FemtoRV32

Omar Elfouly
Bavly Remon

November 5, 2023

Abstract

In this project we are creating a miniature version of an RV32i Processor. In this milestone we create a single cycle implementation but will be implementing a pipelined implementation in the near future. In this paper we will discuss the abillities and limitations of our processor, aswell as the whole development cycle including designing and testing.

1 Introduction

During this milestone, we were tasked to implement a single cycle RV32 processor that supports up all 40 RISC-V instructions that are included in figure 1. They are all implemented according to the specification document provided. There is an exception however in the ECALL, EBREAK and FENCE instructions such that ECALL and FENCE would be interpreted as a pass or a NOP instruction while EBREAK would halt execution of the problem such that the program counter wouldn't increment anymore. It is also worth mentioning that we will have 2 separate memories for data and instructions due to structural hazards in the single cycle implementation, this would change when we implement the pipeline.

2 Development

2.1 Design and Implementation

We started off the design process by using the data path that we were given during classes and later implemented in our labs. This Data path only supported 7 instructions out of the 40 required (Beq, LW, SW, ADD, SUB, AND, OR) but it contained some of the main components that we needed in the full data path, these included the PC, Instruction memory, Register file, ALU, Data Memory, and some basic implementation to support branching. So we already had these major components available and connected we only needed to modify some of them like the ALU and the Data Memory to allow us to execute all the other 40 instructions.

We were provided with some helping code that we integrated into our initial design. We were given a ready made immediate Generator that we fully integrated into our data path and made minor changes to our data path like removing the 1 bit shifter for example as the new IMMGen already did that. We then made use of the ALU module provided, however, we made some minor changes to it. The ALU made use of a shifter module to support some of the shifting instructions, however, this module was not implemented so instead of implementing it, we decided to do the shifting operations inside the ALU itself instead of using the shifter module. This was a design choice we made as it improves readability and removes any risks of errors while integrating the new hardware. As an example, this choice allowed us to quickly implement a fix to an error with one of the shifting instructions and made it easier for us to spot the error. We were also provided with a very handy document that included useful macros that we implemented throughout our whole code which helped us in testing, debugging and also implementing new hardware.

Moving on to the new hardware we that we added on top of the old implementation in figure 3. The main additions to the data path are a 4X1 MUX at the input to the write data of the register file, 2 2X1 MUXes that control the input to the PC and a Branch control unit along with simple logic that detects JAL and JALR instructions that control the PC input MUXes. Starting off with the register file write data MUX, it chooses between PC+4, PC+IMM, IMM, or the output of the original memtoreg mux, this mux is controlled by a signal produced by the control unit. Next we have Branch control unit which takes in the control unit branch signal aswell as all the flags from the alu and func 3 to decide which branch is being excuted and whether to update PC, it produced a signal that is the ORed with simple logic to check if its a JAL or JALR, and then we check to see if its a JALR. All these sinals are then used to choose between PC + imm , PC + 4 and ALU out. So that concludes all the additional hardware we added to the data path, the other changes would be in the components them selves mainly the control unit to handle all the new signals and the ALU control unit to match the new ALU. Also to implement the EBREAK we use one of it bits to control the PC load siganal so when we excute the instruction it turns of the PC load signal.

2.2 Challenges and Solutions

The main issues we faced were in creating the control signals that controlled everything including the ALU control signals but after deep analysis of the control signals we already had and how they worked we were able to modify it accordingly and produce our own signals.

3 Testing

To test our processor, we decided to take a more rigorous approach and test each instruction individually and include multiple test cases that made use of all the different ways of using an instruction like using positive / negative number, using 0 and so on. We have included our Instruction memory that contains all the test cases that we used along with comments of the expected outputs. initial begin // $readmemh("../Test/Hex/test1_mem.hex", mem); //$
$32'b00000000000001100100000010110111; //luix1, 100 mem[7], mem[6], mem[5], mem[4] =$
$32'b11111111111110011100000100110111; //luix2, -100 mem[11], mem[10], mem[9], mem[8] =$
$32'b00000000000000000000000110110111; //luix3, 0$

//AUIPC mem[15],mem[14],mem[13],mem[12]= 32’b00000000000000000000001010010111;//auipc x9, 1 mem[19],mem[18],mem[17],mem[16] = 32’b00000000000000000000001000010111; //auipc x4, 0 mem[23],mem[22],mem[21],mem[20] = 32’b11111111111111001110000100010111; //auipc x4, -100

//JAL mem[27],mem[26],mem[25],mem[24] = 32’b00000000100000000000001011101111; // jal x5, 8 mem[31],mem[30],mem[29],mem[28] = 32’b00000000100000000000001011101111; //jal x5, 8 mem[35],mem[34],mem[33],mem[32] = 32’b11111111110111111111001011101111; //jal x5, -4

//JALR mem[39],mem[38],mem[37],mem[36] = 32’b00000000110000101000001101100111;//jalr x6, 12(x5) mem[47],mem[46],mem[45],mem[44] = 32’b00000000000000110000001111100111;//jalr x7, 0(x6) mem[43],mem[42],mem[41],mem[40] = 32’b00000000000000111000010001100111;//jalr x8, 0(x7)

//BEQ mem[51],mem[50],mem[49],mem[48] = 32’b00000000011100110000011001100011;//beq x6, x7, 12 mem[55],mem[54],mem[53],mem[52] = 32’b00000000000000000000010001100011;//beq x0, x0, 8

//BNE mem[63],mem[62],mem[61],mem[60] = 32’b00000000000000000001110001100011;//bne x0, x0, 24 mem[67],mem[66],mem[65],mem[64] = 32’b00000000011100110001010001100011;//bne x6, x7, 8

//BLT mem[75],mem[74],mem[73],mem[72] = 32’b00000000000000000100110001100011;//blt x0, x0, 24 mem[79],mem[78],mem[77],mem[76] = 32’b00000000011101000100010001100011;//blt x8, x7, 8

//BGE mem[87],mem[86],mem[85],mem[84] = 32’b00000110011101000101001001100011;//bge x8, x7, 100 mem[91],mem[90],mem[89],mem[88] = 32’b00000000000000000101011001100011;//bge x0, x0, 12

//BLTU mem[103],mem[102],mem[101],mem[100] = 32’b00000110010100100110001001100011;//bltu x4, x5, 100 mem[107],mem[106],mem[105],mem[104] = 32’b00000000010000101110010001100011;//bltu x5, x4, 8

//BGEU mem[115],mem[114],mem[113],mem[112] = 32’b11111110010000101111110011100011;//bgeu x5, x4, -8 mem[119],mem[118],mem[117],mem[116] = 32’b00000000010100100111010001100011;//bgeu x4, x5, 8 mem[127],mem[126],mem[125],mem[124] = 32’b11111110010100100111111011100011;//bgeu x4,x5,-4 mem[123],mem[122],mem[121],mem[120] = 32’b00000000010100100111010001100011;//bgeu x4,x5,8

//Memory (store and load) mem[131],mem[130],mem[129],mem[128] = 32’b1111111001010010100000000
x5, -31(x5) 1 mem[135],mem[134],mem[133],mem[132] = 32’b00000000100100101001000000100011;//sh x9, 0(x5) 32 mem[139],mem[138],mem[137],mem[136] = 32’b00000000000100101010001000100011;//sw x1, 4(x5) 36

mem[143],mem[142],mem[141],mem[140] = 32’b11111110000100101000010100000011;//lb x10, -31(x5) mem[147],mem[146],mem[145],mem[144] = 32’b00000000000000101001010110000011;//lh x11, 0(x5) mem[151],mem[150],mem[149],mem[148] = 32’b00000000010000101010011000000011;//lw x12, 4(x5)

//I-type

//ADDI mem[155],mem[154],mem[153],mem[152] = 32’b00000100010100000000111100010011; //addi x30, x0, 69 mem[159],mem[158],mem[157],mem[156] = 32’b11111111111100000000111100010011; //addi x30, x0, -1 mem[163],mem[162],mem[161],mem[160] = 32’b00000000000000101000111100010011; //addi x30, x5, 0 32 +0

//SLTI mem[167],mem[166],mem[165],mem[164] = 32'b00000010000000101010111010010011;
//slti x29, x5, 32 false mem[171],mem[170],mem[169],mem[168] = 32'b0000010001010010101011101001001
//slti x29, x5, 69 true mem[175],mem[174],mem[173],mem[172] = 32'b0000001100000011101011101001001
//slti x29, x7, 48 false mem[179],mem[178],mem[177],mem[176] = 32'b1111111111110010001011101001001
//slti x29, x4, -1 true

//SLTIU mem[179],mem[178],mem[177],mem[176] = 32'b11111111111100101011111000010011;//sltiu
x28, x5, -1 true mem[183],mem[182],mem[181],mem[180] = 32'b00000010000000101011111000010011;//s
x28, x5, 32 false mem[187],mem[186],mem[185],mem[184] = 32'b00000110010000101011111000010011;//
x28, x5, 100 true

//XORI mem[191],mem[190],mem[189],mem[188] = 32'b00111110011100001100111110010011;//xori
x31, x1, 999 409600 ^{9}99 = 410599$mem[195], mem[194], mem[193], mem[192] = 32'b1101110101010000110$
4294557141

//ORI mem[199],mem[198],mem[197],mem[196] = 32'b00111110100001001110110110010011;//ori
x27, x9, 1000 4108 — 1000 = 5100 mem[203],mem[202],mem[201],mem[200] = 32'b111111111111101001110
x27, x9, -1 4108 — -1 = 4294967295 = -1

//ANDI mem[207],mem[206],mem[205],mem[204] = 32'b11111111111101000111110100010011;//andi
x26, x8, -1 44 -1 = 44 mem[211],mem[210],mem[209],mem[208] = 32'b0000000000000100011111010001001
x26, x8, 0 44 0 = 0

//SLLI mem[215],mem[214],mem[213],mem[212] = 32'b00000001111100101001110010010011;//slli
x25,x5, 31 32 ¡¡ 31 = 0 mem[219],mem[218],mem[217],mem[216] = 32'b0000000000000010100111001001001
x25,x5, 0 32 ¡¡ 0 = 32 mem[223],mem[222],mem[221],mem[220] = 32'b0000000000100010100111001001001
x25,x5, 2 32 ¡¡ 2 = 128

//SRLI mem[227],mem[226],mem[225],mem[224] = 32'b00000001111100101101110010010011;//srli
x25,x5, 31 32 ¿¿ 31 = 0 mem[231],mem[230],mem[229],mem[228] = 32'b000000000000001011011100100100
x25,x5, 0 32 ¿¿ 0 = 32 mem[235],mem[234],mem[233],mem[232] = 32'b0000000000100010110111001001001
x25,x5, 2 32 ¿¿ 2 = 8

//SRAI mem[239],mem[238],mem[237],mem[236] = 32'b01000001111100010101110010010011;//srai
x25,x2, 31 -409600 ¿¿¿ 31 = -1 mem[243],mem[242],mem[241],mem[240] = 32'b0100000000000000101011
x25,x2, 0 -409600 ¿¿¿ 0 = -409600 mem[247],mem[246],mem[245],mem[244] = 32'b01000000001000010101
x25,x2, 2 -409600 ¿¿¿ 2 = -102400

//ADD mem[251],mem[250],mem[249],mem[248] = 32'b00000000000000000000110000110011;//add
x24, x0, x0 x24=0 mem[255],mem[254],mem[253],mem[252] = 32'b00000000101001001000110000110011;
x24, x9, x10 x24=4108+32=4140 mem[259],mem[258],mem[257],mem[256] = 32'b000000000000100010000
x24, x2, x1 x24=409600 + (-409600)=0

//SUB mem[263],mem[262],mem[261],mem[260] = 32'b01000000000000000000101110110011;//sub
x23, x0, x0 x23=0 mem[267],mem[266],mem[265],mem[264] = 32'b01000000101001001000101110110011;
x23, x9, x10 x23=4108-32=4076 mem[271],mem[270],mem[269],mem[268] = 32'b0100000000010001000001
x23, x2, x1 x23=-409600 - 409600=-819200

//SLL mem[275],mem[274],mem[273],mem[272] = 32'b00000000000000000001101110110011;//
sll x23, x0,x0 x23 = 0 ¡¡ 0 = 0 mem[279],mem[278],mem[277],mem[276] = 32'b000000011100000001001101
sll x23, x1, x28 x23 = 409600 ¡¡ 1 = 819200 mem[283],mem[282],mem[281],mem[280] =
32'b00000001110000010001101110110011;// sll x23, x2, x28 x23 = -409600 ¡¡ 1 = -819200

//SLT mem[287],mem[286],mem[285],mem[284] = 32'b00000000000000000010101100110011;//
slt x22, x0, x0 x22 = 0 ¡ 0 ? = 0 mem[291],mem[290],mem[289],mem[288] = 32'b0000000000000000100101
slt x22, x2, x0 x22 = -409600 ¡ 0 ? = 1 mem[295],mem[294],mem[293],mem[292] =

32'b00000000100001010010101100110011;// slt x22, x10, x8 x22 = 32 ¡44? =1

//SLTU mem[299],mem[298],mem[297],mem[296] = 32'b00000000000000000011101010110011;// sltu x21, x0, x0 x21 = 0 ¡ 0 ? = 0 mem[303],mem[302],mem[301],mem[300] = 32'b000000000000000001001111
sltu x21, x2, x0 x22 = UNSIGNED -409600 ¡ 0 ? = 0 mem[307],mem[306],mem[305],mem[304] = 32'b00000000100001010011101010110011;// sltu x21, x10, x8 x22 = 32 ¡44? = 1

//XOR mem[311],mem[310],mem[309],mem[308] = 32'b00000000000000000100101000110011;// xor x20, x0, x0 x20 = 0 $^0 = 0 mem[315], mem[314], mem[313], mem[312] = 32'b0000000111000000110010$1
$409600^1 = 409601 mem[319], mem[318], mem[317], mem[316] = 32'b000000011100010101001010001100 1$1
$32^1 = 33$

//SRL mem[323],mem[322],mem[321],mem[320] = 32'b00000000000000000101100110110011;// srl x19, x0,x0 x19 = 0 ¿¿ 0 = 0 mem[327],mem[326],mem[325],mem[324] = 32'b00000001110000001101100
srl x19, x1, x28 x19 = 409600 ¿¿ 1 = 204800 mem[331],mem[330],mem[329],mem[328] = 32'b00000001110000010101100110110011;// srl x19, x2, x28 x19 = -409600 ¿¿ 1

//SRA mem[335],mem[334],mem[333],mem[332] = 32'b01000000000000000101100100110011;// sra x18, x0, x0 x18 = 0 ¿¿¿ 0 = 0 mem[339],mem[338],mem[337],mem[336] = 32'b0100000111000000011011
sra x18, x1, x28 x18 = 409600 ¿¿¿ 1 = 204800 mem[343],mem[342],mem[341],mem[340] = 32'b01000001110000010101100100110011;// sra x18, x2, x28 x18 = -409600 ¿¿¿1 = -204800

//OR mem[347],mem[346],mem[345],mem[344] = 32'b00000000000000000110100010110011;// or x17, x0, x0 x17 = 0 — 0 = 0 mem[351],mem[350],mem[349],mem[348] = 32'b00000000000100000110100
or x17, x0, x1 x17 = 0 — 409600 = 409600 mem[355],mem[354],mem[353],mem[352] = 32'b00000001110000000110100010110011;// or x17, x1, x28 x18 = 409600 — 1 = 409601

//AND mem[359],mem[358],mem[357],mem[356] = 32'b00000000000000000111100000110011;// and x16, x0, x0 x16 = 0 0 = 0 mem[363],mem[362],mem[361],mem[360] = 32'b00000000000100000111100
and x16, x0, x1 x17 = 0 409600 = 0 mem[367],mem[366],mem[365],mem[364] = 32'b000000000110010001
and x16, x8, x6 x16 = 44 40 = 40

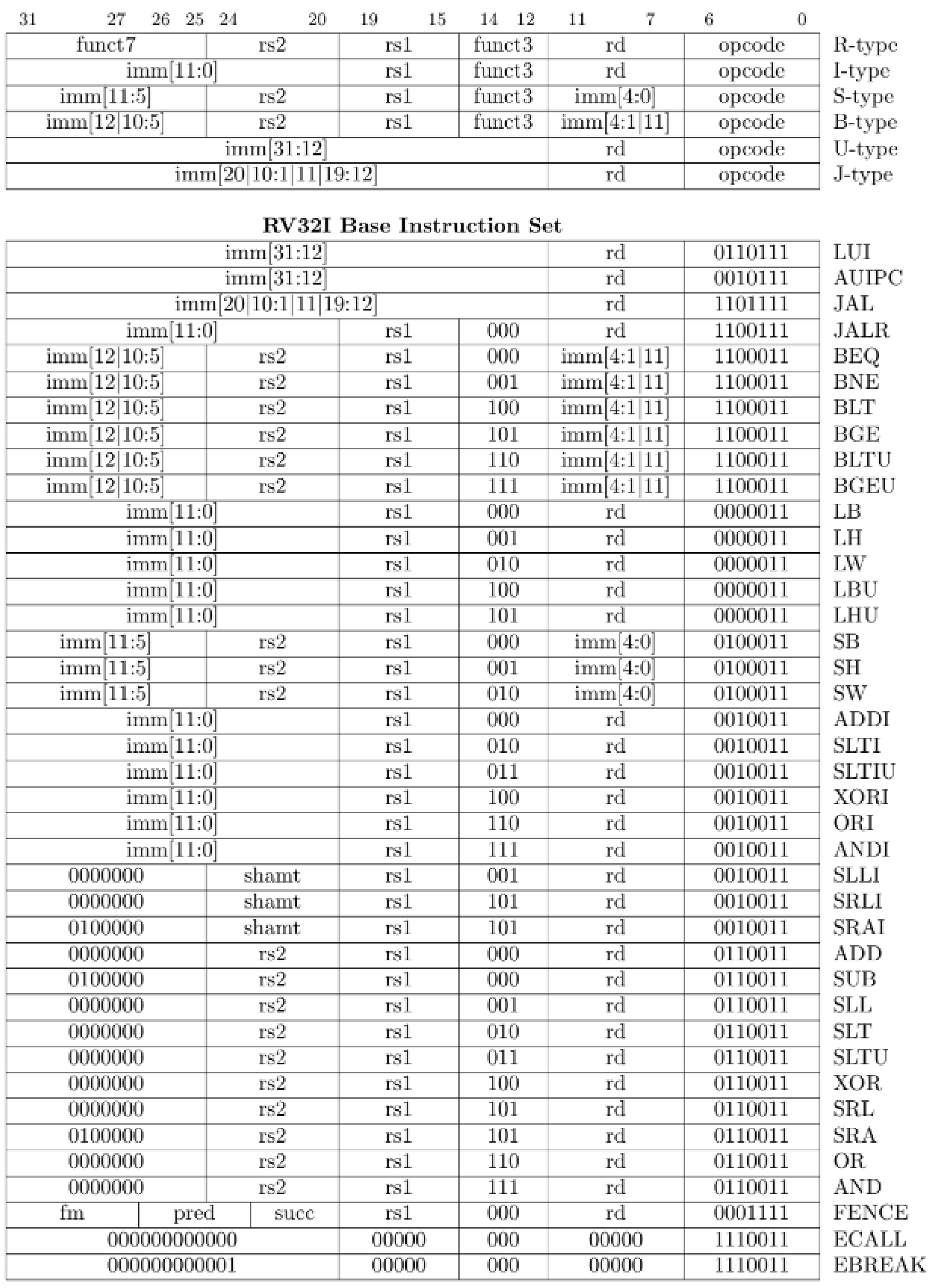
//FENCE mem[371],mem[370],mem[369],mem[368] = 32'b00001111111100000000000000001111;// FENCE = NOP

//ECALL mem[375],mem[374],mem[373],mem[372] = 32'b00000000000000000000000001110011;// ECALL = NOP

//EBREAK mem[379],mem[378],mem[377],mem[376] = 32'b00000000000100000000000001110011;// EBREAK = stop pc

end

This is inluded in the submission for better viewing.



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

RV32I Base Instruction Set

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
fm pred succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Figure 1: Instructions to be supported

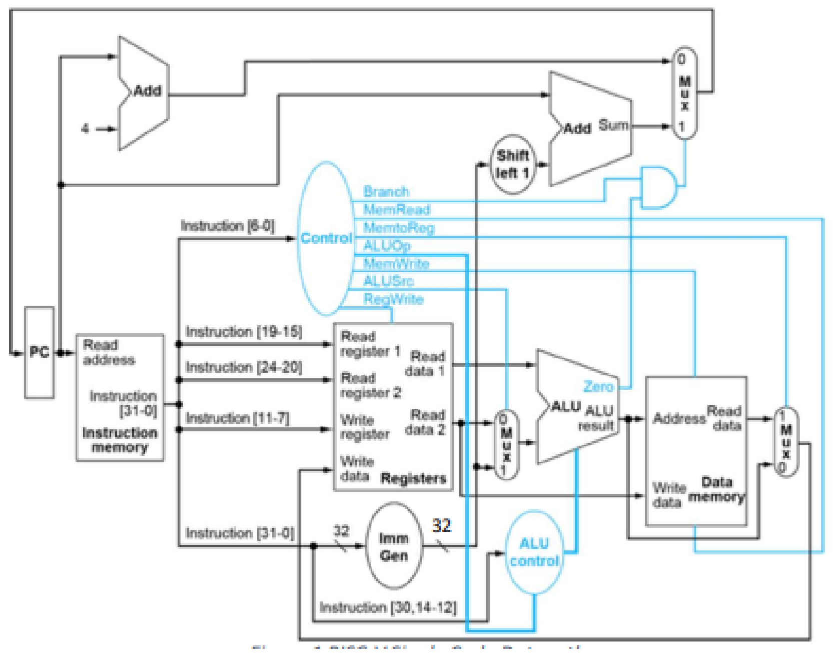


Figure 2: Old datapath supporting 7 instructions

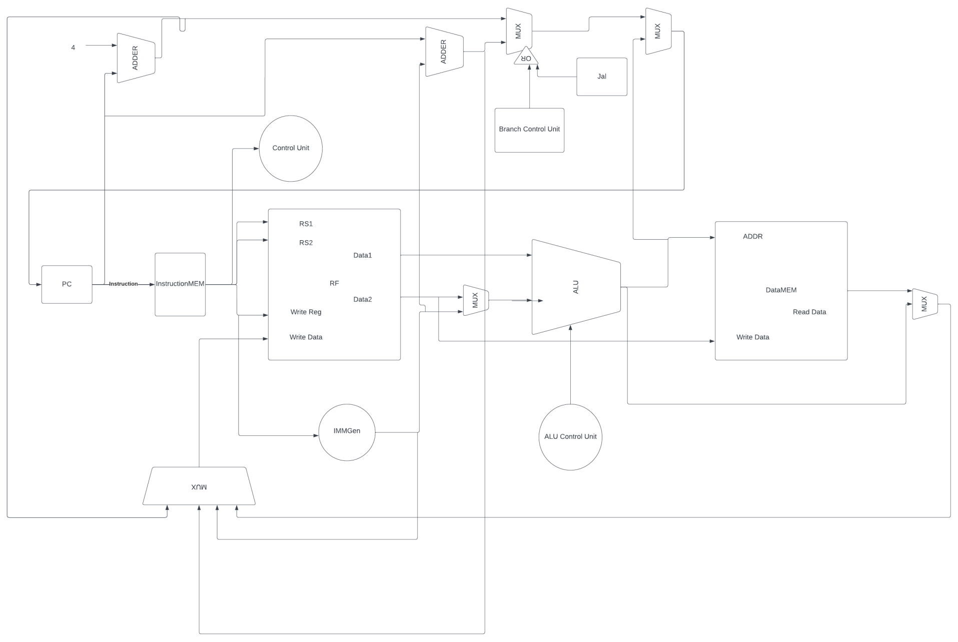


Figure 3: New datapath supporting all instruction