Computer Organization Laboratory CS39001
Assignment 07 — Single Cycle RISC Processor Design

• Instruction Set Architecture

KGP-RISC has the below *Instruction Set Architecture* (ISA).

Class	Instruction	Usage	Meaning
Arithmetic	*Add*	add rs, rt	$rs \leftarrow (rs) + (rt)$
	Complement	comp rs, rt	$rs \leftarrow 2's\ Complement\ (rt)$
	Add Immediate	addi rs, imm	$rs \leftarrow (rs) + imm$
	Complement Immediate	compi rs, imm	$rs \leftarrow 2's\ Complement\ (imm)$
Logic	*Logical AND*	and rs, rt	$rs \leftarrow (rs) \wedge (rt)$
	Logical XOR	xor rs, rt	$rs \leftarrow (rs) \oplus (rt)$
Shift	*Shift Left Logical*	shll rs, sh	$rs \leftarrow (rs)\ log\ left\ shifted\ by\ sh$
	Shift Right Logical	shrl rs, sh	$rs \leftarrow (rs)\ log\ right\ shifted\ by\ sh$
	Shift Left Logical Variable	shllv rs, rt	$rs \leftarrow (rs)\ log\ left\ shifted\ by\ (rt)$
	Shift Right Logical Variable	shrlv rs, rt	$rs \leftarrow (rs)\ log\ right\ shifted\ by\ (rt)$
	Shift Right Arithmetic	shra rs, sh	$rs \leftarrow (rs)\ arith\ right\ shifted\ by\ sh$
	Shift Right Arithmetic Variable	shrav rs, rt	$rs \leftarrow (rs)\ arith\ right\ shifted\ by\ (rt)$
Memory	*Load Word*	lw rt, imm(rs)	$rt \leftarrow mem[(rs) + imm]$
	Store Word	sw rt, imm(rs)	$mem[(rs) + imm] \leftarrow (rt)$
Branch	*Unconditional Branch*	b L	$goto\ L$
	Branch Register	br rs	$goto\ (rs)$
	Branch on < 0	bltz rs, L	$if\ (rs) < 0\ then\ goto\ L$
	Branch on flag 0	bz rs, L	$if\ (rs) == 0\ then\ goto\ L$
	Branch on flag not 0	bnz rs, L	$if\ (rs)\ != 0\ then\ goto\ L$
	Branch and Link	bl L	$goto\ L;\ (ra) \leftarrow (PC) + 4$
	Branch on Carry	bcy L	$if\ Carry == 1\ then\ goto\ L$
	Branch on No Carry	bncy L	$if\ Carry == 0\ then\ goto\ L$

• Assumptions

- Word length of our processor is 32 bits.
- All registers and memory elements have 32 bit data.
- The address line of the memory is also 32 bit.
- The processor has a single-cycle instruction execution unit i.e., every instruction will take one clock cycle to complete execution. There is no pipelining.

• Instruction Encodings

Register Addressing Mode

Instruction Encoding Type: **R-Format (Register Addressing Mode)**
Instructions having the same opcode can be differentiated by the bits in the $funct$ field.

Opcode: **000 000**

[These instructions require bitwise arithmetic to be performed on the operands]

add rs, rt
and rs, rt
xor rs, rt
comp rs, rt

Opcode: **000 001**

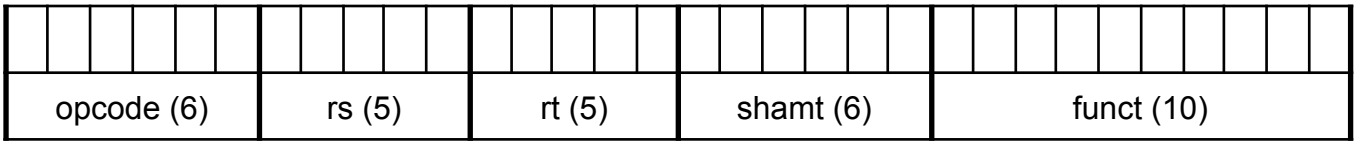
[These instructions require shifting of bits in a register by a 6-bit shift amount]

shll rs, sh
shrl rs, sh
shra rs, sh

Opcode: **000 010**

[These instructions require shifting bits in a register by a 32-bit shift amount stored in another register]

shllv rs, rt
shrlv rs, rt
shrav rs, rt



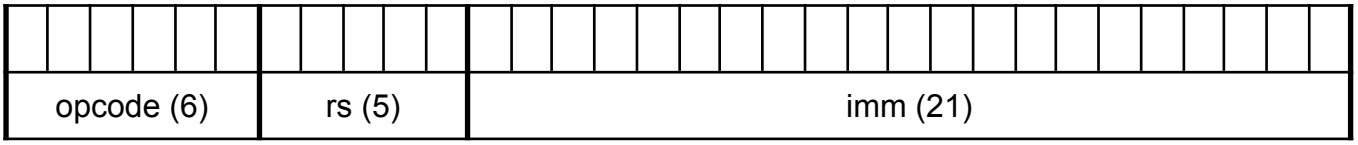
	opcode	rs	rt	shamt	funct
add	000 000	rs	rt	—	0 000 000 000
and		rs	rt	—	0 000 000 001
xor		rs	rt	—	0 000 000 010
comp		rs	rt	—	0 000 000 011
shll	000 001	rs	—	sh	0 000 000 000
shrl		rs	—	sh	0 000 000 001
shra		rs	—	sh	0 000 000 010
shllv	000 010	rs	rt	—	0 000 000 000
shrlv		rs	rt	—	0 000 000 001
shrav		rs	rt	—	0 000 000 010

Immediate Addressing Mode (Non-Memory Based)

Instruction Encoding Type: **I-Format (Immediate Addressing Mode)**
Instructions can only be differentiated by the $opcode$.

Opcode: **000 011**
[This instruction requires adding a 21 bit immediate value to the value stored in a register]
addi rs, imm

Opcode: **000 100**
[This instruction requires computing 2's complement of a 21 bit immediate value]
compi rs, imm



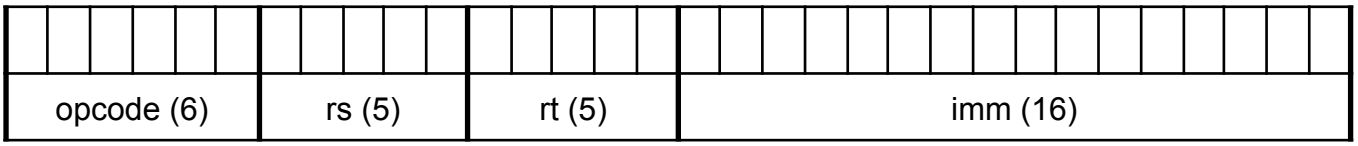
	opcode	rs	imm
addi	000 011	rs	imm
compi	000 100	rs	imm

Immediate Addressing Mode (Memory Based)

Instruction Encoding Type: **I-Format (Immediate Addressing Mode)**
Instructions can only be differentiated by the $opcode$.

Opcode: **000 101**
[This instruction loads a 4 byte word from the specified memory location to a register]
lw rt, imm(rs)

Opcode: **000 110**
[This instruction stores the value in a register at a specified memory location]
sw rt, imm(rs)



	opcode	rs	rt	imm
lw	000 101	rs	rt	imm
sw	000 110	rs	rt	imm

Pseudo-Direct Addressing Mode

Instruction Encoding Type: **(Pseudo-Direct Addressing Mode)**
Instructions can only be differentiated by the $opcode$.

Opcode: **000 111**

[This instruction updates the program counter with a new instruction address.]

b L

Opcode: **001 000**

[This instruction updates the program counter with a new instruction address and stores the current address in the program counter in a register $ra]

bl L

Opcode: **001 001**

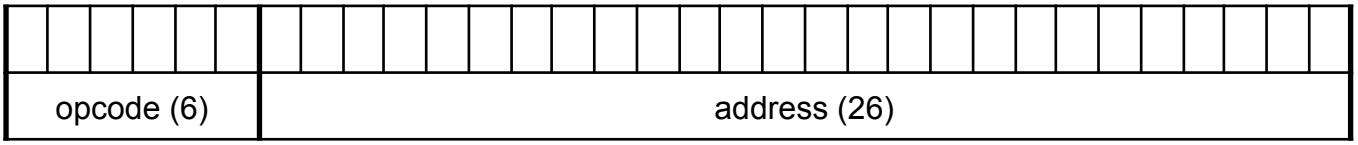
[This instruction updates the program counter with a new instruction address if carry-out of the ALU is 1]

bcy L

Opcode: **001 010**

[This instruction updates the program counter with a new instruction address if carry-out of the ALU is 0]

bncy L



	opcode	address
b	000 111	L
bl	001 000	L
bcy	001 001	L
bncy	001 010	L

PC-Relative Addressing Mode

Instruction Encoding Type: **(PC-relative Addressing Mode)**

Instructions can only be differentiated by the $opcode$.

Opcode: **001 111**

[This instruction updates the program counter with a new instruction address stored in a register]

br rs

Opcode: **001 111**

[This instruction updates the program counter with a new instruction address if the value stored in the register is less than 0]

bltz rs, L

Opcode: **001 111**

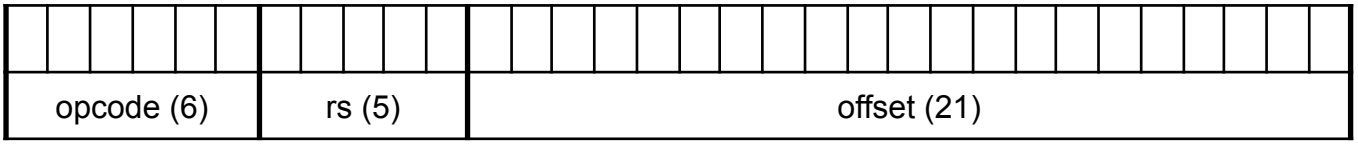
[This instruction updates the program counter with a new instruction address if the value stored in the register is equal to 0]

bz rs, L

Opcode: **001 111**

[This instruction updates the program counter with a new instruction address if the value stored in the register is not equal to 0]

bnz rs, L



	opcode	rs	offset
br	001 011	rs	—
bltz	001 100	rs	L
bz	001 101	rs	L
bnz	001 110	rs	L

• Future Extensions

The *opcode* field is given 6 bits in the instruction encoding. This means that the *KGP-RISC* architecture can have 64 instructions with different opcodes. 15 of the 64 opcodes are used already. Therefore, the **ISA can be extended to include 49 new instructions with different opcodes**. Secondly, for the instructions belonging to R-type formatting, 10 bits are reserved for the *funct* field that identifies the function type for similar types of arithmetic instructions. This means that for each opcode, 1024 distinct R-type instructions can be written in the ISA. 3 opcodes are used already in the R-type instructions for which respectively 4, 3, 3 instructions are written with different *funct* fields. So **for the already existing opcodes in the R-type instructions, a total of 3062 new R-type instructions can be written without using any new opcodes**. Adding to this number in the best case if all the 49 remaining opcodes are used for R-type instructions, since each opcode can support 1024 R-type instructions, **a maximum of 53238 new R-type instructions can be included in the architecture**.

• Datapath Elements

The various datapath elements that will be needed in the processor that implements the KGP-RISC ISA are as follows.

– Program Counter
– Instruction Memory
– Register File
– Arithmetic Logic Unit
– Data Memory
– Sign Extension Modules
– Main Controller
– ALU Control Unit
– 32-Bit Adder
– Multiplexors
– Concatenation Modules
– Zero Padding Modules

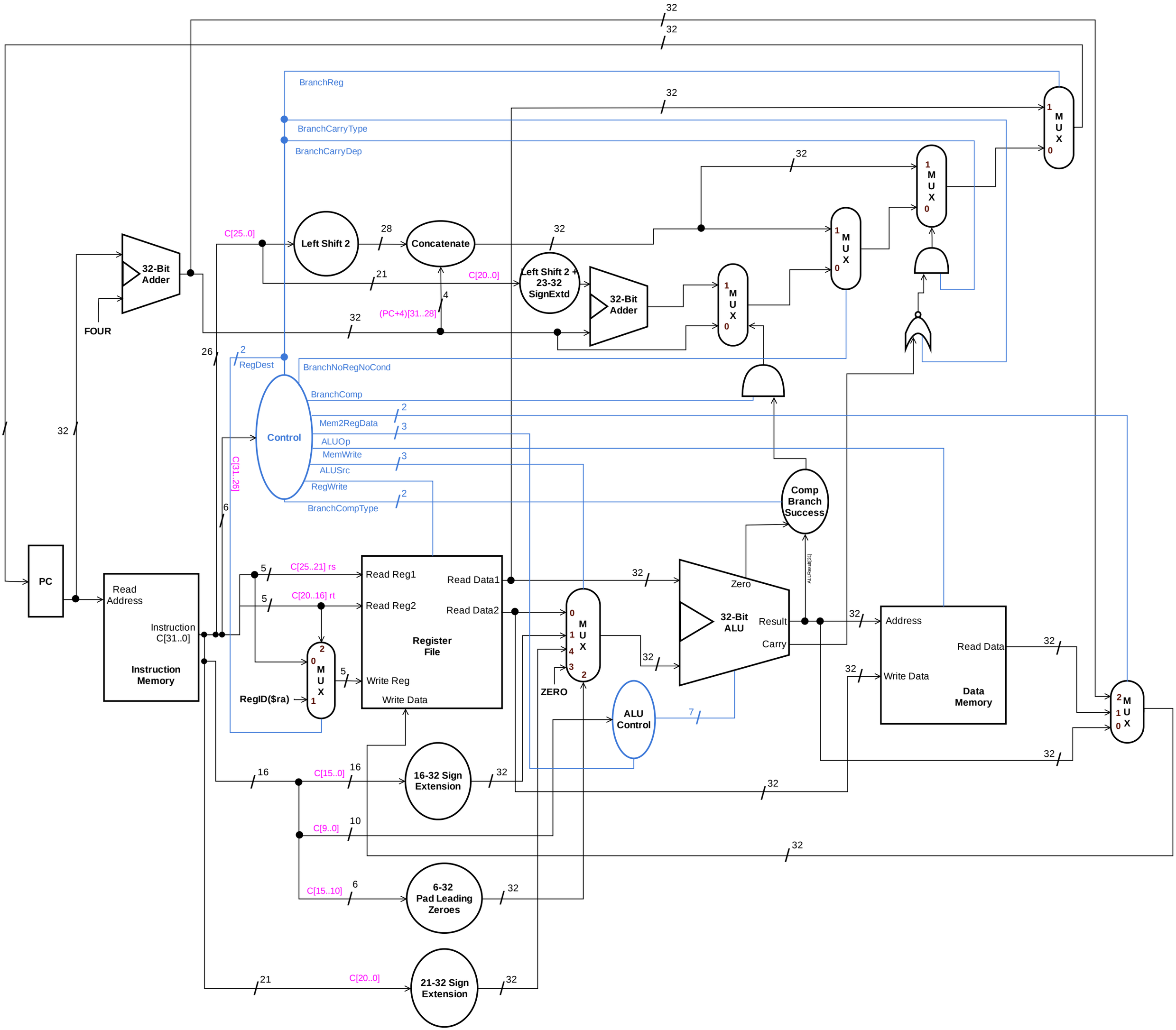
Multiplexors of different bus widths and number of inputs will be needed. Similarly, different sign extension modules for different input bus widths will be needed. Besides these, some other helper modules may also be needed to modularize the hardware design (like a special module that determines the success of a branch instruction). These modules will become absolutely clear in the datapath diagram of the single cycle execution unit.

• Processor Design for Single Cycle Execution Unit

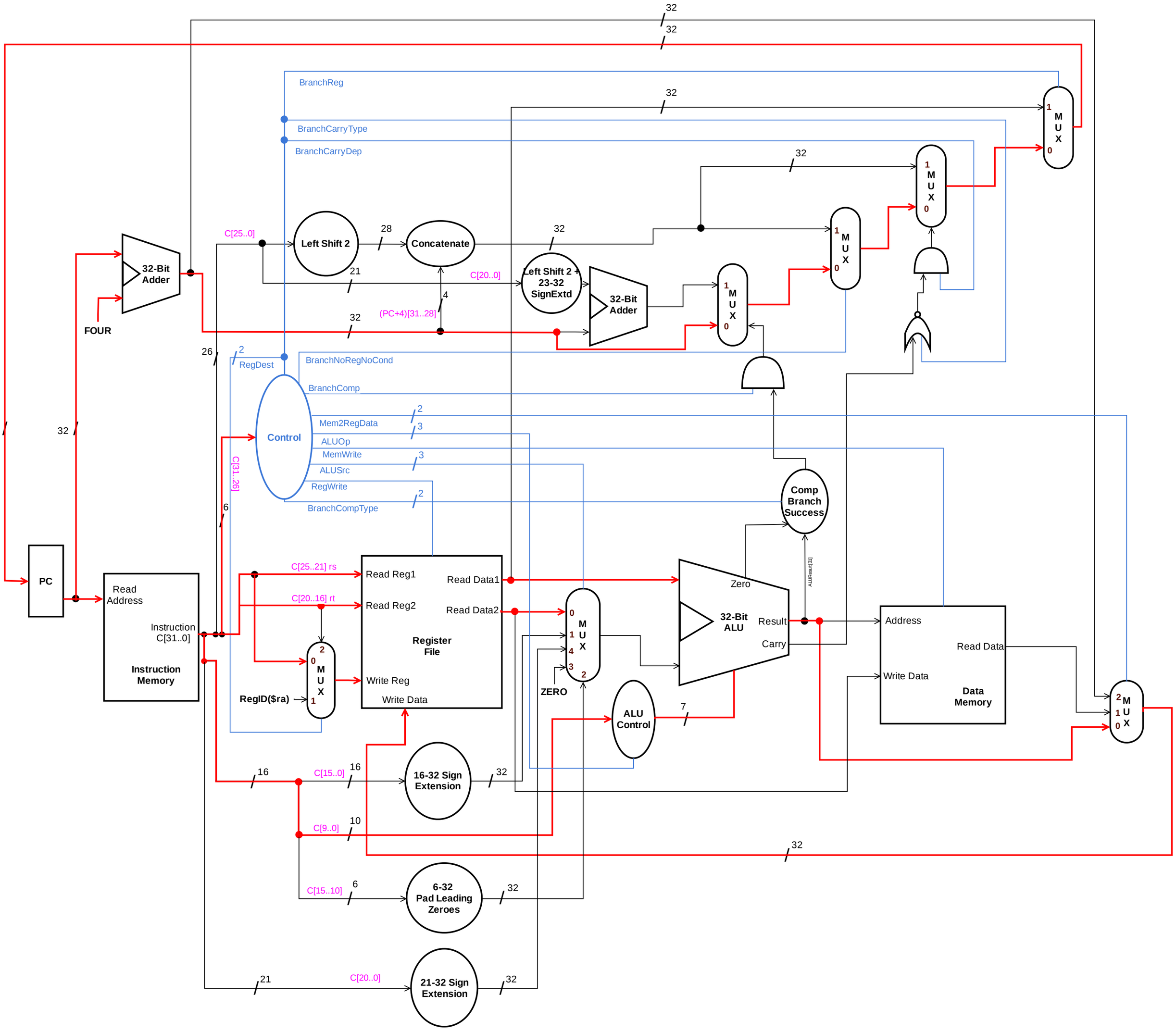
The next page shows the complete datapath of the ***Single Cycle Execution RISC Architecture*** with the control signals (*highlighted in blue*). All the datapath elements are clearly shown.

Following that, we have highlighted (*in bold red*) the datapath for each of the instructions that shows the various phases of single cycle execution, from instruction fetch to write-data, that the processor must go through to execute these instructions successfully. These independent datapath diagrams for each instruction will make the functionality of the respective instructions of *KGP-RISC ISA* much more clearer.

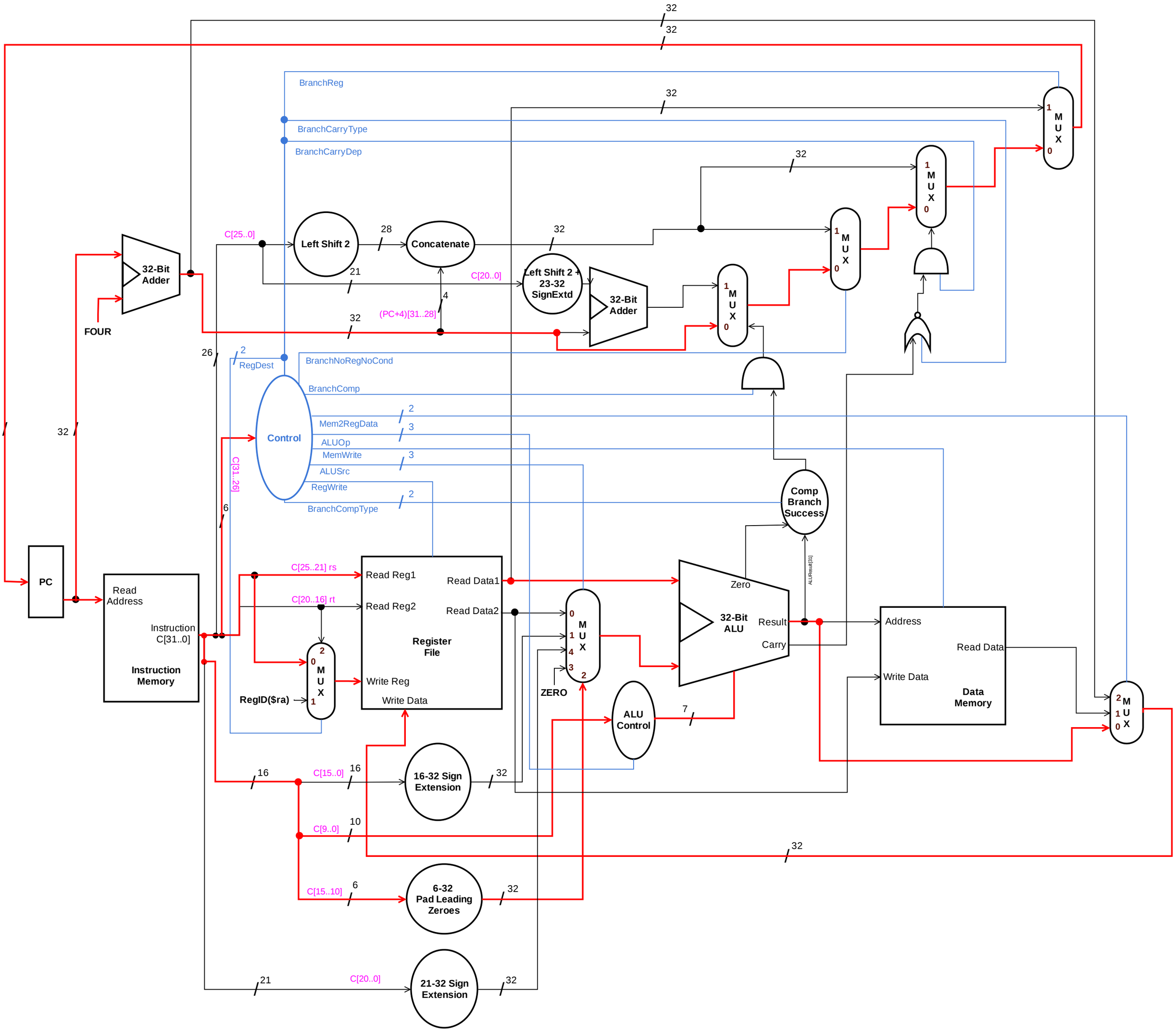
Single Cycle Execution Unit for KGP-RISC Architecture



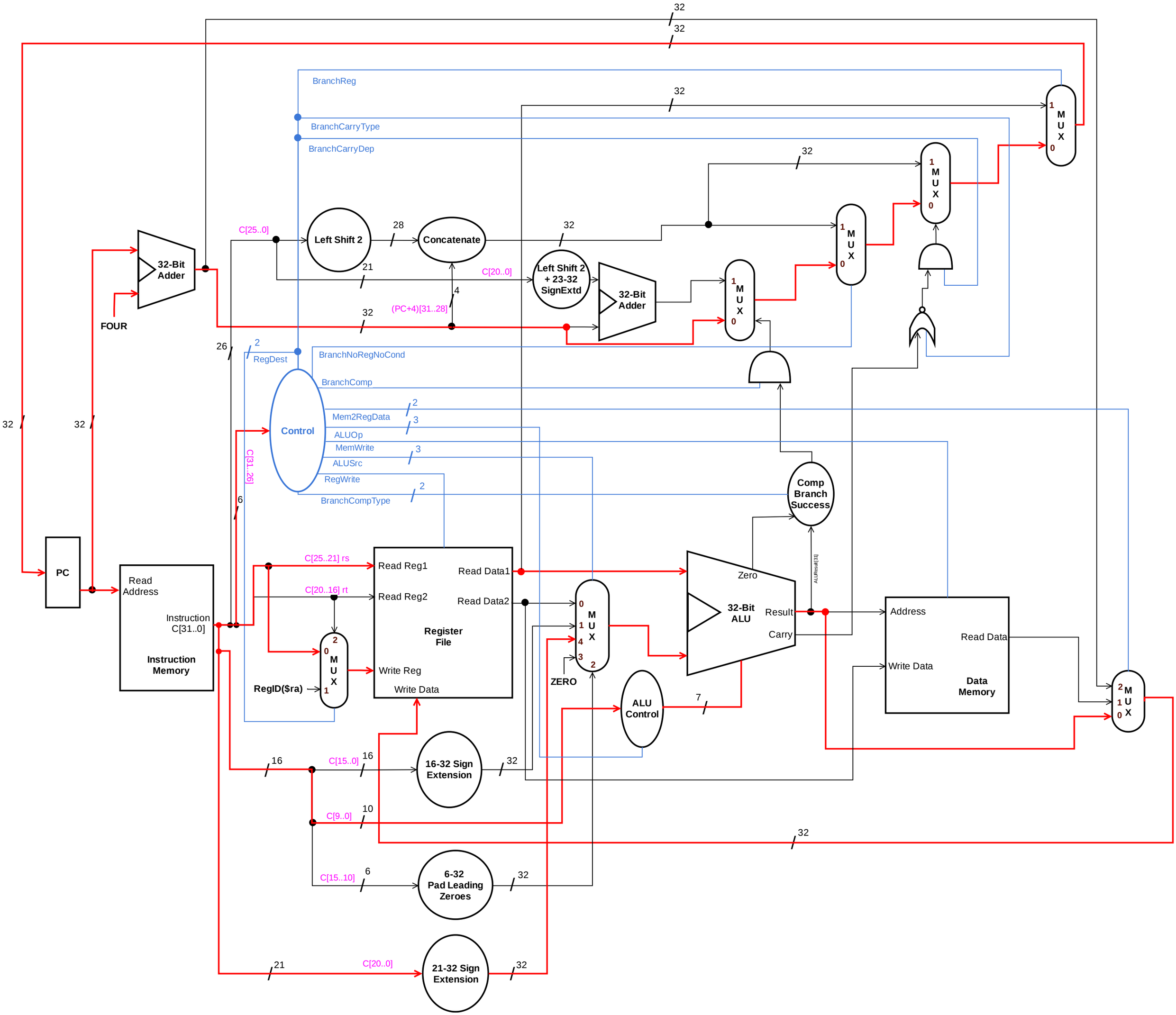
Datapath for R-Type Instructions
(*add*, *and*, *xor*, *comp*, *shllv*, *shrlv*, *shrav*)



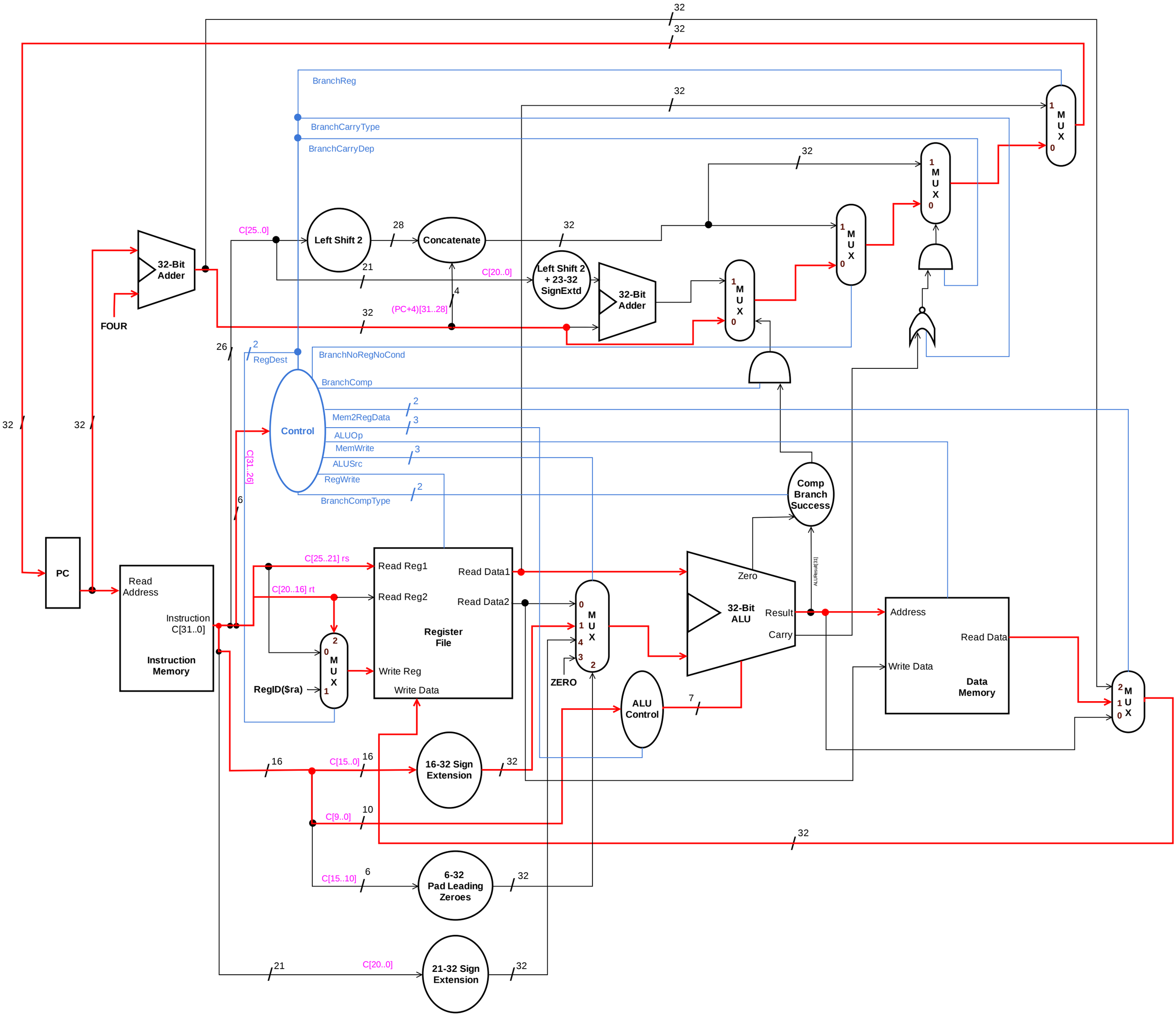
Datapath for R-Type Instructions
(*shll*, *shrl*, *shra*)



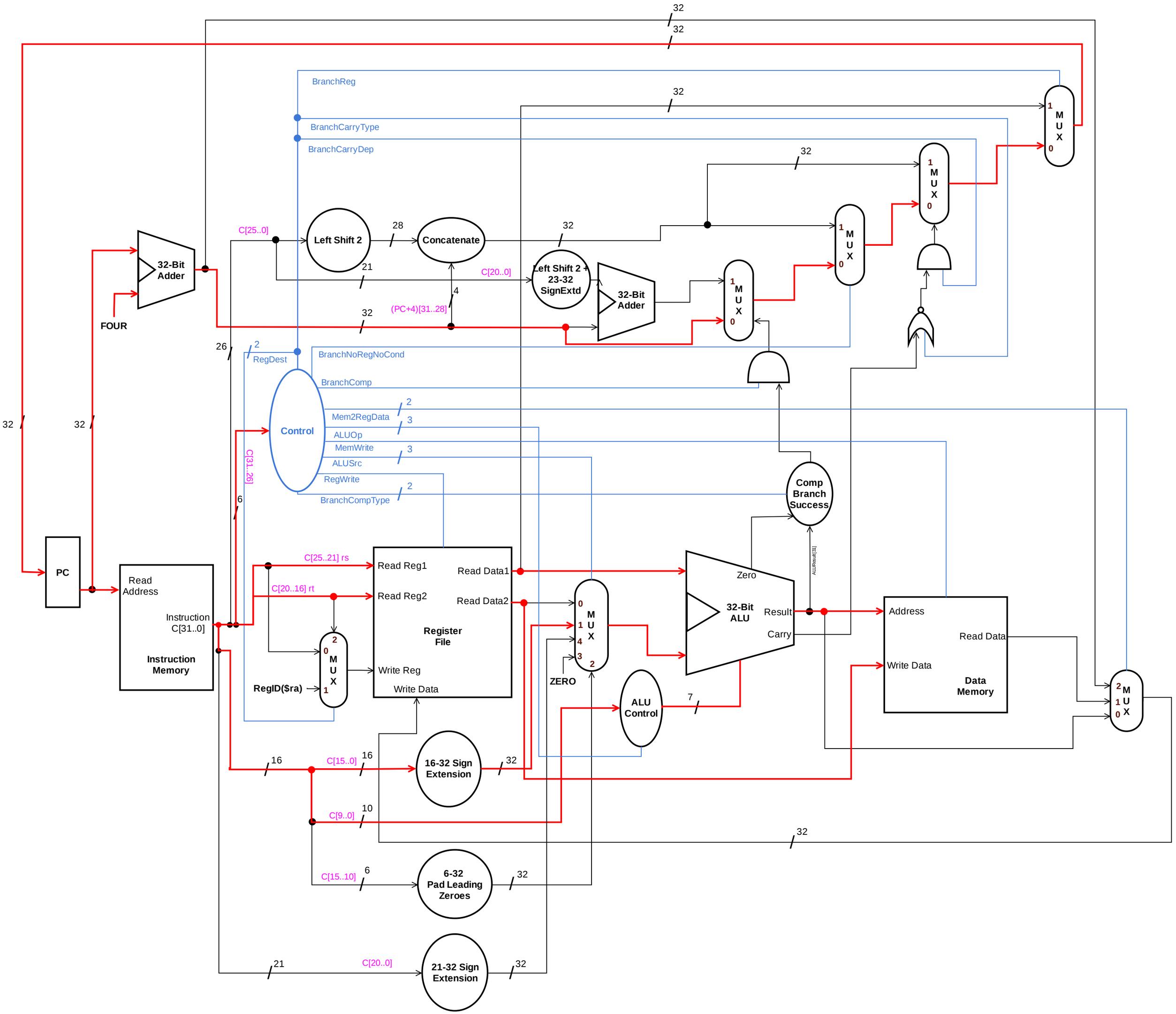
Datapath for Non-Memory Based I-Type Instructions
(*addi*, *compi*)



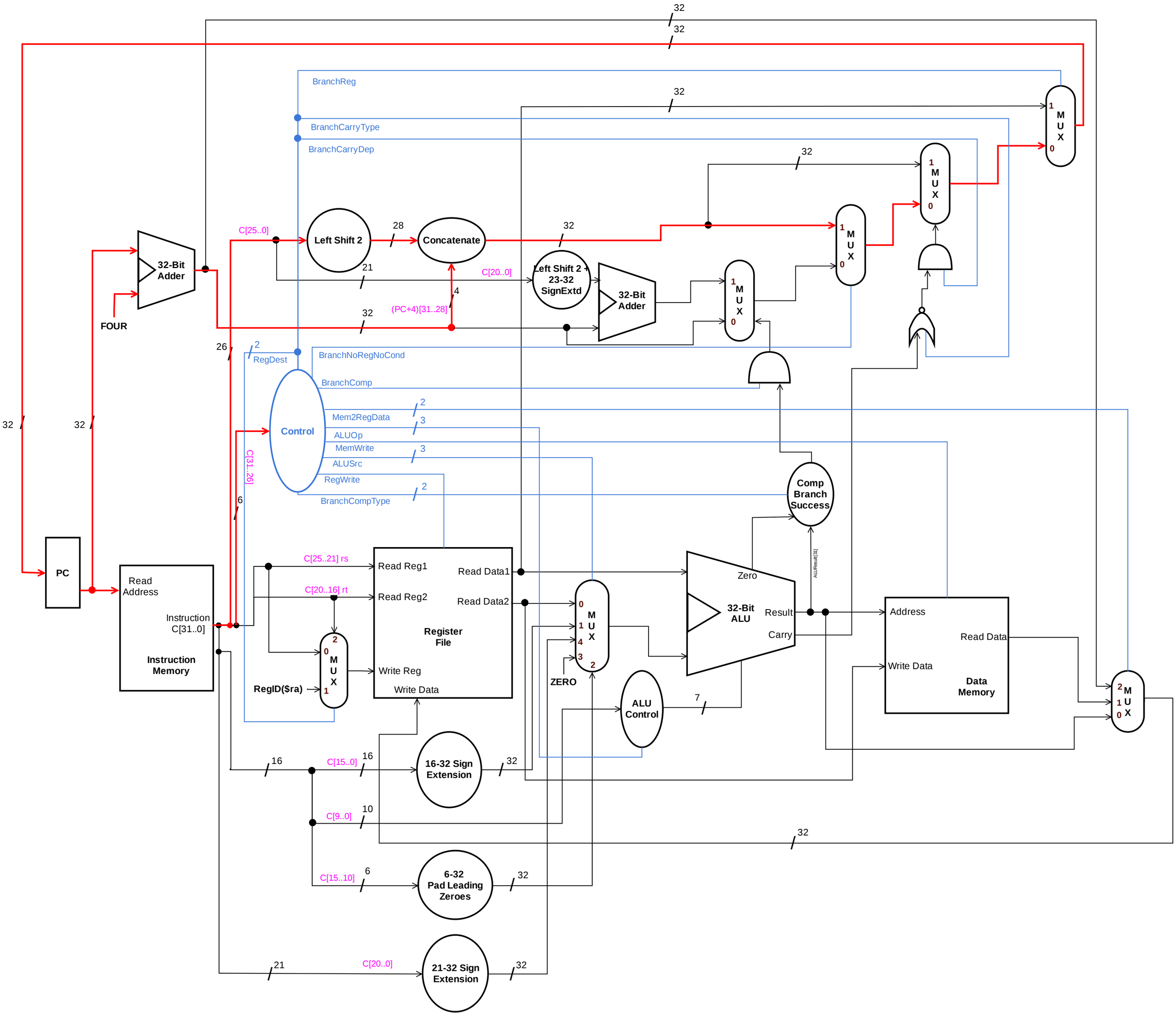
Datapath for Memory-Based I-Type Instructions
(*lw*)



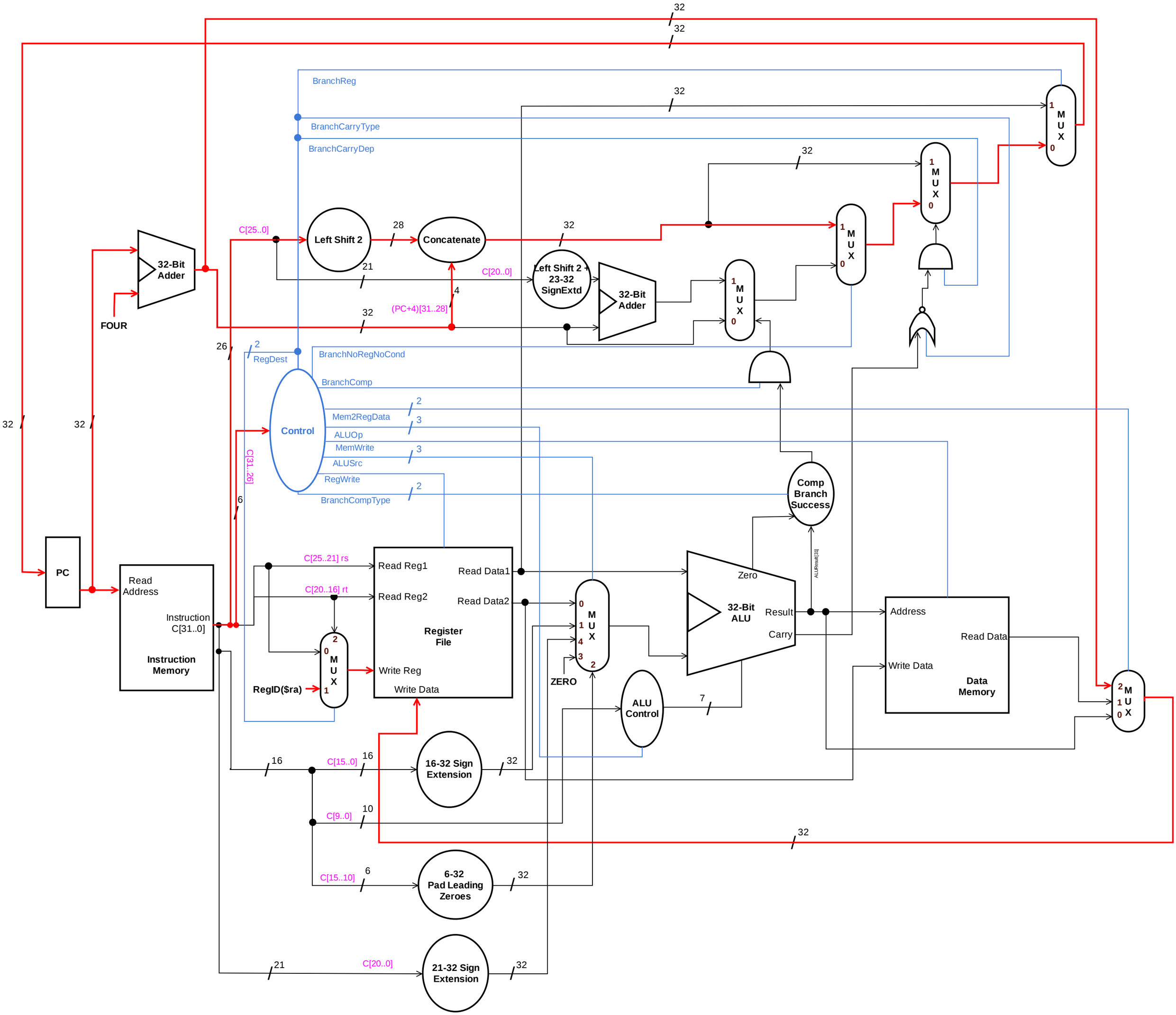
Datapath for Memory-Based I-Type Instructions
(*sw*)



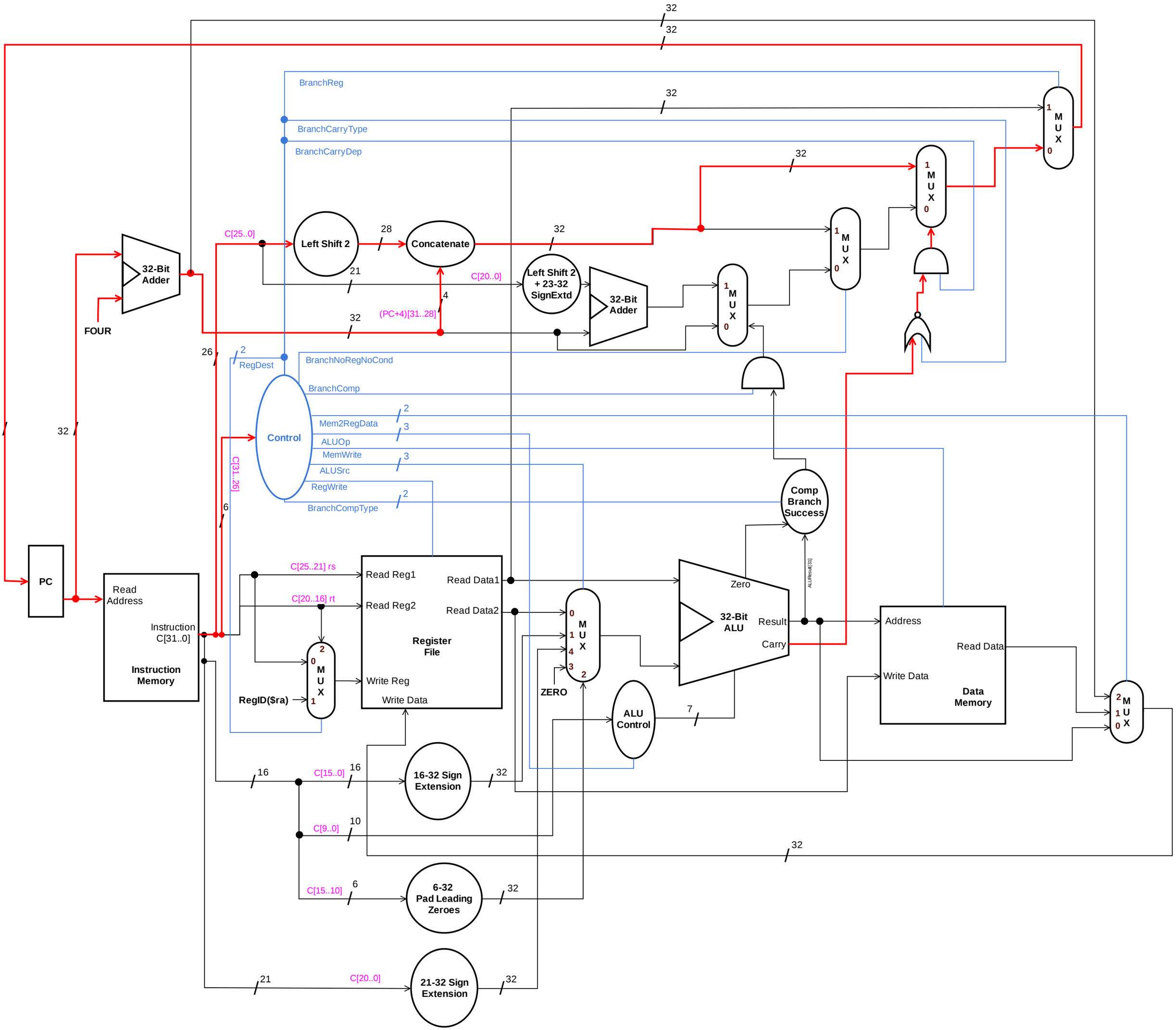
Datapath for Instructions in Pseudo-Direct Addressing Mode
(*b*)



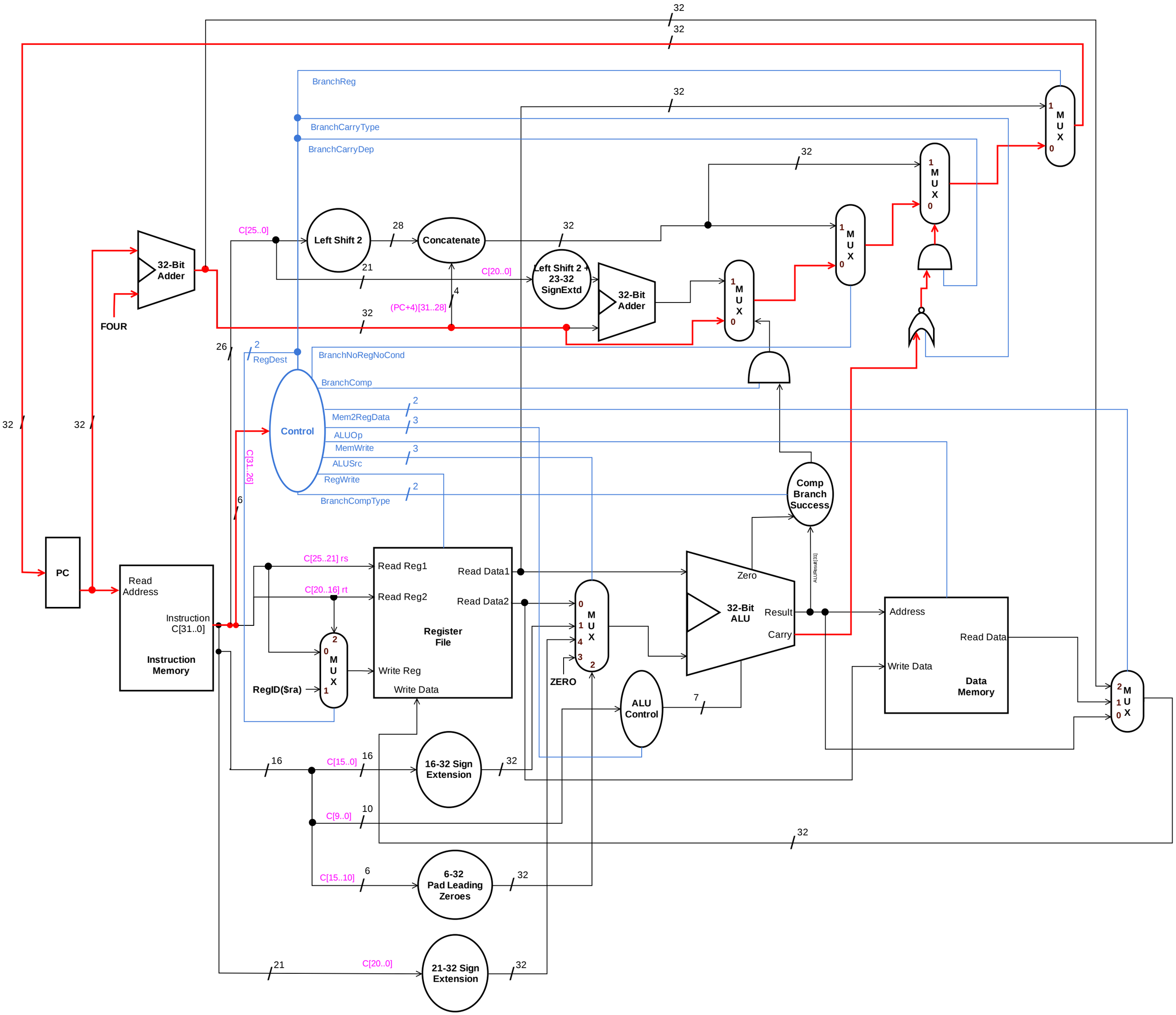
Datapath for Instructions in Pseudo-Direct Addressing Mode
(*bl*)



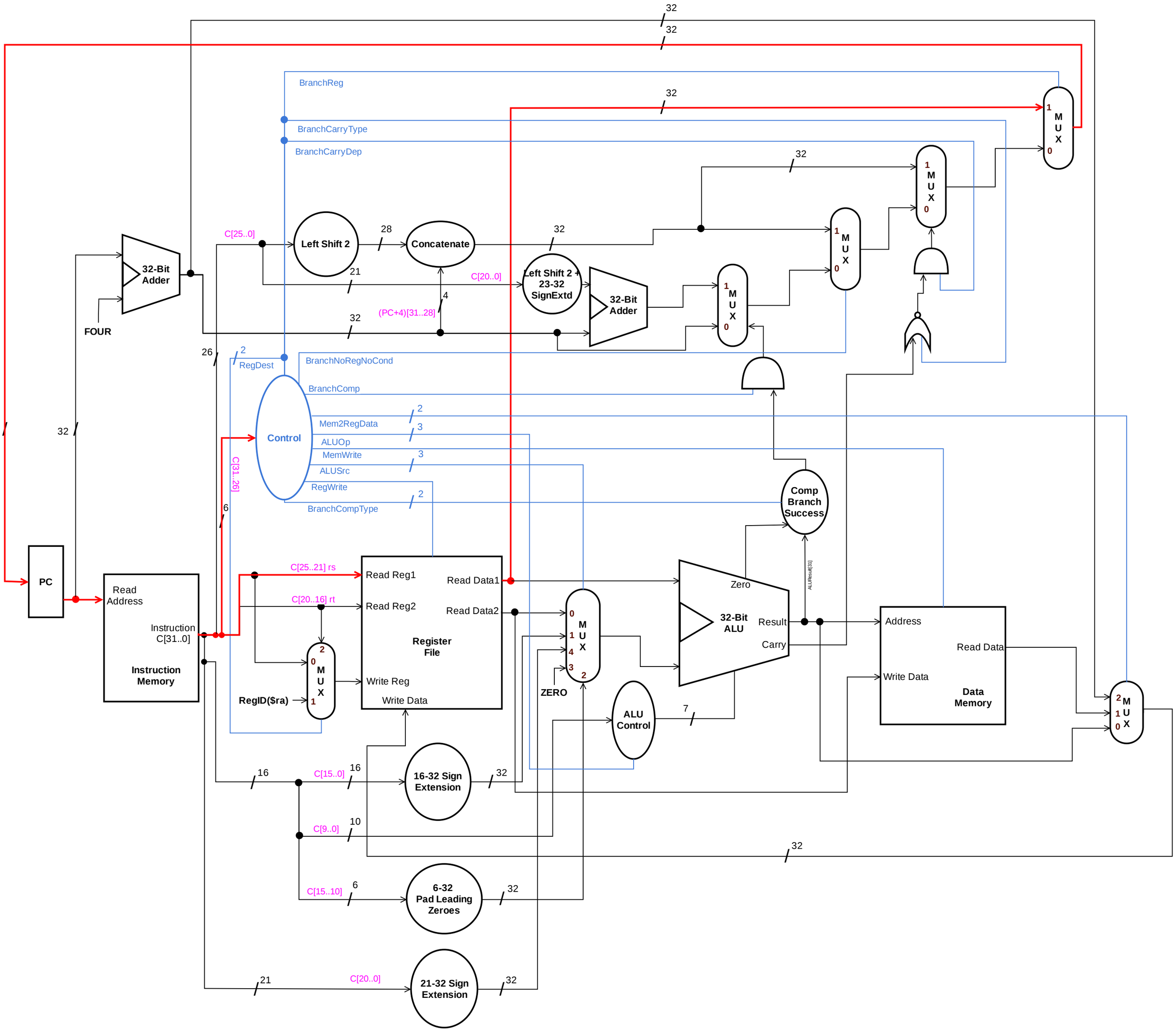
Datapath for Instructions in Pseudo-Direct Addressing Mode
(*bcy*, *bncy*) : Successful Branching



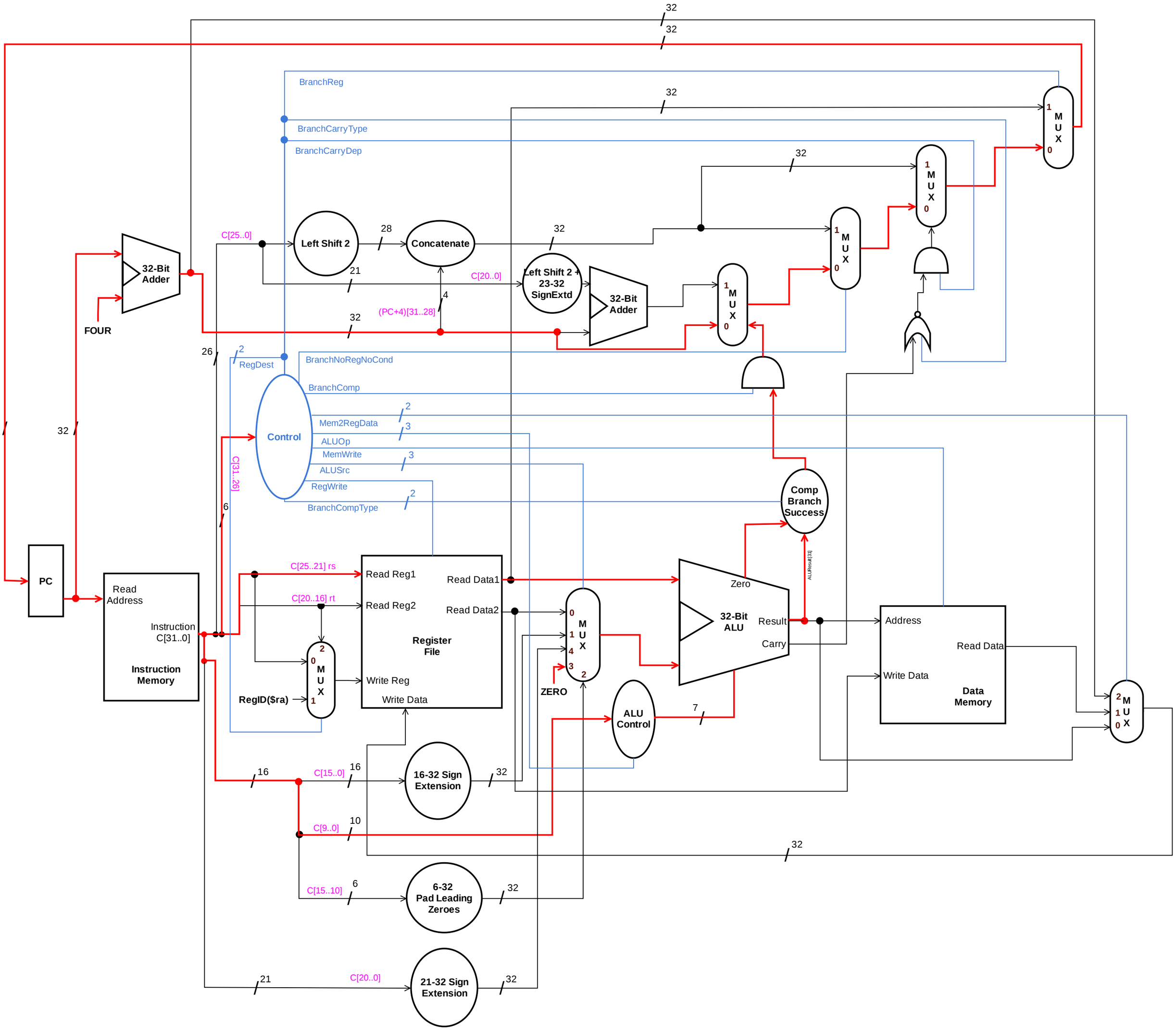
Datapath for Instructions in Pseudo-Direct Addressing Mode
(*bcy*, *bncy*) : Unsuccessful Branching



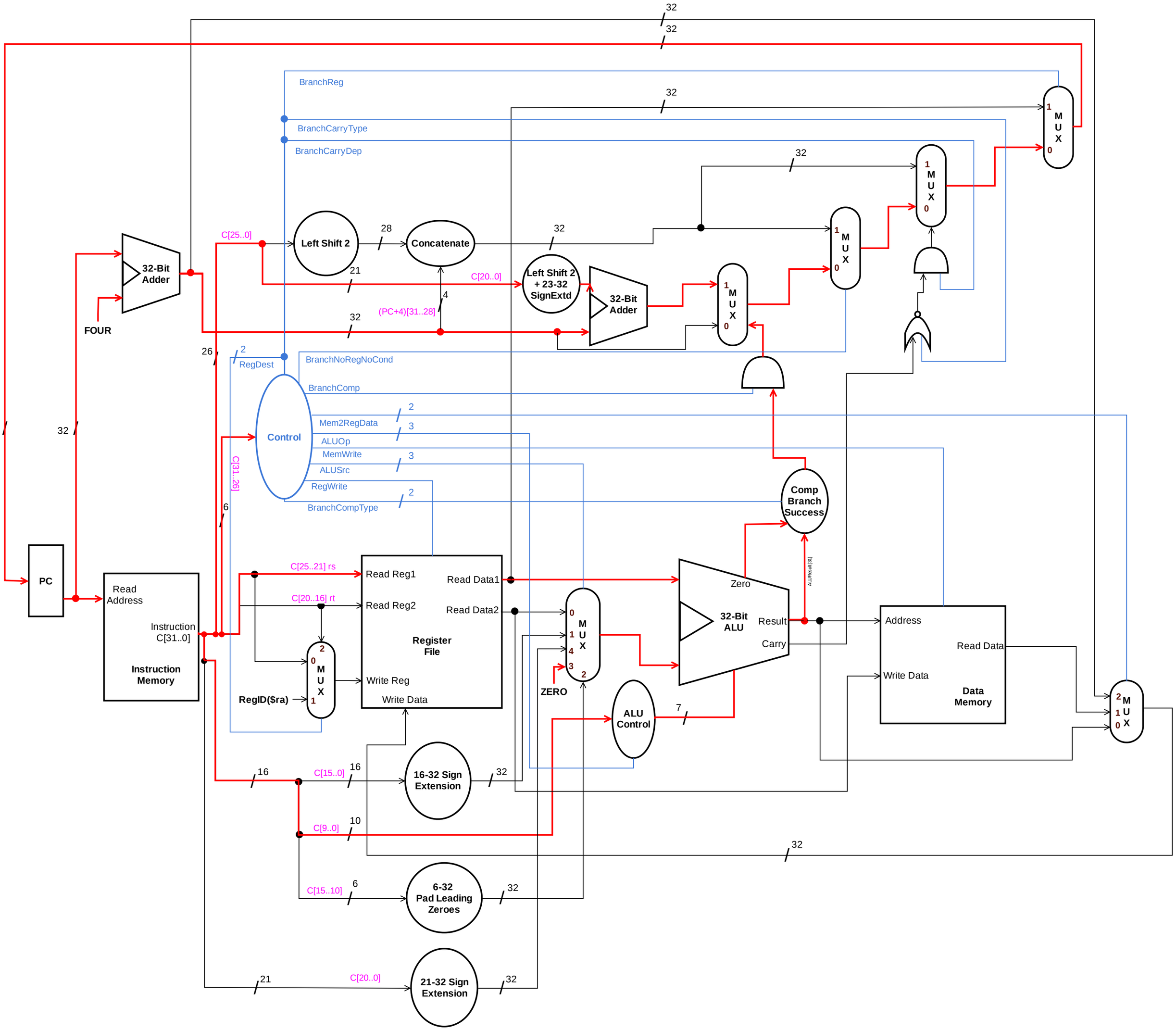
Datapath for Instructions in PC-Relative Addressing Mode
(*br*)



Datapath for Instructions in PC-Relative Addressing Mode (*bltz*, *bz*, *bnz*) : Unsuccessful Branching

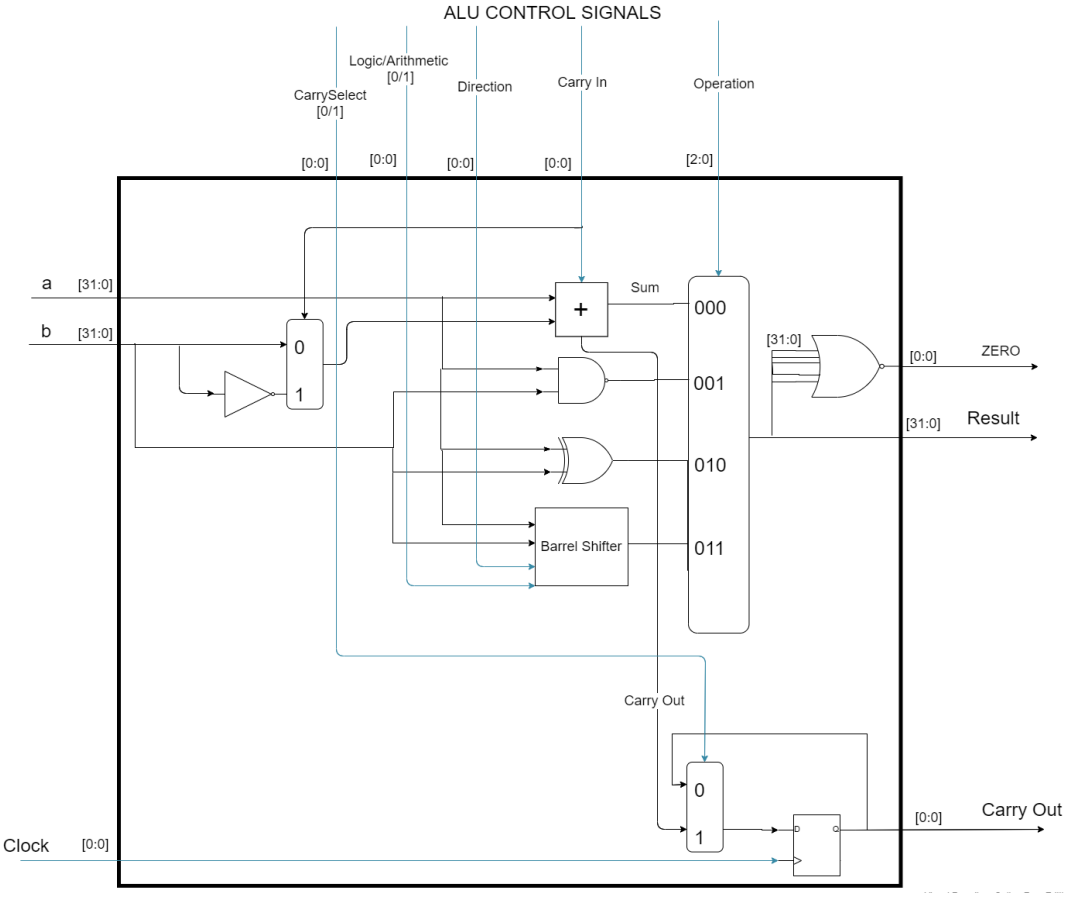


Datapath for Instructions in PC-Relative Addressing Mode
(*bltz*, *bz*, *bnz*) : Successful Branching



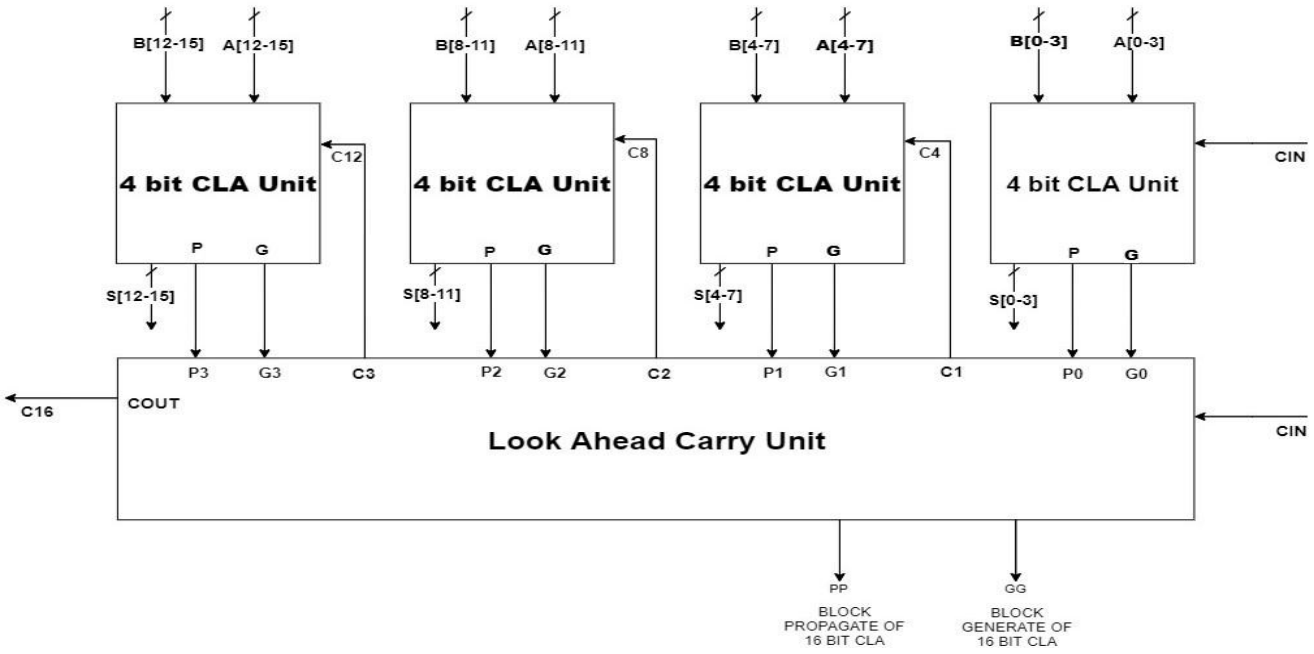
• Design of Arithmetic Logic Unit

Following is the design of the **32-bit ALU** (control lines are shown in blue color). The adder used is a ***32-bit Carry Look Ahead Adder*** and the combinational circuit used for implementing the shift instructions is a ***32-bit Barrel Shifter***. The *AND* and *XOR* instructions are trivial implementations of bitwise-AND and bitwise-XOR using an **array of AND and XOR gates**.



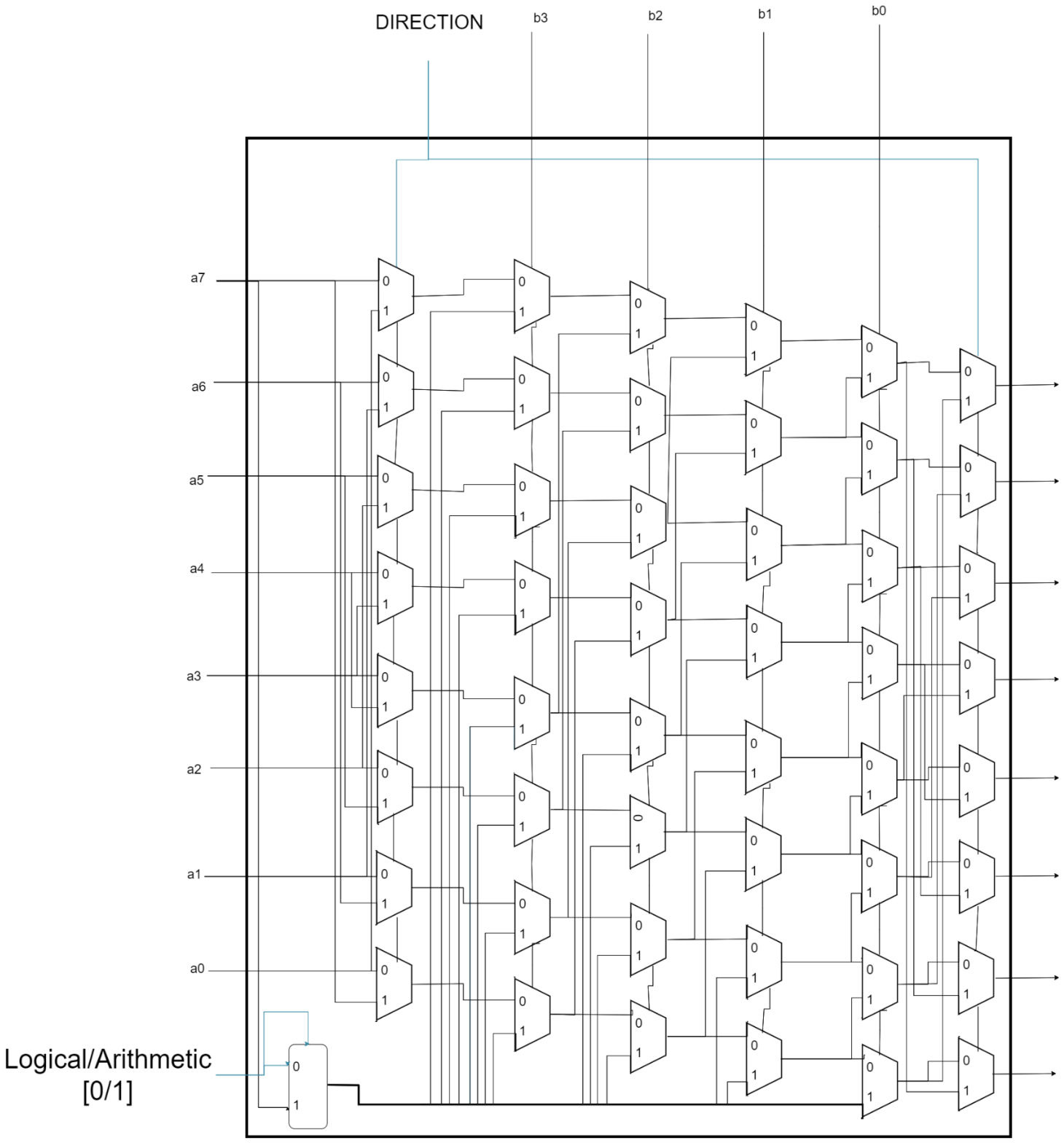
• Design of Addition Circuit

The ***addition*** **component of the ALU** is implemented as a *Carry Look-Ahead Adder (CLA)* with an augmented look-ahead carry unit. The adder shown below is a 16-bit Carry Look Ahead Adder, we are simply extending this implementation to build the 32-bit Carry Look Ahead Adder.



• Design of Shift Circuit

The ***shift* component of the ALU** is implemented as a *Barrel Shifter*. The combinational shifter shown below is an 8-bit Barrel Shifter with a 4-bit shift amount (*shamt*). In our design, we are simply extending this implementation to build the 32-bit Barrel Shifter with a 6-bit shift amount.



• Design of Control Logic

There are two control logic units in our processor. One is the ***Main Controller*** that controls the values of the selectors in all the multiplexers and enablers in all the modules. The other one is the ***ALU Controller*** that determines the function that the ALU module has to operate.

Both the control units are implemented as purely combinational circuits with primitive logic gates. The main controller signal has 13 output ports, emitting the control signals whose significance is already clear with the datapath diagrams. The ALU control has one 7-bit output that determines uniquely the ALU operation and some other determiners that are crucial for some specific operations (refer to the diagram of the ALU design).

Main control signals are purely a function of $opcode$ and the ALU control signals are a function of $ALUOp$ (a main control signal emitted by the Main Controller) and the $funct$ field for the R-Type instructions.

Main Controller

OPCODE	ALUOp	MemWrite	ALUSrc	RegWrite	BranchCompType	RegDest
000 000	001	0	000	1	XX	00
000 001	010	0	010	1	XX	00
000 010	011	0	000	1	XX	00
000 011	100	0	100	1	XX	00
000 100	101	0	100	1	XX	00
000 101	110	0	001	1	XX	10
000 110	110	1	001	0	XX	XX
000 111	000	0	XXX	0	XX	XX
001 000	000	0	XXX	1	XX	01
001 001	000	0	XXX	0	XX	XX
001 010	000	0	XXX	0	XX	XX
001 011	000	0	XXX	0	XX	XX
001 100	110	0	011	0	01	XX
001 101	110	0	011	0	00	XX
001 110	110	0	011	0	10	XX

OPCODE	BranchReg	BranchCarryType	BranchCarryDep	Branch NoRegNoCond	BranchComp	Mem2RegData
000 000	0	X	0	0	0	00
000 001	0	X	0	0	0	00
000 010	0	X	0	0	0	00
000 011	0	X	0	0	0	00
000 100	0	X	0	0	0	00
000 101	0	X	0	0	0	01

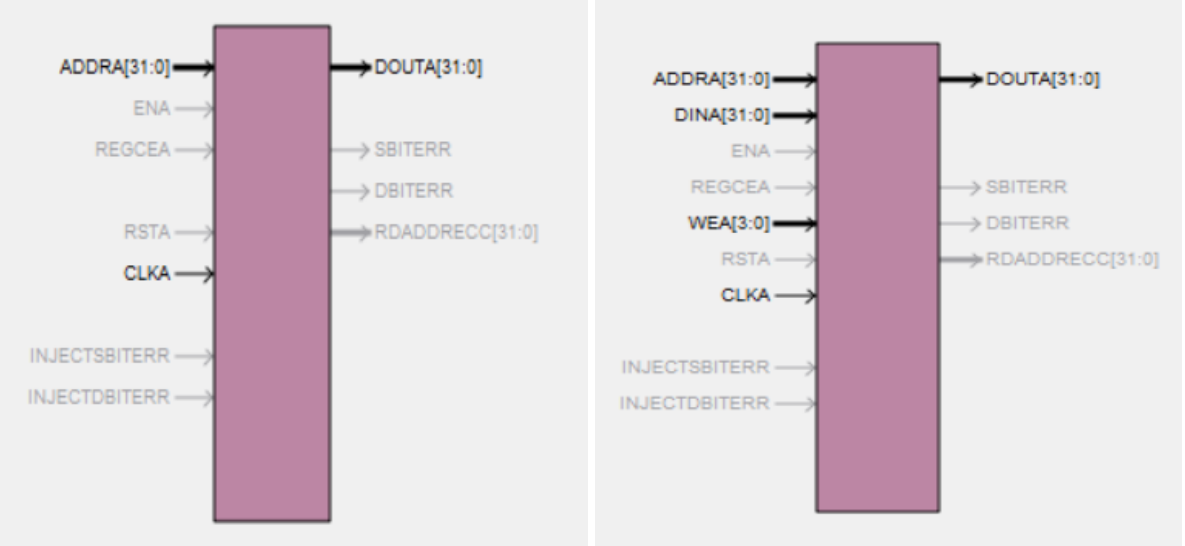
000 110	0	X	0	0	0	XX
000 111	0	X	0	1	0	XX
001 000	0	X	0	1	0	10
001 001	0	1	1	0	0	XX
001 010	0	0	1	0	0	XX
001 011	1	X	0	0	0	XX
001 100	0	X	0	0	1	XX
001 101	0	X	0	0	1	XX
001 110	0	X	0	0	1	XX

ALU Controller

Instruction	ALU Op	FUNCT	ALU Operation	ALU Operation Code
and	001	0000000000	Addition	1000000
add		0000000001	Logical AND	0000001
xor		0000000010	Logical XOR	0000010
comp		0000000011	2’s Complement	0001000
—		XXXXXXXXXX	—	0111111
shll	010	0000000000	Shift Left Logical	0010011
shrl		0000000001	Shift Right Logical	0000011
shra		0000000010	Shift Right Arithmetic	0100011
—		XXXXXXXXXX	—	0111111
shllv	011	0000000000	Shift Left Logical	0010011
shrlv		0000000001	Shift Right Logical	0000011
shrav		0000000010	Shift Right Arithmetic	0100011
—		XXXXXXXXXX	—	0111111
addi	100	XXXXXXXXXX	Addition	1000000
compi	101	XXXXXXXXXX	2’s Complement	0001000
bnz, bz, bltz, lw, sw	110	XXXXXXXXXX	Addition	0000000
b, bl, bcy, bncy	000	XXXXXXXXXX	—	0111111

• Design of Memory Modules

The processor consists of two memory modules — ***Instruction Memory*** and ***Data Memory***. Since, the processor must not be permitted to overwrite the instruction memory, it is implemented as a 32-bit enabled **Single Port ROM (Read-Only Memory)** (left hand side picture). The module takes as input a 32-bit address and emits the 32-bit binary instruction stored at that address.

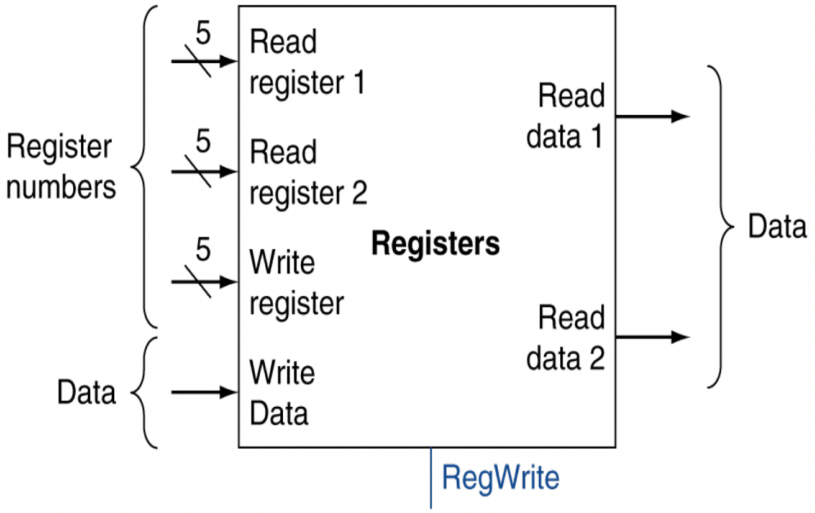


The processor must have both read and write access to the data memory and hence it is implemented as a 32-bit enabled **Single Port RAM (Random-Access Memory)** (right hand side picture). Note that though the processor has both read and write access to the data memory, this access is moderated by the control signals to prevent unwanted write operations on the data memory. Please refer to the truth-table to learn more about the instructions that allow the respective access.

For reading, the module takes as input a 32-bit address and emits the 32-bit data that is stored at that address in the memory. For writing, the write enabling port of the module should be active. It takes as input a 32-bit address and 32-bit data and writes the data to that address in the memory.

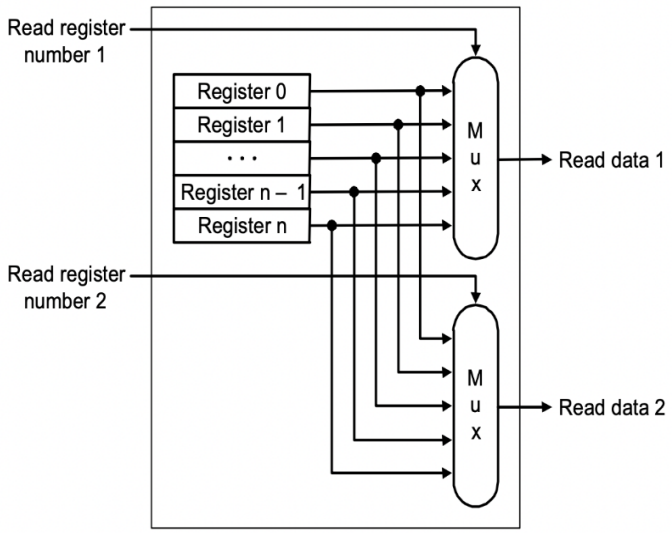
• Design of Register File

This is another very important module in our processor. The ***32x32 register file*** is implemented in a hierarchical fashion using DFFs that are cascaded to build 32-bit registers that are further cascaded to build a file of 32 32-bit registers.

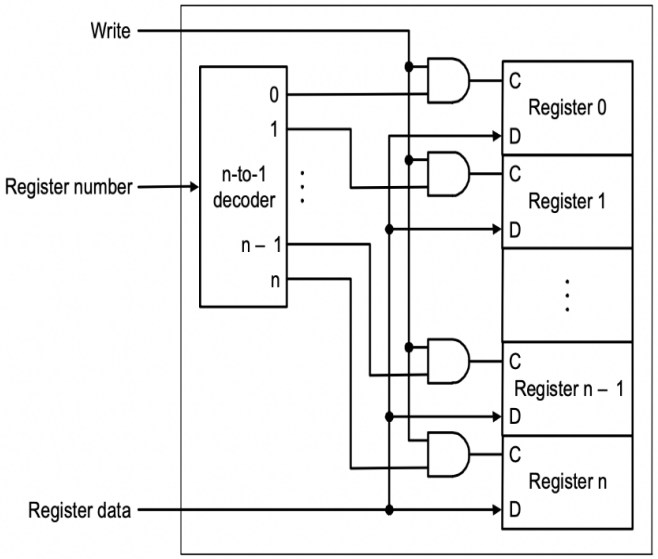


The register file uses multiplexors to decode the 5-bit register ID to select exactly one of the 32 registers for read or write operation. There are in-fact 3 multiplexers, with bus width of 32, one for enabling writing to the destination register (rd), and one for reading from each of the two source registers (rs and rt).

*Register File while **Reading***



*Register File while **Writing***



The 32 registers in the register file are as follows.

Register ID	Register Name	Purpose
0	$0 / $zero	Stores constant value 0
1 – 30	$1 – $30	General purpose registers / temporaries
31	$31 / $ra	Stores the linked return address of *bl* instruction

Unlike in MIPS32, we have not made any special distinction between callee-saved registers ($s<i>), argument registers ($a<i>) and temporaries ($t<i>). All registers can be used for all purposes, except two special registers **$zero** and **$ra**.
$zero stores a constant value of 0. Therefore the register-write-enabled instructions with $zero as the destination register (rd) may not give the desired result.

```
# Value of $0 – 0
addi $0,  100
# Value of $0 – 0
# Mem[4] – 123
lw $0, 4($0)
# Value of $0 – 0
```

Though these instructions are logically illegal because the destination register is read-only, since we have not developed any interrupt or exception mechanism in our custom ISA, the programmer might be unaware of this bug.

CAUTION: **The programmer should never use $zero as the destination register in register-write-enabled instructions**.

Similarly the use of $ra should also be restricted to bl and br instructions only. If $ra is overwritten explicitly (say `addi $31, 100`) and as opposed to only internally by the bl instruction, the function call mechanism in the code might get disrupted, and that might become extremely difficult to debug. But if the programmer still has to jump to some custom address using a br but not the bl instruction, he can use any general purpose register ($1 – $30), copy the value of $ra to that and can do whatever with that register.

```
# Value of $1 – 0
add $1, $31
addi $1, 1200     # addi $31, 1200 X
br $1             # br $31 X
```

CAUTION: **The programmer should never use $ra as a destination register in register-write-enabled instructions**.