UNIVERSITÀ
DI TRENTO

Analysis and Implementation of a Pipelined MIPS 16 Processor

Advanced Computing Architectures

Abdelhakim Rabia

29/07/2024

MIPS overview

My project is a pipelined MIPS processor implemented in Vivado 2023.2 using VHDL. I started from a Single-Cycle MIPS CPU design with one clock cycle per instruction and upgraded it to a pipelined structure. The microprocessor is a MIPS 16 microarchitecture, which means that the width of the instructions and data fields are 16-bits. The four-digit Seven Segment Display and eight LEDs for data display are the main drivers for the implementation of MIPS 16. By doing this, the on-chip debugging procedure is made simpler and alternative multiplexing schemes for signal presentation (32 bits) are avoided.

The report will describe the implemented instructions, followed by a detailed description of the single-cycle MIPS processor, and changes needed to transform it to pipeline.

Instructions

The width of both instructions and data are 16-bits and their formats are according to the formats presented in the MIPS32 ISA, the width of each field being the only difference. The 3 instruction formats are the following:

1. R-type instruction, represented in Figure 1, has six components: the operation code (opcode), the first source register (rs), the second source register (rt), the destination register (rd), the shift amount (sa) and the function bits. ALU operations based on registers are performed by these instructions. Locations in the register file provide the two operands as well as the result's destination. The two addresses that the register file needs to read are indicated by the rs and rt register addresses. Rd specifies the register file's destination (write) register. According to the MIPS standard, each instruction's ALU operation is identified in the function field, and the opcode is 0. Three bits are used to encode the function field. This indicates that a maximum of 15 instructions can be executed by the CPU.

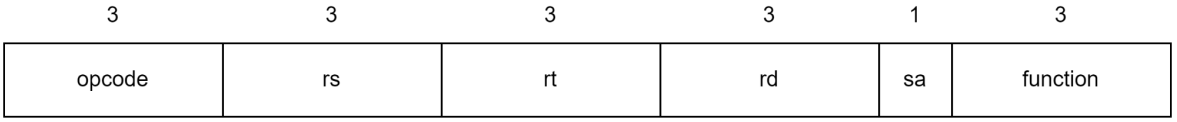

Figure 1. R-type instruction format

2. I-Type Instruction, shown in Figure 2, includes 4 components: the operation code (opcode), source register (rs), destination register (rt) and address/immediate value. Either load/store, branch, or immediate ALU operations may be included in these instructions. These instructions specify the locations of two register files in addition to an immediate value that can be either an operand or an address. In this instance, the instruction to be performed is uniquely encoded by the opcode. The two addresses that the register file needs to read are indicated by the rs and rt register addresses. The destination register in this instance is indicated by rt.

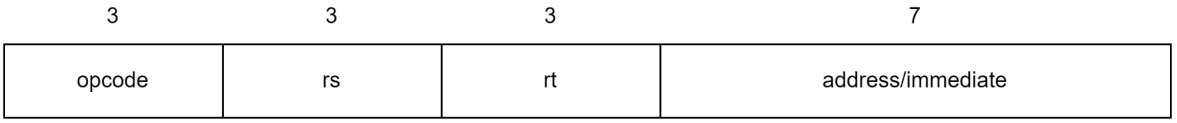

Figure 2. I-type instruction format

3. J-Type Instruction, or Jump instructions, have 2 components shown in Figure 3. The opcode uniquely encodes the instruction to be executed and the rest 13 bits are devoted to a 13-bit jump destination field.

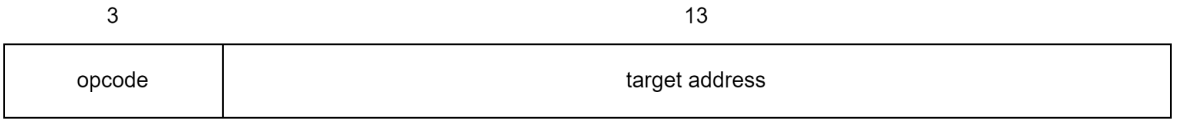

Figure 3. J-type instruction format

Table 1 below presents the instructions, of each type, that will be implemented on the MIPS 16 processor, with the control signals defined as well:

Instruction	Opcode *Instr(15-13)*	RegDst	ExtOp	ALUSrc	Branch	BrNE	Jump	MemWrite	MemtoReg	RegWrite	ALUOp (1:0)	func *Instr(2-0)*	ALUCtrl (2:0)
add	000	1	x	0	0	0	0	0	0	1	00(R)	000	000(+)
Sub	000	1	x	0	0	0	0	0	0	1	00(R)	001	001(-)
SLL	000	1	x	0	0	0	0	0	0	1	00(R)	010	101(<<l)
Srl	000	1	x	0	0	0	0	0	0	1	00(R)	011	110(>>l)
and	000	1	X	0	0	0	0	0	0	1	00(R)	100	010(&)
Or	000	1	X	0	0	0	0	0	0	1	00(R)	101	011(|)
Xor	000	1	x	0	0	0	0	0	0	1	00(R)	110	100(^)
Sra	000	1	X	0	0	0	0	0	0	1	00®	111	111(>>a)
Addi	001	0	1	1	0	0	0	0	0	1	01(+)	X	000(+)
Lw	010	0	1	1	0	0	0	0	1	1	01(+)	X	000(+)
Sw	011	0	1	1	0	0	0	1	X	0	01(+)	X	000(+)
Beq	100	X	1	0	1	0	0	0	X	0	10(-)	X	001(-)
Ori	101	0	0	1	0	0	0	0	0	1	11(|)	X	011(|)
Bne	110	X	1	0	0	1	0	0	X	0	10(-)	X	001(-)
j	111	x	x	x	X	0	1	0	x	0	xx	x	Xxx

Table 1. MIPS 16 instructions

MIPS Processor Description

For a MIPS processor, an instruction execution cycle has the following phases: IF (Instruction Fetch), ID (Instruction Decode), EX (Execute), MEM (Memory) and WB (Write Back). The datapath shown in figure 4 presents how each of the stages has to work with the next stage and execute a specific task to ensure the accurate execution of an instruction for a MIPS 32. For MIPS 16, the datapath is the same, the only difference is the number of bits the signals have.

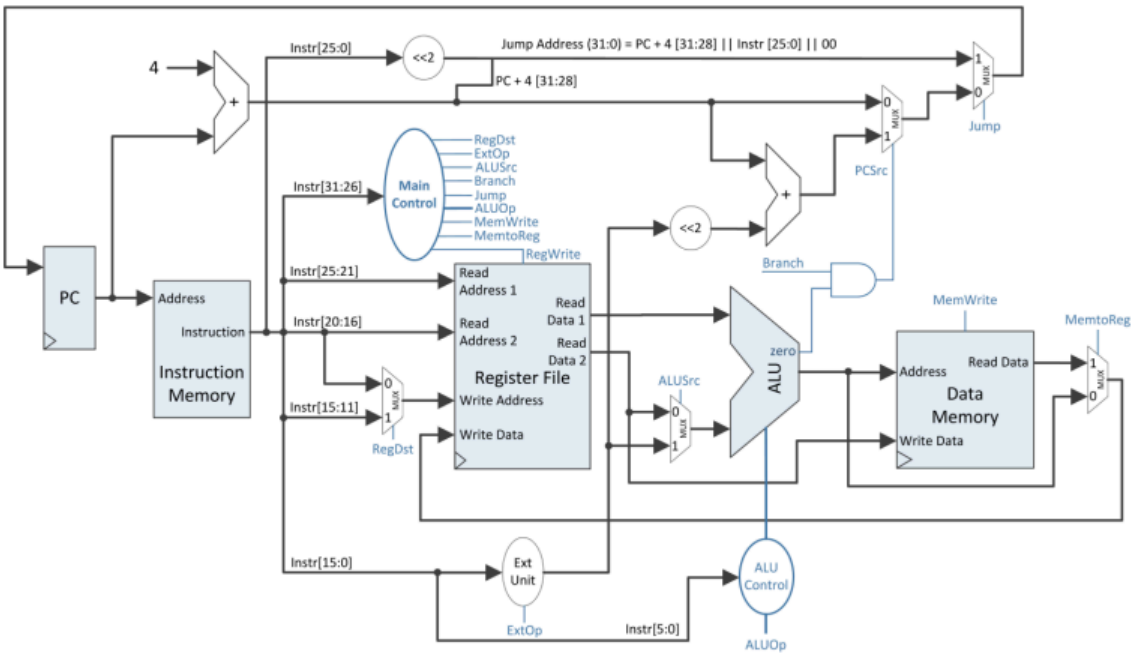

Figure 4. MIPS 32 Single-Cycle Data-Path with control

IF stage

The first stage, IF, focuses on tracking the processor's position within the current program's instruction stack and sending the next instruction through the datapath. This unit consists of the following components: Program Counter (PC), Instruction Memory and Adder.

Additionally, there are 2 multiplexers for selecting the next instruction address (PCSrc or Jump). Typically, the instruction to be performed and the address of the next sequential instruction are sent as outputs by the IF unit. The branch target address, the jump address, and the control signals that determine the next instruction address must all be received by the IF unit as inputs when it comes to jump or branch instructions.

The IF unit receives several inputs, including a clock signal for the Program Counter (PC), a branch target address, a jump address, a Jump Control signal, and a PCSrc Control signal for branching. The outputs of the IF unit are the instruction to be executed by the MIPS processor and the address of the next sequential instruction (PC + 1). The control signals dictate specific operations: if the Jump signal is set to 1, the PC is updated to the jump address; if the Jump signal is 0, the PC's next value is determined by the PCSrc signal. When PCSrc is 0, the PC is incremented by 1; when PCSrc is 1, the PC is set to the branch target address.

In the case of the MIPS 16 data-path components, the characteristics for the Program Counter is a 16-bit edge triggered D flip-flop. For Instruction Memory there is one input bus: Instruction Address, one output bus: Instruction Data, with the memory word of 16-bit (selected by instruction address) and no control signals.

In the IFetch VHDL file, I added an example execution of a program:

1. Initialize R1 with 10: *addi $1, $0, 10*

2. Initialize R2 with 15: *addi $2, $0, 15*
3. Store R2 - R1 in R3: *sub $3, $2, $1*
4. Store R2 + R1 in R4: *add $4, $2, $1*
5. Store value of R3 at memory address 6 + R1: *sw $3, 6($1)*
6. Load from address R2 + 1 to R2: *lw $2, 1($2)*
7. Shift right R1 by 1: *srl $1, $1, 1*
8. Branch if R1 == R2: *beq $1, $2, 1*
9. Jump to instruction 4: *j 4*
10. Store value of R1 at address R2 + 2: *sw $1, 2($1)*
11. Subtract R2 from R4 and store in R4: *sub $4, $4, $2*
12. Shift left R1 by 1: *sll $1, $1, 1*
13. Branch if R1 == R4: *beq $1, $4, 1*
14. Jump to instruction 11: *j 11*
15. Store value of R1 at address 5: *sw $1, 5($0)*

ID stage

In the Instruction Decode (ID) stage, the processor decodes the fetched instruction to understand what operation is required. This involves interpreting the instruction's opcode and identifying the source and destination registers involved. The values from the source registers, specified by the instruction, are read from the register file. This stage also involves the preparation of data and control signals needed for the execution of the instruction.

The ID (Instruction decode) unit consists of a Register File, a Multiplexer and a Sign or Zero Extender (if it is 1, then sign extension; else zero extension).

The ID unit outputs the Read Data 1, Read Data 2, and Extended Immediate data fields that the processor uses in the subsequent execution stages. The shift amount is utilized as an additional input to the ALU for shift operations, and the function field is also sent to the ALU Control Unit.

The ID unit has several key inputs: a clock signal used for Register File Writes, a 16-bit instruction, and a 16-bit Write Data for the Register File. It also receives control signals, which include RegWrite (the Write Enable signal for the Register File), RegDst (which selects the write address for the Register File), and ExtOp (which selects between Sign and Zero extension of the immediate field). The outputs of the ID unit consist of 16-bit Read Data 1 from the rs address register, 16-bit Read Data 2 from the rt address register, a 16-bit Extended Immediate, a 3-bit function field of the instruction, and a 1-bit shift amount for R-type shift instructions. The control signals have specific functions: when RegDst is set to 1, the Write Address for the Register File is the rd field of the instruction; when set to 0, it is the rt field. When RegWrite is set to 1, it writes the value provided by the Write Data Signal into the Write Address Register in the Register File. ExtOp determines the extension type of the 16-bit immediate, performing Zero Extension when set to 0 and Sign Extension when set to 1.

The Main Control unit is also part of the ID stage. For MIPS 16, the input of the Main Control unit consists in the 3-bit opcode field of the instruction while the outputs are represented by the main data-path control signals (except for the ALUCtrl signal). There are 8 x 1-bit control signals and ALUOp which can be 2 or more bits wide depending on the 15 instructions that

are chosen. As it can be seen in Figure 4 the ALUOp line is thicker, meaning that it has more than 1-bit. I identified all the control signals for the 15 instructions, and they are detailed in Table 1.

EX stage

The Execution (EX) stage is where the actual operation specified by the instruction takes place. The Arithmetic Logic Unit (ALU) performs the required computation or logical operation based on the decoded instruction. For instructions that involve memory operations, such as load or store, this stage calculates the effective address where the memory operation will occur. Additionally, if the instruction is a branch, the EX stage evaluates whether the branch condition is met and computes the target address if the branch is taken. The results of these operations are then passed to the next stage for further processing.

The Execute Unit (Ex) consists of an Arithmetic Logic Unit (ALU) which performs arithmetical or logical operations, an ALU Control which in MIPS 16 has 3 bits, a multiplexer, and a Shift Left 2 and adder for branch target address computation.

The output of the EX unit is the ALU Result, which is used to write the results of arithmetical and logical instructions in the Register File or, in the case of lw and sw instructions, as the address for the Data Memory. The ALU also has another output called the Zero Signal, which indicates whether or not the result of the ALU equals zero (if it does, the signal's value is one; otherwise, it is zero). The branch target address computation is also included in the EX unit for the pipeline implementation.

The EX unit receives several inputs, including the next sequential instruction address (PC+1), 16-bit Read Data 1 (RD1), 16-bit Read Data 2 (RD2), 16-bit Extended Immediate (Ext_Imm), a 3-bit function field (func), and a 1-bit shift amount (sa). The control signals for this unit are ALUSrc, which determines whether Read Data 2 or the Extended Immediate is fed into the second port of the ALU, and ALUOp, which provides the ALU operation code from the Main Control Unit. The outputs of the Ex unit include a 16-bit branch target address, a 16-bit ALU result (ALURes), and a 1-bit zero signal. The ALUSrc control signal specifies that when set to 0, Read Data 2 serves as the second input for the ALU, and when set to 1, the Extended Immediate is used instead. The ALUOp signal, defined by the Main Control Unit, specifies the ALU's operation.

The following formula calculates the branch target address for MIPS 16:

Branch Address <- PC + 1 + S_Ext(Imm);

The Zero signal, together with the Branch Control Signal, determines whether to continue with the normal sequential execution of the program (PC + 1) or to jump to the branch target address.

The ALU Control Unit specifies the ALU operations through the ALUCtrl control signal. For I-type instructions, the ALUCtrl encoding is directly determined by the ALUOp signal. In contrast, for R-type instructions, the ALUCtrl value is defined by a fixed ALUOp value combined with the function field.

MEM stage

During the Memory Access (MEM) stage, the processor interacts with memory based on the instruction's requirements. For load instructions, the data located at the address computed in the EX stage is read from memory. Conversely, for store instructions, the data that needs to be written is sent to the memory address computed earlier. This stage ensures that the memory operations are correctly executed, whether it involves reading data from memory or writing data to a specified memory location.

The Memory Unit is composed of the Data Memory component, which functions as RAM with asynchronous read and synchronous write capabilities. The inputs to the Memory Unit are as follows: the clock signal, which manages Data Memory writes; the 16-bit ALURes signal, which provides the address for the Data Memory; the 16-bit RD2 signal, which is the second output from the Register File and is used for store word instructions as the Write Data for Data Memory; and the MemWrite control signal.

The outputs of the Memory Unit include the 16-bit MemData, which represents the Read Data from Data Memory and is used for load word instructions, and the 16-bit ALURes. This ALURes signal not only carries the result of arithmetic-logical operations that need to be stored in the Register File but also serves as an output for the Memory Unit and an input to the Write Back Unit.

In this stage, the only control signal is MemWrite. When MemWrite is set to 0, no data is written to Data Memory. Conversely, when MemWrite is set to 1, the RD2 signal is written into Data Memory at the address specified by the ALURes signal.

WB Stage

The Write Back (WB) stage is the final stage in the instruction cycle, where the results of the instruction are written back into the register file. If the instruction involves a computation or data retrieval (such as from a load instruction), the result obtained from the EX or MEM stages is stored in the destination register specified by the instruction. This stage is essential for updating the register file with the latest results, thereby ensuring that the processor’s state reflects the outcomes of the executed instructions.

The Write Back unit is essentially the final multiplexer in the system, as shown in Figure 4. Additionally, it includes an AND gate used for generating the PCSrc control signal and circuitry for computing the jump address.

The control signal for the Write Back multiplexer is labeled MemtoReg. This signal determines the value that is fed into the Write Data port of the Register File during the ID stage. When MemtoReg is set to 0, the ALURes signal is used as the input to the Write Data port of the Register File. Conversely, when MemtoReg is set to 1, the MemData signal is used for this purpose.

The PCSrc signal is responsible for identifying whether the current instruction is a Branch instruction and whether the contents of the rs and rt registers are equal (e.g., RF[rs] == RF[rt]). It is defined as the logical AND of the Branch signal and the Zero signal, expressed as PCSrc <= Branch AND Zero.

Updating MIPS 16 from Single-Cycle to Pipeline

The primary challenge with the single-cycle MIPS CPU arises from the length of the critical path, especially for the load word instruction. The clock cycle time must be long enough to accommodate the data transmission along this critical path, leading to a slower clock cycle. To address this issue and reduce the clock cycle time, the data path can be partitioned along the critical path using rising edge-triggered registers, such as D flip-flops. As illustrated in Figure 5, these registers are placed between the functional units of the MIPS 32, aligning with the instruction execution phases: IF, ID, EX, MEM, and WB. This approach allows for the simultaneous execution of up to five instructions, with each instruction progressing through one of the five execution phases. The stages of these pipeline execution units are also known as pipeline stages.

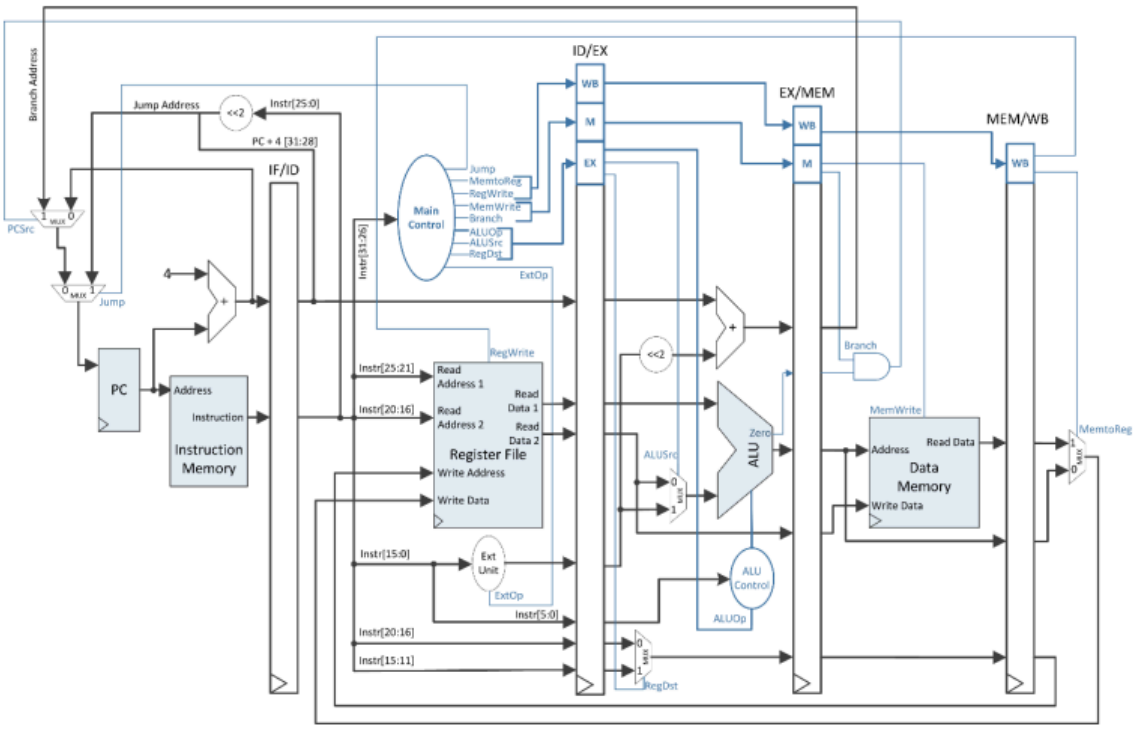

Figure 5. MIPS 32 Pipeline data-path with control

Each intermediate register is named according to its position between pipeline stages. The register between the IF and ID stages is called IF/ID, the one between ID and EX is ID/EX, and so forth. These intermediate registers are responsible for storing intermediate results from instruction execution, which are then passed to the subsequent stage in the next clock cycle. Additionally, the operation of the data path depends on control signals that vary by instruction. Therefore, the intermediate registers, starting with the ID/EX register, also carry the control signals needed for the following stages. These control signals are grouped by the stage to which they belong and are transmitted along with the intermediate results to the stages where they are required.

A key difference between the two data paths is that, in the pipelined version, the multiplexer for selecting the write address for the Register File is positioned in the EX stage, rather than

in the ID stage as in the single-cycle CPU. There are two potential approaches to address this:

1. Retain it in the ID Stage: In this case, the RegDst signal would bypass the ID/EX register and be connected directly from the control unit.
2. Move it to the EX Stage: This would involve modifying the input and output ports of the ID and EX units and transmitting the RegDst signal in line with the updated pipeline data path.

Introducing new pipeline registers needs some minor adjustments in the functional units. For instance, the ID unit will require an additional input port for the Register File write address, which will be sourced from the MEM/WB register. Additionally, the RegWrite control signal will be transmitted through the MEM/WB register and mapped to the input of the ID unit. In describing the behavior of these registers for my MIPS 16, I used concatenation, and the revised signals are detailed in Table 2 below

IF/ID	ID/EX	EX/MEM	MEM/WB
Instruction_IF_ID(16)	RegDst_ID_EX(1)	Branch_EX_MEM(1)	RegWr_MEM_WB(1)
PC_1_IF_ID(16)	Branch_ID_EX(1)	BrNE_EX_MEM(1)	ALUResOut_MEM_WB(16)
	RegWr_ID_EX(1)	RegWr_EX_MEM(1)	MemData_MEM_WB(16)
	BrNE_ID_EX(1)	Zero_EX_MEM(1)	Rd_MEM_WB(3)
	RD1_ID_EX(16)	BranchAddress_EX_MEM(16)	MemtoReg_MEM_WB(1)
	RD2_ID_EX(16)	ALURes_EX_MEM(16)	
	Ext_imm_ID_EX(16)	Rd_EX_MEM(3)	
	Func_ID_EX(3)	RD2_EX_MEM(16)	
	Sa_ID_EX(1)	MemWrite_EX_MEM(1)	
	Rt_ID_EX(3)	MemtoReg_EX_MEM(1)	
	Rd_ID_EX(3)		

	ExtOp_ID_EX(1)		
	AluSrc_ID_EX(1)		
	AluOp_ID_EX(2)		
	MemWrite_ID_EX(1)		
	MemtoReg_ID_EX(1)		

Table 2. MIPS 16 pipeline registers signals

Hazards

Hazards occur when an instruction cannot be executed in the next clock cycle. These hazards can be categorized into three types:

1. Structural Hazards (Resource Dependency): This type arises when two instructions simultaneously attempt to use the same resource for different purposes, leading to resource constraints.
2. Data Hazards (Data Dependency): These occur when an instruction attempts to use data before it is available. For example, an instruction in the ID phase might require operands that are still being processed in other stages of the pipeline.
3. Control Hazards (Control and Condition Dependency): Control hazards arise when the decision and address for a branch are not known until the MEM stage, while the jump address is computed during the ID stage. This includes issues related to pipelining jumps, branches, and other instructions that alter the sequential flow of the program.

For my MIPS 16 pipeline implementation, I implemented the software solution by modifying my initial program such that the data and control hazards are avoided. The primary modification involved inserting NoOp (No Operation) instructions between the instructions where hazards occur. A NoOp instruction does not affect the processor's state; for example, instructions like sll $0, $0, 0 or add $0, $0, $0 are used to fill these gaps without altering any data.

The initial program was this:

1. Initialize R1 with 10: addi $1, $0, 10
2. Initialize R2 with 15: addi $2, $0, 15
3. Store R2 - R1 in R3: sub $3, $2, $1
4. Store R2 + R1 in R4: add $4, $2, $1
5. Store value of R3 at memory address 6 + R1: sw $3, 6($1)
6. Load from address R2 + 1 to R2: lw $2, 1($2)
7. Shift right R1 by 1: srl $1, $1, 1
8. Branch if R1 == R2: beq $1, $2, 1
9. Jump to instruction 4: j 4
10. Store value of R1 at address R2 + 2: sw $1, 2($1)

11. Subtract R2 from R4 and store in R4: sub $4, $4, $2
12. Shift left R1 by 1: sll $1, $1, 1
13. Branch if R1 == R4: beq $1, $4, 1
14. Jump to instruction 11: j 11
15. Store value of R1 at address 5: sw $1, 5($0)

The hazards which I found are the following:

- There is a data hazard on register $1 between instructions 0 and 2.
- There is a structural hazard on register $1 between instructions 0 and 3.
- There is a data hazard on register $2 between instructions 1 and 3, and instructions 1 and 2.
- There is a data hazard on register $3 between instructions 2 and 4.
- There is a data hazard on register $1 between instructions 6 and 7.
- There is a structural hazard on register $1 between instructions 6 and 9.
- There is a control hazard at instruction 7.
- There is a control hazard at instruction 8.
- There is a data hazard on register $4 between instructions 10 and 12.
- There is a data hazard on register $1 between instructions 11 and 12.
- There is a structural hazard on register $1 between instructions 11 and 14.
- There is a control hazard at instruction 13.

The updated program to solve the hazards is the following:

1. Initialize R1 with 10
2. Initialize R2 with 15
3. NoOP
4. NoOp
5. Put in R3 the value R2-R1, which is 5
6. Put in R4 the value R2+R1, which is 25
7. NoOp
8. Store at address 6+R1=16 in memory the value of R3, which is 5
9. Load into memory in R2 the value from address R2+1=16, which is 5 => R2=5
10. Shift R1 to the right by one position => R1=R1/2=10/2=5
11. NoOp
12. NoOp
13. If R1=R2 then jump 2 instructions from the current position
14. NoOp
15. NoOp
16. NoOp
17. Jump to the instruction with index 7, which is the 8th in the code
18. NoOp
19. Store at address R1+2=7 the value of R1, which is 5
20. Put in R4 the value R4-R2, which is 20
21. Shift R1 to the left by one position => R1=R1*2=10
22. NoOp
23. NoOp
24. If R1=R4 then jump 2 instructions from the current position
25. NoOp
26. NoOp

27. NoOp
28. Jump to the instruction with index 20, which is the 21st in the code
29. NoOp
30. Store in memory at address 5 the value of R1, which is 20

Pipelined MIPS 16 VHDL Implementation

I implemented the MIPS 16 in VHDL, following the detailed description from the previous sections. The project has the following structure:

1. test_env.vhd - The top level component which puts all the components together.
2. IF.vhd - The component which contains the behavior of the Instruction Fetch phase.
3. ID.vhd - The component which contains the behavior of the Instruction Decode phase.
4. EX.vhd - The component which contains the behavior of the Execution stage.
5. MEM.vhd - The component which contains the behavior of the Memory Access stage.
6. MainControl.vhd - Generates control signals based on the opcode of the instruction. These signals are RegDst, ExtOp, ALUSrc, Branch, Jump, MemWrite, MemtoReg, RegWrite, BrNE, and ALUOp.
7. MPG.vhd - Generates enable signals for single pulse operations based on the input buttons.
8. SSD.vhd - This interface uses seven LEDs for each digit; each digit is enabled by an anode signal. All the connections (7 common cathode and 4 distinct anode signals) to the SSD interface are active low. The cathode signals control the LEDs of the digit to be displayed, which is selected by the active anode signal. This component has the behavior of it and displays the output on a seven-segment display.
9. testbench.vhd - A simulation file used to verify the functionality of the MIPS processor by using switches in order to control the display on the SSD (multiplexor)
    a. When sw(7:5) = 000, the SSD will show the instruction.
    b. When sw(7:5) = 001, the SSD will show the next sequential PC value (PC + 1).
    c. When sw(7:5) = 010, the SSD will show the RD1 signal.
    d. When sw(7:5) = 011, the SSD will show the RD2 signal.
    e. When sw(7:5) = 100, the SSD will show the Ext_Imm signal.
    f. When sw(7:5) = 101, the SSD will show the ALURes signal.
    g. When sw(7:5) = 110, the SSD will show the MemData signal.
    h. When sw(7:5) = 111, the SSD will show the WD signal.

The project was synthesized and implemented successfully using Vivado, as it can be seen in figure 6 and 7. I generated the .bit file as well, but due to the lack of a FPGA board, I couldn’t test it.

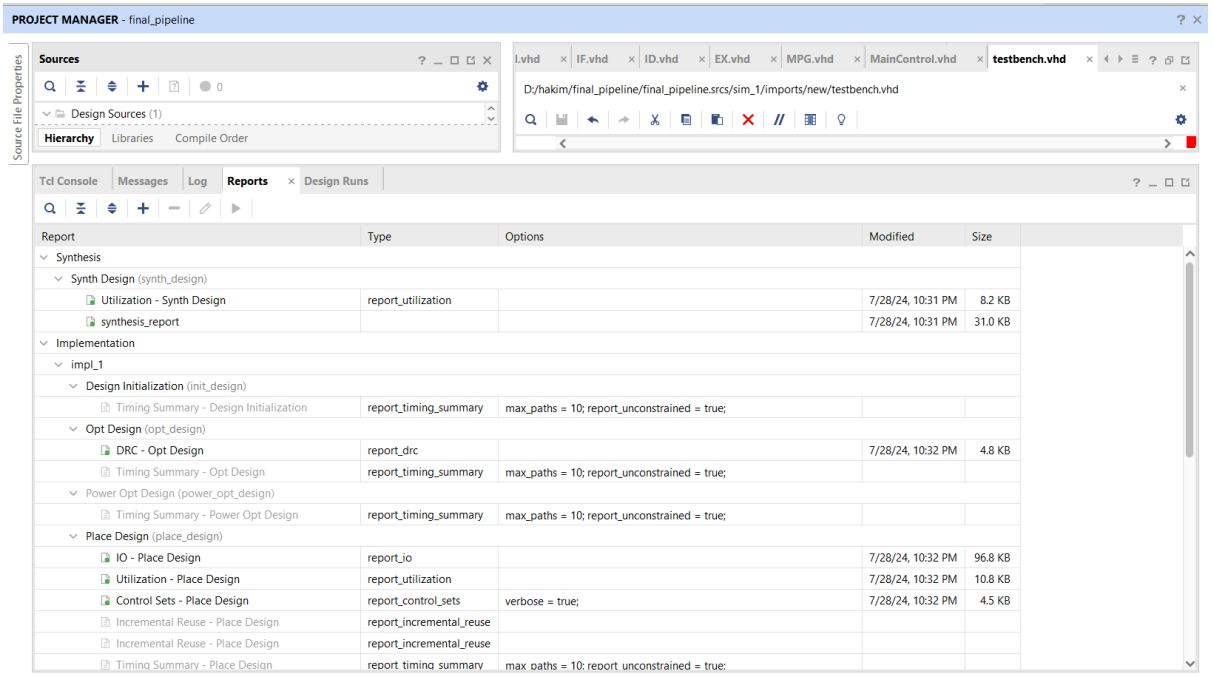

Figure 6. Synthesis and Implementation Vivado Report

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM	DSP
synth_1	constrs_1	synth_design Complete!												271	243	0	0	0
impl_1	constrs_1	route_design Complete!	NA	NA	NA	NA		NA	21.986	0	315 CW			271	250	0	0	0

Figure 7. Synthesis and Implementation Design Runs

Single-cycle vs Pipelined MIPS 16 Analysis

I compared the results of the initial project and the updated pipeline version using 3 features from Vivado reports: timing, utilization and power:

1. *Timing*

As seen in the Timing Summary Report from figure 8, MIPS 16 Single-Cycle has a total of 445 endpoints, while the pipelined version has 845.

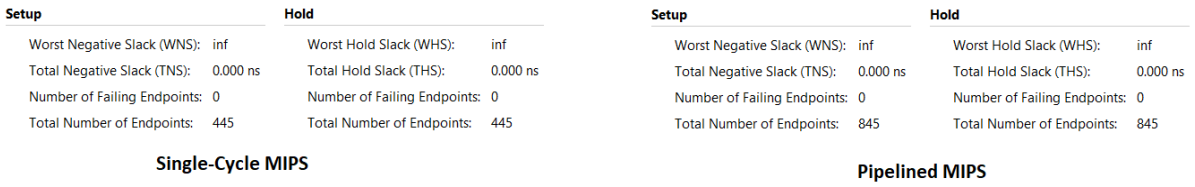

Figure 8. MIPS 16 Single Cycle vs Pipeline Timing Summary

The pipelined MIPS design has almost double the number of endpoints compared to the single-cycle design. This indicates a more complex timing analysis because more paths need to be checked to ensure that each pipeline stage meets its timing requirements. Moreover, each stage in the pipeline adds registers and control logic, contributing to the higher number of endpoints. This overhead is necessary to achieve higher throughput, but it also increases the design complexity and the effort required for timing closure.

2. *Utilization*

Figure 9 shows that the Single Cycle MIPS has utilized 240 LUT, 40 LUTRAM, 54 FF and 33 IO, while pipeline MIPS has used 271 LUT, 41 LUTRAM, 250 FF and 33 IO.

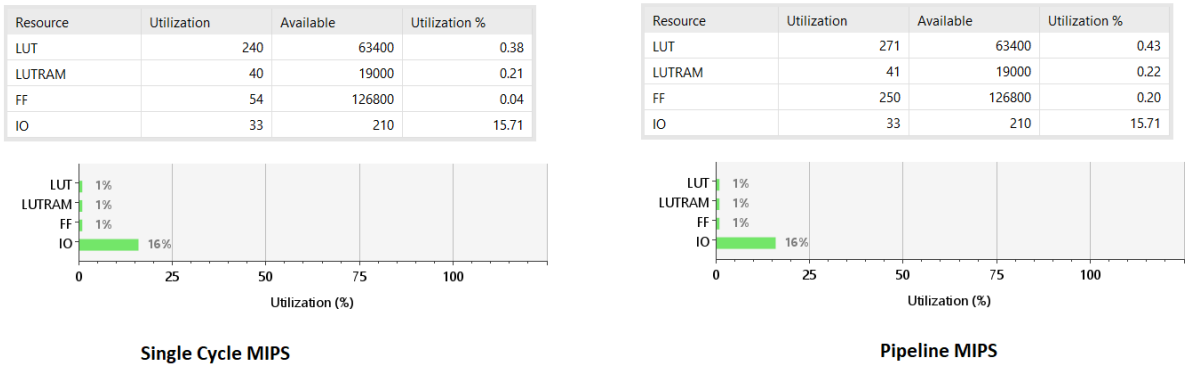

Figure 9. MIPS 16 Single Cycle vs Pipeline Utilization Summary

LUTs (Look-Up Tables)

LUTs are used to implement combinational logic. The number of LUTs used indicates the amount of combinational logic in the design. The Single-Cycle MIPS utilizes 240 LUTs, reflecting the combinational logic required to complete an instruction in one clock cycle. In comparison, the Pipelined MIPS utilizes 271 LUTs, which is slightly higher due to the additional control logic needed for managing the pipeline stages. This slight increase is expected because pipelining introduces complexity in ensuring that data flows correctly between stages without causing hazards.

LUTRAMs (Look-Up Table RAM)

LUTRAMs are LUTs configured as RAM blocks and are used for small memory structures within the design. The similar number of LUTRAMs (40 in the Single-Cycle MIPS versus 41 in the Pipelined MIPS) indicates that both designs use comparable amounts of small RAM structures. These structures are likely used for register files or small buffers, suggesting that the internal memory requirements of both designs are quite similar, despite the differences in their overall architecture.

FFs (Flip-Flops)

Flip-flops are used to implement storage elements such as registers. The Single-Cycle MIPS utilizes 54 FFs, reflecting the registers needed for state holding, including the program counter and general-purpose registers. The Pipelined MIPS, on the other hand, utilizes 250 FFs, which is significantly higher. This substantial increase is due to the additional pipeline registers required to hold intermediate states between pipeline stages. Each stage of the pipeline requires registers to store the outputs of the previous stage and inputs for the next stage, thus dramatically increasing the number of flip-flops needed.

IO (Input/Output Blocks)

IO blocks are used for interfacing with external signals. Both the Single-Cycle and Pipelined MIPS designs use 33 IOs, indicating that the external interfacing requirements are the same for both architectures. This consistency is expected, as both designs likely have the same input and output requirements for communication with memory and peripherals. The

identical number of IO blocks suggests that changes in the internal processing architecture do not affect the external interfacing requirements.

3. Power

Regarding Power, the Single-Cycle MIPS uses 28.27W (97% of total), while Pipeline uses 21.19W (96% of total), as observed in figure 10. For Single Cycle, IF uses 8% of the total, followed by ID with 5%, EX and MEM with 1% and SSD and MPG with <1%. For Pipeline, EX uses 4% of the total, SSD 2% of the total and ID, IF, MEM and MPG 1% of the total.

Utilization	Name	Signals (W)	Data (W)	Clock Enable (W)	Set/Reset (W)	Logic (W)	I/O (W)
28.273 W (97% of total)	test_env						
23.709 W (82% of total)	Leaf Cells (36)						
2.414 W (8% of total)	InstrFetch (IFetch)	1.114	1.111	0.003	<0.001	1.3	<0.001
1.59 W (5% of total)	InstrDecode (ID)	0.987	0.987	<0.001	<0.001	0.603	<0.001
0.245 W (1% of total)	EX1 (EX)	0.183	0.183	<0.001	<0.001	0.062	<0.001
0.161 W (1% of total)	MEM1 (MEM)	0.119	0.119	<0.001	<0.001	0.042	<0.001
0.083 W (<1% of total)	display (SSD)	0.056	0.056	<0.001	<0.001	0.027	<0.001
0.072 W (<1% of total)	monoimpulse (MPG)	0.037	0.037	<0.001	<0.001	0.034	<0.001
<0.001 W (<1% of total)	monoimpulseR (MPG_0)	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001

Single Cycle MIPS

Utilization	Name	Signals (W)	Data (W)	Clock Enable (W)	Set/Reset (W)	Logic (W)	I/O (W)
21.19 W (96% of total)	test_env						
19.015 W (86% of total)	Leaf Cells (247)						
0.904 W (4% of total)	EX1 (EX)	0.398	0.398	<0.001	<0.001	0.506	<0.001
0.461 W (2% of total)	display (SSD)	0.227	0.227	<0.001	<0.001	0.233	<0.001
0.326 W (1% of total)	InstrDecode (ID)	0.168	0.168	<0.001	<0.001	0.158	<0.001
0.326 W (1% of total)	InstrFetch (IFetch)	0.13	0.13	<0.001	<0.001	0.195	<0.001
0.091 W (1% of total)	MEM1 (MEM)	0.046	0.046	<0.001	<0.001	0.045	<0.001
0.067 W (1% of total)	monoimpulse (MPG)	0.034	0.034	<0.001	<0.001	0.033	<0.001
<0.001 W (<1% of total)	monoimpulseR (MPG_0)	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001

Pipeline MIPS

Figure 10. MIPS 16 Single Cycle vs Pipeline Power Summary

The results indicate that the Single-Cycle MIPS is less power-efficient than the Pipelined MIPS, consuming significantly more power despite performing the same overall function. The reduced power consumption in the pipelined design suggests that the distribution of workload across multiple stages can lead to more efficient power usage, even though the pipelined design has a more complex architecture.

In the Single-Cycle MIPS design, the Instruction Fetch (IF) stage consumes 8% of the total power, which is the highest among all stages. This significant power consumption in the IF stage reflects the energy-intensive nature of fetching instructions from memory or cache every clock cycle. The Instruction Decode (ID) stage follows with 5% of the total power, indicating substantial power usage in decoding the instructions to determine the required operations. The Execution (EX) and Memory (MEM) stages each consume 1% of the total power, which suggests that these stages are relatively less power-intensive in this architecture. The SSD and MPG use less than 1% of the total power, showing minimal power requirements for these auxiliary functions.

For the Pipelined MIPS design, the Execution (EX) stage consumes 4% of the total power, highlighting its role as the most power-intensive stage in this architecture. The EX stage's

higher power consumption in the pipelined design compared to the single-cycle design can be attributed to the frequent switching activity and complex operations handled at this stage.

Overall, these metrics indicate that while the pipelined design is more complex and utilizes more resources, it achieves better power efficiency by distributing the power load more evenly across its stages.

Conclusion

By starting from a single-cycle MIPS CPU design, which processes one instruction per clock cycle, and evolving it into a pipelined structure, the project highlights the performance improvements and challenges associated with pipelining in processor design. The MIPS 16 architecture, chosen for its suitability for data display through 8 LEDs and a 4-digit Seven Segment Display, demonstrates the practical considerations in simplifying on-chip debugging and signal display mechanisms.

The detailed exploration of instruction formats, including R-type, I-type, and J-type instructions, sets the foundation for understanding the core functionalities of the MIPS 16 processor. Each instruction type, tailored to fit within the 16-bit architecture, ensures efficient utilization of the reduced instruction width while maintaining compatibility with MIPS32 ISA operations.

The project systematically progresses through each stage of the MIPS processor's execution cycle—Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). By detailing the operations and data flow within each stage, the report provides a comprehensive understanding of the processor's functioning. The inclusion of specific control signals and their implications at each stage further clarifies the processor's control flow.

Transforming the single-cycle CPU into a pipelined CPU addresses the critical path limitations inherent in the single-cycle design. The introduction of intermediate registers between stages allows simultaneous execution of multiple instructions, thereby enhancing the processor's throughput. The strategic placement of these registers and the careful transmission of control signals ensure seamless operation across pipeline stages.

Hazard management is a crucial aspect of pipelined processor design. The report identifies structural, data, and control hazards, and demonstrates a software solution to mitigate these hazards by inserting NoOp (No Operation) instructions. This approach effectively resolves dependencies and ensures correct instruction execution order.

In conclusion, the pipelined MIPS 16 processor project showcases the balance between theoretical principles and practical implementation in CPU design. The transition from single-cycle to pipelined architecture not only improves performance but also offers insights into handling challenges such as hazards and signal management. The detailed VHDL implementation and thorough testing reinforce the robustness of the design, making it a valuable learning experience in computer architecture.

References

- MIPS Architecture for Programmers, Volume I-A: Introduction to the MIPS32 Architecture
- MIPS Architecture for Programmers Volume II-A: The MIPS32 Instruction Set Manual
- XST User Guide, Chapter 6: XST VHDL Language Support