GEORGIA SOUTHERN UNIVERSITY

EENG 5342 Project

Single-Cycle MIPS Processor

Anish Goyal and Killian McCrary

Department of Electrical/Computer Engineering
Georgia Southern University
Statesboro, GA

December 3, 2025

Outline

Introduction

An overview of the task at hand and what we planned to accomplish.

Motivation

Why we chose this project

- Gain hands-on experience with processor architecture
- Understand the datapath and control unit interaction
- Apply VHDL skills to a complex, multi-component system
- Bridge the gap between theoretical CPU design and practical implementation

Motivation

Why we chose this project

- Gain hands-on experience with processor architecture
- Understand the datapath and control unit interaction
- Apply VHDL skills to a complex, multi-component system
- Bridge the gap between theoretical CPU design and practical implementation

Motivation

Why we chose this project

- Gain hands-on experience with processor architecture
- Understand the datapath and control unit interaction
- Apply VHDL skills to a complex, multi-component system
- Bridge the gap between theoretical CPU design and practical implementation

Motivation

Why we chose this project

- Gain hands-on experience with processor architecture
- Understand the datapath and control unit interaction
- Apply VHDL skills to a complex, multi-component system
- Bridge the gap between theoretical CPU design and practical implementation

Abstract

Overview of the project and its goals

- Implemented a single-cycle MIPS processor in VHDL
- Supports R-type, I-type, and memory instructions
- Key components include:
  - Program Counter (PC)
  - Instruction Memory
  - Register File (32 registers)
  - ALU with controller
  - Data Memory
  - Main Control Unit
- Validated via functional simulation in Questa/ModelSim

Abstract

Overview of the project and its goals

- Implemented a single-cycle MIPS processor in VHDL
- Supports R-type, I-type, and memory instructions
- Key components include:
  - Program Counter (PC)
  - Instruction Memory
  - Register File (32 registers)
  - ALU with controller
  - Data Memory
  - Main Control Unit
- Validated via functional simulation in Questa/ModelSim

Abstract

Overview of the project and its goals

- Implemented a single-cycle MIPS processor in VHDL
- Supports R-type, I-type, and memory instructions
- Key components include:
  - Program Counter (PC)
  - Instruction Memory
  - Register File (32 registers)
  - ALU with controller
  - Data Memory
  - Main Control Unit
- Validated via functional simulation in Questa/ModelSim

Abstract

Overview of the project and its goals

- Implemented a single-cycle MIPS processor in VHDL
- Supports R-type, I-type, and memory instructions
- Key components include:
  - Program Counter (PC)
  - Instruction Memory
  - Register File (32 registers)
  - ALU with controller
  - Data Memory
  - Main Control Unit
- Validated via functional simulation in Questa/ModelSim

Abstract

Overview of the project and its goals

- Implemented a single-cycle MIPS processor in VHDL
- Supports R-type, I-type, and memory instructions
- Key components include:
  - Program Counter (PC)
  - Instruction Memory
  - Register File (32 registers)
  - ALU with controller
  - Data Memory
  - Main Control Unit
- Validated via functional simulation in Questa/ModelSim

Abstract

Overview of the project and its goals

- Implemented a single-cycle MIPS processor in VHDL
- Supports R-type, I-type, and memory instructions
- Key components include:
  - Program Counter (PC)
  - Instruction Memory
  - Register File (32 registers)
  - ALU with controller
  - Data Memory
  - Main Control Unit
- Validated via functional simulation in Questa/ModelSim

Abstract

Overview of the project and its goals

- Implemented a single-cycle MIPS processor in VHDL
- Supports R-type, I-type, and memory instructions
- Key components include:
  - Program Counter (PC)
  - Instruction Memory
  - Register File (32 registers)
  - ALU with controller
  - Data Memory
  - Main Control Unit
- Validated via functional simulation in Questa/ModelSim

Abstract

Overview of the project and its goals

- Implemented a single-cycle MIPS processor in VHDL
- Supports R-type, I-type, and memory instructions
- Key components include:
  - Program Counter (PC)
  - Instruction Memory
  - Register File (32 registers)
  - ALU with controller
  - Data Memory
  - Main Control Unit
- Validated via functional simulation in Questa/ModelSim

Abstract

Overview of the project and its goals

- Implemented a single-cycle MIPS processor in VHDL
- Supports R-type, I-type, and memory instructions
- Key components include:
  - Program Counter (PC)
  - Instruction Memory
  - Register File (32 registers)
  - ALU with controller
  - Data Memory
  - Main Control Unit
- Validated via functional simulation in Questa/ModelSim

Abstract

Overview of the project and its goals

- Implemented a single-cycle MIPS processor in VHDL
- Supports R-type, I-type, and memory instructions
- Key components include:
  - Program Counter (PC)
  - Instruction Memory
  - Register File (32 registers)
  - ALU with controller
  - Data Memory
  - Main Control Unit
- Validated via functional simulation in Questa/ModelSim

What was planned

Our objectives

- Create a fully functional single-cycle MIPS datapath
- Support the following instruction types:
  - R-type: `add`, `sub`, `and`, `or`, `slt`
  - I-type: `lw`, `sw`
- Integrate all components via schematic capture in Quartus
- Verify correctness through waveform simulation

What was planned

Our objectives

- Create a fully functional single-cycle MIPS datapath
- Support the following instruction types:
  - R-type: `add`, `sub`, `and`, `or`, `slt`
  - I-type: `lw`, `sw`
- Integrate all components via schematic capture in Quartus
- Verify correctness through waveform simulation

What was planned

Our objectives

- Create a fully functional single-cycle MIPS datapath
- Support the following instruction types:
  - R-type: `add`, `sub`, `and`, `or`, `slt`
  - I-type: `lw`, `sw`
- Integrate all components via schematic capture in Quartus
- Verify correctness through waveform simulation

What was planned

Our objectives

- Create a fully functional single-cycle MIPS datapath
- Support the following instruction types:
  - R-type: `add`, `sub`, `and`, `or`, `slt`
  - I-type: `lw`, `sw`
- Integrate all components via schematic capture in Quartus
- Verify correctness through waveform simulation

What was planned

Our objectives

- Create a fully functional single-cycle MIPS datapath
- Support the following instruction types:
  - R-type: `add`, `sub`, `and`, `or`, `slt`
  - I-type: `lw`, `sw`
- Integrate all components via schematic capture in Quartus
- Verify correctness through waveform simulation

What was planned

Our objectives

- Create a fully functional single-cycle MIPS datapath
- Support the following instruction types:
  - R-type: `add`, `sub`, `and`, `or`, `slt`
  - I-type: `lw`, `sw`
- Integrate all components via schematic capture in Quartus
- Verify correctness through waveform simulation

Methodology

Detailed explanation of the approach and techniques used.

Design

Overview of the design process

- Based on the classic MIPS single-cycle architecture
- Each instruction executes in one clock cycle
- Datapath connects:
  - PC → Instruction Memory → Register File
  - ALU performs computation
  - Data Memory for load/store operations
- Control signals generated by Main Control and ALU Controller

Design

Overview of the design process

- Based on the classic MIPS single-cycle architecture
- Each instruction executes in one clock cycle
- Datapath connects:
  - PC → Instruction Memory → Register File
  - ALU performs computation
  - Data Memory for load/store operations
- Control signals generated by Main Control and ALU Controller

Design

Overview of the design process

- Based on the classic MIPS single-cycle architecture
- Each instruction executes in one clock cycle
- Datapath connects:
  - PC $\rightarrow$ Instruction Memory $\rightarrow$ Register File
  - ALU performs computation
  - Data Memory for load/store operations
- Control signals generated by Main Control and ALU Controller

Design

Overview of the design process

- Based on the classic MIPS single-cycle architecture
- Each instruction executes in one clock cycle
- Datapath connects:
  - PC $\rightarrow$ Instruction Memory $\rightarrow$ Register File
  - ALU performs computation
  - Data Memory for load/store operations
- Control signals generated by Main Control and ALU Controller

Design

Overview of the design process

- Based on the classic MIPS single-cycle architecture
- Each instruction executes in one clock cycle
- Datapath connects:
  - PC $\rightarrow$ Instruction Memory $\rightarrow$ Register File
  - ALU performs computation
  - Data Memory for load/store operations
- Control signals generated by Main Control and ALU Controller

Design

Overview of the design process

- Based on the classic MIPS single-cycle architecture
- Each instruction executes in one clock cycle
- Datapath connects:
  - PC $\rightarrow$ Instruction Memory $\rightarrow$ Register File
  - ALU performs computation
  - Data Memory for load/store operations
- Control signals generated by Main Control and ALU Controller

Design

Overview of the design process

- Based on the classic MIPS single-cycle architecture
- Each instruction executes in one clock cycle
- Datapath connects:
  - PC $\rightarrow$ Instruction Memory $\rightarrow$ Register File
  - ALU performs computation
  - Data Memory for load/store operations
- Control signals generated by Main Control and ALU Controller

Implementation

Details on how the processor was built and tested.

Block Diagram

Schematic overview of the datapath

- Full datapath designed using Quartus Block Diagram File (.bdf)
- Components instantiated as VHDL symbols
- Wiring completed manually in schematic editor
- Video walkthrough of the block schematic:

https://youtu.be/19uMkeUoLpE

Block Diagram

Schematic overview of the datapath

- Full datapath designed using Quartus Block Diagram File (.bdf)
- Components instantiated as VHDL symbols
- Wiring completed manually in schematic editor
- Video walkthrough of the block schematic:

https://youtu.be/19uMkeUoLpE

Block Diagram

Schematic overview of the datapath

- Full datapath designed using Quartus Block Diagram File (.bdf)
- Components instantiated as VHDL symbols
- Wiring completed manually in schematic editor
- Video walkthrough of the block schematic:

https://youtu.be/19uMkeUoLpE

Block Diagram

Schematic overview of the datapath

- Full datapath designed using Quartus Block Diagram File (.bdf)
- Components instantiated as VHDL symbols
- Wiring completed manually in schematic editor
- Video walkthrough of the block schematic:

https://youtu.be/19uMkeUoLpE

Opcode Implementation

Control signal generation based on instruction opcode

- Main Control decodes 6-bit opcode
- Generates control signals:
- ALU Controller uses `ALUOp` and `funct` field

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

Figure: Opcode table for control signal generation

Opcode Implementation

Control signal generation based on instruction opcode

- Main Control decodes 6-bit opcode
- Generates control signals:
  - `RegDst`, `ALUSrc`
  - `MemtoReg`, `RegWrite`
  - `MemRead`, `MemWrite`
  - `Branch`, `ALUOp`
- ALU Controller uses `ALUOp` and `funct` field

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

Figure: Opcode table for control signal generation

Opcode Implementation

Control signal generation based on instruction opcode

- Main Control decodes 6-bit opcode
- Generates control signals:
  - `RegDst`, `ALUSrc`
  - `MemtoReg`, `RegWrite`
  - `MemRead`, `MemWrite`
  - `Branch`, `ALUOp`
- ALU Controller uses `ALUOp` and `funct` field

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

Figure: Opcode table for control signal generation

Opcode Implementation

Control signal generation based on instruction opcode

- Main Control decodes 6-bit opcode
- Generates control signals:
  - `RegDst`, `ALUSrc`
  - `MemtoReg`, `RegWrite`
  - `MemRead`, `MemWrite`
  - `Branch`, `ALUOp`
- ALU Controller uses `ALUOp` and `funct` field

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

Figure: Opcode table for control signal generation

Opcode Implementation

Control signal generation based on instruction opcode

- Main Control decodes 6-bit opcode
- Generates control signals:
  - `RegDst`, `ALUSrc`
  - `MemtoReg`, `RegWrite`
  - `MemRead`, `MemWrite`
  - `Branch`, `ALUOp`
- ALU Controller uses `ALUOp` and `funct` field

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

Figure: Opcode table for control signal generation

Opcode Implementation

Control signal generation based on instruction opcode

- Main Control decodes 6-bit opcode
- Generates control signals:
  - `RegDst`, `ALUSrc`
  - `MemtoReg`, `RegWrite`
  - `MemRead`, `MemWrite`
  - `Branch`, `ALUOp`
- ALU Controller uses `ALUOp` and `funct` field

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

Figure: Opcode table for control signal generation

Opcode Implementation

Control signal generation based on instruction opcode

- Main Control decodes 6-bit opcode
- Generates control signals:
  - `RegDst`, `ALUSrc`
  - `MemtoReg`, `RegWrite`
  - `MemRead`, `MemWrite`
  - `Branch`, `ALUOp`
- ALU Controller uses `ALUOp` and `funct` field

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

Figure: Opcode table for control signal generation

Opcode Implementation

Control signal generation based on instruction opcode

- Main Control decodes 6-bit opcode
- Generates control signals:
  - `RegDst`, `ALUSrc`
  - `MemtoReg`, `RegWrite`
  - `MemRead`, `MemWrite`
  - `Branch`, `ALUOp`
- ALU Controller uses `ALUOp` and `funct` field

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

Figure: Opcode table for control signal generation

Opcode Implementation

Control signal generation based on instruction opcode

- Main Control decodes 6-bit opcode
- Generates control signals:
  - `RegDst`, `ALUSrc`
  - `MemtoReg`, `RegWrite`
  - `MemRead`, `MemWrite`
  - `Branch`, `ALUOp`
- ALU Controller uses `ALUOp` and `funct` field

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

Figure: Opcode table for control signal generation

Opcode Implementation

Control signal generation based on instruction opcode

- Main Control decodes 6-bit opcode
- Generates control signals:
  - `RegDst`, `ALUSrc`
  - `MemtoReg`, `RegWrite`
  - `MemRead`, `MemWrite`
  - `Branch`, `ALUOp`
- ALU Controller uses `ALUOp` and `funct` field

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

Figure: Opcode table for control signal generation

Key Components

Two important components of the datapath

ALU Controller

- Receives `ALUOp` from Main Control
- Decodes `funct` field for R-type
- Outputs 4-bit ALU control signal
- Supports: ADD, SUB, AND, OR, SLT

Register File

- 32 general-purpose registers
- Dual read ports, single write port
- Synchronous write on clock edge
- Register $0 hardwired to zero

Key Components

Two important components of the datapath

ALU Controller

- Receives `ALUOp` from Main Control
- Decodes `funct` field for R-type
- Outputs 4-bit ALU control signal
- Supports: ADD, SUB, AND, OR, SLT

Register File

- 32 general-purpose registers
- Dual read ports, single write port
- Synchronous write on clock edge
- Register $0 hardwired to zero

Key Components

Two important components of the datapath

ALU Controller

- Receives `ALUOp` from Main Control
- Decodes `funct` field for R-type
- Outputs 4-bit ALU control signal
- Supports: ADD, SUB, AND, OR, SLT

Register File

- 32 general-purpose registers
- Dual read ports, single write port
- Synchronous write on clock edge
- Register $0 hardwired to zero

Key Components

Two important components of the datapath

ALU Controller

- Receives `ALUOp` from Main Control
- Decodes `funct` field for R-type
- Outputs 4-bit ALU control signal
- Supports: ADD, SUB, AND, OR, SLT

Register File

- 32 general-purpose registers
- Dual read ports, single write port
- Synchronous write on clock edge
- Register $0 hardwired to zero

Key Components

Two important components of the datapath

ALU Controller

- Receives `ALUOp` from Main Control
- Decodes `funct` field for R-type
- Outputs 4-bit ALU control signal
- Supports: ADD, SUB, AND, OR, SLT

Register File

- 32 general-purpose registers
- Dual read ports, single write port
- Synchronous write on clock edge
- Register $0 hardwired to zero

Key Components

Two important components of the datapath

ALU Controller

- Receives `ALUOp` from Main Control
- Decodes `funct` field for R-type
- Outputs 4-bit ALU control signal
- Supports: ADD, SUB, AND, OR, SLT

Register File

- 32 general-purpose registers
- Dual read ports, single write port
- Synchronous write on clock edge
- Register $0 hardwired to zero

Key Components

Two important components of the datapath

ALU Controller

- Receives `ALUOp` from Main Control
- Decodes `funct` field for R-type
- Outputs 4-bit ALU control signal
- Supports: ADD, SUB, AND, OR, SLT

Register File

- 32 general-purpose registers
- Dual read ports, single write port
- Synchronous write on clock edge
- Register $0 hardwired to zero

Key Components

Two important components of the datapath

ALU Controller

- Receives `ALUOp` from Main Control
- Decodes `funct` field for R-type
- Outputs 4-bit ALU control signal
- Supports: ADD, SUB, AND, OR, SLT

Register File

- 32 general-purpose registers
- Dual read ports, single write port
- Synchronous write on clock edge
- Register $0 hardwired to zero

Key Components

Two important components of the datapath

ALU Controller

- Receives `ALUOp` from Main Control
- Decodes `funct` field for R-type
- Outputs 4-bit ALU control signal
- Supports: ADD, SUB, AND, OR, SLT

Register File

- 32 general-purpose registers
- Dual read ports, single write port
- Synchronous write on clock edge
- Register $0 hardwired to zero

Key Components

Two important components of the datapath

ALU Controller

- Receives `ALUOp` from Main Control
- Decodes `funct` field for R-type
- Outputs 4-bit ALU control signal
- Supports: ADD, SUB, AND, OR, SLT

Register File

- 32 general-purpose registers
- Dual read ports, single write port
- Synchronous write on clock edge
- Register $0 hardwired to zero

Key Components

Two important components of the datapath

ALU Controller

- Receives `ALUOp` from Main Control
- Decodes `funct` field for R-type
- Outputs 4-bit ALU control signal
- Supports: ADD, SUB, AND, OR, SLT

Register File

- 32 general-purpose registers
- Dual read ports, single write port
- Synchronous write on clock edge
- Register $0 hardwired to zero

Results

Summary of simulation results.

Sample Waveform

Functional simulation output

Initial Conditions:

- Registers `$0` – `$31` initialized to values `0` – `31`

Test Instructions:

```
x"012a4020"  -- 0x012A4020 -> add $8, $9, $10
x"01285022"  -- 0x01285022 -> sub $10, $9, $8
x"8e510000"  -- 0x8E510000 -> lw $17, 0($18)
x"ae290004"  -- 0xAE290004 -> sw $9, 4($17)
```

Sample Waveform

Simulation results

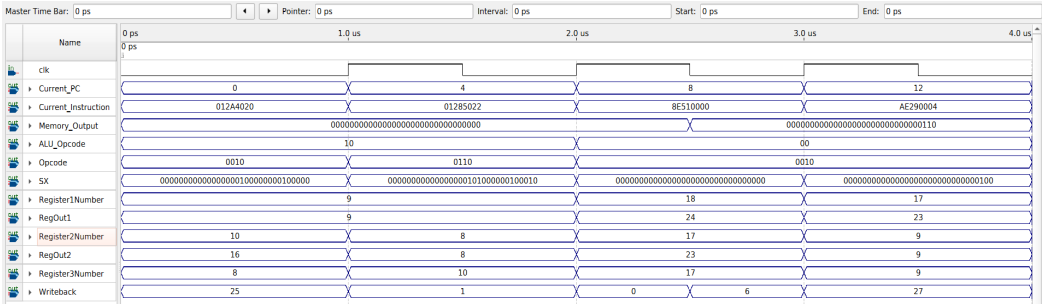

Figure: Waveform output from Questa/ModelSim functional simulation

Challenges and Improvements

Challenges encountered and potential future work.

Problems During Design

Issues encountered during implementation

- **Bus naming convention** – inconsistent naming caused wiring issues
- **Quartus schematic editor** – unreliable click registration and net changes
- **Input/output pin naming** – required careful matching
- **Constant value generation** – needed explicit constant blocks
- **Functional simulation** – required `voptargs="+acc"` on personal laptop

Problems During Design

Issues encountered during implementation

- **Bus naming convention** – inconsistent naming caused wiring issues
- **Quartus schematic editor** – unreliable click registration and net changes
- **Input/output pin naming** – required careful matching
- **Constant value generation** – needed explicit constant blocks
- **Functional simulation** – required `voptargs="+acc"` on personal laptop

Problems During Design

Issues encountered during implementation

- **Bus naming convention** – inconsistent naming caused wiring issues
- **Quartus schematic editor** – unreliable click registration and net changes
- **Input/output pin naming** – required careful matching
- **Constant value generation** – needed explicit constant blocks
- **Functional simulation** – required `voptargs="+acc"` on personal laptop

Problems During Design

Issues encountered during implementation

- **Bus naming convention** – inconsistent naming caused wiring issues
- **Quartus schematic editor** – unreliable click registration and net changes
- **Input/output pin naming** – required careful matching
- **Constant value generation** – needed explicit constant blocks
- **Functional simulation** – required `voptargs="+acc"` on personal laptop

Problems During Design

Issues encountered during implementation

- **Bus naming convention** – inconsistent naming caused wiring issues
- **Quartus schematic editor** – unreliable click registration and net changes
- **Input/output pin naming** – required careful matching
- **Constant value generation** – needed explicit constant blocks
- **Functional simulation** – required `voptargs="+acc"` on personal laptop

Problems During Design

Additional issues

- **XOR overriding SUB** – class code had XOR opcode conflicting with SUB; required control value changes
- **Questa license server** – needed to set `SALT_LICENSE_SERVER` environment variable
- **MemToReg mux polarity** – flipped polarity inherent in reference diagram
- **5-bit 2-to-1 mux** – needed to create a special 5-bit mux before register file

Problems During Design

Additional issues

- **XOR overriding SUB** – class code had XOR opcode conflicting with SUB; required control value changes
- **Questa license server** – needed to set `SALT_LICENSE_SERVER` environment variable
- **MemToReg mux polarity** – flipped polarity inherent in reference diagram
- **5-bit 2-to-1 mux** – needed to create a special 5-bit mux before register file

Problems During Design

Additional issues

- **XOR overriding SUB** – class code had XOR opcode conflicting with SUB; required control value changes
- **Questa license server** – needed to set `SALT_LICENSE_SERVER` environment variable
- **MemToReg mux polarity** – flipped polarity inherent in reference diagram
- **5-bit 2-to-1 mux** – needed to create a special 5-bit mux before register file

Problems During Design

Additional issues

- **XOR overriding SUB** – class code had XOR opcode conflicting with SUB; required control value changes
- **Questa license server** – needed to set `SALT_LICENSE_SERVER` environment variable
- **MemToReg mux polarity** – flipped polarity inherent in reference diagram
- **5-bit 2-to-1 mux** – needed to create a special 5-bit mux before register file

Potential Improvements

Future work and extensions

- **Half-period asymmetry** – add half cycle of zero before first rising edge
- **Pipelining** – implement 5-stage pipeline with data hazard detection
- **Jump support** – add unconditional jump (`j`) instruction
- **Real-time output** – display register/memory values to console
- **Live instruction input** – avoid recompilation for instruction changes

Potential Improvements

Future work and extensions

- **Half-period asymmetry** – add half cycle of zero before first rising edge
- **Pipelining** – implement 5-stage pipeline with data hazard detection
- **Jump support** – add unconditional jump (`j`) instruction
- **Real-time output** – display register/memory values to console
- **Live instruction input** – avoid recompilation for instruction changes

Potential Improvements

Future work and extensions

- **Half-period asymmetry** – add half cycle of zero before first rising edge
- **Pipelining** – implement 5-stage pipeline with data hazard detection
- **Jump support** – add unconditional jump (`j`) instruction
- **Real-time output** – display register/memory values to console
- **Live instruction input** – avoid recompilation for instruction changes

Potential Improvements

Future work and extensions

- **Half-period asymmetry** – add half cycle of zero before first rising edge
- **Pipelining** – implement 5-stage pipeline with data hazard detection
- **Jump support** – add unconditional jump (`j`) instruction
- **Real-time output** – display register/memory values to console
- **Live instruction input** – avoid recompilation for instruction changes

Potential Improvements

Future work and extensions

- **Half-period asymmetry** – add half cycle of zero before first rising edge
- **Pipelining** – implement 5-stage pipeline with data hazard detection
- **Jump support** – add unconditional jump (`j`) instruction
- **Real-time output** – display register/memory values to console
- **Live instruction input** – avoid recompilation for instruction changes

Potential Improvements

Additional enhancements

- **Pipeline registers** – add registers between stages
- **Additional operations** – multiplication, division, shifting
- **Data forwarding** – bypass paths for hazard mitigation
- **FPGA deployment** – synthesize and run on actual hardware

Potential Improvements

Additional enhancements

- **Pipeline registers** – add registers between stages
- **Additional operations** – multiplication, division, shifting
- **Data forwarding** – bypass paths for hazard mitigation
- **FPGA deployment** – synthesize and run on actual hardware

Potential Improvements

Additional enhancements

- **Pipeline registers** – add registers between stages
- **Additional operations** – multiplication, division, shifting
- **Data forwarding** – bypass paths for hazard mitigation
- **FPGA deployment** – synthesize and run on actual hardware

Potential Improvements

Additional enhancements

- **Pipeline registers** – add registers between stages
- **Additional operations** – multiplication, division, shifting
- **Data forwarding** – bypass paths for hazard mitigation
- **FPGA deployment** – synthesize and run on actual hardware

Conclusion

Summary of results

- Successfully implemented a single-cycle MIPS processor
- Supports R-type (`add`, `sub`) and I-type (`lw`, `sw`) instructions
- Verified functionality through waveform simulation
- Gained practical experience with datapath design and control logic
- Identified areas for future improvement and extension

Conclusion

Summary of results

- Successfully implemented a single-cycle MIPS processor
- Supports R-type (`add`, `sub`) and I-type (`lw`, `sw`) instructions
- Verified functionality through waveform simulation
- Gained practical experience with datapath design and control logic
- Identified areas for future improvement and extension

Conclusion

Summary of results

- Successfully implemented a single-cycle MIPS processor
- Supports R-type (`add`, `sub`) and I-type (`lw`, `sw`) instructions
- Verified functionality through waveform simulation
- Gained practical experience with datapath design and control logic
- Identified areas for future improvement and extension

Conclusion

Summary of results

- Successfully implemented a single-cycle MIPS processor
- Supports R-type (`add`, `sub`) and I-type (`lw`, `sw`) instructions
- Verified functionality through waveform simulation
- Gained practical experience with datapath design and control logic
- Identified areas for future improvement and extension

Conclusion

Summary of results

- Successfully implemented a single-cycle MIPS processor
- Supports R-type (`add`, `sub`) and I-type (`lw`, `sw`) instructions
- Verified functionality through waveform simulation
- Gained practical experience with datapath design and control logic
- Identified areas for future improvement and extension

References

[1] M. Ahad, EENG 5342 lecture notes. Aug. 2025.

[2] D. A. Patterson and J. L. Hennessy, *Computer organization and design: The hardware/software interface (MIPS edition)*, 6th Edition. Morgan Kaufmann, 2020.

GEORGIA SOUTHERN UNIVERSITY

EENG 5342 Project

Single-Cycle MIPS Processor

Thank you for your attention!

Anish Goyal and Killian McCrary

Department of Electrical/Computer Engineering
Georgia Southern University
Statesboro, GA

December 3, 2025