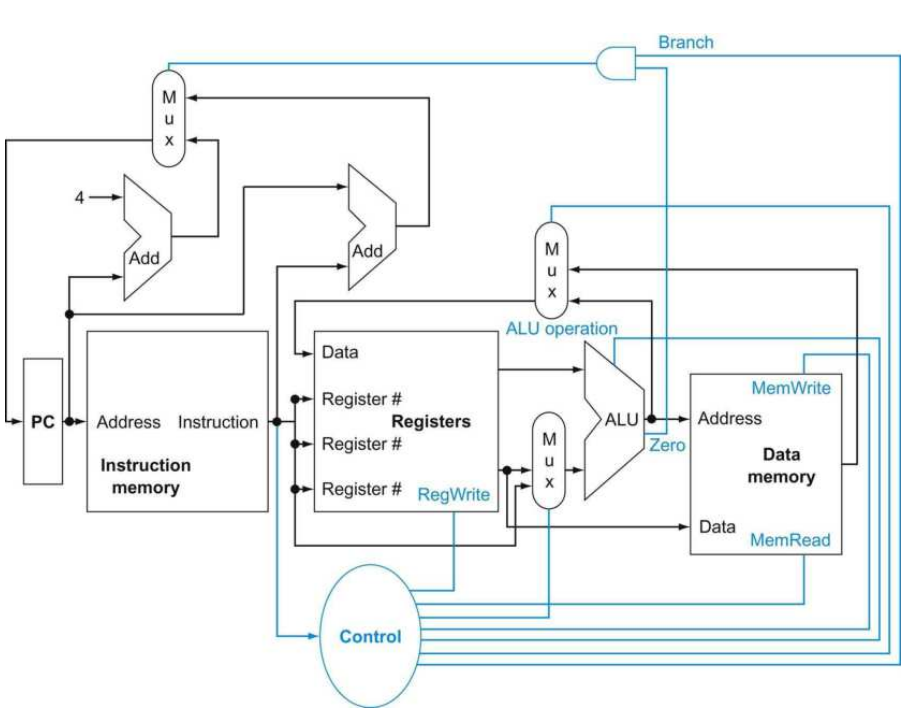
Introduction

* What is RISC-V?
* RISC-V is an open-source instruction set architecture (ISA) used for the development of custom processors targeting a variety of end applications. Originally developed at the University of California, Berkeley, the RISC-V ISA is considered the fifth generation of processors built on the concept of the reduced instruction set computer (RISC). Due to its openness and its technical merits, it has become very popular in recent years. The standard is now managed by RISC-V International, which has more than 3,000 members and which reported that more than 10 billion chips containing RISC-V cores had shipped by the end of 2022. Many implementations of RISC-V are available, both as open-source cores and as commercial IP products.
* How does RISC-V work?
* As an open-standard architecture, RISC-V is defined by member companies of RISC-V International, the global nonprofit organization behind the ISA. The intent is that through collaboration, the member companies can contribute new avenues of processor innovation while promoting new degrees of design freedom.
* The royalty-free RISC-V ISA features a small core set of instructions upon which all the design’s software runs. Its optional extensions allow designers to tailor the architecture for a variety of different end markets. Essentially, the RISC-V architecture allows designers to customize and build their processor in a way that’s tailored to their target end applications, so they can optimize the power, performance, and area (PPA) for those applications. The RISC-V ISA also provides the flexibility to pick and choose from available features, rather than having to use the full feature set.
* While the initial market adoption of RISC-V has been with embedded applications and microcontrollers, the open-source architecture also holds promise for high-performance computing and data centers.
* What are the benefits of RISC-V?
* Its open-standard nature, which allows collaboration and innovation across the industry
* Common ISA, which helps make software development easier since all processors could potentially use the same architecture. Designers can use the same base ISA, from simple embedded devices to the largest supercomputers, tailoring their device to the needs of the market. Compared to previous ISAs, RISC-V ISAs have unique features and can be customized based on their requirements.
* Availability of smaller, energy-efficient, and modular options
* Security features, which are available through open-source reference designs, software composition analysis tools, and security extensions. In addition, its open-source nature means that the entire RISC-V architecture can be scrutinized closely in the public domain, eliminating back doors and hidden channels.
* What are key RISC-V applications?
* **Wearables, Industrial, IoT, and Home Appliances**
* RISC-V processors are ideal for meeting the power requirements of space-constrained and battery-operated designs.
* **Smartphones**
* RISC-V cores can be customized to handle the performance needed to power smartphones, or can be used as part of a larger SoC to handle specific tasks for phone operation.
* **Automotive, High-Performance Computing (HPC), and Data Centers**
* RISC-V cores can handle complex computational tasks with customized ISAs, while RISC-V extensions enable development of simple, secure, and flexible cores for greater energy efficiency.
* **Aerospace and Government**
* RISC-V offers high reliability and security for these use applications.
* RISC-V Implementation
* In this project, we will design a simple Single Cycle Non Pipelined 32-bit RISC-V processor with 32, 16- bit general purpose registers: R1 through R32.There is also one special-purpose 32-bit register, which is the program counter (PC). All instructions are only 32 bits. There are four instruction formats, R-type, I-type, S-type and SB-type.
* Instruction Set of this project includes:
* The memory-reference instructions load doubleword (ld) and store doubleword (sd).
* The arithmetic-logical instructions add, sub, and, and or.
* The conditional branch instruction branch if equal (beq).
* For every instruction, the first two steps are identical:
* Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.
* Read one or two registers, using fields of the instruction to select the registers to read. For the ld instruction, we need to read only one register, but most other instructions require reading two registers.
* After these two steps, the actions required to complete the instruction depend on the instruction class. Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the exact instruction.
* All instruction classes use the arithmetic-logical unit (ALU) after reading the registers. The memory-reference instructions use the ALU for an address calculation, the arithmeticlogical instructions for the operation execution, and conditional branches for the equality test. After using the ALU, the actions required to complete various instruction classes differ. A memoryreference instruction will need to access the memory either to read data for a load or write data for a store. An arithmetic-logical or load instruction must write the data from the ALU or memory back into a register. Lastly, for a conditional branch instruction, we may need to change the next instruction address based on the comparison; otherwise, the PC should be incremented by four to get the address of the subsequent instruction.
* All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or an equality check (for a branch).
* If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes either from the adder (where the PC and branch offset are summed) or from an adder that increments the current PC by four.
* The top multiplexor (“Mux”) controls what value replaces the PC (PC + 4 or the branch destination address); the multiplexor is controlled by the gate that “ANDs” together the Zero output of the ALU and a control signal that indicates that the instruction is a branch. The middle multiplexor, whose output returns to the register file, is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file. Finally, the bottom-most multiplexor is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch) or from the offset field of the instruction (for a load or store). The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation.



* Building a Datapath for RISC-V
* The first Datapath element we need is Instruction Memory. It is a memory unit to store the instructions of a program and supply instructions given an address. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.
* Design Code of Instruction Memory.

// Designing Instruction Memory for RISC-V(Single Cycle Processor)

`timescale 1ns/1ns

module Instruction\_Memory(Instruction\_out, read\_address, clk, reset);

// port declarations.

input clk, reset;

input [31:0]read\_address; //Reading Instruction Address sent PC.

output [31:0]Instruction\_out;

//Making Memory

reg [31:0] InstructionMemory [31:0]; //considering PC 32-bit, memory consists of 31 cells each of 32 bit wide.

assign Instruction\_out = InstructionMemory[read\_address]; //Assigning Memory value.

integer i;

always @(posedge clk)

begin

if(reset == 1'b1)

begin

for(i = 0; i < 32; i = i+1)

InstructionMemory[i] = 32'b0; //Initialize when reset.

end

else if(reset == 1'b0)

begin

InstructionMemory[4] = 32'b0000000\_00010\_00001\_000\_00001\_0110011; // add R1, R1, R2

InstructionMemory[8] = 32'b0100000\_00010\_00001\_000\_00001\_0110011; // sub R1, R1, R2

InstructionMemory[12] = 32'b0000000\_00010\_00001\_111\_00001\_0110011; // and R1, R1, R2

InstructionMemory[16] = 32'b0000000\_00010\_00001\_110\_00001\_0110011; // or R1, R1, R2

InstructionMemory[20] = 32'b000000000111\_00010\_000\_00001\_0000011; // ld R1, 0(R2)

InstructionMemory[24] = 32'b0000000\_00001\_00010\_000\_00011\_0100011; // sd R1, 0(R2)

InstructionMemory[28] = 32'b0000000\_00010\_00001\_000\_11111\_1100011; // beq R1, R2, offset

end

end

endmodule

* Testbench Code for Instruction Memory.

// Testbench for Instruction Memory for RISC-V(Single Cycle Processor)

module Instruction\_Memory\_tb();

// port declarations.

reg clk, reset;

reg [31:0]read\_address; //Reading Instruction Address sent PC.

wire [31:0]Instruction\_out;

Instruction\_Memory uut(Instruction\_out, read\_address, clk, reset);

//clock generations.

initial

begin

clk=1;

end

always #10 clk = ~clk;

//reset generations.

initial

begin

reset=1'b1;

#10 reset=1'b0;

end

initial

begin

//$dumpfile("Instruction\_Memory.vcd");

//$dumpvars(0,Instruction\_Memory\_tb);

$monitor($time , " reset=%b, clk=%b, read\_address=%h, Instruction\_out=%h ", reset, clk, read\_address, Instruction\_out);

//#20 read\_address=32'h00000000;

#20 read\_address=32'h00000004;

#20 read\_address=32'h00000008;

#20 read\_address=32'h0000000C;

#20 read\_address=32'h00000010;

#20 read\_address=32'h00000014;

#20 read\_address=32'h00000018;

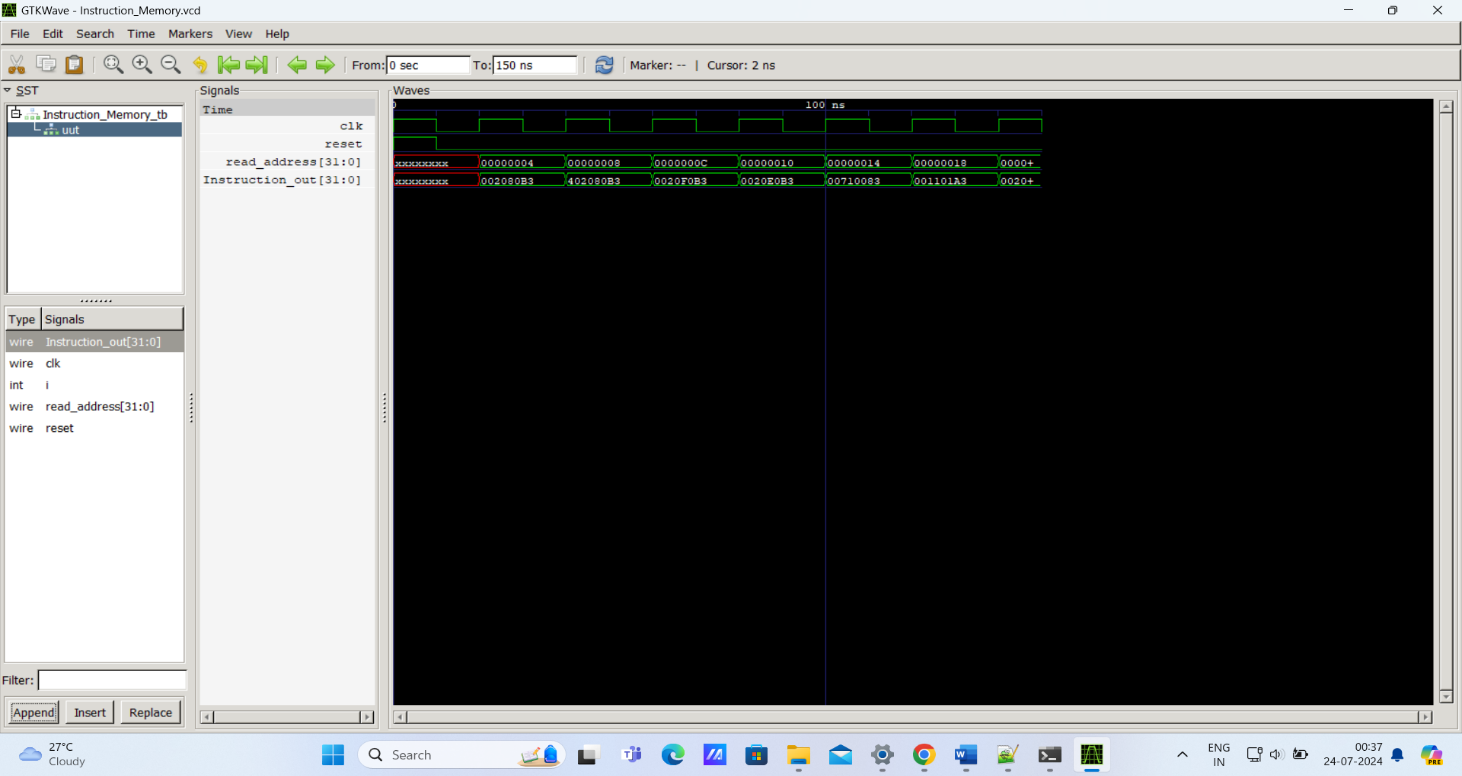
#20 read\_address=32'h0000001C;

#10 $finish;

end

endmodule

* Waveform for Instruction Memory.



* The second Datapath element we need is Program Counter (PC), which is a 32-bit register that holds the address of the current instruction. It is written at the end of every clock cycle and thus does not need a write control signal.
* Design Code of Program Counter.

// Designing Program Counter(PC) for RISC-V(Single Cycle Processor)

`timescale 1ns/1ns

module Program\_Counter(PC\_OUT, PC\_IN, clk, reset);

// port declarations.

input clk, reset;

input [31:0]PC\_IN;

output reg [31:0]PC\_OUT;

always @(posedge clk or posedge reset) //PC and reset is triggered at positive edge of clock.

begin

if(reset==1'b1)

PC\_OUT <= 0; //reset is high initialize PC\_OUT with zero.

else

PC\_OUT <= PC\_IN;

end

endmodule

* Testbench Code for Program Counter.

// TestBench for Program Counter(PC) for RISC-V(Single Cycle Processor)

module Program\_Counter\_tb();

// port declarations.

reg clk, reset;

reg [31:0]PC\_IN;

wire [31:0]PC\_OUT;

Program\_Counter uut(PC\_OUT, PC\_IN, clk, reset);

//clock generations.

initial

begin

clk=0;

end

always #10 clk = ~clk;

//reset generations.

initial

begin

reset=1'b1;

#10 reset=1'b0;

end

initial

begin

PC\_IN=32'h00000000;

end

always #20 PC\_IN = PC\_IN + 4;

initial

begin

$dumpfile("Program\_Counter.vcd");

$dumpvars(0,Program\_Counter\_tb);

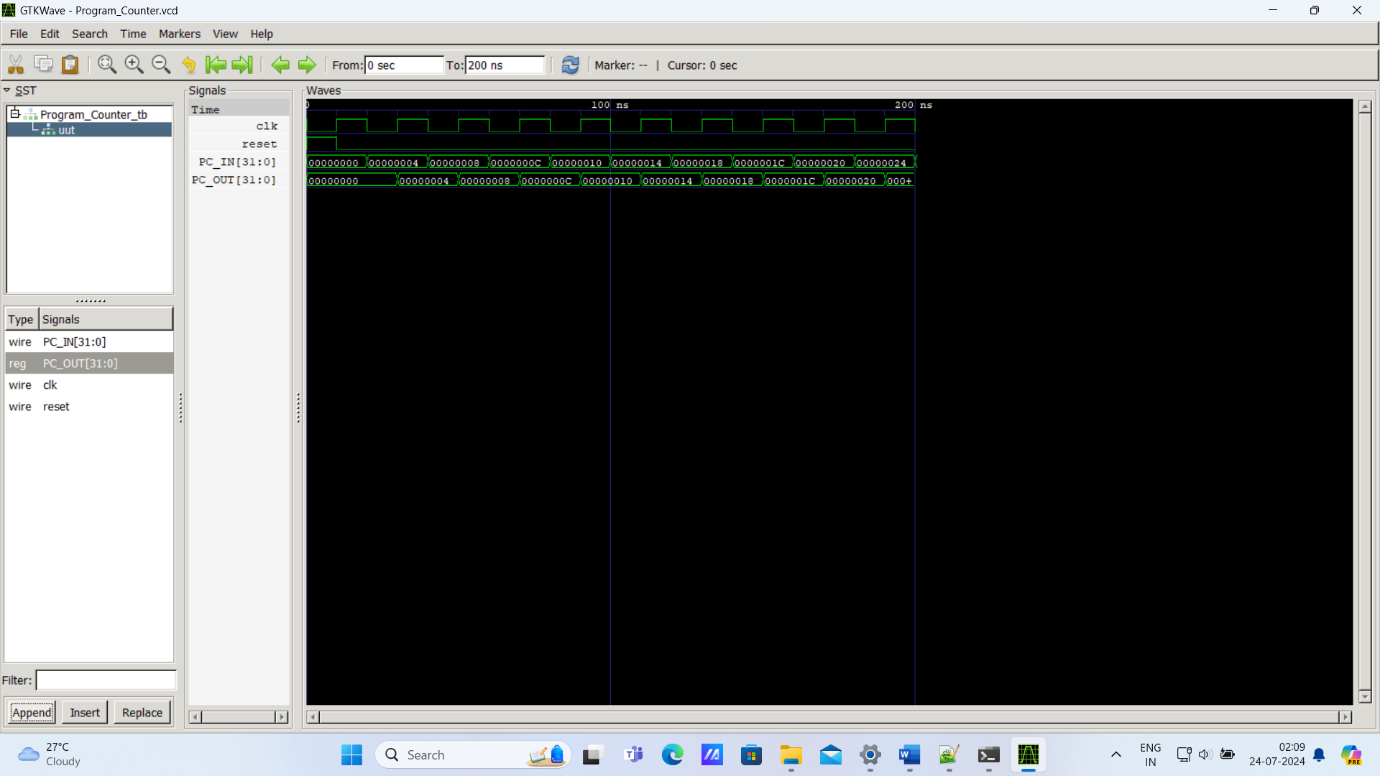
$monitor($time , " clk=%b, reset=%b, PC\_IN=%h, PC\_OUT=%h ", clk, reset, PC\_IN, PC\_OUT);

#200 $finish;

end

endmodule

* Waveform for Program Counter.



* The third Datapath element we need is an Adder to increment the PC to the address of the next instruction. It has been permanently made an adder and cannot perform the other ALU functions.
* Design Code for Program Counter Adder.

// Designing Program Counter(PC) adder for next Instruction for RISC-V(Single Cycle Processor)

`timescale 1ns/1ns

module Program\_Counter\_Next(PC\_OUT\_NEXT, PC\_IN\_NEXT);

// port declarations.

input [31:0] PC\_IN\_NEXT;

output [31:0] PC\_OUT\_NEXT;

assign PC\_OUT\_NEXT = PC\_IN\_NEXT + 32'h00000004;

endmodule

* Testbench Code for Program Counter Adder.

//Testbench for Program Counter(PC) adder for next Instruction for RISC-V(Single Cycle Processor)

module Program\_Counter\_Next\_tb();

// port declarations.

reg [31:0] PC\_IN\_NEXT;

wire [31:0] PC\_OUT\_NEXT;

Program\_Counter\_Next uut(PC\_OUT\_NEXT, PC\_IN\_NEXT);

initial

begin

$dumpfile("Program\_Counter\_Next.vcd");

$dumpvars(0,Program\_Counter\_Next\_tb);

$monitor($time , " PC\_IN\_NEXT=%h, PC\_OUT\_NEXT=%h ", PC\_IN\_NEXT, PC\_OUT\_NEXT);

#5 PC\_IN\_NEXT=32'hFFCE6798;

#5 PC\_IN\_NEXT=32'h00000004;

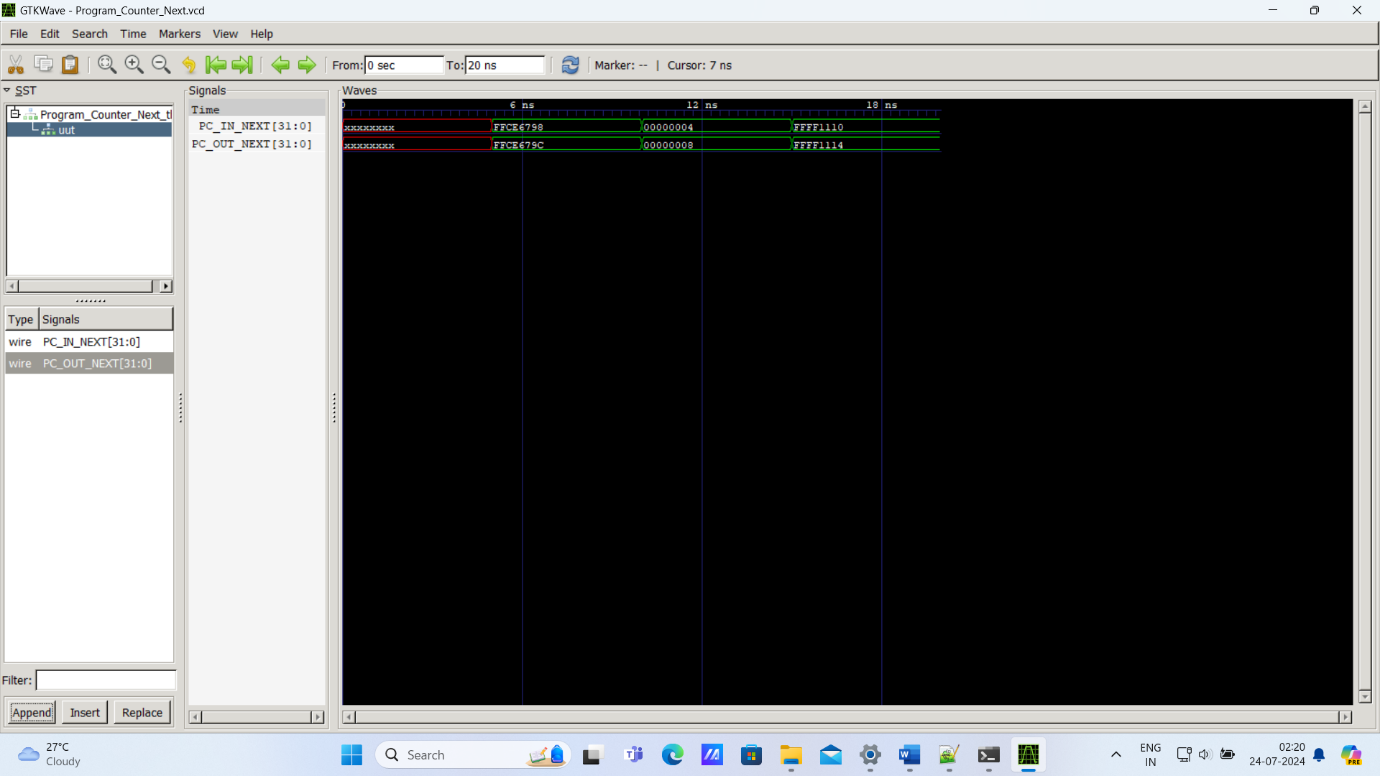
#5 PC\_IN\_NEXT=32'hFFFF1110;

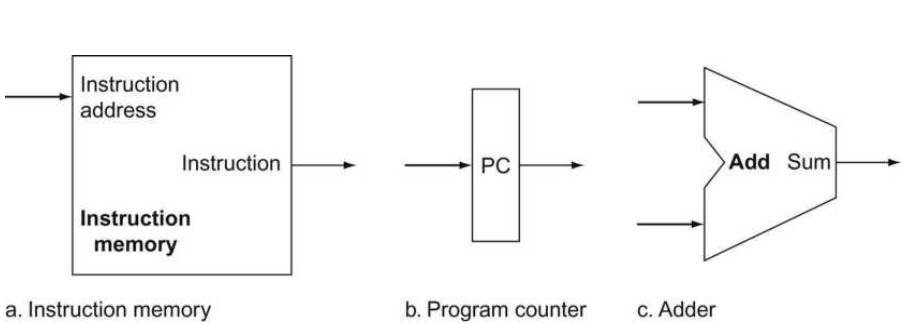
#5 $finish;

end

endmodule

* Waveform for Program Counter Adder.





* Design Code for Branch Target Adder.

// Designing Branch target for RISC-V(Single Cycle Processor)

`timescale 1ns/1ns

module Branch\_Target(Branch\_target, Imm\_Gen\_out, PC\_from\_Next);

// port declarations.

input [31:0] Imm\_Gen\_out;

input [31:0] PC\_from\_Next;

output [31:0] Branch\_target;

assign Branch\_target = (Imm\_Gen\_out << 1) + PC\_from\_Next;

endmodule

* Testbench for Branch Target Adder.

// Testbench for Branch target for RISC-V(Single Cycle Processor)

module Branch\_Target\_tb();

// port declarations.

reg [31:0] Imm\_Gen\_out;

reg [31:0] PC\_from\_Next;

wire [31:0] Branch\_target;

Branch\_Target uut(Branch\_target, Imm\_Gen\_out, PC\_from\_Next);

initial

begin

//$dumpfile("Branch\_Target");

//$dumpvars(0,Branch\_Target\_tb);

$monitor($time , " Imm\_Gen\_out=%h, PC\_from\_Next=%h, Branch\_target=%h ", Imm\_Gen\_out, PC\_from\_Next, Branch\_target);

#5 Imm\_Gen\_out=32'h00000001; PC\_from\_Next=32'h00000001;

#5 Imm\_Gen\_out=32'h00000010; PC\_from\_Next=32'h00000001;

#5 Imm\_Gen\_out=32'h00000100; PC\_from\_Next=32'h00000001;

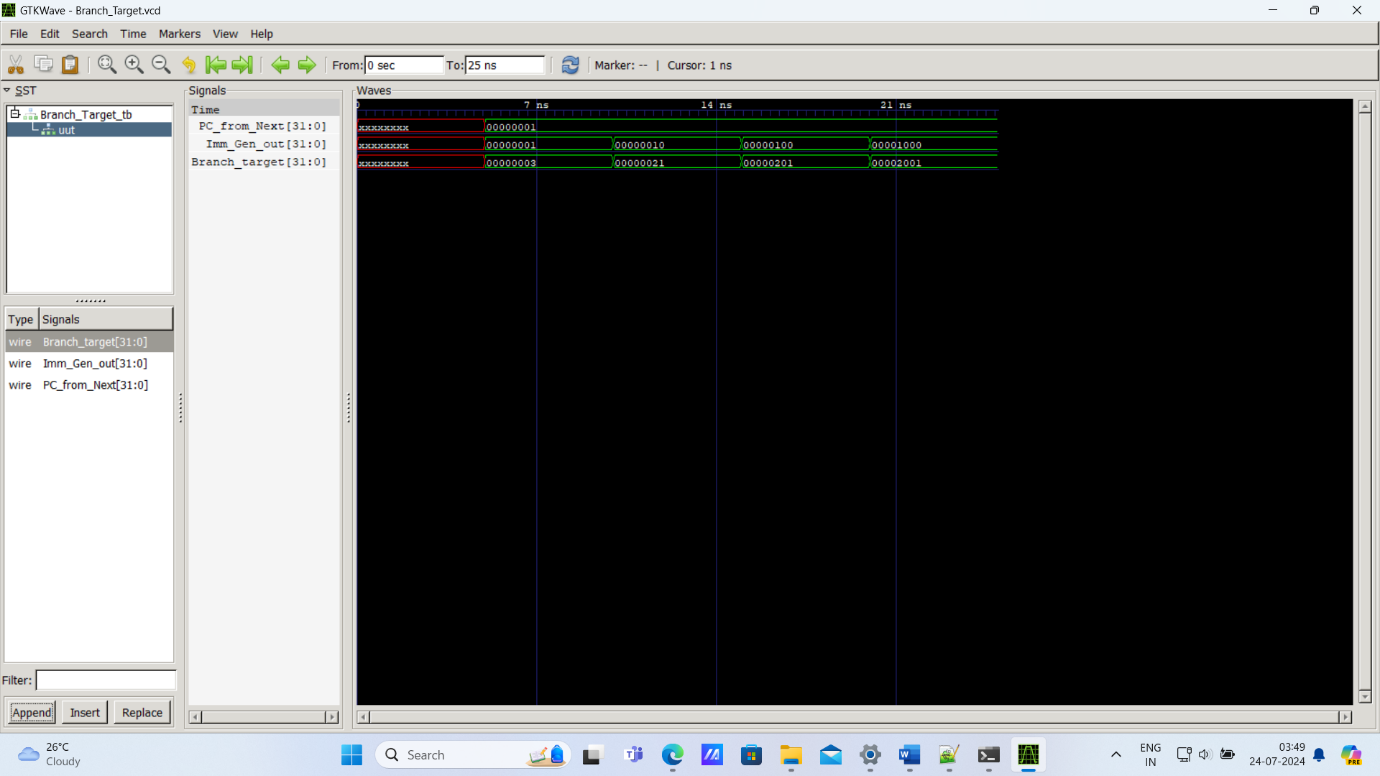
#5 Imm\_Gen\_out=32'h00001000; PC\_from\_Next=32'h00000001;

#5 $finish;

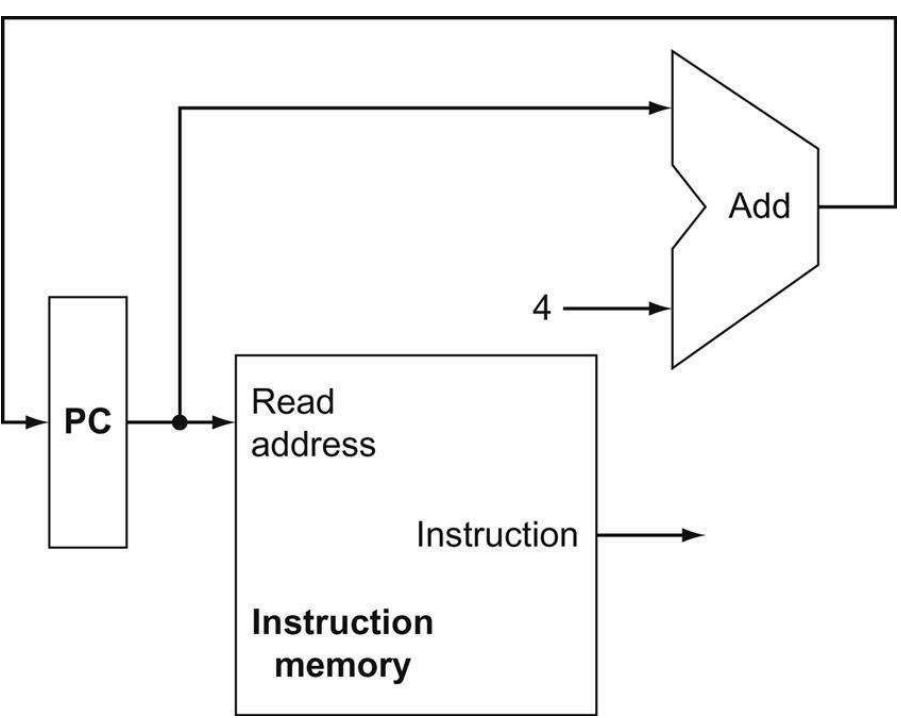
end

endmodule

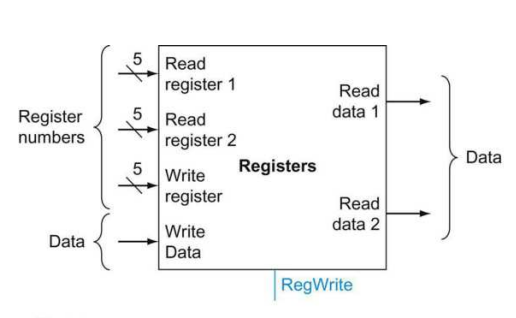
* Waveform for Branch Target Adder.



* To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later. Figure below shows how to combine the three elements from Figure above to form a datapath that fetches instructions and increments the PC to obtain the address of the next sequential instruction.



* The fourth Datapath element we need is Register file. The processor’s 32 general-purpose registers are stored in a structure called a register file. A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file. We need Alu to operate on the values read from the registers. R-format instructions have three register operands i.e. read two data words from the register file and write one data word into the register file for each instruction. For each data word to be read from the registers, we need an input to the register file that specifies the register number to be read and an output from the register file that will carry the value that has been read from the registers. To write a data word, we will need two inputs: one to specify the register number to be written and one to supply the data to be written into the register. we need a total of three inputs (two for register numbers and one for data) and two outputs (both for data). The register number inputs are 5 bits wide to specify one of 32 registers (32 = 2^5 ), whereas the data input and two data output buses are each 64 bits wide.



* RISC-V load register and store register instructions, which have the general form ld x1, offset(x2) or sd x1, offset(x2). These instructions compute a memory address by adding the base register, which is x2, to the 12-bit signed offset field contained in the instruction. If the instruction is a store, the value to be stored must also be read from the register file where it resides in x1. If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is x1. Thus, we will need both the register file and the ALU.
* Design Code for Register File.

// Designing Register File for RISC-V(Single Cycle Processor)

`timescale 1ns/1ns

module Register\_File(Read\_data1, Read\_data2, Write\_data, Read\_register1,

Read\_register2, Write\_register, RegWrite, clk, reset);

// port declarations.

input RegWrite, clk, reset;

input [19:15] Read\_register1;

input [24:20] Read\_register2;

input [11:7] Write\_register;

input [31:0] Write\_data;

output [31:0] Read\_data1, Read\_data2;

//Making Register File

reg [31:0] Registers [31:0]; // 32 Registers each of 32-bit wide.

integer i;

initial

begin

for(i = 0; i < 32; i = i + 1)

begin

Registers[i] = i\*10;

end

end

always @(posedge clk) // Assigning values with write control signal.

begin

if(reset == 1'b1)

begin

for(i = 0; i < 32; i = i + 1)

begin

Registers[i] = 32'h00000000;

end

end

else if(RegWrite == 1'b1)

begin

Registers[Write\_register] = Write\_data;

end

end

assign Read\_data1 = Registers[Read\_register1]; // Assigning values with no read control signal.

assign Read\_data2 = Registers[Read\_register2]; // Assigning values with no read control signal.

endmodule

* Testbench Code for Register File.

// Testbench for Register File for RISC-V(Single Cycle Processor)

module Register\_File\_tb();

// port declarations.

reg RegWrite, clk, reset;

reg [19:15] Read\_register1;

reg [24:20] Read\_register2;

reg [11:7] Write\_register;

reg [31:0] Write\_data;

wire [31:0] Read\_data1, Read\_data2;

Register\_File uut(Read\_data1, Read\_data2, Write\_data, Read\_register1,

Read\_register2, Write\_register, RegWrite, clk, reset);

//clock generations.

initial

begin

clk=1;

end

always #10 clk = ~clk;

//reset generations.

initial

begin

reset=1'b1;

#10 reset=1'b0;

end

initial

begin

$dumpfile("Register\_File.vcd");

$dumpvars(0,Register\_File\_tb);

$monitor($time , " reset=%b, clk=%b, Rd1=%d, Rd2=%d, Wd=%d, Rs1=%d, Rs2=%d, Ws=%d, RW=%b", reset, clk, Read\_data1, Read\_data2, Write\_data, Read\_register1,Read\_register2, Write\_register, RegWrite);

#20 RegWrite=0; Read\_register1=0; Read\_register2=1;

#20 RegWrite=0; Read\_register1=1; Read\_register2=19;

#20 RegWrite=1; Read\_register1=25; Write\_register=25; Write\_data=1025;

#20 RegWrite=1; Read\_register1=10; Write\_register=10; Write\_data=100025;

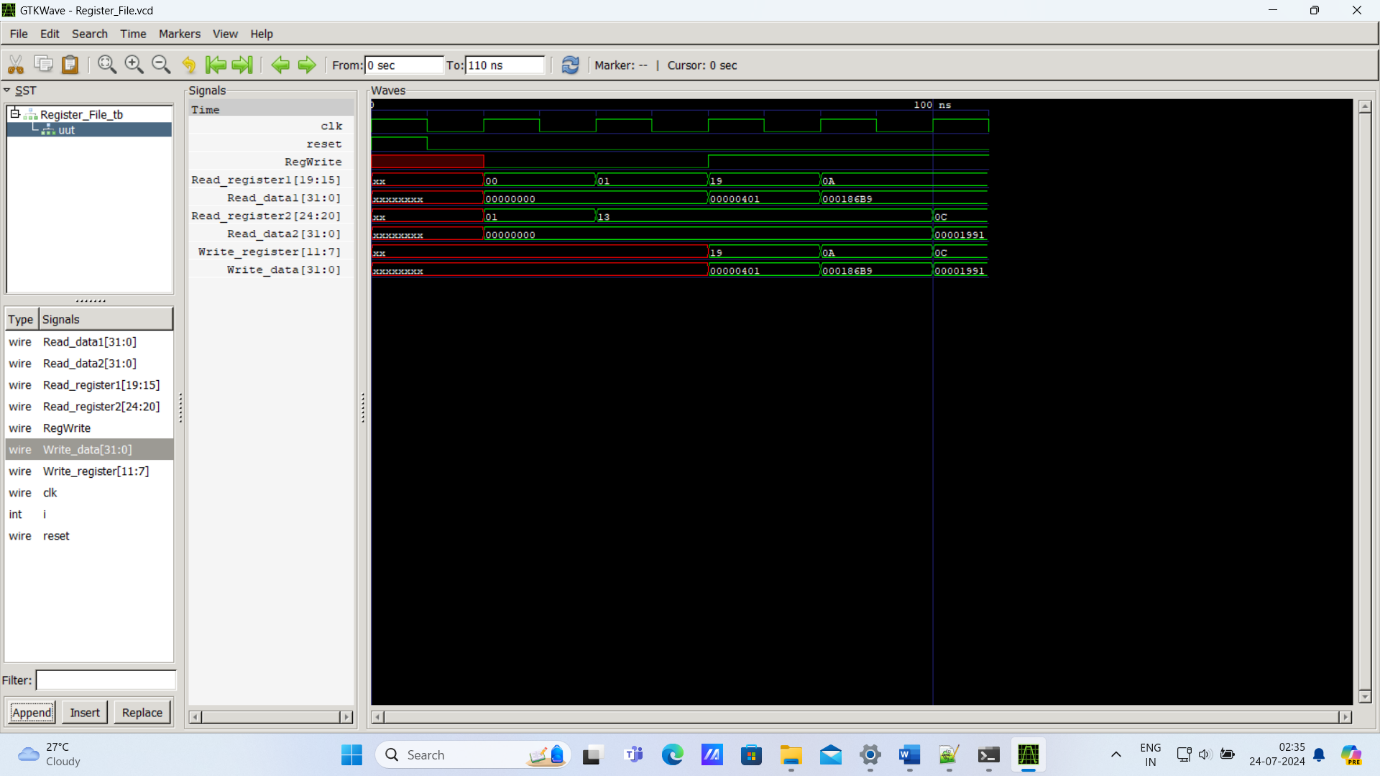
#20 RegWrite=1; Read\_register1=10; Read\_register2=12; Write\_register=12; Write\_data=6545;

#10 $finish;

end

endmodule

* Waveform of Register File.



* The fifth Datapath element we need is Immediate Generation. The immediate generation unit (ImmGen) has a 32-bit instruction as input that selects a 12-bit field for load, store, and branch if equal that is sign-extended into a 32-bit result appearing on the output .

sign-extended means to increase the size of a data item by replicating the high-order sign bit of the original data item in the high-order bits of the larger, destination data item.

* Design Code for Immediate Generation.

// Designing Immediate Generator for RISC-V(Single Cycle Processor)

`timescale 1ns/1ns

module Immediate\_Generator(Imm\_Gen, Instruction, Opcode);

// port declarations.

input [6:0] Opcode;

input [31:0] Instruction;

output reg [31:0] Imm\_Gen;

always @(\*)

begin

case(Opcode)

7'b0000011:Imm\_Gen <= {{20{Instruction[31]}}, Instruction[31:20]}; // ld type Instruction.

7'b0100011:Imm\_Gen <= {{20{Instruction[31]}}, Instruction[31:25], Instruction[11:7]}; // sd type Instruction.

7'b1100011:Imm\_Gen <= {{20{Instruction[31]}}, Instruction[31], Instruction[7], Instruction[30:25], Instruction[11:8]}; // beq type Instruction.

default:Imm\_Gen <= 32'h00000000;

endcase

end

endmodule

* Testbench Code for Immediate Generation.

module Immediate\_Generator\_tb();

// port declarations.

reg [6:0] Opcode;

reg [31:0] Instruction;

wire [31:0] Imm\_Gen;

Immediate\_Generator uut(Imm\_Gen, Instruction, Opcode);

initial

begin

$dumpfile("Immediate\_Generator.vcd");

$dumpvars(0,Immediate\_Generator\_tb);

$monitor($time , " Imm\_Gen=%h, Instruction=%h, Opcode=%h",Imm\_Gen, Instruction, Opcode);

#20 Instruction=32'b111100001111\_00010\_000\_00001\_0000011; Opcode=7'b0000011;//ld

#20 Instruction=32'b1111111\_00001\_00010\_000\_00000\_0100011; Opcode=7'b0100011;//sd

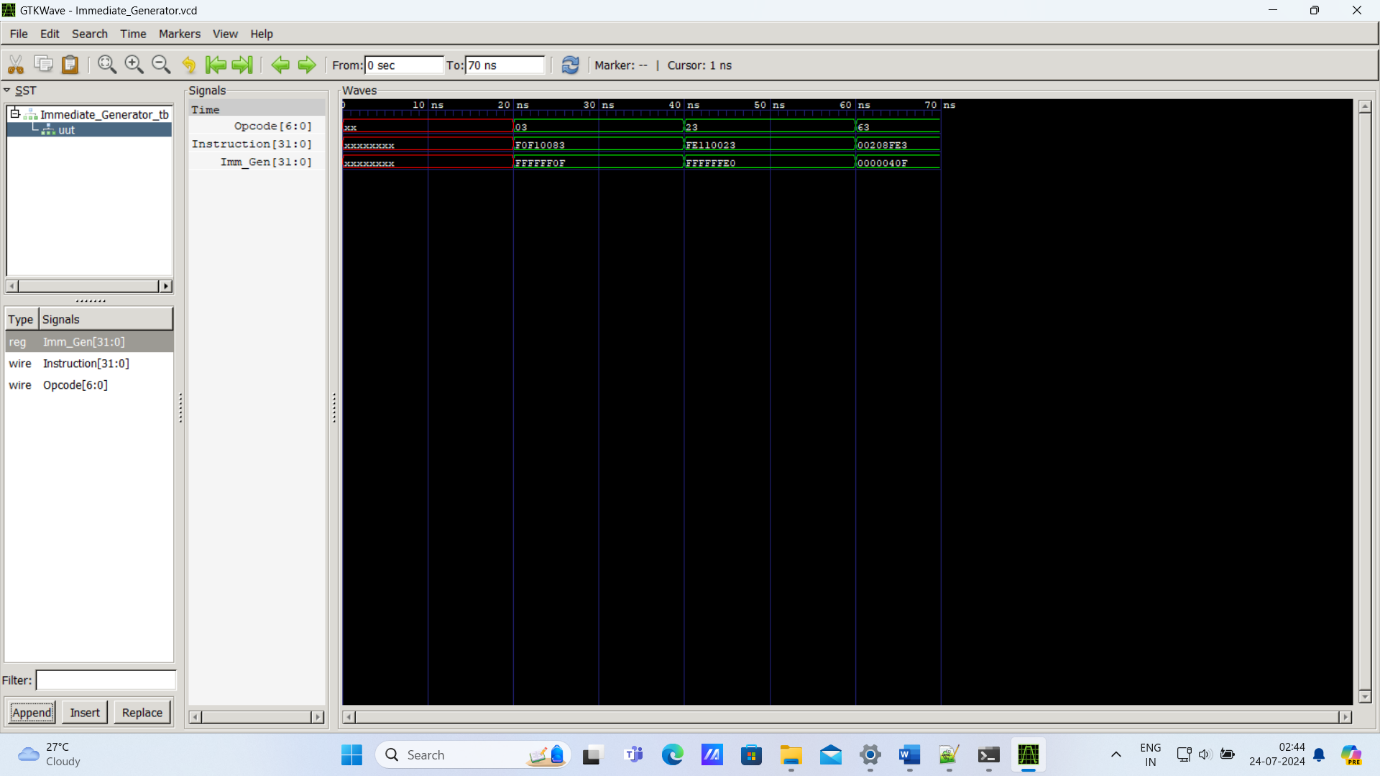
#20 Instruction=32'b0000000\_00010\_00001\_000\_11111\_1100011; Opcode=7'b1100011;//beq

#10 $finish;

end

endmodule

* Waveform for Immediate Generation.



* The sixth Datapath element we need is Data Memory. The data memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock.
* Design Code for Data Memory.

// Designing Data Memory for RISC-V(Single Cycle Processor)

`timescale 1ns/1ns

module Data\_Memory(Read\_data, Write\_data, MemWrite, MemRead, address, clk, reset);

// port declarations.

input MemWrite, MemRead, clk, reset;

input [31:0] Write\_data, address;

output [31:0] Read\_data;

//Making Memory

reg [31:0] DataMemory [31:0];

//considering address 32-bit, data memory consists of 32 cells each of 32 bit wide.

assign Read\_data= (MemRead) ? DataMemory[address] : 32'h00000000; //Assigning DataMemory value.

integer i;

always @(posedge clk or posedge reset)

begin

if(reset == 1'b1)

begin

for(i=0;i<32;i=i+1)

begin

DataMemory[i] = i\*20; //Initialize when reset.

end

end

else if(MemWrite)

begin

DataMemory[address]=Write\_data;

end

end

endmodule

* Testbench Code for Data Memory.

// Testbench for Data Memory for RISC-V(Single Cycle Processor)

module Data\_Memory\_tb();

// port declarations.

reg MemWrite, MemRead, clk, reset;

reg [31:0] Write\_data, address;

wire [31:0] Read\_data;

Data\_Memory uut(Read\_data, Write\_data, MemWrite, MemRead, address, clk, reset);

//clock generations.

initial

begin

clk=1;

end

always #10 clk = ~clk;

//reset generations.

initial

begin

reset=1'b1;

#10 reset=1'b0;

end

initial

begin

$dumpfile("Data\_Memory.vcd");

$dumpvars(0,Data\_Memory\_tb);

$monitor($time , " reset=%b, clk=%b, Read\_data=%h, Write\_data=%h, MemWrite=%b, MemRead=%b, address=%h ", reset, clk, Read\_data, Write\_data, MemWrite, MemRead, address);

#20 MemRead=1; address=25;

#20 Write\_data=32'hffff0001; MemWrite=1; MemRead=0; address=25;

#20 MemWrite=0; MemRead=1; address=25;

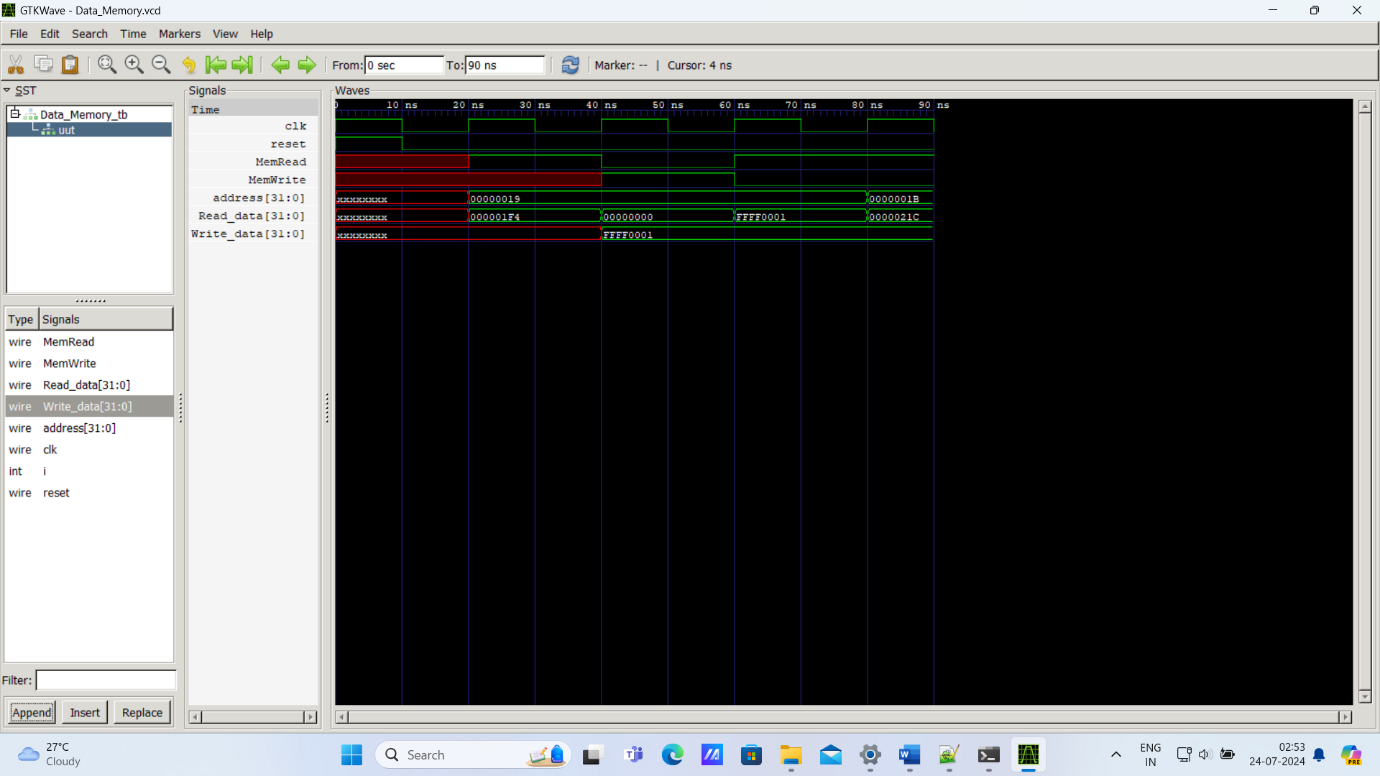
#20 MemRead=1; address=27;

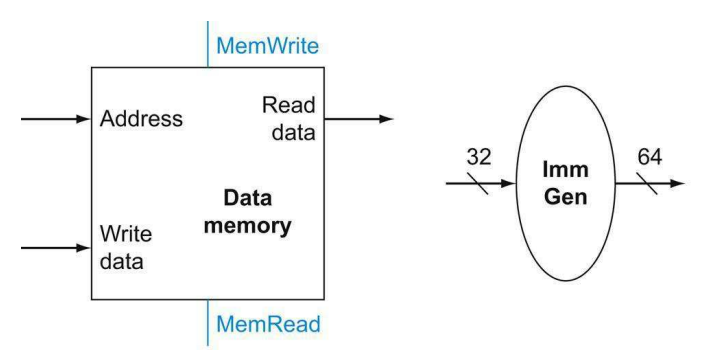
#10 $finish;

end

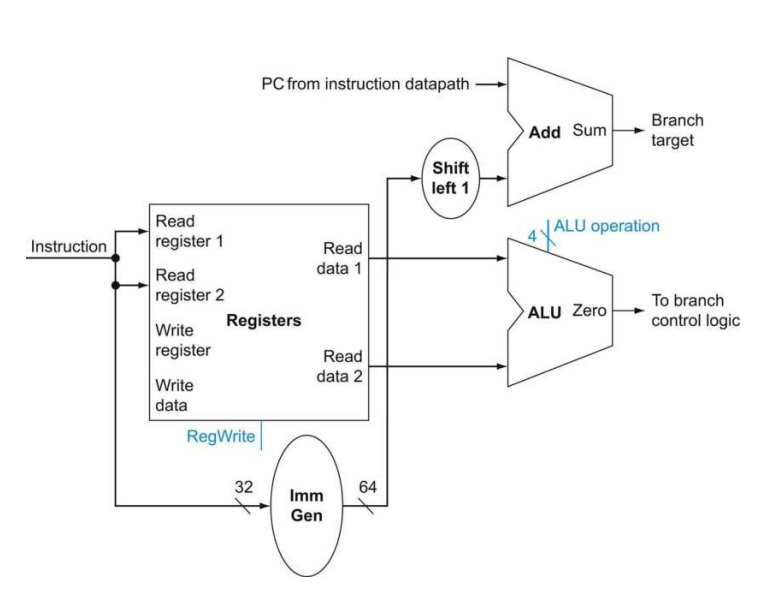
endmodule

* Waveform for Data Memory.





* The beq instruction has three operands, two registers that are compared for equality, and a 12-bit offset used to compute the branch target address relative to the branch instruction address. Its form is beq x1, x2, offset. Branch target address means the address specified in a branch, which becomes the new program counter (PC) if the branch is taken. In the RISC-V architecture, the branch target is given by the sum of the sign-extended offset field of the instruction and the address of the branch(PC). In the RISC-V architecture, the offset field is shifted left 1 bit so that it is a half word offset; this shift increases the effective range of the offset field by a factor of 2.
* The branch datapath must do two operations: compute the branch target address and test the register contents. To compute the branch target address, the branch datapath includes an immediate generation unit and an adder. To perform the compare, we need to use the register file to supply two register operands. ALU provides an output signal that indicates whether the result was 0, we can send both register operands to the ALU with the control set to subtract two values. If the Zero signal out of the ALU unit is asserted, we know that the register values are equal.



* The seventh Datapath element we need is ALU. All Arithmetic and logical operations are performed in this unit. The RISC-V ALU combinations of four control inputs:

|  |  |
| --- | --- |
| ALU control lines | Function |
| 0000 | AND |
| 0001 | OR |
| 0010 | ADD |
| 0110 | SUBTRACT |

* Depending on the instruction class, the ALU will need to perform one of these four functions. For load and store instructions, we use the ALU to compute the memory address by addition. For the Rtype instructions, the ALU needs to perform one of the four actions (AND, OR, add, or subtract), depending on the value of the 7-bit funct7 field (bits 31:25) and 3-bit funct3 field (bits 14:12) in the instruction. For the conditional branch if equal instruction, the ALU subtracts two operands and tests to see if the result is 0.
* Design Code for ALU.

// Designing ALU for RISC-V(Single Cycle Processor)

`timescale 1ns/1ns

module ALU(ALU\_Result, ALU\_Control, ALU\_In1, ALU\_In2, Zero);

// port declarations.

input [31:0] ALU\_In1, ALU\_In2;

input [3:0] ALU\_Control;

output reg [31:0] ALU\_Result;

output reg Zero;

always @(\*)

begin

case(ALU\_Control)

4'b0000:begin Zero <= 0; ALU\_Result <= ALU\_In1 & ALU\_In2; end

4'b0001:begin Zero <= 0; ALU\_Result <= ALU\_In1 | ALU\_In2; end

4'b0010:begin {Zero,ALU\_Result} <= ALU\_In1 + ALU\_In2; end

4'b0110:begin {Zero,ALU\_Result} <= ALU\_In1 - ALU\_In2; end

default:begin Zero <= 0; ALU\_Result <= 0; end

endcase

end

endmodule

* Testbench Code for ALU.

// Designing ALU for RISC-V(Single Cycle Processor)

module ALU\_tb();

// port declarations.

reg [31:0] ALU\_In1, ALU\_In2;

reg [3:0] ALU\_Control;

wire [31:0] ALU\_Result;

wire Zero;

ALU uut(ALU\_Result, ALU\_Control, ALU\_In1, ALU\_In2, Zero);

initial

begin

$dumpfile("ALU.vcd");

$dumpvars(0,ALU\_tb);

#5 ALU\_Control=4'b0000;ALU\_In1=32'h00000020;ALU\_In2=32'h00000010;

#5 ALU\_Control=4'b0001;ALU\_In1=32'h00000020;ALU\_In2=32'h00000010;

#5 ALU\_Control=4'b0010;ALU\_In1=32'h00000020;ALU\_In2=32'h00000010;

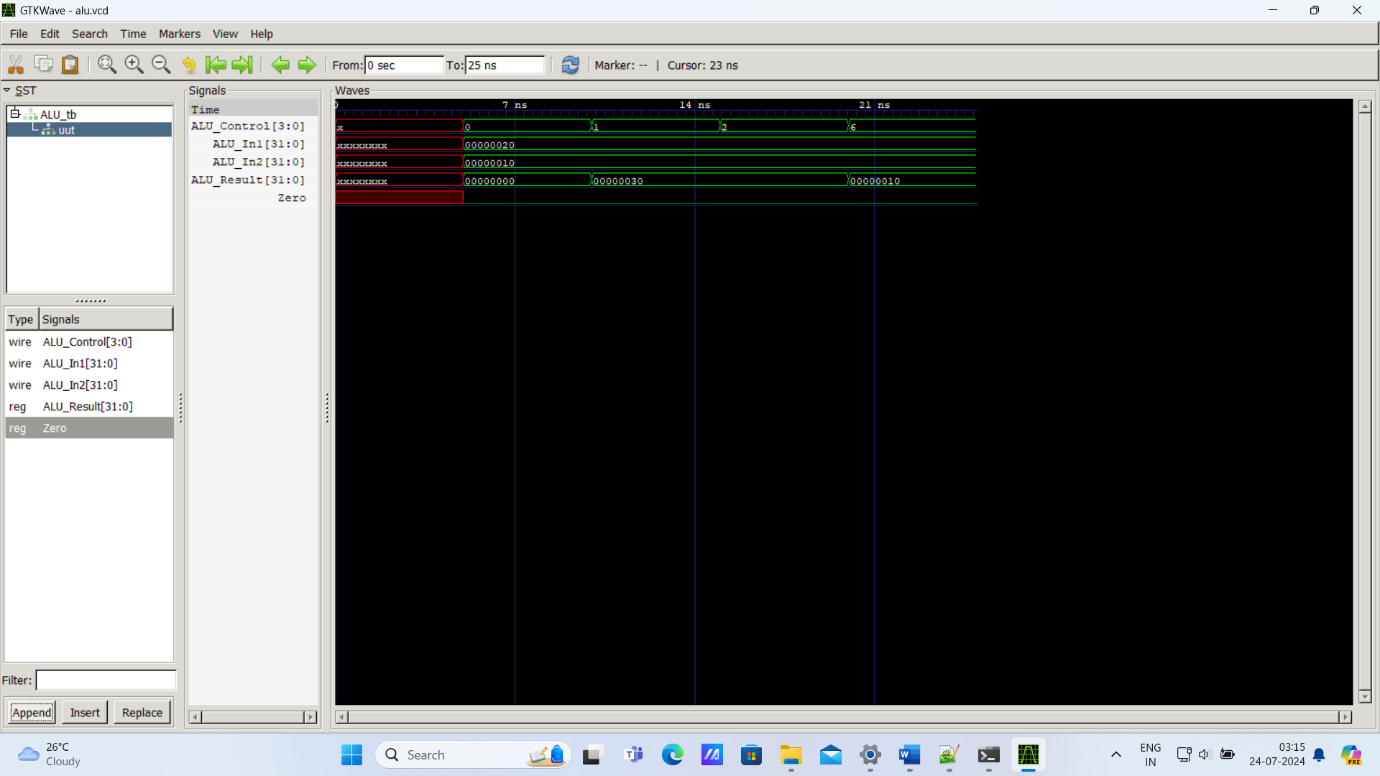
#5 ALU\_Control=4'b0110;ALU\_In1=32'h00000020;ALU\_In2=32'h00000010;

#5 $finish;

end

endmodule

* Waveform for ALU.



* We can generate the 4-bit ALU control input using a small control unit that has as inputs the funct7 and funct3 fields of the instruction and a 2-bit control field, which we call ALUOp. ALUOp indicates whether the operation to be performed should be add (00) for loads and stores, subtract and test if zero (01) for beq, or be determined by the operation encoded in the funct7 and funct3 fields (10). The output of the ALU control unit is a 4-bit signal that 500 directly controls the ALU by generating one of the 4-bit combinations.
* Design Code for ALU Control.

// Designing ALU Control for RISC-V(Single Cycle Processor)

`timescale 1ns/1ns

module ALU\_Control(ALU\_ControlOut, ALU\_OpIn, func7, func3);

// port declarations.

input [1:0] ALU\_OpIn;

input [31:25] func7;

input [14:12] func3;

output reg [3:0] ALU\_ControlOut;

always @(\*)

begin

case({ALU\_OpIn, func7, func3})

12'b00\_0000000\_000:ALU\_ControlOut=4'b0010;

12'b01\_0000000\_000:ALU\_ControlOut=4'b0110;

12'b10\_0000000\_000:ALU\_ControlOut=4'b0010;

12'b10\_0100000\_000:ALU\_ControlOut=4'b0110;

12'b10\_0000000\_111:ALU\_ControlOut=4'b0000;

12'b10\_0000000\_110:ALU\_ControlOut=4'b0001;

default:ALU\_ControlOut=4'b1111;

endcase

end

endmodule

* Testbench Code for ALU Control.

// TestBench for ALU Control for RISC-V(Single Cycle Processor)

module ALU\_Control\_tb();

// port declarations.

reg [1:0] ALU\_OpIn;

reg [31:25] func7;

reg [14:12] func3;

wire [3:0] ALU\_ControlOut;

ALU\_Control uut(ALU\_ControlOut, ALU\_OpIn, func7, func3);

initial

begin

$dumpfile("ALU\_Control.vcd");

$dumpvars(0,ALU\_Control\_tb);

$monitor($time , " ALU\_OpIn=%b, func7=%b, func3=%b, ALU\_ControlOut=%b ", ALU\_OpIn, func7, func3, ALU\_ControlOut);

#5 ALU\_OpIn=2'b00; func7=7'b0000000; func3=3'b000;

#5 ALU\_OpIn=2'b01; func7=7'b0000000; func3=3'b000;

#5 ALU\_OpIn=2'b10; func7=7'b0000000; func3=3'b000;

#5 ALU\_OpIn=2'b10; func7=7'b0100000; func3=3'b000;

#5 ALU\_OpIn=2'b10; func7=7'b0000000; func3=3'b111;

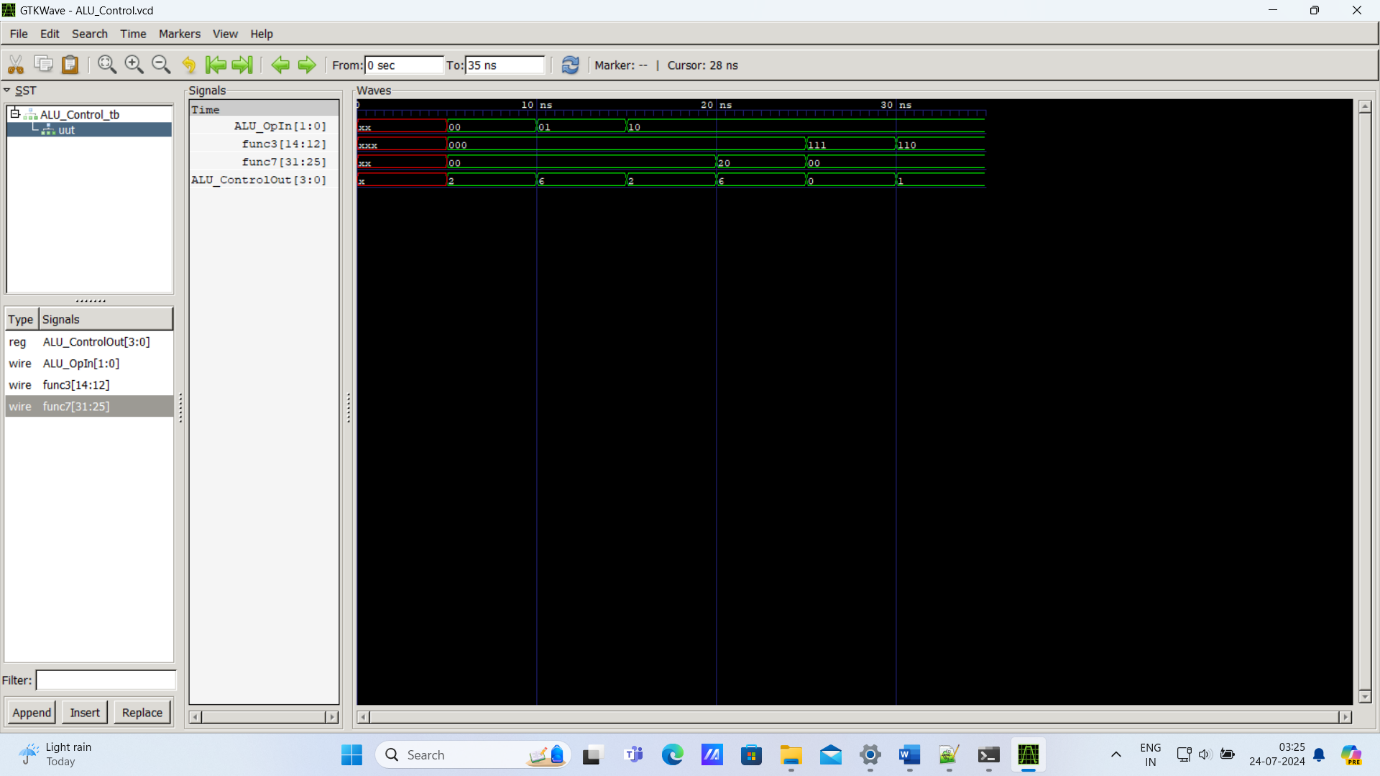
#5 ALU\_OpIn=2'b10; func7=7'b0000000; func3=3'b110;

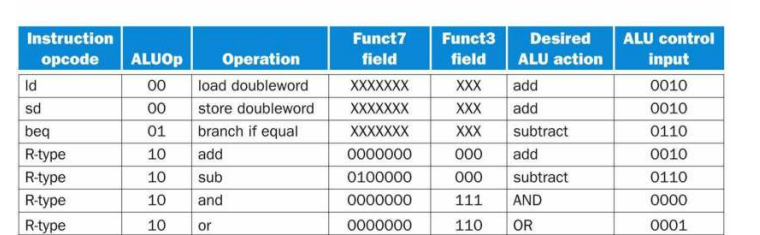
#5 $finish;

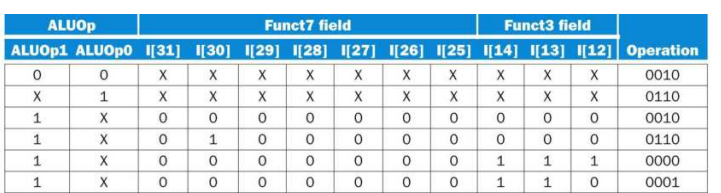
end

endmodule

* Waveform for ALU Control.







* The eighth Datapath element we need is Control Unit.
* Design Code for Control Unit.

// Designing Control Unit for RISC-V(Single Cycle Processor)

`timescale 1ns/1ns

module Control\_Unit(Instruction\_Opcode, Branch, MemRead, MemtoReg, ALU\_OpOut, MemWrite, ALUSrc, RegWrite, reset);

// port declarations.

input reset;

input [6:0] Instruction\_Opcode;

output reg Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite;

output reg [1:0] ALU\_OpOut;

always @(\*)

begin

if(reset)

begin

{ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, ALU\_OpOut} <= 7'b0000000;

end

else

begin

case(Instruction\_Opcode)

7'b0110011:begin // for R type instructions.

ALUSrc<=0;

MemtoReg<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

Branch<=0;

ALU\_OpOut[1]<=1;

ALU\_OpOut[0]<=0;

end

7'b0000011:begin // for ld type instructions.

ALUSrc<=1;

MemtoReg<=1;

RegWrite<=1;

MemRead<=1;

MemWrite<=0;

Branch<=0;

ALU\_OpOut[1]<=0;

ALU\_OpOut[0]<=0;

end

7'b0100011:begin // for sd type instructions.

ALUSrc<=1;

MemtoReg<=1;

RegWrite<=0;

MemRead<=0;

MemWrite<=1;

Branch<=0;

ALU\_OpOut[1]<=0;

ALU\_OpOut[0]<=0;

end

7'b1100011:begin // for beq type instructions.

ALUSrc<=0;

MemtoReg<=1;

RegWrite<=0;

MemRead<=0;

MemWrite<=0;

Branch<=1;

ALU\_OpOut[1]<=0;

ALU\_OpOut[0]<=1;

end

default:begin // for R type instructions.

ALUSrc<=0;

MemtoReg<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

Branch<=0;

ALU\_OpOut[1]<=1;

ALU\_OpOut[0]<=0;

end

endcase

end

end

endmodule

* Testbench Code for Control Unit.

// Designing Testbench for Control Unit for RISC-V(Single Cycle Processor)

module Control\_Unit\_tb();

// port declarations.

reg reset;

reg [6:0] Instruction\_Opcode;

wire Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite;

wire [1:0] ALU\_OpOut;

//Module instantiations

Control\_Unit uut(Instruction\_Opcode, Branch, MemRead, MemtoReg, ALU\_OpOut, MemWrite, ALUSrc, RegWrite, reset);

initial

begin

$dumpfile("Control\_Unit.vcd");

$dumpvars(0,Control\_Unit\_tb);

$monitor($time," reset=%b, Instruction\_Opcode=%b, Branch=%b, MemRead=%b, MemtoReg=%b, ALU\_OpOut=%b, MemWrite=%b, ALUSrc=%b, RegWrite=%b",

reset, Instruction\_Opcode, Branch, MemRead, MemtoReg, ALU\_OpOut, MemWrite, ALUSrc, RegWrite);

reset=1'b1;

#5 reset=1'b0;

#5 Instruction\_Opcode=7'b0110011;

#5 Instruction\_Opcode=7'b0000011;

#5 Instruction\_Opcode=7'b0100011;

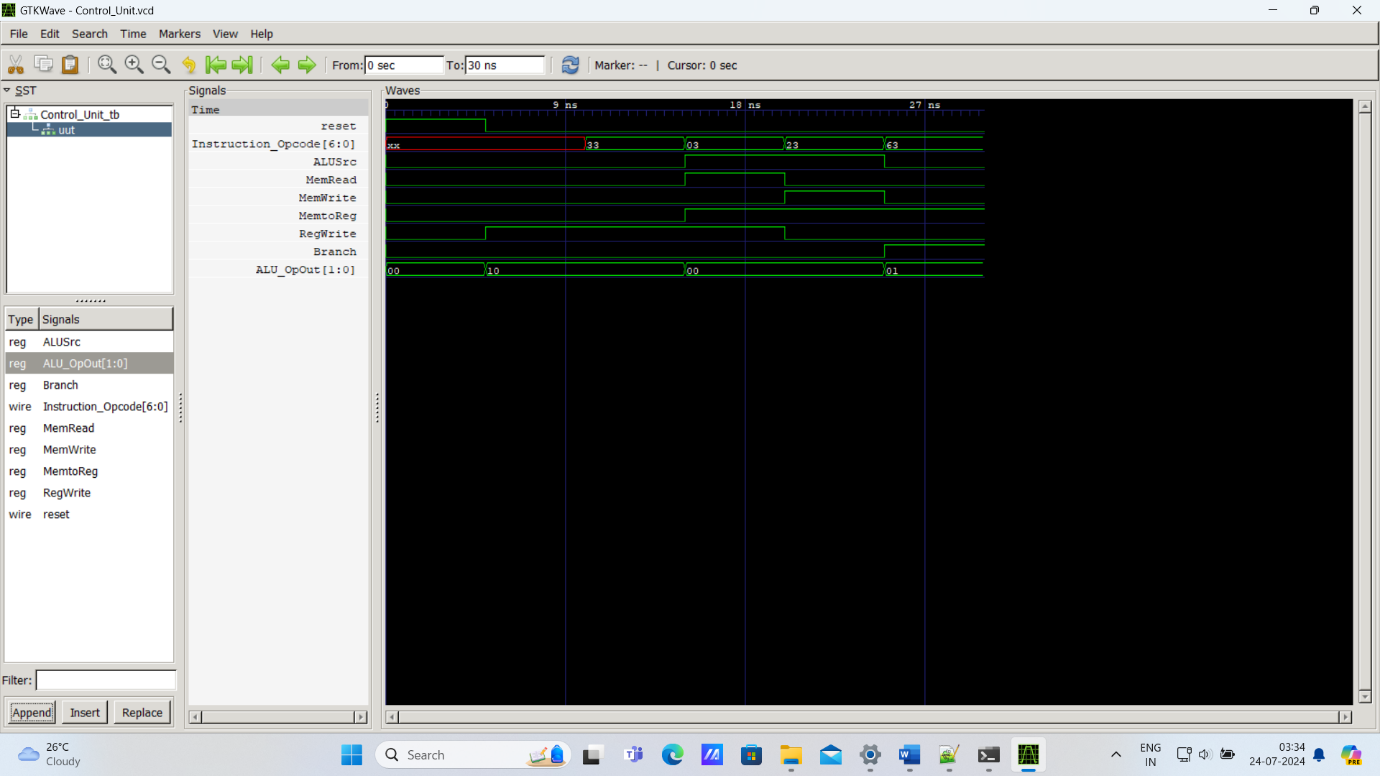
#5 Instruction\_Opcode=7'b1100011;

#5 $finish;

end

endmodule

* Waveform for Control Unit.



* Design Code for mux.

module mux2\_1(out, in0, in1, sel);

// port declarations

input [31:0] in0, in1;

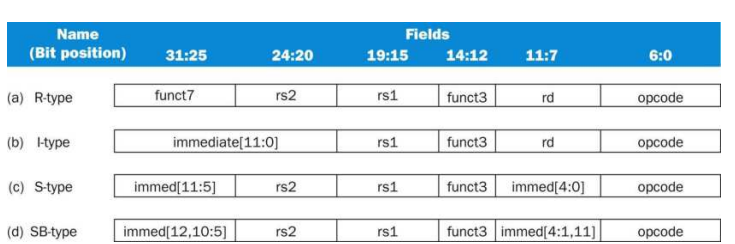
input sel;

output [31:0] out;

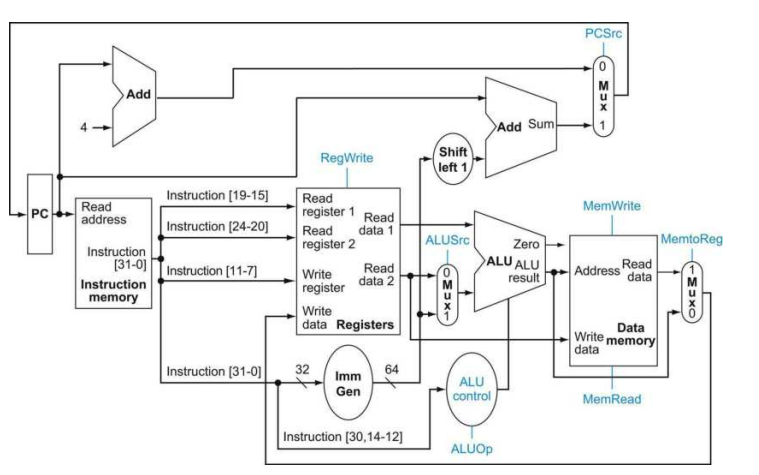
assign out = sel ? in1 : in0;

endmodule

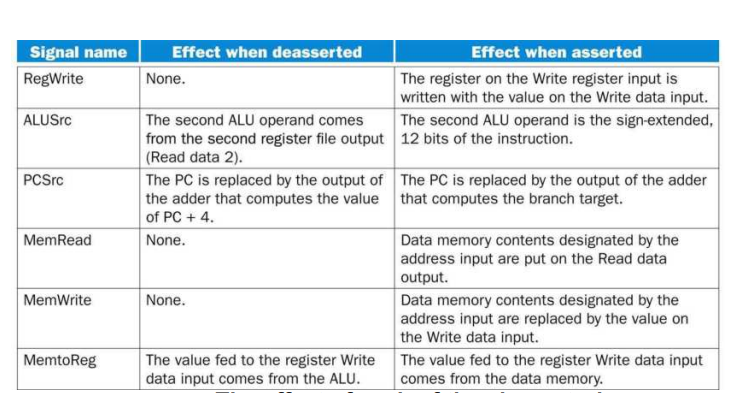
* Instruction Format in RISC-V:



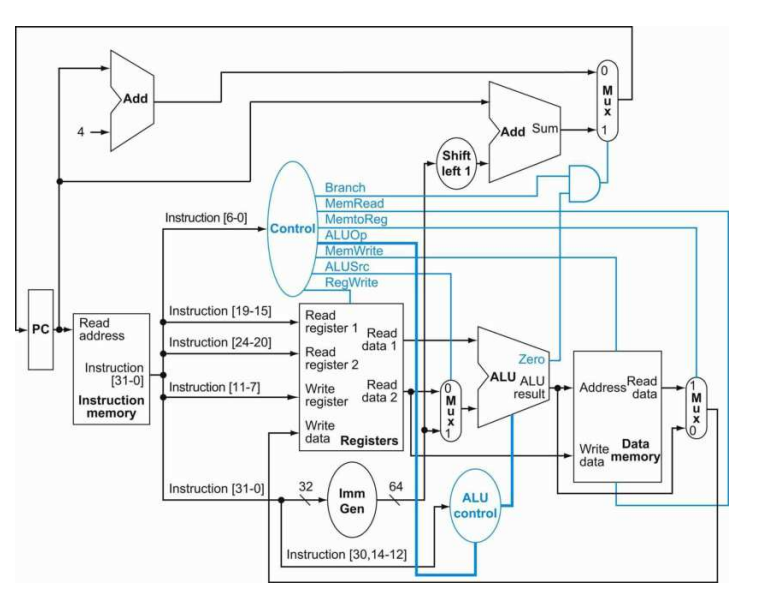
* Instruction format for R-type arithmetic instructions (opcode = 51ten), which have three register operands: rs1, rs2, and rd. Fields rs1 and rd are sources, and rd is the destination. The ALU function is in the funct3 and funct7 fields and is decoded by the ALU control design in the previous section. The R-type instructions that we implement are add, sub, and, and or.
* Instruction format for I-type load instructions (opcode = 3ten). The register rs1 is the base register that is added to the 12-bit immediate field to form the memory address. Field rd is the destination register for the loaded value.
* Instruction format for S-type store instructions (opcode = 35ten). The register rs1 is the base register that is added to the 12-bit immediate field to form the memory address. (The immediate field is split into a 7-bit piece and a 5-bit piece.) Field rs2 is the source register whose value should be stored into memory.
* Instruction format for SB-type conditional branch instructions (opcode =99ten). The registers rs1 and rs2 compared. The 12-bit immediate address field is sign-extended, shifted left 1 bit, and added to the PC to compute the branch target address.
* Opcode is the field that denotes the operation and format of an instruction.
* The opcode field, which as we saw in, is always in bits 6:0. Depending on the opcode, the funct3 field (bits 14:12) and funct7 field (bits 31:25) serve as an extended opcode field.
* The first register operand is always in bit positions 19:15 (rs1) for R-type instructions and branch instructions. This field also specifies the base register for load and store instructions.
* The second register operand is always in bit positions 24:20 (rs2) for R-type instructions and branch instructions. This field also specifies the register operand that gets copied to memory for store instructions.
* Another operand can also be a 12-bit offset for branch or loadstore instructions.
* The destination register is always in bit positions 11:7 (rd) for R-type instructions and load instructions.



* The control lines are shown in color. The six single-bit control lines plus the 2-bit ALUOp control signal.
* Working of different control signal:



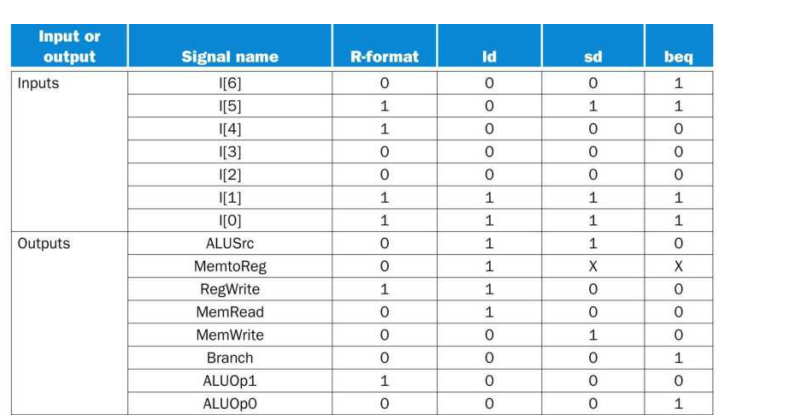
* Final Datapath with control unit:



* The control signals should be set for each opcode:



* The first row of the table corresponds to the R-format instructions (add, sub, and, and or). For all these instructions, the source register fields are rs1 and rs2, and the destination register field is rd; this defines how the signals ALUSrc is set. Furthermore, an R-type instruction writes a register (RegWrite =1), but neither reads nor writes data memory. When the Branch control signal is 0, the PC is unconditionally replaced with PC +4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high.
* The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct fields. The second and third rows of this table give the control signal settings for ld and sd. These ALUSrc and ALUOp fields are set to perform the address calculation. The MemRead and MemWrite are set to perform the memory access. Finally, RegWrite is set for a load to cause the result to be stored in the rd register. The ALUOp field for branch is set for subtract (ALU control =01), which is used to test for equality.
* Notice that the MemtoReg field is irrelevant when the RegWrite signal is 0: since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don’t care.



* Instruction Execution in this RISC processor.
* The operation of the datapath for an R-type instruction, such as add x1, x2, x3. Although everything occurs in one clock cycle, we can think of four steps to execute the instruction; these steps are ordered by the flow of information:
* 1. The instruction is fetched, and the PC is incremented.
* 2. Two registers, x2 and x3, are read from the register file; also, the main control unit computes the setting of the control lines during this step.
* 3. The ALU operates on the data read from the register file, using portions of the opcode to generate the ALU function.
* 4. The result from the ALU is written into the destination register (x1) in the register file.
* The execution of a load register, such as ld x1, offset(x2).
* We can think of a load instruction as operating in five steps (similar to how the Rtype executed in four):
* 1. An instruction is fetched from the instruction memory, and the PC is incremented.
* 2. A register (x2) value is read from the register file.
* 3. The ALU computes the sum of the value read from the register file and the sign-extended 12 bits of the instruction (offset).
* 4. The sum from the ALU is used as the address for the data memory.
* 5. The data from the memory unit is written into the register file (x1).
* The operation of the branch-if-equal instruction, such as beq x1, x2, offset, in the same fashion. It operates much like an R-format instruction, but the ALU output is used to determine whether the PC is written with PC +4 or the branch target address.
* The four steps in execution:
* 1. An instruction is fetched from the instruction memory, and the PC is incremented.
* 2. Two registers, x1 and x2, are read from the register file.
* 3. The ALU subtracts one data value from the other data value, both read from the register file. The value of PC is added to the signextended, 12 bits of the instruction (offset) left shifted by one; the result is the branch target address.
* 4. The Zero status information from the ALU is used to decide which adder result to store in the PC.
* Final Design Code for Top Module of RISC V :

// Designing Program Counter(PC) for RISC-V(Single Cycle Processor)

`timescale 1ns/1ns

module Program\_Counter(PC\_OUT, PC\_IN, clk, reset);

// port declarations.

input clk, reset;

input [31:0]PC\_IN;

output reg [31:0]PC\_OUT;

always @(posedge clk or posedge reset) //PC and reset is triggered at positive edge of clock.

begin

if(reset)

PC\_OUT <= 32'h00000000; //reset is high initialize PC\_OUT with zero.

else

PC\_OUT <= PC\_IN;

end

endmodule

// Designing Program Counter(PC) adder for next Instruction for RISC-V(Single Cycle Processor)

module Program\_Counter\_Next(PC\_OUT\_NEXT, PC\_IN\_NEXT);

// port declarations.

input [31:0] PC\_IN\_NEXT;

output [31:0] PC\_OUT\_NEXT;

assign PC\_OUT\_NEXT = PC\_IN\_NEXT + 32'h00000004;

endmodule

// Designing Branch target for RISC-V(Single Cycle Processor)

module Branch\_Target(Branch\_target, Imm\_Gen\_out, PC\_from\_Next);

// port declarations.

input [31:0] Imm\_Gen\_out;

input [31:0] PC\_from\_Next;

output [31:0] Branch\_target;

assign Branch\_target = (Imm\_Gen\_out << 1) + PC\_from\_Next;

endmodule

// Designing Instruction Memory for RISC-V(Single Cycle Processor)

module Instruction\_Memory(Instruction\_out, read\_address, clk, reset);

// port declarations.

input clk, reset;

input [31:0]read\_address; //Reading Instruction Address sent PC.

output [31:0]Instruction\_out;

//Making Memory

reg [31:0] InstructionMemory [31:0]; //considering PC 32-bit, memory consists of 31 cells each of 32 bit wide.

assign Instruction\_out = InstructionMemory[read\_address]; //Assigning Memory value.

integer i;

always @(posedge clk)

begin

if(reset == 1'b1)

begin

for(i = 0; i < 32;i = i+1)

InstructionMemory[i] = 32'h00000000; //Initialize when reset.

end

else if(reset == 1'b0)

begin

InstructionMemory[4] = 32'b0000000\_00010\_00001\_000\_00001\_0110011; // add R1, R1, R2

InstructionMemory[8] = 32'b0100000\_00010\_00001\_000\_00001\_0110011; // sub R1, R1, R2

InstructionMemory[12] = 32'b0000000\_00010\_00001\_111\_00001\_0110011; // and R1, R1, R2

InstructionMemory[16] = 32'b0000000\_00010\_00001\_110\_00001\_0110011; // or R1, R1, R2

InstructionMemory[20] = 32'b000000000111\_00010\_000\_00001\_0000011; // ld R1, offset(R2)

InstructionMemory[24] = 32'b0000000\_00001\_00010\_000\_00011\_0100011; // sd R1, offset(R2)

InstructionMemory[28] = 32'b0000000\_00010\_00001\_000\_11111\_1100011; // beq R1, R2, offset

end

end

endmodule

// Designing Register File for RISC-V(Single Cycle Processor)

module Register\_File(Read\_data1, Read\_data2, Write\_data, Read\_register1,

Read\_register2, Write\_register, RegWrite, clk, reset);

// port declarations.

input RegWrite, clk, reset;

input [19:15] Read\_register1;

input [24:20] Read\_register2;

input [11:7] Write\_register;

input [31:0] Write\_data;

output [31:0] Read\_data1, Read\_data2;

//Making Register File

reg [31:0] Registers [31:0]; // 32 Registers each of 32-bit wide.

integer i;

initial

begin

for(i = 0; i < 32; i = i + 1)

begin

Registers[i] = i\*10;

end

end

always @(posedge clk) // Assigning values with write control signal.

begin

if(reset == 1'b1)

begin

for(i=0; i<32; i=i+1)

begin

Registers[i] = 32'b0;

end

end

else if(RegWrite == 1'b1)

begin

Registers[Write\_register] = Write\_data;

end

end

assign Read\_data1 = Registers[Read\_register1]; // Assigning values with no read control signal.

assign Read\_data2 = Registers[Read\_register2]; // Assigning values with no read control signal.

endmodule

// Designing Immediate Generator for RISC-V(Single Cycle Processor)

module Immediate\_Generator(Imm\_Gen, Instruction, Opcode);

// port declarations.

input [6:0] Opcode;

input [31:0] Instruction;

output reg [31:0] Imm\_Gen;

always @(\*)

begin

case(Opcode)

7'b0000011:Imm\_Gen <= {{20{Instruction[31]}}, Instruction[31:20]}; // ld type Instruction.

7'b0100011:Imm\_Gen <= {{20{Instruction[31]}}, Instruction[31:25], Instruction[11:7]}; // sd type Instruction.

7'b1100011:Imm\_Gen <= {{20{Instruction[31]}}, Instruction[31], Instruction[7], Instruction[30:25], Instruction[11:8]}; // beq type Instruction.

default:Imm\_Gen <= 32'h00000000;

endcase

end

endmodule

// Designing Control Unit for RISC-V(Single Cycle Processor)

module Control\_Unit(Instruction\_Opcode, Branch, MemRead, MemtoReg, ALU\_OpOut, MemWrite, ALUSrc, RegWrite, reset);

// port declarations.

input reset;

input [6:0] Instruction\_Opcode;

output reg Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite;

output reg [1:0] ALU\_OpOut;

always @(\*)

begin

if(reset)

begin

{ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, ALU\_OpOut} <= 7'b0000000;

end

else

begin

case(Instruction\_Opcode)

7'b0110011:begin // for R type instructions.

ALUSrc<=0;

MemtoReg<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

Branch<=0;

ALU\_OpOut[1]<=1;

ALU\_OpOut[0]<=0;

end

7'b0000011:begin // for ld type instructions.

ALUSrc<=1;

MemtoReg<=1;

RegWrite<=1;

MemRead<=1;

MemWrite<=0;

Branch<=0;

ALU\_OpOut[1]<=0;

ALU\_OpOut[0]<=0;

end

7'b0100011:begin // for sd type instructions.

ALUSrc<=1;

MemtoReg<=1;

RegWrite<=0;

MemRead<=0;

MemWrite<=1;

Branch<=0;

ALU\_OpOut[1]<=0;

ALU\_OpOut[0]<=0;

end

7'b1100011:begin // for beq type instructions.

ALUSrc<=0;

MemtoReg<=1;

RegWrite<=0;

MemRead<=0;

MemWrite<=0;

Branch<=1;

ALU\_OpOut[1]<=0;

ALU\_OpOut[0]<=1;

end

default:begin // for R type instructions.

ALUSrc<=0;

MemtoReg<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

Branch<=0;

ALU\_OpOut[1]<=1;

ALU\_OpOut[0]<=0;

end

endcase

end

end

endmodule

// Designing ALU for RISC-V(Single Cycle Processor)

module ALU(ALU\_Result, ALU\_Control, ALU\_In1, ALU\_In2, Zero);

// port declarations.

input [31:0] ALU\_In1, ALU\_In2;

input [3:0] ALU\_Control;

output reg [31:0] ALU\_Result;

output reg Zero;

always @(\*)

begin

case(ALU\_Control)

4'b0000:begin Zero <= 0; ALU\_Result <= ALU\_In1 & ALU\_In2; end

4'b0001:begin Zero <= 0; ALU\_Result <= ALU\_In1 | ALU\_In2; end

4'b0010:begin {Zero,ALU\_Result} <= ALU\_In1 + ALU\_In2; end

4'b0110:begin {Zero,ALU\_Result} <= ALU\_In1 - ALU\_In2; end

default:begin Zero <= 0; ALU\_Result <= 0; end

endcase

end

endmodule

// Designing ALU Control for RISC-V(Single Cycle Processor)

module ALU\_Control(ALU\_ControlOut, ALU\_OpIn, func7, func3);

// port declarations.

input [1:0] ALU\_OpIn;

input [31:25] func7;

input [14:12] func3;

output reg [3:0] ALU\_ControlOut;

always @(\*)

begin

case({ALU\_OpIn, func7, func3})

12'b00\_0000000\_000:ALU\_ControlOut=4'b0010;

12'b01\_0000000\_000:ALU\_ControlOut=4'b0110;

12'b10\_0000000\_000:ALU\_ControlOut=4'b0010;

12'b10\_0100000\_000:ALU\_ControlOut=4'b0110;

12'b10\_0000000\_111:ALU\_ControlOut=4'b0000;

12'b10\_0000000\_110:ALU\_ControlOut=4'b0001;

default:ALU\_ControlOut=4'b1111;

endcase

end

endmodule

// Designing Data Memory for RISC-V(Single Cycle Processor)

module Data\_Memory(Read\_data, Write\_data, MemWrite, MemRead, address, clk, reset);

// port declarations.

input MemWrite, MemRead, clk, reset;

input [31:0] Write\_data, address;

output [31:0] Read\_data;

//Making Memory

reg [31:0] DataMemory [31:0];

//considering address 32-bit, data memory consists of 32 cells each of 32 bit wide.

assign Read\_data= (MemRead) ? DataMemory[address] : 32'h00000000; //Assigning DataMemory value.

integer i;

always @(posedge clk or posedge reset)

begin

if(reset == 1'b1)

begin

for(i=0;i<32;i=i+1)

begin

DataMemory[i] = i\*20; //Initialize when reset.

end

end

else if(MemWrite)

begin

DataMemory[address]=Write\_data;

end

end

endmodule

// Designing mux for RISC-V(Single Cycle Processor)

module mux2\_1(out, in0, in1, sel);

// port declarations

input [31:0] in0, in1;

input sel;

output [31:0] out;

assign out = sel ? in1 : in0;

endmodule

// Designing and gate for RISC-V(Single Cycle Processor)

module and\_gate(out, in0, in1);

// port declarations

input in0, in1;

output out;

assign out = in0 & in1;

endmodule

// Designing Top for RISC-V(Single Cycle Processor)

module Top(clk, reset);

// port declarations.

input clk, reset;

// internal wire declarations.

wire [31:0] PC\_Top; //wire connect the o/p of PC to PC adder.

wire [31:0] PC\_Next\_Top; //wire connect the o/p of PC\_OUT\_NEXT to PC\_IN.

wire [31:0] Instruction\_out\_Top; //wire connect o/p of Instruction\_Memory to i/p Register\_File.

wire [31:0] Read\_data1\_Top; //wire connect o/p of Read\_data1 to i/p of ALU.

wire [31:0] Read\_data2\_Top; //wire connect o/p of Read\_data2 to i/p of mux1.

wire [31:0] mux1ToAlu; //wire connect o/p of mux1 to i/p of ALU.

wire [3:0] ALU\_ControlOut\_Top;//wire connect o/p of ALU\_Control to i/p of ALU.

wire [31:0] ALU\_Result\_Top;//wire connect o/p of ALU to i/p of Data\_Memory.

wire [31:0] Read\_data\_Top;//wire connect o/p of Data\_Memory to i/p of mux2.

wire [31:0] mux2ToAlu;//wire connect o/p of mux2 to i/p of Register\_File.

wire [31:0] Imm\_Gen\_Top;//wire connect o/p of Immediate\_Generator to i/p of Branch\_Target.

wire [31:0] Branch\_target\_Top;//wire connect o/p of Branch\_target to i/p of mux3.

wire [31:0] mux3ToPc;//wire connect o/p of mux3 to i/p of Program\_Counter.

wire Branch\_control\_top, Zero\_Top, Branch\_Top, RegWrite\_Top, ALUSrc\_Top, MemWrite\_Top, MemRead\_Top, MemtoReg\_Top;//wire connect o/p of Control\_Unit.

wire [1:0] ALU\_OpOut\_Top;

// Program\_Counter\_Next instantiations.

Program\_Counter\_Next Program\_Counter\_Next\_Top(.PC\_OUT\_NEXT(PC\_Next\_Top), .PC\_IN\_NEXT(PC\_Top));

//Program\_Counter instantiations.

Program\_Counter Program\_Counter\_Top(.PC\_OUT(PC\_Top), .PC\_IN(mux3ToPc), .clk(clk), .reset(reset));

//Instruction\_Memory instantiations.

Instruction\_Memory Instruction\_Memory\_Top(.Instruction\_out(Instruction\_out\_Top), .read\_address(PC\_Top), .clk(clk), .reset(reset));

//Register\_File instantiations.

Register\_File Register\_File\_Top(.Read\_data1(Read\_data1\_Top), .Read\_data2(Read\_data2\_Top), .Write\_data(mux2ToAlu), .Read\_register1(Instruction\_out\_Top[19:15]),

.Read\_register2(Instruction\_out\_Top[24:20]), .Write\_register(Instruction\_out\_Top[11:7]), .RegWrite(RegWrite\_Top), .clk(clk), .reset(reset));

//ALU instantiations.

ALU ALU\_Top(.ALU\_Result(ALU\_Result\_Top), .ALU\_Control(ALU\_ControlOut\_Top), .ALU\_In1(Read\_data1\_Top), .ALU\_In2(mux1ToAlu), .Zero(Zero\_Top));

//mux\_1 instantiations.

mux2\_1 mux2\_1\_1\_Top(.out(mux1ToAlu), .in0(Read\_data2\_Top), .in1(Imm\_Gen\_Top), .sel(ALUSrc\_Top));

//ALU\_Control instantiations.

ALU\_Control ALU\_Control\_Top(.ALU\_ControlOut(ALU\_ControlOut\_Top), .ALU\_OpIn(ALU\_OpOut\_Top), .func7(Instruction\_out\_Top[31:25]), .func3(Instruction\_out\_Top[14:12]));

//Data\_Memory instantiations.

Data\_Memory Data\_Memory\_Top(.Read\_data(Read\_data\_Top), .Write\_data(Read\_data2\_Top), .MemWrite(MemWrite\_Top), .MemRead(MemRead\_Top), .address(ALU\_Result\_Top), .clk(clk), .reset(reset));

//mux\_2 instantiations.

mux2\_1 mux2\_1\_2\_Top(.out(mux2ToAlu), .in0(ALU\_Result\_Top), .in1(Read\_data\_Top), .sel(MemtoReg\_Top));

//Control\_Unit instantiations.

Control\_Unit Control\_Unit\_Top(.Instruction\_Opcode(Instruction\_out\_Top[6:0]), .Branch(Branch\_Top), .MemRead(MemRead\_Top), .MemtoReg(MemtoReg\_Top), .ALU\_OpOut(ALU\_OpOut\_Top), .MemWrite(MemWrite\_Top), .ALUSrc(ALUSrc\_Top), .RegWrite(RegWrite\_Top), .reset(reset));

//Immediate\_Generator instantiations.

Immediate\_Generator Immediate\_Generator\_Top(.Imm\_Gen(Imm\_Gen\_Top), .Instruction(Instruction\_out\_Top), .Opcode(Instruction\_out\_Top[6:0]));

//Branch\_Target instantiations.

Branch\_Target Branch\_Target\_Top(.Branch\_target(Branch\_target\_Top), .Imm\_Gen\_out(Imm\_Gen\_Top), .PC\_from\_Next(PC\_Top));

//mux\_3 instantiations.

mux2\_1 mux2\_1\_3\_Top(.out(mux3ToPc), .in0(PC\_Next\_Top), .in1(Branch\_target\_Top), .sel(Branch\_control\_top));

//and gate instantiations.

and\_gate and\_gate\_Top(.out(Branch\_control\_top), .in0(Branch\_Top), .in1(Zero\_Top));

endmodule

* Final Testbench Code for Top Module of RISC V:

module Top\_tb();

// port declarations.

reg clk, reset;

Top uut(clk, reset);

//clock generations.

initial

begin

clk=0;

end

always #20 clk = ~clk;

//reset generations.

initial

begin

reset=1'b1;

#10 reset=1'b0;

end

initial

begin

$dumpfile("Top.vcd");

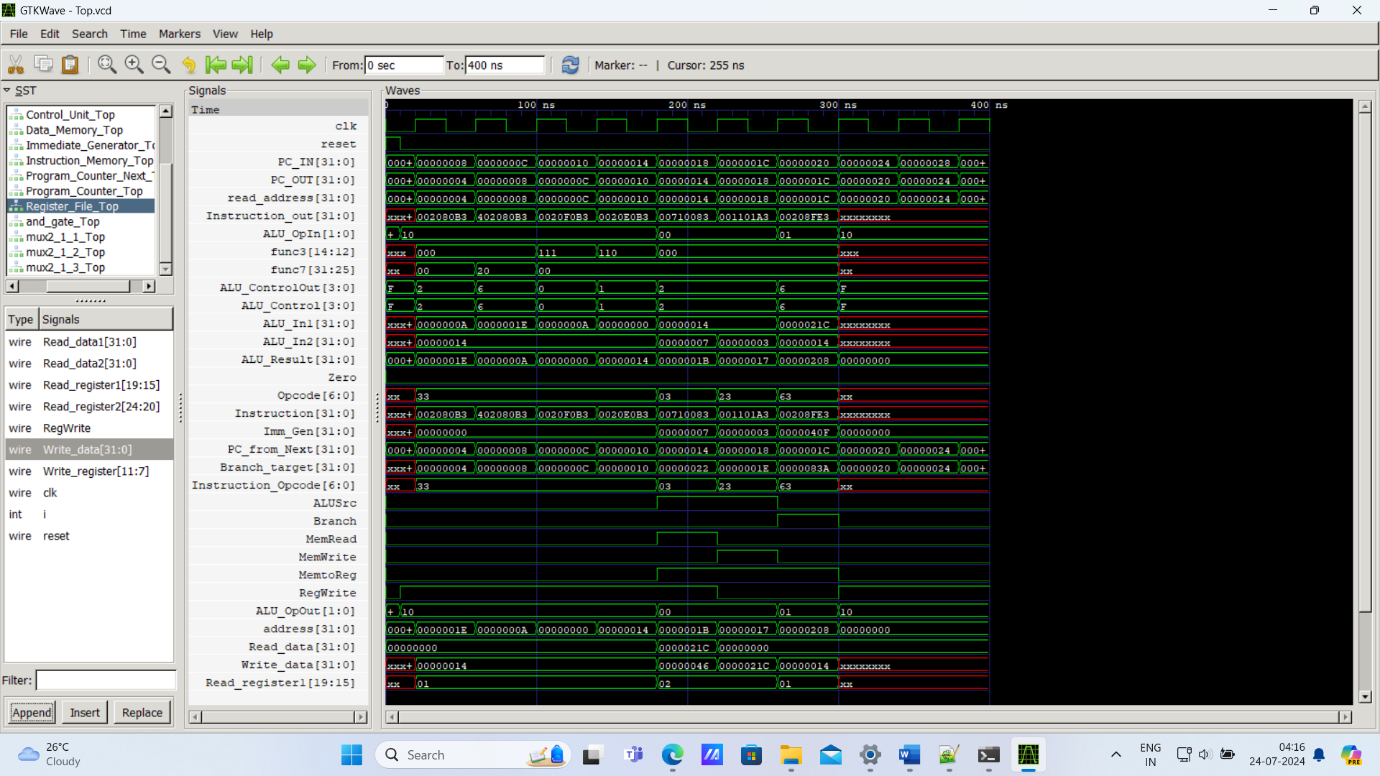
$dumpvars(0,Top\_tb);

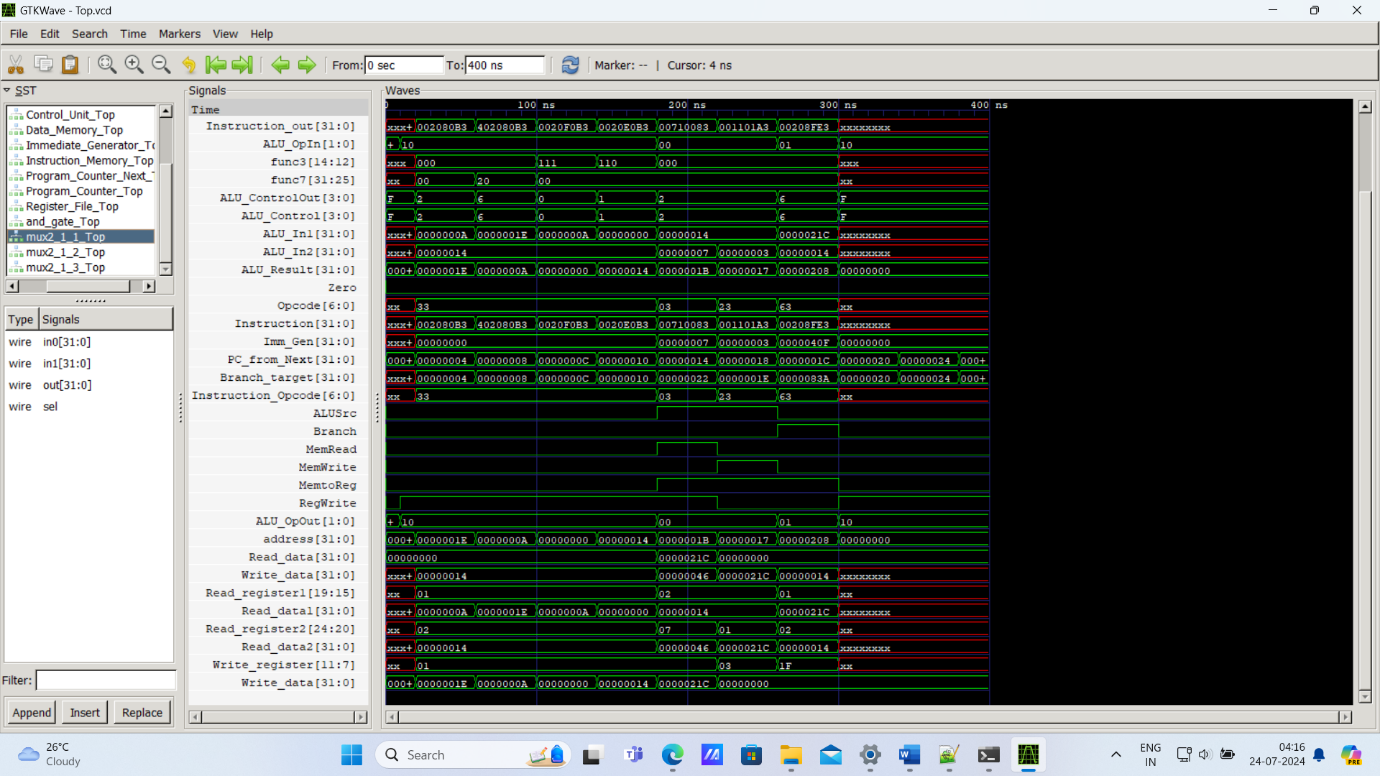
#400 $finish;

end

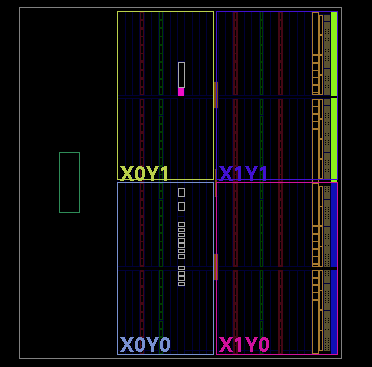
endmodule

* Waveform for Final Top Module of RISC V:





* Synthesized in AMD Xilinx Vivado.



* References:
* YouTube channel link:
* <https://www.youtube.com/watch?v=LsC2LdgxHyQ> Part 1.
* <https://www.youtube.com/watch?v=TUzbl0cy310&t=1s> Part 2.
* Computer Organization and Design, The Hardware/Software Interface: RISC-V Edition By

David A. Patterson and John L. Hennessy. (book).

* In this project, I Designed 32‐bit RISC‐V Processor in Verilog HDL.
* Verified its performance and timing analysis.
* Used AMD Xilinx Vivado and Icarus Verilog with GTKWave to model the hardware implementation of the RISC‐V Processor comprising of the data path (which includes the Program Counter, Instruction Memory, Register File, ALU, ALU Control, Control Unit, Immediate Generation, PC and Branch Adder.
* Report and Project Submitted By
* Name: Subham Mahankud
* Roll No: 121EI0400