|  |
| --- |
| 台科大 |
| PA2 |
| 別扣分了… |

|  |
| --- |
| 周 柏宇  2024/4/26 |

內容

[Part1 5](#_Toc166105081)

[R\_FormatCPU 5](#_Toc166105082)

[Description 6](#_Toc166105083)

[Code Explanation 6](#_Toc166105084)

[IO 7](#_Toc166105085)

[Wires 7](#_Toc166105086)

[Components 7](#_Toc166105087)

[Process 9](#_Toc166105088)

[Adder 9](#_Toc166105089)

[Description 9](#_Toc166105090)

[Code Explanation 9](#_Toc166105091)

[IO 10](#_Toc166105092)

[Wires 10](#_Toc166105093)

[Components 10](#_Toc166105094)

[Process 10](#_Toc166105095)

[IM 11](#_Toc166105096)

[Description 11](#_Toc166105097)

[Code Explanation 11](#_Toc166105098)

[IO 12](#_Toc166105099)

[Wires 12](#_Toc166105100)

[Components 12](#_Toc166105101)

[Process 12](#_Toc166105102)

[RF 13](#_Toc166105103)

[Description 13](#_Toc166105104)

[Code Explanation 13](#_Toc166105105)

[IO 14](#_Toc166105106)

[Wires 14](#_Toc166105107)

[Components 14](#_Toc166105108)

[Process 15](#_Toc166105109)

[Control 15](#_Toc166105110)

[Description 15](#_Toc166105111)

[Code Explanation 15](#_Toc166105112)

[IO 16](#_Toc166105113)

[Wires 16](#_Toc166105114)

[Components 16](#_Toc166105115)

[Process 16](#_Toc166105116)

[ALU\_Control 17](#_Toc166105117)

[Description 17](#_Toc166105118)

[Code Explanation 17](#_Toc166105119)

[IO 18](#_Toc166105120)

[Wires 18](#_Toc166105121)

[Components 18](#_Toc166105122)

[Process 18](#_Toc166105123)

[ALU 19](#_Toc166105124)

[Description 19](#_Toc166105125)

[Code Explanation 19](#_Toc166105126)

[IO 20](#_Toc166105127)

[Wires 20](#_Toc166105128)

[Components 20](#_Toc166105129)

[Process 21](#_Toc166105130)

[Part2 22](#_Toc166105131)

[I\_FormatCPU 22](#_Toc166105132)

[Description 23](#_Toc166105133)

[Code Explanation 24](#_Toc166105134)

[IO 24](#_Toc166105135)

[Wires 24](#_Toc166105136)

[Components 25](#_Toc166105137)

[Process 25](#_Toc166105138)

[Adder 26](#_Toc166105139)

[IM 26](#_Toc166105140)

[RegMUX 26](#_Toc166105141)

[Description 26](#_Toc166105142)

[Code Explanation 27](#_Toc166105143)

[IO 27](#_Toc166105144)

[Wires 27](#_Toc166105145)

[Components 27](#_Toc166105146)

[Process 28](#_Toc166105147)

[RF 28](#_Toc166105148)

[Control 28](#_Toc166105149)

[Description 29](#_Toc166105150)

[Code Explanation 29](#_Toc166105151)

[IO 30](#_Toc166105152)

[Wires 30](#_Toc166105153)

[Components 30](#_Toc166105154)

[Process 31](#_Toc166105155)

[GPRMUX 31](#_Toc166105156)

[Description 31](#_Toc166105157)

[Code Explanation 31](#_Toc166105158)

[IO 32](#_Toc166105159)

[Wires 32](#_Toc166105160)

[Components 32](#_Toc166105161)

[Process 32](#_Toc166105162)

[ALU\_Control 33](#_Toc166105163)

[Description 33](#_Toc166105164)

[Code Explanation 33](#_Toc166105165)

[IO 34](#_Toc166105166)

[Wires 34](#_Toc166105167)

[Components 34](#_Toc166105168)

[Process 34](#_Toc166105169)

[ALU 35](#_Toc166105170)

[Description 35](#_Toc166105171)

[Code Explanation 35](#_Toc166105172)

[IO 36](#_Toc166105173)

[Wires 36](#_Toc166105174)

[Components 36](#_Toc166105175)

[Process 37](#_Toc166105176)

[DM 37](#_Toc166105177)

[Description 37](#_Toc166105178)

[Code Explanation 38](#_Toc166105179)

[IO 39](#_Toc166105180)

[Wires 39](#_Toc166105181)

[Components 39](#_Toc166105182)

[Process 39](#_Toc166105183)

[Part3 40](#_Toc166105184)

[SimpleCPU 40](#_Toc166105185)

[Description 42](#_Toc166105186)

[Code Explanation 42](#_Toc166105187)

[IO 43](#_Toc166105188)

[Wires 43](#_Toc166105189)

[Components 44](#_Toc166105190)

[Process 45](#_Toc166105191)

[Adder 45](#_Toc166105192)

[GPRMUX 45](#_Toc166105193)

[IM 45](#_Toc166105194)

[RegMUX 46](#_Toc166105195)

[RF 46](#_Toc166105196)

[Control 46](#_Toc166105197)

[Description 48](#_Toc166105198)

[Code Explanation 48](#_Toc166105199)

[IO 48](#_Toc166105200)

[Wires 49](#_Toc166105201)

[Components 49](#_Toc166105202)

[Process 49](#_Toc166105203)

[ALU\_Control 49](#_Toc166105204)

[ALU 50](#_Toc166105205)

[Description 50](#_Toc166105206)

[Code Explanation 50](#_Toc166105207)

[IO 51](#_Toc166105208)

[Wires 52](#_Toc166105209)

[Components 52](#_Toc166105210)

[Process 52](#_Toc166105211)

[DM 52](#_Toc166105212)

[Execution Results 53](#_Toc166105213)

[Helper programs 53](#_Toc166105214)

[Translator.py 53](#_Toc166105215)

[Convertor.py 55](#_Toc166105216)

[Part1 58](#_Toc166105217)

[Part2 69](#_Toc166105218)

[Part3 80](#_Toc166105219)

[Custom Test Program 87](#_Toc166105220)

[Part1 87](#_Toc166105221)

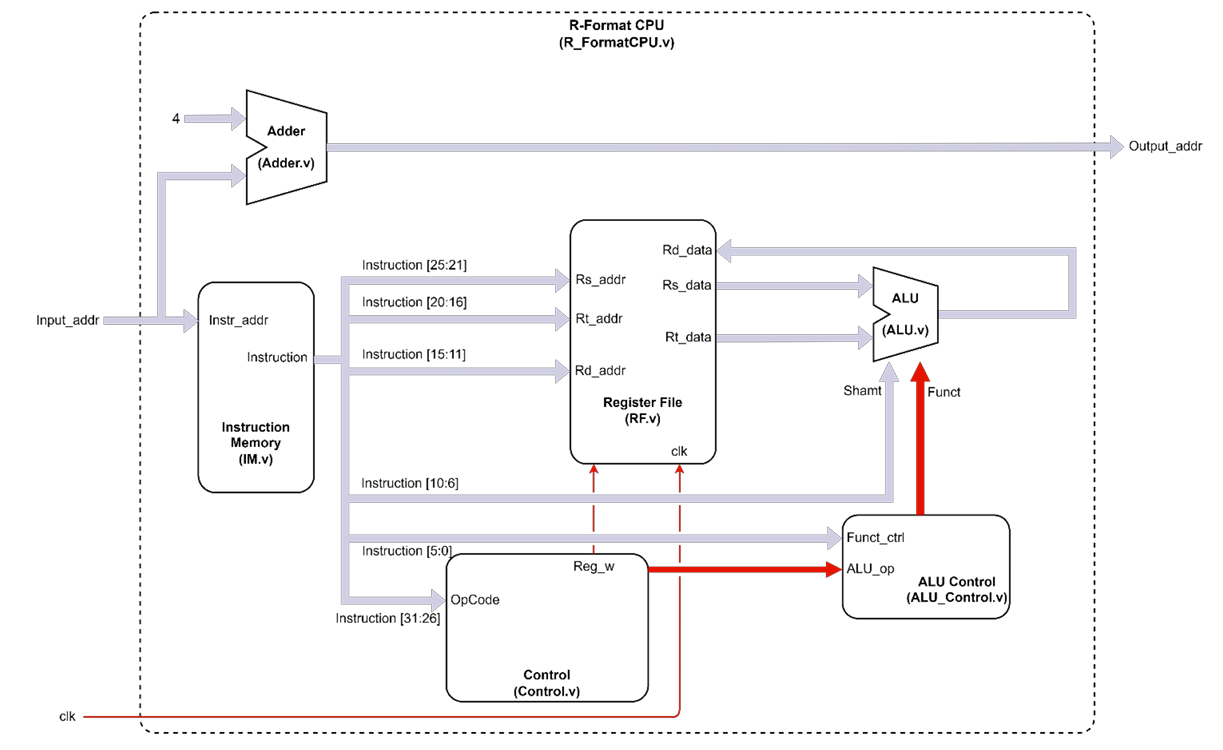
[Part2 88](#_Toc166105222)

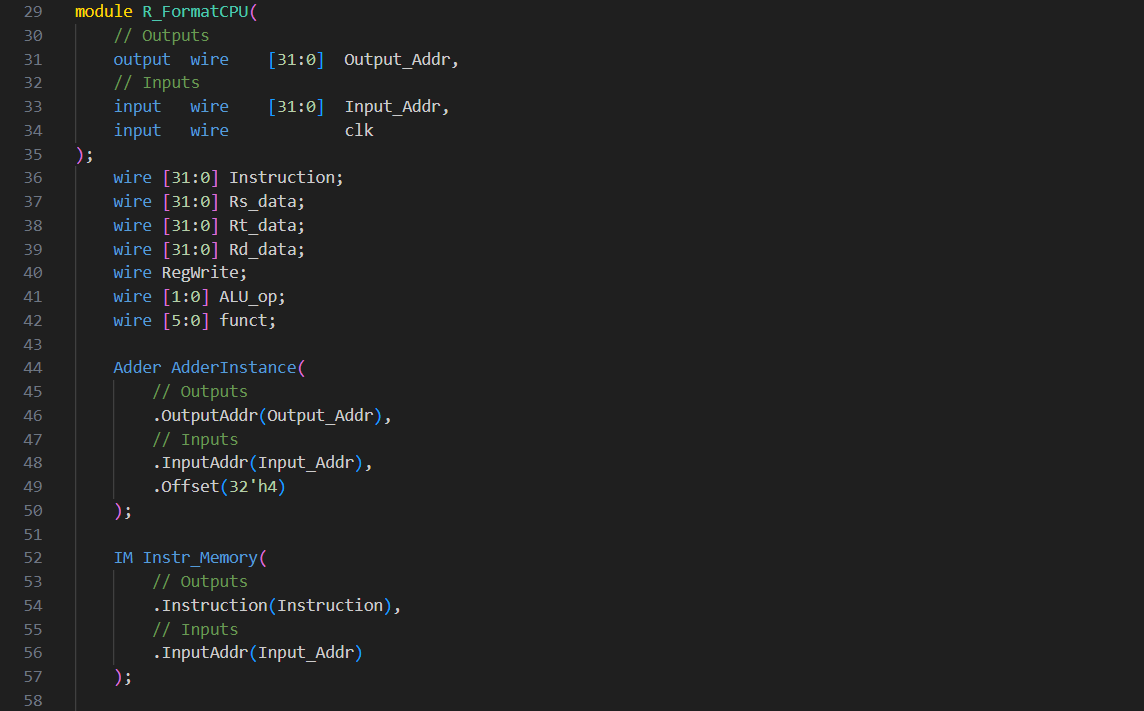
[Part3 89](#_Toc166105223)

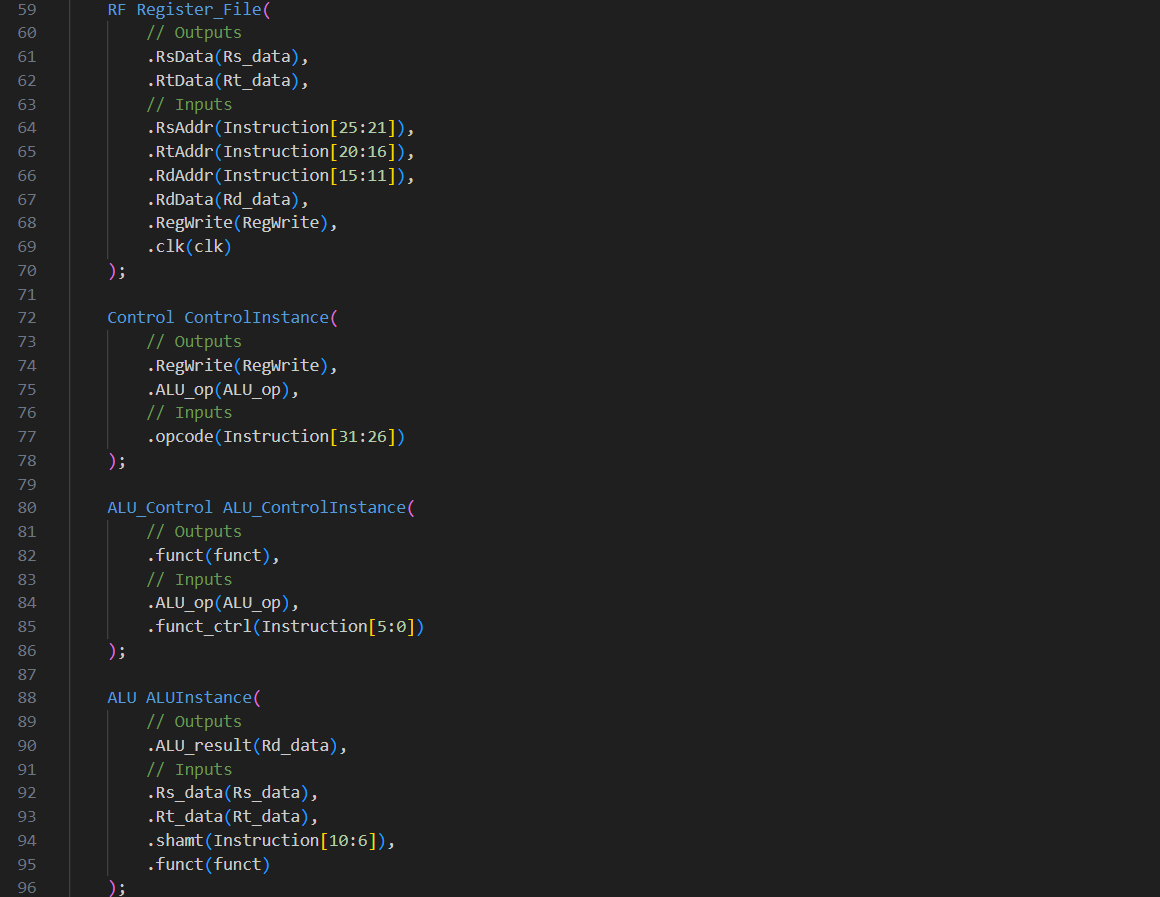
[Conclusion and Insights 99](#_Toc166105224)

# Part1

## R\_FormatCPU







### Description

This module forms the core of a simple MIPS processor, capable of executing R-type instructions. It comprises components for instruction fetching, decoding, register operations, ALU operations, and memory address calculation, all controlled by synchronous clock signals.

### Code Explanation

Line 29~35

This declares a Verilog module named R\_FormatCPU.

It defines three ports:

1. Output\_Addr: Output port for the calculated memory address.
2. Input\_Addr: Input port for the memory address or instruction fetch.
3. clk: Input port for the clock signal.

Line 36~42

These lines declare internal wires to hold various signals and data within the module.

They represent components like registers, ALU control signals, and instruction data.

Line 44~96

These are functional modules which describe how this module work.

### IO

Inputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| Input\_Addr (32-bit) | Address input for fetching instructions and accessing memory. |
| clk (1-bit) | Clock input for synchronous operation. |

Outputs:

|  |  |
| --- | --- |
| Wire name | Description |
| Output\_Addr (32-bit) | Output address for accessing memory or performing other operations. |

### Wires

|  |  |
| --- | --- |
| Wire Name | Description |
| Instruction (32-bit) | Holds the current instruction fetched from memory. |
| Rs\_data, Rt\_data, Rd\_data (32-bit each) | Data read from registers, corresponding to the source and destination operands of instructions. |
| RegWrite (1-bit) | Control signal indicating whether to write data into registers. |
| ALU\_op (2-bit) | Control signal determining the operation to be performed by the Arithmetic Logic Unit (ALU). |
| funct (6-bit) | Function code extracted from the instruction, used for ALU operations or control purposes |

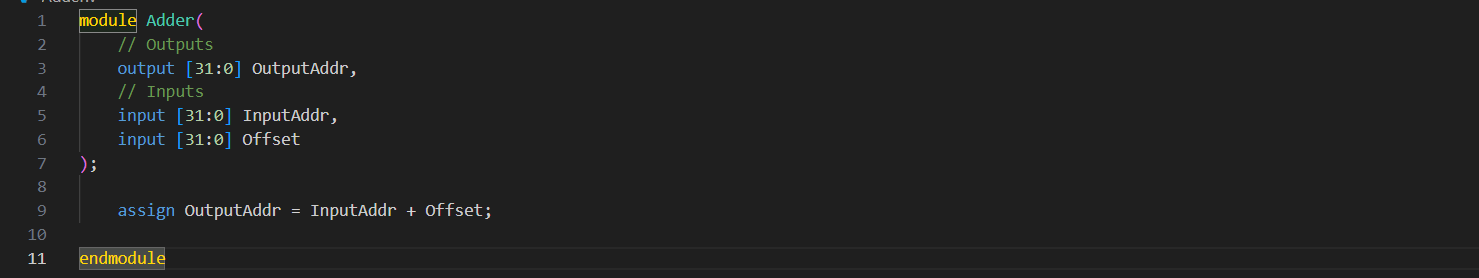
### Components

|  |  |
| --- | --- |
| Component Name | Description |
| AdderInstance | An instance of an adder component, used for calculating memory addresses. It adds the input address (Input\_Addr) with an offset (32’h4) to generate the output address (Output\_Addr). |
| Instr\_Memory | Represents an Instruction Memory component. It outputs the instruction (Instruction) stored at the memory address provided by Input\_Addr. |
| Register\_File | This component simulates a register file, responsible for reading data from and writing data into registers. It reads data from two source registers (Rs\_data, Rt\_data) based on the register addresses extracted from the instruction (Instruction[25:21], Instruction[20:16]). It also allow to store data in the destination register (Rd\_data) using RdData(Instruction[15:11]) and a control signal (RegWrite) to enable register write operations. |
| ControlInstance | Implements control logic based on the opcode of the instruction (Instruction[31:26]). It generates control signals like RegWrite and ALU\_op to coordinate operations within the processor. |
| ALU\_ControlInstance | Determines ALU operation based on the ALU control signals (ALU\_op) and the function code (funct\_ctrl) extracted from the instruction. |
| ALUInstance | Represents the Arithmetic Logic Unit (ALU), performing arithmetic and logical operations. It takes two input data (Rs\_data, Rt\_data), a shamt value (Instruction[10:6]), and a function code (funct) to produce the result (ALU\_result). |

### Process

* The processor fetches an instruction from memory using the Input\_Addr.
* The instruction is decoded, and control signals are generated based on the opcode and function code.
* Data is read from registers based on the instruction's source register addresses.
* The ALU operation is determined based on control signals and executed.
* Resultant data is written back to the destination register if RegWrite is asserted.
* The output address is calculated for the next memory access or operation.

## Adder



### Description

The Adder module performs a basic arithmetic addition operation by adding an input address (InputAddr) with an offset (Offset) to generate an output address (OutputAddr).

### Code Explanation

Line 1~7

Module Declaration:

Declares a Verilog module named Adder.

Defines three ports:

1. OutputAddr: Output port for the calculated address.
2. InputAddr: Input port for the base address.
3. Offset: Input port for the offset to be added to the base address.

Line 9

Logic:

Uses an assign statement to perform the addition operation.

OutputAddr is assigned the result of adding InputAddr and Offset.

### IO

Inputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| InputAddr (32-bit) | Base address for addition. |
| Offset (32-bit) | Offset value to be added to the base address. |

Outputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| OutputAddr (32-bit) | Resultant address obtained by adding the input address and offset. |

### Wires

No internal wires are explicitly declared in this module. The addition operation is directly performed using the assign statement.

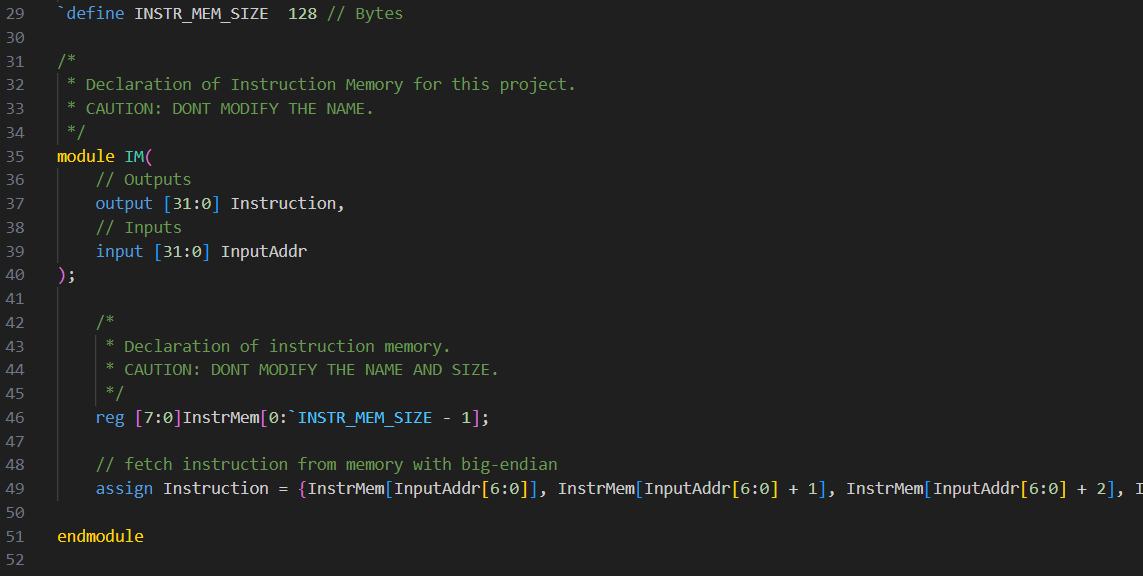
### Components

The Adder module itself acts as a basic arithmetic component performing addition.

### Process

* Input Acquisition: Receives the base address (InputAddr) and offset value (Offset) as inputs.
* Addition Operation: Adds the input address and offset to calculate the output address.
* Output Generation: Provides the calculated output address (OutputAddr) as the module's output.

## IM



### Description

The IM module simulates an instruction memory unit that stores instructions and fetches them based on the input address using big-endian.

### Code Explanation

Line 29

Define constant for later usage.

Line 35~40

Module Declaration:

Defines a Verilog module named IM.

Specifies two ports:

* Instruction: Output port for the fetched instruction.
* InputAddr: Input port for the address used to fetch the instruction.

Line 46

Memory Declaration:

Declares an array InstrMem to represent the instruction memory.

Each element of InstrMem is 8 bits wide (reg [7:0]).

The size of InstrMem is defined by the parameter INSTR\_MEM\_SIZE, which is set to 128 bytes.

Line 49

Instruction Fetch:

Utilizes big-endian byte addressing.

Concatenates four bytes fetched from InstrMem based on the input address InputAddr.

Forms a 32-bit instruction stored in Instruction.

### IO

Inputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| InputAddr (32-bit) | Address input for fetching instructions from memory. |

Outputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| Instruction (32-bit) | Fetched instruction output from memory. |

### Wires

No internal wires are explicitly declared in this module. The instruction is directly assigned using the assign statement.

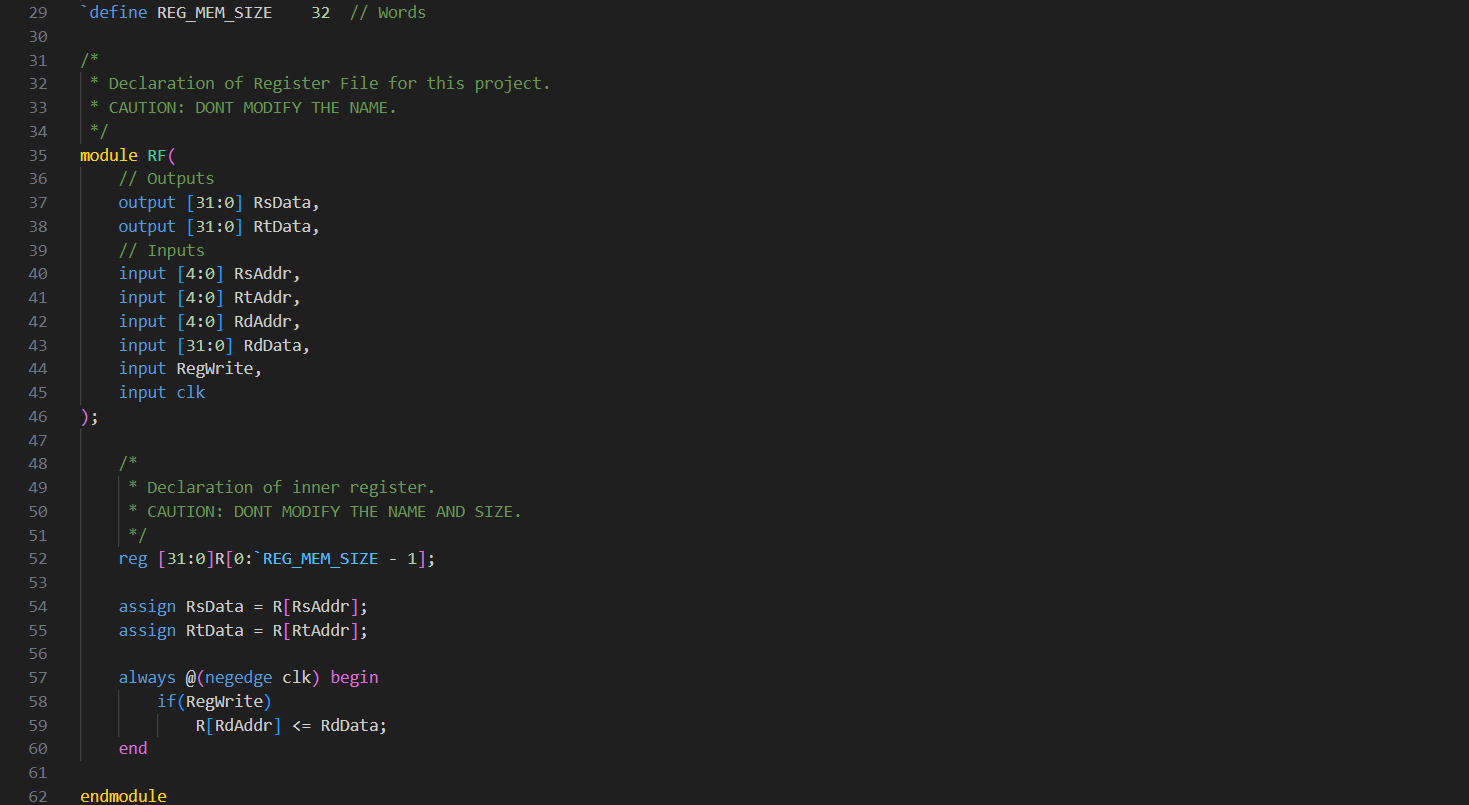
### Components

The IM module itself acts as a memory component to store and fetch instructions.

### Process

* Memory Initialization: The InstrMem array is initialized to store instructions.
* Instruction Fetch: Based on the input address, four bytes are fetched from memory and concatenated to form a 32-bit instruction.
* Output Generation: The fetched instruction is provided as the module's output.

## RF



### Description

The RF module emulates a register file that provides read and write functionality for registers. It has multiple read ports (RsData and RtData) and a write port (RdData) along with control signals (RegWrite and clk) for writing data into registers.

### Code Explanation

Line 29

Define constant for later usage.

Line 35~46

Module Declaration:

Defines a Verilog module named RF.

Specifies two output ports for data read from registers: RsData and RtData.

Specifies input ports for register addresses (RsAddr, RtAddr), write data (RdData), write address (RdAddr), write enable signal (RegWrite), and clock signal (clk).

Line 52

Register Declaration:

Declares an array R to represent the register file.

Each element of R is 32 bits wide (reg [31:0]).

The size of R is defined by the parameter REG\_MEM\_SIZE, which is set to 32 words.

Line 54~55

Register Read:

Assigns the data from the specified registers (RsAddr and RtAddr) to the output ports RsData and RtData, respectively.

Line 57~60

Register Write:

Utilizes an always block triggered on the negative edge(avoid racing) of the clock signal (clk).

If the RegWrite signal is asserted, the data RdData is written into the register specified by the address RdAddr.

### IO

Inputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| RsAddr, RtAddr, RdAddr (5-bit each) | Addresses for reading from or writing to registers. |
| RdData (32-bit) | Data to be written into the register specified by RdAddr. |
| RegWrite (1-bit) | Write enable signal. It controls register writing. |
| clk (1-bit) | Clock signal for synchronous operation. |

Outputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| RsData, RtData (32-bit each) | Data read from the registers specified by RsAddr and RtAddr, respectively. |

### Wires

No internal wires are explicitly declared in this module. Register data is directly assigned to output ports using the assign statement.

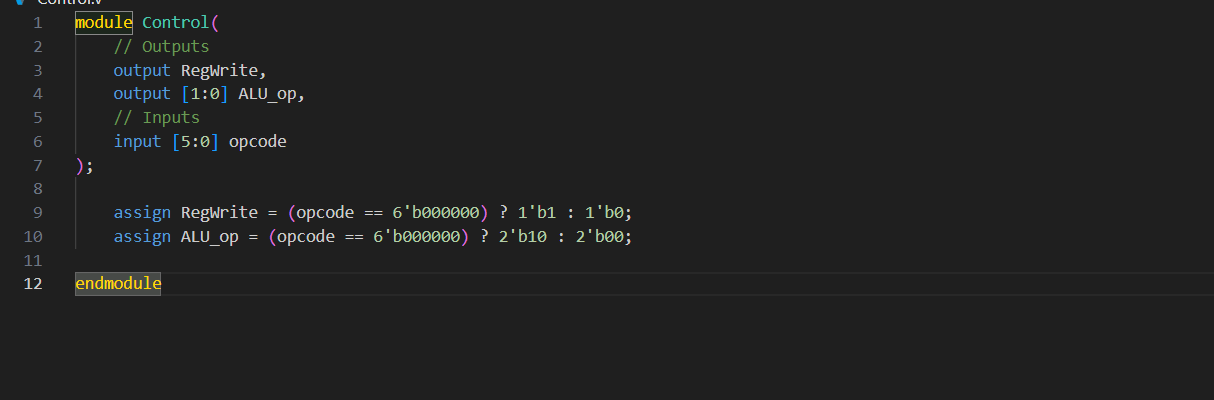
### Components

The RF module itself acts as a register file component, providing read and write functionality for registers.

### Process

* Read Operation: Data from specified registers (RsAddr and RtAddr) is assigned to output ports RsData and RtData, respectively.
* Write Operation: On the negative edge of the clock signal, if the RegWrite signal is asserted, data RdData is written into the register specified by RdAddr.

## Control



### Description

The Control module is responsible for generating control signals based on the opcode of the instruction. It determines whether the instruction requires register writes and specifies the operation to be performed by the Arithmetic Logic Unit (ALU)

### Code Explanation

Line 1~7

Module Declaration:

Defines a Verilog module named Control.

Specifies two output ports for control signals: RegWrite (1-bit) and ALU\_op (2-bit).

Specifies an input port opcode (6-bit) to receive the opcode of the instruction.

Line 9~10

The RegWrite signal is determined by checking if the opcode matches a specific value (6'b000000). If it does, RegWrite is set to 1'b1; otherwise, it is set to 1'b0.

The ALU\_op signal is determined similarly. If the opcode matches 6'b000000, indicating an R-type instruction, ALU\_op is set to 2'b10 to indicate that the ALU should perform an operation. Otherwise, it is set to 2'b00.

### IO

Inputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| opcode (6-bit) | Input port to receive the opcode of the instruction. |

Outputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| RegWrite (1-bit) | Output port indicating whether register write is enabled. |
| ALU\_op (2-bit) | Output port specifying the ALU operation code. |

### Wires

No internal wires are explicitly declared in this module. Control signals are directly assigned using the assign statement.

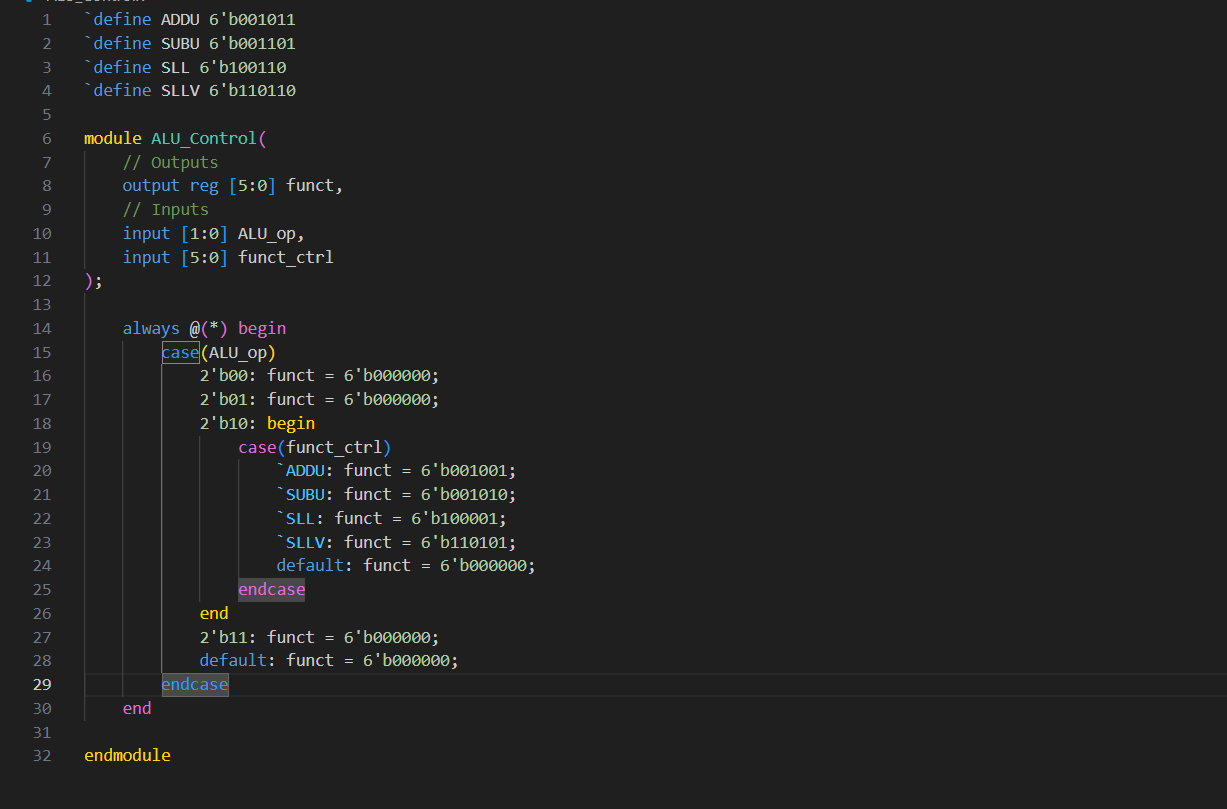
### Components

The Control module itself acts as a control unit responsible for generating control signals.

### Process

* Opcode Decoding: Receives the opcode of the instruction.
* Control Signal Generation: Based on the opcode, generates control signals such as RegWrite and ALU\_op.
* Output: Provides the generated control signals as module outputs.

## ALU\_Control



### Description

The ALU\_Control module decodes the ALU operation code and selects the appropriate function code for the ALU based on both the ALU operation code and an additional control signal (funct\_ctrl). The function code determines the specific operation to be performed by the ALU.

### Code Explanation

Line 1~4

Function code table.

Line 6~12

Module Declaration:

Defines a Verilog module named ALU\_Control.

Specifies an output port funct (6-bit) for the ALU function code.

Specifies input ports ALU\_op (2-bit) and funct\_ctrl (6-bit) for the ALU operation code and additional control signal, respectively.

Line 14~30

Control Signal Generation:

Utilizes an always block triggered whenever the inputs change (@(\*)).

Decodes the ALU operation code ALU\_op and selects the appropriate function code based on the provided control signal funct\_ctrl.

Generates the function code funct according to the selected operation.

### IO

Inputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| ALU\_op (2-bit) | ALU operation code input. |
| funct\_ctrl (6-bit) | Additional control signal used for selecting the ALU function code. |

Outputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| funct (6-bit) | Output port representing the ALU function code. |

### Wires

No internal wires are explicitly declared in this module. The function code is directly assigned using an always block.

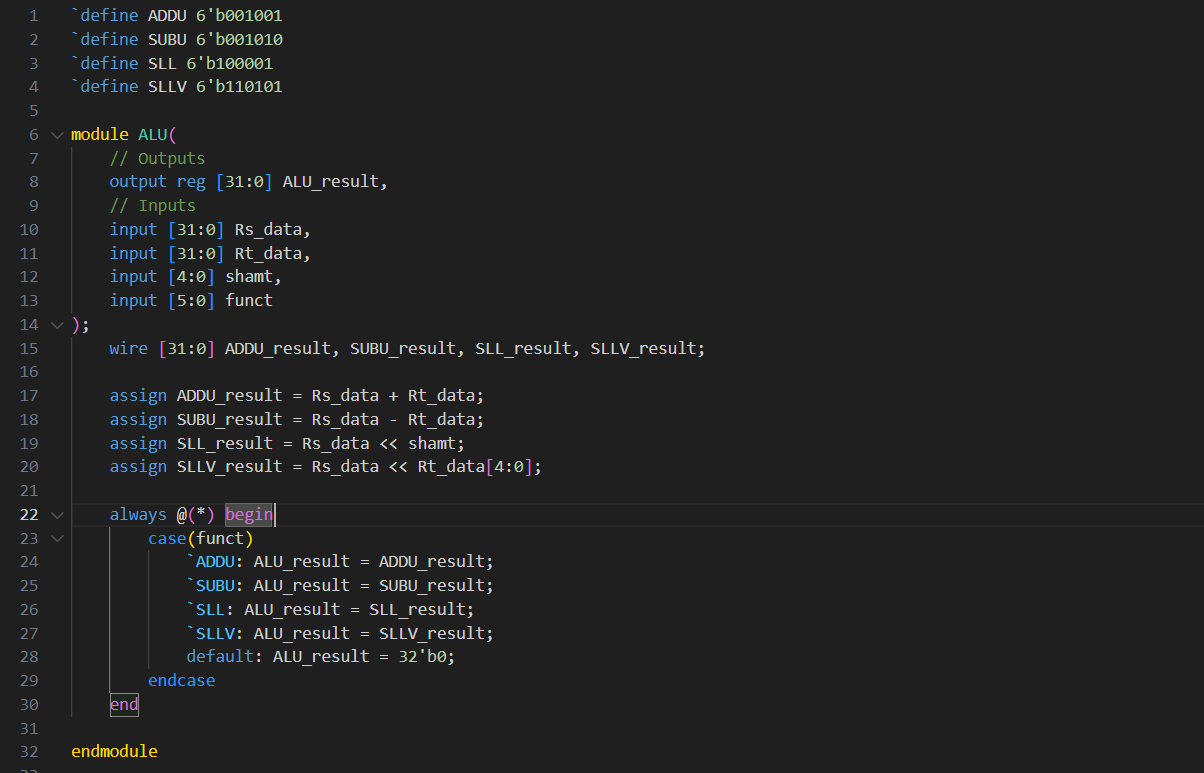
### Components

The ALU\_Control module itself acts as a control unit responsible for generating the ALU function code.

### Process

* ALU Operation Code Decoding: Receives the ALU operation code (ALU\_op).
* Function Code Selection: Based on the ALU operation code and the provided control signal (funct\_ctrl), selects the appropriate function code for the ALU.
* Output Generation: Provides the generated function code (funct) as the module's output.

## ALU



### Description

The ALU module simulates an Arithmetic Logic Unit (ALU) which executes arithmetic and logical operations such as addition, subtraction, left logical shift, and variable left logical shift. It takes two input data operands (Rs\_data, Rt\_data), a shamt value (shamt), and a function code (funct) to produce the result (ALU\_result).

### Code Explanation

Line 1~4

Operation code table.

Line 6~14

Module Declaration:

Defines a Verilog module named ALU.

Specifies an output port ALU\_result (32-bit) for the result of ALU operations.

Specifies input ports for the two data operands (Rs\_data, Rt\_data), shamt value (shamt), and function code (funct).

Line 15~20

ALU Operation:

Computes the results of various ALU operations (ADDU, SUBU, SLL, SLLV) and stores them in separate wires (ADDU\_result, SUBU\_result, SLL\_result, SLLV\_result).

Line 22~30

Result Selection:

Uses an always block triggered whenever the inputs change (@(\*)).

Selects the result based on the function code (funct) using a case statement.

Assigns the selected result to the output port ALU\_result.

### IO

Inputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| Rs\_data, Rt\_data (32-bit each) | Input data operands for the ALU operations. |
| shamt (5-bit) | Shift amount for left logical shift operation. |
| funct (6-bit) | Function code specifying the operation to be performed by the ALU. |

Outputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| ALU\_result (32-bit) | Result of the ALU operation which is a register because it needs to be assigned as left value in always block. |

### Wires

ADDU\_result, SUBU\_result, SLL\_result, SLLV\_result (32-bit each): Wires to store the intermediate results of ALU operations.

### Components

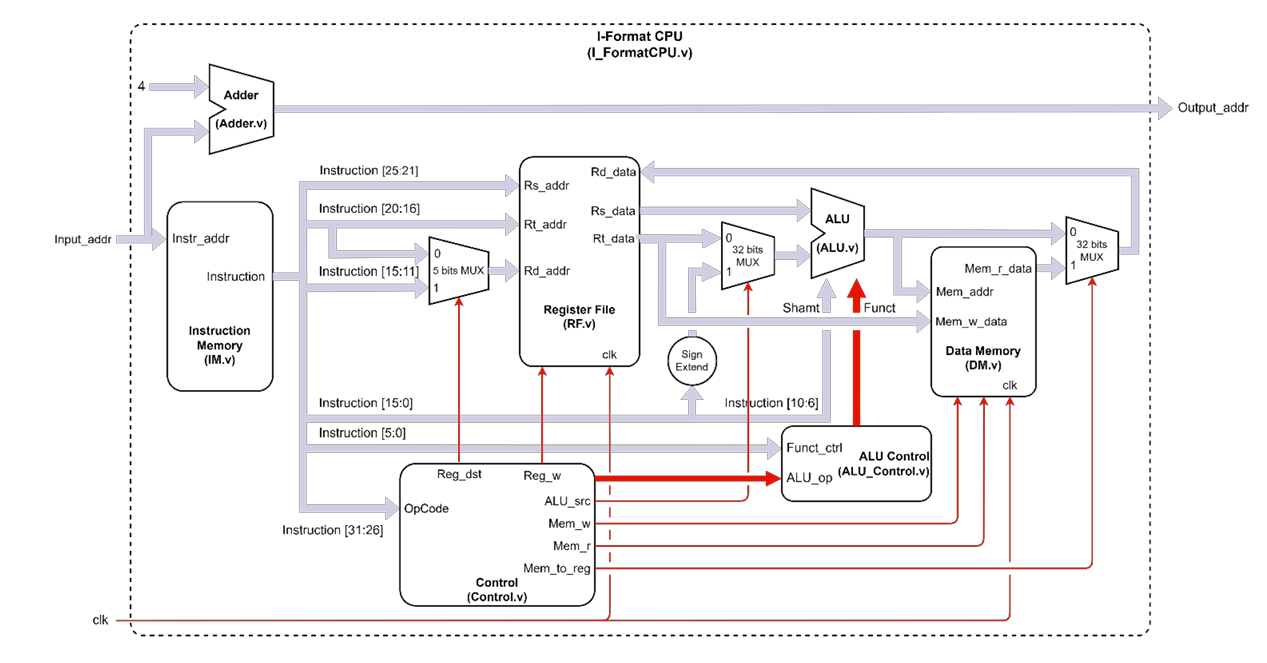
The ALU module itself acts as an Arithmetic Logic Unit (ALU) component capable of performing various arithmetic and logical operations.

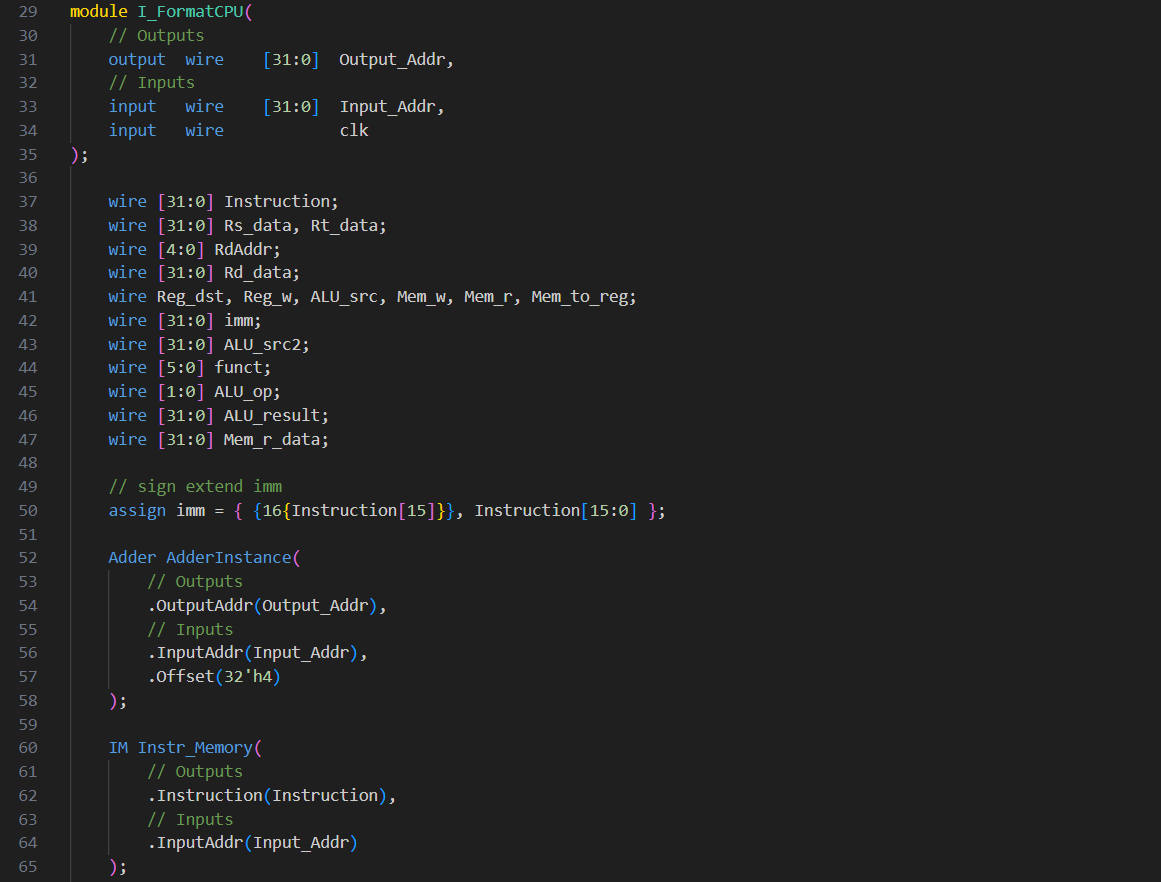
### Process

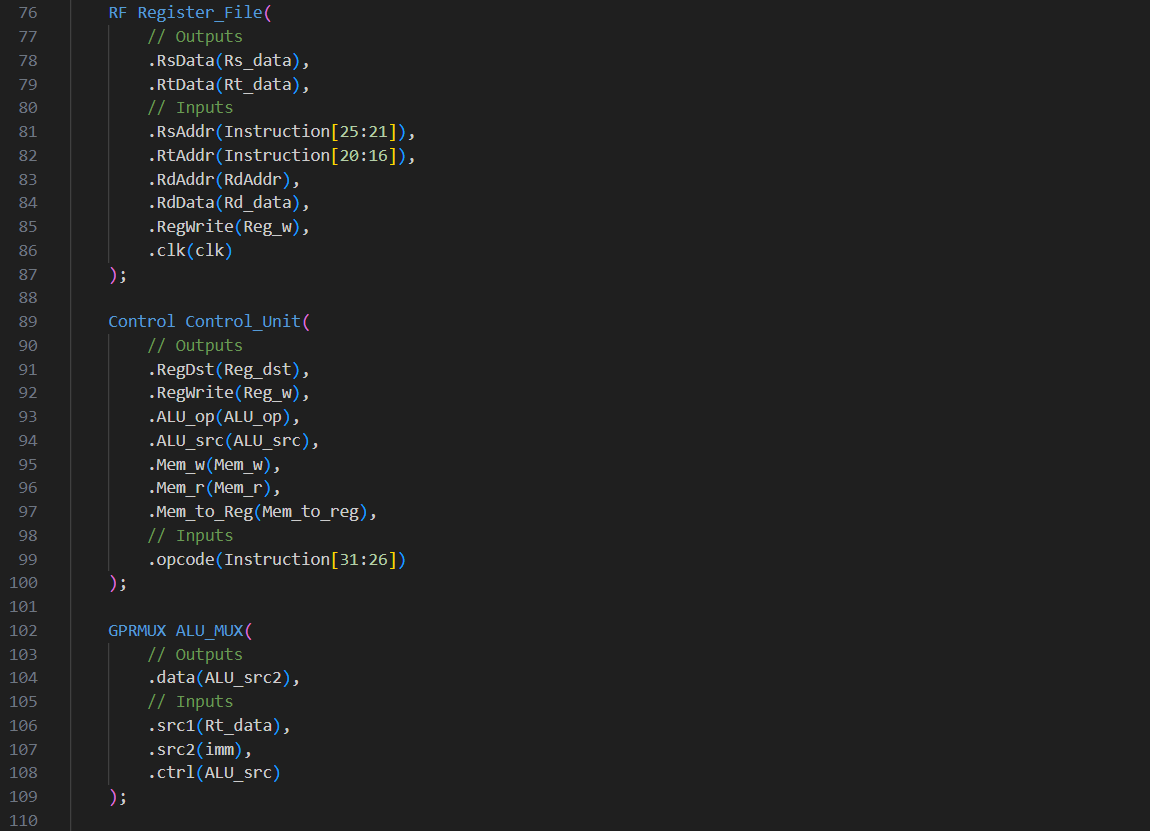
* ALU Operation: Computes the results of various ALU operations (ADDU, SUBU, SLL, SLLV).
* Result Selection: Selects the appropriate result based on the function code (funct) and assigns it to the output port ALU\_result.

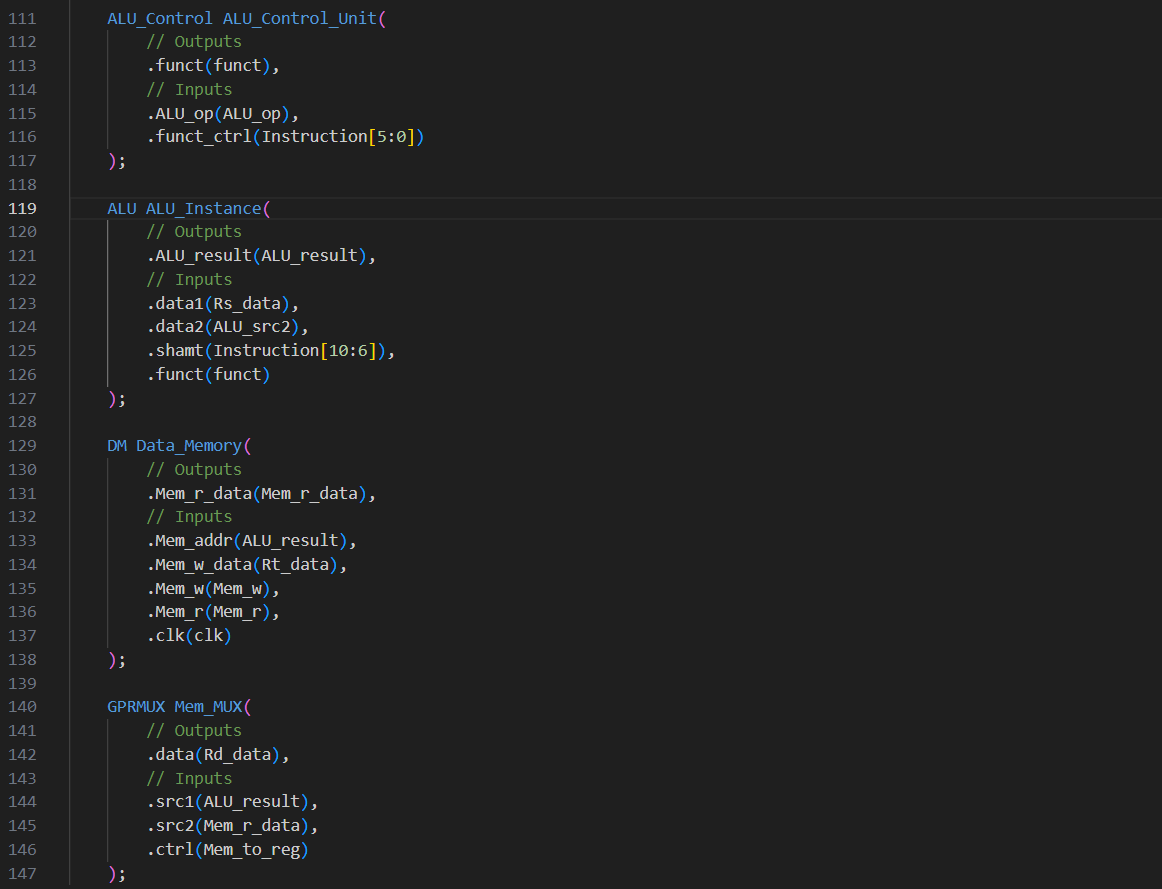
# Part2

## I\_FormatCPU









### Description

The I\_FormatCPU module is a CPU implementation tailored to execute instructions encoded in the I-format, which typically involve operations with immediate values. It incorporates various components such as instruction memory, register file, ALU, and data memory to execute instructions and perform data operations.

### Code Explanation

Line 29~35

Module Declaration:

Defines a Verilog module named I\_FormatCPU.

Specifies output port Output\_Addr (32-bit) for the computed output address.

Specifies input ports for the input address Input\_Addr (32-bit) and clock signal clk.

Line 37~47

Wires used in this scope.

Line 50

Signed extension

Line 52~147

Functional modules

### IO

Inputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| Input\_Addr (32-bit) | Input address for fetching instructions. |
| clk (1-bit) | Clock signal for synchronous operation. |

Outputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| Output\_Addr (32-bit) | Computed output address. |

### Wires

Various wires are used within the module to connect different components and facilitate data flow and control signals. These wires include:

|  |  |
| --- | --- |
| Wire Name | Description |
| Instruction (32-bit) | Instruction fetched from the instruction memory. |
| Rs\_data, Rt\_data (32-bit each) | Data read from the register file. |
| RdAddr (5-bit) | Register address selected for writing data. |
| Rd\_data | Data to be written into the selected register. |
| Reg\_dst, Reg\_w, ALU\_src, Mem\_w, Mem\_r, Mem\_to\_reg (1-bit each) | Control signals for various components. |
| Imm (32-bit) | Sign-extended immediate value extracted from the instruction. |
| ALU\_src2 (32-bit) | Second input operand for the ALU. |
| funct (6-bit), ALU\_op (2-bit) | Control signals for ALU operations. |
| ALU\_result (32-bit) | Result produced by the ALU. |
| Mem\_r\_data | Data read from the data memory. |

### Components

The I\_FormatCPU module integrates several components to execute instructions and perform data operations:

|  |  |
| --- | --- |
| Component Name | Description |
| IM | Stores and provides instructions to the CPU. |
| Adder | Compute addresses for the next instruction and branch targets. |
| GPRMUX, RegMUX | Select between different data sources based on control signals. |
| RF | Stores and provides data from registers. |
| ALU | Performs arithmetic and logical operations. |
| DM | Stores and retrieves data values. |
| Control | Generates control signals based on instruction opcode. |

### Process

* Instruction Fetch: Fetches instructions from the instruction memory based on the input address.
* Control Signal Generation: Generates control signals based on the opcode of the fetched instruction.
* Register Read: Reads data from the register file based on the register addresses specified in the instruction.
* Immediate Extraction: Extracts and sign-extends the immediate value from the instruction.
* ALU Operation Selection: Selects appropriate ALU operations based on control signals and instruction encoding.
* ALU Operation Execution: Performs arithmetic and logical operations using the ALU.
* Data Memory Access: Reads or writes data to the data memory based on control signals.
* Register Write: Writes data into the register file based on control signals and selected register address.
* Output Address Computation: Computes and provides the output address as the module's output.

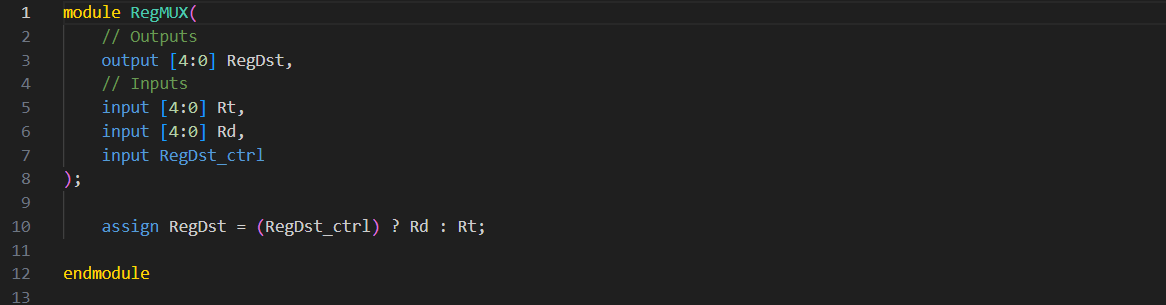
## Adder

It’s the same as [Part1’s](#_Adder).

## IM

It’s the same as [Part1’s](#_IM).

## RegMUX



### Description

The RegMUX module is a multiplexer that selects one of two input register addresses (Rt and Rd) as the output based on the control signal RegDst\_ctrl. This module is typically used in digital processor designs to determine the destination register address for data writes.

### Code Explanation

Line 1~8

Module Declaration:

Defines a Verilog module named RegMUX.

Specifies an output port RegDst (5-bit) for the selected register address.

Specifies input ports Rt and Rd (5-bit each) for the two input register addresses, and RegDst\_ctrl (1-bit) for the control signal.

Line 10

Multiplexing Operation:

Utilizes an assign statement to select one of the input register addresses (Rt or Rd) based on the value of the control signal RegDst\_ctrl.

If RegDst\_ctrl is asserted (1), the output RegDst will be equal to Rd; otherwise, it will be equal to Rt.

### IO

Inputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| Rt, Rd (5-bit each) | Input register addresses. |
| RegDst\_ctrl (1-bit) | Control signal for selecting the destination register address. |

Outputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| RegDst (5-bit) | Selected register address. |

### Wires

No internal wires are explicitly declared in this module. The output RegDst is directly assigned using the assign statement.

### Components

The RegMUX module itself acts as a multiplexer component responsible for selecting between two input register addresses based on a control signal.

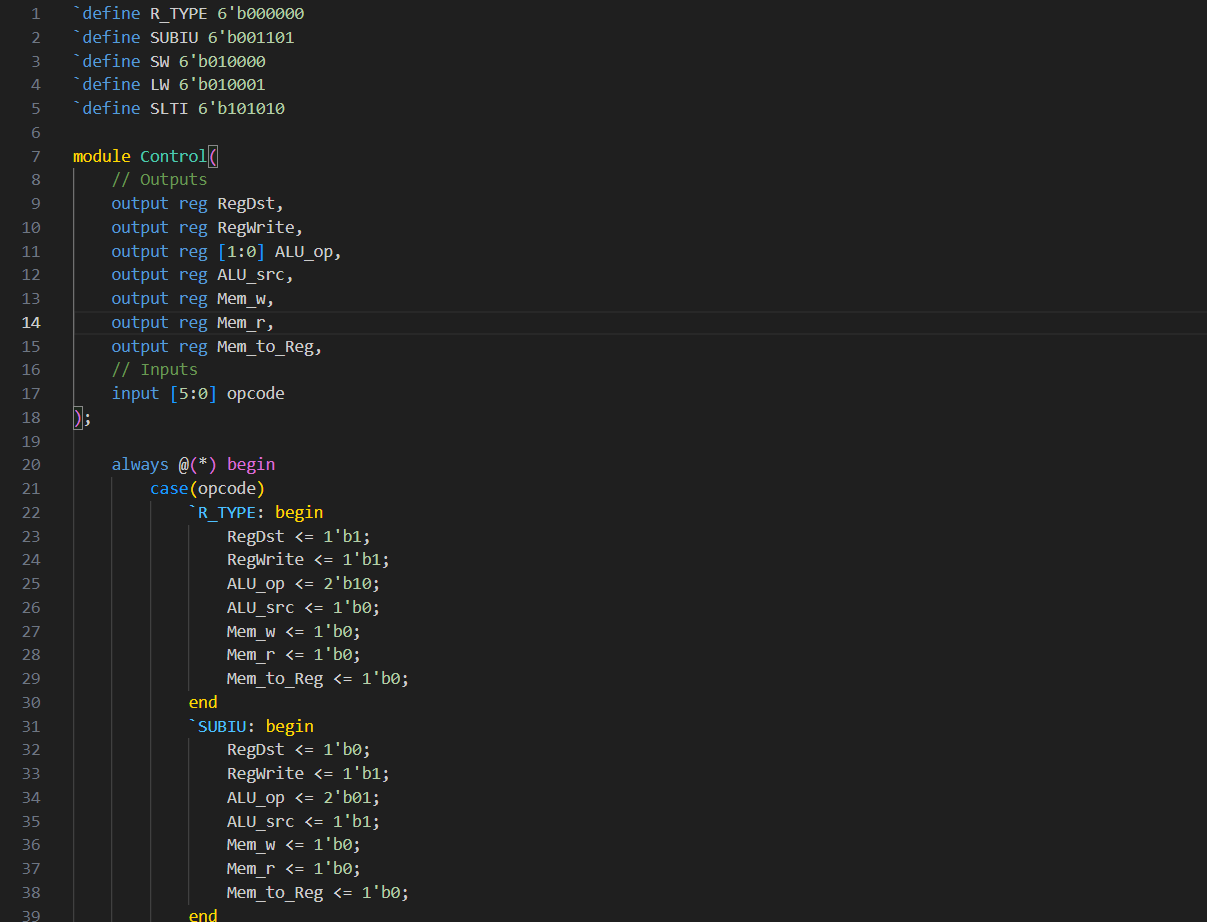
### Process

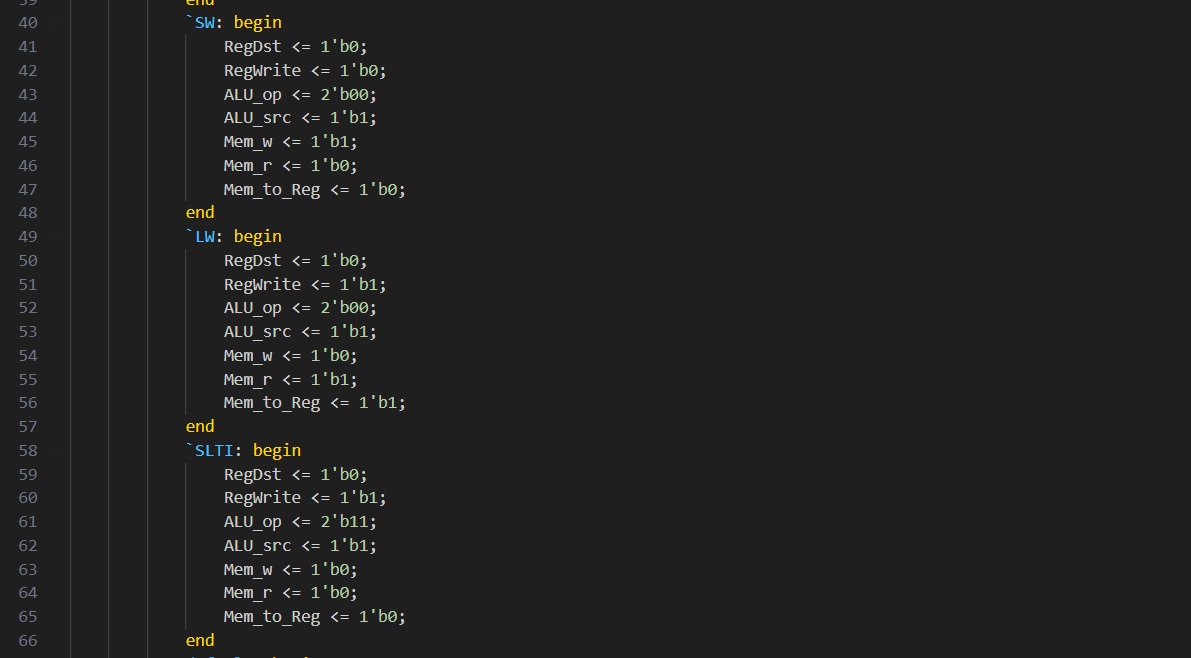
* Control Signal Interpretation: Receives the control signal RegDst\_ctrl.
* Input Selection: Determines the selected output based on the control signal:
  + If RegDst\_ctrl is asserted (1), selects the input register address Rd.
  + Otherwise, selects the input register address Rt.
* Output Assignment: Assigns the selected input register address to the output port RegDst.

## RF

It’s the same as [Part1’s](#_RF).

## Control







### Description

The Control module serves as the control unit of a CPU, responsible for generating control signals based on the opcode of an instruction. It determines how the CPU should process the instruction, including register operations, ALU operations, and memory access.

### Code Explanation

Line 1~5

Opcode table.

Line 7~18

Module Declaration:

Defines a Verilog module named Control.

Specifies output ports for various control signals:

* RegDst: Register destination control signal.
* RegWrite: Register write enable control signal.
* ALU\_op: ALU operation control signal.
* ALU\_src: ALU source control signal.
* Mem\_w: Memory write enable control signal.
* Mem\_r: Memory read enable control signal.
* Mem\_to\_Reg: Memory to register control signal.

Specifies an input port opcode (6-bit) for the opcode of the instruction.

Line 20~77

Control Signal Generation:

Utilizes an always block triggered whenever the inputs change (@(\*)).

Uses a case statement to select control signals based on the opcode of the instruction.

Generates specific control signals for different types of instructions, including R-type, immediate arithmetic, and memory access instructions. If not recognized, it’ll use default condition.

### IO

Inputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| opcode (6-bit) | Opcode of the instruction. |

Outputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| RegDst (1-bit) | Register destination control signal. |
| RegWrite (1-bit) | Register write enable control signal. |
| ALU\_op (2-bit) | ALU operation control signal. |
| ALU\_src (1-bit) | ALU source control signal. |
| Mem\_w (1-bit) | Memory write enable control signal. |
| Mem\_r (1-bit) | Memory read enable control signal. |
| Mem\_to\_Reg (1-bit) | Memory to register control signal. |

### Wires

No internal wires are explicitly declared in this module. Control signals are directly assigned based on the opcode using the always block.

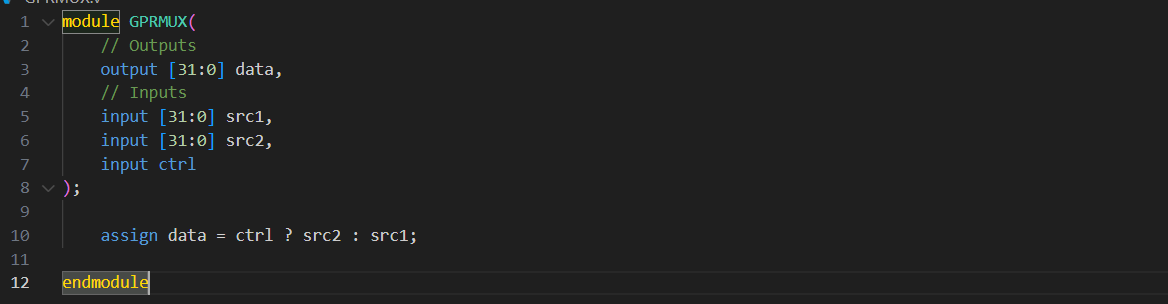
### Components

The Control module acts as a standalone component responsible for generating control signals based on the opcode of an instruction.

### Process

* Opcode Interpretation: Receives the opcode of the instruction.
* Control Signal Selection: Determines the appropriate control signals based on the opcode using a case statement.
* Control Signal Assignment: Assigns the selected control signals to the output ports.

## GPRMUX



### Description

The GPRMUX module functions as a multiplexer, allowing for the selection of one of two input data sources based on the value of a control signal. This type of multiplexer is commonly used in digital circuit design to facilitate data selection and routing.

### Code Explanation

Line 1~8

Module Declaration:

Declares a Verilog module named GPRMUX.

Defines an output port data (32-bit) for the selected data.

Specifies two input ports src1 and src2 (both 32-bit) for the two input data sources.

Specifies an input port ctrl (1-bit) for the control signal.

Line 10

Multiplexing Operation:

Utilizes an assign statement to select one of the input data sources (src1 or src2) based on the value of the control signal ctrl.

If ctrl is asserted (1), the output data will be equal to src2; otherwise, it will be equal to src1.

### IO

Inputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| src1 (32-bit) | First input data source. |
| src2 (32-bit) | Second input data source. |
| ctrl (1-bit) | Control signal for selecting the data source. |

Outputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| data (32-bit) | Selected output data. |

### Wires

No internal wires are explicitly declared in this module. The output data is directly assigned using the assign statement.

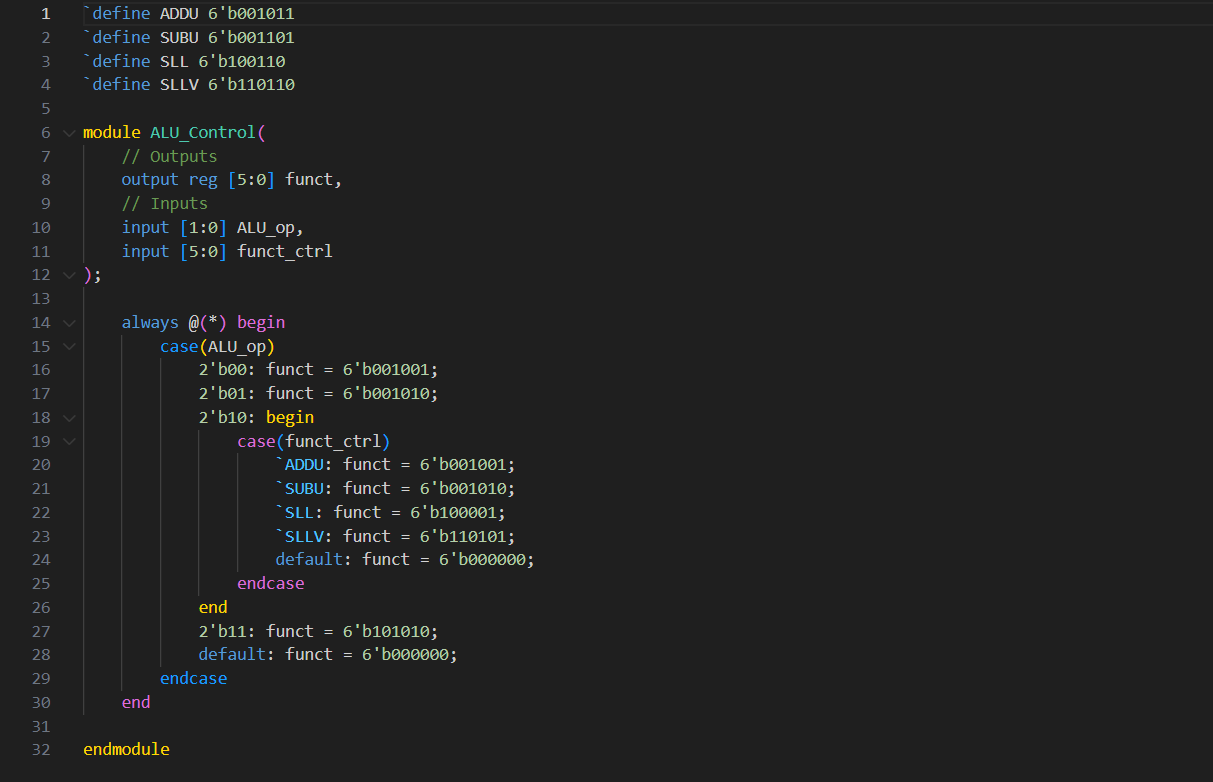
### Components

The GPRMUX module itself acts as a multiplexer component responsible for selecting between two input data sources based on a control signal.

### Process

* Control Signal Interpretation: Receives the control signal ctrl.
* Input Selection: Determines the selected output based on the control signal:
  + If ctrl is asserted (1), selects the second input data source src2.
  + Otherwise, selects the first input data source src1.
* Output Assignment: Assigns the selected input data source to the output port data.

## ALU\_Control



### Description

The ALU\_Control module functions as the control unit for the ALU (Arithmetic Logic Unit) in a CPU. It generates the appropriate function code for the ALU based on the ALU operation code and the function code control signal.

### Code Explanation

Line 1~4

LUT.

Line 6~12

Module Declaration:

Defines a Verilog module named ALU\_Control.

Specifies an output port funct (6-bit) for the generated ALU function code.

Specifies input ports for:

ALU\_op (2-bit): ALU operation code.

funct\_ctrl (6-bit): Function code control signal.

Line 14~30

ALU Function Code Generation:

Utilizes an always block triggered whenever the inputs change (@(\*)).

Uses a case statement to select the appropriate ALU function code based on the ALU operation code (ALU\_op) and the function code control signal (funct\_ctrl).

Generates specific function codes for different ALU operations, including addition, subtraction, logical shift left, logical shift left variable, and set on less than (SLT). Opcode==00 is specified as ADD; opcode==01 is specified as SUB; opcode==11 is specified as SLTI.

### IO

Inputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| ALU\_op (2-bit) | ALU operation code. |
| funct\_ctrl (6-bit) | Function code control signal. |

Outputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| funct (6-bit) | Generated ALU function code. |

### Wires

No internal wires are explicitly declared in this module. The output funct is directly assigned within the always block.

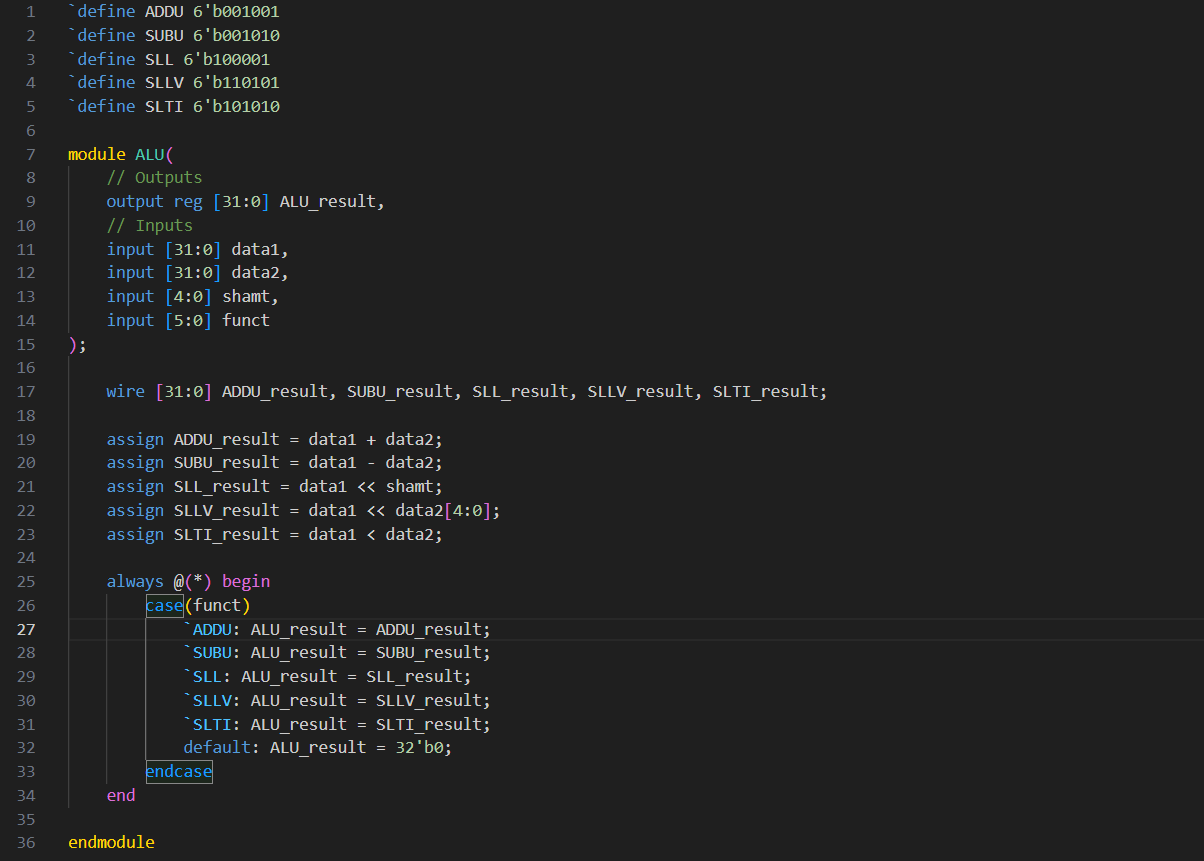
### Components

The ALU\_Control module serves as a standalone component responsible for generating the ALU function code based on the provided inputs.

### Process

* ALU Operation Interpretation: Receives the ALU operation code ALU\_op.
* Function Code Selection: Determines the appropriate ALU function code based on the ALU operation code and the function code control signal.
* Function Code Assignment: Assigns the selected ALU function code to the output port funct.

## ALU



### Description

The Verilog module ALU represents the Arithmetic Logic Unit (ALU) of a processor. It performs arithmetic and logical operations on two input data sources (data1 and data2) based on the given function code (funct) and shift amount (shamt). The result of the operation is stored in the output ALU\_result.

### Code Explanation

Line 1~5

Operation code table.

Line 7~15

Module Declaration:

Defines a Verilog module named ALU.

Declares an output port ALU\_result (32-bit) to store the result of the ALU operation.

Specifies input ports for:

data1 (32-bit): First input data source.

data2 (32-bit): Second input data source.

shamt (5-bit): Shift amount for logical shift operations.

funct (6-bit): Function code for selecting the ALU operation.

Line 17~23

ALU Operation Assignment:

Defines wires for the intermediate results of various ALU operations: ADDU\_result, SUBU\_result, SLL\_result, SLLV\_result, and SLTI\_result.

Calculates the results of different ALU operations using assign statements based on the input data and shift amount.

Line 25~34

ALU Result Assignment:

Utilizes an always block triggered whenever the inputs change (@(\*)).

Uses a case statement to select the appropriate ALU result based on the given function code (funct).

Assigns the selected ALU result to the output port ALU\_result.

### IO

Inputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| data1 (32-bit) | First input data source. |
| data2 (32-bit) | Second input data source. |
| shamt (5-bit) | Shift amount for logical shift operations. |
| funct (6-bit) | Function code for selecting the ALU operation. |

Outputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| ALU\_result (32-bit) | Result of the ALU operation. |

### Wires

ADDU\_result, SUBU\_result, SLL\_result, SLLV\_result, SLTI\_result: Intermediate results of various ALU operations.

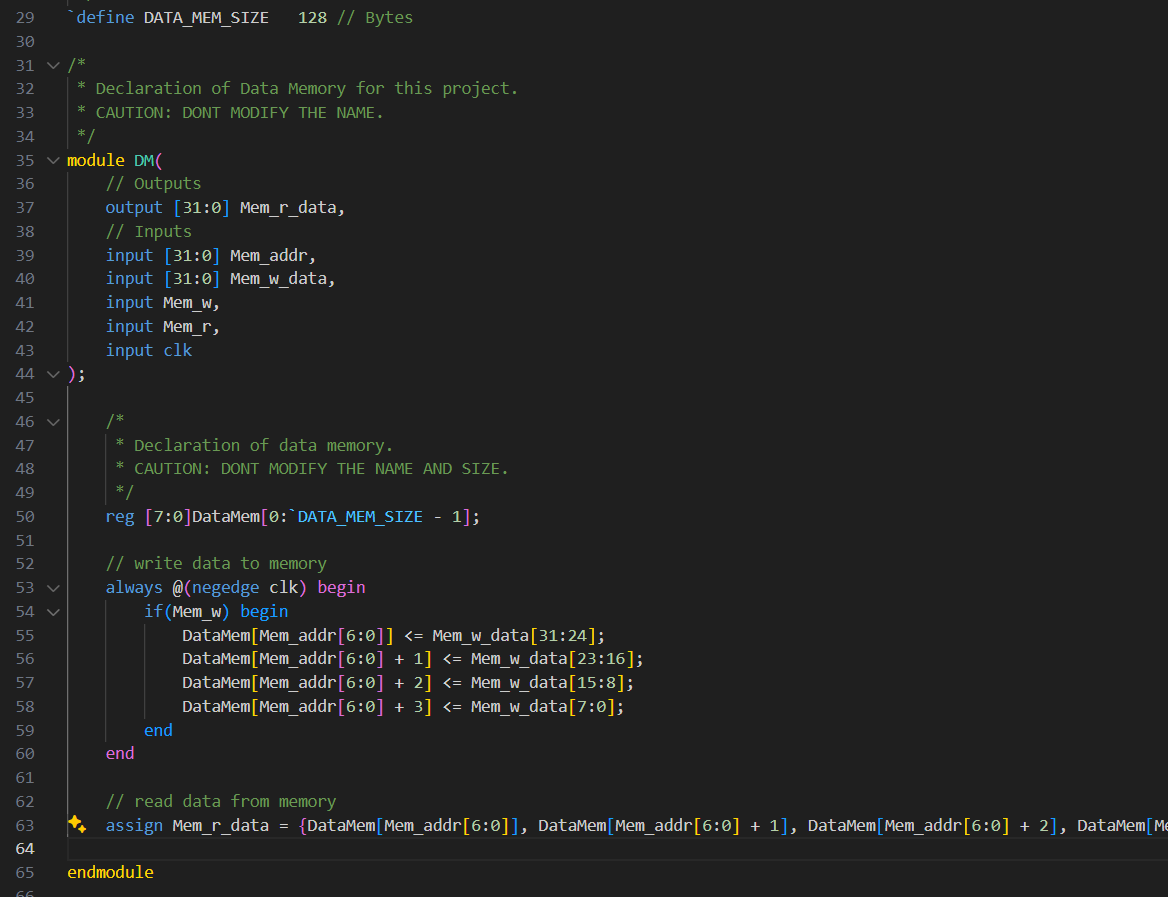
### Components

The ALU module contains combinatorial logic to perform arithmetic and logical operations based on the given function code.

### Process

* Operation Calculation: Calculates the results of different ALU operations based on the input data sources and shift amount.
* Result Selection: Selects the appropriate ALU result based on the given function code.
* Result Assignment: Assigns the selected ALU result to the output port ALU\_result.

## DM



### Description

The Verilog module DM represents the Data Memory component of a processor. It is responsible for storing and retrieving data values based on memory addresses. The module consists of a memory array (DataMem) and logic for writing data to memory on the falling edge of the clock signal (clk) when the write enable signal (Mem\_w) is asserted. Additionally, it provides logic(big-endian) for reading data from memory based on the memory address provided (Mem\_addr). The output Mem\_r\_data contains the retrieved data.

### Code Explanation

Line 29

Define the memory size.

Line 35~44

Module Declaration:

Defines a Verilog module named DM.

Specifies an output port Mem\_r\_data (32-bit) to provide the retrieved data from memory.

Specifies input ports for:

Mem\_addr (32-bit): Memory address for read/write operations.

Mem\_w\_data (32-bit): Data to be written into memory.

Mem\_w: Write enable signal.

Mem\_r: Read enable signal.

clk: Clock signal.

Line 50

Data Memory Declaration:

Declares a memory array named DataMem to store data values.

The memory array has a size determined by the parameter DATA\_MEM\_SIZE.

Line 53~60

Write Operation:

Utilizes an always block triggered on the falling edge(avoid racing) of the clock (negedge clk).

Checks if the write enable signal (Mem\_w) is asserted.

If asserted, writes the 32-bit data Mem\_w\_data into the memory array DataMem at the specified address Mem\_addr.

Line 63

Read Operation:

Utilizes an assign statement to concatenate four bytes from memory and assign them to Mem\_r\_data.

Reads data from memory at the specified address Mem\_addr and provides it as output.

### IO

Inputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| Mem\_addr (32-bit) | Memory address for read/write operations. |
| Mem\_w\_data (32-bit) | Data to be written into memory. |
| Mem\_w (1-bit) | Write enable signal. |
| Mem\_r (1-bit) | Read enable signal. |
| clk (1-bit) | Clock signal. |

Outputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| Mem\_r\_data (32-bit) | Retrieved data from memory. |

### Wires

No internal wires are explicitly declared in this module.

### Components

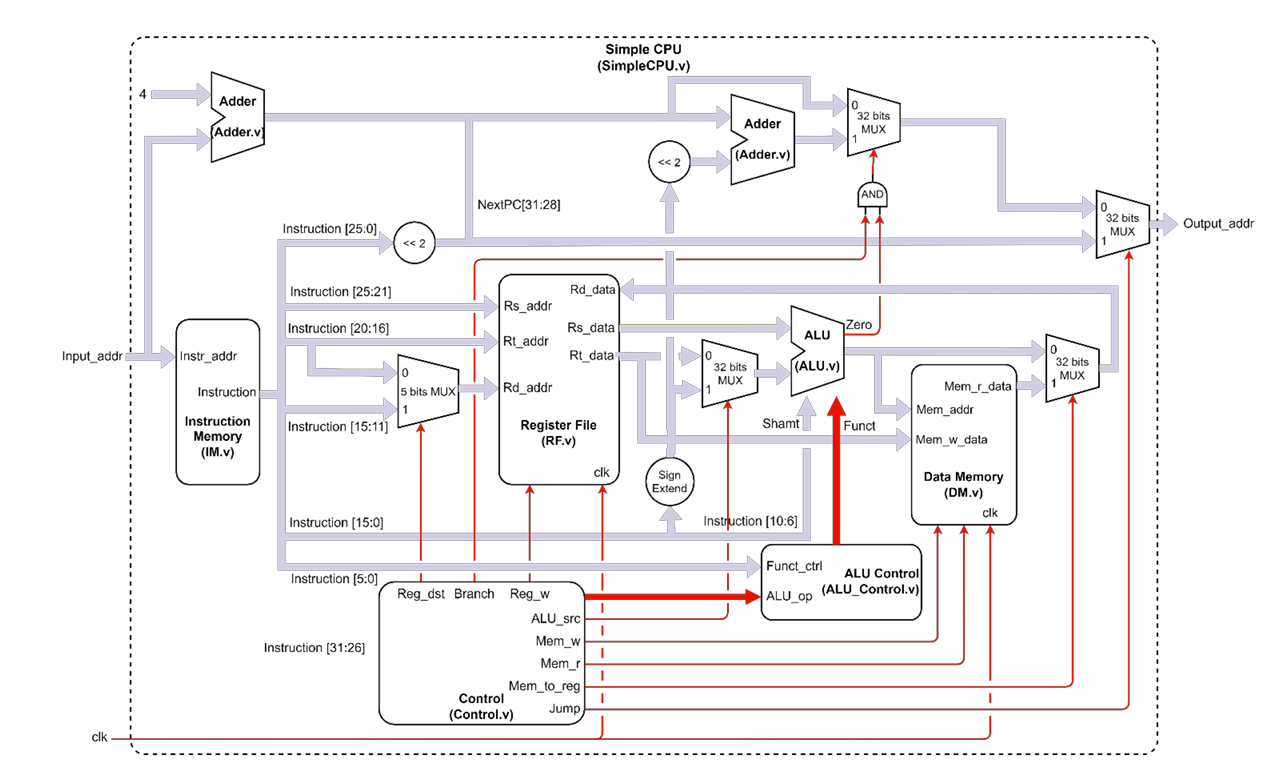
The DM module consists of a memory array (DataMem) and associated logic for read and write operations.

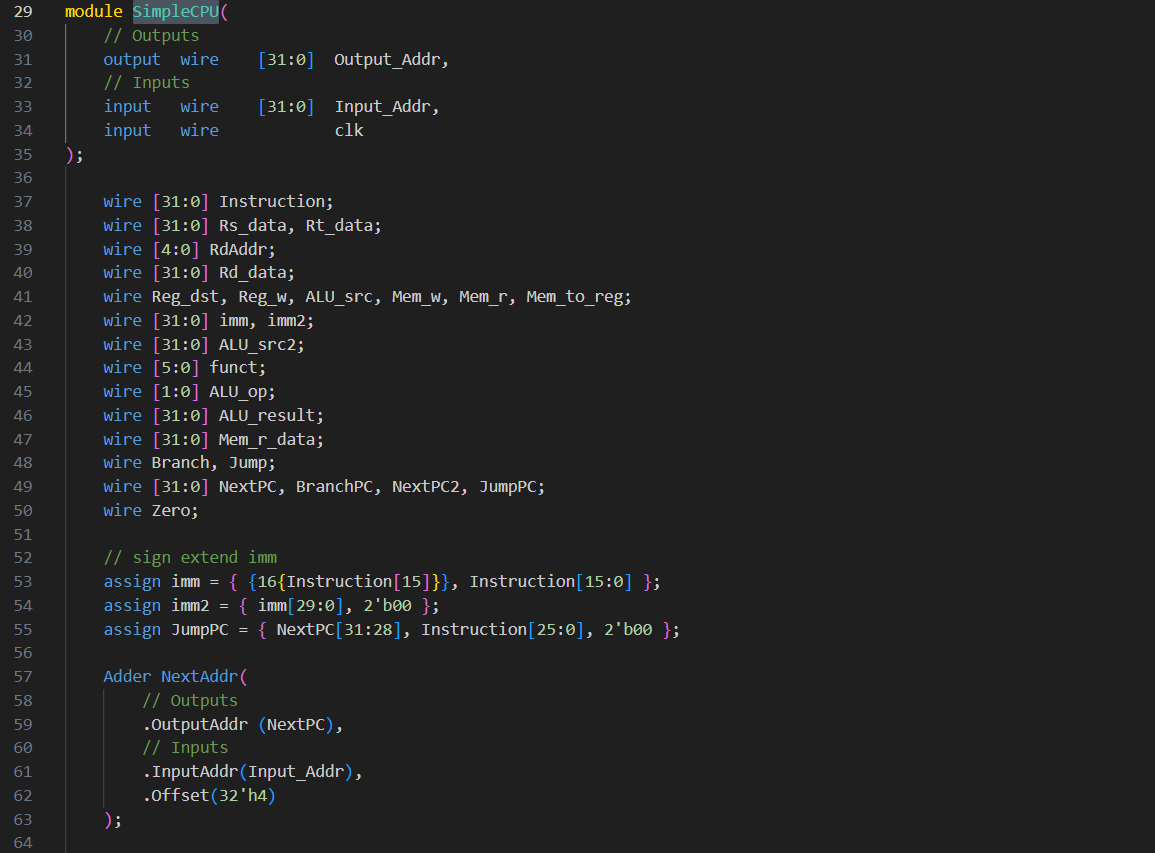
### Process

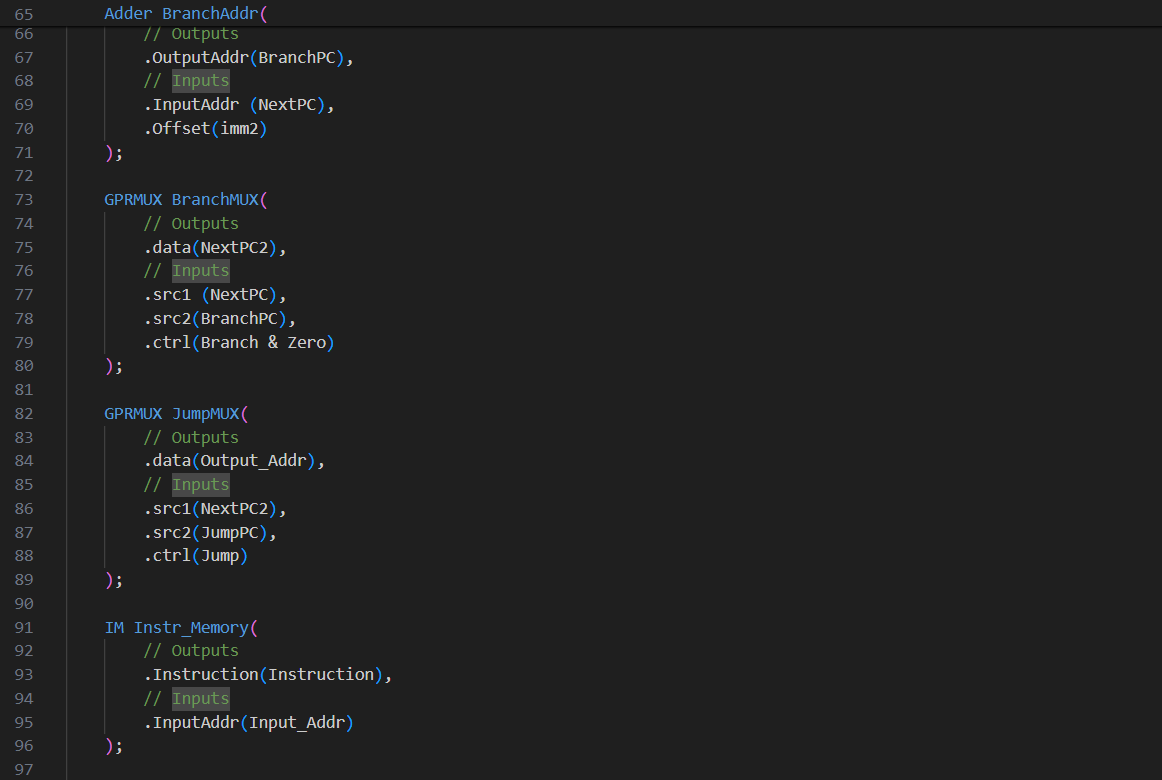
* Write Operation: On the falling edge of the clock, if the write enable signal (Mem\_w) is asserted, write the provided data into memory at the specified address.
* Read Operation: When the read enable signal (Mem\_r) is asserted, retrieve data from memory at the specified address and provide it as output (Mem\_r\_data).

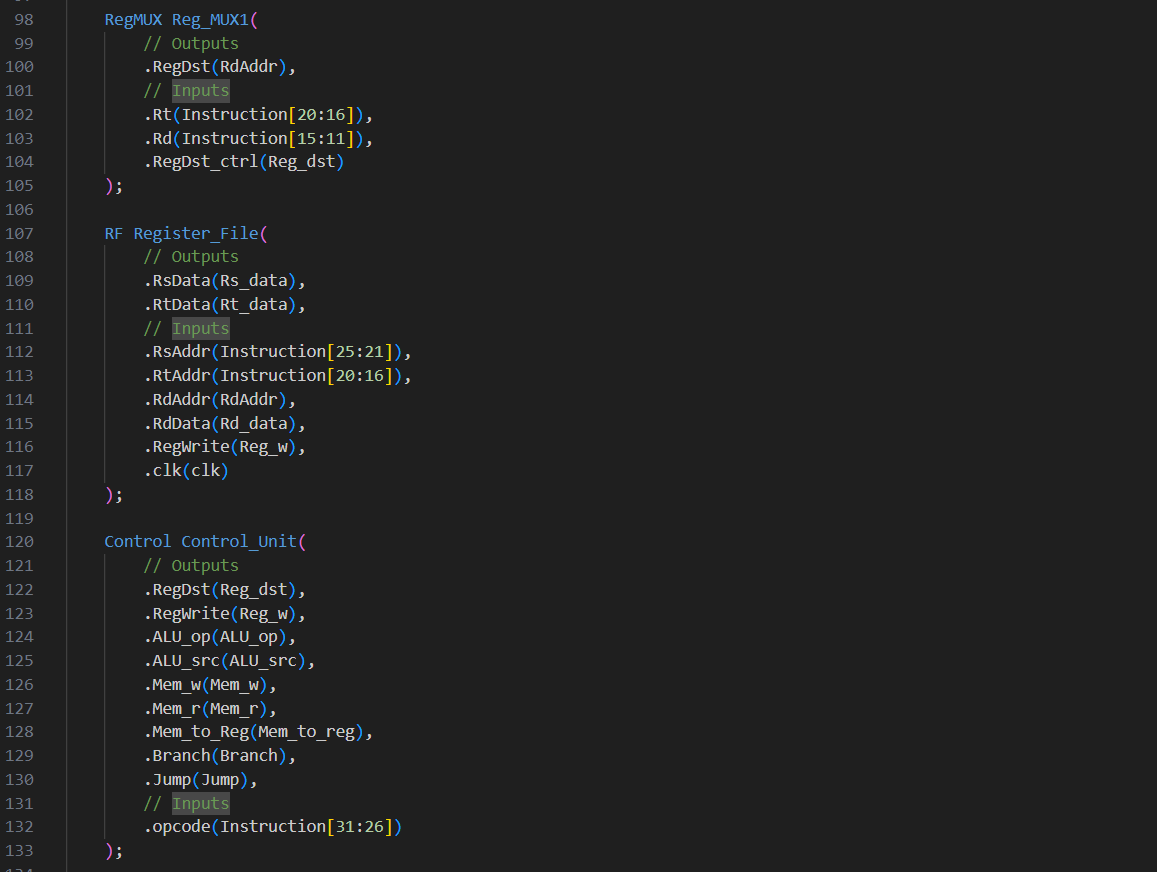
# Part3

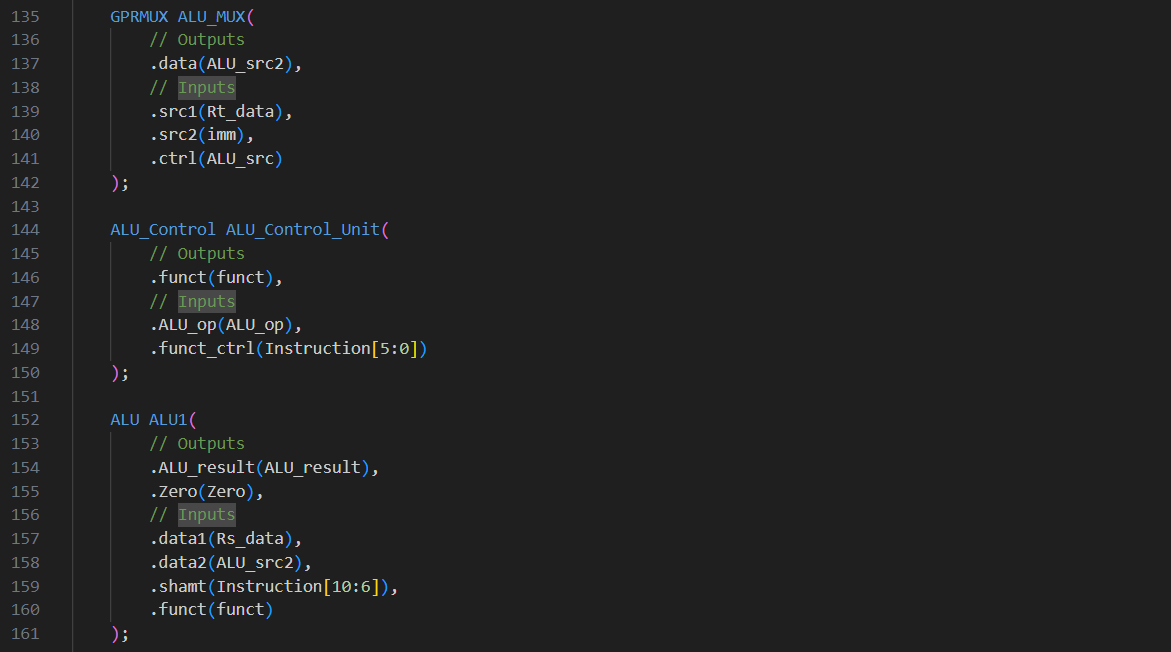
## SimpleCPU

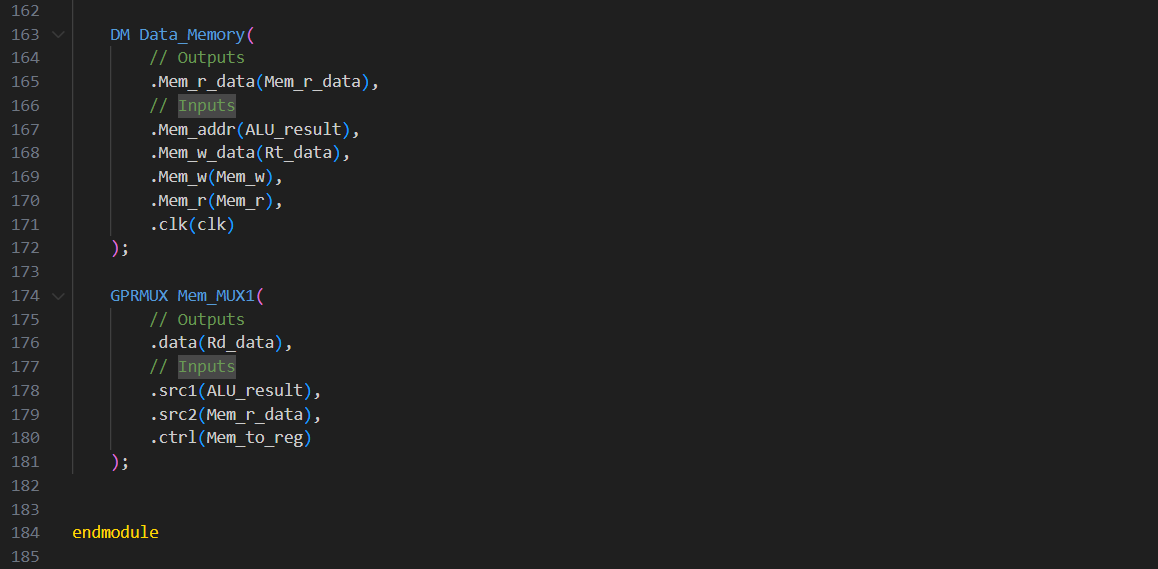












### Description

The Verilog module SimpleCPU represents a basic single-cycle CPU design. It consists of various components such as instruction memory, register file, ALU (Arithmetic Logic Unit), data memory, control unit, and multiplexers. The CPU executes instructions fetched from memory, performs arithmetic and logical operations, accesses registers and memory, and controls the flow of instructions.

### Code Explanation

Line 29~35

Module Declaration:

Defines a Verilog module named SimpleCPU.

Specifies an output port Output\_Addr (32-bit) to provide the address of the next instruction to be executed.

Specifies input ports for:

Input\_Addr (32-bit): Address input for fetching instructions from memory.

clk: Clock signal.

Line 37~50

Internal Signals:

These declarations specify various wires for managing internal signals and data paths within the CPU.

Line 53~55

imm for I-type operation. imm2 for branching. JumpPC is J’s address.

Line 57~181

Component Instantiation:

The module instantiates various components required for CPU operation, including instruction memory (Instr\_Memory), adders (NextAddr and BranchAddr), multiplexers (BranchMUX and JumpMUX), register file (Register\_File), ALU (ALU1), data memory (Data\_Memory), and control unit (Control\_Unit).

Each component is instantiated with appropriate inputs and outputs, and connections are made between these components to establish the CPU datapath and control logic.

### IO

Inputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| Input\_Addr (32-bit) | Address input for fetching instructions from memory. |
| clk (1-bit) | Clock signal. |

Outputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| Output\_Addr (32-bit) | Address of the next instruction to be executed. |

### Wires

|  |  |
| --- | --- |
| Wire Name | Description |
| Instruction (32-bit) | Instruction fetched from IM using Input\_Addr. |
| Rs\_data, Rt\_data (32-bit each) | Data read from the register file corresponding to the source registers specified in the instruction. They are used in ALU operations or data transfers. |
| RdAddr (5-bit) | Address of the destination register where the result of an operation will be stored. |
| Rd\_data (32-bit) | Data read from the register file corresponding to the destination register specified in the instruction. |
| Reg\_dst, Reg\_w, ALU\_src, Mem\_w, Mem\_r, Mem\_to\_reg (1-bit each) | Flags to control the whole cpu. |
| imm, imm2 (32-bit each) | Immediate value extracted from the instruction. imm is sign-extended to 32 bits, while imm2 is sign-extended and shifted left by two bits, used for branch instructions. |
| ALU\_src2 (32-bit) | The second operand for ALU operations. |
| funct (6-bit), ALU\_op (2-bit) | Control signals for ALU operations. |
| ALU\_result (32-bit) | Result of ALU operation. |
| Mem\_r\_data | Data read from the data memory. |
| Branch, Jump (1-bit each) | Whether a branch or jump instruction will be being executed. |
| NextPC, BranchPC, NextPC2, JumpPC (32-bit each) | These wires hold the addresses of the next instruction, branch target, next instruction after a branch, and jump target, respectively. |
| Zero (1-bit) | Indicates whether the result of an ALU operation is zero, used for branch instructions to determine whether to take the branch or not. |

### Components

|  |  |
| --- | --- |
| Component Name | Description |
| IM | Stores and provides instructions to the CPU. |
| Adder | Compute addresses for the next instruction and branch targets. |
| GPRMUX, RegMUX | Select between different data sources based on control signals. |
| RF | Stores and provides data from registers. |
| ALU | Performs arithmetic and logical operations. |
| DM | Stores and retrieves data values. |
| Control | Generates control signals based on instruction opcode. |

### Process

* Instruction Fetch: Fetches instructions from memory based on the provided address.
* Decode and Control: Decodes the fetched instruction and generates control signals.
* Register and Memory Access: Accesses registers and memory based on the decoded instruction.
* ALU Operation: Performs arithmetic or logical operations based on the instruction and operand values.
* Writeback: Writes results back to registers or memory.
* Branch and Jump Handling: Computes next instruction address based on branch or jump instructions.

## Adder

It’s the same as [Part1’s](#_Adder).

## GPRMUX

It’s the same as [Part2’s](#_GPRMUX).

## IM

It’s the same as [Part1’s](#_IM).

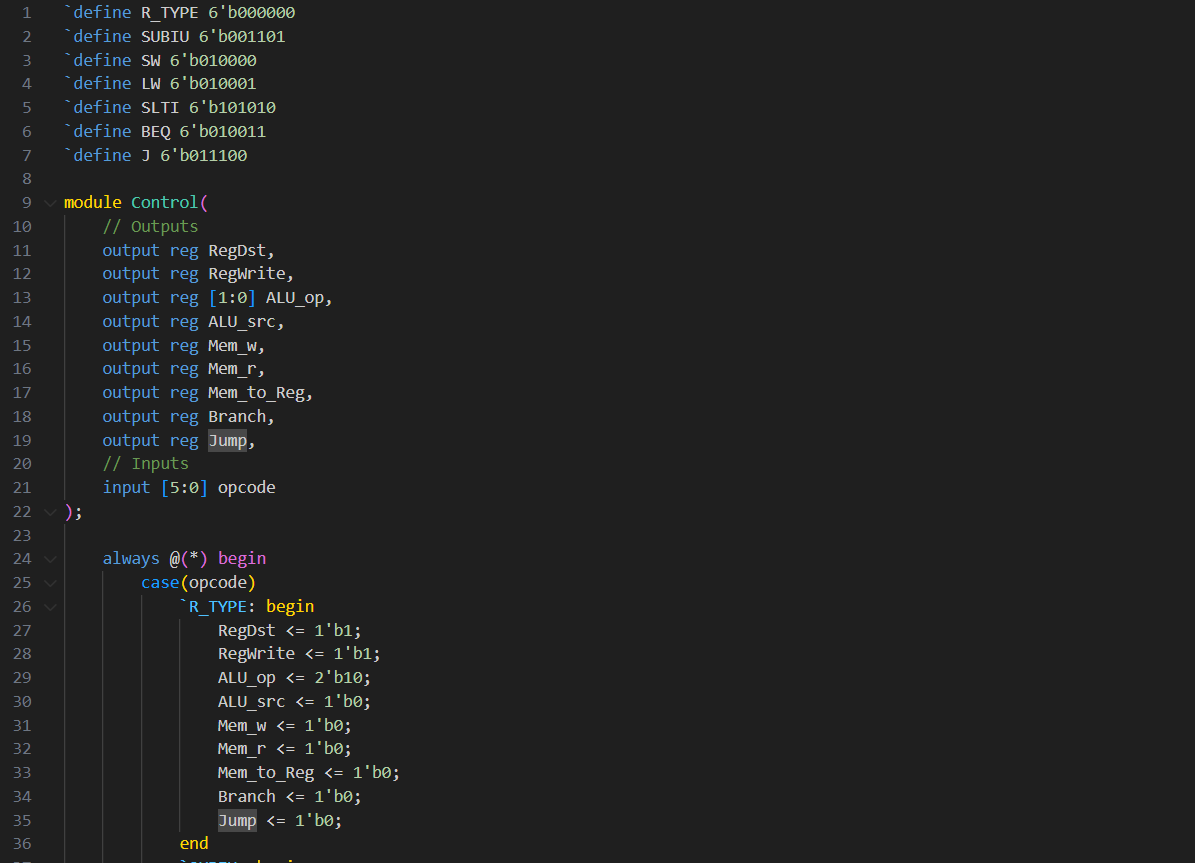
## RegMUX

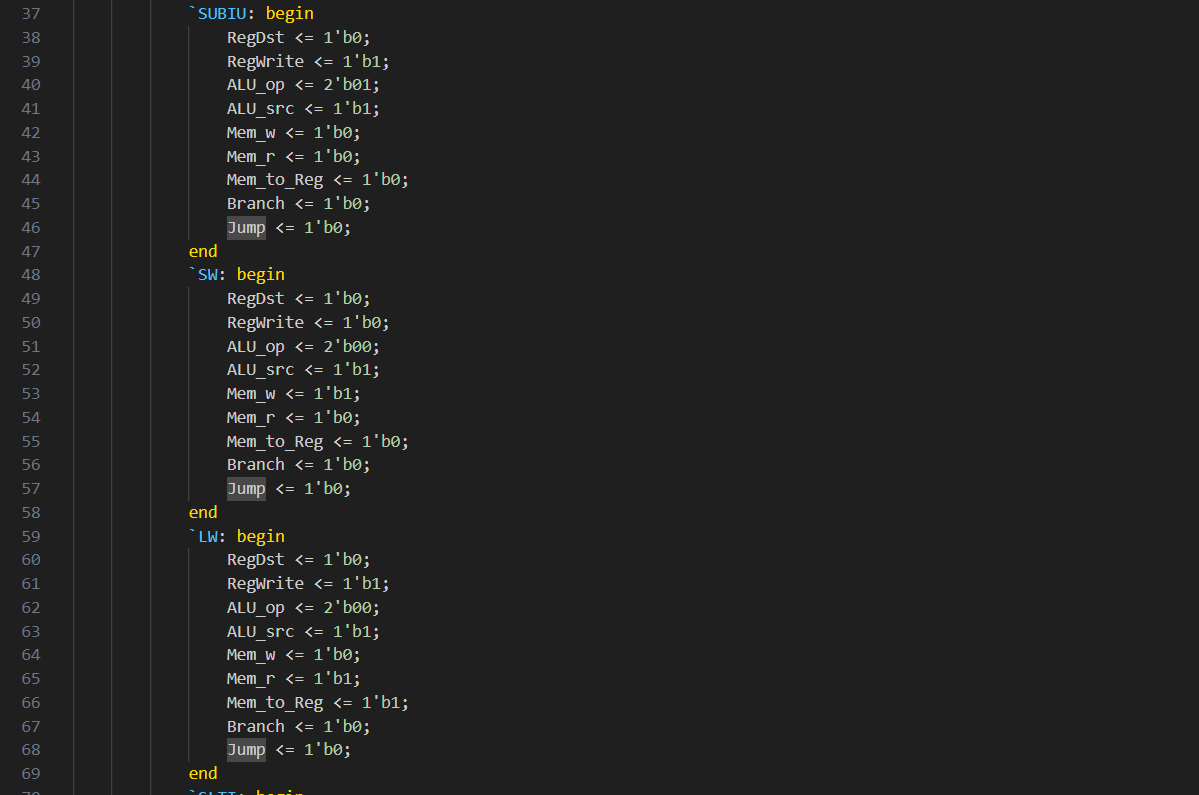
It’s the same as [Part2’s](#_RegMUX).

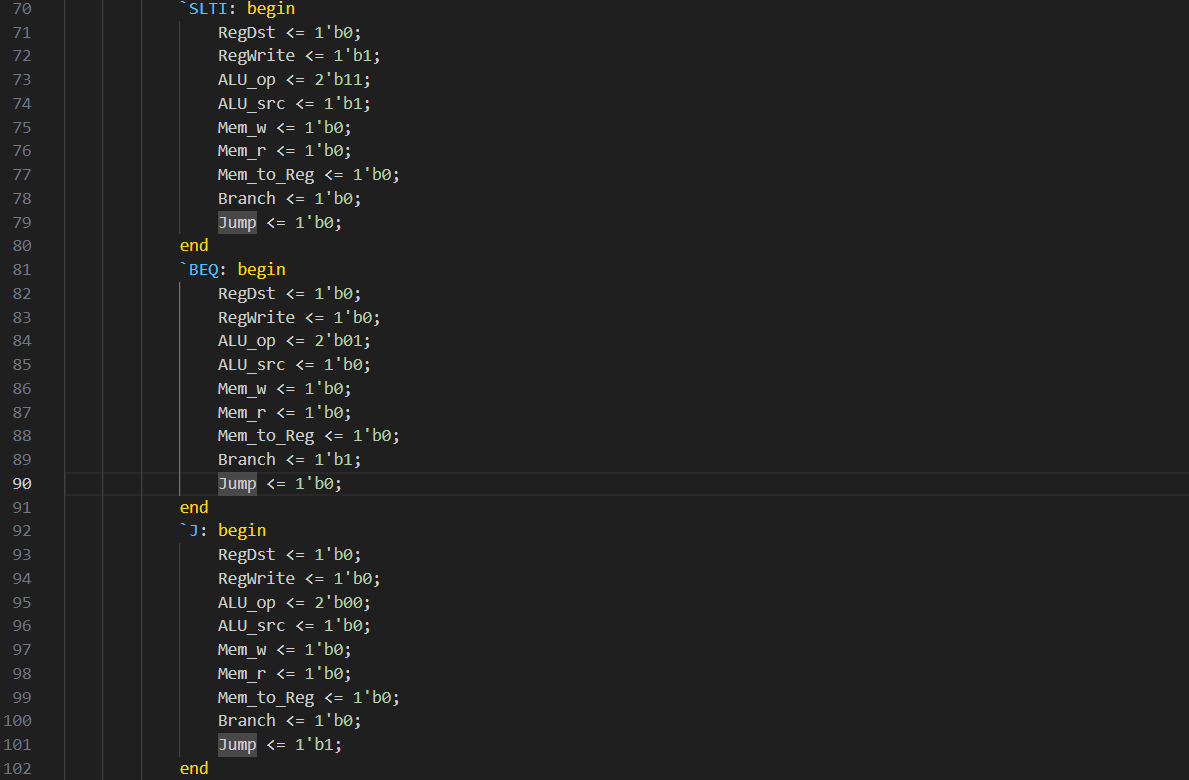
## RF

It’s the same as [Part1’s](#_RF).

## Control









### Description

The Control module serves as the control unit of a CPU, responsible for generating control signals based on the opcode of an instruction. It determines how the CPU should process the instruction, including register operations, ALU operations, and memory access.

### Code Explanation

Line 1~7

Opcode table.

Line 9~22

Module Declaration:

Defines a Verilog module named Control.

Specifies output ports for various control signals:

* RegDst: Register destination control signal.
* RegWrite: Register write enable control signal.
* ALU\_op: ALU operation control signal.
* ALU\_src: ALU source control signal.
* Mem\_w: Memory write enable control signal.
* Mem\_r: Memory read enable control signal.
* Mem\_to\_Reg: Memory to register control signal.
* Branch, Jump: Control signal for branching.

Specifies an input port opcode (6-bit) for the opcode of the instruction.

Line 24~113

Control Signal Generation:

Utilizes an always block triggered whenever the inputs change (@(\*)).

Uses a case statement to select control signals based on the opcode of the instruction.

Generates specific control signals for different types of instructions, including R-type, immediate arithmetic, and memory access instructions. If not recognized, it’ll use default condition.

### IO

Inputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| opcode (6-bit) | Opcode of the instruction. |

Outputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| RegDst (1-bit) | Register destination control signal. |
| RegWrite (1-bit) | Register write enable control signal. |
| ALU\_op (2-bit) | ALU operation control signal. |
| ALU\_src (1-bit) | ALU source control signal. |
| Mem\_w (1-bit) | Memory write enable control signal. |
| Mem\_r (1-bit) | Memory read enable control signal. |
| Mem\_to\_Reg (1-bit) | Memory to register control signal. |
| Branch, Jump (1-bit each) | Flags for branching |

### Wires

No internal wires are explicitly declared in this module. Control signals are directly assigned based on the opcode using the always block.

### Components

The Control module acts as a standalone component responsible for generating control signals based on the opcode of an instruction.

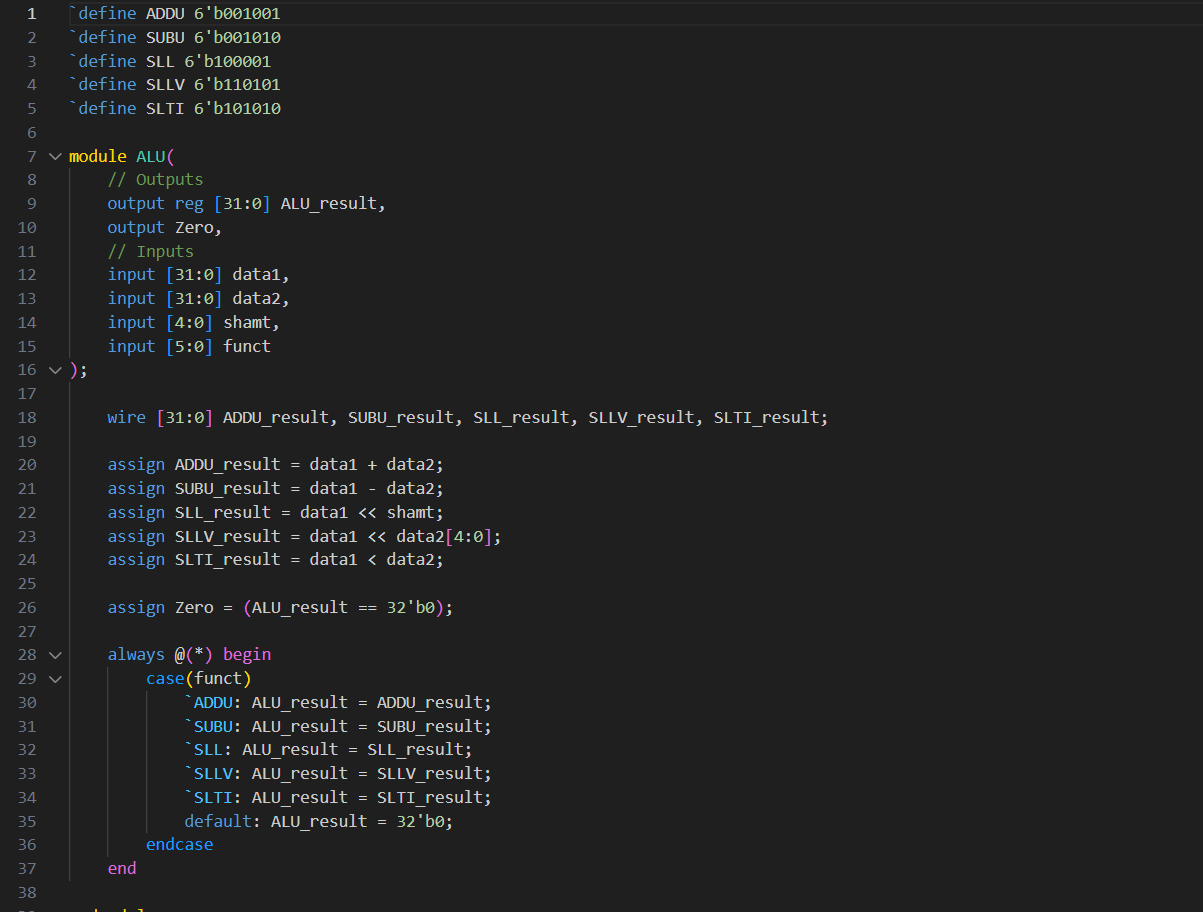
### Process

* Opcode Interpretation: Receives the opcode of the instruction.
* Control Signal Selection: Determines the appropriate control signals based on the opcode using a case statement.
* Control Signal Assignment: Assigns the selected control signals to the output ports.

## ALU\_Control

It’s the same as [Part2’s](#_ALU_Control).

## ALU



### Description

The Verilog module ALU represents an Arithmetic Logic Unit (ALU) designed to perform arithmetic and logical operations on two input operands. It supports operations such as addition, subtraction, left logical shift, and set less than (SLT) comparison.

### Code Explanation

Line 1~5

LUT.

Line 7~16

Module Declaration:

Defines a Verilog module named ALU.

Specifies output ports:

ALU\_result (32-bit): Result of the ALU operation.

Zero: Output indicating whether the ALU result is zero.

Specifies input ports for:

data1, data2 (32-bit): Input operands to the ALU.

shamt (5-bit): Shift amount for logical shift operations.

funct (6-bit): Function code determining the operation to be performed by the ALU.

Line 18~24

Operation Execution:

Intermediate results are computed based on the input operands and specified operations:

ADDU\_result: Addition of data1 and data2.

SUBU\_result: Subtraction of data2 from data1.

SLL\_result: Left logical shift of data1 by shamt.

SLLV\_result: Left logical shift of data1 by the lower 5 bits of data2.

SLTI\_result: Comparison of data1 and data2 for the SLT operation.

Line 26

Zero Flag:

The Zero output is assigned based on whether the ALU\_result is equal to zero.

Line 28~37

ALU Operation:

The ALU\_result is determined based on the function code (funct):

If funct matches predefined operation codes, the corresponding intermediate result is assigned to ALU\_result.

If no match is found, ALU\_result is set to zero.

### IO

Inputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| data1, data2 (32-bit) | Input operands to the ALU. |
| shamt (5-bit) | Shift amount for logical shift operations. |
| funct (6-bit) | Function code determining the operation to be performed by the ALU. |

Outputs:

|  |  |
| --- | --- |
| Wire Name | Description |
| ALU\_result (32-bit) | Result of the ALU operation. |
| Zero (1-bit) | Output indicating whether the ALU result is zero. |

### Wires

ADDU\_result, SUBU\_result, SLL\_result, SLLV\_result, SLTI\_result: Wires for storing intermediate computation results of various ALU operations.

### Components

This module does not instantiate any external components. It internally computes the ALU operations based on input operands and function codes.

### Process

* Input Processing: Receive input operands (data1, data2), shift amount (shamt), and function code (funct).
* Operation Execution: Perform the specified ALU operation based on the function code and input operands.
* Result Assignment: Assign the computed result (ALU\_result) to the output port.
* Zero Flag Evaluation: Determine if the result is zero and assign the appropriate value to the Zero output.

## DM

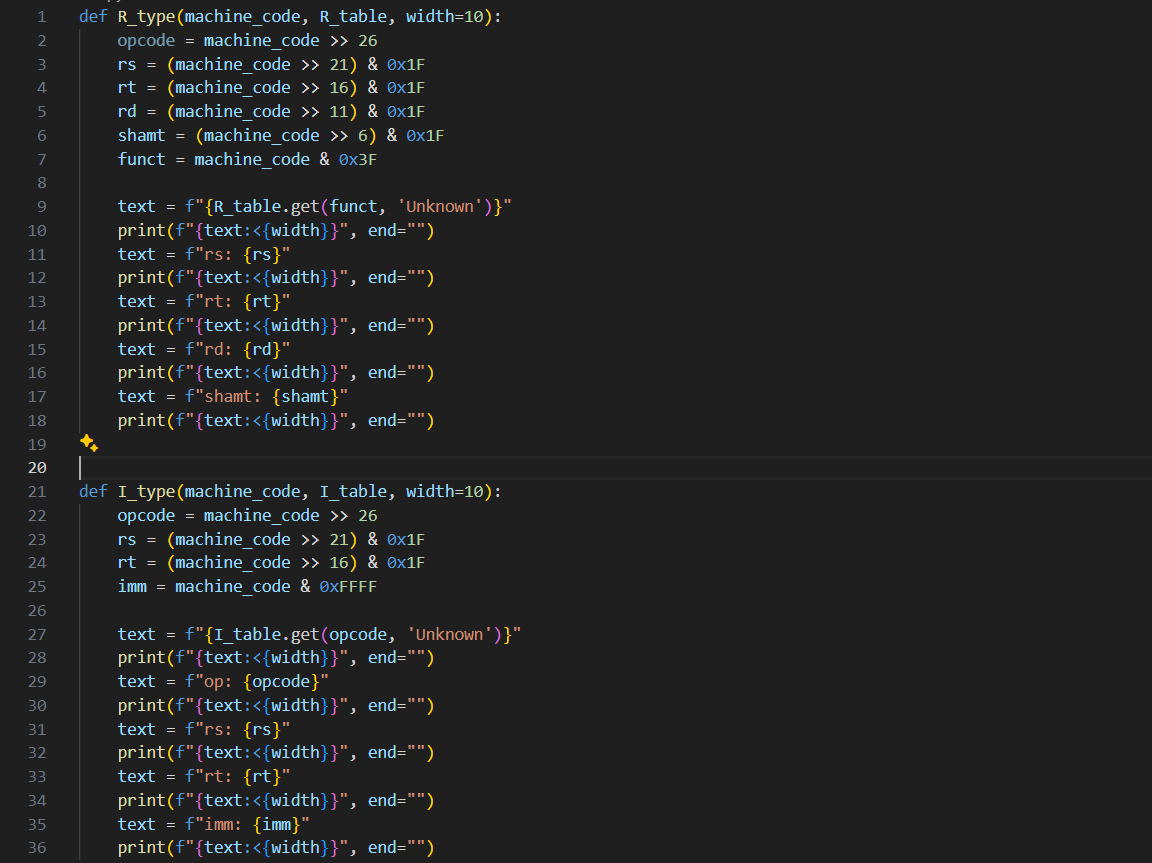
It’s the same as [Part2’s](#_DM).

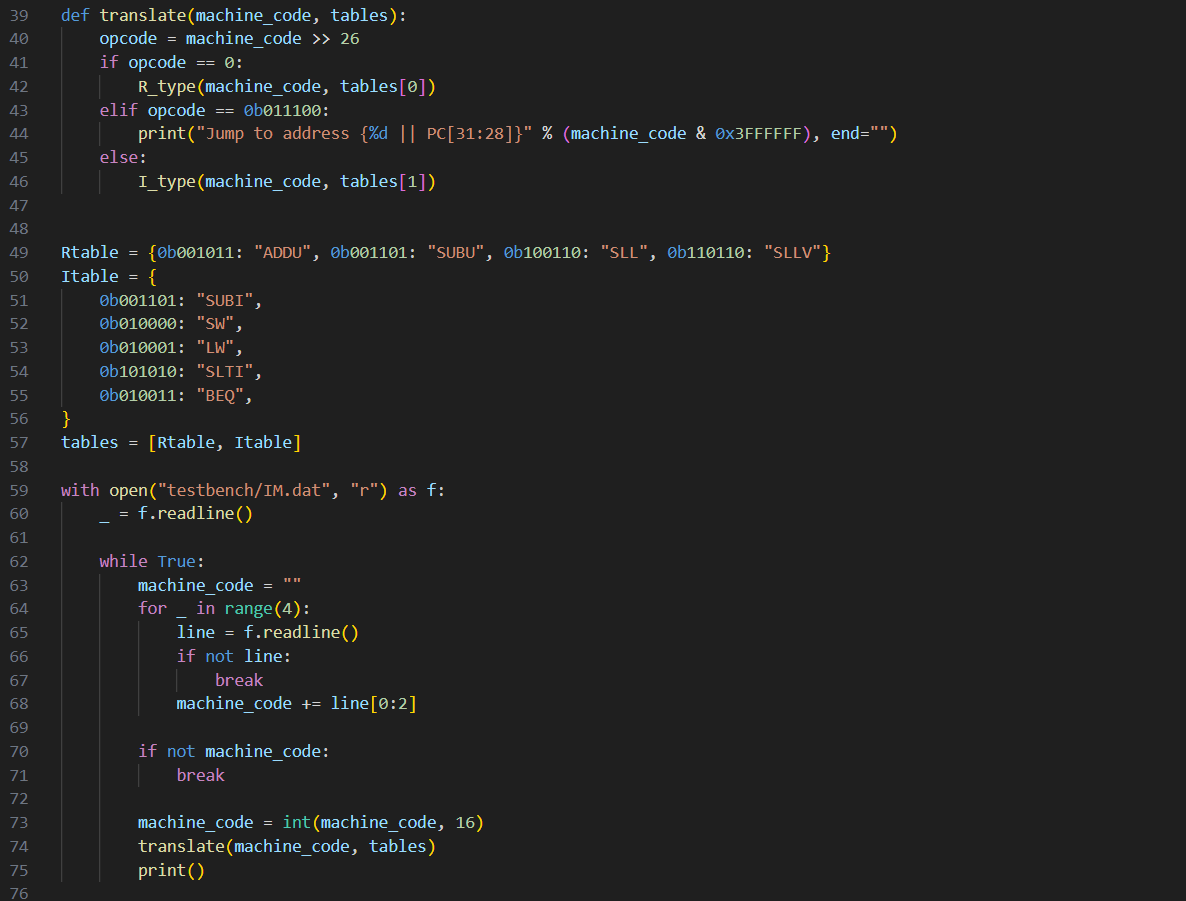
# Execution Results

All the registers’ data are using the data from testbench. There is no change in those files.

## Helper programs

### Translator.py





#### Description

This Python script is a simple translator for MIPS machine code instructions. It translates the instructions into human-readable assembly mnemonics along with their corresponding fields, such as opcode, registers, and immediate values.

#### Conversion Results

Part1:

ADDU rs: 10 rt: 11 rd: 20 shamt: 0

SUBU rs: 12 rt: 13 rd: 21 shamt: 0

SLL rs: 14 rt: 15 rd: 24 shamt: 3

SLLV rs: 15 rt: 16 rd: 26 shamt: 6

Part2:

SUBI op: 13 rs: 10 rt: 25 imm: 9

SW op: 16 rs: 12 rt: 25 imm: 3

LW op: 17 rs: 12 rt: 27 imm: 3

SLTI op: 42 rs: 10 rt: 26 imm: 20

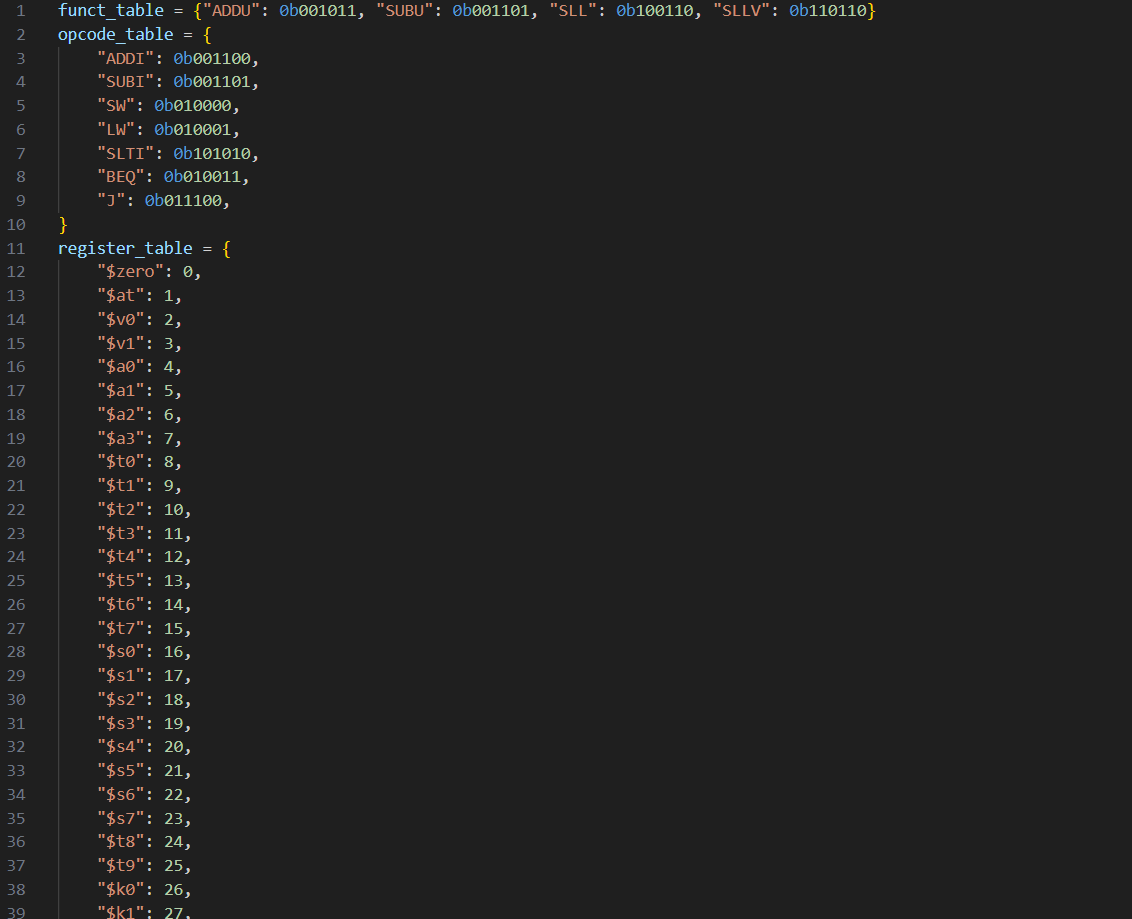
Part3:

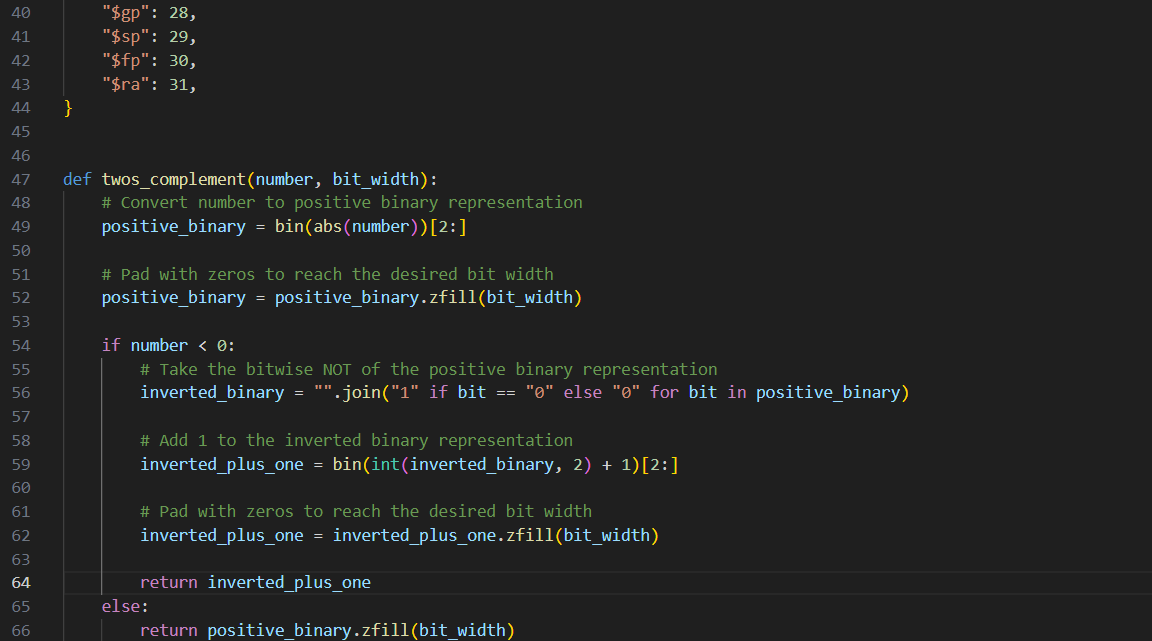
BEQ op: 19 rs: 0 rt: 19 imm: 30

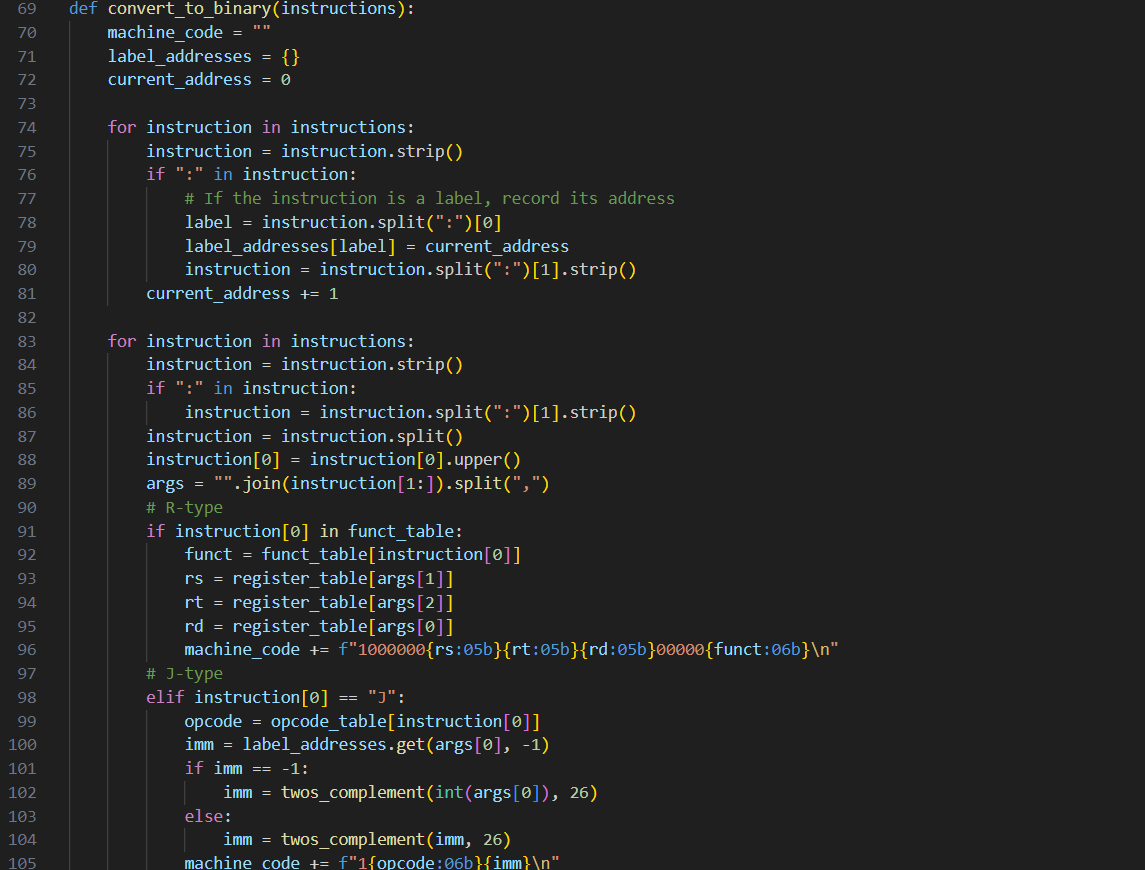
SUBU rs: 19 rt: 2 rd: 19 shamt: 0

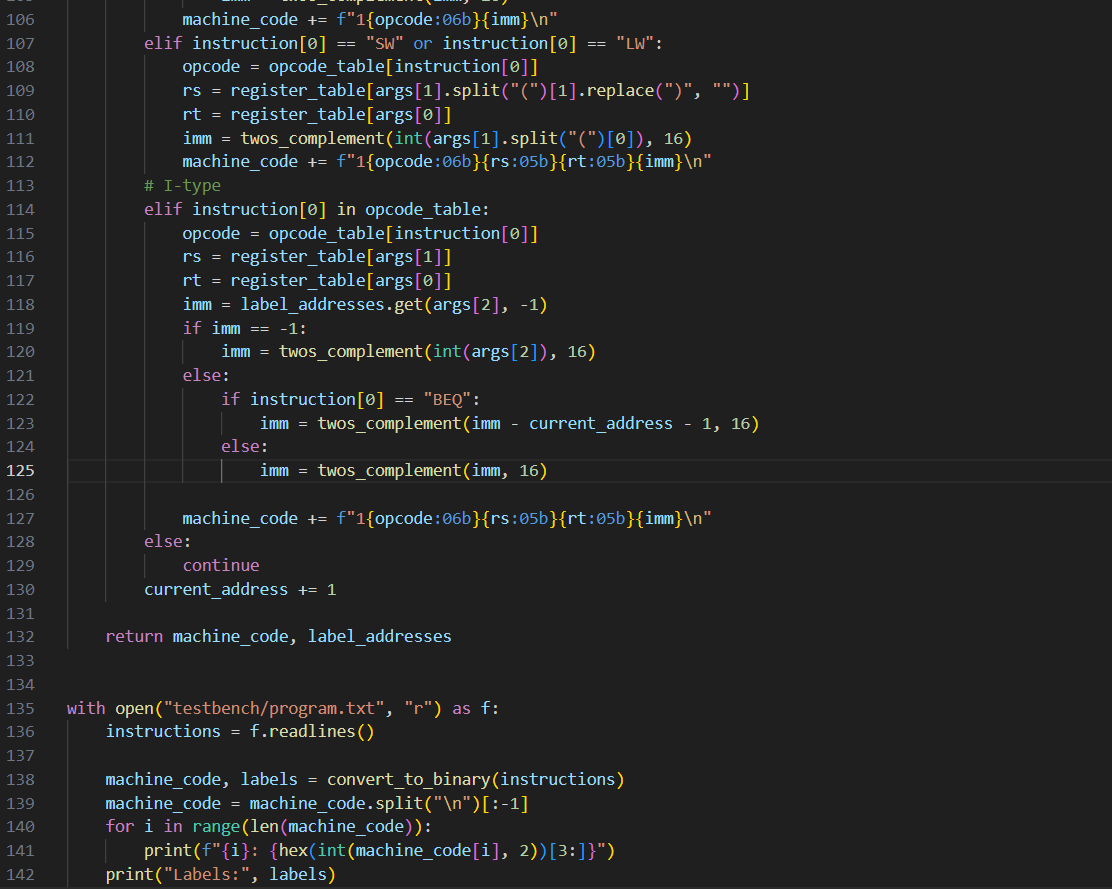
Jump to address {0 || PC[31:28]}

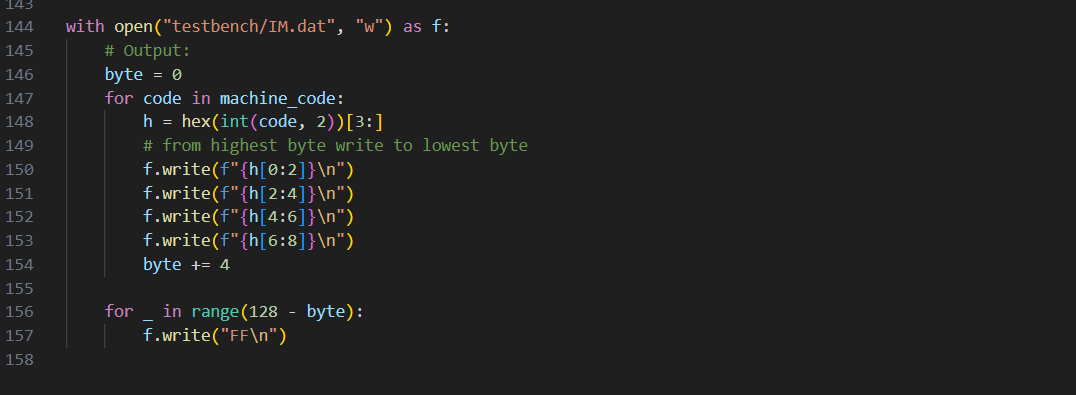
### Convertor.py







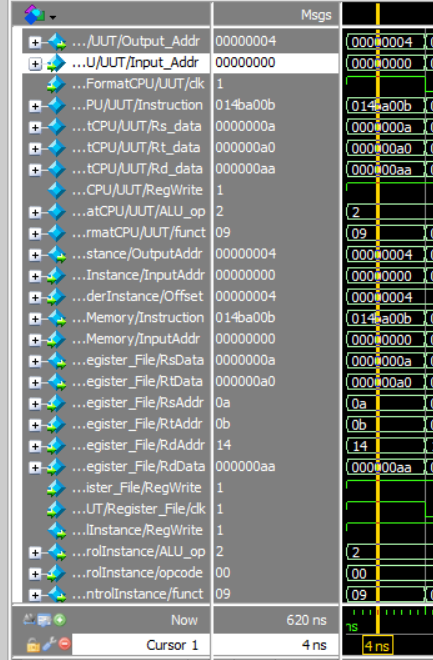


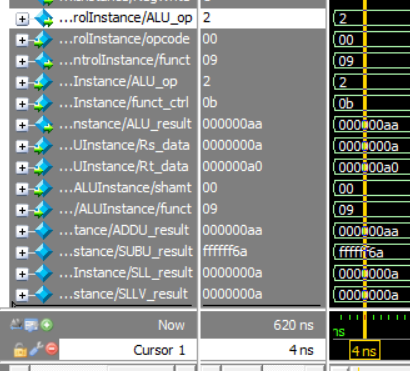


#### Description

This program can convert MIPS assembly into machine code required in this PA. This can compile labels.

## Part1





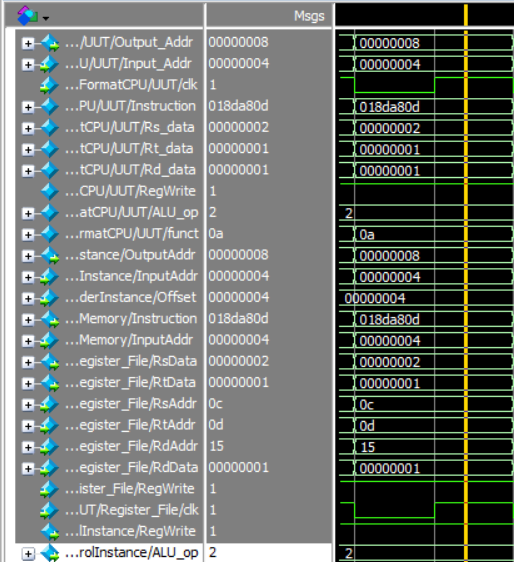
Descriptions:

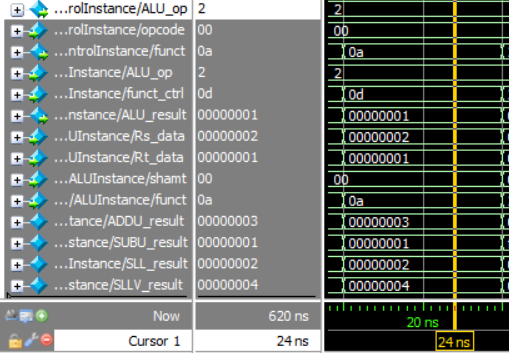
Cycle 1

* Input\_Addr is 0, Output\_Addr should be 0+4=4 which is correct.



* Instruction is read from IM in big-endian. 0x014ba00b is valid operation. For specific, it is ADDU $20, $10, $11.
* Rs\_data, Rt\_data are 0x0000\_000A and 0x0000\_00A0, respectively verified by looking up in the input file.
* Rd\_data is the ALU result. In this case, it’s rd=rs+rt=0x0000\_00AA which is correct.
* Because it’s R-type, RegWrite is high; ALU\_op is 2’b10 which is correct.
* UUT/funct is 0x09. This is generated from ALU\_Control to tell ALU to execute ADDU.
* The following 3 vars is IO of Adder to compute next PC addr. Offset is a constant, 32’h4.
* And then is the operation of fetching instruction from IM with address 0 and store to Instruction.
* Following is the result of fetching RsData, RtData from RF with address specified by RsAddr(10), RtAddr(11). This process also save RdData to RdAddr(20). The saving process will be done at negedge clk to avoid racing.
* ALU\_op=2, opcode=0 for R-type. funct\_ctrl is 0x0b specified for ADDU by looking up the PA2 pdf. This should set funct to 0x09 to tell ALU to do ADDU operation.
* The following vars are the IO of ALU. We can see that the ALU is actually generating 4 outputs which are ADDU, SUBU, SLL, SLLV, and I select the ADDU by funct(0x09).



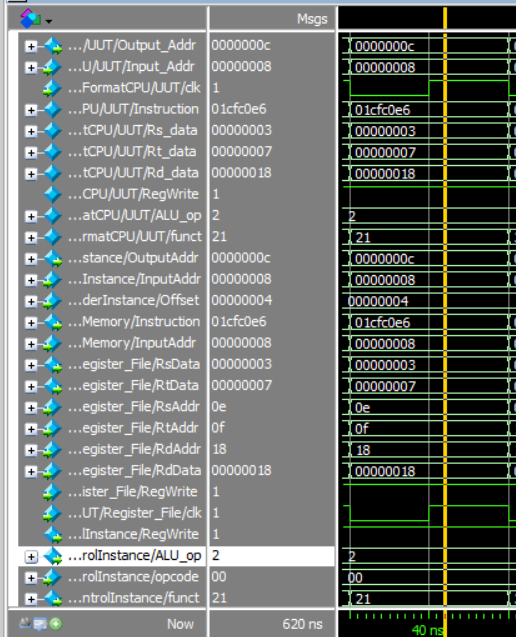


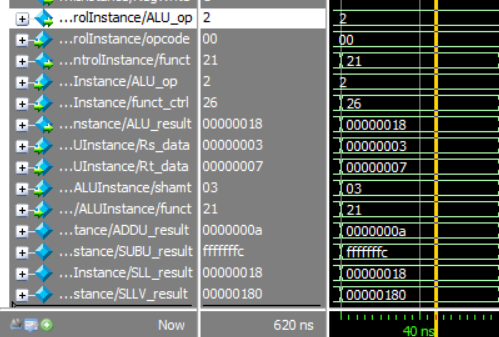
Cycle 2

* The next address is given after the negedge clk because all by component is combinational circuit we can see that the output is already generated as the instruction address is updated.



* This instruction is 0x018da80d which is SUBU $21, $12, $13. That is R[21]=R[12]-R[13]=2-1=1
* UUT/funct is 0x0a which is SUBU’s funct
* The other parts’ logic remain the same as Cycle 1.

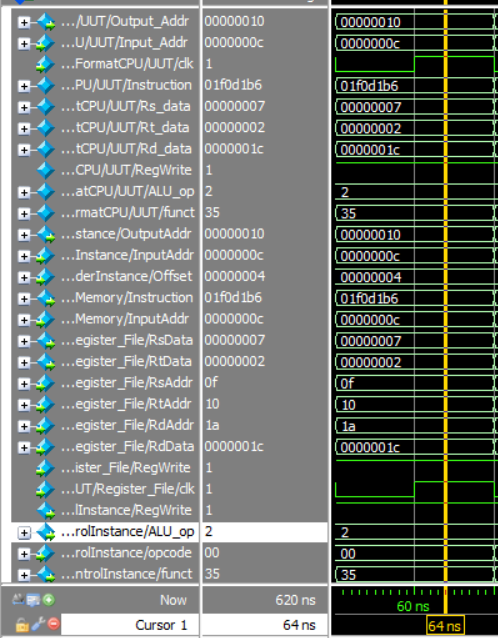




Cycle 3

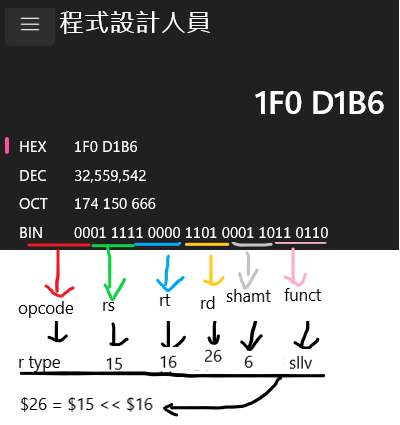


* The instruction is 0x01cfc0e6 which is SLL $24, $14, 3. R[24]=R[14]<<3 = 3 << 3 = 24 = 0x18. Rt’s address is also given which is 15 and the data(0x7) is fetched, but it didn’t used
* The remaining logic remains the same as Cycle 1.

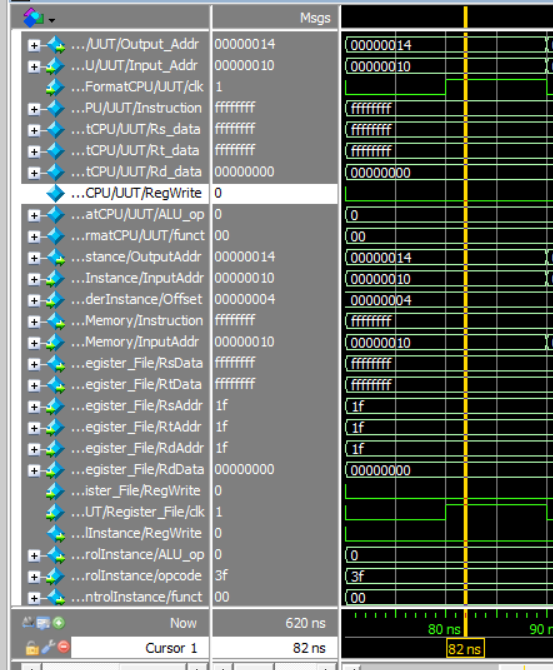




Cycle 4

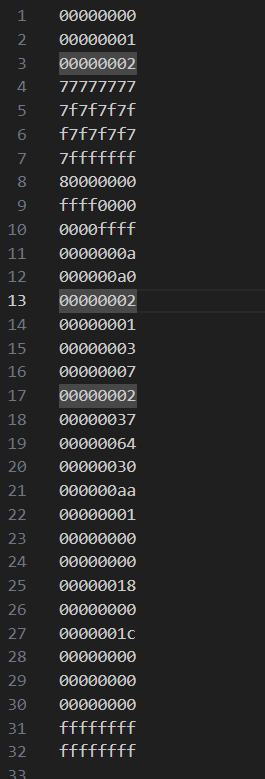


* The next instruction is 0x01f0d1b6 which is SLLV $26, $15, $16. R[26]=R[15]<<R[16][4:0]=7<<2=28=0x1c.
* The remaining logic is the same as Cycle 1.



The program will execute all the way to the end. To not incorrectly affect the output, I add default condition to Control. If the opcode is not recognized. It’ll use default operation which is do nothing.

After the operations above. This is the result:



$20 (line 21) = $10+$11 = 0xaa

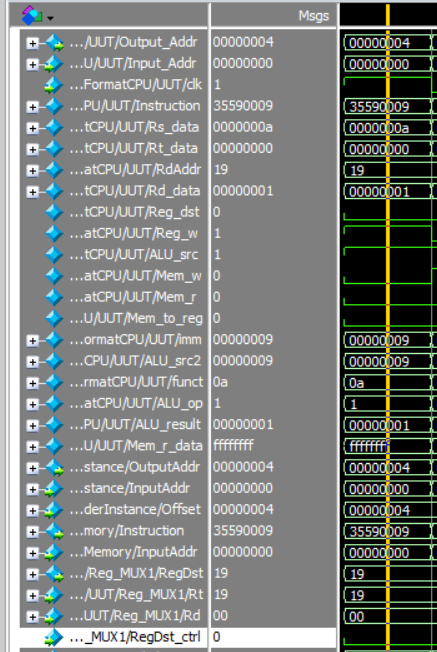
$21 (line 22) = $12-$13 = 0x01

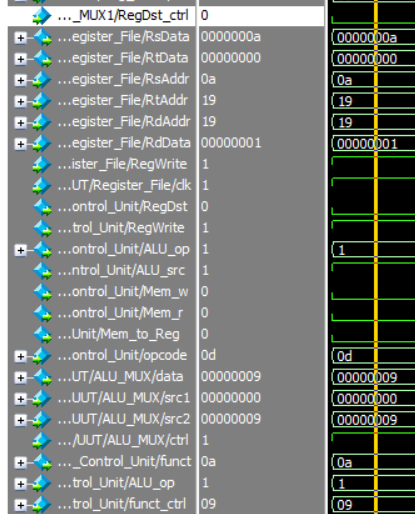
$24 (line 25) = $14<<3 = 0x18

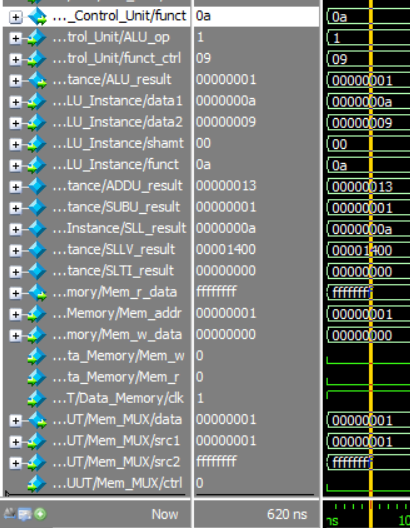
$26 (line 27) = $15<<$16[4:0] = 0x1c

And other register remain the same protected by the default condition of the control units.

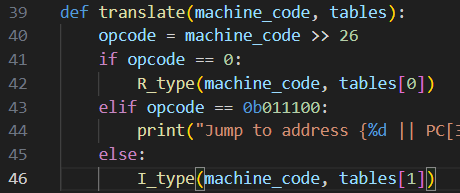
## Part2



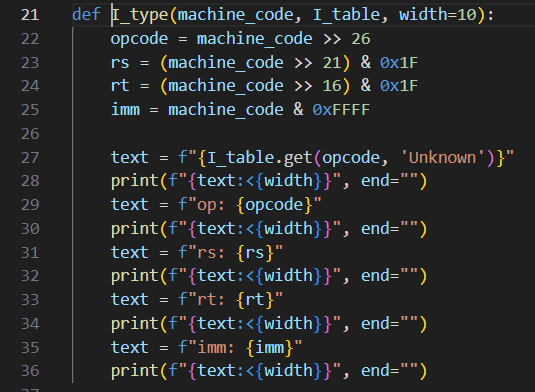


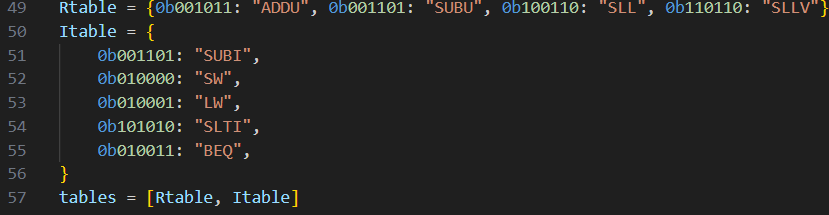


Cycle 1



1. opcode != 0 && opcode != 0b011100 -> I type



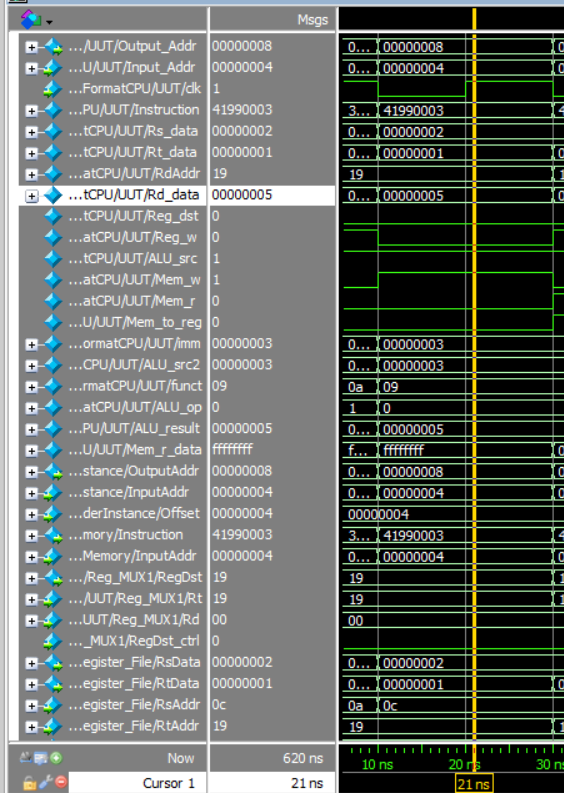


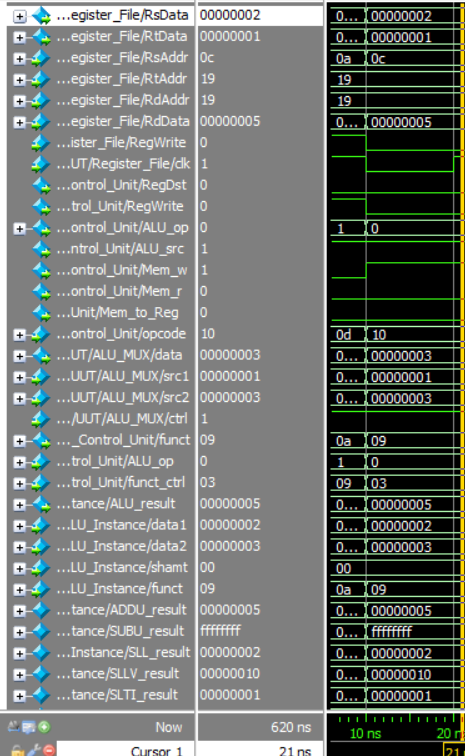


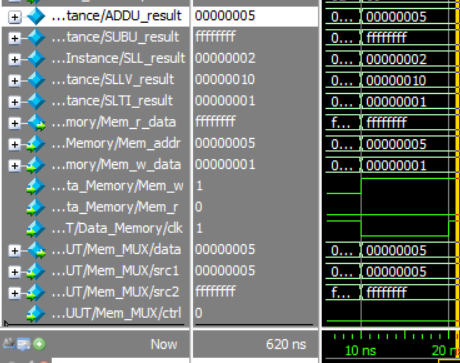
1. table[1] == Itable, find 0b001101 get SUBI; rs = 10; rd = 25; imm = 9.

Above is the flow of using [translator](#_Translator.py).

* The instruction fetched is 0x35590009 which is SUBI $25, $10, 9. R[25]=R[10]-0x9=0xa-0x9=0x1.
* Control now has more flag to control. In this case, Reg\_w, ALU\_src is set to ALU use imm as src2 and store the result back to R[RdAddr]
* UUT/funct = 0a to tell ALU to do subtraction.
* UUT/Mem\_r\_data is read from DM. Mem\_r will decide it should pass the MUX or not.
* There are many MUXs. The mechanisms are the same as the text book.
* The remaining logic is the same as previous.

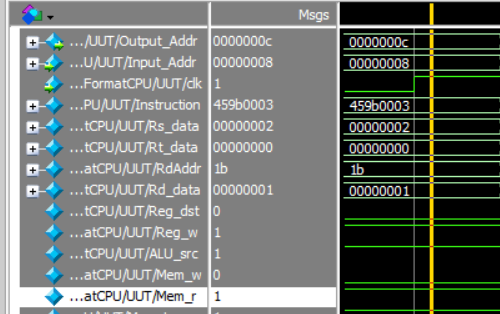






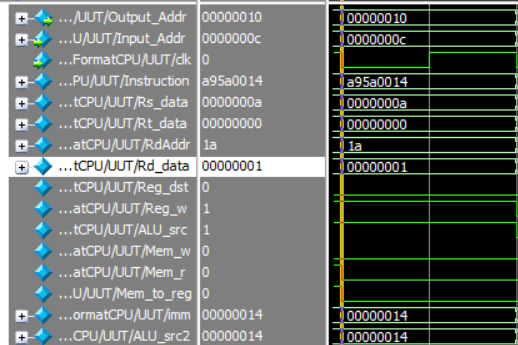
Cycle 2

* SW op: 16 rs: 12 rt: 25 imm: 3 which is saving the previous Rd into memory R[12]+3=5. So the DM’s {5,6,7,8} will be 0x00000001. How I know it’s sw with these parms is using [translator](#_Translator.py).
* The remaining logic is the same as previous.



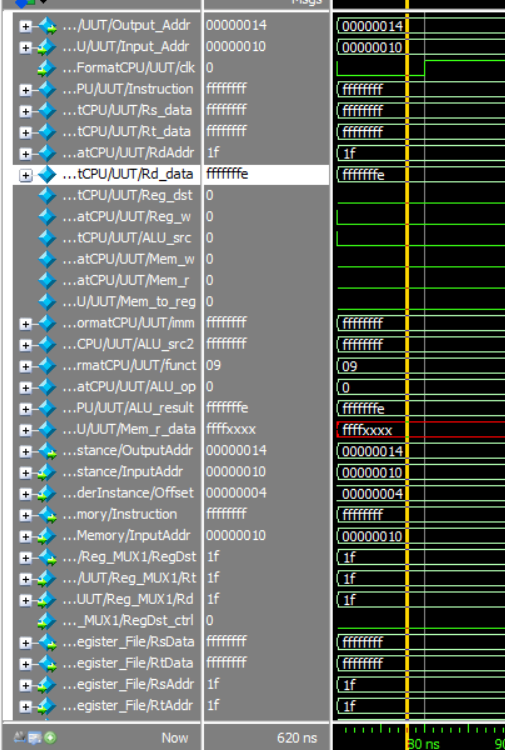
Cycle 3

* LW op: 17 rs: 12 rt: 27 imm: 3 which is loading the previously saved data into R[27]. Mem addr is R[12]+3. From [translator](#_Translator.py).
* The remaining part isn’t important.



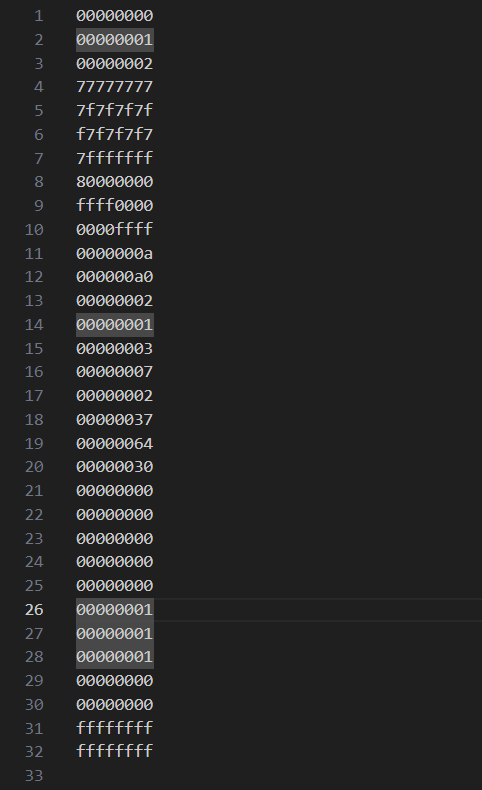
Cycle 4

* SLTI op: 42 rs: 10 rt: 26 imm: 20 which is if R[10]=10<20, R[26]=1 else R[26]=0.
* The remaining part isn’t important



This is the default condition. All the write flags are disabled to prevent modifying the data.

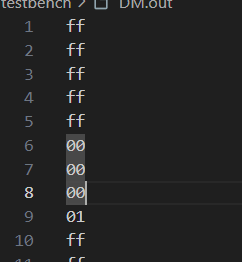
The corresponding output is below:



$25=1 by subiu

$27=1 by lw

$26=1 by slti



{5,6,7,8}(0 index) is set to 0x00000001 by sw

## Part3

The test program is this using [translator](#_Translator.py):

BEQ op: 19 rs: 0 rt: 19 imm: 30

SUBU rs: 19 rt: 2 rd: 19 shamt: 0

Jump to address {0 || PC[31:28]}

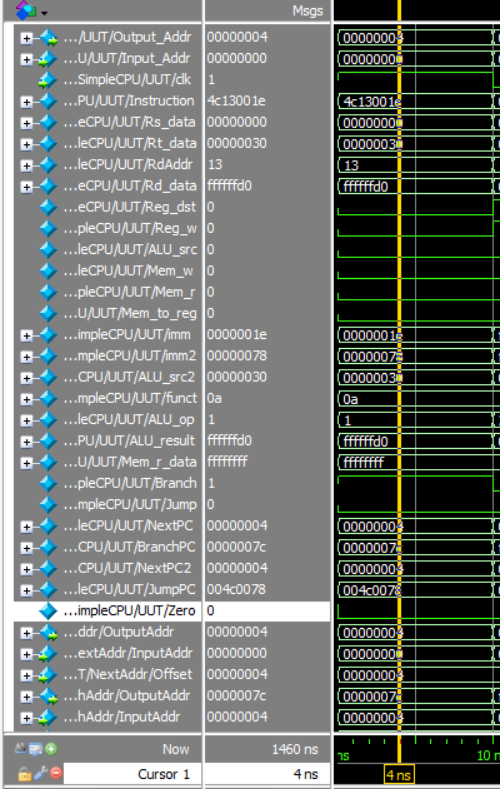
Which means

While (R[19] != R[0])

{

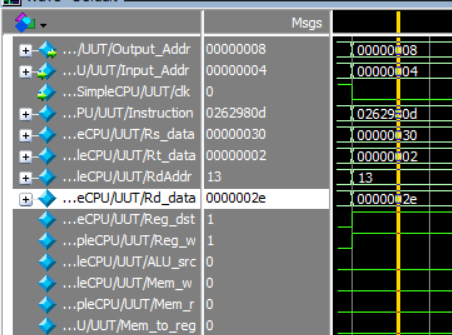
R[19] -= R[2]

}

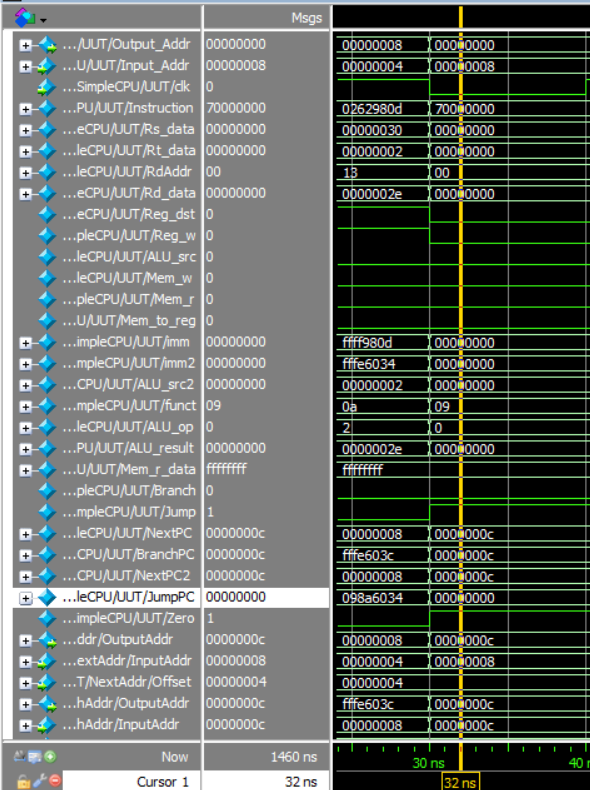


BEQ

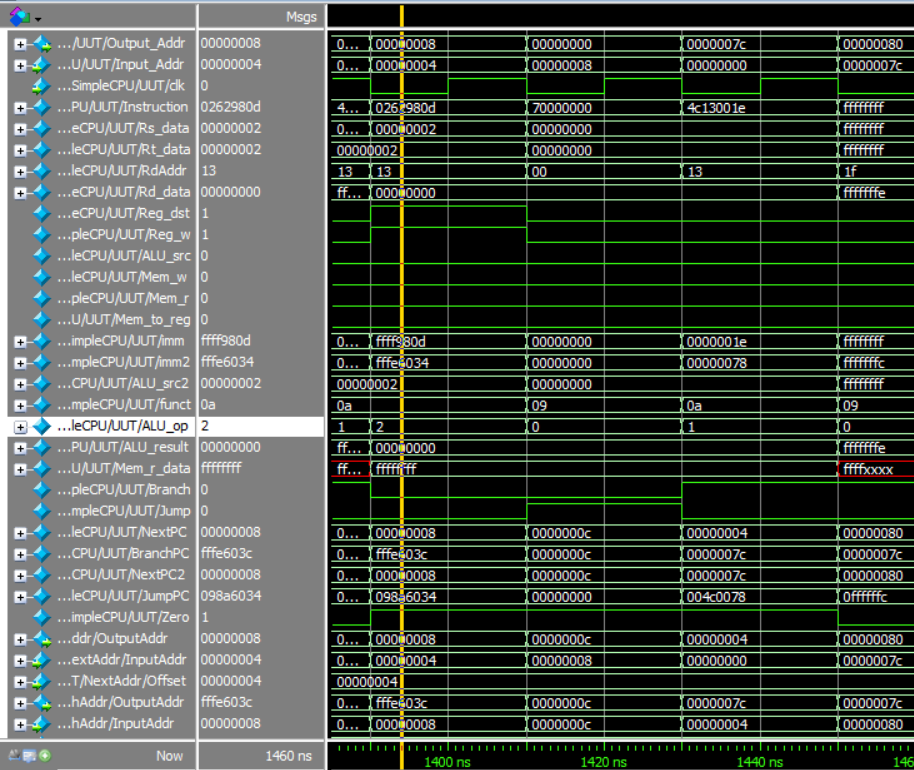
* Zero = 0, not branching



SUBU: 0x30-0x2 = 0x2e



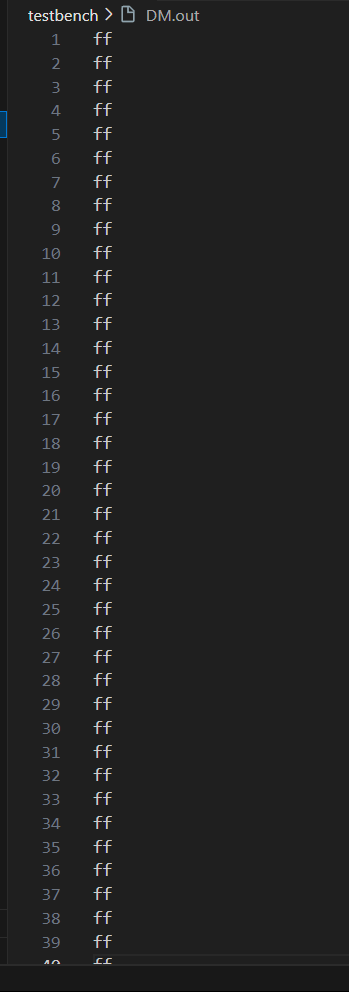
Jump to PC = 0



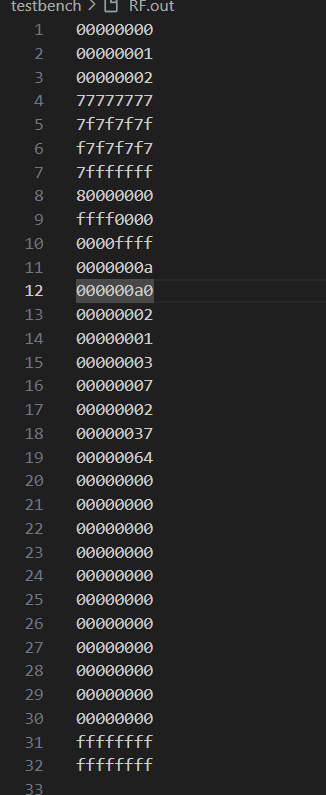
This the last iteration.

1. 2-2=0
2. Jump to 0
3. 0==0, branch to 30\*4+4=124 which is the last position of IM

The cycles passed: ((0x30/2) times \* 3 clk/time + 1) \* 20 ns/clk = 1460 ns. The data between 1450~1460 is the next cycles’ data, so the commands here are executed correctly.



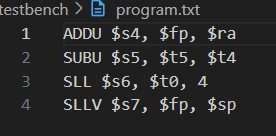
All 0xff in DM.out

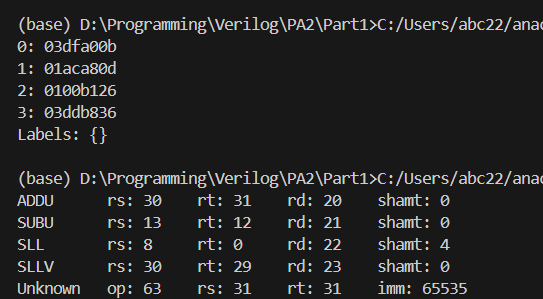


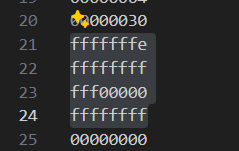
RF.out is just like using “cat RF.dat > RF.out”.

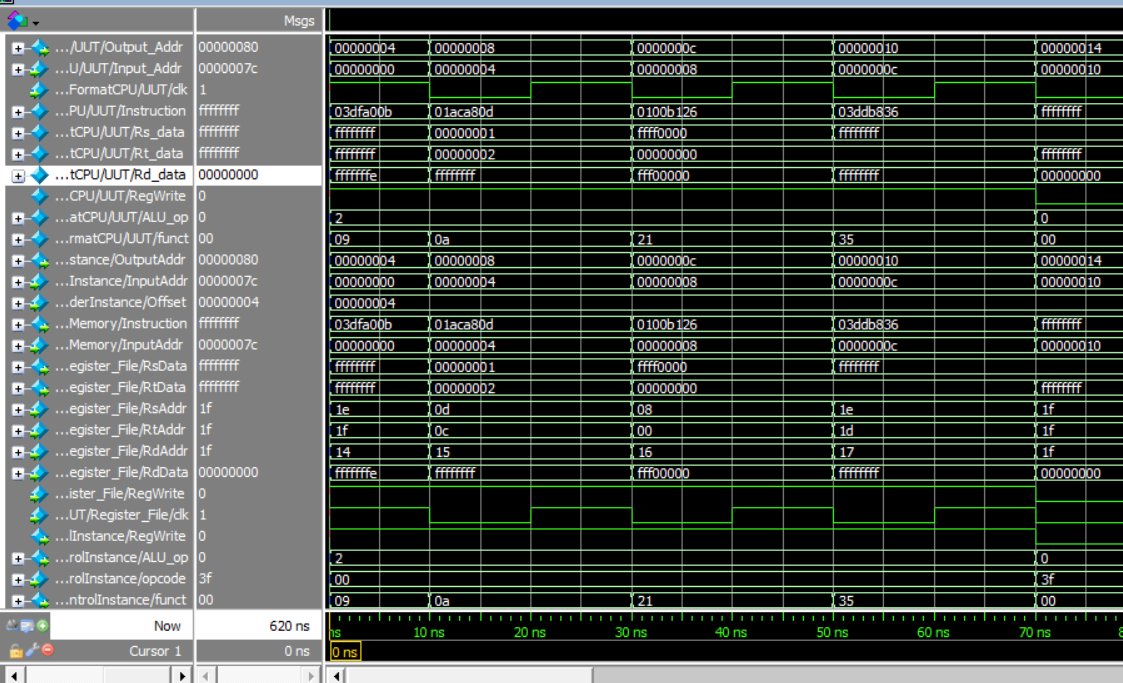
# Custom Test Program

## Part1









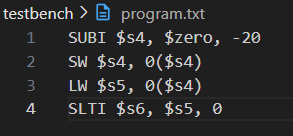
R[20]=R[30]+R[31]=0xFFFF\_FFFF+0xFFFF\_FFFF=0xFFFF\_FFFE. This is test for overflow.

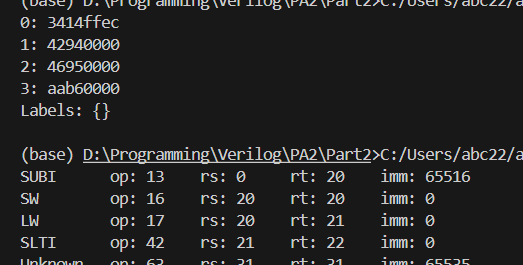
R[21]=R[13]-R[12]=0x0000\_0001-0x0000\_0002=0xFFFF\_FFFF. Test for negative result.

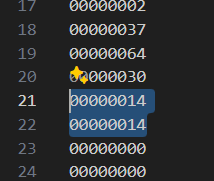
R[22]=R[8]<<4=0xFFFF\_0000<<4=0XFFF0\_0000. Test for overflow and functionality.

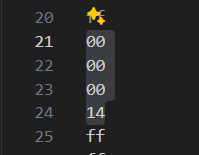
R[23]=R[30]<<R[29][4:0]=0xFFFF\_FFFF<<0b00000=0xFFFF\_FFFF. Test for shift amount = 0.

## Part2









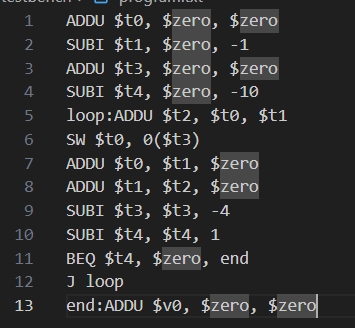
R[20]=R[0]-(-20)=0x0000\_0014. Test for minus negative number.

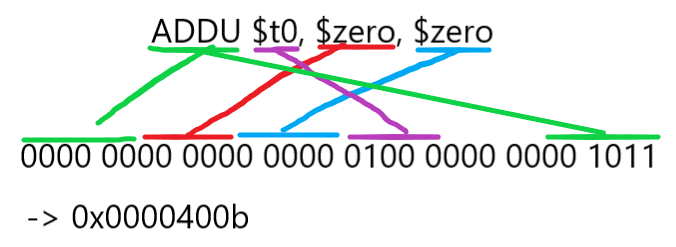
Mem[R[20]+0]=R[20]=0x0000\_0014. Save 20 to position {20, 21, 22, 23}

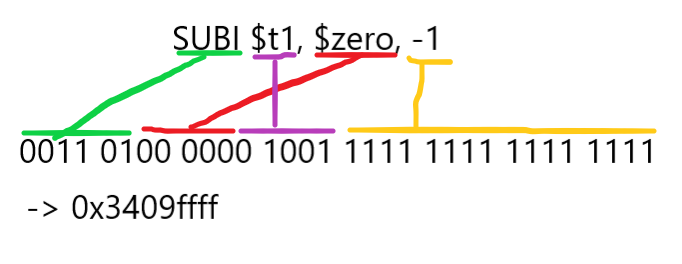
R[21]=Mem[R[20]+0]=0x0000\_0014. Fetch 20 from position {20, 21, 22, 23} and save to R[21]. Test for memory save/load functionality.

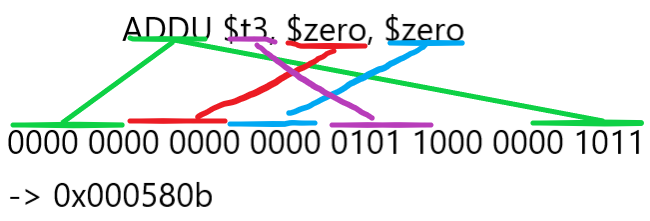
## Part3

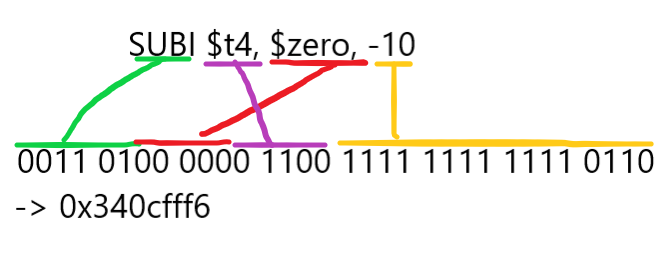
Fibonacci sequence converted by [convertor](#_Convertor.py)

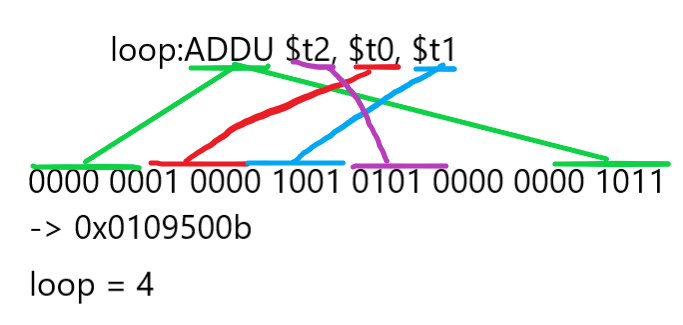


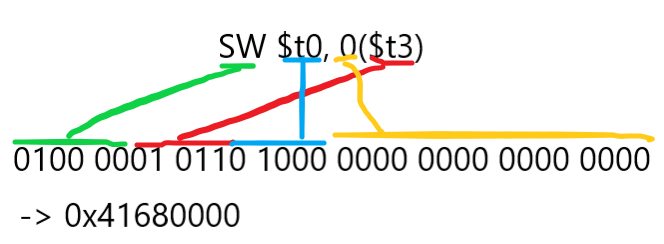


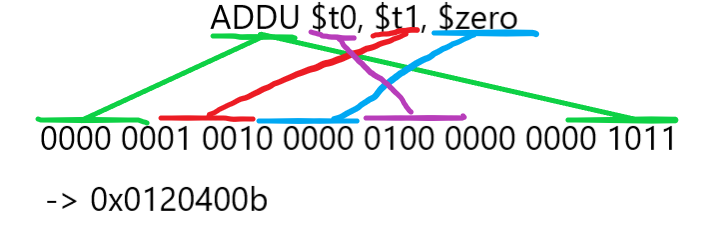


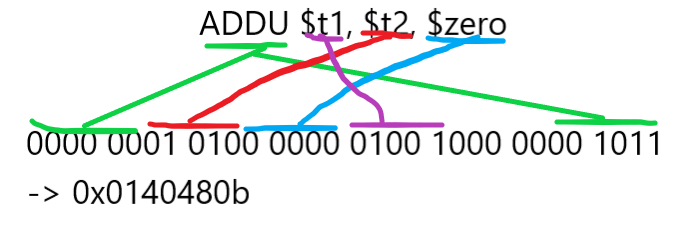


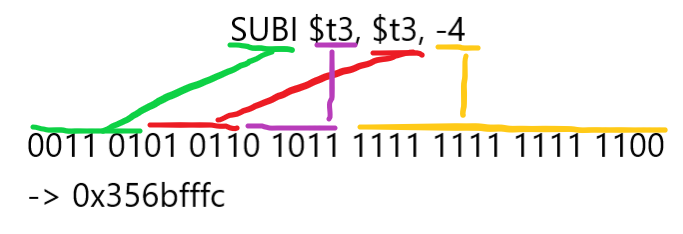


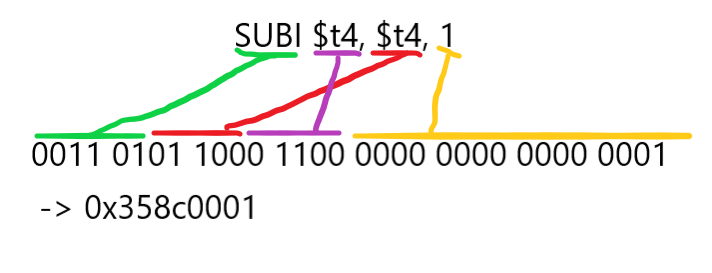


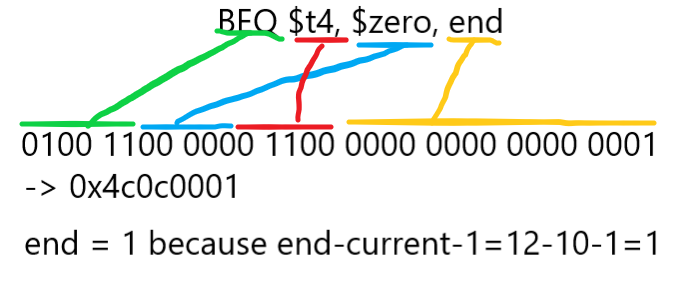




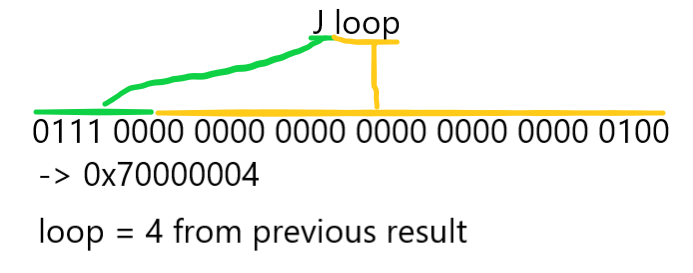


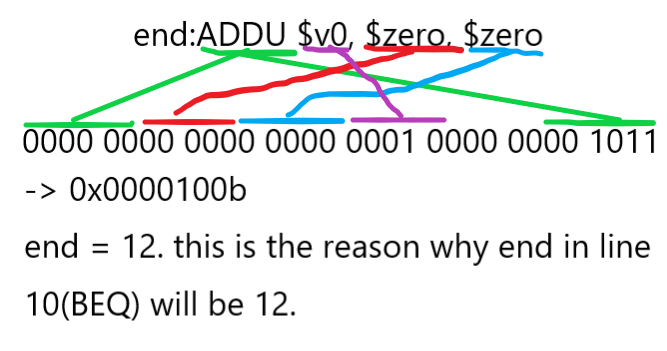


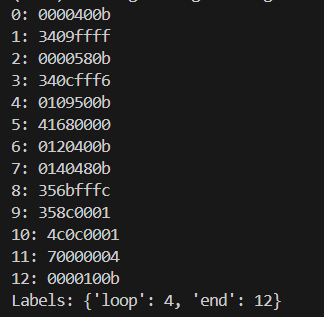


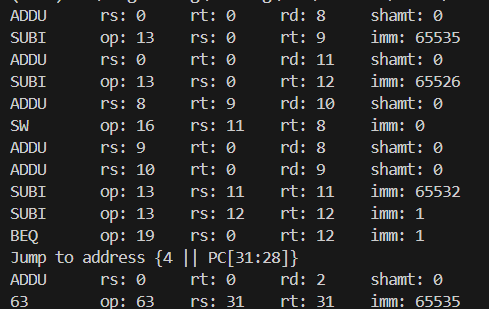


I converted wrong here, but it’s not important. There is no difference between $t4==$zero and $zero==$t4. The circuit is still using rs\_data and rt\_data to judge.





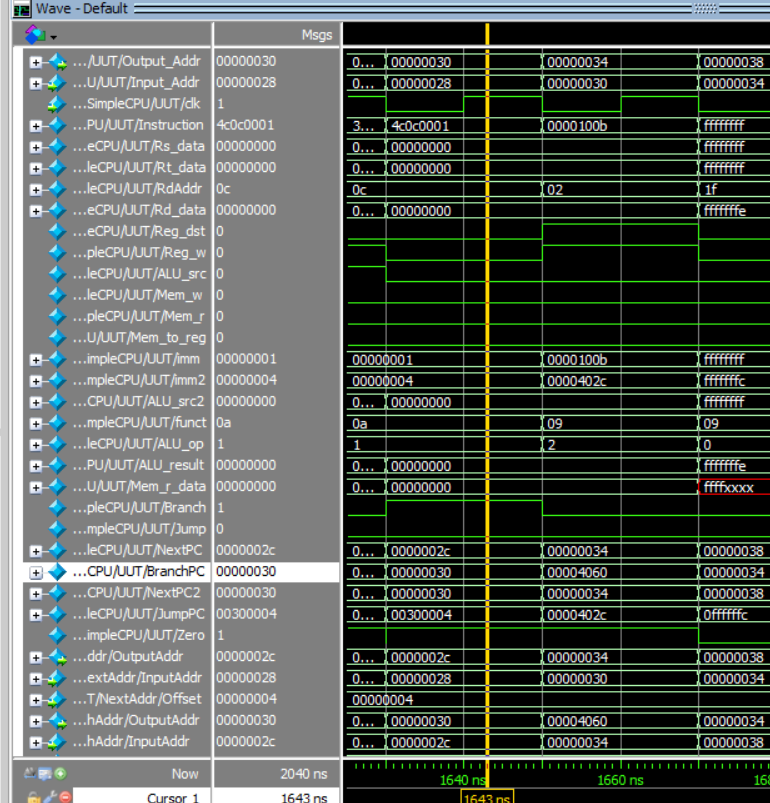








Big endian



The functionality is tested previously. Here I skipped the detail. The correctness of my design can be proven by the result of this program.

Lines in 0~3 are initialization. It set $t0 to 0; $t1 to 1; $t3 to 0; $t4 to 10.

For 1 iteration, the program will do {

ADDU $t2, $t0, $t1 // compute the next Fibonacci num

SW $t0, 0($t3) // save the first register

ADDU $t0, $t1, $zero // set the first’s value to the second’s

ADDU $t1, $t2, $zero // set the second’ value to the next num

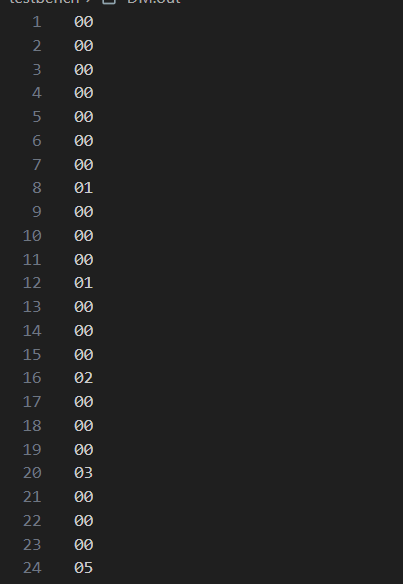
SUBI $t3, $t3, -4 // add 4 to the base address

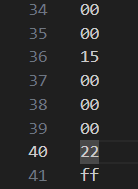
SUBI $t4, $t4, 1 // counter--

}

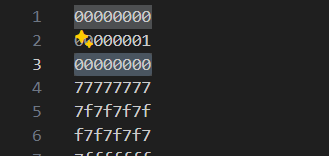
After finish 10 iteration, PC branches to <end> which is +8 bytes from current, to IM = {48, 49, 50, 51} to perform ADDU $v0, $zero, $zero.

Time: (4 setup +8 instr/iter \* 10 iter – 1 the jump is not perform in the last iter) \* 20ns = 1660ns. The last one cycle is ADDU $v0, $zero, $zero.





[00000000, 00000001, 00000001, 00000002, 00000003, …, 00000022] which is exactly 10 Fibonacci sequence numbers.



And the R[2] is set to 0 as well. This program works.

# Conclusion and Insights

By implementing the whole MIPS single cycle CPU, I can say that I’m already fully understand how this structure works.

By observing the testbench, I find that machine code has 0 readability, so I write a [python script](#_Translator.py) to convert it to MIPS instruction with important info printed to cmd.

And the report also requires custom program, so I make a [simple compiler](#_Convertor.py) to compile the program which can use label for BEQ, J to branch.

Because of the last PA I’ve got -12 point in the report, I write as detailed as possible this time. And for each module, I follow the same format whether it has that functionality or not.