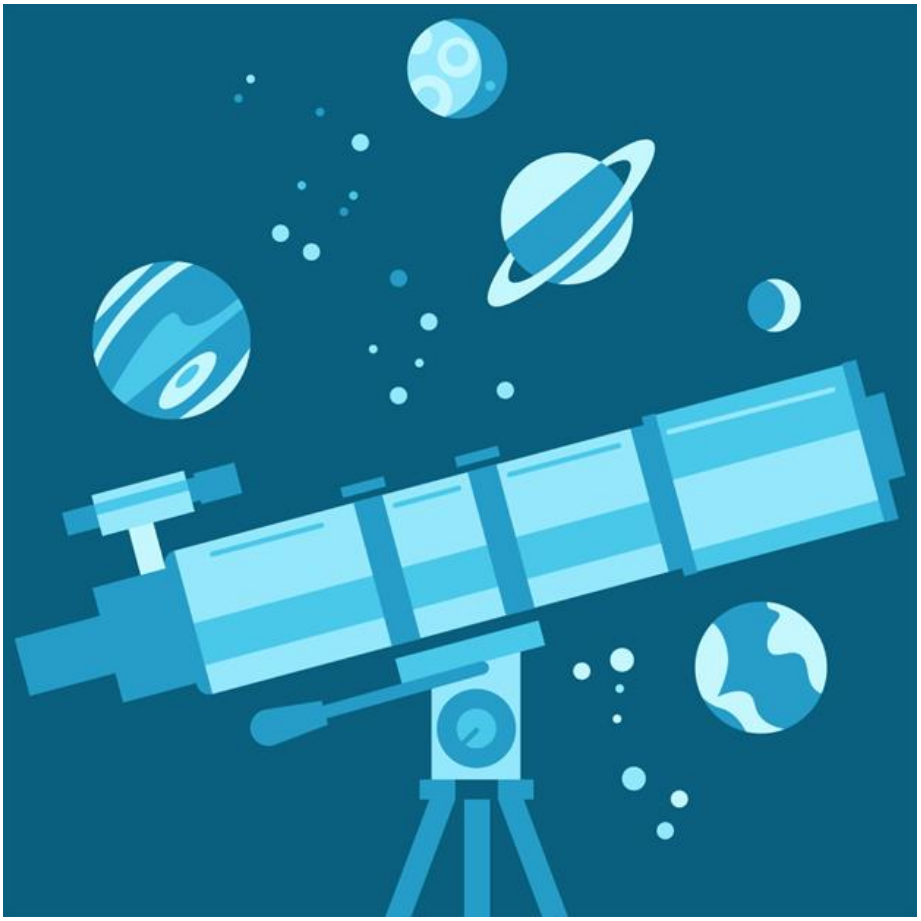
MIPS PROCESSOR FULLY PIPELINE & EXCEPTION



Author

Abdelrahman Mohammed Ali

15 July, 2024

Project Link: [MIPS Processor]

Table of contents

I. Introduction

The document in brief describes the project design, testbenches, a specific two assembly test code for testing the normal functionality and the other for Exceptions handing, snippets from Questa waveform and an elaboration design from VIVADO.

In this document I will show the design and implementation of an efficient processor is a fundamental challenge in computer architecture. This project focuses on the development of a MIPS (Microprocessor without Interlocked Pipeline Stages) processor core that incorporates pipelining, hazard solutions, and exception handling capabilities.

MIPS instructions are simple, fixed-length (32-bit) commands that perform basic operations like arithmetic, memory access, and control flow.

II. The Architecture

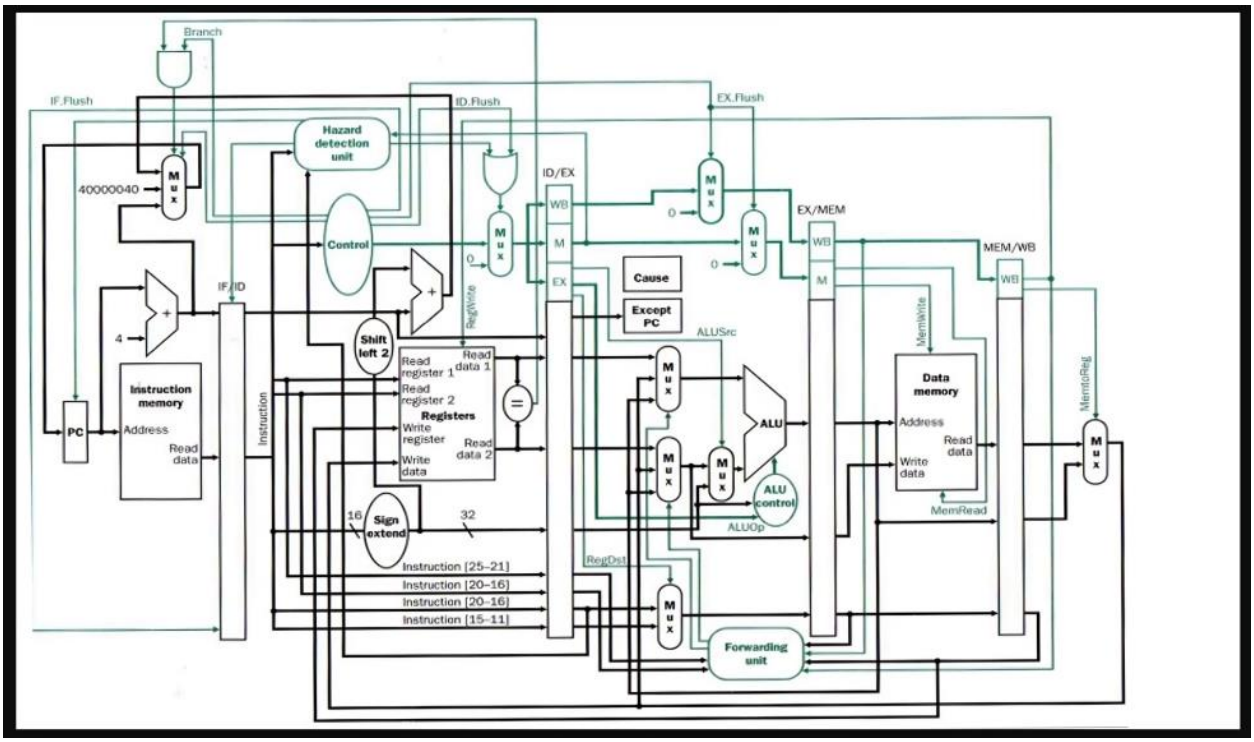


Figure 1

1. Instructions set Architecture (ISA)

The MIPS Instruction Set Architecture (ISA) is a prominent RISC architecture which defines three primary instruction formats: R-format, I-format, and J-format.

a. R-format (Register-format)

The R-format instruction is used for arithmetic and logical operations and for certain control transfer operations.
(ADD, SUB, AND, OR, SLL, SRA, SRL, XOR, SLT)

bits	31-26	25-21	20-16	15-11	10-6	5-0
NO.of bits	6	5	5	5	5	6
R-format	op	rs	rt	rd	shamt	funct
Example	add $t0 $s1 $s2					
decimal	0	17	18	8	0	32
binary	00000	10001	10010	01000	00000	100000

b. I-format (Immediate-format)

The I-format instruction is used for memory access operations (load and store) and for control transfer operations (branches and jumps). (ADDI, LW, SW, BEQ, BNE)

bits	31-26	25-21	20-16	15-11	10-6	5-0
NO.of bits	6	5	5	5	5	6
R-format	op	rs	rt	16-bit immediate/address		
Example	LW $t0, 32 ($s2)					
decimal	35	19	8	32		
binary	10011	10011	01000	0000000000100000		

The MIPS ISA also includes the J-format (Jump-format) instructions, which are used for unconditional jumps. However, for the sake of brevity, I have focused on the R-format and I-format instructions in this response.

2. Pipeline

Implement a 5-stage pipeline (Instruction Fetch, Instruction Decode, Execution, Memory Access, Write Back) to increase instruction-level parallelism and improve the overall throughput of the processor.

2.1. Instruction Fetch (IF): In this stage, the processor fetches the next instruction from the instruction memory and stores it in the instruction register.

2.2. Instruction Decode (ID): The fetched instruction is decoded, and the necessary control signals are generated. Additionally, the operands are read from the register file.

2.3. Execution (EX): The decoded instruction is executed, and the result is computed. For arithmetic and logical instructions, the Arithmetic Logic Unit (ALU) performs the required operations.

2.4. Memory Access (MEM): If the instruction requires memory access (e.g., load or store), the memory operation is performed in this stage.

2.5. Write Back (WB): The result of the executed instruction is written back to the register file.

The pipelining technique allows the processor to execute multiple instructions concurrently, improving the overall throughput and reducing the latency of individual instructions.

3. Stalling

Stalling: is to make the Instruction with no effect in other word it's **nops**

```
if (ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
      (ID/EX.RegisterRt = IF/ID.RegisterRt)))
      stall the pipeline
```

How to Stall:

1. Stop PC from writing any new value
2. Stop IF/ID pipeline from writing any new value
3. Set all Control signal to zeros except the Branch (`ControlSignalSelector`)

4. Hazard

i. Data Hazard

Data hazards arise when an instruction depends on the result of an earlier instruction that has not yet been produced. The processor can use forwarding (also known as bypassing) to resolve data hazards by providing the necessary data directly from the execution stage or memory stage to the next instruction.

Solution:

The values in the pipeline registers show that the desired value is available before it is written into the register file. We assume that the register file forwards values that are read and written during the same clock cycle, so the add does not stall, but the values come from the register file instead of a pipeline register.

This operation called "Forwarding"

```
always @(*) begin
    if (
        EX_MEM_RegWrite
        && (EX_MEM_RegisterRd != 0)
        && (EX_MEM_RegisterRd == ID_EX_RegisterRs)
    ) begin
        ForwardA <= 2'b10;
    end
    else if (
        MEM_WB_RegWrite
        && (MEM_WB_RegisterRd != 0)
        && ( MEM_WB_RegisterRd == ID_EX_RegisterRs)
    ) begin
        ForwardA <= 2'b01;
    end
    else begin
        ForwardA = 2'b00;
    end
end
```

```
always @(*) begin
    if (
        EX_MEM_RegWrite
        && (EX_MEM_RegisterRd != 0)
        && (EX_MEM_RegisterRd == ID_EX_RegisterRt)
    ) begin
        ForwardB <= 2'b10;
    end
    else if (
        MEM_WB_RegWrite
        && (MEM_WB_RegisterRd != 0)
        && (MEM_WB_RegisterRd == ID_EX_RegisterRt)
    ) begin
        ForwardB <= 2'b01;
    end
    else begin
        ForwardB <= 2'b00;
    end
end
```

Figure 2

ii. Control Hazard

Control hazards occur when the outcome of a branch or jump instruction is not known until the execution stage.

To address control hazards, the processor may use techniques like branch prediction, speculative execution, and branch delay slots.

Solution:

Assume the branch is not taken, if this assumption is true then go ahead and if it's not true then we flush current Instruction in IF/ID pipeline stage

To flush instructions in the IF stage, we add a control line, called IF_Flush, that zeroes the instruction fi eld of the IF/ID pipeline register. Clearing the register transforms the fetched instruction into a nop, an instruction that has no action and changes no state.

When the Branch is taken or Exception then IF_Flush set to one.

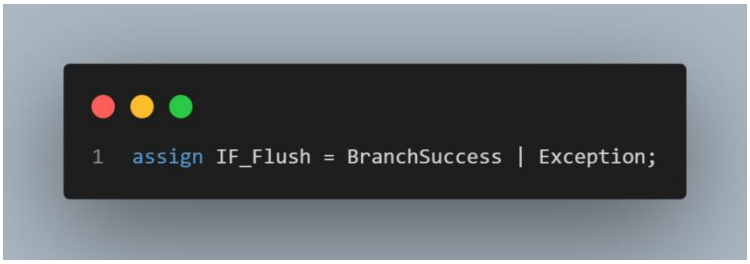


Figure 3

5. Exceptions / Interrupts

They were initially created to handle unexpected events from within the processor, like arithmetic overflow or Invalid Instruction.

In brief when Exception occurs, save the current Instruction Address in Exception Program Counter (EPC), then PCNext take Address from Exception vector address

Exception type	Exception vector address (in hex)
Undefined instruction	$8000\ 0000_{hex}$
Arithmetic overflow	$8000\ 0180_{hex}$

Figure 4

NOTE: for simplicity verification I make the Exception Address (0000 0100)hex

III. Assembly

1. Normal Test Code

- It initializes an array with 6 elements and then performs operations on the array elements, such as:
  - Loading values from the array into registers
  - Performing arithmetic operations on the array elements
  - Storing the results back into the array

```
N_O_P                 // 00000000 (1)
addi t2 $zero 0x5   // 200A0005 (2) max
addi t3 $zero 0x5   // 200B0005 (3) min
addi s0 $zero 0x5   // 20100005 (4) sum
//// Storing elements in the array ////
addi t7 $zero 0x5   // 200F0005 (5)
sw t7 0x0 (t0)      // AD0F0000 (6)
addi t7 $zero 0x9   // 200F0009 (7)
sw t7 0x4 (t0)      // AD0F0004 (8)
addi t7 $zero 0x19  // 200F0019 (9)
sw t7 0x8 (t0)      // AD0F0008 (10)
addi t7 $zero 0x12  // 200F0012 (11)
sw t7 0xC (t0)      // AD0F000C (12)
addi t7 $zero 0x1   // 200F0001 (13)
sw t7 0x10 (t0)     // AD0F0010 (14)
addi t7 $zero 0x15  // 200F0015 (15)
sw t7 0x14 (t0)     // AD0F0014 (16)
////////////////////////////////////
// for loop
        addi t1 $zero 0x1   // 20090001 (17) i = 1
        addi t6 $zero 0x6   // 200E0006 (18) arr size = 6
    // Loading element from memory
loop:   addi t0 t0 0x4      // 21080004 (19)
        lw t5 0x0 (t0)      // 8D0D0000 (20) // t5 = arr[i]
        add s0 s0 t5        // 020D8020 (21) // sum += arr[i]
        slt t8 t2 t5        // 014DC02A (22) // if(t2 < t5) => t8 = 1;
        slt t9 t5 t3        // 01ABC82A (23) // if(t5 < t3) => t9 = 1;
        addi t1 t1 0x1      // 21290001 (24) // i = i + 1;
        beq t8 $zero NotMax // 13000001 (25)
        add t2 $zero t5     // 000D5020 (26) // max = arr[i]
NotMax: beq t9 $zero NotMin // 13200001 (27)
        add t3 $zero t5     // 000D5820 (28) // min = arr[i]
NotMin: bne t1 t6 loop      // 152EFFF5 (29) // loop condition (i < arr.size)
```

2. Exception Test Code

- This file contains MIPS assembly code that tests the exception handling capabilities of the processor.
- It includes instructions that should generate various exceptions, such as:
  - overflow
  - Illegal instructions (e.g., INVALID_INST)
- The code checks that the processor correctly detects and handles these exceptions, jumping to the exception handler.
- It also verifies that the processor correctly saves the necessary state (e.g., program counter, register values) when an exception occurs.

```
// Overflow Exception
0x0000    NOP                 // 00000000
0x0004    addi s1 s1 0x0A16   // 2231FFFF
0x0008    addi s5 s5 0x0AAA   // 22B50AAA
0x000C    add s0 s0 s0        // 02108020 // Overflow
0x0010    addi t0 t0 0x4      // 21080004
0x0014    sw s0 0x0 (t0)      // AD100000
// ...
// ...
// ...
// Invalid Exception
0x00DC    addi t2 t2 0x0AAA   // 214A0AAA 214A0100 => 1024 -> 1027
0x00E0    SWL $s5 0x8404 $s0  // AA158404 // Invalid Instruction becouse
SWL is not supported
0x00E4    addi t0 t0 0x4      // 21080004
0x00E8    sw s0 0x0 (t0)      // AD100000
// ...
// ...
// ...
// Exception Address
0x0100    bne t2 s5 0xFFF6    // 1555FFF6 offcet =>  -10
```

IV. QuestaSim Snippets (Result)

a. normal testbench Simulation Result

1. Registers t2 represent MAX => regmem[10]
2. Registers t3 represent MIN => regmem[11]
3. Registers s0 represent SUM => regmem[16]

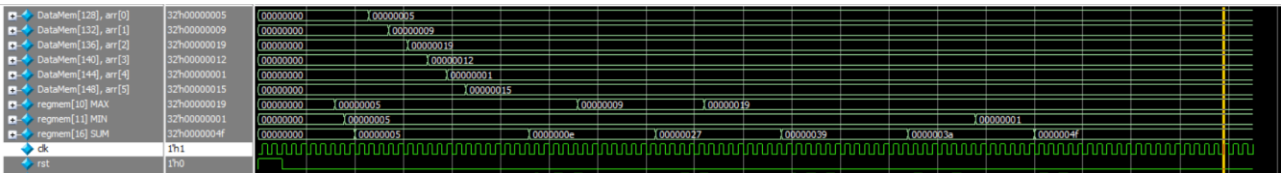


Figure 7

b. Exception testbench Simulation Result

1. EPC Hold the current Instruction Address
2. EX_Flush set to one when overflow
   - All Control signal in MEM stage set to zeroes
3. ID_Flush set to one when Invalid Instruction (Not supported)
   - All Control signals set to zeroes in EX Stage
4. Exception signal set to one
   - Next Instruction should be at Exception Address

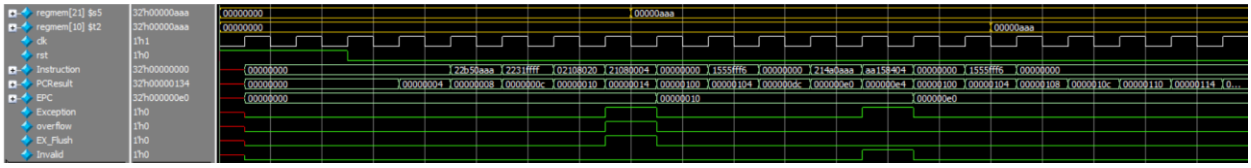


Figure 5

V. VIVADO Elaboration

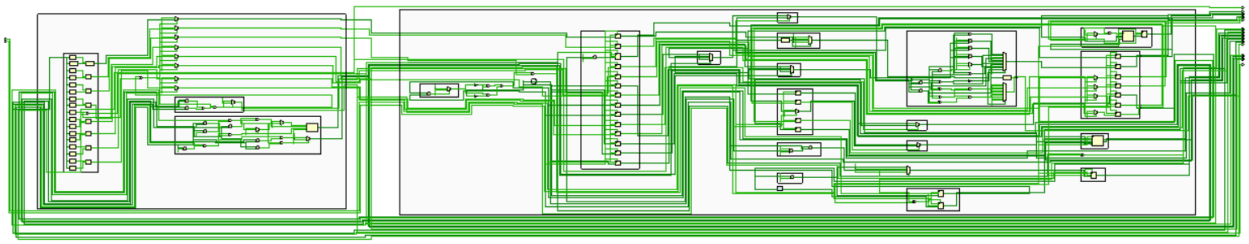


Figure 6

VI. References

Computer Organization and Design- The HW_SW Interface 5th edition - David A. Patterson, John L. Hennessy