# Project Report: Group 3

**Group Members:** Anugu Arun Reddy, Ayush Modi, Anish Karnik

**Project Title:** Develop a simulator for the SDLX processor. The simulator shall take the instructions from a file containing a sequence of 32-bit instruction codes and execute the program. The output shall be generated using a memory-mapped IO displayed on the screen.

## Introduction

In this project, we used Python and its libraries, numpy and pandas, to develop a working simulator for the SDLX processor.

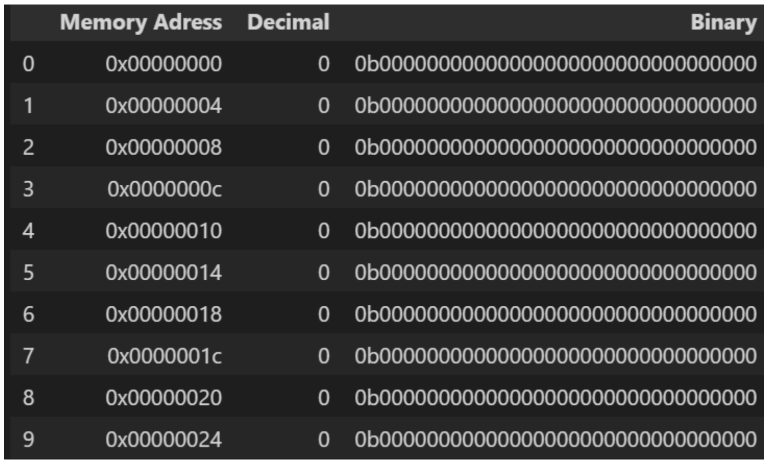
A simulator is a program used to simulate and test the working of a processor on a computer. Simulators are also widely used for debugging applications by hardware architects. This is done by giving various instructions into the simulator as inputs and testing the respective contents of the register file and the memory for the given instructions.

The simulator we have developed displays and updates the contents of the register file, the memory, and the Program Counter after all the instructions given as input through a file are executed by the SDLX processor. The entire methodology and the working of the Simulator are explained in the following sections.

## Methodology

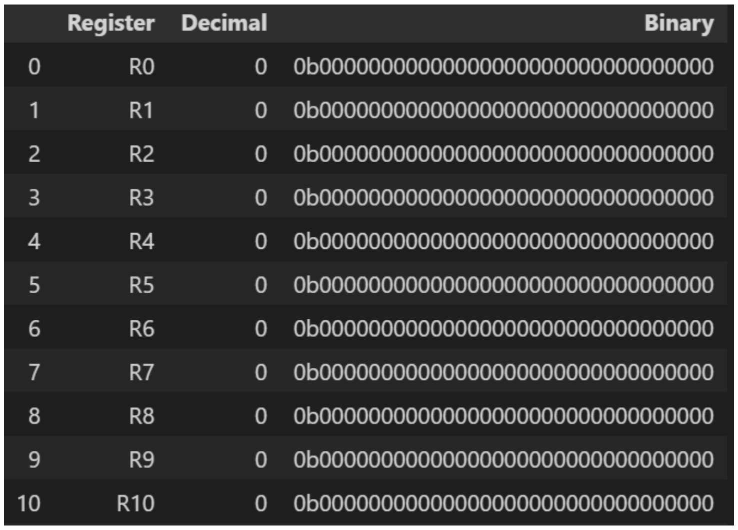
- The implementation of SDLX is done by making various functions in Python. Libraries such as pandas and NumPy are used to create data frames and arrays.

- The memory is made using a data frame. Memory address, data stored in 32 bits, and the value stored in decimal are the arrays used as columns in the data frame. We initially considered our memory to be (10 x 4 )bytes. We can extend the memory size, but it provides convenience in selecting a smaller memory when there are few instructions to be executed. Initially, all the data in the memory is stored as 0. We would be using memory for loading and storing instructions. Below is the image of the display of the memory in the simulation.



| | Memory Adress | Decimal | Binary |
|---|---|---|---|
| 0 | 0x00000000 | 0 | 0b00000000000000000000000000000000 |
| 1 | 0x00000004 | 0 | 0b00000000000000000000000000000000 |
| 2 | 0x00000008 | 0 | 0b00000000000000000000000000000000 |
| 3 | 0x0000000c | 0 | 0b00000000000000000000000000000000 |
| 4 | 0x00000010 | 0 | 0b00000000000000000000000000000000 |
| 5 | 0x00000014 | 0 | 0b00000000000000000000000000000000 |
| 6 | 0x00000018 | 0 | 0b00000000000000000000000000000000 |
| 7 | 0x0000001c | 0 | 0b00000000000000000000000000000000 |
| 8 | 0x00000020 | 0 | 0b00000000000000000000000000000000 |
| 9 | 0x00000024 | 0 | 0b00000000000000000000000000000000 |

- The register file is also made using a data frame. Register Number, data stored in 32 bits, and the value stored in decimal are the arrays used as columns in the data frame. The register file is (34 x 4 )bytes, as in the SDLX processor. R0 is always kept as zero. R31 is used only for linking instructions. Most of the operations performed are on the registered file data. All the register file data is initially stored as 0; subsequent operations would change the file data. Below is the image of the display of the register file in the simulation.



| | Register | Decimal | Binary |
|---|---|---|---|
| 0 | R0 | 0 | 0b00000000000000000000000000000000 |
| 1 | R1 | 0 | 0b00000000000000000000000000000000 |
| 2 | R2 | 0 | 0b00000000000000000000000000000000 |
| 3 | R3 | 0 | 0b00000000000000000000000000000000 |
| 4 | R4 | 0 | 0b00000000000000000000000000000000 |
| 5 | R5 | 0 | 0b00000000000000000000000000000000 |
| 6 | R6 | 0 | 0b00000000000000000000000000000000 |
| 7 | R7 | 0 | 0b00000000000000000000000000000000 |
| 8 | R8 | 0 | 0b00000000000000000000000000000000 |
| 9 | R9 | 0 | 0b00000000000000000000000000000000 |
| 10 | R10 | 0 | 0b00000000000000000000000000000000 |

- We have initiated the PC to 0. The value of the PC increases by one on executing an instruction. The branching instructions also change the value of the PC depending on the value at register file and signed offset.

- The instructions are included in a CSV file. We write the 32-bit instructions in Excel in a different column. Then it is converted to a CSV file. Then we read this CSV file and append the 32-bit instruction into an array.

- If the memory and register files have data and we repeat the exact instructions, we get a different answer than expected. This is because the instructions we need to execute are executed before, and changes are made in memory and register files. So we need to restart the kernel and initiate all the values to zero. Then we again execute this set of instructions.

- Then we have written the code for the central simulating unit. This is a function that takes its input as instructions and makes changes in memory, register files, and PC. Depending upon the instructions, the function will execute them according to R type, dyadic, R-I type, and J type instructions.

- Finally, we print the results on the screen displaying memory, register file, and PC. We can also find the outputs of ALU operation for any instruction at any period.

- The simulator is coded in Python, so the outputs are instantaneous. Hazards occur when we want the results from previous instructions to be used in our current one and the result hasn’t arrived. But in this case, we are getting instantaneous results and won't get any hazards. The SDLX has a RAW hazard where we try to read after writing. If pipelining exists, then RAW hazard will appear.

## Working

- Here, we look at how we execute various instructions based on the given inputs.

- Firstly, the file consisting of all the 32-bit instructions is read, and then all the given instructions are stored in an array named the Instruction_array in the following manner:

```
Instruction_array = ["00000100101001010000000000000101",
"00000100101001010000000000000100", "01100100000001010000000000000100",
"01011000000000110000000000000100", "01000100101011010000000011111111",
"01001000000010100000000000000100"]
```

- Then, we iterate through the above Instruction_array and execute each of the instructions present in the array. This is done using a while loop which runs until the PC value becomes more than the length of the Instruction_array.

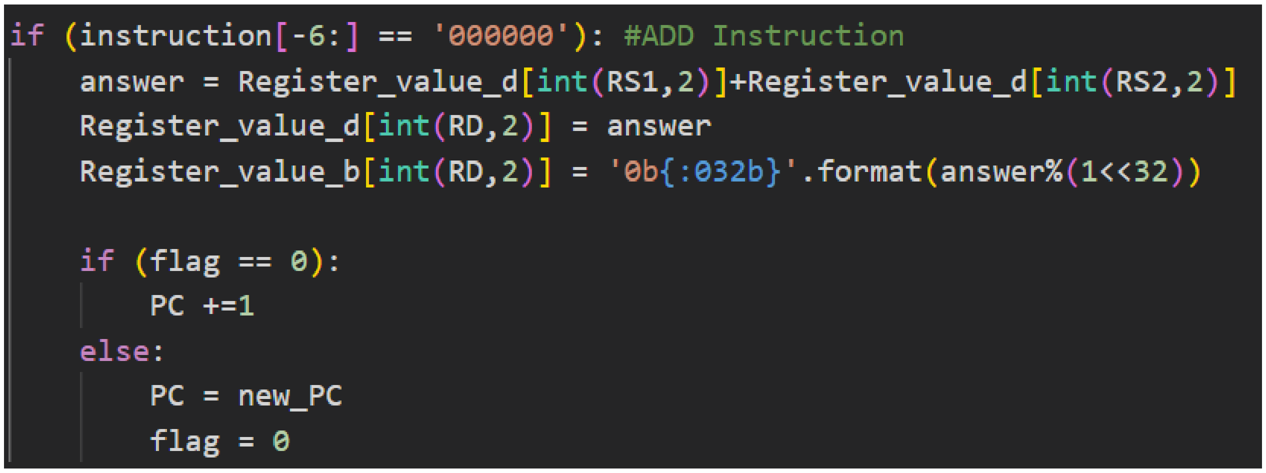
```
while(PC<len(Instruction_array)):
    instruction = Instruction_array[PC]
```

- The Program Counter, PC, is initialized with zero at the beginning.

```
PC = 0
```

- The new_PC is also a variable used when we want to do branching. As we always execute the instruction in PC+1 after branching, it is important to retain the value of PC. So, we store the value of the updated PC which we wanted into new_PC, and after the execution of the next instruction, we change the value of PC to new_PC. For branching, we have a flag indicating that we need to change the PC one instruction after branching.

- For the R-type instructions in SDLX, the instruction has to be defined in a particular sequence. Here, for all R-type instructions, we have maintained this sequence. The R-type instructions include ADD, SUB, AND, OR, XOR, and many more operations, which take two values from RS1 and RS2. The answer is stored in RD. The flag for all the R-type instructions is zero as there is no branching, and PC goes to PC+1.



```python
if (instruction[-6:] == '000000'): #ADD Instruction
    answer = Register_value_d[int(RS1,2)]+Register_value_d[int(RS2,2)]
    Register_value_d[int(RD,2)] = answer
    Register_value_b[int(RD,2)] = '0b{:032b}'.format(answer%(1<<32))

    if (flag == 0):
        PC +=1
    else:
        PC = new_PC
        flag = 0
```

*Fig1. General Code Snippet for R-type Instruction*

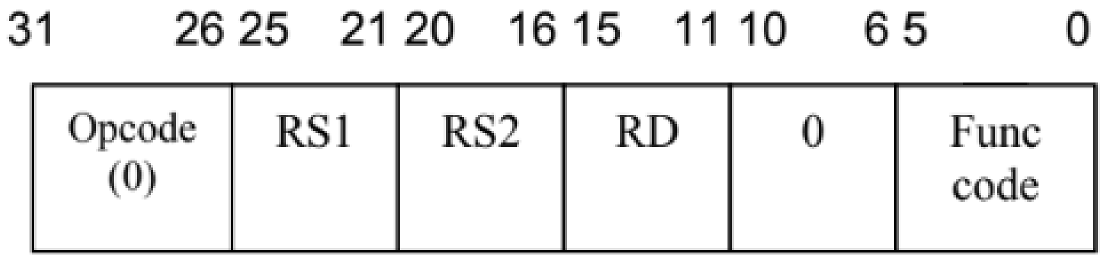
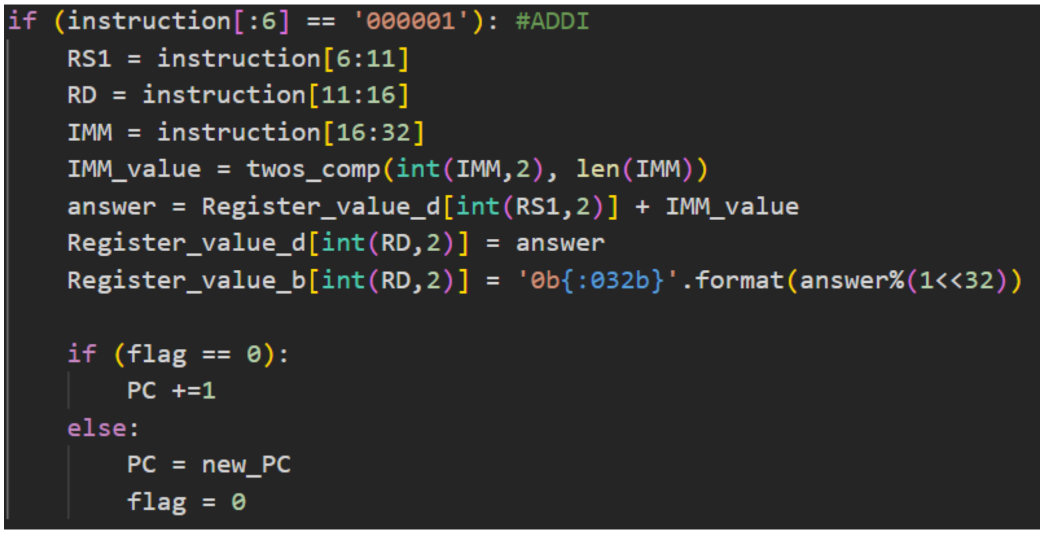


*Fig2.32 bit R type instruction, from ES 215 Notes: Design of SDLX Processor*

- For the R-I type instructions, we do operations of an immediate value with the value stored in RS1. This code includes function two's complement, which returns two's complement of a number. The immediate value also has to be sign-extended before we operate. After the operation, we store the result in the designated RD from the instruction. The R-I instructions also don't have branching, so we change don't change the flag and increment the PC by one.



```python
if (instruction[:6] == '000001'): #ADDI
    RS1 = instruction[6:11]
    RD = instruction[11:16]
    IMM = instruction[16:32]
    IMM_value = twos_comp(int(IMM,2), len(IMM))
    answer = Register_value_d[int(RS1,2)] + IMM_value
    Register_value_d[int(RD,2)] = answer
    Register_value_b[int(RD,2)] = '0b{:032b}'.format(answer%(1<<32))

    if (flag == 0):
        PC +=1
    else:
        PC = new_PC
        flag = 0
```

*Fig3. General Code Snippet for R-I type Instruction*

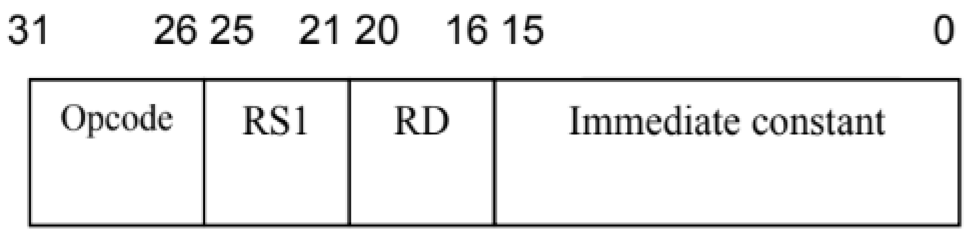
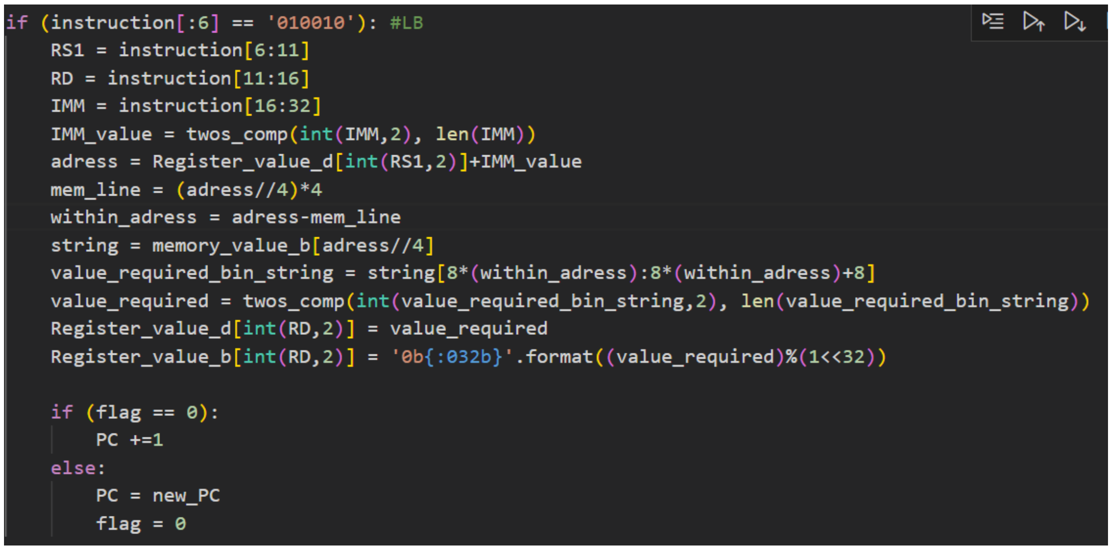


*Fig4.32 bit R-I type instruction, from ES 215 Notes: Design of SDLX Processor*

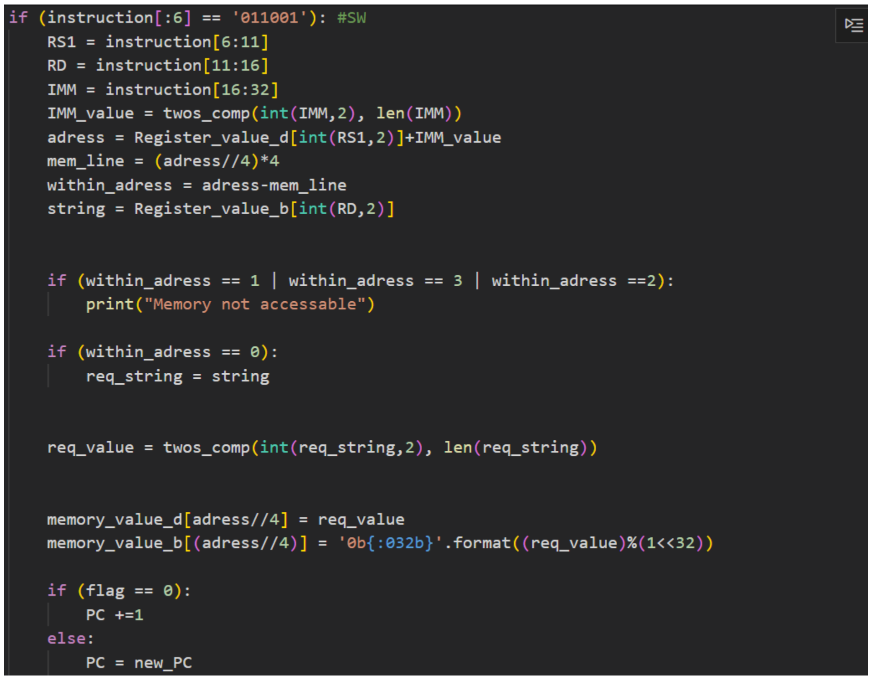
- Load is also R-I type instruction. The value at RS1 and the immediate value provide the address from which we have to take the value and load it in the register RD. Instructions like load byte, load halfword unsigned, load word, and many more are available in SDLX. If the user has given a memory address that is not accessible, then in this case, we print -Memory not accessible. To check this, we have a variable within_address which checks whether the memory address is following the instruction given. For example, in the case of a load half a byte, if the within_address is 1 or 3, it means that we can't load 16 bits( as it requires within_address to be 2 or 0). So, we print that memory is not accessible. For load word, within_address has to be 0. We can load a byte as we care for this condition using floor operation.



```
if (instruction[:6] == '010010'): #LB
    RS1 = instruction[6:11]
    RD = instruction[11:16]
    IMM = instruction[16:32]
    IMM_value = twos_comp(int(IMM,2), len(IMM))
    adress = Register_value_d[int(RS1,2)]+IMM_value
    mem_line = (adress//4)*4
    within_adress = adress-mem_line
    string = memory_value_b[adress//4]
    value_required_bin_string = string[8*(within_adress):8*(within_adress)+8]
    value_required = twos_comp(int(value_required_bin_string,2), len(value_required_bin_string))
    Register_value_d[int(RD,2)] = value_required
    Register_value_b[int(RD,2)] = '0b{:032b}'.format((value_required)%(1<<32))

    if (flag == 0):
        PC +=1
    else:
        PC = new_PC
        flag = 0
```

*Fig5. General Code Snippet for Load Instruction*

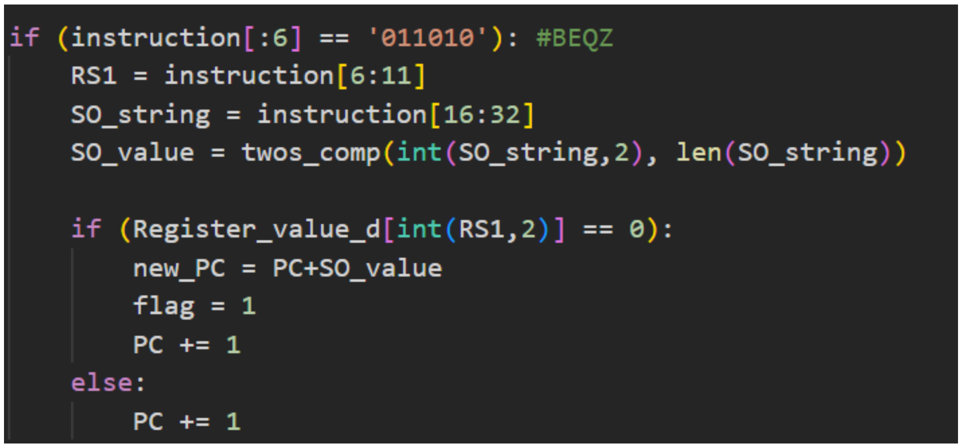
- Store is also R-I type instruction. The value at RS1 and the immediate value provide the address from which we have to take the value and load it in the register RD. Instructions like store byte, store half word, store byte, and many others are available in SDLX. Here also, we have implemented the same logic to verify if the memory to be accessible is following the instruction. The value is changed at the memory address given once it is verified. The flag need not be changed, and one store instructions increment PC.



```
if (instruction[:6] == '011001'): #SW
    RS1 = instruction[6:11]
    RD = instruction[11:16]
    IMM = instruction[16:32]
    IMM_value = twos_comp(int(IMM,2), len(IMM))
    adress = Register_value_d[int(RS1,2)]+IMM_value
    mem_line = (adress//4)*4
    within_adress = adress-mem_line
    string = Register_value_b[int(RD,2)]


    if (within_adress == 1 | within_adress == 3 | within_adress ==2):
        print("Memory not accessable")

    if (within_adress == 0):
        req_string = string


    req_value = twos_comp(int(req_string,2), len(req_string))


    memory_value_d[adress//4] = req_value
    memory_value_b[(adress//4)] = '0b{:032b}'.format((req_value)%(1<<32))

    if (flag == 0):
        PC +=1
    else:
        PC = new_PC
```

*Fig6. General Code Snippet for Store Instruction*

- The BNEZ and BEQZ are R-type dyadic instructions that change the PC value to PC value plus signed offset depending on the value at RS1. The signed offset is 16 bits which have to be sign-extended. If the conditions are true, we are changing the new_PC value to the PC value added to the signed offset and incrementing the PC. The flag also has to be changed to one. Here, we must implement the SDLX processor, which has a delayed branch. In the simulation, we considered a delayed branch and executed the next instruction after the branching instruction. If conditions are false, the flag and new_PC remain the same, and PC is incremented by one.



```
if (instruction[:6] == '011010'): #BEQZ
    RS1 = instruction[6:11]
    SO_string = instruction[16:32]
    SO_value = twos_comp(int(SO_string,2), len(SO_string))

    if (Register_value_d[int(RS1,2)] == 0):
        new_PC = PC+SO_value
        flag = 1
        PC += 1
    else:
        PC += 1
```

*Fig7. General Code Snippet for BNEZ and BEQZ Instruction*

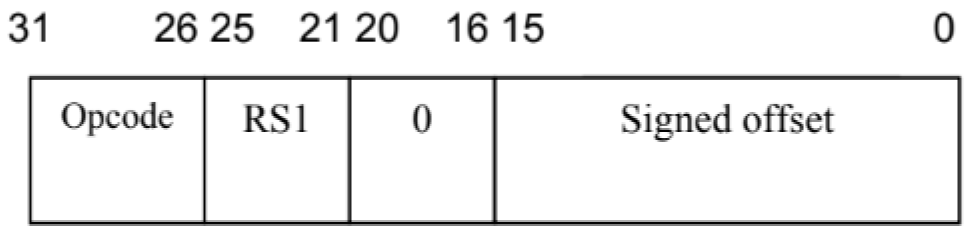
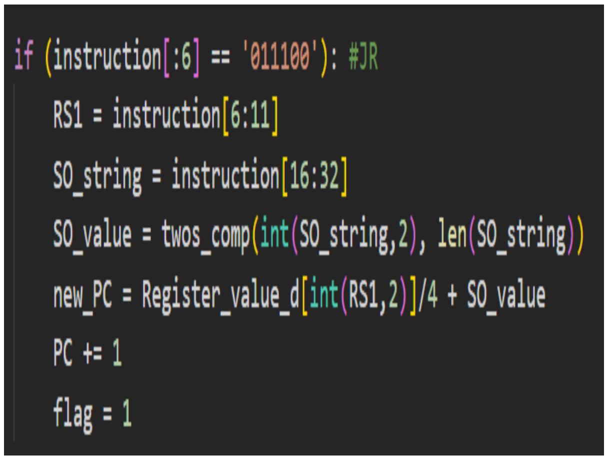
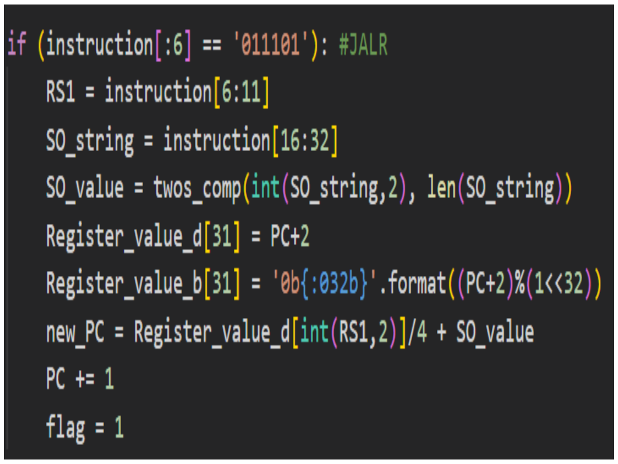


*Fig4.32 bit R-type dyadic instruction, from ES 215 Notes: Design of SDLX Processor*

- The JR and JALR are also R-type dyadic instructions. The next instruction is executed in these instructions, and then it jumps to the instruction at address RS1 added to 4 times signed offset. For these, the new_PC and flag have to be always updated. For JALR, the value of incremented PC by 2 has to be stored in R31. As J and JALR are branching instructions, new_PC and flag must be updated.

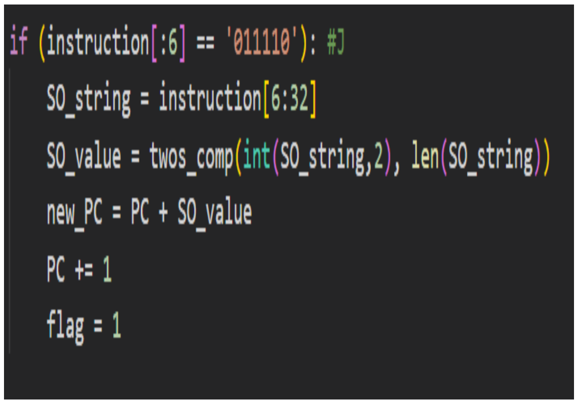


```
if (instruction[:6] == '011100'): #JR
    RS1 = instruction[6:11]
    SO_string = instruction[16:32]
    SO_value = twos_comp(int(SO_string,2), len(SO_string))
    new_PC = Register_value_d[int(RS1,2)]/4 + SO_value
    PC += 1
    flag = 1
```

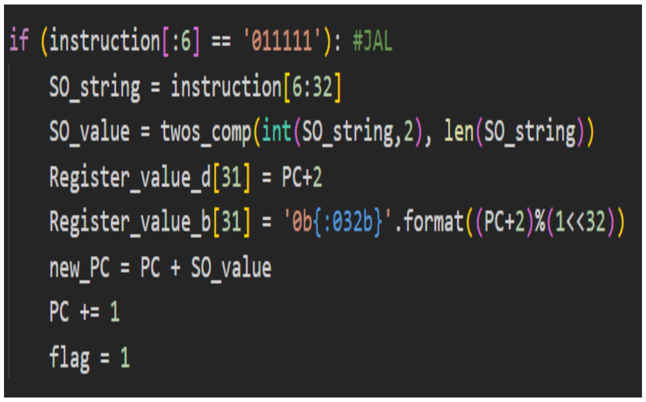


```
if (instruction[:6] == '011101'): #JALR
    RS1 = instruction[6:11]
    SO_string = instruction[16:32]
    SO_value = twos_comp(int(SO_string,2), len(SO_string))
    Register_value_d[31] = PC+2
    Register_value_b[31] = '0b{:032b}'.format((PC+2)%(1<<32))
    new_PC = Register_value_d[int(RS1,2)]/4 + SO_value
    PC += 1
    flag = 1
```

- The J instruction is used for unconditional jumping. The signed offset is 26 bits which have to be sign-extended. We are changing the new_PC value to the PC value added to the signed offset and incrementing the PC. The flag also has to be changed to one. For JAL, the value of incremented PC by 2 has to be stored in R31.



```
if (instruction[:6] == '011110'): #J
    SO_string = instruction[6:32]
    SO_value = twos_comp(int(SO_string,2), len(SO_string))
    new_PC = PC + SO_value
    PC += 1
    flag = 1
```



```
if (instruction[:6] == '011111'): #JAL
    SO_string = instruction[6:32]
    SO_value = twos_comp(int(SO_string,2), len(SO_string))
    Register_value_d[31] = PC+2
    Register_value_b[31] = '0b{:032b}'.format((PC+2)%(1<<32))
    new_PC = PC + SO_value
    PC += 1
    flag = 1
```