CS-204

Design Document : RISC-V 32I Simulator

Phase - 1

Group Number : 3

Ayush Verma	*2019CSB1147*
Bhumika	*2019CSB1152*
Keshav Krishna	*2019CSB1224*
Rishabh Jain	*2019CSB1286*
Vishwam Datta	*2019CSB1305*

RiscSim.py (Testbench) : -

This file basically calls all the necessary methods of different modules in different stages of instruction execution i.e different methods in fetch, decode ,execute ,memory access and register writeback.All the different outputs of different modules are linked as input of required modules in this file and this is basically a driver of the entire instruction and is responsible for the proper flow of data and control during execution of all the instructions.All the muxes (except those in the IAG module) are also simulated in this file.
Important Methods : -

- fetch - Contains all the connections between modules and method calls of the different modules that take place in the fetch stage of execution.
- decode - Contains all the connections between modules and method calls of the different modules that take place in the decode stage of execution.
- execute-Contains all the connections between modules and method calls of the different modules that take place in the execute stage of execution.
- mem_access-Contains all the connections between modules and method calls of the different modules that take place in the memory access stage of execution.
- reg_writeback-Contains all the connections between modules and method calls of the different modules that take place in the write back stage of execution.

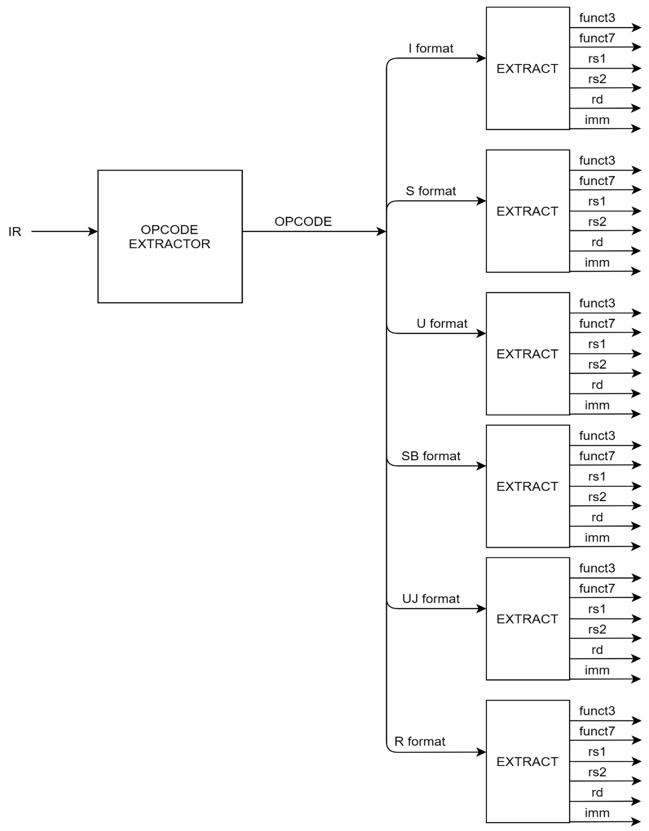
CONTROL CIRCUIT :

- **Decode Unit** :-
  Input: The input to the Decode unit is IR (instruction in hexadecimal format) and PC.
  The opcode of the machine instruction is calculated using bit masking.

If the opcode equals “0x11” (17), the program gets terminated.
Based on the opcode of the machine instruction, following functions are called which extract the respective fields (rs1, rs2, rd, function 3, function 7 and immediate value) from the respective instruction format :

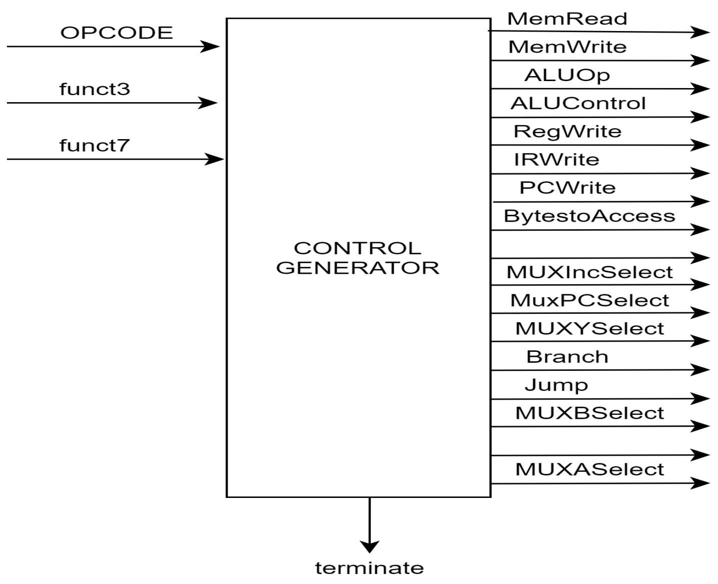
- decodeR # for R format instruction (add, and, or, sll, slt, sra, srl, sub, xor, mul, div, rem)
- decodeI # for I format instruction (addi, andi, ori, lb, lh, lw, jalr)
- decodeS # for S format instruction (sb, sw, sh)
- decodeSB # for SB format instruction (beq, bne, bge, blt)
- decodeU # for U format instruction (auipc, lui)
- decodeUJ # for UJ format instruction (jal)

In each of the above functions, bit masking is used to extract the fields from the machine code.
The immediate field is taken special care, as it is a signed value. Signed value is ensured by use of ‘&’ and ‘|’ logical operators.
Output: Control signal generator function is called by the decode unit after extracting the fields from the machine instructions.



- **Control Unit**:
The control path of the design is divided into two main units: Primary Control Signal Generator and ALU Control.

1. Control Signal Generator: It takes the opcode, funct3 of the instruction(fields identified by the decoder) and produces the following signals, as per the encoding:
    - **MemRead:** Boolean, decides if memory read is required
    - **MemWrite:** Boolean, decides whether to write to memory or not
    - **ALUOp:** Decides if ALU usage is required, since for certain instructions offset addition is handled by IAG
    - **RegWrite:** Controls whether to update register or not
    - **IRWrite:** Boolean, permits writing to IR- True only in fetch stage
    - **PcWrite:** Boolean, permits writing to PC- True only in execute stage
    - **BytesToAccess:** No of bytes accessed from memory(0/1/2/4)
    - **MuxASelect:** To select input1 of ALU from rs1 or PC(for auipc)
    - **MuxBSelect:** To select input2 of ALU from rs2 or imm value
    - **MuxYSelect:** Present at output of ALU, to select the value from RZ buffer, MDR or Return Address from IAG
    - **MuxINCSelect:** Present in the IAG to decide the value to add in the PC from 4(sequentially next PC) or the Branch offset
    - **MuxPCSelect:** Present in the IAG to choose input from the return address or current PC
    - **branch:** Identifies if the instruction is of branch type, useful for updating MuxINCSelect for SB instructions after execution step
    - **jump:** Identifies if the signal is of jump type, useful in control state update



2. ALU Control: It is required for the ALU, in the execution stage, to decide what particular operation is to be performed for the current instruction. The ALUcontrol signal is produced as follows:

Instruction opcode	funct3	funct7	ALUControl
add(51)	0	0	0(+)
and(51)	7	0	1(&)
or(51)	6	0	2(|)
sll(51)	1	0	3(<<)
srl(51)	5	0	4(>>)
sra(51)	5	32	5(sra)
slt(51)	2	0	6(slt)
sub(51)	0	32	7(-)
xor(51)	4	0	8(xor)
mul(51)	0	1	9(*)
div(51)	4	1	10(/)
rem(51)	6	1	11(%)
addi(19)	0	0	0(+)
andi(19)	7	0	1(&)
ori(19)	6	0	2(|)
lb(3)	0	0	0(+)
lh(3)	1	0	0(+)
lw(3)	2	0	0(+)
jalr(103)	0	0	Don't Care
sb(35)	0	0	0(+)
sh(35)	1	0	0(+)
sw(35)	2	0	0(+)
auipc(23)	0	0	12(auipc+lui fn)
lui(55)	0	0	12
jal(111)	0	0	Don't Care

beq(99)	0	0	13(eq)
bne(99)	1	0	14(neq)
bge(99)	5	0	15(geq)
blt(99)	4	0	16(lte)

Register file-
Register file is contained in Registers.py
It contains the registers as a python list of size 32 and the instruction register IR.
General purpose registers store values as a signed number.

Inputs-
- RegNumber-Register to access
- RegWrite- register enable
- WriteVal- Value to be written
- IRwrite- IR enable

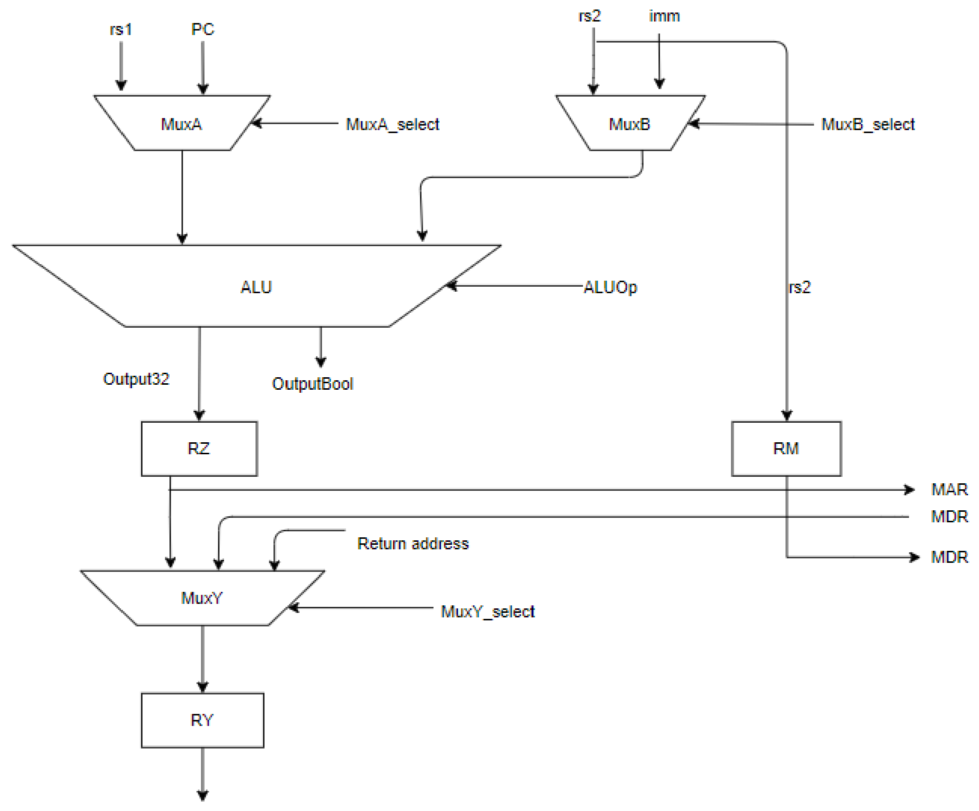
Outputs-
- Methods return the register value

Methods-
- ReadIR-
- WriteIR-
- WriteGpRegisters- Writes value to the register
- ReadGpRegisters- Method returns register value according to the register number

BUFFERS:

Four buffers, namely RA, RY, RZ, RM are used. These act as placeholders for result values and are useful since decisions are made using muxes at various steps, depending on the instruction type.
RA- used in IAG for return address
RZ- holds the output of the ALU
RY- holds the value to be written to the register(if any)
RM- holds the rs2 value, required for Memory Access stage

ALU-



ALU.py contains the code for the arithmetic logic unit.
It contains the class ArithmeticLogicUnit.

Inputs to the ALU-

- ALUOp- ‘on off switch of the ALU’
- ALUcontrol- Dictates the operation to execute
- MuxA
- MuxB

Outputs-

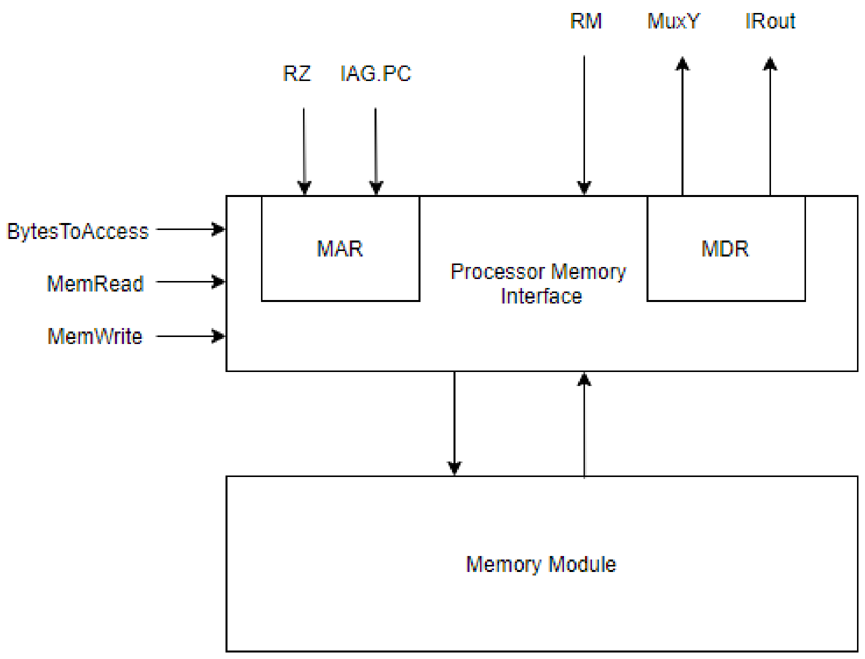
- Output32- 32 bit output
- outputBool- For branch instructions

Methods-

- add()- for add, addi, lb, lw, lh, jalr, sb, sw, sh
- subtract()- for sub
- AUIPC_LUI() - for auipc, lui
- setIfLessThan()- for slt
- multiply()- for mul
- division()- for div
- remainder()- for rem

- rightShiftLogical()- for srl
- rightshiftArithmetic()- for sra
- leftShift- for sll
- bitwiseAND- for and, andi
- bitwiseXOR- for xor
- bitwiseOR- for or, ori
- areEqual- for beq
- areNotEqual- for bne
- GreaterThanEqualTo- for bge
- LessThan()- blt

Memory Module-



The memory module code is contained in the file Memory.py. The implementation consists of 2 classes, “ProccessorMemoryInterface” and “ByteAddressableMemory”.

1. ProccessorMemoryInterface

This modules contains methods to receive control signals and request the ByteAddressableMemory module for data.
Inputs-
- MemRead
- MemWrite
- BytesToAccess

- RZ
- RM
- IAG

The module contains-
- MDR- Memory data register
- MAR- Memory address register

Methods contained-
- InitMemory- Used to load instructions to memory at the start of the program
- LoadInstruction- Fetch instruction from memory according to address in PC
- AccessMemory- Used in store and load instructions; calls the following methods according to control signals MEM_read, MEM_write
- ReadMemory- Obtains data from memory according to base_address and BytesToAccess
- WriteMemory- Used to write data to memory according to base address, BytesToAccess and data from RM

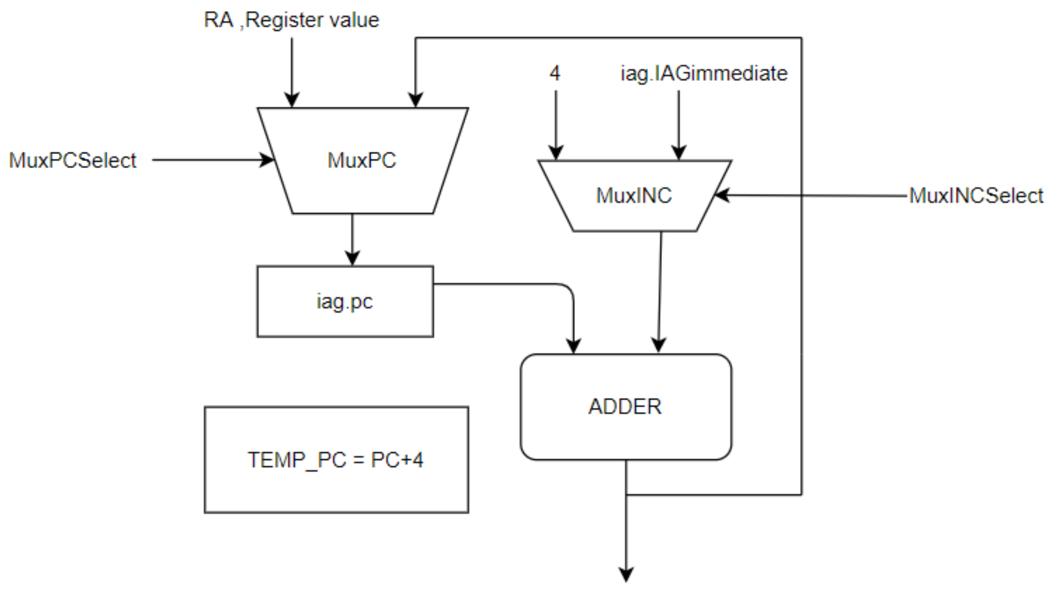
2. ByteAddressableMemory

This module stores data in a hashtable, with the key as the byte address. Every key corresponds to a byte of memory. The data is stored as an unsigned value corresponding to the bit expression of the data. Only those addresses exist in the hashtable which have been written to at least once during program execution, and other non-existent keys return 0 whenever accessed.

Module contains-
- memory- Hash Table for storing data

Methods contained-
- WriteValueAtAddress- Used to write data to memory according to base address and BytesToAccess. Appropriate bit masks and shift operations have been used to extract the relevant bytes and store them with each byte corresponding to a key in the hash table.
- GetUnsignedValueAtAddress- Used to load instructions from memory as an unsigned number without sign extension.
- GetSignedValueAtAddress- Used to load data from memory according to the base address and BytesToAccess. The data loaded after appropriate shift and addition operations is appropriately sign extended as per the the value of BytesToAccess (For eg, if 1 byte is loaded, the 8th bit of the obtained data will be extended).

IAG Module : -



Inputs:-

- MuxINCselect - Control signal of MuxINC
- MuxPCselect - Control signal of MuxPC
- Boffset - To set the IAG value.

Output:-

- PC_temp - To store this value for jal and jalr statement as return address in a register.

Important Variables:-

- PC - SImulates the PC register in IAG.
- PC_temp- Simulates the PC_temp register in IAG
- IAGimmediate- Simulates the immediate input given to MuxINC

Methods:-

- ReadPC - Used to get the current value of PC
- WritePC - Used to update/ write the value of PC
- PCTempUpdate - Used to set the value of PC_temp to PC + 4
- PCset - Used to select , based on control signals whether to put PC as RA (register input in case of Jalr) or let PC remain PC.
- SetBranchOffset - Used to update the immediate value that is connected to MuxINC
- PCUpdate- Based on the control signals that are given as arguments , this function decides whether to set PC as PC + 4 or PC + immediate.