Designing for picoProcessor

The picoProcessor (pP) is an 8-bit processor intended for educational purposes. It is similar to 8-bit microprocessors
for small embedded applications, but has an instruction set architecture more similar to RISC processors. The pP has separate instruction and data memories. The instruction memory is 4K instructions in size, and the data memory is 256 bytes. The pP can also address I/O devices using up to 256 input ports and 256 output ports. Within the processor there are eight 8-bit general purposes registers, r0 to r7. R0 is always read as zero and ignores writes. There is also a return-address stack of implementation-defined depth (at least four entries), an interrupt return register and Zero (Z) and Carry (C) condition codes. In this project we will design this processor and the design include the following steps:

- RTL description
- Datapath components
- List of all control signals
- The Datapath with all necessary multiplexors and all control lines identified
- The design of control unit

The RTL description for instructions types:

In our processor we have seven type of instruction :

Instructions in the pP are all 19 bits long, and are encoded in several formats.

1. ALU Reg-Reg Instructions:

2	3	3	3	3	5
0 0	Fn	Rd	R1	R2	X X X X X X

This format is used for arithmetic and logical instructions that operate on two register values (*r1* and *r2*). The
result is stored in a destination register (*rd*). The function code (*fn*) values are:

Instruction	Operation	fn
ADD	Add without carry in	000
ADDC	Add with carry in	001
SUB	Subtract without carry in	010
SUBC	Subtract with carry in	011
AND	Bitwise logical and	100
OR	Bitwise logical inclusive or	101
XOR	Bitwise logical exclusive or	110
MSK	Bitwise logical mask (and-not)	111

The Z condition code is set to 1 if the result is zero, otherwise it is set to 0. For ADD and ADDC, the C condition code is the carry out of the addition. For SUB and SUBC, C is the borrow out of the subtraction and indicates that the operation overflowed to a negative result. For logical operations, C is always set to 0.

Instruction	RTL description
ADD	Reg(Rd) ← Reg(R1) + Reg(R2); PC ← PC + 3
ADDC	Reg(Rd) ← Reg(R1) + Reg(R2)+C; PC ← PC + 3
SUB	Reg(Rd) ← Reg(R1) - Reg(R2); PC ← PC + 3
SUBC	Reg(Rd) ← Reg(R1) - Reg(R2)+C; PC ← PC + 3
AND	Reg(Rd) ← Reg(R1) AND Reg(R2); PC ← PC + 3
OR	Reg(Rd) ← Reg(R1) OR Reg(R2); PC ← PC + 3
XOR	Reg(Rd) ← Reg(R1) XOR Reg(R2); PC ← PC + 3
MSK	Reg(Rd) ← Reg(R1) NAND Reg(R2); PC ← PC + 3

2. ALU Reg-Immed Instructions:

2	3	3	3	8
0 1	Fn	Rd	R1	Immediate value.

This format is used for arithmetic and logical instructions that operate on a register value (*r1*) and an immediate
constant value (*const*). The result is stored in a destination register (*rd*). The function code (*fn*) values and the condition code settings are the same as for ALU Reg-Reg instructions.

Instruction	RTL description
ADDI	Reg(Rd) ← Reg(R1) + Immediate 8; PC ← PC + 3
ADDCI	Reg(Rd) ← Reg(R1) + Immediate 8 + C; PC ← PC + 3
SUBI	Reg(Rd) ← Reg(R1) - Immediate 8; PC ← PC + 3
SUBCI	Reg(Rd) ← Reg(R1) - Immediate 8+ C; PC ← PC + 3
ANDI	Reg(Rd) ← Reg(R1) AND Immediate 8; PC ← PC + 3
ORI	Reg(Rd) ← Reg(R1) OR Immediate 8; PC ← PC + 3
XORI	Reg(Rd) ← Reg(R1) XOR Immediate 8; PC ← PC + 3
MSKI	Reg(Rd) ← Reg(R1) NAND Immediate 8; PC ← PC + 3

3. Shift Instructions :

3	2	3	3	3	5
110	Fn	Rd	R1	Sc	X X X X X X

This format is used for shift and rotate instructions that operate on a register value (*r1*). The number of bit positions by which the value is shifted or rotated is given by the shift count (*sc*). The result is stored in a destination register (*rd*). The shift function code (*fn*) values are:

Instruction	Operation	fn
SHL	Shift left	00
SHR	Shift right	01
ROL	Rotate left	10
ROR	Rotate right	11

Instruction	RTL description
SHL	Reg(Rd) ← Reg(R1) shift left by sc; PC ← PC + 3
SHR	Reg(Rd) ← Reg(R1) shift right by sc; PC ← PC + 3
ROL	Reg(Rd) ← Reg(R1) rotate left by sc; PC ← PC + 3
ROR	Reg(Rd) ← Reg(R1) rotate right by sc; PC ← PC + 3

The Z condition code is set to 1 if the result is zero, otherwise it is set to 0. The C condition code is set to the
value of the bit shifted or rotated out of the operand value.

4. Memory and I/O Instructions:

3	2	3	3	8
100	Fn	Rd	R1	Disp.

This format is used for memory and I/O load and store instructions. The effective address is calculated by adding a signed displacement (*disp*) to the value of a base address register (*r1*). For memory instructions, the effective address is used to access the data memory, and for I/O instructions, it is used as the I/O port number. For oad instructions, *rd* is the destination register, and for store instruction, *rd* is the source register. The function code (*fn*) values are:

Instruction	Operation	fn
LDM	Load from memory	00
STM	Store to memory	01
INP	Input from port	10
OUT	Output to port	11

Instruction	RTL description
LDM	Reg(Rd) ← MEM[Reg(R1) +(disp.8)]; PC ← PC + 3
STM	MEM[Reg(R1) +(disp.8)] ← Reg(Rd); PC ← PC + 3
INP	Reg(Rd) ← PORT IN[Reg(R1) +(disp.8)]; PC ← PC + 3
OUT	PORT OUT [Reg(R1) +(disp.8)] ← Reg(Rd); PC ← PC + 3

The Z and C condition codes are unaffected by these instructions.

5. Conditional Branch Instructions:

3	2	6	8
101	Fn	X X X X X X X X X X X X X X	Disp.

This format is used for conditional branch instructions. The target address is calculated by adding a signed displacement (*disp*) to the address of the instruction following the branch. If the condition specified by the branch is true, control is transferred to the instruction at the target address; otherwise control continues with the instruction following the branch. The function code (*fn*) values are:

Instruction	Operation	fn
BZ	Branch if zero	00
BNZ	Branch if not zero	01
BC	Branch if carry	10
BNC	Branch if not carry	11

Instruction	RTL description
BZ	if (Z == 0) PC ← PC + 3+ (disp.8) else PC ← PC + 3
BNZ	if (Z != 0) PC ← PC + 3+ (disp.8) else PC ← PC + 3
BC	if (C != 0) PC ← PC + 3+ (disp.8) else PC ← PC + 3
BNC	if (C == 0) PC ← PC + 3+ (disp.8) else PC ← PC + 3

6. Jump Instructions:

	5	2	12
JMP	11100	X X	Address

	5	2	12
JSB	11101	X X	Address

This format is used for unconditional jump instructions. Control is transferred to the instruction at the target
address (*addr*). The JSB instruction, however, first pushes the address of the instruction following the JSB onto the return address stack. For both instructions, the condition codes are unaffected.

Instruction	RTL description
JMP	PC ← Address
JSB	Stack pointer ← Stack pointer+1; Stack [Stack pointer] ← PC + 3 ; PC ← Address

7. Miscellaneous Instructions:

	6	13
RET	111100	X X X X X X X X X X X X X X X X X X X X X X X X X X

This format is used for miscellaneous instructions that have no operands. RET returns from a subroutine by popping the top of the return stack to the program counter.

Instruction	RTL description
RET	PC ← Stack [Stack pointer] ; Stack pointer ← Stack pointer-1;

Instructions are Executed in Steps:

R-type	Fetch instruction Fetch operands Execute operation Write ALU result Next PC address	Instruction ← MEM[PC] data1 ← Reg(R1), data2 ← Reg(R2) result ← func(data1, data2) Reg(Rd) ← ALU result PC ← PC + 3
I-type	Fetch instruction Fetch operands Execute operation Write ALU result Next PC address	Instruction ← MEM[PC] data1 ← Reg(R1), data2 ← immediate.8 result ← func(data1, data2) Reg(Rd) ← ALU result PC ← PC + 3
Shift-type	Fetch instruction Fetch operands Execute operation Write ALU result Next PC address	Instruction ← MEM[PC] data1 ← Reg(R1), data2 ← sc.3 result ← func(data1, data2) Reg(Rd) ← ALU result PC ← PC + 3

LDM	Fetch instruction Fetch base register Calculate address Read memory: Write register Rt Next PC address	Instruction ← MEM[PC] base ← Reg(R1) address ← base + disp.8 data ← MEM[address] Reg(Rd) ← data PC ← PC + 3
INP	Fetch instruction Fetch base register Calculate address Read memory: Write register Rt Next PC address	Instruction ← MEM[PC] base ← Reg(R1) address ← base + disp.8 data ← PORT IN[address] Reg(Rd) ← data PC ← PC + 3
STM	Fetch instruction Fetch registers Calculate address Write memory Next PC address	Instruction ← MEM[PC] base ← Reg(R1), data ← Reg(Rd) address ← base + disp.8 MEM[address] ← data PC ← PC + 3
OUT	Fetch instruction Fetch registers Calculate address Write memory Next PC address	Instruction ← MEM[PC] base ← Reg(R1), data ← Reg(Rd) address ← base + disp.8 PORT OUT[address] ← data PC ← PC + 3
BZ	Fetch instruction Branch	Instruction ← MEM[PC] if (Z == 0) PC ← PC + 3+ (disp.8) else PC ← PC + 3
BNZ	Fetch instruction Branch	Instruction ← MEM[PC] if (Z != 0) PC ← PC + 3+ (disp.8) else PC ← PC + 3

BC	Fetch instruction Branch	Instruction ← MEM[PC] if (C != 0) PC ← PC + 3+ (disp.8) else PC ← PC + 3
BNC	Fetch instruction Branch	Instruction ← MEM[PC] if (C == 0) PC ← PC + 3+ (disp.8) else PC ← PC + 3
JMP	Fetch instruction Jump	Instruction ← MEM[PC] PC ← Address
JSB	Fetch instruction Increment s.pointer Push to stack Jump	Instruction ← MEM[PC] Stack pointer ← Stack pointer+1; Stack [Stack pointer] ← PC + 3 ; PC ← Address
RET	Fetch instruction Pop from stack decrement s.pointer	Instruction ← MEM[PC] PC ← Stack [Stack pointer] ; Stack pointer ← Stack pointer-1;

Datapath components:

1. PC (Program Counter) register :12-bits to select the instruction from instruction memory.
2. Instruction memory: 4K-byte. Each instruction store in 3-byte. It also has one input for address 12-bits and one output for instruction 19-bits
3. Adder: to increment the PC by 3 to next instruction
4. Register file: consist from 8 registers R0 to R7 each one 8-bits, R0 always zero. The register file also has three input registers each one 3-bits (Read source register r1, Read source register r1, Write destination register rd), the fourth input is BusW that hold the date that will write in rd. It also has two output registers (Bus1 that hold the data from register r1, Bus2 that hold the data from register r2)
5. ALU: to do the operations
6. One flipflop for Z
7. One flipflop for C.
8. Data memory: 256 bytes. It has two input each one 8-bits one for address and the anther one for data that stor in memory.
9. Return-address stack: to save the return address
10. Stake pointer: to determine the place for puch and pop from stake
11. Port in : 256 bytes. To accepted data from outside. It has one input 8-bits for address and one output 8-bits for data

12. Port out : 256 bytes. To receive data. It has two input each one 8-bits one for address and the another one for date.
13. Multiplexers: we will show the details of them in Datapath part.

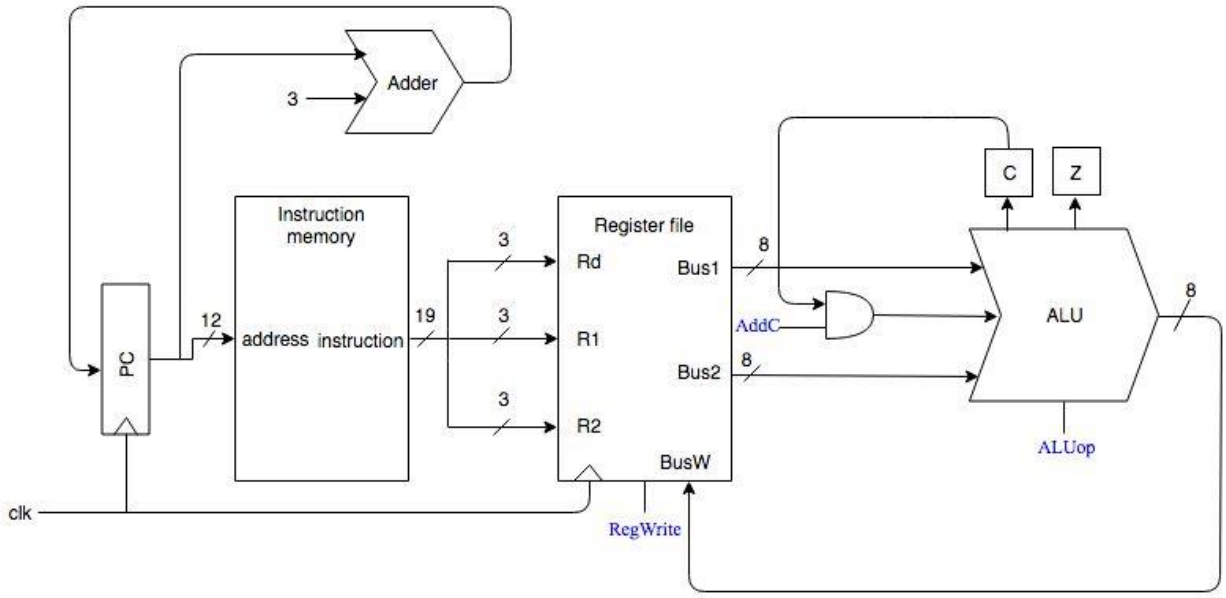
List of all control signals :

Signal	Effect when (0)	Effect when (1)
ALUop (4-bits)	None	The ALU do the operation depends on the signal from ALU control unit
StackWrite	None	Stake is written Stack [Stack pointer] ← data in
RegWrite	None	Enable to write the value in BusW to Rd
PortWrite	None	PORT is written PORT [Address] ← data in
MemWrite	None	Stake is written Memory[Address] ← data in
StaPointer	Stack pointer ← Stack pointer+1	Stack pointer ← Stack pointer-1
AddC	None	Add the carry to the operand of ALU
JMB	The result of branch multiplexor	PC ← JMB or JSBAddress
BET	The result of JMB multiplexor	PC ← Stack [Stack pointer]
BZ	PC ← PC+3	PC ← PC + 3+ (disp.8)
BNZ	PC ← PC+3	PC ← PC + 3+ (disp.8)
BC	PC ← PC+3	PC ← PC + 3+ (disp.8)
BNC	PC ← PC+3	PC ← PC + 3+ (disp.8)
Shift	Second ALU operand is not Sc	Second ALU operand is Sc
Immed.	Second ALU operand is r2	Second ALU operand is immediate value 8-bits
Bus2	The value of Bus2 from r2	The value of Bus2 from rd
PorToReg	BusW=ALU result	BusW= data out from PORT IN
MemToReg	The result from PorToReg multiplexor	BusW= data out from Memory

The Datapath:

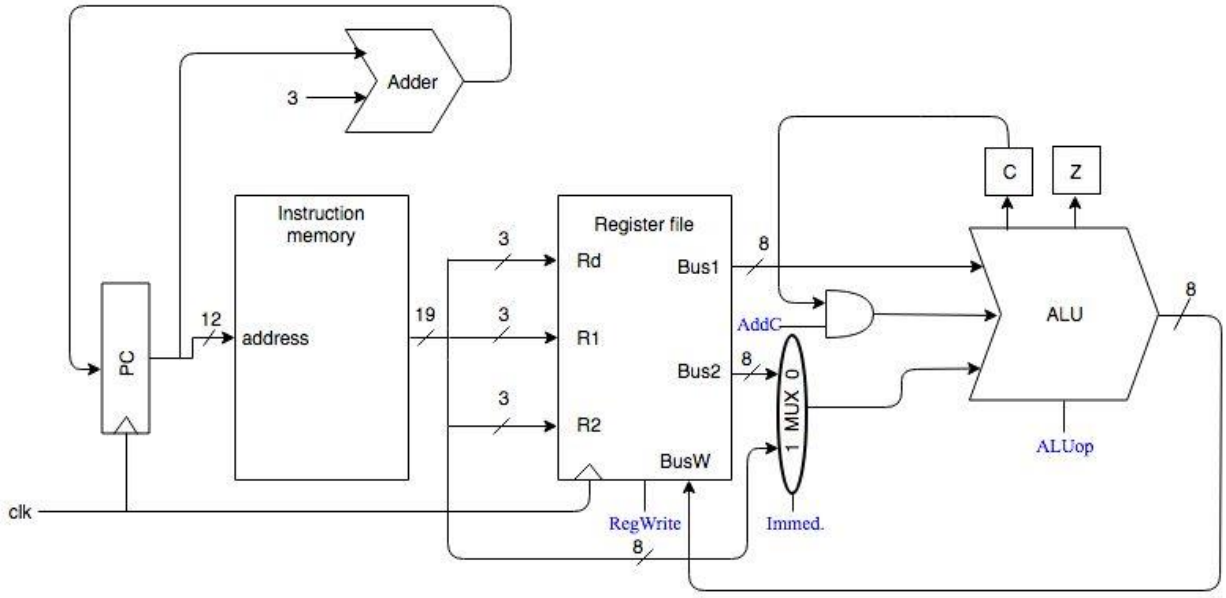
All these design without control signal

1. ### The Datapath of R-type:



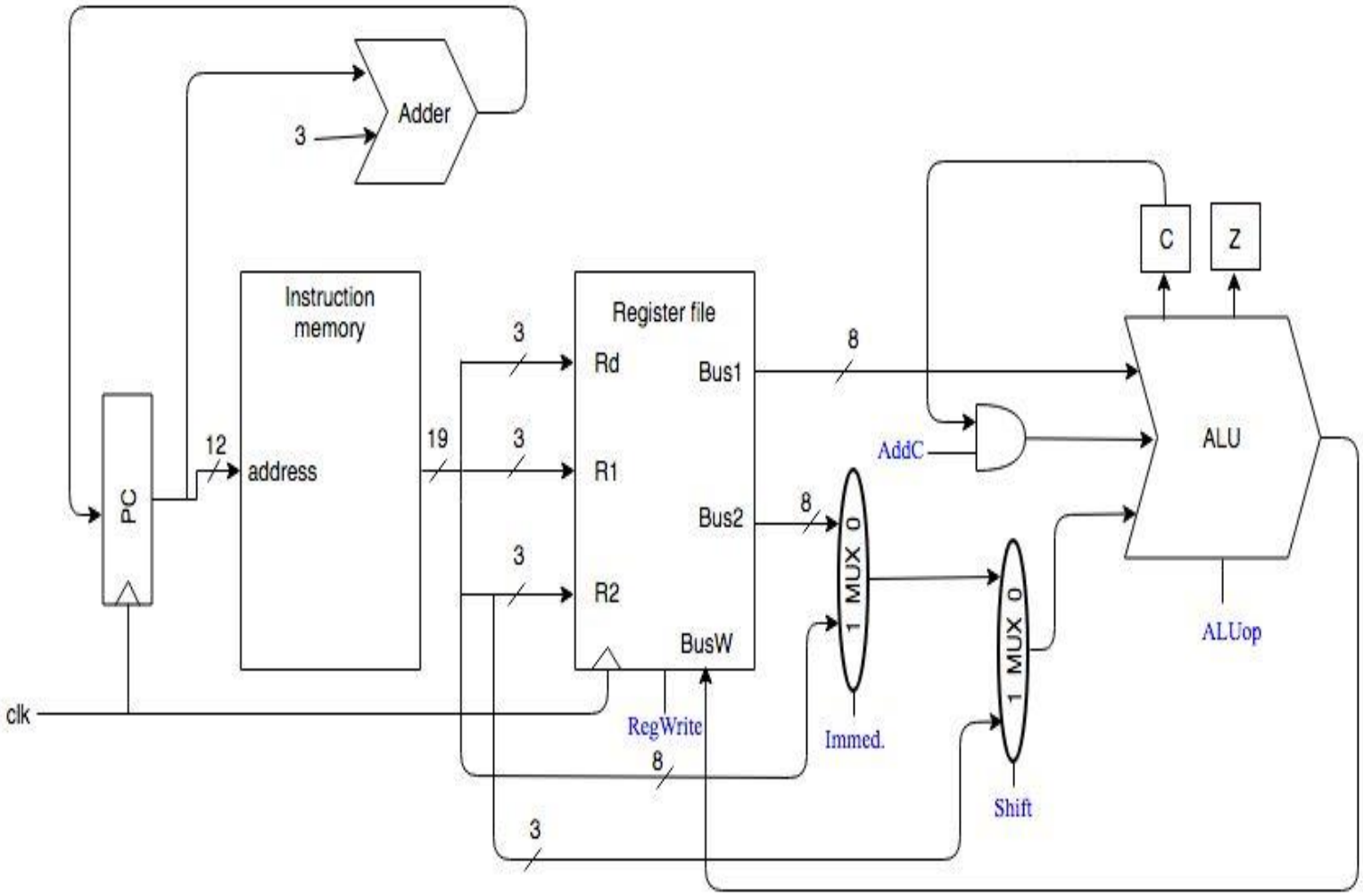
2. ### Add The Datapath of I-type:

We add immed. Multiplexor to select the immediate value as a second operand for ALU.



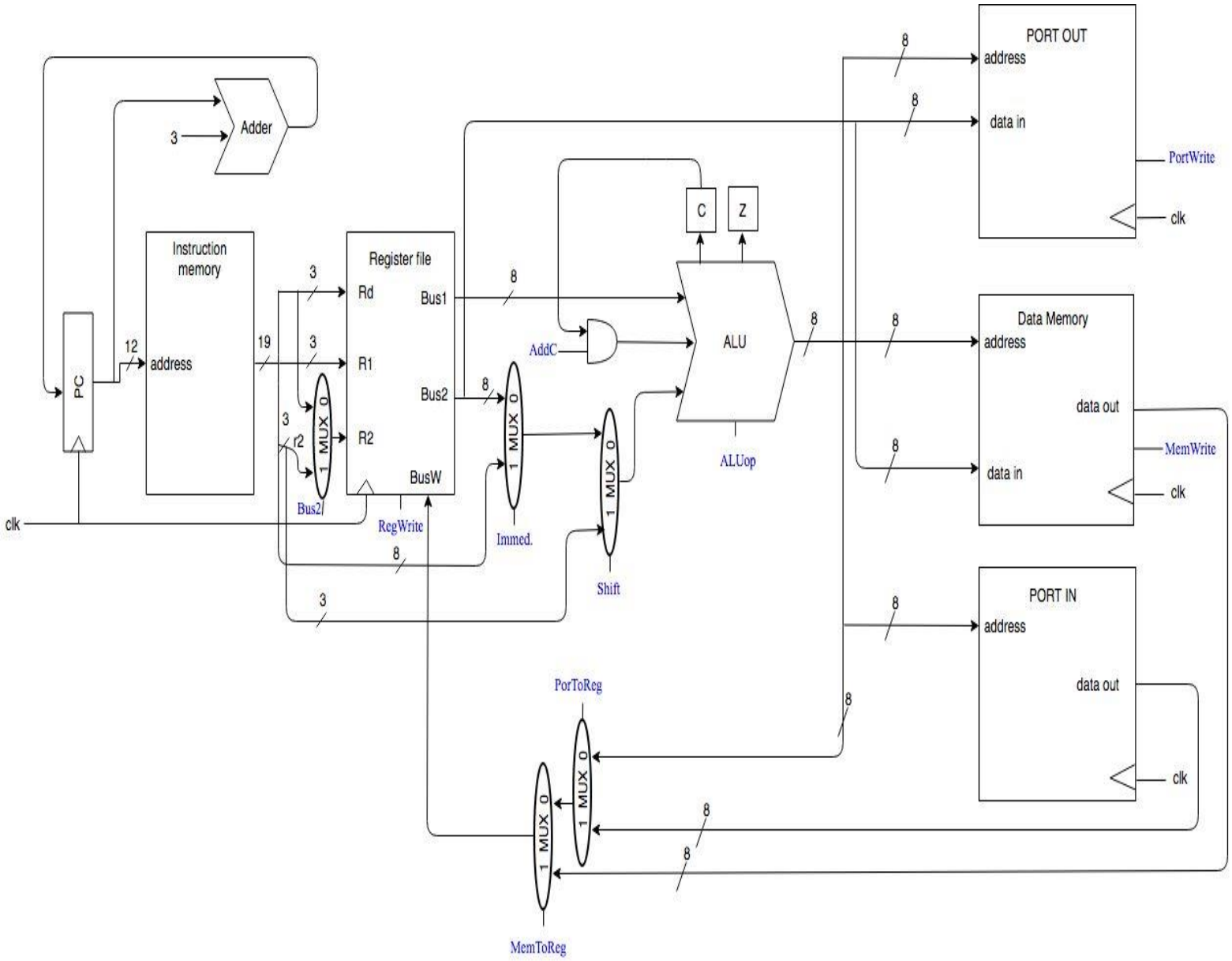
3. Add The Datapath of Shift-type:

We add Shift Multiplexor to select the Sc value as a second operand for ALU.



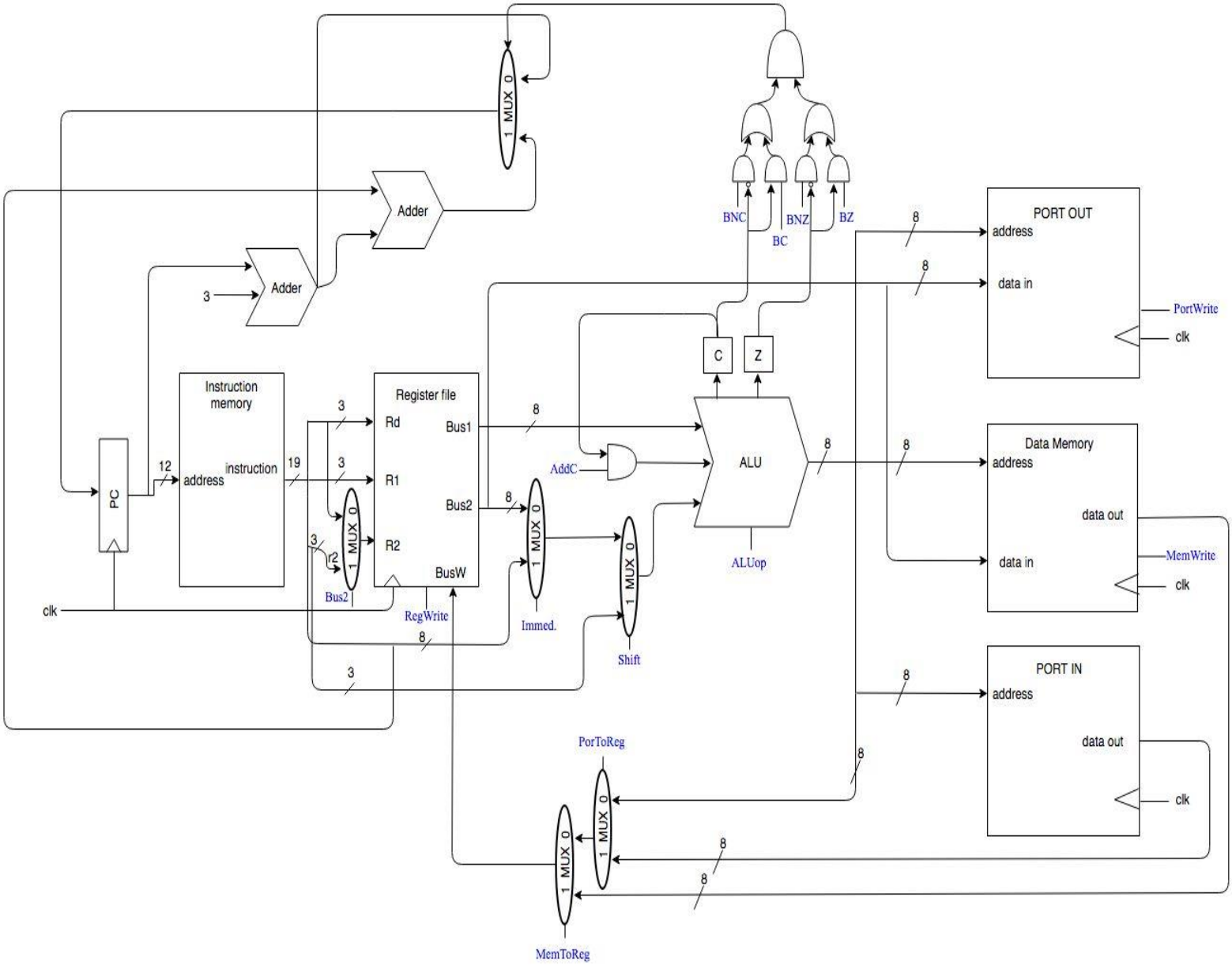
4. Add The Datapath of Memory and I/O type:

We add the PORT IN, PORT OUT and Data Memory and PorToReg Multiplexor to select the data from PORT IN to write in register file and MemToReg Multiplexor to select the data from Data Memory to write in register file and Bus2 Multiplexor to put the value in Rd in Bus2 register.



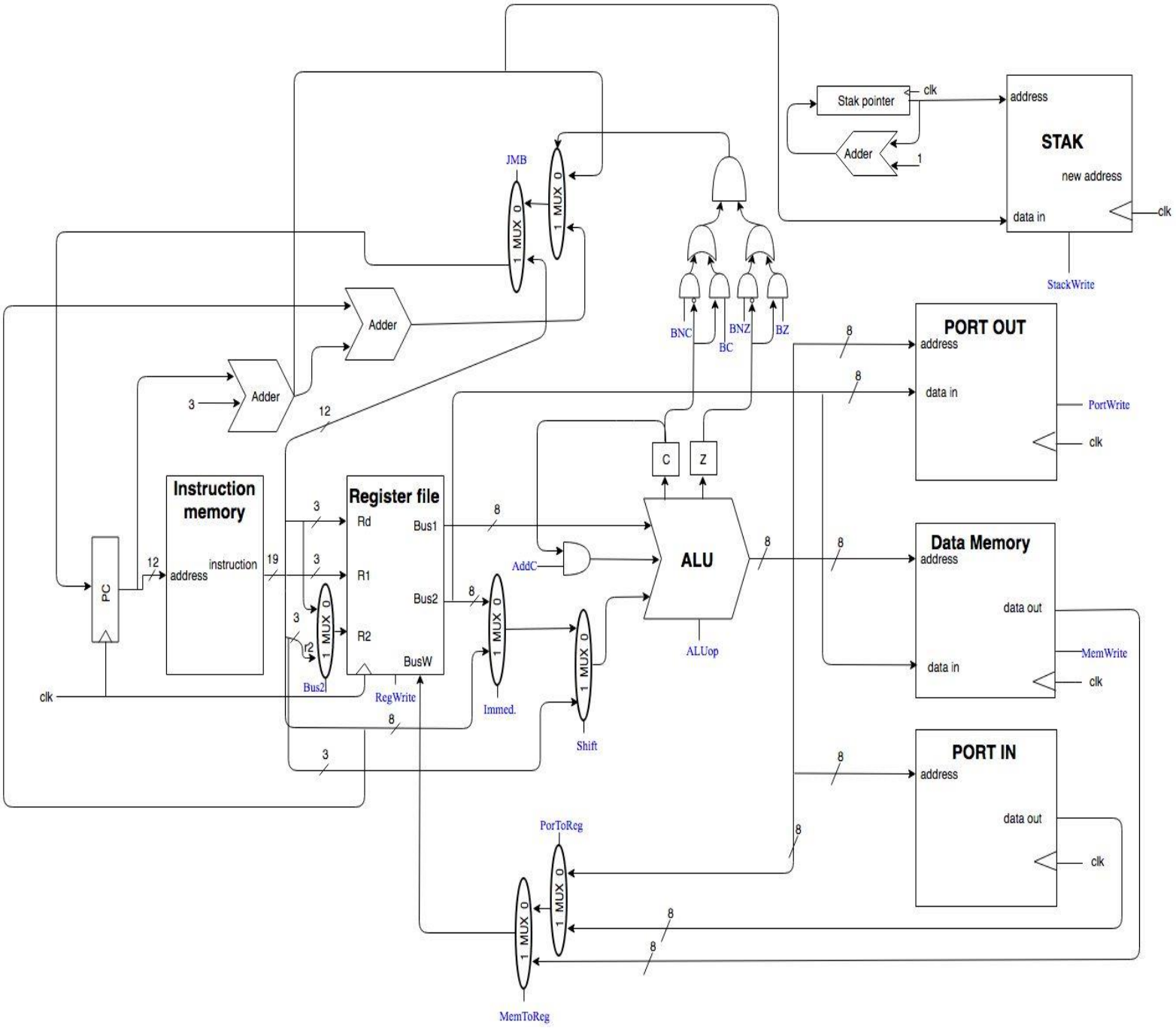
5. Add The Datapath of Conditional Branch:

We add Multiplexor that select the branch value depend the output of the branch combination circuit



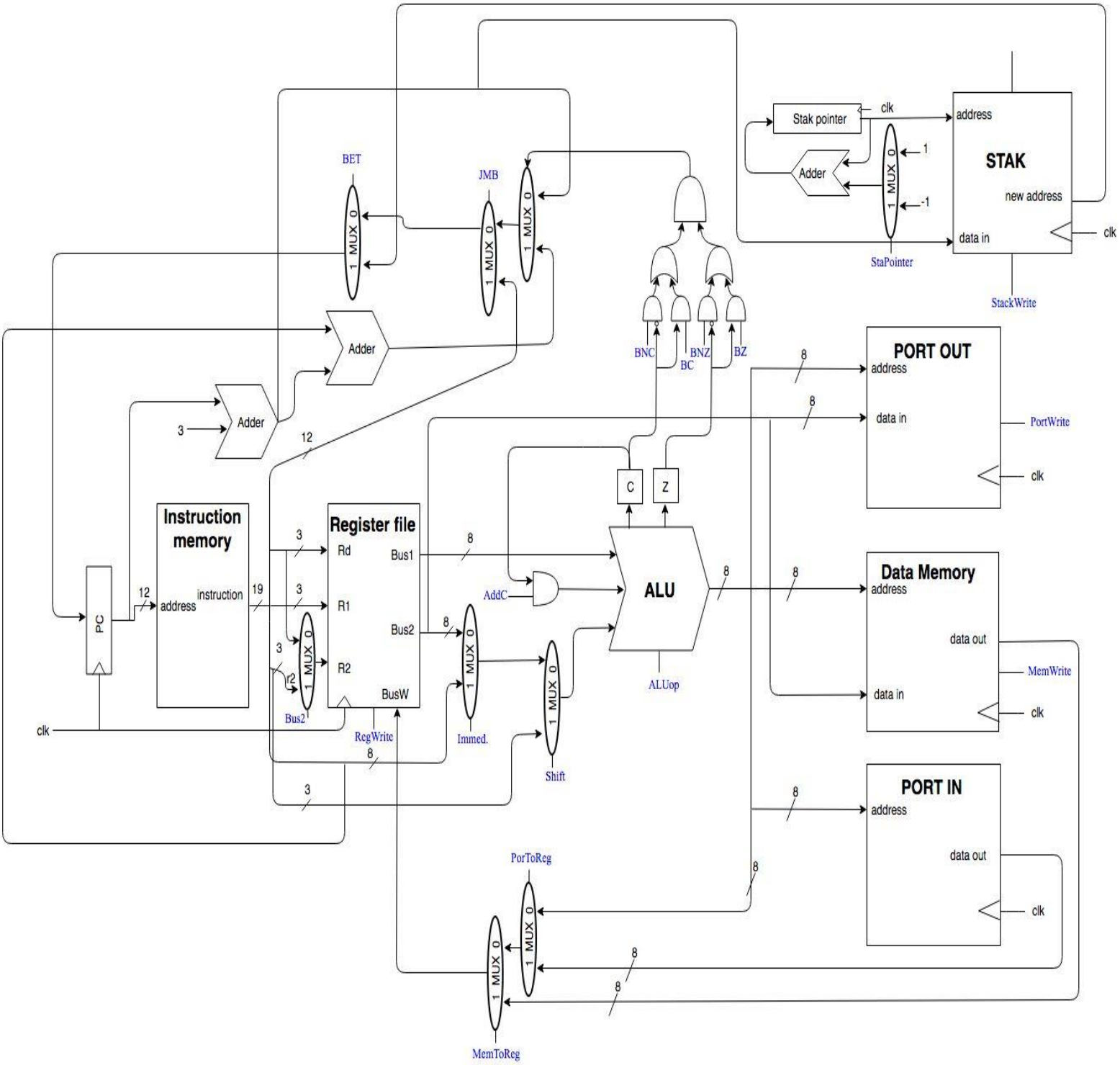
6. Add The Datapath of Jump:

We add JMB Multiplexor that select the jump value as a next PC and the STAK block that save the next instruction after JMB instruction.

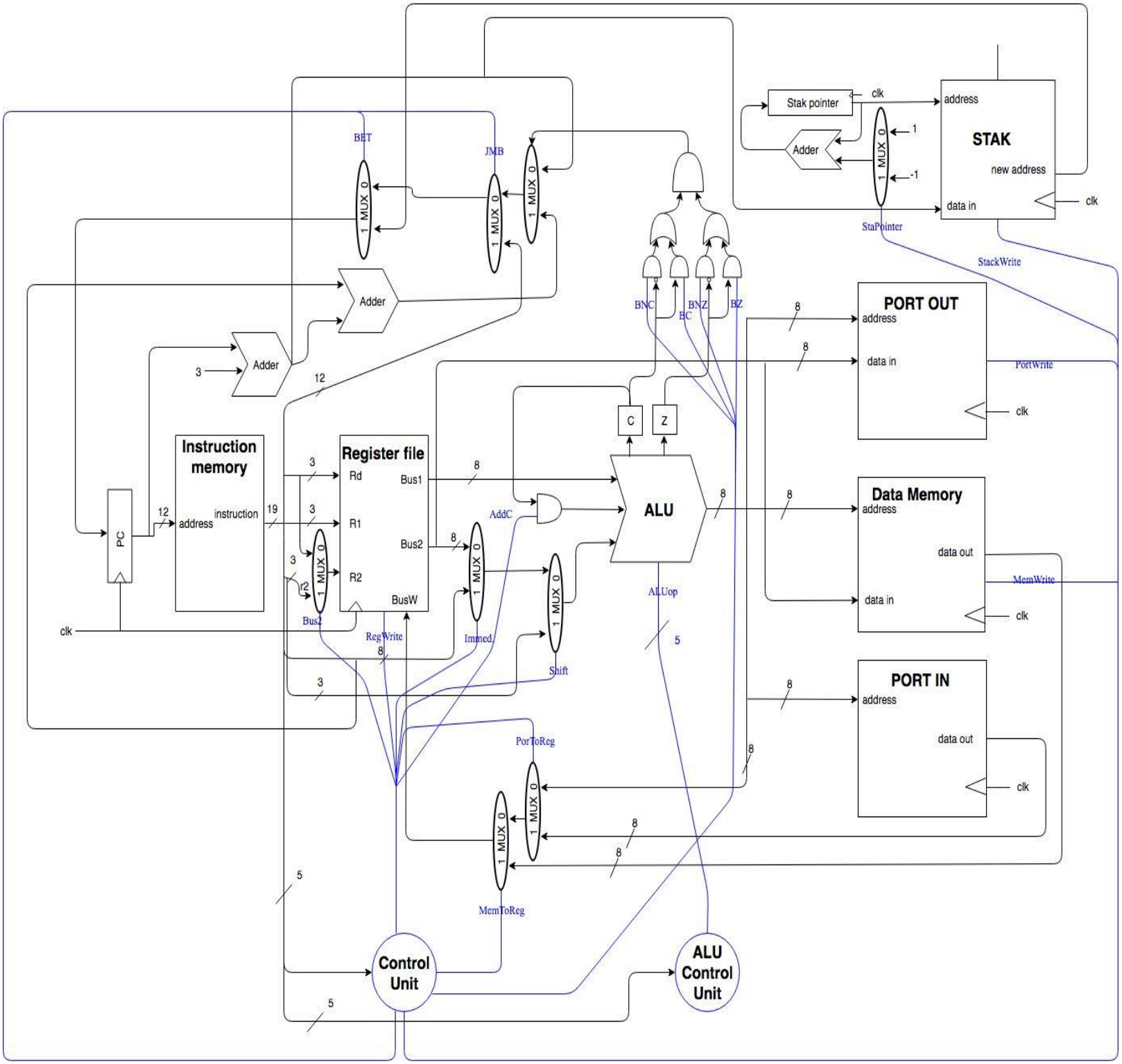


7. Add The Datapath of Miscellaneous:

We add RET Multiplexor that select the popping value from stake as next PC



The Datapath with all necessary multiplexors and all control lines:



ALU input signals:

Function	ALUop
ADD	0000
SUB	0001
AND	0010
OR	0011
XOR	0100
MSK	0101
SHL	0110
SHR	0111
ROL	1000
ROR	1001

The design of control units :

In our design we will use two control unit (the control unit and the ALU control unit) each one will take the first 5-bits from the instruction to know the operation and the function. The operation bits for Miscellaneous Instructions are 6-bits but we will just take 5-bits because we will not implement the interrupt and there will be no repetition.

- ### The design of ALU control unit :

Input		Output	
ADD	00000	ADD	0000
ADDC	00001	ADD	0000
SUB	00010	SUB	0001
SUBC	00011	SUB	0001
AND	00100	AND	0010
OR	00101	OR	0011
XOR	00110	XOR	0100
MSK	00111	MSK	0101
ADDI	01000	ADD	0000
ADDCI	01001	ADD	0000
SUBI	01010	SUB	0001
SUBCI	01011	SUB	0001
ANDI	01100	AND	0010
ORI	01101	OR	0011
XORI	01110	XOR	0100
MSKI	01111	MSK	0101
SHL	11000	SHL	0110
SHR	11001	SHR	0111
ROL	11010	ROL	1000
ROR	11011	ROR	1001
LDM	10000	ADD	0000
STM	10001	ADD	0000
INP	10010	ADD	0000
OUT	10011	ADD	0000
BZ	10100	X	X
BNZ	10101	X	X

BC	10110	X	X
BNC	10111	X	X
JMP	11100	X	X
JSB	11101	X	X
RET	11110	X	X

- The design of the control unit :

Input		output															
		Stack Write	Reg Write	Port Write	Mem Write	Sta Pointer	AddC	JMB	BET	BZ	BNZ	BC	BNC	Shift	Immed.	Bus2	Por To Reg	Mem To Reg
ADD	00000	0	1	0	0	X	0	0	0	0	0	0	0	0	0	0	0	0
ADDC	00001	0	1	0	0	X	1	0	0	0	0	0	0	0	0	0	0	0
SUB	00010	0	1	0	0	X	0	0	0	0	0	0	0	0	0	0	0	0
SUBC	00011	0	1	0	0	X	1	0	0	0	0	0	0	0	0	0	0	0
AND	00100	0	1	0	0	X	0	0	0	0	0	0	0	0	0	0	0	0
OR	00101	0	1	0	0	X	0	0	0	0	0	0	0	0	0	0	0	0
XOR	00110	0	1	0	0	X	0	0	0	0	0	0	0	0	0	0	0	0
MSK	00111	0	1	0	0	X	0	0	0	0	0	0	0	0	0	0	0	0
ADDI	01000	0	1	0	0	X	0	0	0	0	0	0	0	0	1	0	0	0
ADDCI	01001	0	1	0	0	X	1	0	0	0	0	0	0	0	1	0	0	0
SUBI	01010	0	1	0	0	X	0	0	0	0	0	0	0	0	1	0	0	0
SUBCI	01011	0	1	0	0	X	1	0	0	0	0	0	0	0	1	0	0	0
ANDI	01100	0	1	0	0	X	0	0	0	0	0	0	0	0	1	0	0	0
ORI	01101	0	1	0	0	X	0	0	0	0	0	0	0	0	1	0	0	0
XORI	01110	0	1	0	0	X	0	0	0	0	0	0	0	0	1	0	0	0
MSKI	01111	0	1	0	0	X	0	0	0	0	0	0	0	0	1	0	0	0
SHL	11000	0	1	0	0	X	0	0	0	0	0	0	0	1	0	0	0	0
SHR	11001	0	1	0	0	X	0	0	0	0	0	0	0	1	0	0	0	0
ROL	11010	0	1	0	0	X	0	0	0	0	0	0	0	1	0	0	0	0
ROR	11011	0	1	0	0	X	0	0	0	0	0	0	0	1	0	0	0	0
LDM	10000	0	1	0	0	X	0	0	0	0	0	0	0	0	1	0	0	1
STM	10001	0	0	0	1	X	0	0	0	0	0	0	0	0	1	1	X	X
INP	10010	0	1	0	0	X	0	0	0	0	0	0	0	0	1	0	1	0
OUT	10011	0	0	1	0	X	0	0	0	0	0	0	0	0	1	1	X	X
BZ	10100	0	0	0	0	X	X	0	0	1	0	0	0	X	X	X	X	X
BNZ	10101	0	0	0	0	X	X	0	0	0	1	0	0	X	X	X	X	X
BC	10110	0	0	0	0	X	X	0	0	0	0	1	0	X	X	X	X	X
BNC	10111	0	0	0	0	X	X	0	0	0	0	0	1	X	X	X	X	X
JMP	11100	0	0	0	0	X	X	1	0	0	0	0	0	X	X	X	X	X
JSB	11101	1	0	0	0	0	X	1	0	0	0	0	0	X	X	X	X	X
RET	11110	0	0	0	0	1	X	0	1	0	0	0	0	X	X	X	X	X