Modified Single Cycle Architecture with DSP Unit

ECEN438 - Advanced Computer Architecture

Mohamed Dewidar	211001890
Jaser Kasim	211001801
Karim Fawzy	211000606
Mohamed Tamer	211000379
Youssef Mansy	211001591

Dr. Ahmed Soltan

Jun, 2025

Table of Contents

1. Abstract

This report details the design, implementation, and evaluation of a Modified MIPS Single-Cycle Architecture enhanced with a Digital Signal Processing (DSP) unit. The baseline MIPS single-cycle processor was developed using Vivado 2020.1 and System Verilog, supporting R-, I-, and J-type instructions (e.g., add, or lw, sw, beq, j). To offload high-speed signal-processing tasks, a third-order FIR moving-average filter (coefficients = 1/4) was integrated as a dedicated DSP unit. Input samples (16-bit) were generated in Python using the NumPy library to simulate a noisy sine wave, then supplied to the DSP module via a custom testbench (dsp_tb). Filtered outputs were written back to general-purpose registers and subsequently to data memory. Results demonstrate successful real-time noise reduction and validate correct integration: the processor correctly executes conventional instructions (e.g., arithmetic, memory, branch) and the new DSP instruction to perform FIR filtering within a single cycle.

2. Introduction:

MIPS single-cycle architecture was developed using VIVADO software along with System Verilog. The processor is capable of executing R, I, and J-type instructions including operations like add, OR, load, store, beq, and jump. Core components of the design comprise the Program Counter (PC), Instruction Memory, Register File (RF), Arithmetic Logic Unit (ALU), Data Memory, adders, multiplexers, a sign extension module, shift-left units, and a control unit. To enhance its functional scope, a Digital Signal Processing (DSP) block was added within the MIPS Single Cycle. A DSP instruction, formatted as an R-type, was added to the instruction set. This new instruction enables data from a register to be processed by the DSP unit, with the output written back into another register.

We used a 3rd-order Finite Impulse Response (FIR) filter, and the input samples were generated using a Python script with the NumPy library to simulate a sine wave corrupted with additive noise. The 16-bit values output from this signal were supplied to the DSP module through a dedicated test bench (dsp_tb). The DSP block then successfully filtered out the noise and restored a smooth and clean signal using a moving average filter with 6-bit coefficients. The processed results were stored in 16 general-purpose registers. This paper presents the enhanced MIPS architecture with DSP integration, detailing the design and implementation of the DSP instruction. The results demonstrate the architecture's ability to efficiently perform real-time signal processing, extending the capabilities of the traditional MIPS processor.

3. Traditional MIPS vs Integrating a DSP Unit

Before diving into the methodology, it's important to clarify the rationale behind integrating a DSP unit into a MIPS architecture.

A traditional MIPS processor executes arithmetic and logical operations efficiently but lacks native support for high-speed signal processing. In scenarios where applications demand frequent filtering, transformation, or noise reduction of real-time data—such as in audio, biomedical, or communication systems, general-purpose operations become inefficient both in speed and resource usage.

By integrating a DSP unit directly into the MIPS architecture:

- Specialized tasks like FIR filtering can be offloaded from the ALU, optimizing execution time and reducing instruction count.
- A dedicated instruction can be introduced to interface with the DSP, streamlining the data flow between memory and processing units.
- The overall architecture becomes better suited for mixed workloads, balancing control logic and signal processing within a single cycle model.

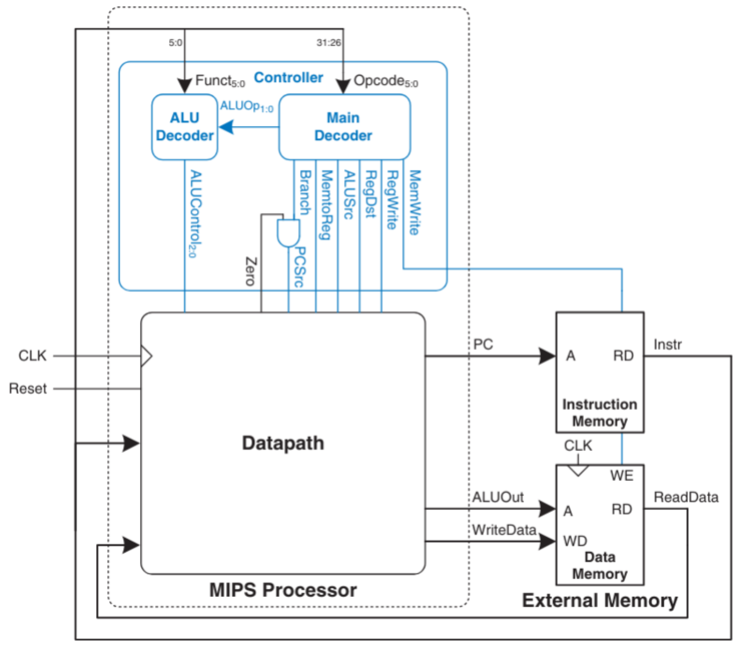
This modification retains compatibility with MIPS fundamentals while enhancing it with domain-specific acceleration, making the architecture more versatile for embedded systems.

4. Methodology:

The development of the modified single-cycle architecture incorporating a DSP unit was carried out in two main phases. The standard single-cycle architecture was implemented in the first phase without any enhancements. This foundational design followed the conventional MIPS architecture, serving as the base for further extensions. The second phase involved adapting the initial design to integrate a DSP unit capable of performing moving average operations using a Finite Impulse Response (FIR) filter. This modification required adjustments to the original structure to support the added DSP functionality.

1.1. Implementation of the MIPS Single Cycle Architecture

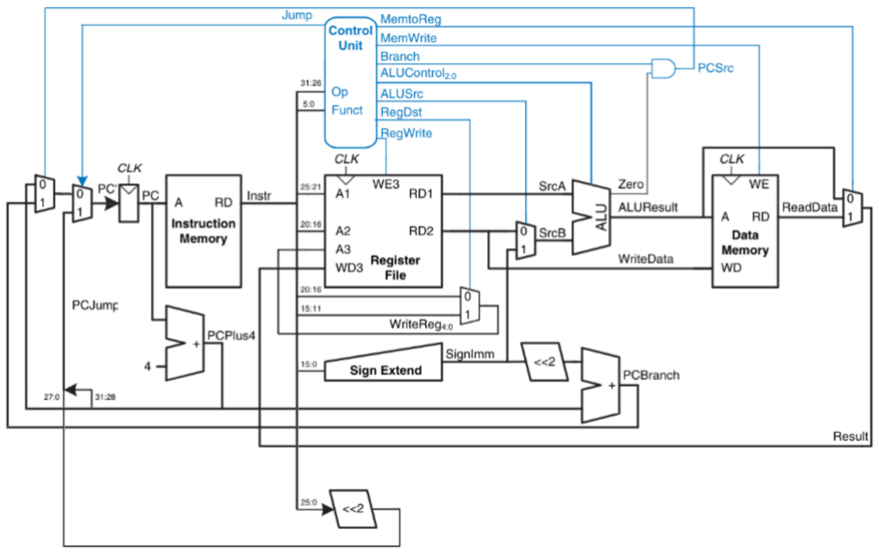
The process will be further broken into two phases, as illustrated in the Figure. Phase one involves building the core MIPS processor, including both the control unit and the data path components. In phase two, the focus shifts to implementing the external memory modules,



specifically, the instruction and data memories, and establishing their connection to the MIPS processor to form a complete system.

1.1.1. MIPS:

The MIPS single-cycle processor comprises two main components: the control unit and the data path, as depicted in the figure. The control unit generates the appropriate control signals required to operate the data path and memory during instruction execution. Meanwhile, the data path defines the route through which data flows to carry out the specified instruction.

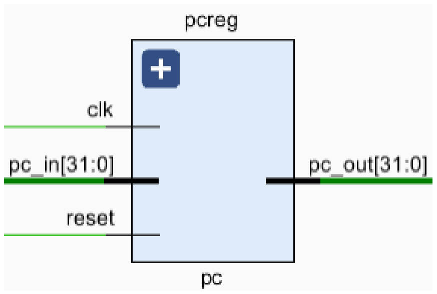


1.1.1.1. Datapath:

The data path of the MIPS single-cycle processor is composed of several fundamental modules, including the Program Counter (PC), an adder, two 32-bit input multiplexers (MUX), two 16-bit input multiplexers, a sign extension unit, a shift-left-by-2 unit, and an Arithmetic Logic Unit (ALU). The data path we intend to implement will support a subset of instructions, specifically R-type, sw, lw, beq, addi, and jump.

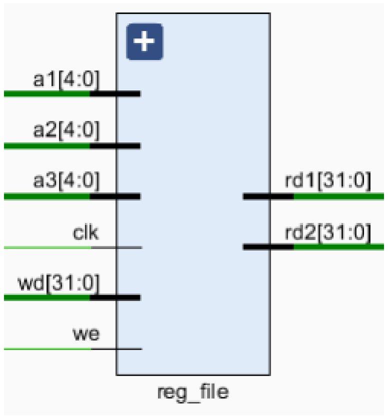
A. PC:

The Program Counter (PC) is a 32-bit register that stores the address of the next instruction to be fetched from the instruction memory. After each instruction is fetched, the PC is incremented by 4 bytes through the use of an adder, as illustrated in the figure.



B. RF:

The Register File (RF) contains 32 general-purpose registers, each 32 bits wide. The use of each register within an instruction depends on the specified input addresses and the associated control



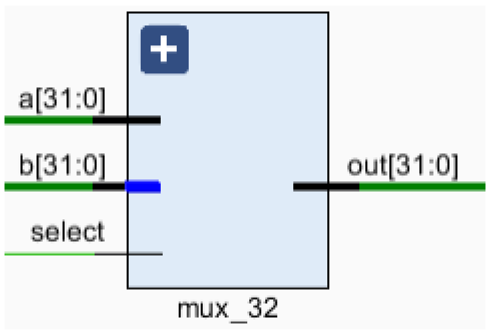
signals.

C. Adder:

The adder in the architecture is designed to perform addition on two 32-bit values. As illustrated in following Figure, adders are utilized in two key operations: incrementing the Program Counter (PC) by 4 after each instruction fetch, and computing the target address for branch instructions.

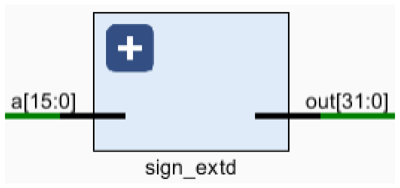
D. MUXs:

Within the architecture, multiplexers (MUXs) are employed to choose between multiple data sources based on control signals. As depicted in the following Figure, MUXs are used to determine the next value of the Program Counter (PC), to select the address for the third read register (RD3), and to choose the second source operand (SrcB) for the ALU.



E. Sign Extend:

The sign extend module functions to extend the sign bit of a 16-bit signed value so that it becomes a 32-bit value, preserving its original sign. As shown in following Figure, this module is specifically used to extend 16-bit immediate values in I-type instructions, enabling them to be correctly processed within the 32-bit architecture.

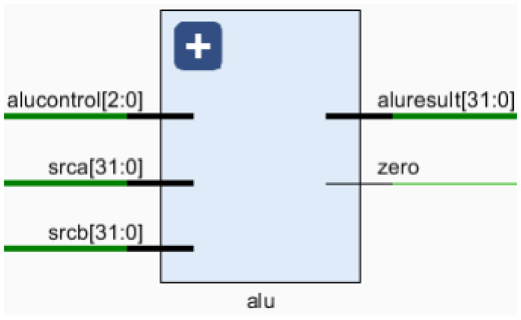


F. Shift Left 2:

The shift left 2 module shifts a binary value two bits to the left, inserting zeros into the least significant bits. As illustrated in Figure, this module plays a role in calculating the branch target address by shifting the sign-extended immediate value before it is added to the updated Program Counter (PC).

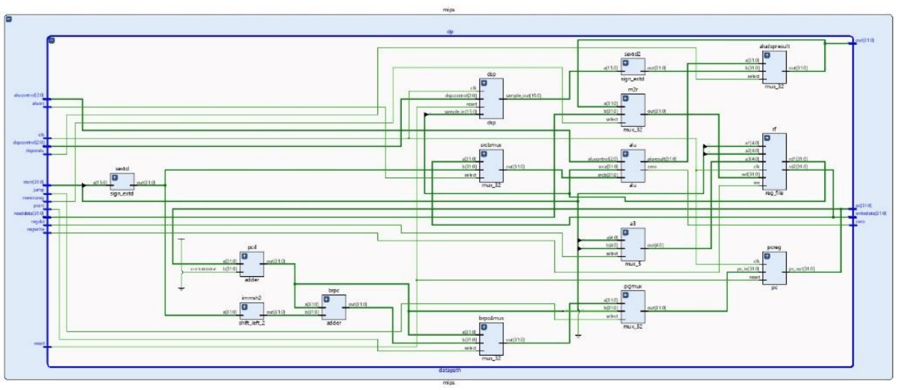
G. ALU:

The ALU module is a critical component responsible for executing arithmetic and logical operations on input operands. In our implementation of the MIPS single-cycle architecture, the ALU supports five distinct operations: subtraction, addition, bitwise AND, bitwise OR, and set-on-less-than (SLT).



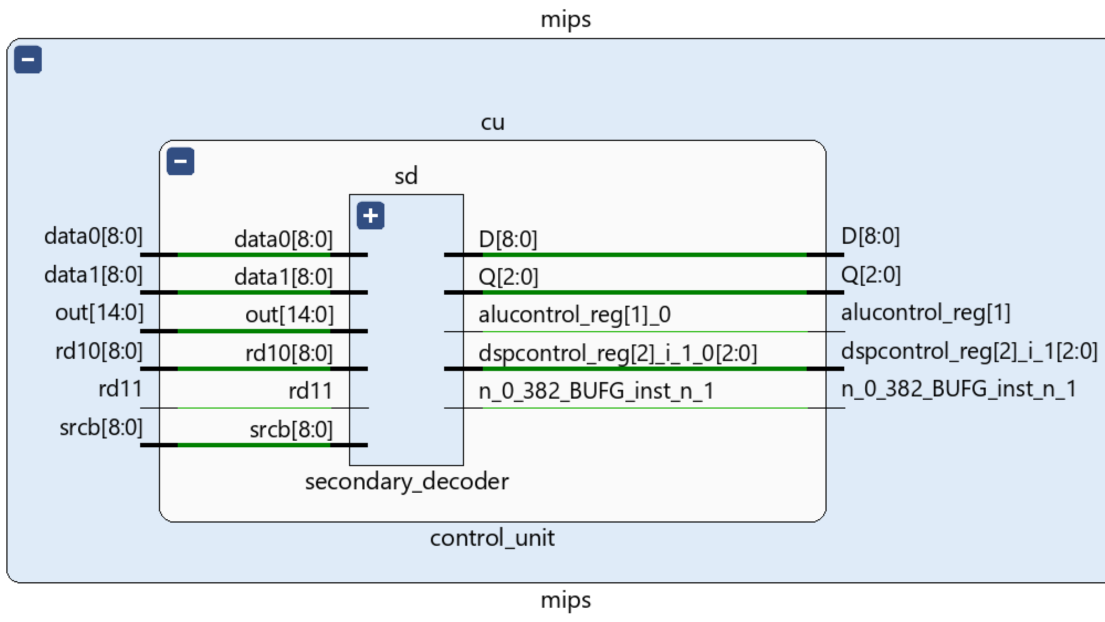
H. Datapath Schematic:

Once all the modules are implemented and interconnected to construct the required data path in Verilog, the RTL schematic can be generated. This schematic provides a visual representation of the designed data path, illustrating how each component interacts within the architecture.



1.1.1.2. Control Unit:

The control unit of a MIPS single-cycle architecture includes two decoders, as illustrated in Figure. The first is the Main Decoder, which generates control signals for all components in the data path, excluding the ALU. The second is the ALU Decoder, which specifically determines the operation to be performed by the ALU by producing the appropriate control signals based on the instruction's function code.



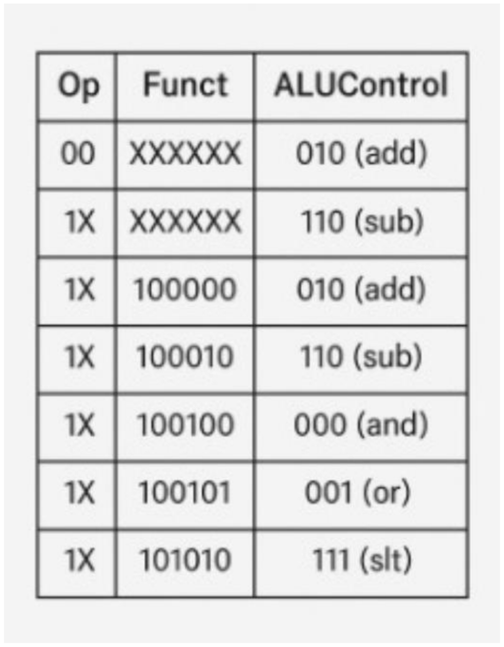
A. Main Decoder:

The Main Decoder functions by receiving the opcode portion of an instruction and using it to determine the required control signals. This decision process is typically based on a predefined truth table that maps each opcode to its corresponding control signal values. The following Table presents the truth table used for our Main Decoder implementation.

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
J	000010	0	X	X	X	0	X	XX	1

B. ALU Decoder:

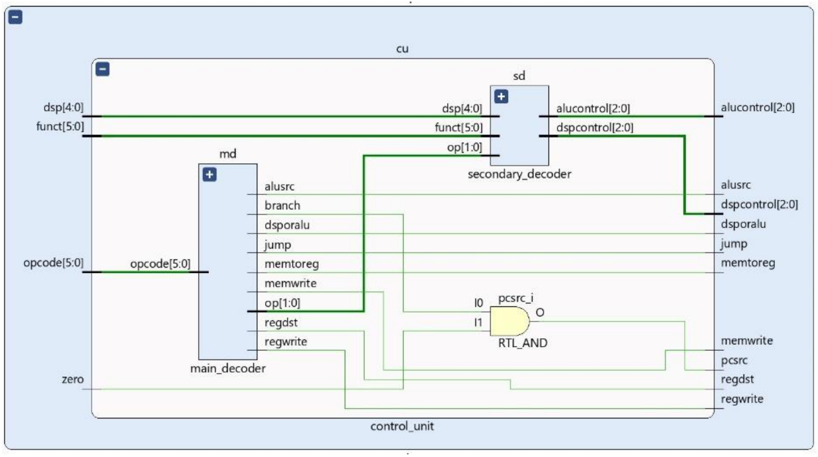
The ALU Decoder operates by receiving the ALUOp signal produced by the Main Decoder and using it to identify the instruction type and corresponding operation through a truth table, as shown in the following Table. Based on this interpretation, the ALU Decoder generates an ALU Control signal, which is then sent to the ALU to specify the exact operation it should perform.



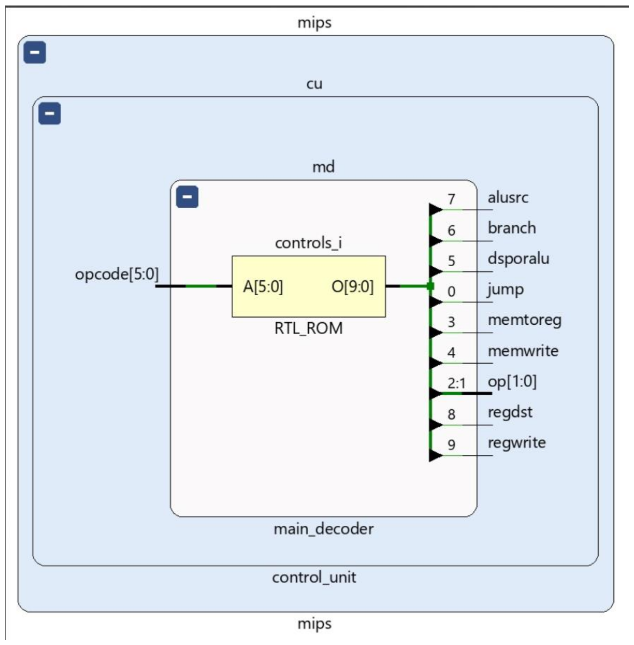
Op	Funct	ALUControl
00	XXXXXX	010 (add)
1X	XXXXXX	110 (sub)
1X	100000	010 (add)
1X	100010	110 (sub)
1X	100100	000 (and)
1X	100101	001 (or)
1X	101010	111 (slt)

C. Control Unit RTL Schematics:

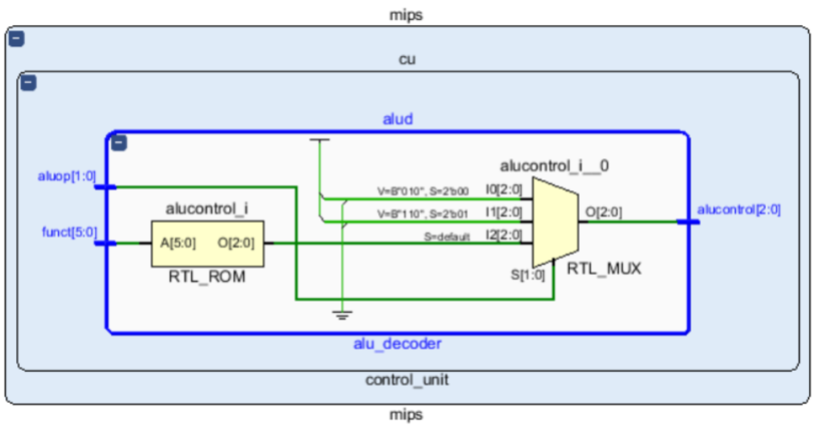
After implementing the main decoder and the ALU decoder and connecting them we can view the RTL schematic of the control unit. In figure 12, we can view the top view of the control unit while figure 13 and 14 we can view the inside of the decoders.



Control Unit Top RTL Schematic



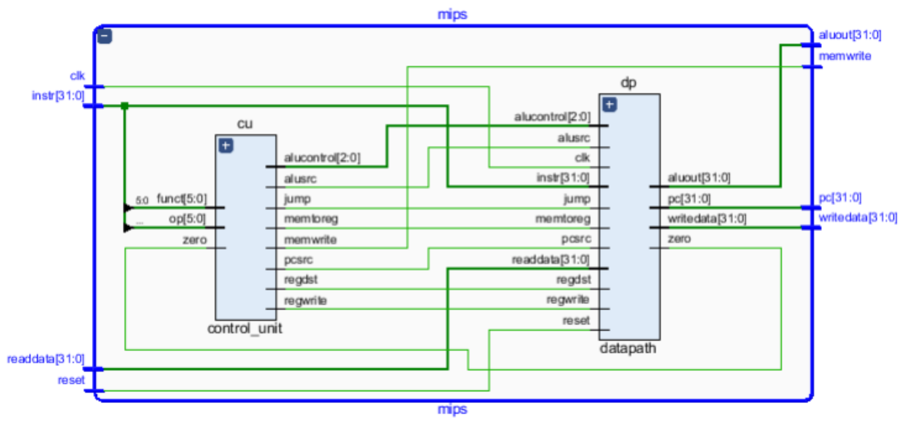
Main Decoder RTL Schematic



ALU Decoder RTL Schematic

1.1.1.3. MIPS Schematic:

Upon completing the implementation of both the data path and the control unit, the two components were integrated by connecting them to enable the data path to receive the necessary control signals from the control unit.



1.1.2. External Memory:

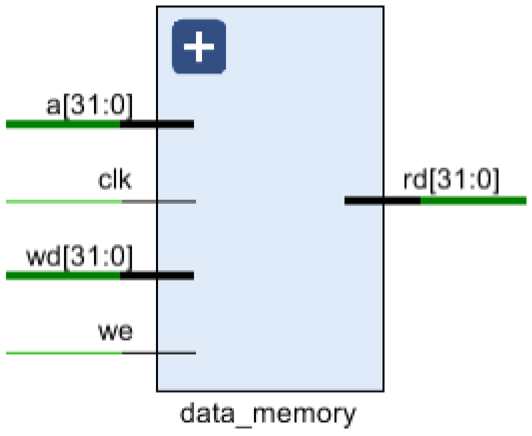
1.1.2.1. Instruction Memory:

The instruction memory module serves the purpose of storing the instructions that the processor is set to execute. It receives as input the address of the instruction to be fetched and outputs the corresponding 32-bit instruction. In our design, the instruction memory has a size of 4KB, providing 1024 addressable memory locations.

a[31:0]
rd[31:0]
instr_mem

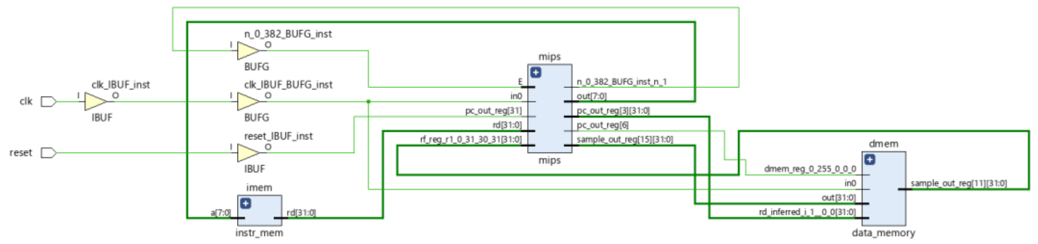
1.1.2.2. Data Memory:

The data memory module is tasked with storing and retrieving data during program execution. It is accessed via load and store instructions. The module receives the address to be accessed, the data to be written (in the case of store operations), and outputs the data read from memory (for load operations). In our design, the data memory is 4KB in size, providing 1024 addressable memory locations—matching the configuration of the instruction memory.



1.1.3. Top:

After completing the implementation of both the MIPS processor and the external memory modules, the final step involves integrating these components to complete the MIPS single-cycle architecture. This is done by connecting the instruction and data memory modules to the MIPS processor, as illustrated in the figure.



1.1.4. Testbench:

To verify the correctness of our implementation, a testbench was developed to supply the required clock (clk) and reset signals. Additionally, we prepared a sequence of instructions—outlined below—that exercises various instruction types to comprehensively test processor functionality. Successful execution is indicated by the following final outcomes:

Hex Code	***Binary Fields Breakdown***	***MIPS Instruction***	***Explanation***
20020005	*001000 00000 00010 0000 0000 0000 0101*	*addi $2, $0, 5*	*Set $2 = 5*
2003000C	*001000 00000 00011 0000 0000 0000 1100*	*addi $3, $0, 12*	*Set $3 = 12*
2067FFF7	*001000 00011 00111 1111 1111 1111 0111*	*addi $7, $3, -9*	*Set $7 = $3 - 9 = 3*
00E22025	*000000 00111 00010 00100 00000 100101*	*or $4, $7, $2*	*$4 = $7 OR $2 = 3 OR 5 = 7*
00642824	*000000 00011 00100 00101 00000 100100*	*and $5, $3, $4*	*$5 = $3 AND $4 = 12 & 7 = 4*
00A42820	*000000 00101 00100 00101 00000 100000*	*add $5, $5, $4*	*$5 = $5 + $4 = 4 + 7 = 11*
10A70008	*000100 00101 00111 0000 0000 0000 1000*	*beq $5, $7, 8*	*Branch if $5 == $7; here, not taken (11 ≠ 3)*

Table of instructions and their translations

1.2. Modified Architecture with DSP Unit:

To extend the implemented architecture by integrating a DSP unit capable of performing a moving average FIR filter operation, several preparatory steps were necessary. First, we determined the required architectural modifications, including the introduction of a new instruction format to support DSP functionality. Next, we implemented the moving average FIR filter algorithm within the DSP unit itself. Finally, we used MATLAB to generate a sample input signal with added noise, which served as a test case for validating the DSP unit's filtering performance.

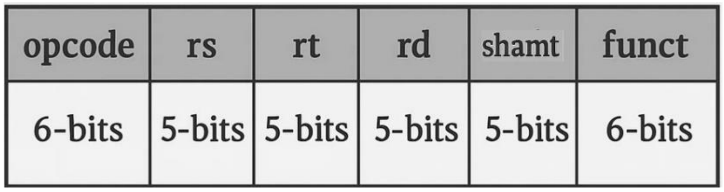
1.2.1. Architecture Modifications:

1.2.1.1. Modified R-type instruction format:

To integrate a DSP unit into our MIPS single-cycle architecture, we needed to introduce a modified instruction format. Our solution involved adapting the existing R-type instruction format by adding a new field labeled "dsp" to support operations handled by the DSP unit. We opted for this approach because we intended for the DSP unit to interact directly with registers—loading input samples from one and storing the filtered output into another.

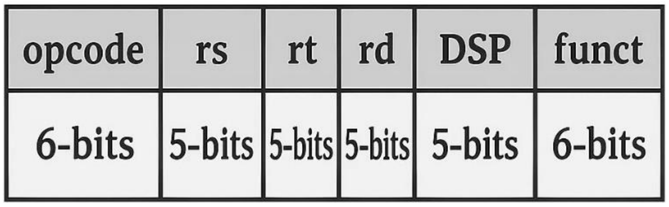
Due to the constraint that MIPS instructions must remain 32 bits in length, and considering that altering the widths of fields like `opcode`, `rs`, `rt`, or `rd` would cause widespread disruption across the architecture, we repurposed the `shamt` field. Since our implementation does not support any shift instructions, this field was unused and thus an ideal candidate for substitution.

In this modified format, the `rs` field supplies the input sample to the DSP, the `rd` field specifies the destination register for the filtered output, and the new `dsp` field determines the DSP operation to be performed. Currently, the only supported operation is a moving average FIR filter, which is encoded as `00001` for simplicity.



opcode	rs	rt	rd	shamt	funct
6-bits	5-bits	5-bits	5-bits	5-bits	6-bits

Old R-type Format

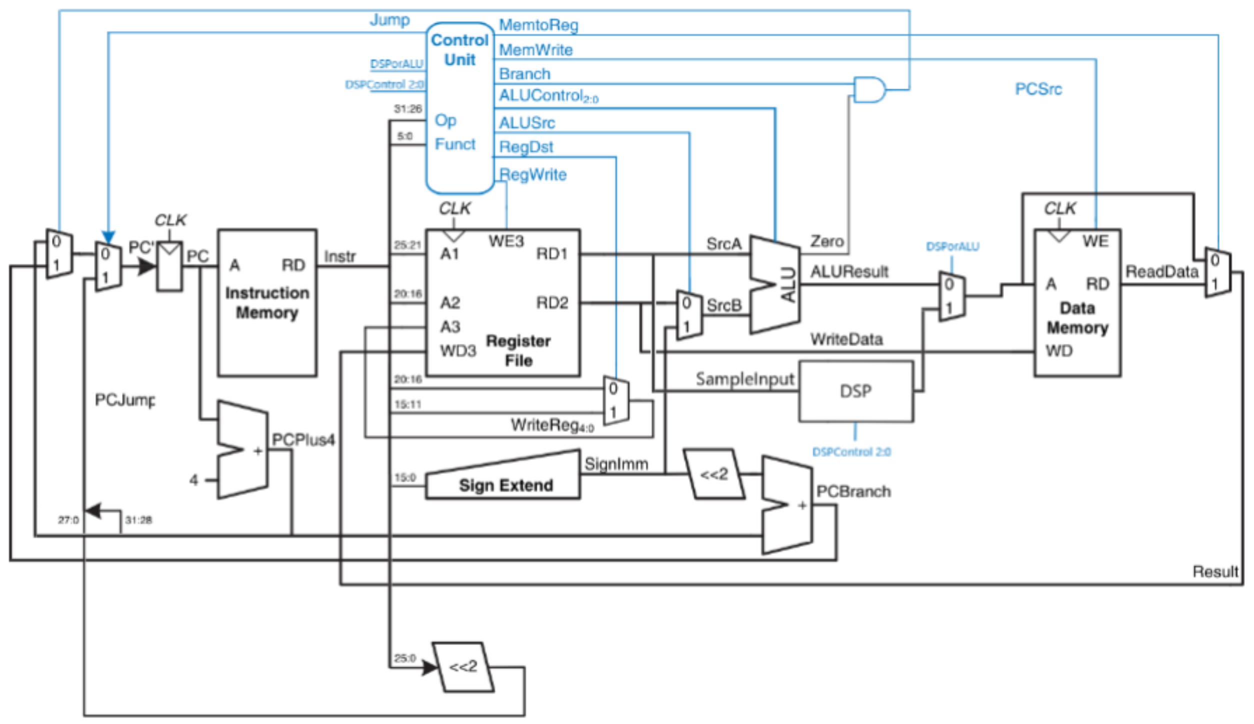


opcode	rs	rt	rd	DSP	funct
6-bits	5-bits	5-bits	5-bits	5-bits	6-bits

New R-type Format

1.2.1.2. Modified Datapath:

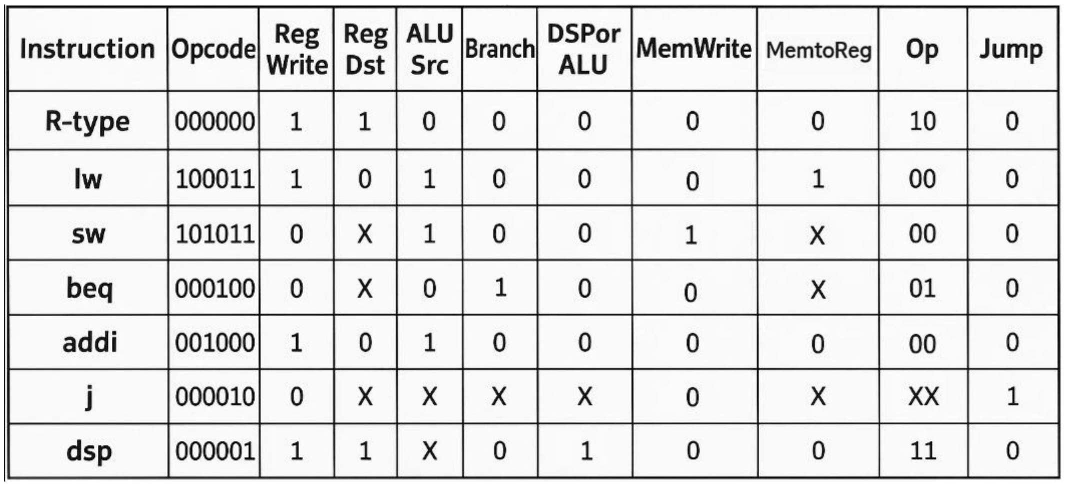
To implement the DSP unit into the architecture, we made necessary modifications to the data path. Specifically, the DSP unit was positioned at the execution stage. It receives input samples directly from the register file, similar to how operands are routed to the ALU. To handle the selection between the DSP and ALU outputs, we introduced a new multiplexer. This multiplexer is controlled by a signal named DSPorALU, which determines whether the final output comes from the ALU or the DSP unit. Figure illustrates the revised data path with these modifications in place.



Single Cycle Architecture

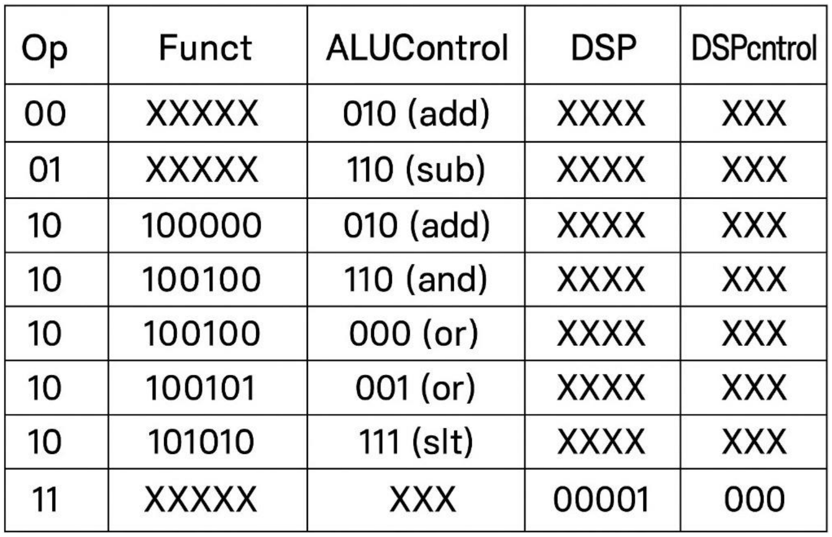
1.2.1.3. Modified Control Unit:

To support the updated data path and the introduction of the new control signals—DSPorALU and DSPControl—the control unit also required modifications. Specifically, the Main Decoder was updated to recognize a new instruction labeled "dsp", which uses the opcode 000001. For this instruction, the Main Decoder sets the ALUOp to 11, signaling the ALU Decoder to defer operation control to the DSP unit. Additionally, a new control signal, DSPorALU, was introduced to manage the selection between DSP and ALU outputs. These changes are reflected in the revised Main Decoder Truth table, as shown in the updated table.



Instruction	Opcode	Reg Write	Reg Dst	ALU Src	Branch	DSPor ALU	MemWrite	MemtoReg	Op	Jump
R-type	000000	1	1	0	0	0	0	0	10	0
lw	100011	1	0	1	0	0	0	1	00	0
sw	101011	0	X	1	0	0	1	X	00	0
beq	000100	0	X	0	1	0	0	X	01	0
addi	001000	1	0	1	0	0	0	0	00	0
j	000010	0	X	X	X	X	0	X	XX	1
dsp	000001	1	1	X	0	1	0	0	11	0

Secondly, the ALU Decoder—now functioning more broadly as a secondary decoder—was enhanced to manage both ALU and DSP operations. To support the new DSP functionality, two new fields were introduced: DSP and DSPControl. These fields are activated when the ALUOp is set to 11, which indicates that the instruction is targeting the DSP unit rather than the standard ALU. The updated truth table for this secondary decoder includes logic for both ALU and DSP instructions and defines how DSPControl is derived based on the funct or dsp field in the instruction.

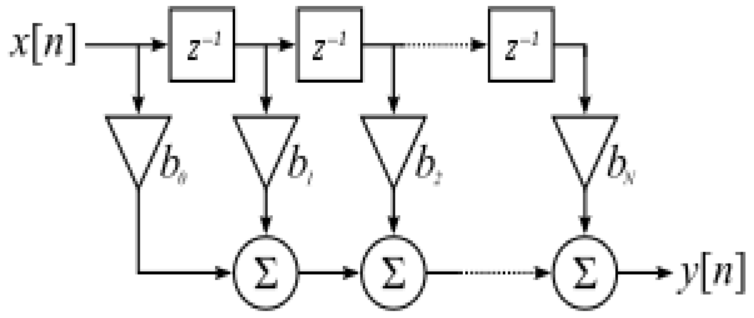


Op	Funct	ALUControl	DSP	DSPcntrol
00	XXXXX	010 (add)	XXXX	XXX
01	XXXXX	110 (sub)	XXXX	XXX
10	100000	010 (add)	XXXX	XXX
10	100100	110 (and)	XXXX	XXX
10	100100	000 (or)	XXXX	XXX
10	100101	001 (or)	XXXX	XXX
10	101010	111 (slt)	XXXX	XXX
11	XXXXX	XXX	00001	000

1.2.2. DSP Unit FIR Filter Operation:

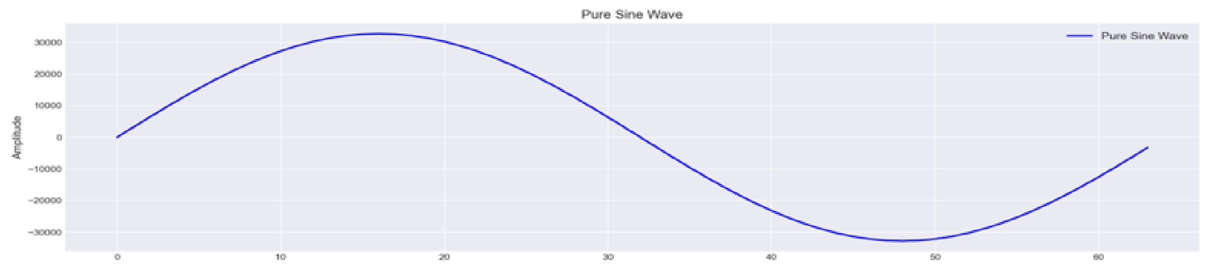
The DSP unit implemented in this project is a third-order, 4-tap Moving Average FIR filter, designed to suppress high-frequency noise in sequential input data. Each coefficient is equal to 0.25, scaled to an integer value of 32 using fixed-point arithmetic. Output normalization is performed by dividing the sum of weighted samples by 128 (shifted right by 7 bits).
To evaluate the DSP's performance, a noisy sine wave was generated using Python and NumPy. The signal was transformed into 16-bit signed samples and injected into the dsp_tb testbench. This testbench directly drove the sample_in input of the DSP module with the corrupted sine wave. As the DSP processed the input, it produced a smoothed output through the sample port, clearly demonstrating noise reduction while preserving the sinusoidal trend. This process can be represented mathematically as $y[n]$=1$N\Sigma x[n-k]N-1k$=0, where x[n] is the input sample, y[n] is the output sample, and 1/N is the coefficient where N is the size of the averaging window (Order+1). Furthermore, because of its simplicity, all the coefficients have the same value and can be easily calculated using the equation b=1$\mathrm{Order}+1$, since in our case we are using a third-order version, the coefficients will just be b=1/(3+1)=0.25.

The DSP operation is controlled via a 3-bit dspcontrol signal:
- 3'b000 and 3'b001: normal operation for start and intermediate samples
- 3'b010: flush/reset, forcing the output to zero

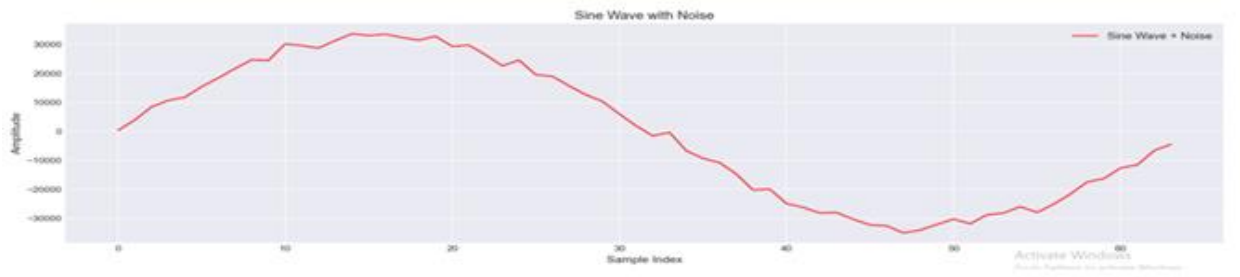


1.2.3. Testing:

After successfully integrating the DSP unit into our single-cycle architecture, the next logical step is to begin testing. To initiate this process, signal samples needed to be generated using Python. In this environment, we constructed a basic sine wave and introduced noise to it, as illustrated in the subsequent figures.



Pure Sine Wave



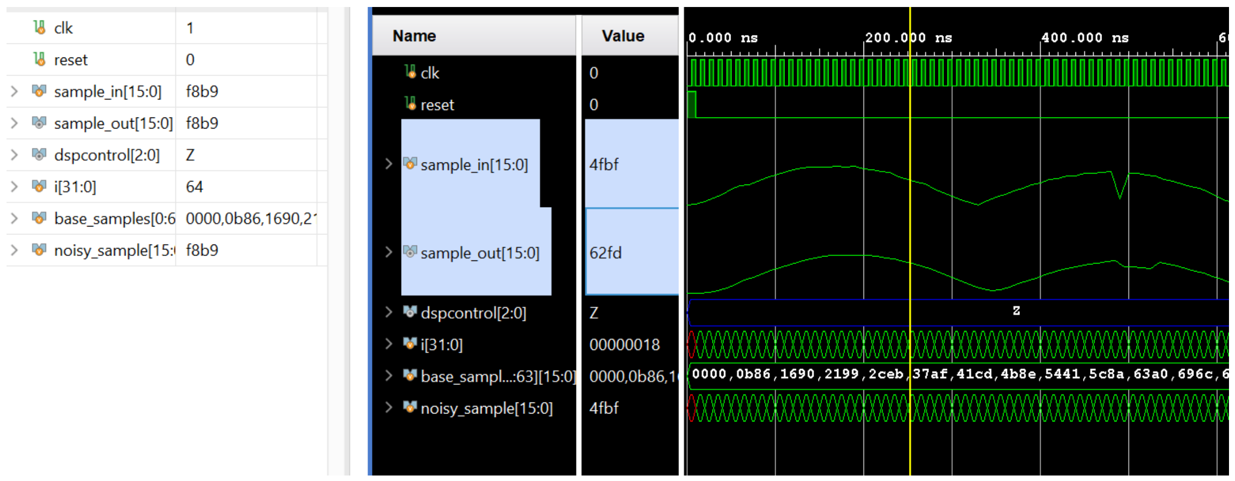
Sine Wave + noise

From the resulting waveform, we selected 32 samples—each 16 bits in size to serve as our input dataset for testing purposes.

1.2.3.1. Testing the DSP Module Individually:

To ensure the correct functionality of the implemented DSP unit, a dedicated testbench was developed to evaluate the unit in isolation using the previously extracted input samples.

The outcomes of this testbench indicate that the DSP operations are functioning as intended. As shown in the figure, the noisy input sine wave is clearly visible, while the corresponding output waveform demonstrates effective smoothing, confirming the DSP unit's proper behavior.



clk	1
reset	0
sample_in[15:0]	f8b9
sample_out[15:0]	f8b9
dspcontrol[2:0]	Z
i[31:0]	64
base_samples[0:6	0000,0b86,1690,2
noisy_sample[15:	f8b9

Name	Value
clk	0
reset	0
sample_in[15:0]	4fbf
sample_out[15:0]	62fd
dspcontrol[2:0]	Z
i[31:0]	00000018
base_sampl...:63][15:0]	0000,0b86,1
noisy_sample[15:0]	4fbf

0.000 ns
200.000 ns
400.000 ns
Z
0000,0b86,1690,2199,2ceb,37af,41cd,4b8e,5441,5c8a,63a0,696c,6

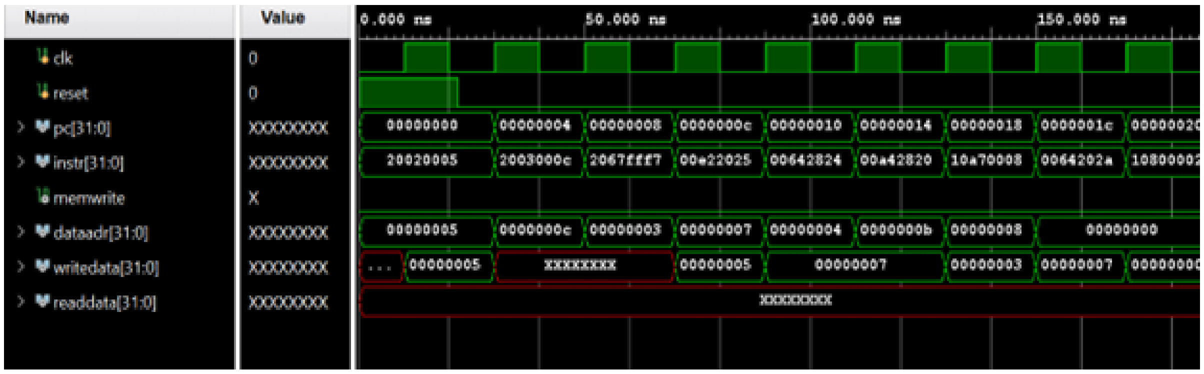
Sample in (Sine Wave + Noise) vs Sample out (DSP flirting sample in)

1.2.3.2. Full System Integration Testing

After verifying the DSP block independently, the next phase involved integrating it into a custom single-cycle MIPS processor using a dedicated dsp instruction. A new memfile.dat was created containing 32 instructions. These instructions used register values preloaded with the sine + noise data (via regfile.dat), simulating a real embedded signal processing scenario.

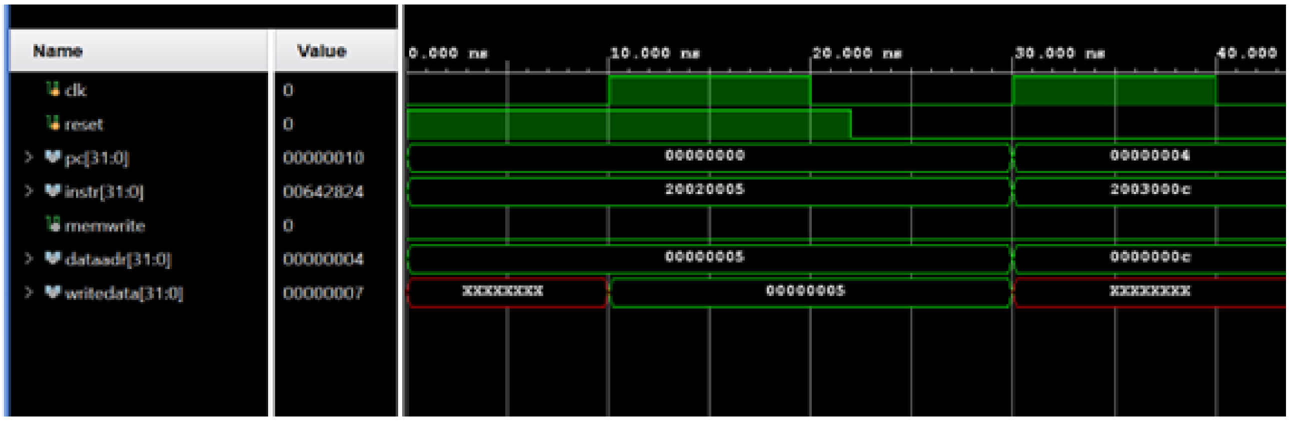
A test bench was constructed to provide only the clock and reset signals. The instruction memory module initialized with the DSP instructions immediately after resetting. However, during the first simulation run (default duration: 1000 ns), the architecture did not complete all instruction cycles. As a result, many register and memory values remained undefined.

To address this, the simulation time was extended to 8000 ns, allowing sufficient time for all 32 instructions to execute. The processor completed its operations at approximately 7222 ns, just before the $finish call, confirming that the full DSP instruction sequence was processed correctly.



Name
Value
0.000 ns
50.000 ns
100.000 ns
150.000 ns
clk 0
reset 0
pc[31:0] XXXXXXXX
00000000 00000004 00000008 0000000c 00000010 00000014 00000018 0000001c
instr[31:0] XXXXXXXX
20020005 2003000c 2067fff7 00e22025 00642824 00a42820 10a70008 0064202a
memwrite X
dataadr[31:0] XXXXXXXX
00000005 0000000c 00000003 00000007 00000004 0000000b 00000008 00000000
writedata[31:0] XXXXXXXX
... 00000005 XXXXXXXX 00000005 00000007 00000003 00000007
readdata[31:0] XXXXXXXX
XXXXXXXX

1000ns Simulation Time



8000ns Simulation Time

The results from both the standalone DSP testbench and full system simulation demonstrate that the DSP unit:

- Successfully filtered noisy input data
- Integrated cleanly into a MIPS processor via a custom instruction
- Produced stable, smooth outputs consistent with expected FIR filter behavior

This confirms that the extended MIPS architecture is capable of performing real-time signal processing tasks, bridging the gap between general-purpose computation and embedded DSP operations.

5. Conclusion:

This project successfully extended a traditional single-cycle MIPS architecture by integrating a dedicated DSP (Digital Signal Processing) unit capable of executing custom filtering instructions. The goal was to enable real-time signal processing capabilities—specifically, smoothing noisy input data using a FIR (Finite Impulse Response) filter—within a minimal hardware pipeline.

The DSP unit was implemented as a 4-tap Moving Average filter using fixed-point arithmetic and was controlled via a 3-bit dspcontrol signal. Standalone verification using a custom dsp_tb testbench confirmed that the DSP module could effectively filter out high-frequency noise from a sine wave signal, producing a cleaner, smoother output.

The modified instruction set architecture supported a new dsp instruction that allowed filtered results to be stored back into the register file. Integration testing showed that the DSP instructions were properly decoded, executed, and synchronized with the rest of the datapath. Simulation results confirmed that the system could process noisy input stored in registers and write filtered output either back into registers or memory.

Overall, the project demonstrates that extending a simple MIPS processor with a custom DSP instruction is not only feasible but also practical for embedded signal processing tasks. This integration allows computational efficiency and real-time filtering without the need for external DSP cores or complex multicore systems. The success of this architecture opens the door to future enhancements, including support for additional DSP operations, variable-length filters, or pipeline optimization for performance scaling.

6. References:

1. Bursky, D. (2004, October 18). DSP extensions turn MIPS CPUs into media stars. Electronic Design. Retrieved from https://www.electronicdesign.com/technologies/embedded/digital-ics/processors/dsp/article/21752051/dsp-extensions-turn-mips-cpus-into-media-stars([Electronic Design][1])

2. EE Times. (2005, May 17). MIPS32 24KE core integrates DSP application-specific extension. EE Times. Retrieved from https://www.eetimes.com/mips32-24ke-core-integrates-dsp-application-specific-extension/([EE Times][2])

3. MIPS Technologies. (2005, July 6). MIPS32 architecture for programmers: Volume IV-e: The MIPS DSP application-specific extension to the MIPS32 architecture (Revision 1.00). Retrieved from https://www.teklib.com/library/mips32-architecture-for-programmers-volume-iv-e/([teklib: tech doc library][3])

4. GCC. (n.d.). MIPS DSP built-in functions. In Using the GNU Compiler Collection (GCC). Retrieved from https://gcc.gnu.org/onlinedocs/gcc/MIPS-DSP-Built-in-Functions.html([GCC][4])

5. EDN. (2004, October 5). DSP extension is available for MIPS architecture. EDN Network. Retrieved from https://www.edn.com/dsp-extension-is-available-for-mips-architecture/([EDN][5])

6. Clarke, P. (2004, October 4). MIPS adds SIMD instructions in DSP extension. EE Times. Retrieved from https://www.eetimes.com/mips-adds-simd-instructions-in-dsp-extension/([EE Times][6])

7. Appendices

This is Python script to create sine signal:

```
Import numpy as np

samples = 64

amplitude = 32767  # Max for 16-bit signed

noise_amplitude = 1000



for i in range(samples):

   angle = 2 * np.pi * i / samples

   sine_val = int(amplitude * np.sin(angle))

   noise = int(np.random.normal(0, noise_amplitude))

   total = sine_val + noise

   print(f"#10 sample_in = 16'sd{total};")
```