University of Moratuwa

Circuits and Systems
EN3030
Team Zigma

RISC-V processor Design

Group members :
Jayashinghe D.R. 190262L
Mathotaarachchi M.M. 190388D
Rukmal M.A.D. 190531L
Udara A.G.N. 190636M

Lecturer :
Ajith Pasqual

12/02/2023

Contents

1 Introduction **2**
- 1.1 Functionality . . . 2

2 Method **3**
- 2.1 Program counter . . . 3
- 2.2 PC selection and branching . . . 4
- 2.3 CLK and input management . . . 5
- 2.4 Instruction Memory . . . 6
- 2.5 Instruction Decode . . . 6
- 2.6 Register Bank . . . 6
- 2.7 ALU input selection . . . 7
- 2.8 ALU . . . 8
- 2.9 Data Memory . . . 8

3 Control Signals **9**
- 3.1 Main Controller . . . 9
- 3.2 ALU Controller . . . 10

1 Introduction

A processor is an integrated electronic circuit that performs the calculations that run a computer. A processor performs arithmetical, logical, input/output, and other basic instructions that are passed from an operating system. Most other processes are dependent on the operations of a processor.

As a fulfillment for EN3030, for our final project, we were tasked with designing and implementing a RISC-V RV32I processor with a direct mapping cache having victim cache functionality. This document is the Final fulfillment of our project.

1.1 Functionality

We designed our processor to support single-cycle instructions and the instruction types that the processor supports are as follows.

- R Type
- I Type
- S Type
- SB Type
- U Type
- UJ Type

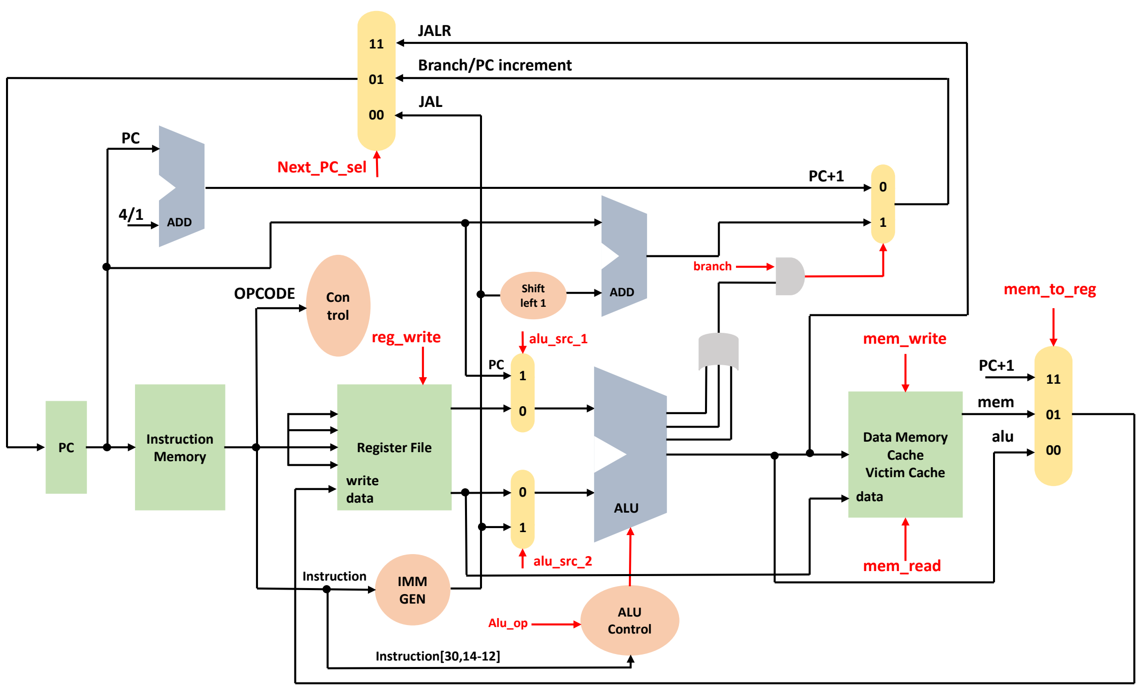

Figure 1: Datapath

2 Method

Our design comprises with following sub parts.

- Program counter
- PC selection and branching
- CLK input and management
- Main Control Unit
- Instruction Memory
- Instruction Decode
- Register File
- ALU input selection
- ALU
- ALU Control
- Data Memory

2.1 Program counter

In RISC-V, the program counter (PC) is a special register that holds the address of the instruction being executed. This is 32-bit in size. The PC is automatically incremented by 4 (for 32-bit implementations) after each instruction is executed, to keep track of the sequence of instructions in the program.

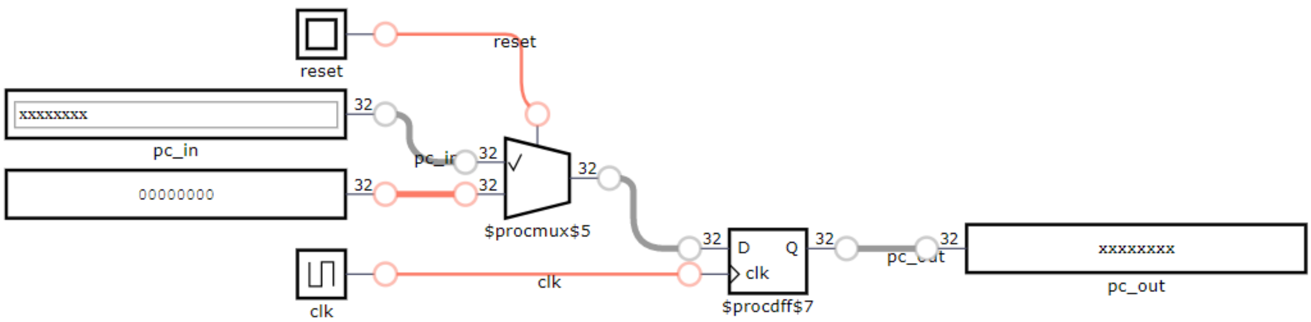

Figure 2: PC RTL view

The RISC-V architecture provides a set of instructions for manipulating the PC, including jump and branch instructions.The branch instructions can be used to change the flow of execution in a program by updating the PC with the address of a target instruction. In RISC-V, the PC is also used to implement subroutines and other control structures. When a subroutine is called, the current value of the PC is pushed onto the stack, and the PC is updated with the address of the first instruction in the subroutine. When the subroutine returns, the PC is popped from the stack, returning control to the instruction called the subroutine.

In our implementation, PC is initiated to 0 and we have designed the instruction memory with the memory width of a word. So after the instruction is executed we need to increment the PC only by 1 (not by 4). The triggering signal to the PC is the clock. At the positive edge of the clock, the PC register gives the next PC value as the output. For increment the PC, a simple adder is used. This adder adds 1 to the current value of the PC and generates the new PC value.

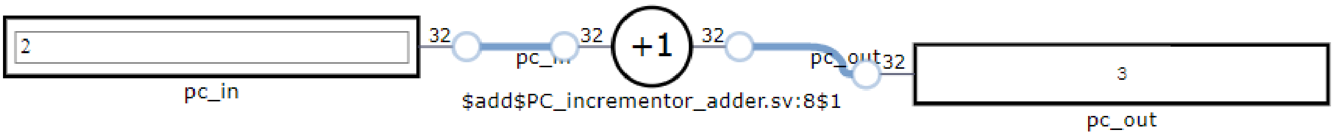

Figure 3: PC incrementor RTL view

2.2 PC selection and branching

Branching is a fundamental aspect of computer architecture, and it refers to the process of determining the next instruction to be executed based on the outcome of a previous instruction. In the RISC-V architecture, branching is performed using the program counter (PC) register.

In RISC-V, branching can occur in two ways: unconditional branching and conditional branching.

- Unconditional Branching: In unconditional branching, the program execution is transferred to a specified address regardless of the outcome of any previous instruction. This is achieved by loading the desired target address into the PC register."jal" and "jalr" are the two instructions responsible for this.
- Conditional Branching: In conditional branching, the program execution is transferred to a specified address only if a certain condition is met. This is achieved using conditional branches, such as "beq" (branch if equal) or "bne" (branch if not equal). These instructions compare the values in two registers, and based on the result of the comparison, the PC register is updated with the target address or the next instruction address is fetched from the current PC value.

In both cases, the branching operation is performed by updating the value in the PC register. So there should be a separate adder to increment the PC count by a supplied value from the instruction. For that, an adder was implemented to add the immediate offset to the current PC value and calculate the next PC value.

For branching instructions, we raised three flags from ALU.

- "eq" - branch if r1 and r2 register values are equal.
- "a_lt_b" - branch if r1 value is less than r2 value .
- "a_lt_ub" - branch if unsigned r1 value less than unsigned r2 value.

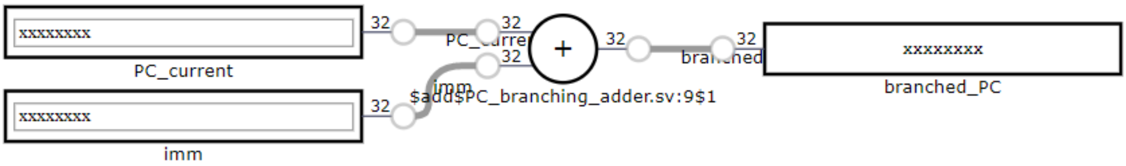

Figure 4: Immediate offset adder to PC RTL view

The logic was designed to generate a branching command if at least one flag is raised with branch control signal from the control unit.

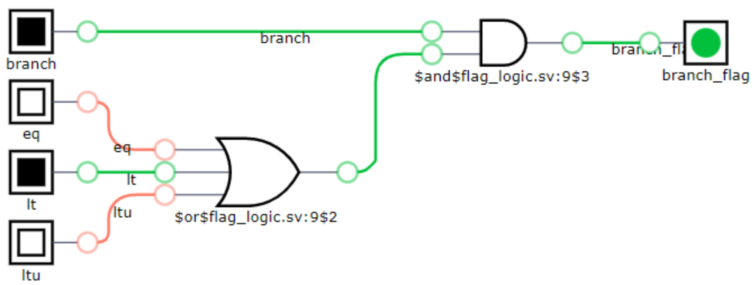

Figure 5: Branching logic RTL view

The branching command is used to select how next PC address is calculated with a help of multiplexer.

Branch command	**Operation**
0	PC increment by 1
1	PC shift by given offset

Other than branching instructions, unconditional jump instructions also change the PC value by a given amount. So we had to choose the next PC value from "branched_PC", "JAL" and "JALR". For this, we generated 'next_pc_sel' command from control logic and gives the correct next PC value to PC with the help of another multiplexer.

next_pc_sel	**next_PC**
10	JALR
00	PC+imm or PC+1
01	JAL

2.3 CLK and input management

The inputs for the RISC V are,

- Input clock.
- Reset.

- Instructions.

The Input clock provides timing sense for the RISC V and the reset pin input provides the reset functionality for the register bank and PC counter. It will initiate the modules to be 0. The instructions are provided via a .mif file containing HEX codes for the program. You can pass the instruction data in the real-time via Quartus In-System Memory Content Editor. The memory is created using an IP core of a 1 port RAM. The clock is provided via a high 50MHZ for the module to get combinations logic behaviour. The base clock for the other modules are 50KHZ. A clock divider is used for the clock slowing. We have to downgrade the clock to provide enough timing for the combinations delays.

2.4 Instruction Memory

A Single clocked, 1 port RAM IP core from the Quartus to use an inbuilt memory module. This provides the ability to Read and write data to the registers in run time. There will be 32Bit instructions and 1024 total instructions at a time. This can be extended if there is need to be.

2.5 Instruction Decode

With the different type of instructions, the instruction encoding is different. We can differentiate them mostly by the 7 bit opcode. We have identified 9 types of instructions,

- Arithmetic and Shift Operations based on R instructions.
- Immediate Operations with Arithmetic and logic operations.
- Load Operations, with similar syntax to Immediate Operations.
- Store Operations.
- Branch - Jump Operations.
- J type.
- JALR with similar syntax to Immediate Operations.
- Upper type, LUI.
- Upper AUIPC.

2.6 Register Bank

With Total of 32 registers, our design have only one hardwired register which provides integer 0, located at address 0. All other registers are to be used as anyway user like. There are no constrains.

Decoded data from instructions will provide input addresses for the module. The Write enable is provided by the Main Control Unit. With a clock event, after the all calculations are done, we will write to the provided address, if the instruction is required to do so.

Inputs	
Register Address 1	For register output selection
Register Address 2	For register output selection
Write Address	Where to be stored the data
Write Enable	If writing needs to be done
Write Data	The data which need to be stored
Clock	Timing event for the Data writing process
Outputs	
Data 1	Data stored at location of Address 1
Data 2	Data stored at location of Address 2

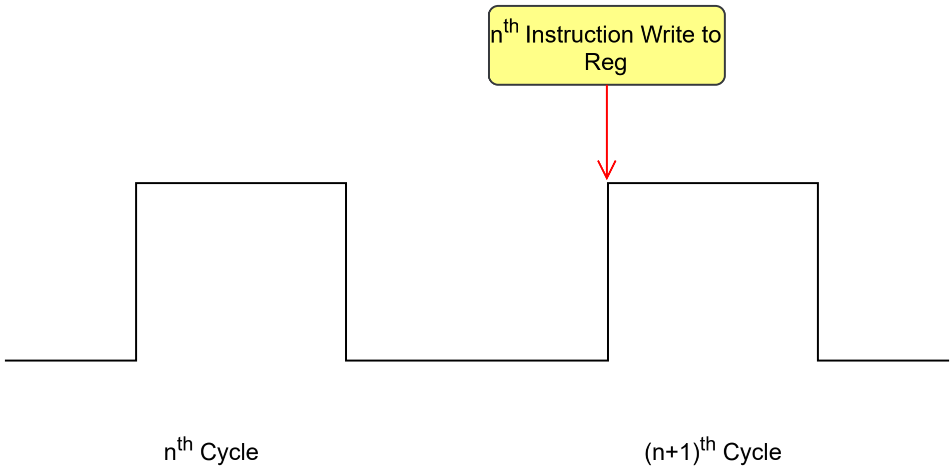

Figure 6: Clock Details for reg

In the single-cycle design, we have to restrict clock usage as much as possible. Here, the clock trigger is used only for register write.

Since the PC increment also happen at the same clock trigger, with the path delays, the access of the written data from the previous instruction writing will not be a problem.

2.7 ALU input selection

For the inputs of ALU, We have to choose between 4 inputs.

- Register input 1
- PC
- Register input 2
- Immediate value

There are two independent 2×1 multiplexers to Select for two inputs of the ALU unit. The multiplexers are controlled with 2 flags provided by the control unit and the flags are dependent on the opcodes. For further clarifications refer to the Main controller section.

2.8 ALU

To execute all the Arithmetic and Logic tasks except for the PC increment and branching increment, we rely on the ALU of our processor.
It has total of 11 tasks. Getting control signals from ALU control unit with help of 4bit *FCODE*, we can differentiate those unique tasks.

Task	**FCODE**
Add Two Numbers	0000
Subtract Two Numbers	0001
Left Shift One Number	0010
Signed Compare	0011
Unsigned Compare	0100
XOR bitwise	0101
Right Shift	0110
Signed Right Shift	0111
OR	1000
AND	1001
Address Save and Jump	1010

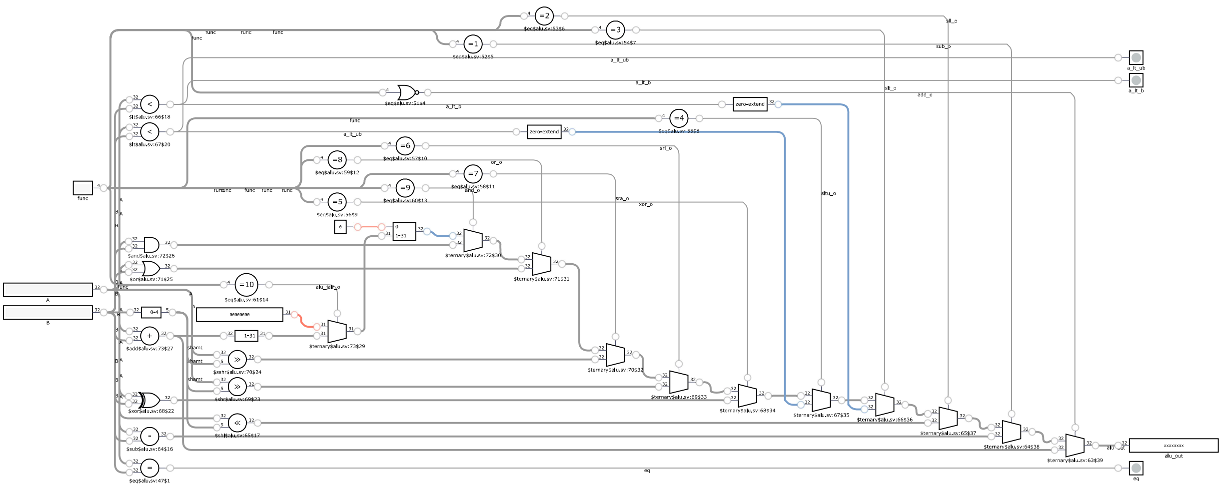

Figure 7: ALU RLT view

2.9 Data Memory

RISC-V is a load-store architecture, which means that all data processing operations are performed on data stored in registers. The data memory of a RISC-V processor consists of two types of memory: cache memory(with victim cache functionality) and main memory. Cache memory is a small, fast memory that is used to temporarily store frequently accessed data. It is typically divided into multiple levels, with each level having a larger size and a longer access time than the previous level. The purpose of the cache memory is to reduce the time it takes to access data from the main memory, which is slower. Main memory is the primary storage area for data in a RISC-V processor. It is typically

composed of DRAM (dynamic random access memory) chips and is organized into a hierarchy of memory banks, with each bank consisting of multiple memory modules. The size and speed of the main memory depend on the specific implementation of the RISC-V processor.
A victim cache is a hardware cache designed to decrease conflict misses and improve hit latency for direct-mapped caches. It is employed at the refill path of a Level 1 cache, such that any cache line which gets evicted from the cache is cached in the victim cache. The RISC-V architecture provides instructions for loading data from memory into registers and storing data from registers into memory. These instructions can be used to read and write data from the cache memory and main memory.
Two control signals and two inputs (one from ALU and one from the register file) come into the data memory and one output comes out from the data memory.

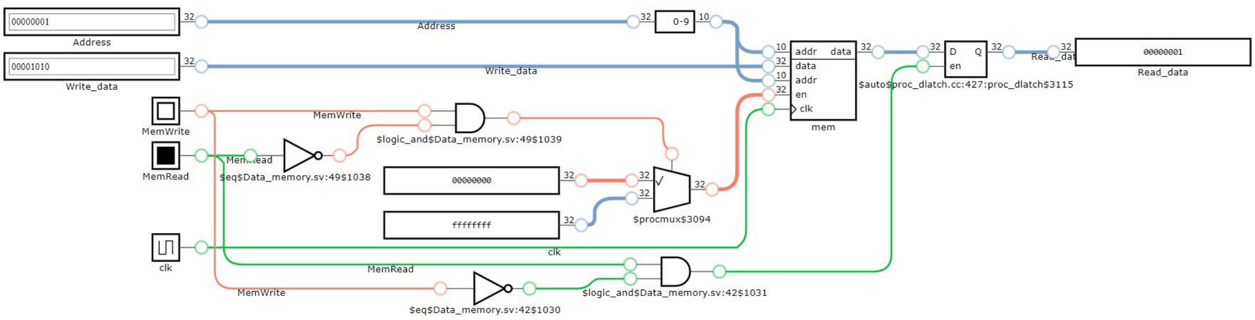

Figure 8: Data Memory RTL view

3 Control Signals

The control signals for the datapath are generated by 2 units; main controller and ALU controller.

3.1 Main Controller

The main controller gives the necessary control signals to the modules with the single cycle datapath. The main control unit takes as input the 7-bit opcode from the instruction and it outputs 9 control signals. The following table contains the control signal types and their destinations.

Control Signal	Signal bit size	Function
alu_src_1	1	Select the first input to the ALU (data from register file or PC)
alu_src_2	1	Select the second input to the ALU (data from register file or immediate value from instruction)
mem_to_reg	2	Select which data is written to the register file at the mux (data from memory, ALU output or incremented PC)
reg_write	1	Enable data write to registerfile
mem_read	1	Enable memory read for load instruction
mem_write	1	Enable memory write for store instruction
branch	1	Enable branching
alu_op	2	This signal is sent to the ALU controller indicating the ALU operation type
next_pc_sel	2	Selects the next value of the PC

Table 1: Main control signals

3.2 ALU Controller

The ALU controller takes 2 inputs. One input is the 2-bit *alu_op* which comes from the main control unit and other inputs is 4 bits decoded from the intructions. These 4 bits comes from the *function 3 (Instruction[14:12])* and the *6th bit of function 7 (Instruction[30])* in the instruction. Based on the input signals the ALU controller will output a 4-bit signal which will correspond to the ALU function required by the instruction such as addition, subtraction, bit shifting, comparison, etc.

The ALU OP from main controller categorises the signals into 4 types based on the opcode.

ALU Op	Relevant instructions
00	I-type (load) and S-type
01	SB-type (branching)
10	R-type and I-type (execpt load)
11	U-type and J-type

Table 2: ALU OP signal taken as input to ALU controller

Mnemonic	Instruction	Description
R type		
ADD rd, rs1, rs2	Addition	rd <= rs2 + rs1
SUB rd, rs1, rs2	Subtraction	rd <= rs2 - rs1
SLL rd, rs1, rs2	Shift Left Logic	rd <= rs1 <<< rs2 (only by lower 5 bits in rs2)
SLT rd, rs1, rs2	Set Less Than	if rs1 < rs2: rd <= 1, if rs1 >= rs2: rd <= 0
SLTU rd, rs1, rs2	Set Less Than Unsigned	if rs1 < unsigned(rs2): rd <= 1, if rs1 >= unsigned(rs2): rd <= 0
XOR rd, rs1, rs2	XOR Logic	rd <= rs1 ^ rs2
SRL rd, rs1, rs2	Shift Right Logic	rd <= rs1 >>> rs2 (only by lower 5 bits in rs2)
SRA rd, rs1, rs2	Shift Right Arithmetic	rd <= rs1 >> rs2 (only by lower 5 bits in rs2)
OR rd, rs1, rs2	OR Logic	rd <= rs1 | rs2
AND rd, rs1, rs2	AND Logic	rd <= rs1 & rs2
I type		
ADDI rd, rs1, imm	Add Immediate	rd <= rs1 + imm
SLTI rd, rs1, imm	Set Less Than Immediate	if rs1 < imm: rd <= 1, If rs1 >= imm: rd <= 0
SLTIU rd, rs1, imm	Set Less Than Immediate Unsigned	if rs1 < unsigned(imm): rd <= 1, If rs1 >= unsigned(imm): rd <= 0

XORI rd, rs1, imm	XOR Immediate	rd <= rs1 ^ imm
ORI rd, rs1, imm	OR Immediate	rd <= rs1 | imm
ANDI rd, rs1, imm	AND Immediate	rd <= rs1 & imm
SLLI rd, rs1, shamt	Shift Left Logic Immediate	rd <= rs1 <<< shamt
SRLI rd, rs1, shamt	Shift Right Logic Immediate	rd <= rs1 >>> shamt
SRAI rd, rs1, shamt	Shift Right Arithmetic Immediate	rd <= rs1 >> shamt
JALR rd, imm(rs1)	Jump And Link Register	Rd <= PC + 4 PC <= rs1 + imm
Load Instructions (I type)		
LW rd, imm(rs1)	Load word	rd <= mem[rs1 + imm]
LBU rd, imm(rs1)	Load Byte Unsigned	rd <= mem[rs1 + imm] The fetched value from memory is zero
LHU rd, imm(rs1)	Load Halfword Unsigned	rd <= mem[rs1 + imm] The fetched value from memory is zero extended to make it as 32 bit
S type		
SW rs2, imm(rs1)	Store Word	rs2[31:0] <= mem[rs1 + imm]
SB type		
BEQ rs1, rs2, imm	Branch Equal	if rs1 = rs2: PC <= PC + imm*4 else: PC <= PC + 4
BNE rs1, rs2, imm	Branch Not Equal	if rs1 != rs2: PC <= PC + imm*4 else: PC <= PC + 4

BLT rs1, rs2, imm	Branch Less Than	if rs1 < rs2: PC <= PC + imm*4 else: PC <= PC + 4
BGE rs1, rs2, imm	Branch Greater Than or Equal	if rs1 >= rs2: PC <= PC + imm*4 else: PC <= PC + 4
BLTU rs1, rs2, imm	Branch Less Than Unsigned	if unsigned(rs1) < unsigned(rs2): PC <= PC + imm*4 else: PC <= PC + 4
BGEU rs1, rs2, imm	Branch Greater Than or Equal Unsigned	if unsigned(rs1) >= unsigned(rs2): PC <= PC + imm*4 else: PC <= PC + 4
U type		
LUI rd, imm	Load Upper Immediate	rd[31:12] <= imm rd[11:0] <= 12'b0
AUIPC rd, imm	Add Upper Immediate to PC	rd <= PC + offset (offset[32] = imm[20] + zero[12])
UJ type		
JAL rd, imm	Jump And Link	rd <= PC + 4 PC = PC + imm