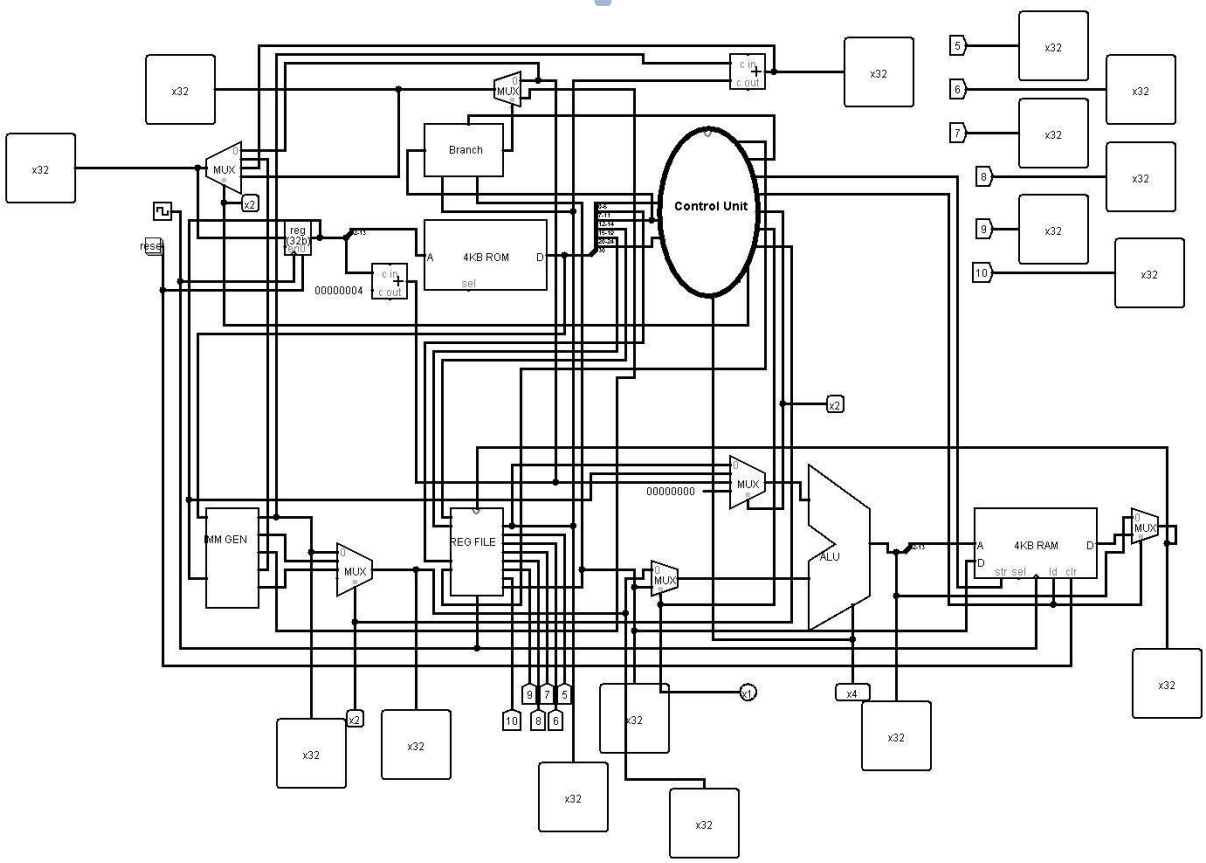
CAO-ASSIGNMENT-1 CHAP #4 RISC-V - SUMMARY

RV-32I SINGLE CYCLE PROCESSOR



MADE BY: MUHAMMAD OSAMA NADEEM
19B-057-CE - SEC: E

Computer Architecture & Organization
Assignment # 1

Summar chap 4 RISC-V

RV-32I Single Cycle CPU is based on 32 Bit RISC-V architecture. As the abbreviation itself describes that RISC (Reduced Instructions Set Computer) architecture provides an ease to the users to while comparing to the CISC (Complex Instructions Set Computer) architecture. Chapter 4 of Risc-V edition book covers the basic implementation of risc-v single cycle processor. This chapter contains an explanation of the principles and techniques used in implementing a processor, starting with a highly abstract and simplified overview. The base RISC-V instruction set is composed of ISA called Instruction set architecture which has 47 instructions. Eight are system instructions that perform system calls and access performance counters. The remaining 39 instructions fall into the categories of computational instructions, control flow instructions, and memory access instructions….. Risc is an open source architecture of processor that assures its users that by implementing ISA the user can build and learn the processor. Risc-V is widely used as an open source in private and professional fields depending on its use somewhere it is used as a learning processor, students can easily learn about computer architecture if they are aware with the ISA that Risc-v offers. We learned about its uses and its impact in the world of computing, now we wil see its working & functions. Electrons are the base of risc-v as we see that digital logic gates(AND ,OR, inverter) are the bases of the single cycle RV-32I processor and inside these gates are built of electrons. From these gates we built logic by following ISA instructions. These gates are used to build flip flops and D-flip flops makes half adder and full adder. Thus by using these logic we make risc v.

PC program counter tells the instruction memory about the next address whether it's the preceding one or a jump to another address. ROM stores the program i.e., number of instructions that are going to be implemented or executed.

ALU Control defines four combinations that are AND(0000) ,OR (0001), ADD(0010), SUBTRACT(0110). These are ALU control lines functions. 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. 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 directly controls the ALU by generating one of the 4-bit combinations.

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

The instruction, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the funct7 or funct3 fields; in this case, we say that we "don't care" about the value of the opcode, and the bits are shown as Xs. When the ALUOp value is 10, then the funct7 and funct3 fields are used to set the ALU control input.

This table shows how ALU bits are sets depends on the ALUop control bits and the different opcodes for the R-type instruction. ALU control inputs based on the 2-bit ALUOp control, funct7, and funct3 fields. **Control unit** generates the ALUOp bits, which then are used as input to the ALU control that generates the actual signals to control the ALU unit. 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. There are several different ways to implement the mapping from the 2-bit ALUOp field and the funct fields to the four ALU operation control bits. Because only a small number of the possible funct field values are of interest and funct fields are used only when the ALUOp bits equal 10, we can use a small piece of logic that recognizes the subset of possible values and generates the appropriate ALU control signals. As a step in designing this logic, it is useful to create a truth table for the interesting combinations of funct fields and the ALUOp signals, this truth table shows how the 4-bit ALU control is set depending on these input fields. Since the full truth table is very large, and we don't care about the value of the ALU control for many of these input combinations, we show only the truth table entries for which the ALU control must have a specific value.

ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

Truth table isa representation of a logical operation by listing all the values of the inputs and then in each case showing what the resulting outputs should be.
Note that the only bits with different values for the four R-format instructions are bits 30, 14, 13, and 12. Thus, we only need these four funct field bits as input for ALU control instead of all 10.

A don’t-care term in this truth table (represented by an X in an input column) indicates that the output does not depend on the value of the input corresponding to that column. For example, when the ALUOp bits are 00, as in the first row, we always set the ALU control to 0010, independent of the funct fields. In this case, then the funct inputs will be don’t cares in this line of the truth table. Once the truth table has been constructed, it can be optimized and then turned into gates.
For Main Control Unit we have to understand the data path and for it we need to get familiar with four instruction classes arithmetic, load, store, and conditional branch instructions.

Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

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. 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.
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. 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 load store instructions. The destination register is always in bit positions 11:7 (rd) for Rtype instructions and load instructions.

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes.
The control unit can set all but one of the control signals based solely on the opcode and funct fields of the instruction. The PCSrc control line is the exception. That control line should be asserted if the instruction is branch if equal (a decision that the control unit can make) and the Zero output of the ALU, which is used for the equality test, is asserted. To generate the PCSrc signal, we will need to AND together a signal from the control unit, which we call Branch, with the Zero signal out of the ALU. These eight control signals (six from and two for ALUOp) can now be set based on the input signals to the control unit, which are the opcode bits 6:0.

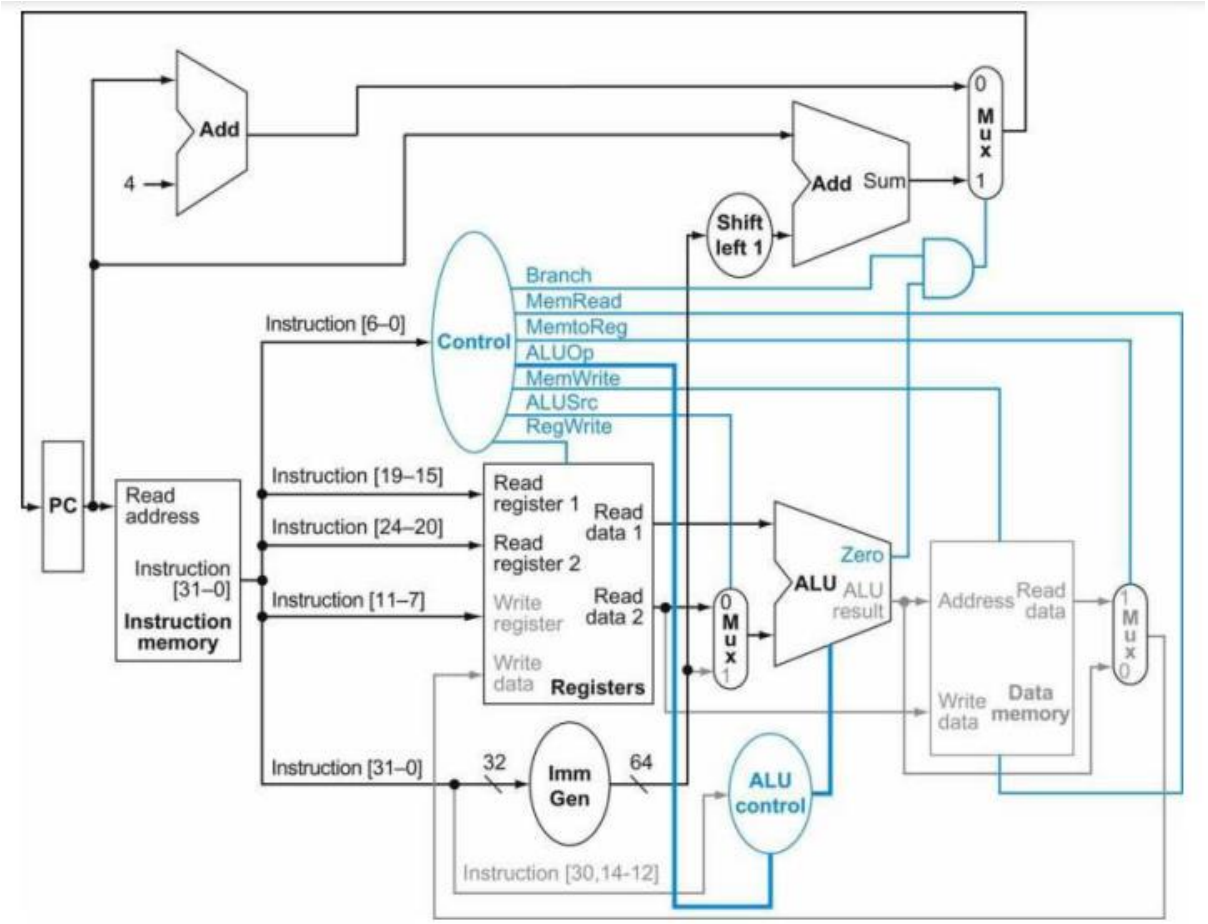
The input to the control unit is the 7-bit opcode field from the instruction. The outputs of the control unit consist of two 1-bit signals that are used to control multiplexors (ALUSrc and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU the AND gate output controls the selection of the next PC. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Before we try to write a set of equations or a truth table for the control unit, it will be useful to try to define the control function informally. Because the setting of the control lines depends only on the opcode, we define whether each control signal should be 0, 1, or don't care (X) for each of the opcode values. How the control signals should be set for each opcode; this information follows directly.The setting of the control lines is completely determined by the opcode fields of the instruction.

Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

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. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

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 datapath in operation for an Rtype instruction, such as add x1, x2, x3. The control lines, datapath units, and connections that are active are highlighted. Similarly, we can illustrate the execution of a load register, such as ld x1, offset(x2) in a style similar to the active functional units and asserted control lines for a load.
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 datapath in operation for a load instruction. The control lines, datapath units, and connections that are active are highlighted. A store instruction would operate very similarly. The main difference would be that the memory control would indicate a write rather than a read, the second register value read would be used for the data to store, and the operation of writing the data memory value to the register file would not occure.
Finally, we can show 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 sign extended, 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. The data path in operation for a branch-if-equal instruction.



Note that this design is for a 64 bit risc-v processor but we are implementing for 32 bit risc-v processor so this pic is only attached to understand the general structure design rv-32i.

Now finally is to verify your control unit and alu by checking the instructions behavior. The outputs are the control lines, and the inputs are the opcode bits. Thus, we can create a truth table for each of the outputs based on the binary encoding of the opcodes.The logic in the control unit as one large truth table that combines all the outputs and that uses the opcode bits as inputs. It completely specifies the control function, and we can implement it directly in gates in an automated fashion.

Input or output	Signal name	R-format	ld	sd	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

The control function for the simple single-cycle implementation is completely specified by this truth table.

The top half of the table gives the combinations of input signals that correspond to the four instruction classes, one per column, that determine the control output settings. The bottom portion of the table gives the outputs for each of the four opcodes. Thus, the output RegWrite is asserted for two different combinations of the inputs. If we consider only the four opcodes shown in this table, then we can simplify the truth table by using don't cares in the input portion. For example, we can detect an R-format instruction with the expression Op4 · Op5, since this is sufficient to distinguish the R-format instructions from ld, sd, and beq. We do not take advantage of this simplification, since the rest of the RISC-V opcodes are used in a full implementation.

Immediate selector selects the immediate to be given input to the ALU. The RV32I core format possesses only two types of immediate either 12-bit or 20-bit.

Branch selector determines the branch operations because the operations are on either true or false condition, that further initiates the instruction on which it has to be jumped.

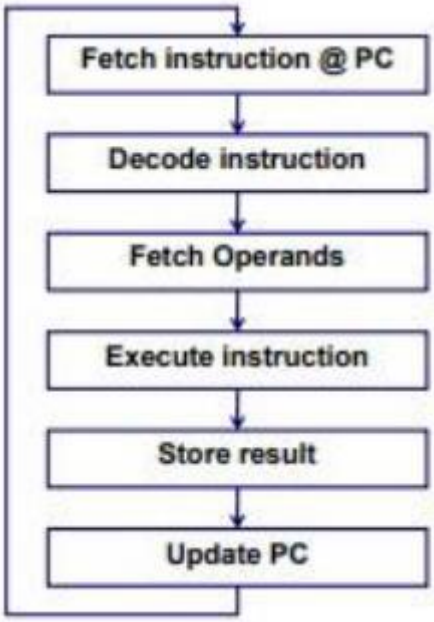
RAM is the data memory on which the store and load commands are executed. It stores data on its specific address

Store

This circuit is enabling the store pin in the RAM.

Load
This circuit is enabling the load pin in the RAM.

Operand A Selector
Basically, the operand A can be selected between rs1 from register file, current PC from U type, PC with the increment of 4 for jump commands and simply zero. This circuit is designed to select single output from these for which this is connected to the multiplexer that is connected with these four inputs providing only a single output to operand A at the instance



Operand B Selector
This selects whether the operand B is to be selected from rs2 or from the immediate generator.

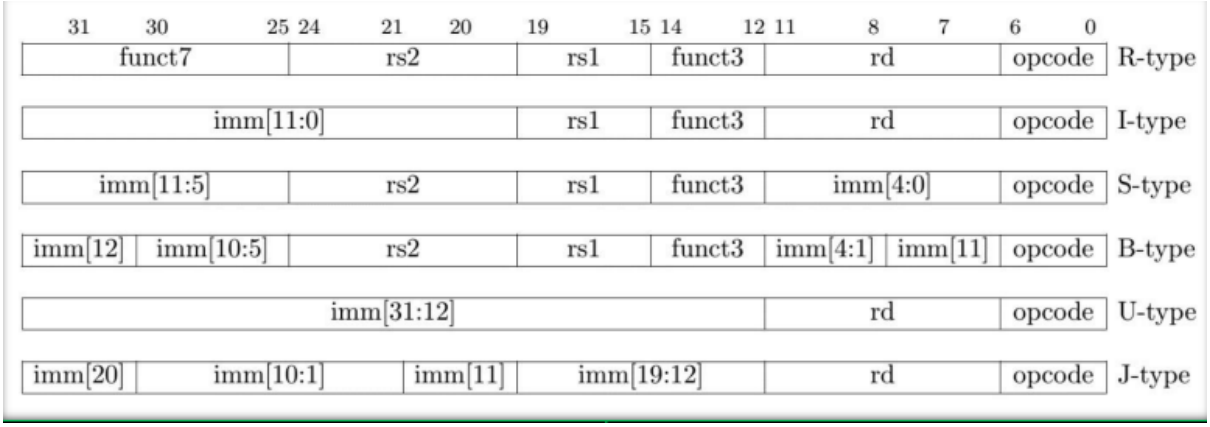
Next Pc select circuit is specially designed for the PC as there are few instructions on which the next instruction that is to be executed is not the preceding one but there is a skip of number of instructions and a jump that can be seen in SB, U and UJ types. This enables to select from these three either or simply provides PC+4 to the PC.

ALU Operations Selector
As it is previously mentioned that ALU is one of the main components in the circuit, therefore the importance of this component is described by its construction that this circuit accepts the func_3 and func_7 that is being transmitted into control unit for this along with the personally assigned opcodes of 3 bits for the supported instructions. The output of this circuit is actually the input of the ALU operation select pin.

The verification process is not a hard nut to crack. In the software, after the code has been written, it is being executed stepwise from which the results are obtained in decimal system. In the hardware, we have to connect a 32-bit output from the pin or bus and the result is matched by converting the decimal values into binary system. The ease that we get is that if the concepts and views are clear then we can have more further check and balance connecting the outputs to any wire for more clarification.

In a basic single-cycle implementation all operations take the same amount of time a single cycle. All instructions will execute in the same amount of time; this will determine the clock cycle time for our performance equations. Firstly, the program counter or PC register holds the address of the current instruction and updated at the end of every clock cycle. Secondly, we have instruction memory. Instruction memory is a state element that provides read-access to the instructions of a program. Thirdly, the control unit is responsible for setting all the control signals so that each instruction is executed properly. Type decode specify the types of instruction. Control Decoder depending upon the type of instruction that specifies a particular arithmetic operation. Fourthly, immediate generation

that uses complete instruction and PC to generate the relevant immediate then register file stores thirty-two 32bit values. The ALU performs the desired operation. Its result is stored in the destination register. Built a 32-bit ALU with 4- bit ALU operation Select. It performs the arithmetic or logical operation specified by the instruction. Add RAM and set its data bits to 32 bits and address width to 12 bits. Memory requires 32 (address) bits and 12 (write data) bits and just one output 32 bits. Finally, built a branch circuit to control branch instructions if branch function is true so to fetch the next instruction sequentially. Similar to branch, the jump instruction requires operand, immediate and perform add function so that why required 32- bit adder for jalr. Connect the wiring using the splitters, multiplexers, constants, tunnels, and clocks.



Reference:

Computer%20Organization%20and%20Design%20RISC-V%20edition.pdf