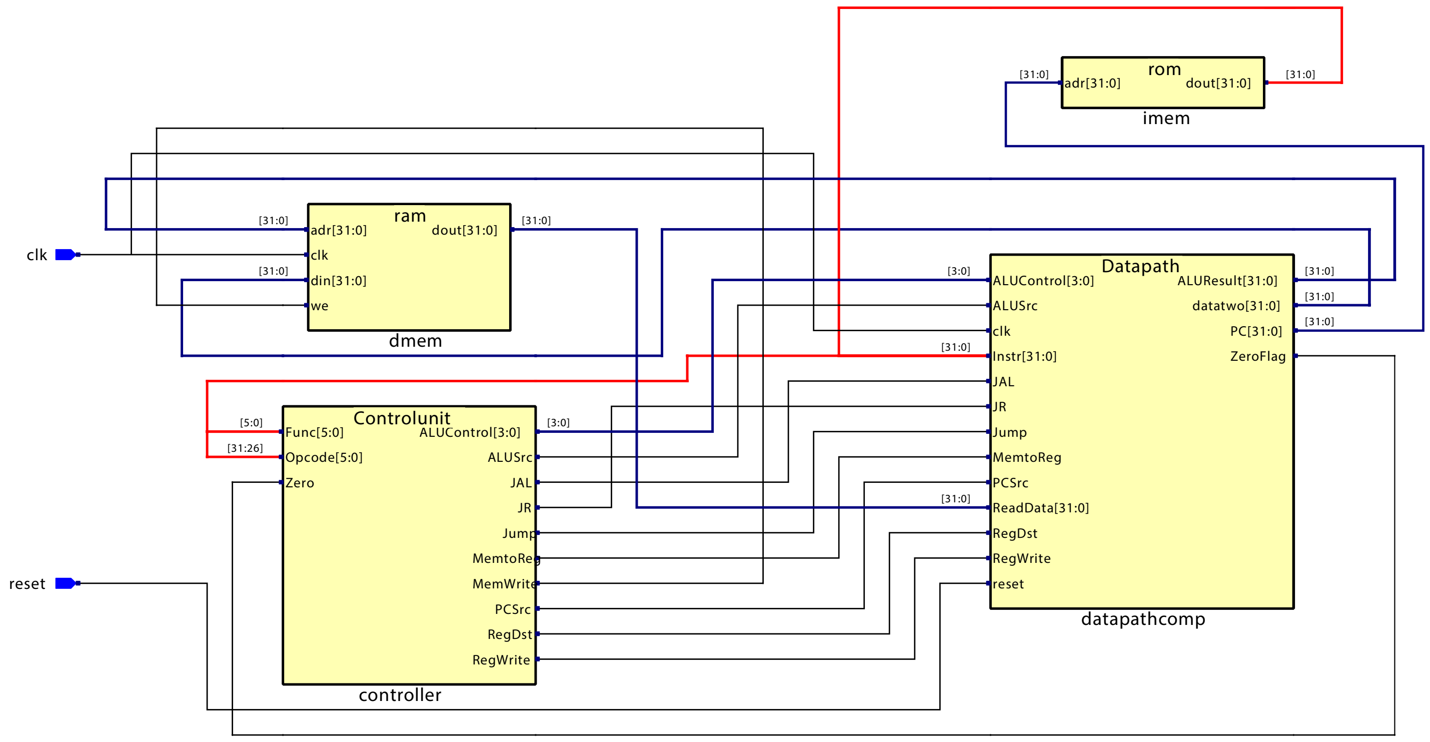


Figure 5: RTL synthesis diagram of the single-cycle MIPS CPU (Top-module) showing the synthesized hardware structure with combined opcode-function code control unit.

8 Simulation and Testing

8.1 Simulation Setup

ModelSim was used to simulate the Verilog design. Assembled machine code were loaded into instruction memory, and data memory was initialized as needed.

8.2 Test Cases

Four comprehensive programs were used to validate the CPU implementation:

- **minMaxMean**: Complete operating system-style program computing minimum, maximum, and mean of an array using function calls (`jal`, `jr`), loops, conditional branches, and arithmetic operations.
- **Fibonacci Sequence**: Iterative computation testing arithmetic operations, register management, and control flow.
- **Array Manipulation**: Program demonstrating array traversal and element modification using load/store operations and indexed addressing.
- **Branch Test**: Dedicated program testing conditional branching instructions (`beq`, `bne`, `bgez`) with various branch conditions and targets.

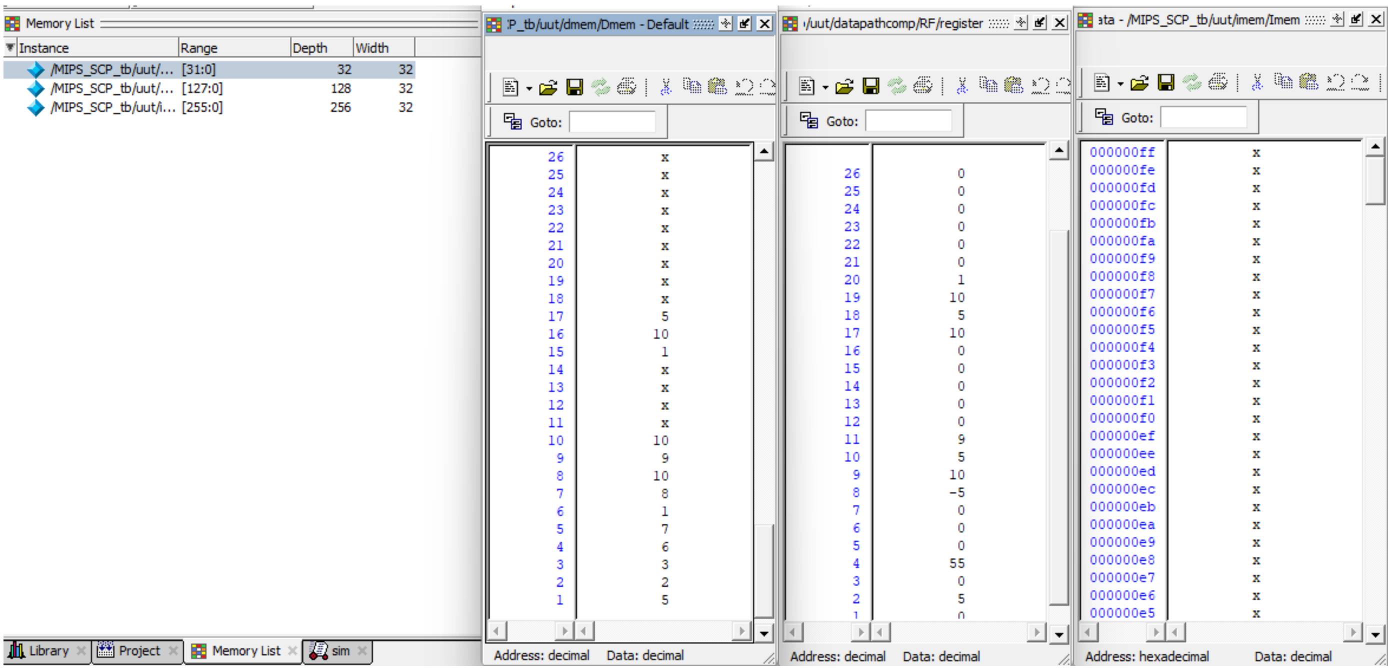
8.3 Results

All simulation results matched expected outputs and were consistent with CPUlator reference simulations. Key validation points included:

- **Register File Updates**: All register modifications occurred correctly according to instruction semantics.
- **Memory Operations**: Load and store instructions properly accessed data memory with correct addressing.
- **Control Flow**: Branch conditions, jump targets, and function calls executed as expected.
- **Function Call Mechanism**: `jal` correctly stored return addresses in `$ra`, and `jr` successfully returned to calling functions.

The simulation confirmed correct execution behavior for all test programs. Additional simulation screenshots for the remaining benchmark programs are provided in the appendix section, demonstrating comprehensive validation of the CPU implementation.

8.4 Screenshots



Memory List

Instance	Range	Depth	Width
/MIPS_SCP_tb/uut/...	[31:0]	32	32
/MIPS_SCP_tb/uut/...	[127:0]	128	32
/MIPS_SCP_tb/uut/i...	[255:0]	256	32

P_tb/uut/dmem/Dmem - Default

```
26   x
25   x
24   x
23   x
22   x
21   x
20   x
19   x
18   x
17   5
16  10
15   1
14   x
13   x
12   x
11   x
10  10
 9   9
 8  10
 7   8
 6   1
 5   7
 4   6
 3   3
 2   2
 1   5
Address: decimal    Data: decimal
```

/uut/datapathcomp/RF/register

```
26   0
25   0
24   0
23   0
22   0
21   0
20   1
19  10
18   5
17  10
16   0
15   0
14   0
13   0
12   0
11   9
10   5
 9  10
 8  -5
 7   0
 6   0
 5   0
 4  55
 3   0
 2   5
 1   0
Address: decimal    Data: decimal
```

ata - /MIPS_SCP_tb/uut/imem/Imem

```
000000ff  x
000000fe  x
000000fd  x
000000fc  x
000000fb  x
000000fa  x
000000f9  x
000000f8  x
000000f7  x
000000f6  x
000000f5  x
000000f4  x
000000f3  x
000000f2  x
000000f1  x
000000f0  x
000000ef  x
000000ee  x
000000ed  x
000000ec  x
000000eb  x
000000ea  x
000000e9  x
000000e8  x
000000e7  x
000000e6  x
000000e5  x
Address: hexadecimal    Data: decimal
```

Figure 6: ModelSim simulation of the minMaxMean program showing Memory List, Dmem, Register, and Imem windows.

9 Benchmarking and Performance Analysis

Comprehensive benchmarking was conducted using two computationally intensive programs: the Fibonacci sequence and a division subroutine from the minMaxMean program. These workloads exercise memory operations, arithmetic, conditional branches, and function calls, allowing evaluation of CPU efficiency across instruction types. Results and analysis are presented in Table 6 and Figure 7.

Fibonacci Sequence Benchmarking: The Fibonacci program computes the n-th Fibonacci number using iterative calculation with extensive memory read/write operations, arithmetic instructions, and loop control. Performance was measured for Fibonacci numbers ranging from F(10) to F(45).

Division Subroutine Benchmarking: The division subroutine performs repeated division operations, testing arithmetic operations, register management, and loop control structures. Performance was evaluated for input values ranging from 100 to 10,000.

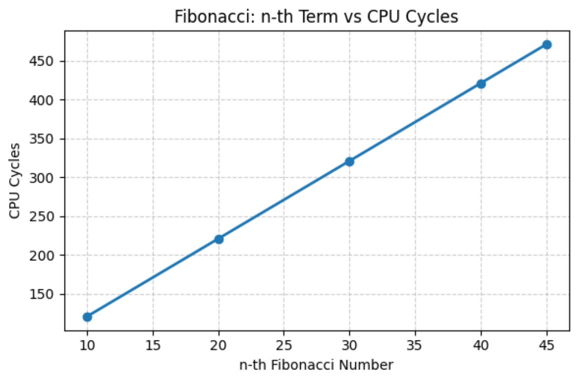
Table 6: Performance Benchmarking Results

Fibonacci Term (n)	Cycle Count
10	121
20	221
30	321
40	421
45	471

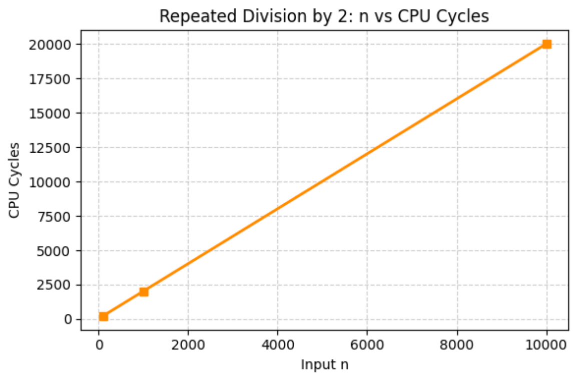
(a) Fibonacci Sequence Performance

Division Input (n)	Cycle Count
100	216
1,000	2,016
10,000	20,016

(b) Division Subroutine Performance



(a) Fibonacci sequence performance showing linear relationship between n-th term and CPU cycles



(b) Division subroutine performance demonstrating linear scaling with input size

Figure 7: Performance benchmarking results showing linear scaling characteristics

The benchmarking results demonstrate excellent linear scaling characteristics for both programs, confirming the predictable performance behavior of our single-cycle MIPS CPU implementation:

- **Fibonacci Performance**: Exhibits a linear relationship between the target Fibonacci number and CPU cycles, with approximately 10 cycles per increment in n. This consistent scaling validates the CPU's efficient handling of iterative arithmetic operations and loop control structures.
- **Division Performance**: Shows linear scaling with approximately 2 cycles per unit increase in input size. This demonstrates the CPU's consistent performance for repetitive computational tasks involving arithmetic operations and memory access patterns.

The linear scaling behavior in both benchmarks confirms the deterministic single-cycle execution model and validates the correct implementation of all instruction types across varying computational loads.

10 Discussion

10.1 Challenges Faced

Throughout this project, several significant challenges were encountered and successfully overcome:

- **Assembler Bug Resolution (Critical)**: The most significant challenge was discovering and fixing critical bugs in the provided assembler. The assembler was producing incorrect branch offsets for conditional instructions (`beq`, `bne`, `bgez`), calculating absolute addresses instead of relative offsets. This led to programs jumping to incorrect memory locations, causing complete system failures. As detailed in Section 5.6, extensive debugging was required to identify the root cause and implement proper offset calculations: `Offset = target_address - (PC + 4)`. This fix was crucial for enabling any meaningful CPU testing.

- **JAL and JR Implementation Complexity**: Understanding and implementing the `jal` and `jr` instructions in Verilog required careful analysis of the existing datapath and control unit. The challenge involved adding three new multiplexers for JAL implementation (return address selection, destination register selection, and jump target selection) and one multiplexer for JR implementation (PC source selection). Coordinating these additions with existing control signals while maintaining timing constraints required iterative debugging and verification.

- **Control Signal Synchronization**: Ensuring proper synchronization between the control unit and datapath components posed significant challenges, particularly for function call mechanisms. Debugging involved extensive use of ModelSim waveform analysis to verify that control signals were asserted at correct clock cycles and that register file updates occurred synchronously with instruction execution.

- **Memory Initialization and Format Compatibility**: Adapting the assembler output formats to be compatible with Verilog's `$readmemb` system tasks required understanding both the assembler's binary generation process and Verilog's memory initialization requirements. This involved modifying the assembler to produce separate instruction and data memory files in the correct binary format.

10.2 Design Trade-offs

Several important design decisions were made that involved performance and complexity trade-offs:

- **Single-Cycle vs. Pipelined Architecture**: The decision to implement a single-cycle MIPS CPU prioritized design simplicity and easier debugging over performance. While this approach enabled successful completion within project constraints, the benchmarking results in Section 9 clearly demonstrate the performance penalty. The Fibonacci sequence benchmark required 471 cycles to compute F(45), and the division subroutine needed 20,016 cycles for input value 10,000. A pipelined implementation could potentially achieve 4-5x performance improvement through parallel instruction execution.

- **Memory Architecture**: Implementing separate instruction and data memories (Harvard architecture) instead of unified memory (von Neumann) improved access bandwidth but increased hardware complexity and resource requirements. This trade-off was beneficial for simulation purposes but may not reflect real-world memory constraints.

10.3 Extensions and Future Work

Several enhancements could significantly improve the CPU's performance and functionality:

- **Pipelined Architecture**: Upgrading the CPU to a 5-stage pipeline (Instruction Fetch, Instruction Decode, Execute, Memory Access, Write Back) would significantly increase instruction throughput. Based on our benchmarks, this could reduce the cycle count for the Fibonacci program from 471 to approximately 100–120 cycles, achieving up to a 4x speedup. This would require implementing hazard detection, forwarding, and branch prediction mechanisms.
- **Expanded Instruction Set**: Incorporating IEEE 754 floating-point instructions would enable support for scientific and engineering applications that require decimal and floating-point arithmetic.
- **Enhanced Operating System Features**: Extending the OS-style programs to include more features like file I/O, and system calls.

11 Conclusion

We successfully designed a 32-bit CPU, assembler, and OS-style program. The CPU executed all benchmark programs correctly. This project reinforced concepts of ISA design, assembly programming, and hardware implementation.

12 References

- David A. Patterson, & John L. Hennessy *Computer Organization and Design: The Hardware/-Software Interface, 5th Edition.*
- GitHub: `https://github.com/RoySRC/UpgradedMIPS32Assembler.git`
- Course materials and lecture notes.
- Updated assembler code: `https://github.com/infinity236/UpgradedMIPS32Assembler.git`

A Verilog Source Codes

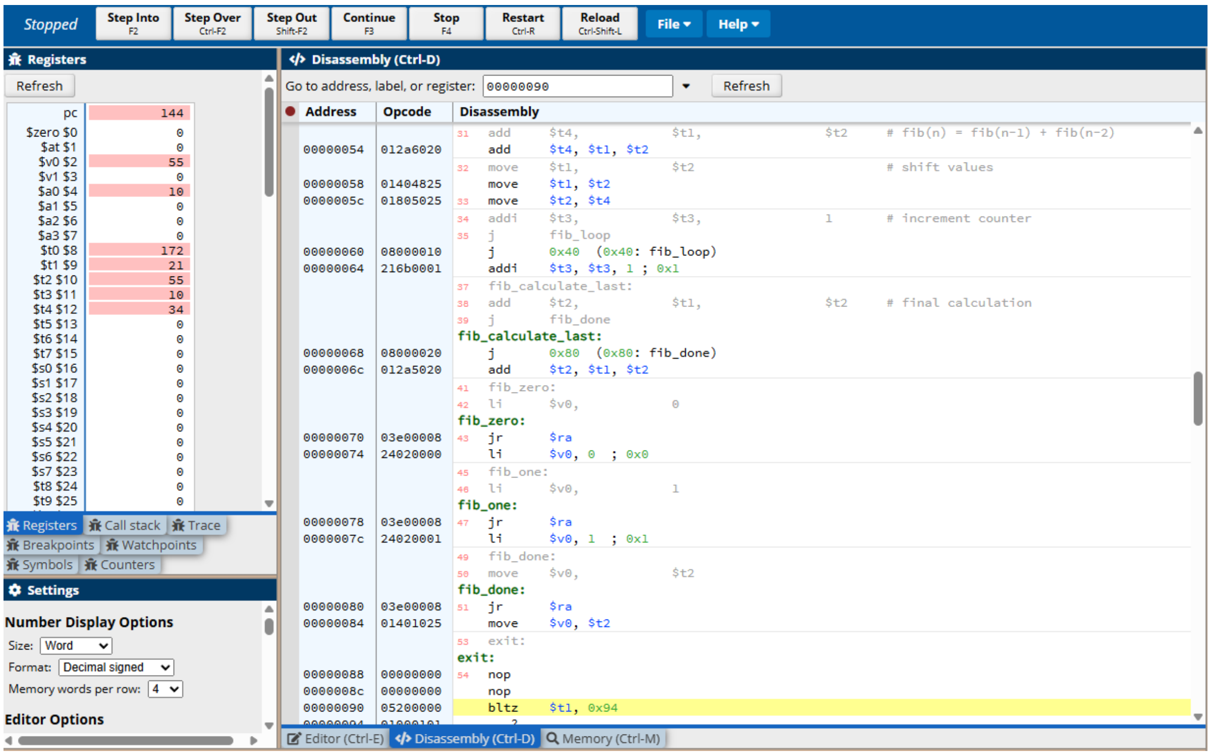
`https://github.com/infinity236/nsu-cse332-project/tree/main/Verilog`

B Assembly Source Codes

- minMaxMean:
  `https://github.com/infinity236/nsu-cse332-project/blob/main/Programs/minMaxMean.s`
- Fibonacci sequence:
  `https://github.com/infinity236/nsu-cse332-project/blob/main/Programs/Fibonacci.s`
- Array Manipulation:
  `https://github.com/infinity236/nsu-cse332-project/blob/main/Programs/increment_array.s`
- Branch Test:
  `https://github.com/infinity236/nsu-cse332-project/blob/main/Programs/beqTest.s`

C Screenshots

C.1 CPUlator Simulations



```
pc        144
$zero $0    0
$at $1      0
$v0 $2     55
$v1 $3      0
$a0 $4     10
$a1 $5      0
$a2 $6      0
$a3 $7      0
$t0 $8    172
$t1 $9     21
$t2 $10    55
$t3 $11    10
$t4 $12    34
$t5 $13     0
$t6 $14     0
$t7 $15     0
$s0 $16     0
$s1 $17     0
$s2 $18     0
$s3 $19     0
$s4 $20     0
$s5 $21     0
$s6 $22     0
$s7 $23     0
$t8 $24     0
$t9 $25     0
```

Number Display Options

Size: Word

Format: Decimal signed

Memory words per row: 4

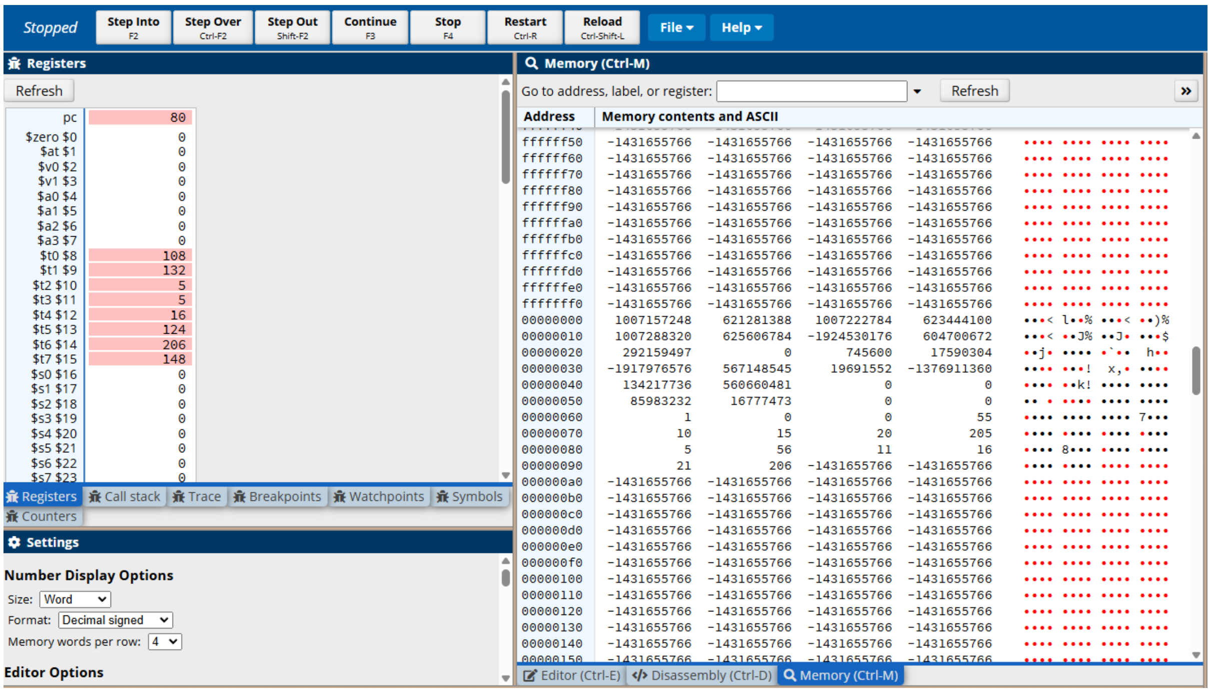
Editor Options

Disassembly (Ctrl-D)

Go to address, label, or register: 00000090

```
Address   Opcode    Disassembly
                       add     $t4,         $t1,         $t2     # fib(n) = fib(n-1) + fib(n-2)
00000054  012a6020     add     $t4, $t1, $t2
                       move    $t1,         $t2                  # shift values
00000058  01404825     move    $t1, $t2
0000005c  01805025     move    $t2, $t4
                       addi    $t3,         $t3,         1       # increment counter
                       j       fib_loop
00000060  08000010     j       0x40  (0x40: fib_loop)
00000064  216b0001     addi    $t3, $t3, 1 ; 0x1
                       fib_calculate_last:
                       add     $t2,         $t1,         $t2     # final calculation
                       j       fib_done
                   fib_calculate_last:
00000068  08000020     j       0x80  (0x80: fib_done)
0000006c  012a5020     add     $t2, $t1, $t2
                       fib_zero:
                       li      $v0,         0
                   fib_zero:
00000070  03e00008     jr      $ra
00000074  24020000     li      $v0, 0  ; 0x0
                       fib_one:
                       li      $v0,         1
                   fib_one:
00000078  03e00008     jr      $ra
0000007c  24020001     li      $v0, 1  ; 0x1
                       fib_done:
                       move    $v0,         $t2
                   fib_done:
00000080  03e00008     jr      $ra
00000084  01401025     move    $v0, $t2
                       exit:
                   exit:
00000088  00000000     nop
0000008c  00000000     nop
00000090  05200000     bltz    $t1, 0x94
```

Figure 8: Fibonacci Sequence



```
pc         80
$zero $0    0
$at $1      0
$v0 $2      0
$v1 $3      0
$a0 $4      0
$a1 $5      0
$a2 $6      0
$a3 $7      0
$t0 $8    108
$t1 $9    132
$t2 $10     5
$t3 $11     5
$t4 $12    16
$t5 $13   124
$t6 $14   206
$t7 $15   148
$s0 $16     0
$s1 $17     0
$s2 $18     0
$s3 $19     0
$s4 $20     0
$s5 $21     0
$s6 $22     0
$s7 $23     0
```

Number Display Options

Size: Word

Format: Decimal signed

Memory words per row: 4

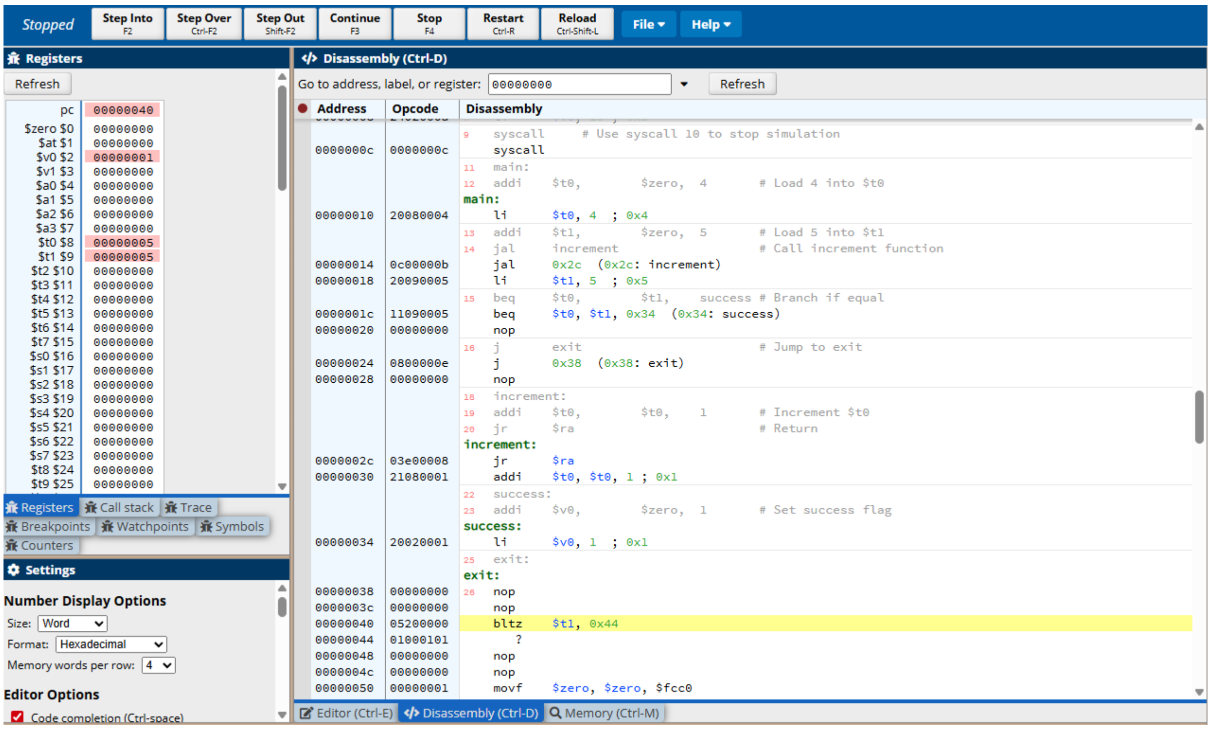
Editor Options

Memory (Ctrl-M)

Go to address, label, or register:

```
Address   Memory contents and ASCII
ffffff50  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
ffffff60  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
ffffff70  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
ffffff80  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
ffffff90  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
ffffffa0  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
ffffffb0  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
ffffffc0  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
ffffffd0  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
ffffffe0  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
fffffff0  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
00000000   1007157248   621281388  1007222784   623444100   •••< l••% •••< ••)%
00000010   1007288320   625606784 -1924530176   604700672   •••< ••J% ••J• •••$
00000020    292159497           0      745600    17590304   ••j• •••• •`••  h••
00000030  -1917976576   567148545    19691552 -1376911360   •••• •••!  x,• ••••
00000040    134217736   560660481           0           0   •••• ••k! •••• ••••
00000050     85983232    16777473           0           0   •• • •••• •••• ••••
00000060            1           0           0          55   •••• •••• •••• 7•••
00000070           10          15          20         205   •••• •••• •••• ••••
00000080            5          56          11          16   •••• 8••• •••• ••••
00000090           21         206 -1431655766 -1431655766   •••• •••• •••• ••••
000000a0  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
000000b0  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
000000c0  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
000000d0  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
000000e0  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
000000f0  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
00000100  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
00000110  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
00000120  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
00000130  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
00000140  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
00000150  -1431655766 -1431655766 -1431655766 -1431655766   •••• •••• •••• ••••
```

Figure 9: Array Manipulation



```
pc         00000040
$zero $0   00000000
$at $1     00000000
$v0 $2     00000001
$v1 $3     00000000
$a0 $4     00000000
$a1 $5     00000000
$a2 $6     00000000
$a3 $7     00000000
$t0 $8     00000005
$t1 $9     00000005
$t2 $10    00000000
$t3 $11    00000000
$t4 $12    00000000
$t5 $13    00000000
$t6 $14    00000000
$t7 $15    00000000
$s0 $16    00000000
$s1 $17    00000000
$s2 $18    00000000
$s3 $19    00000000
$s4 $20    00000000
$s5 $21    00000000
$s6 $22    00000000
$s7 $23    00000000
$t8 $24    00000000
$t9 $25    00000000
```

Number Display Options

Size: Word

Format: Hexadecimal

Memory words per row: 4

Editor Options

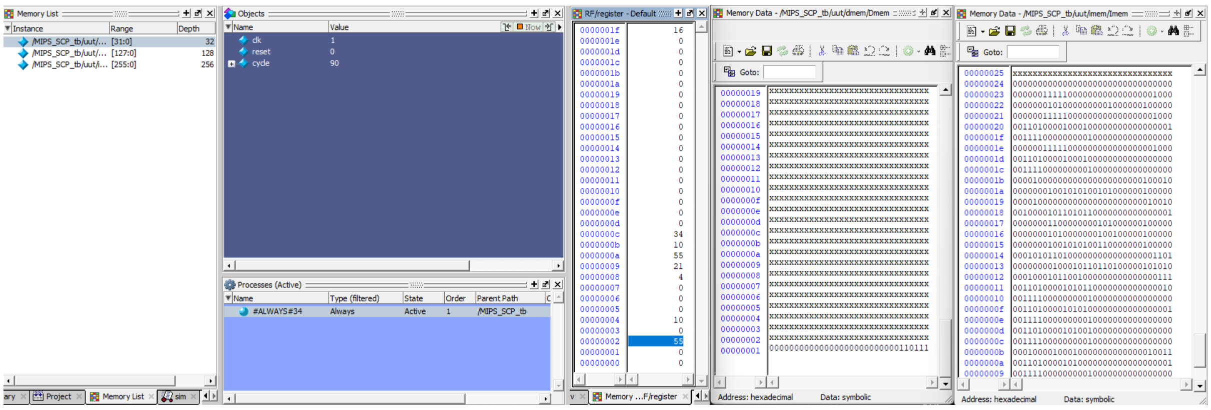
[✓] Code completion (Ctrl-space)

Go to address, label, or register: 00000000

```
Address   Opcode    Disassembly
                    syscall     # Use syscall 10 to stop simulation
0000000c  0000000c  syscall
                    main:
                    addi    $t0,        $zero,  4       # Load 4 into $t0
                    main:
00000010  20080004  li      $t0, 4  ; 0x4
                    addi    $t1,        $zero,  5       # Load 5 into $t1
                    jal     increment                   # Call increment function
00000014  0c00000b  jal     0x2c  (0x2c: increment)
00000018  20090005  li      $t1, 5  ; 0x5
                    beq     $t0,        $t1,    success # Branch if equal
0000001c  11090005  beq     $t0, $t1, 0x34  (0x34: success)
00000020  00000000  nop
                    j       exit                        # Jump to exit
00000024  0800000e  j       0x38  (0x38: exit)
00000028  00000000  nop
                    increment:
                    addi    $t0,        $t0,    1       # Increment $t0
                    jr      $ra                         # Return
                    increment:
0000002c  03e00008  jr      $ra
00000030  21080001  addi    $t0, $t0, 1 ; 0x1
                    success:
                    addi    $v0,        $zero,  1       # Set success flag
                    success:
00000034  20020001  li      $v0, 1  ; 0x1
                    exit:
                    exit:
00000038  00000000  nop
0000003c  00000000  nop
00000040  05200000  bltz    $t1, 0x44
00000044  01000101    ?
00000048  00000000  nop
0000004c  00000000  nop
00000050  00000001  movf    $zero, $zero, $fcc0
```

Figure 10: Branch Test

C.2 ModelSim Simulations



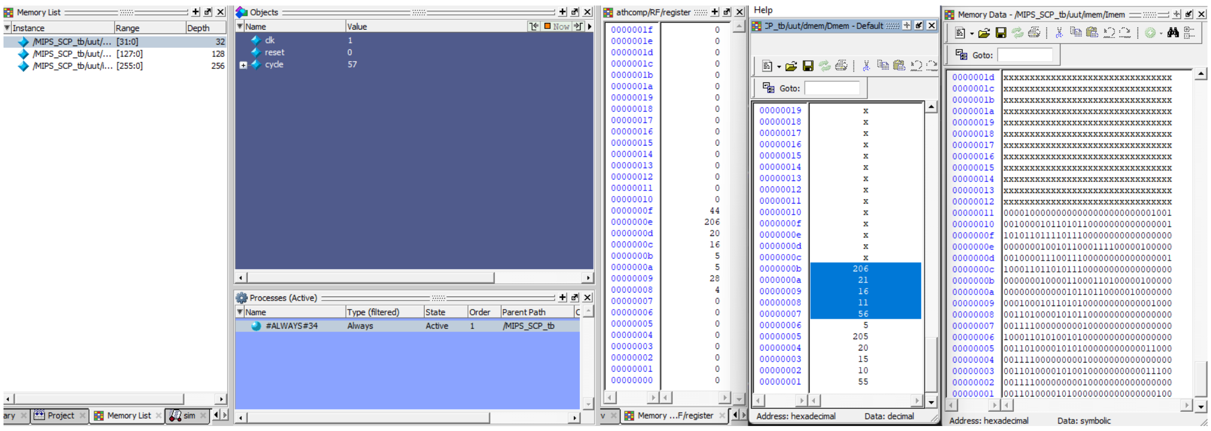
```
Memory List
Instance                  Range     Depth
/MIPS_SCP_tb/uut/...      [31:0]       32
/MIPS_SCP_tb/uut/...      [127:0]     128
/MIPS_SCP_tb/uut/i...     [255:0]     256

Objects
Name     Value
clk      1
reset    0
cycle    90

Processes (Active)
Name          Type (filtered)  State   Order  Parent Path
#ALWAYS#34    Always           Active  1      /MIPS_SCP_tb
```

```
RF/register - Default
0000001f   16
0000001e    0
0000001d    0
0000001c    0
0000001b    0
0000001a    0
00000019    0
00000018    0
00000017    0
00000016    0
00000015    0
00000014    0
00000013    0
00000012    0
00000011    0
00000010    0
0000000f    0
0000000e    0
0000000d    0
0000000c   34
0000000b   10
0000000a   55
00000009   21
00000008    4
00000007    0
00000006    0
00000005    0
00000004   10
00000003    0
00000002   55
00000001    0
00000000    0
```

```
Memory Data - /MIPS_SCP_tb/uut/dmem/Dmem
00000019 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000017 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000016 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000014 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000013 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000011 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000d xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000a xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000008 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000007 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000005 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000004 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000002 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000001 00000000000000000000000000110111
Address: hexadecimal    Data: symbolic
```

```
Memory Data - /MIPS_SCP_tb/uut/imem/Imem
00000025 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000024 00000000000000000000000000000000
00000023 00000011111000000000000000001000
00000022 00000001010000000001000000100000
00000021 00000011111000000000000000001000
00000020 00110100001000100000000000000001
0000001f 00111100000000010000000000000000
0000001e 00000011111000000000000000001000
0000001d 00110100001000100000000000000000
0000001c 00111100000000010000000000000000
0000001b 00010000000000000000000000100010
0000001a 00000001001010100101000000100000
00000019 00010000000000000000000000010010
00000018 00100001011010110000000000000001
00000017 00000001100000000101000000100000
00000016 00000001010000000100100000100000
00000015 00000001001010100110000000100000
00000014 00010101101000000000000000001101
00000013 00000000100010110110100000101010
00000012 00010001011001000000000000000111
00000011 00110100001010110000000000000010
00000010 00111100000000010000000000000000
0000000f 00110100001010100000000000000001
0000000e 00111100000000010000000000000000
0000000d 00110100001010010000000000000000
0000000c 00111100000000010000000000000000
0000000b 00010000100010000000000000010011
0000000a 00110100001010000000000000000001
00000009 00111100000000010000000000000000
Address: hexadecimal    Data: symbolic
```

Figure 11: Fibonacci Sequence



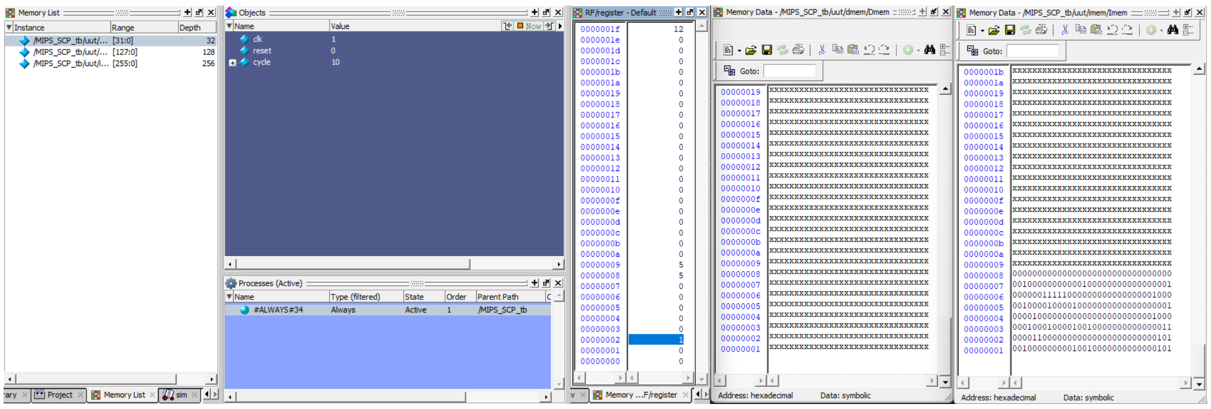
```
Memory List
Instance                  Range     Depth
/MIPS_SCP_tb/uut/...      [31:0]       32
/MIPS_SCP_tb/uut/...      [127:0]     128
/MIPS_SCP_tb/uut/i...     [255:0]     256

Objects
Name     Value
clk      1
reset    0
cycle    57

Processes (Active)
Name          Type (filtered)  State   Order  Parent Path
#ALWAYS#34    Always           Active  1      /MIPS_SCP_tb
```

```
athcomp/RF/register
0000001f    0
0000001e    0
0000001d    0
0000001c    0
0000001b    0
0000001a    0
00000019    0
00000018    0
00000017    0
00000016    0
00000015    0
00000014    0
00000013    0
00000012    0
00000011    0
00000010    0
0000000f   44
0000000e  206
0000000d   20
0000000c   16
0000000b    5
0000000a    5
00000009   28
00000008    4
00000007    0
00000006    0
00000005    0
00000004    0
00000003    0
00000002    0
00000001    0
00000000    0
```

```
Help
CP_tb/uut/dmem/Dmem - Default
00000019     x
00000018     x
00000017     x
00000016     x
00000015     x
00000014     x
00000013     x
00000012     x
00000011     x
00000010     x
0000000f     x
0000000e     x
0000000d     x
0000000c     x
0000000b   206
0000000a    21
00000009    16
00000008    11
00000007    56
00000006     5
00000005   205
00000004    20
00000003    15
00000002    10
00000001    55
Address: hexadecimal    Data: decimal
```

```
Memory Data - /MIPS_SCP_tb/uut/imem/Imem
0000001d xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001a xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000019 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000017 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000016 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000014 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000013 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000011 00001000000000000000000000001001
00000010 00100001011010110000000000000001
0000000f 10101101111011100000000000000000
0000000e 00000001001011000111100000100000
0000000d 00100001110011100000000000000001
0000000c 10001101101011100000000000000000
0000000b 00000001000011000110100000100000
0000000a 00000000000010110110000010000000
00000009 00010001011010100000000000001000
00000008 00110100001010110000000000000000
00000007 00111100000000010000000000000000
00000006 10001101010010100000000000000000
00000005 00110100001010100000000000011000
00000004 00111100000000010000000000000000
00000003 00110100001010010000000000011100
00000002 00111100000000010000000000000000
00000001 00110100001010000000000000000100
Address: hexadecimal    Data: symbolic
```

Figure 12: Array Manipulation



```
Memory List
Instance                  Range     Depth
/MIPS_SCP_tb/uut/...      [31:0]       32
/MIPS_SCP_tb/uut/...      [127:0]     128
/MIPS_SCP_tb/uut/i...     [255:0]     256

Objects
Name     Value
clk      1
reset    0
cycle    10

Processes (Active)
Name          Type (filtered)  State   Order  Parent Path
#ALWAYS#34    Always           Active  1      /MIPS_SCP_tb
```

```
RF/register - Default
0000001f   12
0000001e    0
0000001d    0
0000001c    0
0000001b    0
0000001a    0
00000019    0
00000018    0
00000017    0
00000016    0
00000015    0
00000014    0
00000013    0
00000012    0
00000011    0
00000010    0
0000000f    0
0000000e    0
0000000d    0
0000000c    0
0000000b    0
0000000a    0
00000009    5
00000008    5
00000007    0
00000006    0
00000005    0
00000004    0
00000003    0
00000002    1
00000001    0
00000000    0
```

```
Memory Data - /MIPS_SCP_tb/uut/dmem/Dmem
00000019 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000017 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000016 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000014 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000013 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000011 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000d xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000a xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000008 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000007 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000005 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000004 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000002 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000001 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Address: hexadecimal    Data: symbolic
```

```
Memory Data - /MIPS_SCP_tb/uut/imem/Imem
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001a xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000019 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000017 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000016 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000014 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000013 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000011 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000d xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000a xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000008 00000000000000000000000000000000
00000007 00100000000000100000000000000001
00000006 00000011111000000000000000001000
00000005 00100001000010000000000000000001
00000004 00001000000000000000000000001000
00000003 00010001000010010000000000000011
00000002 00001100000000000000000000000101
00000001 00100000000010010000000000000101
Address: hexadecimal    Data: symbolic
```

Figure 13: Branch Test