Non-pipelined Single Stage(Cycle) CPU Design

EN3021 Digital System Design

Malanban K.
200373X
Electronics and Telecommunication Engineering
University of Moratuwa
SriLanka
15th October 2023

Contents

1 Introduction

The term "RV32I" refers to a specific instruction set architecture (ISA) in the RISC-V family. RISC-V is an open, scalable ISA that can be used in a variety of applications, from small embedded systems to high-performance computing. "RV32I" is one of the base instruction set architectures within RISC-V, characterized by the following features:

32-bit Data Width: The "32" in "RV32I" indicates that the processor uses 32-bit data and address widths. This is a common choice for many embedded and general-purpose computing applications.

RISC (Reduced Instruction Set Computer): RISC-V follows the principles of RISC design, which emphasizes a simple and streamlined set of instructions. The RV32I instruction set is relatively compact and efficient.

Base Integer Instructions: The "I" in "RV32I" signifies that this is the base integer instruction set. RV32I includes a set of basic integer instructions for tasks like data movement, arithmetic, logic, branching, and memory access. These instructions provide the foundation for general-purpose computing.

User-Level ISA: RV32I is primarily designed for user-level application software. It provides the necessary instructions for running programs, and more specialized instructions may be added through extensions for specific application domains.

Extensible: RISC-V is an extensible ISA, which means that you can add additional instruction sets (extensions) to RV32I as needed for specific applications. For example, the RV32IM extension adds hardware support for integer multiplication and division.

Open and Standardized: RISC-V is an open architecture, meaning that it is not owned by a single company, and it has been standardized, making it accessible for both commercial and open-source implementations.

RV32I is a foundational ISA that can be used as a starting point for building processors for various purposes. Depending on the application, additional instruction set extensions can be added to support specific functionalities like floating-point arithmetic, vector processing, cryptography, and more. This flexibility is one of the key advantages of RISC-V architecture.

1. The opcode field, 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.

2. 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.

3. 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.

4. Another operand can also be a 12-bit offset for branch or load store instructions.

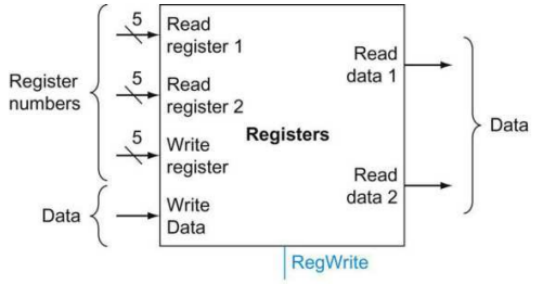
5. The destination register is always in bit positions 11:7 (rd) for R type instructions and load instruction.

2 Processor Design

2.1 Register

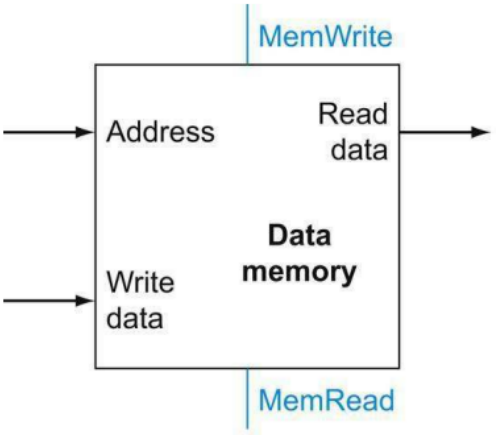
A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed. A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file. R-format instructions have three register operands, so we will need to 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. The register file always outputs the contents of whatever register numbers are on the Read register inputs. Writes, however, are controlled by the write control signal, which must be asserted for a write to occur at the clock edge. This is an word addressable register.



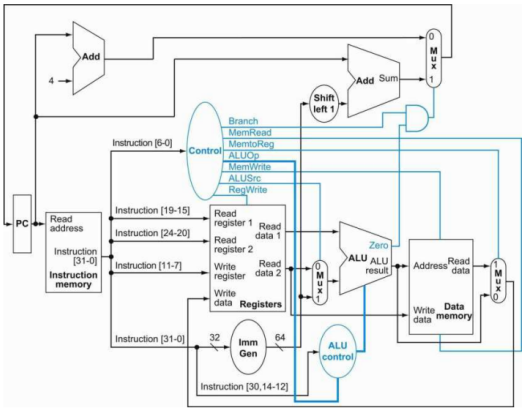
2.2 Memory

The 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. The memory unit needs a read signal, unlike the register file, reading the value of an invalid address can cause problems. This is an word addressable memory.

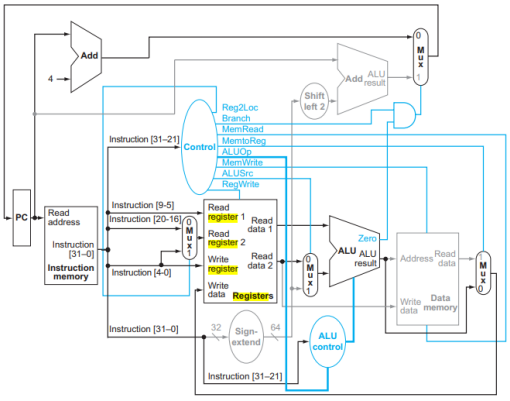


2.3 Datapath

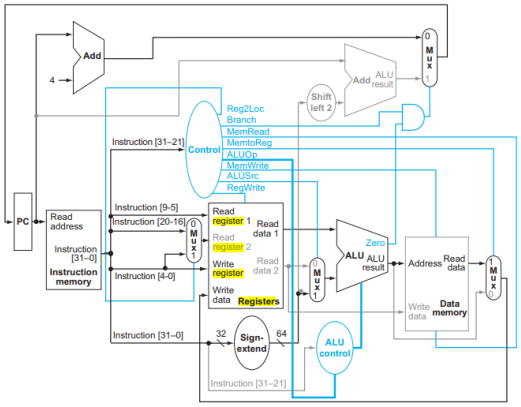
This simplest datapath will attempt to execute all instructions in one clock cycle. This design means that no datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated.



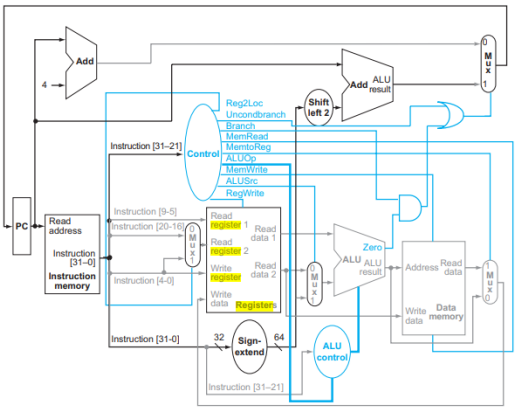
2.3.1 R Type



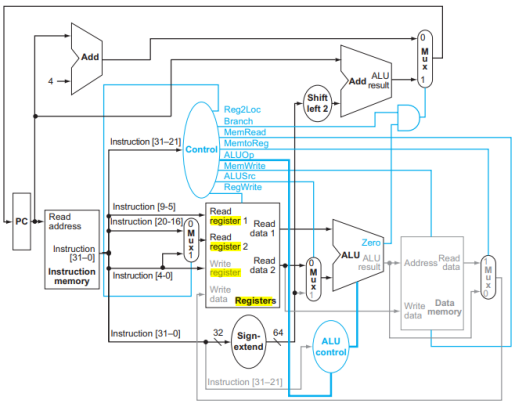
2.3.2 Load



2.3.3 Jump(Unconditional Branch)

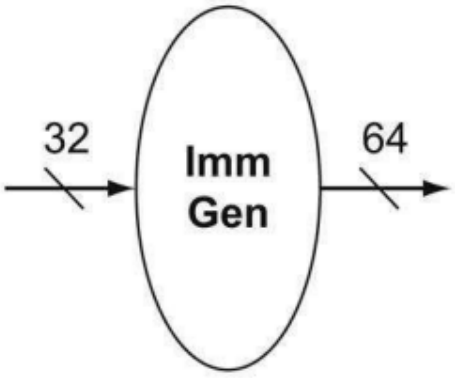


2.3.4 Branch if Equal



2.4 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.



2.5 ALU

An ALU, or Arithmetic Logic Unit, is a crucial component within a CPU (Central Processing Unit) or microprocessor. Its primary function is to perform arithmetic and logical operations on data, which are essential tasks in executing computer programs. 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.

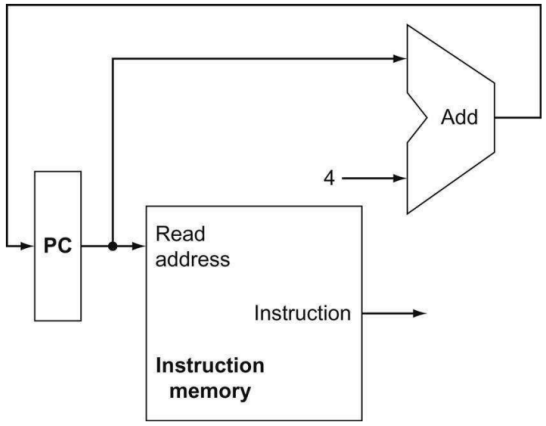
2.6 ALU Control

This style of using multiple levels of decoding—that is, the main control unit generates the ALUOp bits. Using multiple levels of control can reduce the size of the main control unit. Using several smaller control units may also potentially reduce the latency of the control unit. Such

optimizations are important, since the latency of the control unit is often a critical factor in determining the clock cycle time.

2.7 Program Counter

The register containing the address of the instruction in the program being executed. An additional multiplexor is required to select either the sequentially following instruction address (PC +4) or the branch target address to be written into the PC.



2.8 Instruction Memory

Instruction memory is used in the context of computer systems, especially in microprocessors and digital systems, to store and retrieve instructions that are executed by the processor.

2.9 Clocking Methodology

The approach used to determine when data are valid and stable relative to the clock.

2.10 sign-extend

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.

When the condition is true (i.e., two operands are equal), the branch target address becomes the new PC, and we say that the branch is taken. If the operand is not zero, the incremented PC should replace the current PC (just as for any other normal instruction); in this case, we say that the branch is not taken.

2.11 branch target address

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 offset field of the instruction and the address of the branch.

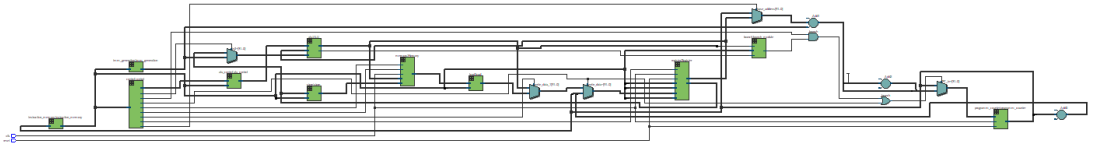
2.12 branch taken

A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional branches are taken branches.

2.13 branch not taken or (untaken branch)

A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch.

3 *RTL Design*



4 *Resource Utilization*

	Resource	Usage	%
1	Logic utilization (ALMs needed / total ALMs on device)	1 / 56,480	< 1 %
2	ALMs needed [=A-B+C]	1	
3			
4	Difficulty packing design	Low	
5			
6	Total LABs: partially or completely used	1 / 5,648	< 1 %
7			
8	Combinational ALUT usage for logic	1	
9	Combinational ALUT usage for route-throughs	0	
10			
11	Dedicated logic registers	0	
12			
13	Virtual pins	0	
14	I/O pins	2 / 268	< 1 %
15			
16	M10K blocks	0 / 686	0 %
17	Total MLAB memory bits	0	
18	Total block memory bits	0 / 7,024,640	0 %
19	Total block memory implementation bits	0 / 7,024,640	0 %
20			
21	Total DSP Blocks	0 / 156	0 %
22			
23	Fractional PLLs	0 / 7	0 %
24	Global signals	0	

25	SERDES Transmitters	0 / 120	0 %
26	SERDES Receivers	0 / 120	0 %
27	JTAGs	0 / 1	0 %
28	ASMI blocks	0 / 1	0 %
29	CRC blocks	0 / 1	0 %
30	Remote update blocks	0 / 1	0 %
31	Oscillator blocks	0 / 1	0 %
32	Hard IPs	0 / 1	0 %
33	Standard RX PCSs	0 / 6	0 %
34	HSSI PMA RX Deserializers	0 / 6	0 %
35	Standard TX PCSs	0 / 6	0 %
36	HSSI PMA TX Serializers	0 / 6	0 %
37	Channel PLLs	0 / 6	0 %
38	Impedance control blocks	0 / 3	0 %
39	Hard Memory Controllers	0 / 2	0 %
40	Average interconnect usage (total/H/V)	0.0% / 0.0% / 0.0%	
41	Peak interconnect usage (total/H/V)	0.0% / 0.0% / 0.0%	
42	Maximum fan-out	1	
43	Highest non-global fan-out	1	
44	Total fan-out	2	
45	Average fan-out	0.40	

Remote update blocks

5 *Implementation of a simple program*

A program for adding numbers from 1 to 6

5.1 C++ Programm

```
    int i = 1;
int k = 7;
int sum = 0;

    while (i != k)
sum = sum + i;
i = i + 1;
```

5.2 RV32I Instructions

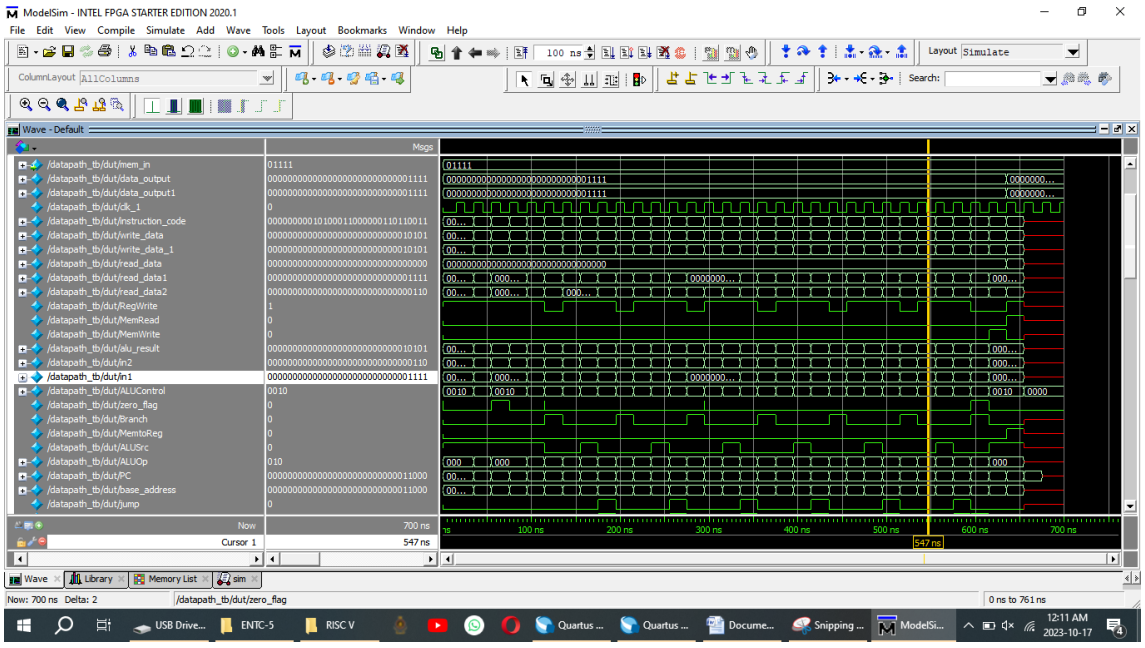
```
    lui x5, 1
srli x5, x5, 12
lui x11, 7
srli x11, x11, 12
lui x3, 0
Loop: beq x5, x11, Exit
add x3, x3, x5
addi x5, x5, 1
```

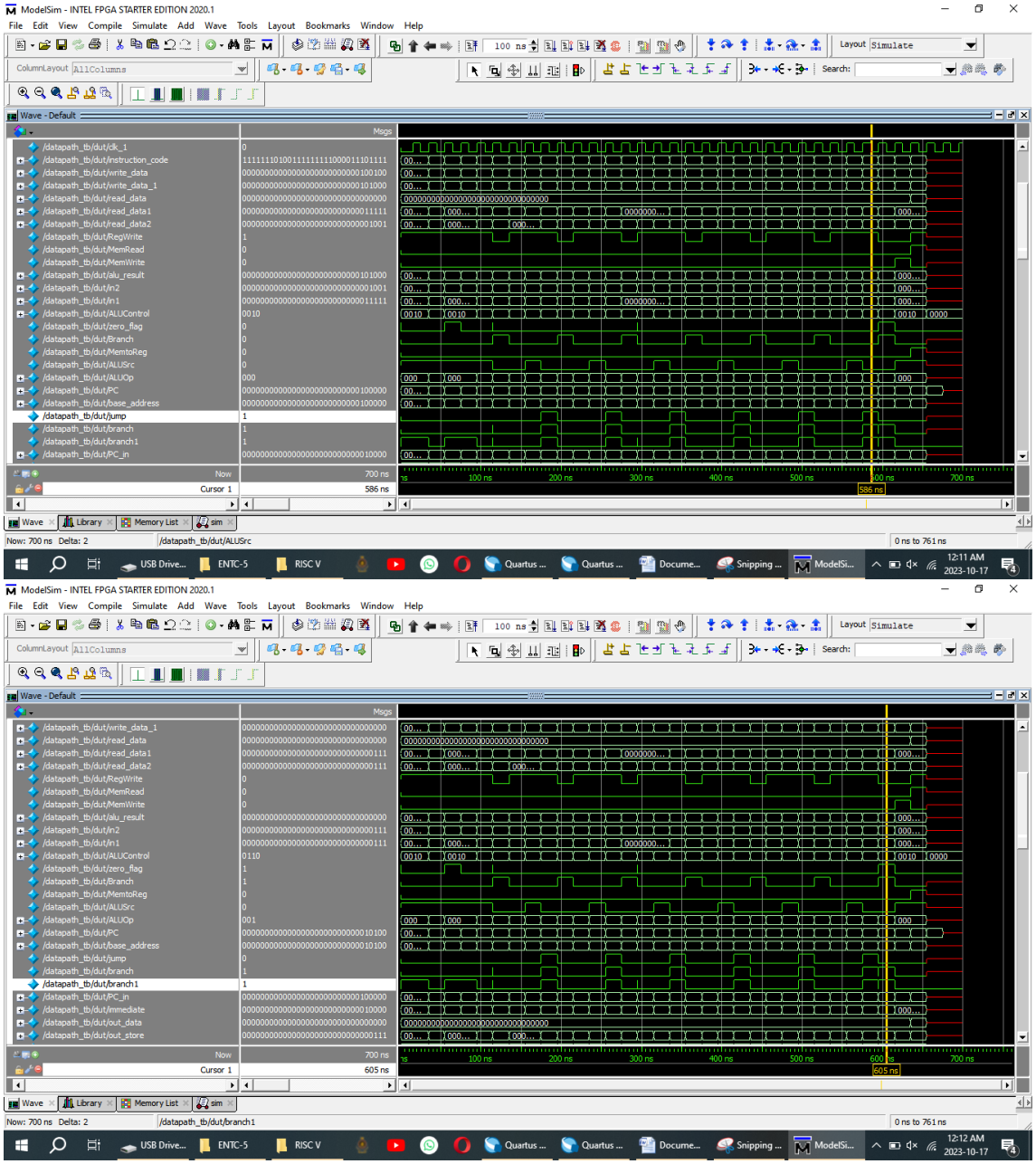
jal rd, -12 Loop

5.3 Instruction Memory

//LUI Memory[3] = 8'h00; Memory[2] = 8'h00; Memory[1] = 8'h12; Memory[0] = 8'hB7;
//SRLI Memory[7] = 8'h00; Memory[6] = 8'hC2; Memory[5] = 8'hD2; Memory[4] = 8'h93;
//LUI Memory[11] = 8'h00; Memory[10] = 8'h00; Memory[9] = 8'h01; Memory[8] = 8'hB7;
//LUI Memory[15] = 8'h00; Memory[14] = 8'h00; Memory[13] = 8'h75; Memory[12] = 8'hB7;
//SRLI Memory[19] = 8'h00; Memory[18] = 8'hC5; Memory[17] = 8'hD5; Memory[16] = 8'h93;
//BEQ Memory[23] = 8'h02; Memory[22] = 8'hB2; Memory[21] = 8'h80; Memory[20] = 8'h63;
//ADD Memory[27] = 8'h00; Memory[26] = 8'h51; Memory[25] = 8'h81; Memory[24] = 8'hB3;
//ADDI Memory[31] = 8'h00; Memory[30] = 8'h12; Memory[29] = 8'h82; Memory[28] = 8'h93;
//JAL Memory[35] = 8'hFE; Memory[34] = 8'h9F; Memory[33] = 8'hF0; Memory[32] = 8'hEF;
//Store Memory[39] = 8'h00; Memory[38] = 8'h37; Memory[37] = 8'hA0; Memory[36] = 8'h23;
//Load Memory[43] = 8'h00; Memory[42] = 8'h07; Memory[41] = 8'hAB; Memory[40] = 8'h03;

5.4 Simulation Result




ModelSim - INTEL FPGA STARTER EDITION 2020.1
File Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
Wave - Default
/datapath_tb/dut/clk_1
/datapath_tb/dut/instruction_code
/datapath_tb/dut/write_data
/datapath_tb/dut/write_data_1
/datapath_tb/dut/read_data
/datapath_tb/dut/read_data1
/datapath_tb/dut/read_data2
/datapath_tb/dut/RegWrite
/datapath_tb/dut/MemRead
/datapath_tb/dut/MemWrite
/datapath_tb/dut/alu_result
/datapath_tb/dut/in2
/datapath_tb/dut/in1
/datapath_tb/dut/ALUControl
/datapath_tb/dut/zero_flag
/datapath_tb/dut/Branch
/datapath_tb/dut/MemtoReg
/datapath_tb/dut/ALUSrc
/datapath_tb/dut/ALUOp
/datapath_tb/dut/PC
/datapath_tb/dut/base_address
/datapath_tb/dut/jump
/datapath_tb/dut/branch
/datapath_tb/dut/branch1
/datapath_tb/dut/PC_in
Now
Cursor 1
Now: 700 ns Delta: 2
/datapath_tb/dut/ALUSrc
0 ns to 761 ns
12:11 AM
2023-10-17

ModelSim - INTEL FPGA STARTER EDITION 2020.1
File Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
Wave - Default
/datapath_tb/dut/write_data_1
/datapath_tb/dut/read_data
/datapath_tb/dut/read_data1
/datapath_tb/dut/read_data2
/datapath_tb/dut/RegWrite
/datapath_tb/dut/MemRead
/datapath_tb/dut/MemWrite
/datapath_tb/dut/alu_result
/datapath_tb/dut/in2
/datapath_tb/dut/in1
/datapath_tb/dut/ALUControl
/datapath_tb/dut/zero_flag
/datapath_tb/dut/Branch
/datapath_tb/dut/MemtoReg
/datapath_tb/dut/ALUSrc
/datapath_tb/dut/ALUOp
/datapath_tb/dut/PC
/datapath_tb/dut/base_address
/datapath_tb/dut/jump
/datapath_tb/dut/branch
/datapath_tb/dut/branch1
/datapath_tb/dut/PC_in
/datapath_tb/dut/immediate
/datapath_tb/dut/out_data
/datapath_tb/dut/out_store
Now
Cursor 1
Now: 700 ns Delta: 2
/datapath_tb/dut/branch1
0 ns to 761 ns
12:12 AM
2023-10-17

6 *Pin Assignment*

Node Name	Direction	Location	I/O Bank	VREF Group	itter Locatior	I/O Standard	Reserved	urrent Streng	Slew Rate	ifferential Pai	ict Preservati
clk	Input	PIN_Y2	2	B2_N0	PIN_Y2	2.5 V		8mA (default)			
display_out[55]	Output	PIN_AA14	3	B3_N0	PIN_AA14	2.5 V		8mA (default)	2 (default)		
display_out[54]	Output	PIN_AG18	4	B4_N2	PIN_AG18	2.5 V		8mA (default)	2 (default)		
display_out[53]	Output	PIN_AF17	4	B4_N2	PIN_AF17	2.5 V		8mA (default)	2 (default)		
display_out[52]	Output	PIN_AH17	4	B4_N2	PIN_AH17	2.5 V		8mA (default)	2 (default)		
display_out[51]	Output	PIN_AG17	4	B4_N2	PIN_AG17	2.5 V		8mA (default)	2 (default)		
display_out[50]	Output	PIN_AE17	4	B4_N2	PIN_AE17	2.5 V		8mA (default)	2 (default)		
display_out[49]	Output	PIN_AD17	4	B4_N2	PIN_AD17	2.5 V		8mA (default)	2 (default)		
display_out[48]	Output	PIN_AC17	4	B4_N2	PIN_AC17	2.5 V		8mA (default)	2 (default)		
display_out[47]	Output	PIN_AA15	4	B4_N2	PIN_AA15	2.5 V		8mA (default)	2 (default)		
display_out[46]	Output	PIN_AB15	4	B4_N2	PIN_AB15	2.5 V		8mA (default)	2 (default)		
display_out[45]	Output	PIN_AB17	4	B4_N1	PIN_AB17	2.5 V		8mA (default)	2 (default)		
display_out[44]	Output	PIN_AA16	4	B4_N2	PIN_AA16	2.5 V		8mA (default)	2 (default)		
display_out[43]	Output	PIN_AB16	4	B4_N2	PIN_AB16	2.5 V		8mA (default)	2 (default)		
display_out[42]	Output	PIN_AA17	4	B4_N1	PIN_AA17	2.5 V		8mA (default)	2 (default)		
display_out[41]	Output	PIN_AH18	4	B4_N2	PIN_AH18	2.5 V		8mA (default)	2 (default)		
display_out[40]	Output	PIN_AF18	4	B4_N1	PIN_AF18	2.5 V		8mA (default)	2 (default)		
display_out[39]	Output	PIN_AG19	4	B4_N2	PIN_AG19	2.5 V		8mA (default)	2 (default)		
display_out[38]	Output	PIN_AH19	4	B4_N2	PIN_AH19	2.5 V		8mA (default)	2 (default)		
display_out[37]	Output	PIN_AB18	4	B4_N0	PIN_AB18	2.5 V		8mA (default)	2 (default)		
display_out[36]	Output	PIN_AC18	4	B4_N1	PIN_AC18	2.5 V		8mA (default)	2 (default)		
display_out[35]	Output	PIN_AD18	4	B4_N1	PIN_AD18	2.5 V		8mA (default)	2 (default)		
display_out[34]	Output	PIN_AE18	4	B4_N2	PIN_AE18	2.5 V		8mA (default)	2 (default)		
display_out[33]	Output	PIN_AF19	4	B4_N1	PIN_AF19	2.5 V		8mA (default)	2 (default)		
display_out[32]	Output	PIN_AE19	4	B4_N1	PIN_AE19	2.5 V		8mA (default)	2 (default)		
display_out[31]	Output	PIN_AH21	4	B4_N2	PIN_AH21	2.5 V		8mA (default)	2 (default)		
display_out[30]	Output	PIN_AG21	4	B4_N2	PIN_AG21	2.5 V		8mA (default)	2 (default)		
display_out[29]	Output	PIN_AA19	4	B4_N0	PIN_AA19	2.5 V		8mA (default)	2 (default)		
display_out[28]	Output	PIN_AB19	4	B4_N0	PIN_AB19	2.5 V		8mA (default)	2 (default)		
display_out[27]	Output	PIN_Y19	4	B4_N0	PIN_Y19	2.5 V		8mA (default)	2 (default)		
display_out[26]	Output	PIN_AF23	4	B4_N0	PIN_AF23	2.5 V		8mA (default)	2 (default)		
display_out[25]	Output	PIN_AD24	4	B4_N0	PIN_AD24	2.5 V		8mA (default)	2 (default)		
display_out[24]	Output	PIN_AA21	4	B4_N0	PIN_AA21	2.5 V		8mA (default)	2 (default)		
display_out[23]	Output	PIN_AB20	4	B4_N0	PIN_AB20	2.5 V		8mA (default)	2 (default)		
display_out[22]	Output	PIN_U21	5	B5_N0	PIN_U21	2.5 V		8mA (default)	2 (default)		
display_out[21]	Output	PIN_V21	5	B5_N1	PIN_V21	2.5 V		8mA (default)	2 (default)		
display_out[20]	Output	PIN_W28	5	B5_N1	PIN_W28	2.5 V		8mA (default)	2 (default)		
display_out[19]	Output	PIN_W27	5	B5_N1	PIN_W27	2.5 V		8mA (default)	2 (default)		
display_out[18]	Output	PIN_Y26	5	B5_N1	PIN_Y26	2.5 V		8mA (default)	2 (default)		
display_out[17]	Output	PIN_W26	5	B5_N1	PIN_W26	2.5 V		8mA (default)	2 (default)		
display_out[16]	Output	PIN_Y25	5	B5_N1	PIN_Y25	2.5 V		8mA (default)	2 (default)		
display_out[15]	Output	PIN_AA26	5	B5_N1	PIN_AA26	2.5 V		8mA (default)	2 (default)		
display_out[14]	Output	PIN_AA25	5	B5_N1	PIN_AA25	2.5 V		8mA (default)	2 (default)		
display_out[13]	Output	PIN_U24	5	B5_N0	PIN_U24	2.5 V		8mA (default)	2 (default)		
display_out[12]	Output	PIN_U23	5	B5_N1	PIN_U23	2.5 V		8mA (default)	2 (default)		
display_out[11]	Output	PIN_W25	5	B5_N1	PIN_W25	2.5 V		8mA (default)	2 (default)		
display_out[10]	Output	PIN_W22	5	B5_N0	PIN_W22	2.5 V		8mA (default)	2 (default)		
display_out[9]	Output	PIN_W21	5	B5_N1	PIN_W21	2.5 V		8mA (default)	2 (default)		
display_out[8]	Output	PIN_Y22	5	B5_N0	PIN_Y22	2.5 V		8mA (default)	2 (default)		
display_out[7]	Output	PIN_M24	6	B6_N2	PIN_M24	2.5 V		8mA (default)	2 (default)		
display_out[6]	Output	PIN_H22	6	B6_N0	PIN_H22	2.5 V		8mA (default)	2 (default)		
display_out[5]	Output	PIN_J22	6	B6_N0	PIN_J22	2.5 V		8mA (default)	2 (default)		
display_out[4]	Output	PIN_L25	6	B6_N1	PIN_L25	2.5 V		8mA (default)	2 (default)		
display_out[3]	Output	PIN_L26	6	B6_N1	PIN_L26	2.5 V		8mA (default)	2 (default)		
display_out[2]	Output	PIN_E17	7	B7_N2	PIN_E17	2.5 V		8mA (default)	2 (default)		
display_out[1]	Output	PIN_F22	7	B7_N0	PIN_F22	2.5 V		8mA (default)	2 (default)		
display_out[0]	Output	PIN_G18	7	B7_N2	PIN_G18	2.5 V		8mA (default)	2 (default)		
mem_in[4]	Input	PIN_AB27	5	B5_N1	PIN_AB27	2.5 V		8mA (default)			
mem_in[3]	Input	PIN_AD27	5	B5_N2	PIN_AD27	2.5 V		8mA (default)			
mem_in[2]	Input	PIN_AC27	5	B5_N2	PIN_AC27	2.5 V		8mA (default)			
mem_in[1]	Input	PIN_AC28	5	B5_N2	PIN_AC28	2.5 V		8mA (default)			
mem_in[0]	Input	PIN_AB28	5	B5_N1	PIN_AB28	2.5 V		8mA (default)			
reset	Input	PIN_M23	6	B6_N2	PIN_M23	2.5 V		8mA (default)			
<<new node>>											

Implemented Instructions RV32I

R – Type Instructions(10)

ADD

0000000	rs2	rs1	000	rd	0110011

add rd, rs1, rs2

Adds the registers rs1 and rs2 and stores the result in rd.

x[rd] = x[rs1] + x[rs2]

SUB

0100000	rs2	rs1	000	rd	0110011

sub rd, rs1 ,rs2

Substract the register rs2 from rs1 and stores the result in rd.

x[rd] = x[rs1] - x[rs2]

SLL

0000000	rs2	rs1	001	rd	0110011

sll rd, rs1, rs2

Performs logical left shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.

x[rd] = x[rs1] << x[rs2]

SLT

0000000	rs2	rs1	010	rd	0110011

slt rd, rs1, rs2

Place the value 1 in register rd if register rs1 is less than register rs2 when both are treated as signed numbers, else 0 is written to rd.

x[rd] = x[rs1] < x[rs2] //Signed

SLTU

0000000	rs2	rs1	011	rd	0110011

sltu rd, rs1, rs2

Place the value 1 in register rd if register rs1 is less than register rs2 when both are treated as unsigned numbers, else 0 is written to rd.

x[rd] = x[rs1] < x[rs2] //Un-Signed

XOR

0000000	rs2	rs1	100	rd	0110011

xor rd, rs1, rs2

Performs bitwise XOR on registers rs1 and rs2 and place the result in rd.

x[rd] = x[rs1] ^ x[rs2]

SRL

0000000	rs2	rs1	101	rd	0110011

srl rd, rs1, rs2

Logical right shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.

x[rd] = x[rs1] >> x[rs2]

SRA

0100000	rs2	rs1	101	rd	0110011

sra rd, rs1, rs2

Performs arithmetic right shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.

x[rd] = x[rs1] >>> x[rs2]

OR

0000000	rs2	rs1	110	rd	0110011

or rd, rs1, rs2

Performs bitwise OR on registers rs1 and rs2 and place the result in rd.

x[rd] = x[rs1] | x[rs2]

AND

0000000	rs2	rs1	111	rd	0110011

and rd, rs1, rs2

Performs bitwise AND on registers rs1 and rs2 and place the result in rd.

x[rd] = x[rs1] & x[rs2]

I – Type Instructions(9)

ADDI

imm[11:0]	rs1	000	rd	0010011

addi rd, rs1, imm

Adds the sign-extended 12-bit immediate to register rs1 and stores the result in rd.

x[rd] = x[rs1] + imm

SLLI

0000000	shamt	rs1	001	rd	0010011

slli rd, rs1, shamt

Performs logical left shift on the value in register rs1 by the shift amount held in the lower 5 bits of the immediate.

x[rd] = x[rs1] << imm

SLTI

imm[11:0]	rs1	010	rd	0010011

slti rd, rs1, imm

Place the value 1 in register rd if register rs1 is less than the sign extended immediate when both are treated as signed numbers, else 0 is written to rd.

x[rd] = x[rs1] < imm //Signed

SLTIU

imm[11:0]	rs1	011	rd	0010011

sltiu rd, rs1, imm

Place the value 1 in register rd if register rs1 is less than the immediate when both are treated as unsigned numbers, else 0 is written to rd.

x[rd] = x[rs1] < imm //Un-Signed

XORI

imm[11:0]	rs1	100	rd	0010011

xori rd, rs1, imm

Performs bitwise XOR on register rs1 and the sign-extended 12-bit immediate and place the result in rd.

x[rd] = x[rs1] ^ imm

SRLI

0000000	shamt	rs1	101	rd	0010011

srli rd, rs1, shamt

Performs logical right shift on the value in register rs1 by the shift amount held in the lower 5 bits of the immediate.

x[rd] = x[rs1] >> imm

SRAI

0100000	shamt	rs1	101	rd	0010011

srai rd, rs1, shamt

Performs arithmetic right shift on the value in register rs1 by the shift amount held in the lower 5 bits of the immediate.

x[rd] = x[rs1] >>> imm

ORI

imm[11:0]	rs1	110	rd	0010011

ori rd, rs1, imm

Performs bitwise OR on register rs1 and the sign-extended 12-bit immediate and place the result in rd.

x[rd] = x[rs1] | imm

ANDI

imm[11:0]	rs1	111	rd	0010011

andi rd, rs1, imm

Performs bitwise AND on register rs1 and the sign-extended 12-bit immediate and place the result in rd.

x[rd] = x[rs1] & imm

B – Type Instructions(6)

BEQ

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011

beq rs1, rs2, offset

Take the branch if registers rs1 and rs2 are equal.

If (x[rs1] == x[rs2]) then PC = PC + offset

BNE

imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011

bne rs1, rs2, offset

Take the branch if registers rs1 and rs2 are not equal.

If (x[rs1] != x[rs2]) then PC = PC + offset

BLT

imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011

blt rs1, rs2, offset

Take the branch if registers rs1 is less than rs2, using signed comparison.

If (x[rs1] < x[rs2]) then PC = PC + offset //Signed

BGE

imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011

bge rs1, rs2, offset

Take the branch if registers rs1 is greater than or equal to rs2, using signed comparison.

If (x[rs1] ≥ x[rs2]) then PC = PC + offset //Signed

BLTU

imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011

bltu rs1, rs2, offset

Take the branch if registers rs1 is less than rs2, using unsigned comparison.

If (x[rs1] < x[rs2]) then PC = PC + offset //Un-Signed

BGEU

imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011

bgeu rs1, rs2, offset

Take the branch if registers rs1 is greater than or equal to rs2, using unsigned comparison.

If (x[rs1] ≥ x[rs2]) then PC = PC + offset //Un-Signed

Load Instructions(5)

LB

imm[11:0]	rs1	000	rd	0110011

lb rd, offset(rs1)

Loads 8-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.

x[rd] = M[x[rs1] + offset][7:0] //Signed

LH

imm[11:0]	rs1	001	rd	0110011

lh rd, offset(rs1)

Loads a 16-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.

x[rd] = M[x[rs1] + offset][15:0] //Signed

LW

imm[11:0]	rs1	010	rd	0110011

lw rd, offset(rs1)

Loads a 32-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.

x[rd] = M[x[rs1] + offset][31:0] //Signed

LBU

imm[11:0]	rs1	100	rd	0110011

lbu rd, offset(rs1)

Loads a 8-bit value from memory and zero-extends this to XLEN bits before storing it in register rd.

x[rd] = M[x[rs1] + offset][7:0] //Un-Signed

LHU

imm[11:0]	rs1	101	rd	0110011

lhu rd, offset(rs1)

Loads a 16-bit value from memory and zero-extends this to XLEN bits before storing it in register rd.

x[rd] = M[x[rs1] + offset][31:0] //Un-Signed

Store Instructions(3)

SB

imm[11:5]	rs2	rs1	000	imm[4:0]	0100011

sb rs2, offset(rs1)

Store 8-bit, values from the low bits of register rs2 to memory.

M[x[rs1] + offset] = x[rs2][7:0]

SH

imm[11:5]	rs2	rs1	001	imm[4:0]	0100011

sh rs2, offset(rs1)

Store 16-bit, values from the low bits of register rs2 to memory.

M[x[rs1] + offset] = x[rs2][15:0]

SW

imm[11:5]	rs2	rs1	010	imm[4:0]	0100011

sw rs2, offset(rs1)

Store 32-bit, values from the low bits of register rs2 to memory.

M[x[rs1] + offset] = x[rs2][31:0]

Jump Instructions(2)

JAL

imm[20|10:1|11|19:12]	rd	1101111

jal rd, offset

Jump to address and place return address in rd.

x[rd] = PC + 4

PC = PC + offset

JALR

imm[11:0]	rs1	000	rd	1100111

jalr rd, rs1, offset

Jump to address and place return address in rd.

x[rd] = PC + 4

PC = x[rs1] + offset

U – Type Instructions(2)

LUI

imm[31:12]	rd	0110111

lui rd, imm

LUI places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros.

x[rd] = imm[31:12] << 12

AUIPC

imm[31:12]	rd	0010111

auipc rd, imm

AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the pc, then places the result in register rd.

x[rd] = PC + (imm[31:12] << 12)

Multiplication (Sum of Operand size < 31)

But here operand size should be below 16.

```
addi x10, x0, 16
addi x7, x0, 1
add x8, x0, x0

jal x0, WHEXP

WHLOOP: and x9, x4, x7
        Beq x9, x7, MULT

ENDIF:  addi x8, x8, 1
        Slli x7, x7, 1
        Beq x0, x0, WHEXP

MULT:   sll x11, x3, x0
        Add x5, x5, x11
        Beq x0, x0, ENDIF

WHEXP:  blt x8, x10, WHLOOP
```

Appendix I(System Verilog Codes)

0.1 ALU

```
module alu(
    input [31:0] in1, in2,
    input [3:0] alu_control,
    output reg [31:0] alu_result,
    output reg zero_flag
);

    always @(*) begin
        case (alu_control)
            4'b0000: alu_result = in1 & in2;
            4'b0001: alu_result = in1 | in2;
            4'b0010: alu_result = in1 + in2;
            4'b0110: alu_result = in1 - in2;
            4'b0011: alu_result = in1 << in2;
            4'b0100: alu_result = in1 >> in2;
            4'b0101: alu_result = in1 >>> in2;
            4'b0111: alu_result = in1 ^ in2;
            4'b1000: if ($signed(in1) < $signed(in2)) alu_result = 1'b1;
                        else alu_result = 1'b0;
            4'b1001: if (in1 < in2) alu_result = 1'b1;
                        else alu_result = 1'b0;
            default: alu_result = 32'b0;
        endcase
        if (alu_result == 0)
            zero_flag = 1'b1;
        else
            zero_flag = 1'b0;
    end
endmodule
```

0.2 Register

```
module register(
    input [4:0] rs1,
    input [4:0] rs2,
    input [4:0] write_reg,
    input [31:0] write_data,
    output [31:0] read_data1,
    output [31:0] read_data2,
    input RegWrite,
    input clk,
    input reset
);

    reg [31:0] reg_memory [31:0];

    // Initial values for reg_memory
    initial begin
        for (int i = 0; i < 32; i = i + 1) begin
            reg_memory[i] = i;
        end
    end

    assign read_data1 = reg_memory[rs1];
    assign read_data2 = reg_memory[rs2];
```

```
    always @(posedge clk) begin
        if (reset) begin
            // Reset values for reg_memory
            for (int i = 0; i < 32; i = i + 1) begin
                reg_memory[i] = i;
            end
        end else if (RegWrite) begin
            reg_memory[write_reg] = write_data;
        end
    end
endmodule
```

0.3 Data Memory

```
module memory(
        input MemRead,
        input MemWrite,
        input [31:0] ram_addr,
        input [31:0] write_data,
        output [31:0] read_data,
        input clk
);
wire [31:0] memory_addr = ram_addr[31:0];
reg [31:0] memory [31:0];
    // Initial values for memory
    initial begin
        for (int i = 0; i < 31; i = i + 1) begin
            memory[i] = i;
        end
    end

always @(posedge clk) begin
        if (MemWrite) memory[memory_addr] = write_data;
end
assign read_data = (MemRead == 1'b1) ? memory[memory_addr]: 16'd0;
endmodule
```

0.4 Program Counter

```
module programm_counter(
        input clk,
        input reset,
        input [31:0] PC_in,
        output reg [31:0] PC
);
always @(posedge clk or posedge reset) begin
    if (reset) begin
        PC <= 32'h00000000;
    end else begin
        PC = PC_in + 32'h00000004;
    end
end
endmodule
```

0.5 Instruction Memory

```
module instruction_memory(
        input [31:0] PC,
        //input reset,
        output reg [31:0] instruction_code
);
```

```
reg [7:0] Memory [1240:0];
assign instruction_code = {Memory[PC+3],Memory[PC+2],Memory[PC+1],Memory[PC]};
initial begin
        // Setting 32-bit instruction: add t1, s0,s1 =>0x004A0566
        Memory[3] = 8'h00;
        Memory[2] = 8'h4A;
        Memory[1] = 8'h05;
        Memory[0] = 8'h66;

        //ADD t2, s3, s4 (encoded as 0x01CD0733):
        Memory[7] = 8'h01;
        Memory[6] = 8'hCD;
        Memory[5] = 8'h07;
        Memory[4] = 8'h33;

        //sub t2, s3, s4 (encoded as 0x41CD0733):
        Memory[11] = 8'h41;
        Memory[10] = 8'hCD;
        Memory[9] = 8'h07;
        Memory[8] = 8'h33;
end
endmodule
```

0.6 Control

```
module control(
        input [31:0] instruction_code,
        output reg Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite, Jump, jalr,
        output reg [2:0] ALUOp
);
//R type
//OPCODE   |   rd   | funct3 |   rs1   |   rs2   | funct7
//0110011  |  5 bits|  000   |  5 bits |  5 bits|  0000000

//S Type
//OPCODE   | imm[11:5] | funct3 |   rs1   |   rs2   | imm[4:0]
//0100011  |   7 bits  |  010   |  5 bits |  5 bits|  5 bits

//BEQ
//OPCODE   |   imm[12] | imm[10:5] | imm[4:1] | imm[11] | funct3 |   rs1   |   rs2
| imm[8:7] | imm[6] | imm[5:2] | imm[1:0]
//1100011  |  1 bit    |  6 bits   |  4 bits  |  1 bit  |  000   |  5 bits |  5 bits |
2 bits   |  1 bit |  4 bits  |  2 bits

//Load
//OPCODE   |   rd   | funct3 |   rs1   |   imm[11:0]
//0000011  |  5 bits|  000   |  5 bits |  12 bits

//ADDI
//OPCODE   |   rd   | funct3 |   rs1   |   imm[11:0]
//0010011  |  5 bits|  000   |  5 bits |  12 bits

assign lui = (instruction_code[6:0]==7'b0110111) ? 1'b1 : 1'b0;
assign auipc = (instruction_code[6:0]==7'b0010111) ? 1'b1 : 1'b0;
assign jalr = (instruction_code[6:0]==7'b1100111) ? 1'b1 : 1'b0;
assign Jump = ((instruction_code[6:0]==7'b1101111) | (instruction_code[6:0]==7'b110011
assign Branch = (instruction_code[6:0]==7'b1100011) ? 1'b1 : 1'b0; //B Type
assign MemRead = (instruction_code[6:0]==7'b0000011) ? 1'b1 : 1'b0; //Load
assign MemtoReg = (instruction_code[6:0]==7'b0000011) ? 1'b1 : 1'b0; //Load
assign MemWrite = (instruction_code[6:0]==7'b0100011) ? 1'b1 : 1'b0; //Store
```

```
assign RegWrite = ((instruction_code[6:0]==7'b0000011) | (instruction_code[6:0]==7'b01
assign ALUSrc = ((instruction_code[6:0]==7'b0010011) | (instruction_code[6:0]==7'b0000
AUIPC
assign ALUOp[0] = (instruction_code[6:0]==7'b1100011) ? 1'b1 : 1'b0; //BEQ Type
assign ALUOp[1] = (instruction_code[6:0]==7'b0110011) ? 1'b1 : 1'b0; //R Type
assign ALUOp[2] = (instruction_code[6:0]==7'b0010011) ? 1'b1 : 1'b0; //I Type
endmodule
```

0.7 ALU Control

```
module alu_control(
    input reg [31:0] instruction_code,
    input wire [2:0] ALUOp,
    output reg [3:0] ALUControl
);

always @(*) begin
    if (ALUOp == 3'b000) begin
        ALUControl = 4'b0010; // Common case for ALUOp 00 and 01 (ADD or SUB)

    end else if (ALUOp == 3'b010) begin
         //register
        if (instruction_code[14:12] == 3'b000) begin
            if (instruction_code[31:25] == 7'b0000000) begin
                ALUControl = 4'b0010; // ADD
            end else if (instruction_code[31:25] == 7'b0100000) begin
                ALUControl = 4'b0110; // SUB
            end
        end else if (instruction_code[14:12] == 3'b101) begin
            if (instruction_code[31:25] == 7'b0000000) begin
                ALUControl = 4'b0100; // SRL
            end else if (instruction_code[31:25] == 7'b0100000) begin
                ALUControl = 4'b0101; // SRA
            end
        end else if (instruction_code[14:12] == 3'b111) begin
            if (instruction_code[31:25] == 7'b0000000) begin
                ALUControl = 4'b0000; // AND
            end
        end else if (instruction_code[14:12] == 3'b110) begin
            if (instruction_code[31:25] == 7'b0000000) begin
                ALUControl = 4'b0001; // OR
            end
        end else if (instruction_code[14:12] == 3'b001) begin
            if (instruction_code[31:25] == 7'b0000000) begin
                ALUControl = 4'b0011; // SLL
            end
        end else if (instruction_code[14:12] == 3'b100) begin
            if (instruction_code[31:25] == 7'b0000000) begin
                ALUControl = 4'b0111; // XOR
            end
        end else if (instruction_code[14:12] == 3'b010) begin
            if (instruction_code[31:25] == 7'b0000000) begin
                ALUControl = 4'b1000; // SLT
            end
        end else if (instruction_code[14:12] == 3'b011) begin
            if (instruction_code[31:25] == 7'b0000000) begin
                ALUControl = 4'b1001; // SLTU
            end
        end else begin
            ALUControl = 4'b0010; // Default value for other combinations
```

```
        end

    end else if (ALUOp == 3'b100) begin
    //IMMEDIATES
        if (instruction_code[14:12] == 3'b000) begin
            ALUControl = 4'b0010; // ADD
        end else if (instruction_code[14:12] == 3'b101) begin
            if (instruction_code[31:25] == 7'b0000000) begin
                ALUControl = 4'b0100; // SRLI
            end else if (instruction_code[31:25] == 7'b0100000) begin
                ALUControl = 4'b0101; // SRAI
            end
        end else if (instruction_code[14:12] == 3'b111) begin
                ALUControl = 4'b0000; // AND
        end else if (instruction_code[14:12] == 3'b110) begin
                ALUControl = 4'b0001; // OR
        end else if (instruction_code[14:12] == 3'b001) begin
                ALUControl = 4'b0011; // SLL
        end else if (instruction_code[14:12] == 3'b100) begin
                ALUControl = 4'b0111; // XOR
        end else if (instruction_code[14:12] == 3'b010) begin
                ALUControl = 4'b1000; // SLT
        end else if (instruction_code[14:12] == 3'b011) begin
                ALUControl = 4'b1001; // SLTU
        end else begin
                ALUControl = 4'b0010; // Default value for other combinations
         end

    end else if (ALUOp == 3'b001) begin
    //BRANCHES
        if (instruction_code[14:12] == 3'b000) begin //BEQ
                ALUControl = 4'b0110; // SUB
        end else if (instruction_code[14:12] == 3'b001) begin //BNE
                ALUControl = 4'b0110; // SUB
        end else if (instruction_code[14:12] == 3'b100) begin //BLT
                ALUControl = 4'b1000; // SLT
        end else if (instruction_code[14:12] == 3'b110) begin //BLTU
                ALUControl = 4'b1001; // SLTU
        end else if (instruction_code[14:12] == 3'b101) begin //BGE
                ALUControl = 4'b1000; // SLT
        end else if (instruction_code[14:12] == 3'b111) begin //BGEU
                ALUControl = 4'b1001; // SLTU
        end else begin
                ALUControl = 4'b0010; // Default value for other combinations
        end

    end else begin
        ALUControl = 4'b0000; // Default value for other ALUOp values.
    end
end
endmodule
```

0.8 Datapath

```
module datapath(
        input clk,
        input reset
);

reg [31:0] instruction_code;
```

```
reg [31:0] write_data;
reg [31:0] write_data_1;
reg [31:0] read_data;
reg [31:0] read_data1;
reg [31:0] read_data2;
reg RegWrite;
reg MemRead;
reg MemWrite;
reg [31:0] alu_result;
reg [31:0] in2;
reg [31:0] in1;
reg [3:0] ALUControl;
reg zero_flag;
reg Branch;
reg MemtoReg;
reg ALUSrc;
reg [2:0] ALUOp;
reg [31:0] PC;
reg [31:0] base_address;
reg jump;
reg branch;
reg branch1;
reg [31:0] PC_in;
//reg [31:0] PC_mux1;
reg [31:0] immediate;
reg [31:0] out_data;
reg [31:0] out_store;
reg [31:0] in1_b;
reg jalr;
reg auipc;
reg lui;

assign in2 = (ALUSrc) ? immediate:read_data2;
assign write_data_1 = (MemtoReg) ? out_data:alu_result;
assign write_data = (jump) ? (PC + 32'h00000004) :write_data_1;
assign branch = ((Branch && branch1) || jump);
assign PC_in = (branch) ? (base_address + immediate  - 32'h00000004):PC;  //jump mux
assign base_address = (jalr) ? read_data1 : PC; //base_reg
assign in1 = (lui) ? 32'h00000000:in1_b;
assign in1_b = (auipc) ? PC:read_data1;

register Register(
        .rs1(instruction_code [19:15]),
        .rs2(instruction_code [24:20]),
        .write_reg(instruction_code [11:7]),
        .write_data(write_data),
        .read_data1(read_data1),
        .read_data2(read_data2),
        .RegWrite(RegWrite),
        .clk(clk),
        .reset(reset)
);

memory Memory(
        .MemRead(MemRead),
        .MemWrite(MemWrite),
        .ram_addr(alu_result),
        .write_data(out_store),
        .read_data(read_data),
```

```
        .clk(clk)
);

alu ALU(
    .in1(in1),
         .in2(in2),
    .alu_control(ALUControl),
    .alu_result(alu_result),
    .zero_flag(zero_flag)
);

programm_counter programm_counter(
        .clk(clk),
        .reset(reset),
        .PC_in(PC_in),
        .PC(PC)
);

instruction_memory instruction_memory(
        .PC(PC),
        .instruction_code(instruction_code)
);

control control(
        .instruction_code(instruction_code),
        .Branch(Branch),
        .MemRead(MemRead),
        .MemtoReg(MemtoReg),
        .MemWrite(MemWrite),
        .ALUSrc(ALUSrc),
        .RegWrite(RegWrite),
        .Jump(jump),
        .jalr(jalr),
        .lui(lui),
        .auipc(auipc),
        .ALUOp(ALUOp)
);

alu_control alu_control(
    .instruction_code(instruction_code),
    .ALUOp(ALUOp),
    .ALUControl(ALUControl)
);

imm_generation imm_generation(
        .instruction(instruction_code),
        .immediate(immediate)
);

branch branch_module(
        .instruction(instruction_code),
        .alu_result(alu_result[0]),
        .zero_flag(zero_flag),
        .branch(branch1)
);

load load(
        .instruction(instruction_code[14:12]),
        .in_data(read_data),
```

```
        .out_data(out_data)
);

store store(
        .instruction(instruction_code[14:12]),
        .in_store(read_data2),
        .out_store(out_store)
);
endmodule
```

0.9 Immediate Generation

```
module imm_generation(
        input   [31:0] instruction,
        output reg [31:0] immediate
);
always @(*) begin
    if (instruction[6:0] == 7'b0010011) begin
        if (instruction[14:12] == 3'b101 || instruction[14:12] == 3'b001) begin
            immediate = 32'h00000000 + instruction[24:20];  //I TYPE
        end else begin
                                immediate = 32'h00000000 + instruction[31:20]; // Defa
                end
         end else if (instruction[6:0] == 7'b1100011) begin //BRANCH
                                immediate = $signed({instruction[31],instruction[7],in
         end else if (instruction[6:0] == 7'b1100111) begin //JALR
                                immediate = $signed(instruction[31:20]);
         end else if (instruction[6:0] == 7'b1101111) begin //JAL
                                immediate = $signed({instruction[31],instruction[19:12
         end else if (instruction[6:0] == 7'b0000011) begin //Load
                                immediate = 32'h00000000 + instruction[31:20];
         end else if (instruction[6:0] == 7'b0100011) begin //Store
                                immediate = 32'h00000000 + {instruction[31:25],instruc
         end else if (instruction[6:0] == 7'b0110111) begin //LUI
                                immediate = 32'h00000000 + (instruction[31:12]<<12);
         end else if (instruction[6:0] == 7'b0010111) begin //AUIPC
                                immediate = 32'h00000000 + (instruction[31:12]<<12);
    end else begin
        immediate = 32'h00000000 + instruction[31:20]; // Default value
    end
end
endmodule
```

0.10 Branch

```
module branch(
        input [31:0] instruction,
        input alu_result,
        input zero_flag,
        output reg branch
);
always @(*) begin
                if (instruction[14:12] == 7'b000) begin //BEQ
                        branch = zero_flag;
                end else if (instruction[14:12] == 7'b001) begin //BNE
                        branch = ~(zero_flag);
                end else if (instruction[14:12] == 7'b100) begin //BLT
                        branch = alu_result;
                end else if (instruction[14:12] == 7'b110) begin //BLTU
                        branch = alu_result;
```

```
                end else if (instruction[14:12] == 7'b101) begin //BGE
                        branch = ~(alu_result);
                end else if (instruction[14:12] == 7'b111) begin //BGEU
                        branch = ~(alu_result);
                end else begin
                        branch = 1'b0;
                end
end
endmodule
```

0.11 Load

```
module load(
        input [2:0] instruction,
        input [31:0] in_data,
        output reg [31:0] out_data
);
always @(*) begin
        if (instruction == 3'b000) out_data = $signed(in_data[7:0]) + $signed(32'h0000
        else if (instruction == 3'b100) out_data = (in_data[7:0] + 32'h00000000); //LB
        else if (instruction == 3'b001) out_data = $signed(in_data[15:0]) + $signed(32
        else if (instruction == 3'b101) out_data = (in_data[15:0] + 32'h00000000); //L
        else out_data = in_data; //LW
end
endmodule
```

0.12 Store

```
module store(
        input [2:0] instruction,
        input [31:0] in_store,
        output reg [31:0] out_store
);
always @(*) begin
        if (instruction == 3'b000) out_store = in_store[7:0] + 32'h00000000; //SB
        else if (instruction == 3'b001) out_store = in_store[15:0] + 32'h00000000; //S
        else out_store = in_store; //LW
end
endmodule
```

7 *References*

Computer Organization and Design RISC-V Edition: The Hardware Software Interface
https://pdf1.alldatasheet.com/datasheet-pdf/download/487677/ALTERA/DE2-115.html
https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=EnglishCategoryNo=139No=502PartNo=4con