Hilmi Taşkın 2232700
Bilal Yuksu 2110047

EE446 Laboratory Project

Single Cycle RISC-V Processor

1. Datapath

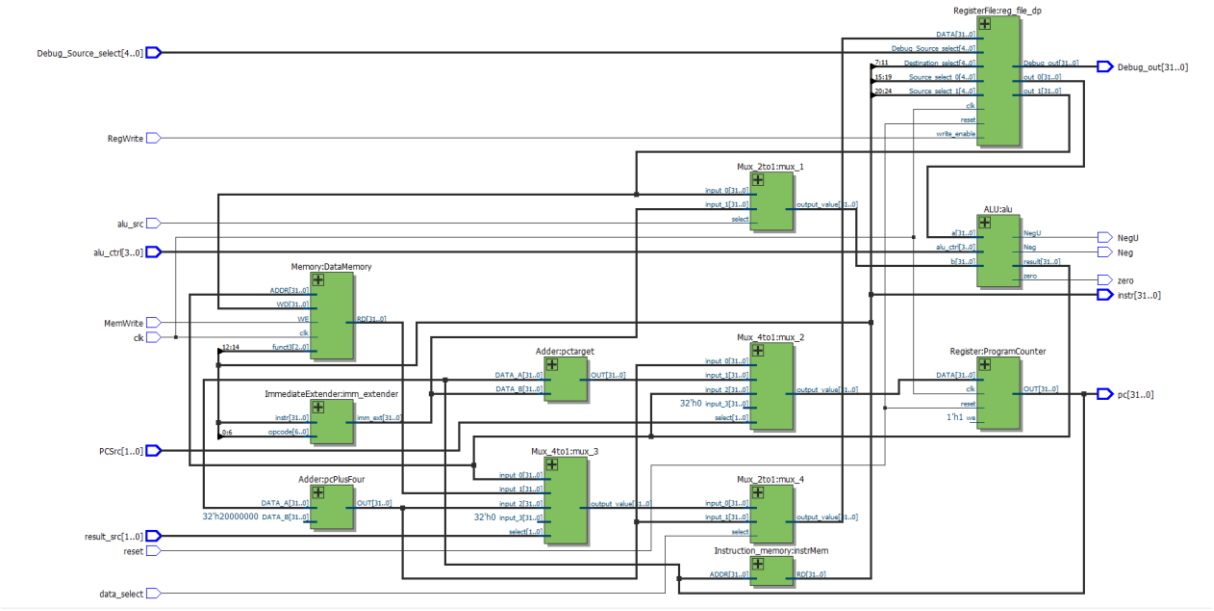


Figure 1. RTL View of the datapath.

We have designed our datapath described on the Harris & Harris book, but the extended version which includes JAL, JALR, LUI and AUIPC instructions. We also included XORID operation but it did not require us to modify the datapath. We also included Neg and NegU flags from the ALU to have the comparison info from the ALU. We added a mux right before WD3 input of the register file to implement the operations that PC+4 to a register in the register file when there is an operation which writes to PC. Also we extended the PCSrc mux to have 3 inputs. The extra input comes from the ALUResult, which is needed for JALR operation. We extended ALUControl signal to 4 bits to cover all of the ALU operations. You can see the extended and modified datapath from the Harris & Harris book in the next pages. We followed Harris & Harris book for datapath and controller design and did some modifications on it wherever necessary.

Hilmi Taşkın 2232700
Bilal Yuksu 2110047

Here is an example data flow and control signals for AND instruction.

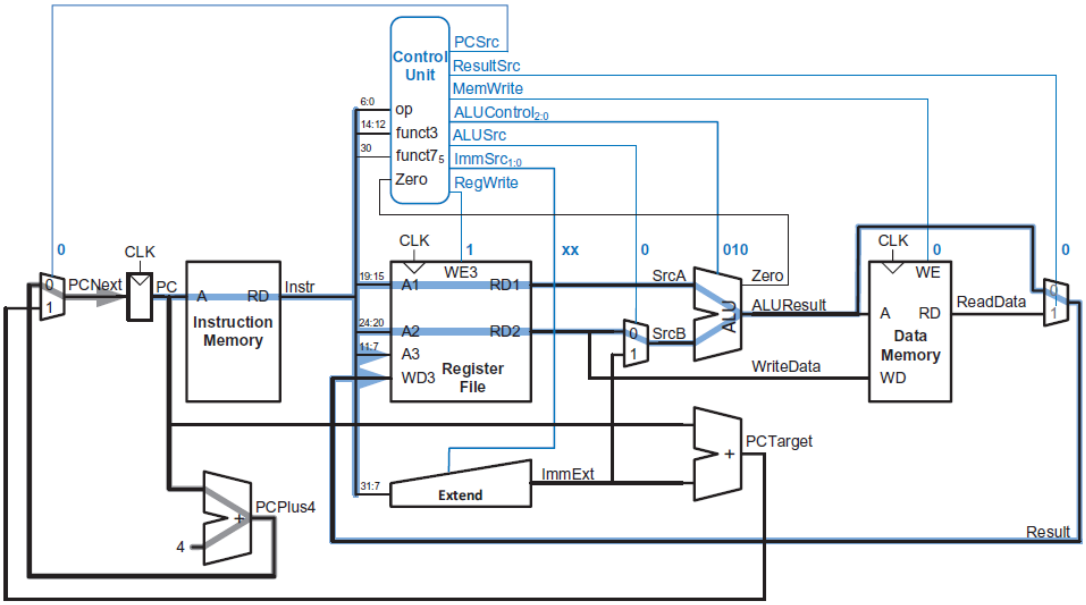


Figure 7.14 Control signals and data flow while executing an `and` instruction

We extended datapath and controller to include JALR, LUI, AUIPC, BLT, BGE, BLTU, BGEU and XORID operations. Our design does not have ImmSrc as a control signal from the controller but it takes opcode from the instruction and decides on the type of immediate extension depending on the opcode. Here is another implementation on the datapath and controller:

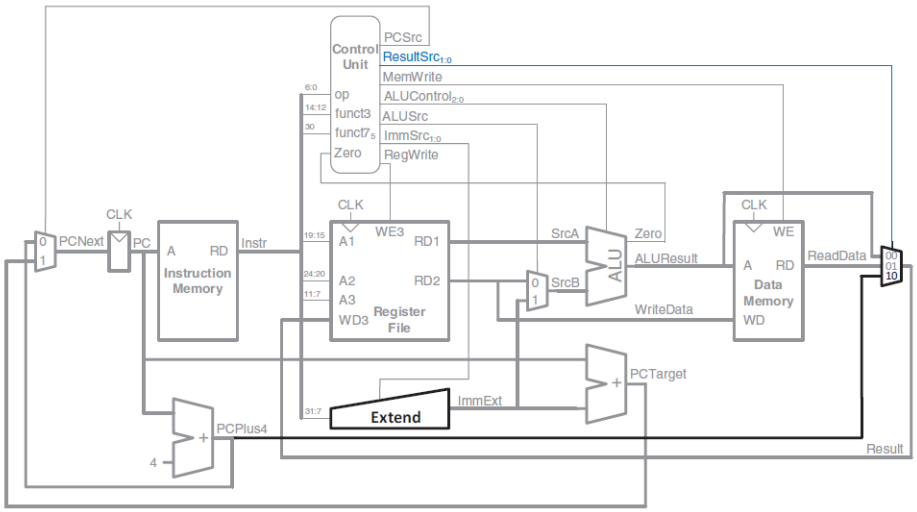


Figure 7.15 Enhanced datapath for `jal`

Table 7.5 *ImmSrc* encoding.

ImmSrc	ImmExt	Type	Description
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-bit signed immediate
11	{{12{*Instr*[31]}}, *Instr*[19:12], *Instr*[20], *Instr*[30:21], 1'b0}	J	21-bit signed immediate

Figure 2. Extended datapath for JAL instruction.

Hilmi Taşkın 2232700
Bilal Yuksu 2110047

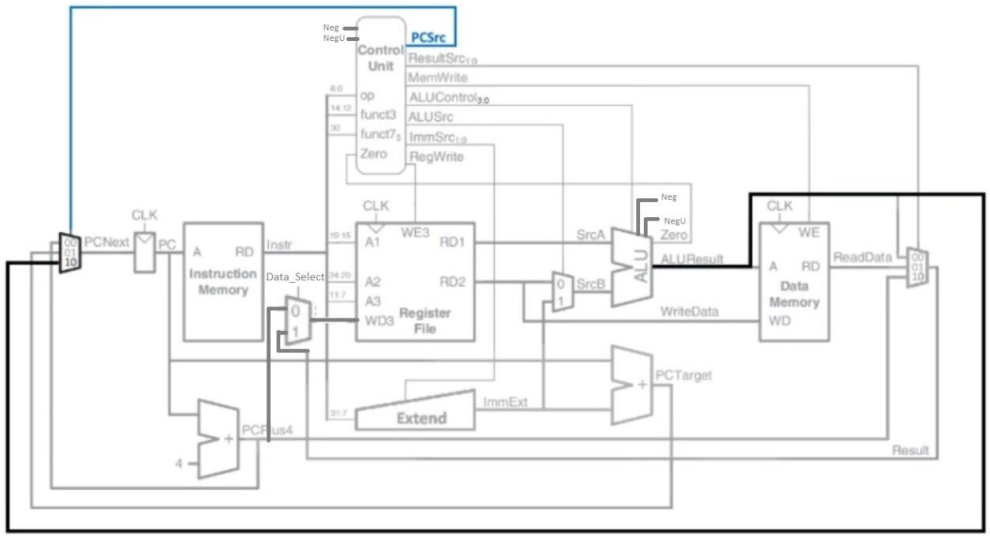


Figure 3. Modified complete datapath from the Harris & Harris book with our additional mux.

2. Controller

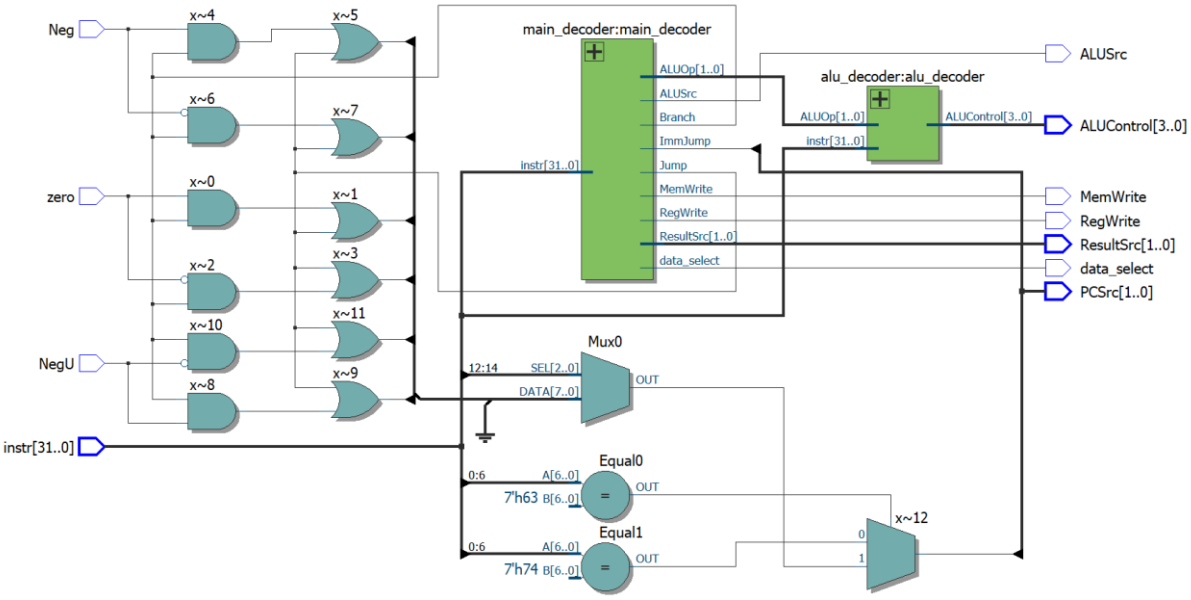


Figure 4. RTL View of the controller.

We designed the controller in two submodules namely, main decoder and alu decoder and some logic to determine the PCSrc signal. Main decoder creates all the other control signals except ALUControl. ALUControl is created at alu decoder. The data_select signal is for the mux before the WD3 input of the register file. It chooses between PC+4 and result. The other signals are as it is in the design in the book. Neg and NegU are needed and used for Branch instructions, as well as zero. ImmJump is the PCSrc[1] signal which is only asserted in JALR instructions. On the next page you can see a controller which is not complete according to our requirements from the Harris & Harris book.

Hilmi Taşkın 2232700
Bilal Yuksu 2110047

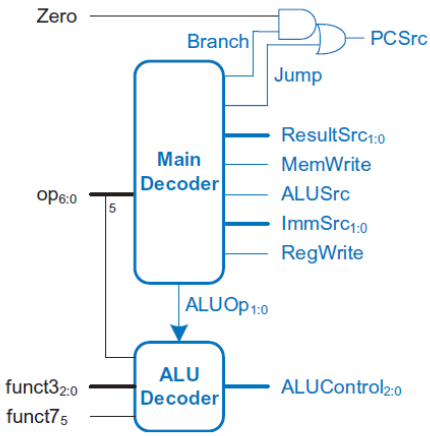


Figure 7.16 Enhanced control unit for `jal`

Table 7.6 Main Decoder truth table enhanced to support `jal`

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw`	0000011	1	00	1	0	01	0	00	0
`sw`	0100011	0	01	1	1	xx	0	00	0
R-type	0110011	1	xx	0	0	00	0	10	0
`beq`	1100011	0	10	0	0	xx	1	01	0
I-type ALU	0010011	1	00	1	0	00	0	10	0
`jal`	1101111	1	11	x	0	10	0	xx	1

Hilmi Taşkın 2232700
Bilal Yuksu 2110047

3. Testbench

Here you can see the types of operations and their instruction encodings from Harris & Harris book.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

For creating the instructions, we used the list below from the Harris & Harris book:

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

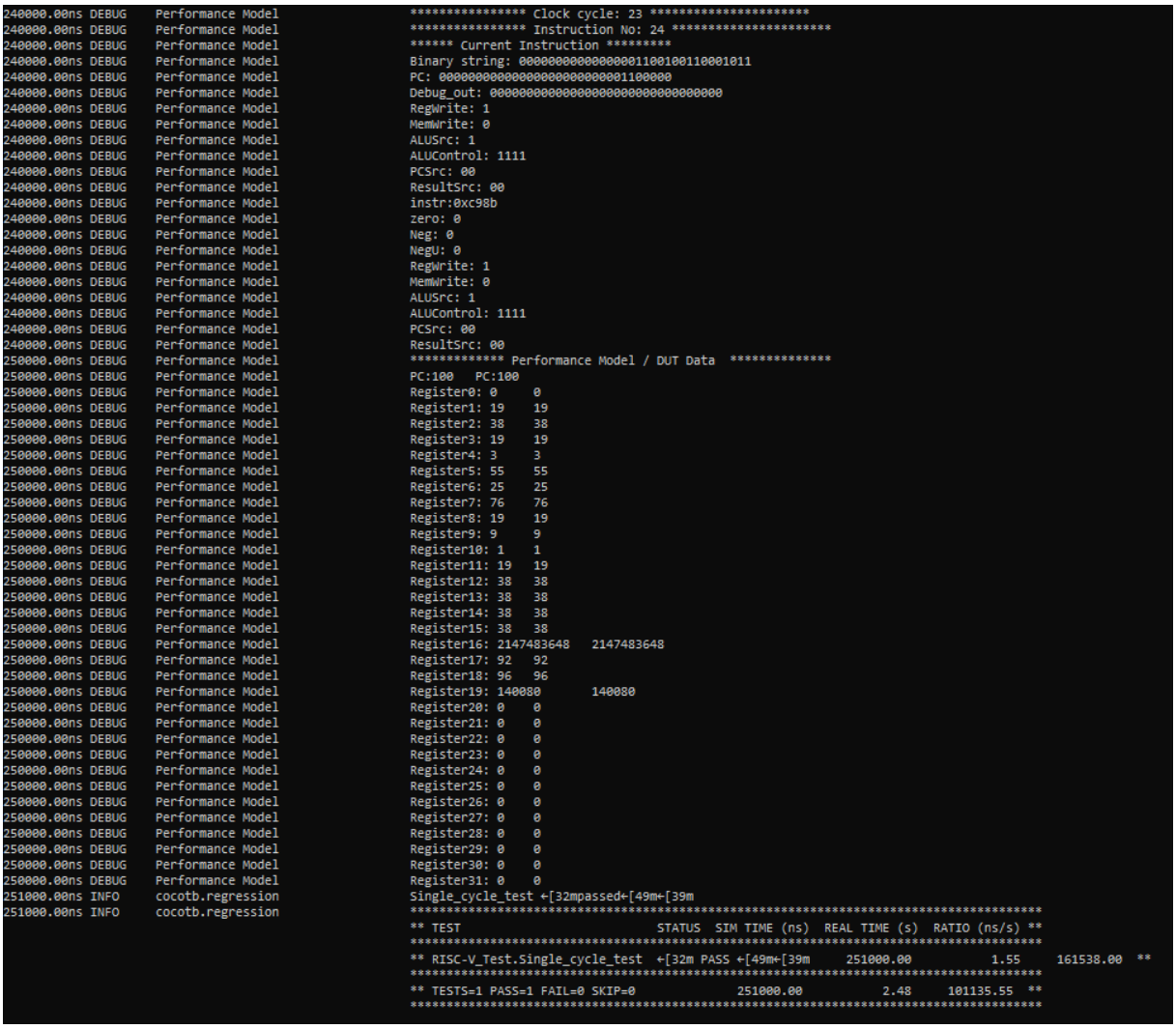
Hilmi Taşkın 2232700
Bilal Yuksu 2110047

Our design can operate all the instructions listed above and XORID instruction that is asked in the project description. The instructions that are used in testbench which cover all the operations in the extended instruction set are below:

```
93 00 30 01     # addi x1, x0, 0x13
33 81 10 00     # add x2, x1, x1
B3 01 11 40     # sub x3, x2, x1
13 F2 F0 00     # andi x4, x1, 0xF
B3 E2 21 00     # or x5, x3, x2
13 C3 A0 00     # xori x6, x1, 0xA
93 93 20 00     # slli x7, x1, 2
13 54 11 00     # srli x8, x2, 1
93 D4 11 40     # srai x9, x3, 1
13 A5 00 02     # slti x10, x1, 0x10
B3 35 31 00     # sltu x11, x2, x3
63 84 11 00     # beq x1, x3, 0x8
E3 88 30 FE     # beq x1, x3, -16
23 20 10 00     # sw x1, x0, 0
83 25 00 00     # lw x11, x0, 0
23 10 20 00     # sh x2, x0, 0
03 16 00 00     # lh x12, x0, 0
23 00 20 00     # sb x2, x0, 0
83 06 00 00     # lb x13, x0, 0
03 57 00 00     # lhu x14, x0, 0
83 47 00 00     # lbu x15, x0, 0
37 08 00 80     # lui x16, 0x80000
97 88 00 00     # auipc x17, 0x8
EF 08 C0 05     # jal x17, 92
67 89 D0 04     # jalr x18, 77(x1)
8B C9 00 00     # xorid x19, x1
```

Hilmi Taşkın 2232700
Bilal Yuksu 2110047

For the testbench we used testbench used for ARM Single Cycle Processor and modified it according to our requirements. You can see the testbench code in the code section. The results of the testbench run are below:



Hilmi Taşkın 2232700
Bilal Yuksu 2110047

Also, you can see how some of the operations are implemented in the performance_model in the testbench:

RISC-V_Test.py 4

C: > Users > Msi > OneDrive > Belgeler > risc-v_test > Test > RISC-V_Test.py > ...

```
8   class TB:
51      def performance_model(self):
77
78          # Embedded constant for XORID
79          xorid_constant = 2232700 ^ 2110047
80
81          if opcode == 0x33:  # R-type
82              if funct3 == 0x0:
83                  if funct7 == 0x00:  # ADD
84                      self.Register_File[rd] = self.Register_File[rs1] + self.Register_File[rs2]
85                  elif funct7 == 0x20:  # SUB
86                      self.Register_File[rd] = self.Register_File[rs1] - self.Register_File[rs2]
87              elif funct3 == 0x1:  # SLL
88                  self.Register_File[rd] = self.Register_File[rs1] << (self.Register_File[rs2] & 0x1F)
89              elif funct3 == 0x2:  # SLT
90                  self.Register_File[rd] = 1 if self.Register_File[rs1] < self.Register_File[rs2] else 0
91              elif funct3 == 0x3:  # SLTU
92                  self.Register_File[rd] = 1 if (self.Register_File[rs1] & 0xFFFFFFFF) < (self.Register_File[rs2] & 0xFFFFFFFF) else 0
93              elif funct3 == 0x4:  # XOR
94                  self.Register_File[rd] = self.Register_File[rs1] ^ self.Register_File[rs2]
95              elif funct3 == 0x5:
96                  if funct7 == 0x00:  # SRL
97                      self.Register_File[rd] = (self.Register_File[rs1] & 0xFFFFFFFF) >> (self.Register_File[rs2] & 0x1F)
98                  elif funct7 == 0x20:  # SRA
99                      self.Register_File[rd] = self.Register_File[rs1] >> (self.Register_File[rs2] & 0x1F)
100             elif funct3 == 0x6:  # OR
101                 self.Register_File[rd] = self.Register_File[rs1] | self.Register_File[rs2]
102             elif funct3 == 0x7:  # AND
103                 self.Register_File[rd] = self.Register_File[rs1] & self.Register_File[rs2]
104
105         elif opcode == 0x13:  # I-type
106             if funct3 == 0x0:  # ADDI
107                 self.Register_File[rd] = self.Register_File[rs1] + imm_i
108             elif funct3 == 0x1:  # SLLI
109                 self.Register_File[rd] = self.Register_File[rs1] << (imm_i & 0x1F)
110             elif funct3 == 0x2:  # SLTI
111                 self.Register_File[rd] = 1 if self.Register_File[rs1] < imm_i else 0
```

We implemented the XORID instruction with a xoring the selected register content with the xorid_constant which is the xor of the student ids of us. You can also see the some different instruction types below:

RISC-V_Test.py 4

C: > Users > Msi > OneDrive > Belgeler > risc-v_test > Test > RISC-V_Test.py > ...

```
8   class TB:
51      def performance_model(self):
151         elif opcode == 0x63:  # B-type
152             if funct3 == 0x0:  # BEQ
153                 if self.Register_File[rs1] == self.Register_File[rs2]:
154                     self.PC += imm_b
155             elif funct3 == 0x1:  # BNE
156                 if self.Register_File[rs1] != self.Register_File[rs2]:
157                     self.PC += imm_b
158             elif funct3 == 0x4:  # BLT
159                 if self.Register_File[rs1] < self.Register_File[rs2]:
160                     self.PC += imm_b
161             elif funct3 == 0x5:  # BGE
162                 if self.Register_File[rs1] >= self.Register_File[rs2]:
163                     self.PC += imm_b
164             elif funct3 == 0x6:  # BLTU
165                 if (self.Register_File[rs1] & 0xFFFFFFFF) < (self.Register_File[rs2] & 0xFFFFFFFF):
166                     self.PC += imm_b
167             elif funct3 == 0x7:  # BGEU
168                 if (self.Register_File[rs1] & 0xFFFFFFFF) >= (self.Register_File[rs2] & 0xFFFFFFFF):
169                     self.PC += imm_b
170
171         elif opcode == 0x6F:  # J-type
172             self.Register_File[rd] = self.PC + 4
173             self.PC = imm_j
174
175         elif opcode == 0x67:  # I-type for JALR
176             self.Register_File[rd] = self.PC + 4
177             self.PC = (self.Register_File[rs1] + imm_i) & ~1
178
179         elif opcode == 0x37:  # U-type LUI
180             self.Register_File[rd] = imm_u
181
182         elif opcode == 0x17:  # U-type AUIPC
183             self.Register_File[rd] = self.PC + 4
184             self.PC = self.PC + imm_u
```