Lecture 9/10

SFU

A SINGLE-CYCLE PROCESSOR
PART 3

Instruction Classes

SFU

There are 59 instructions in rv64I.

I have ignored 8 of these instructions. { fence, fenci.i } (thread synchronization) &

{ csrrs, csrrsi, csrrw, csrrwi, csrrc, csrrci } (control/status register - read/write, read/set, read/clear)

We will design a processor that can execute the 51 remaining instructions.

	R-Format	I-Format	S-Format	SB-Format	U-Format	UJ-Format
Flow Control (10)		ebreak, ecall, jalr		bne, beq, blt, bge, bltu, bgeu		jal
Execution (30)	add, addw, sub, subw	addi, addiw			lui, auipc	
	and, or, xor	andi, ori, xori				
	slt, sltu	slti, sltiu				
	sll, srl, sra, sllw, srlw, sraw	slli, srli, srai, slliw, srliw, sraiw				
Memory (11)		lb, lh, lw, ld, lbu, lhu, lwu	sd, sw, sh, sb			

Execution: (**30**)

- Logic: (6) → 3R-format + 3I-format
- Arithmetic: (12) → 6R-format + 4I-format +2U-format
- Shift: (12) → 6R-format + 6I-format

Flow Control: (**10**)

- Branch: (6) → 6SB-format
- Jump: (2) → 2UJ-format
- Environment: (2) → 2I-format

Memory: (**11**)

- Load: (7) → 7I-format
- Store: (4) → 4S-format

The (51) Instructions

SFU

	funct6	funct3	opcode
sllw	0000000	001	0111011
srlw	0000000	101	0111011
sraw	0100000	101	0111011
slliw	0000000	001	0011011
srliw	0000000	101	0011011
sraiw	0100000	101	0011011

	funct6	funct3	opcode
sll	0000000	001	0110011
srl	0000000	101	0110011
sra	0100000	101	0110011
slli	0000000	001	0010011
srli	0000000	101	0010011
srai	0100000	101	0010011

	funct7	funct3	opcode
xor	0000000	100	0110011
or	0000000	110	0110011
and	0000000	111	0110011
xori	-	000	0010011
ori	-	010	0010011
andi	-	011	0010011

	funct7	funct3	opcode
add	0000000	000	0110011
addw	0000000	000	0111011
sub	0100000	000	0110011
subw	0100000	000	0111011
slt	0000000	010	0110011
sltu	0000000	011	0110011

	funct7	funct3	opcode
addi	-	000	0010011
addiw	-	000	0011011
slti	-	010	0010011
sltiu	-	011	0011011
lui	-	-	0110111
auipc	-	-	0010111

	funct7	funct3	opcode
lb	-	000	0000011
lh	-	001	0000011
lw	-	010	0000011
ld	-	011	0000011
lbu	-	100	0000011
lhu	-	101	0000011
lwu	-	110	0000011

	funct7	funct3	opcode
sb	-	000	0100011
sh	-	001	0100011
sw	-	010	0100011
sd	-	011	0100011

	funct7	funct3	opcode
beq	-	000	1100011
bne	-	001	1100011
blt	-	100	1100011
bge	-	101	1100011
bltu	-	110	1100011
bgeu	-	111	1100011

ebreak	imm=0x000	000	1110011
ecall	imm=0x001	000	1110011

jal	-	-	1101111
jalr	-	000	1100111

The Primary Elements of a Processor Datapath

SFU

So far we have the two most important elements of a Processor Datapath.

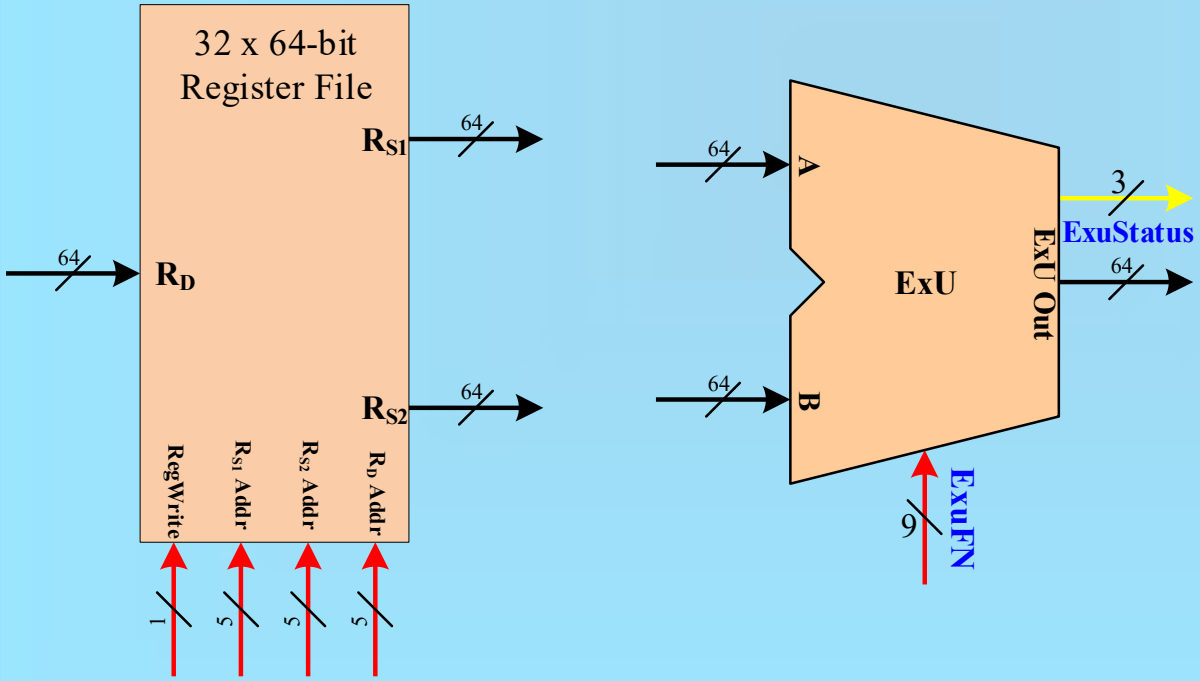
The Execution Unit (**ExU**) and the Register File (**RegF**)

There are 9-bits that control the behaviour of the execution unit, FuncClass, LogicFN, ShiftFN, ExtWord, AddnSub and NotA. (We don't actually need NotA)

Let us define a 9-bit signal,


```
ExuFN <= NotA & FuncClass & LogicFN & ShiftFN & AddnSub & ExtWord;
```

```
ExuStatus <= Cout,& Ovfl & Zero;
```



FuncClass		operation
0	0	sltu
0	1	slt
1	0	shift/arith
1	1	logic

LogicFn		operation
0	0	Lui
0	1	A xor B
1	0	A or B
1	1	A and B

ShiftFN		operation
0	0	arith
0	1	sll
1	0	srl
1	1	sra

Simplified Datapath

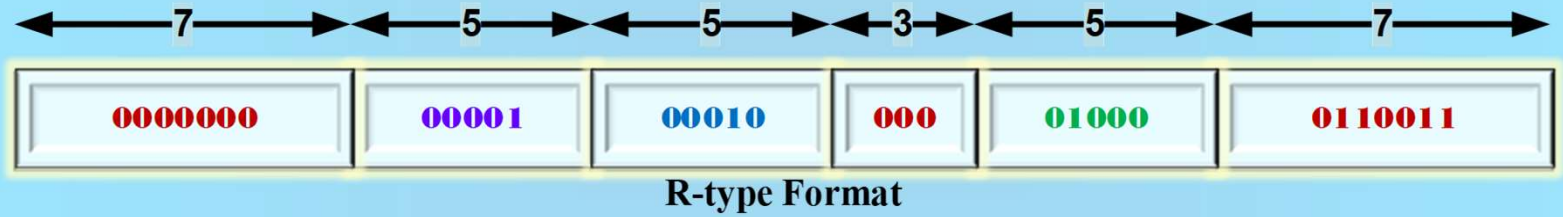
SFU

High-Level Language Statement **a = b + c;**

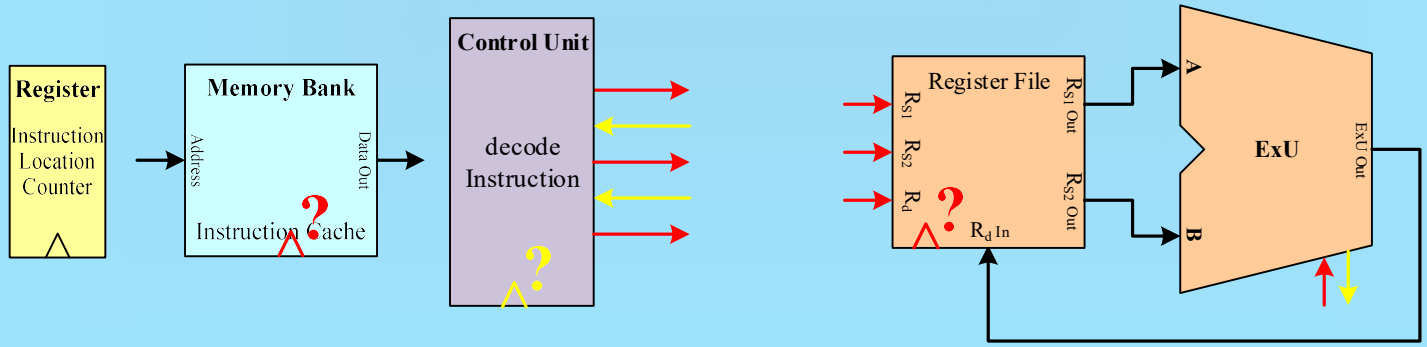
Assembly Language Instruction **add x8, x2, x1**

Machine Language Instruction **0000000 00001 00010 000 01000 0110011**

0x 0011 0433



R-type Format



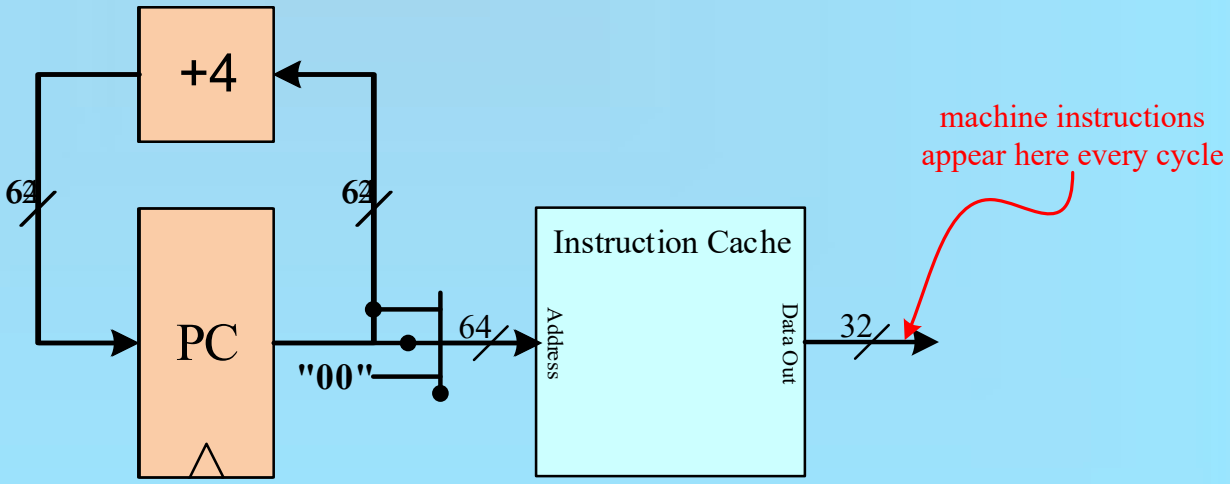
The Instruction Cycle

SFU

The Datapath contains a special Register called the **Program Counter, PC**.

- This register holds a 64-bit value that represents a memory address of a location in the Instruction Memory Unit. (The **Lowest bit** is always **'0'** because the RISC-V specification requires that Instructions must be **16-bit halfword-aligned**)
- Instructions are only read from the Instruction Memory Unit and thus the behaviour is similar to that of a combinational circuit. (The clocking scheme is not an issue.)

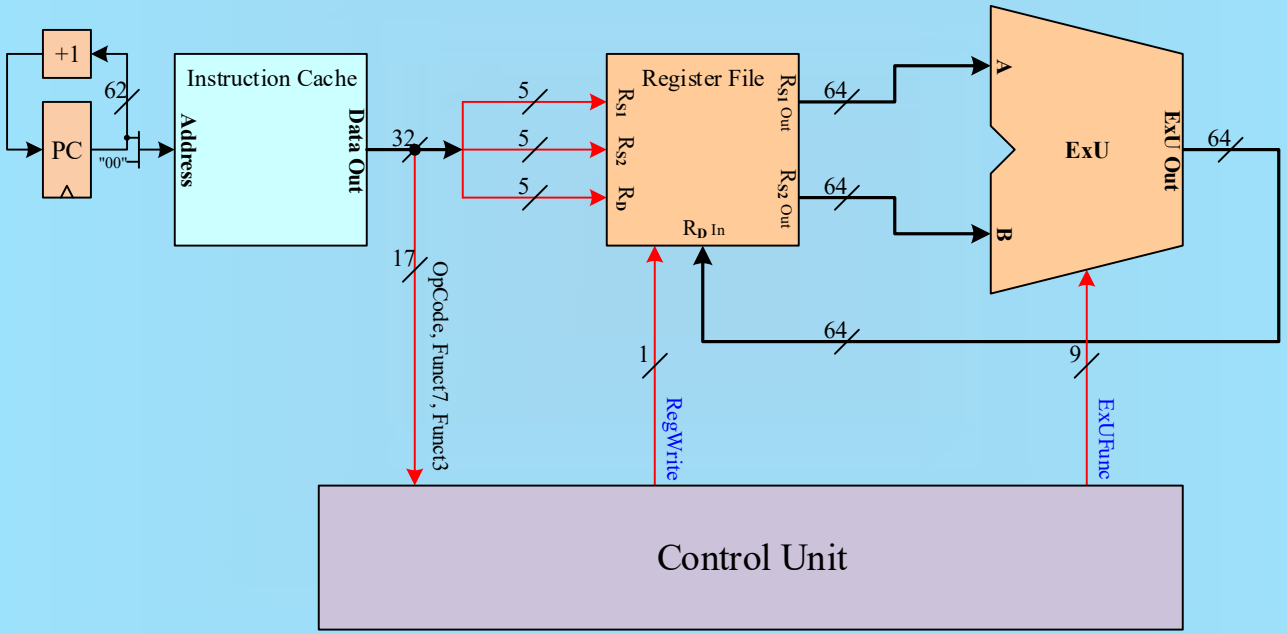
The value inside the PC is used to calculate a new value for the PC each cycle. The new value represents the location of the instruction to be processed in the next cycle.
The new value will be PC ← PC + 4, for all instructions except Flow Control Instructions



R-Format "Execute" Instructions

Execute Instructions: (30 instructions)

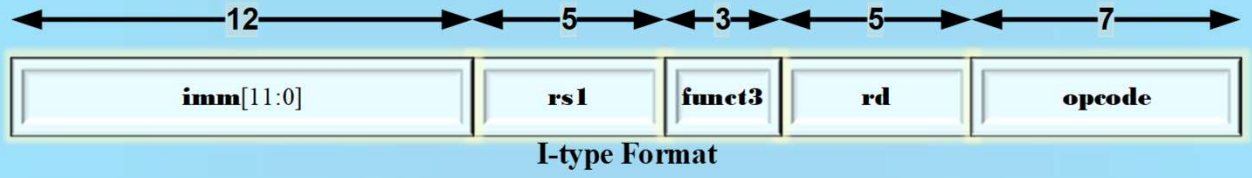
- Pass operand data through an execution unit and store a result back into a destination register.
- 15 of these instructions are **R-Format**; with operands coming from two source registers.



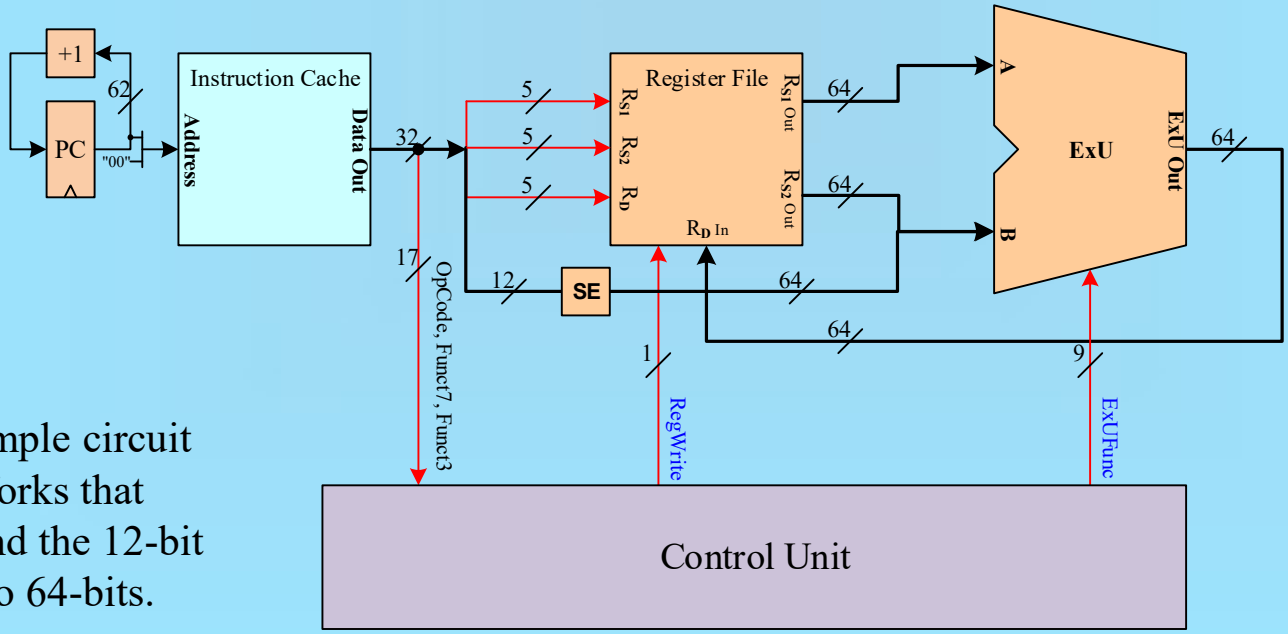
I-Format "Execute" Instructions

Execute Instructions: (30 instructions)

- Pass operand data through an execution unit and store a result back into a destination register.
- 13 of these instructions are **I-Format**; with operands coming from one source register , and a 12-bit sign-extended immediate constant.

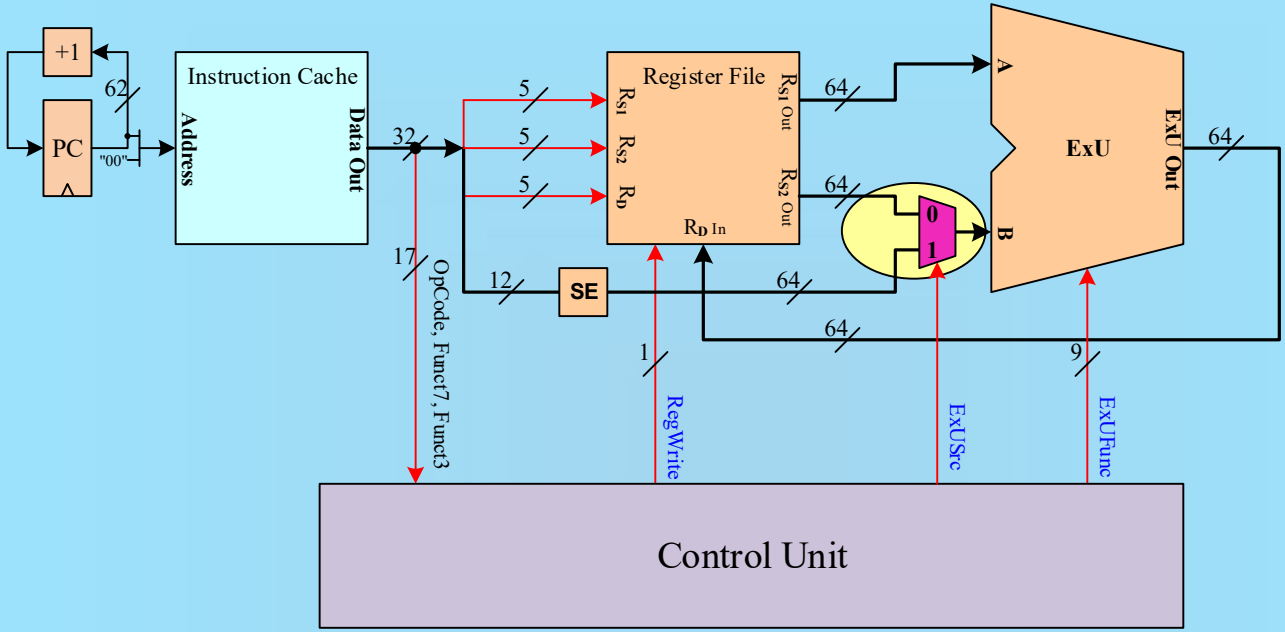


I-type Format



SE is a simple circuit contains forks that sign-extend the 12-bit constant to 64-bits.

Execution Unit Source MUX



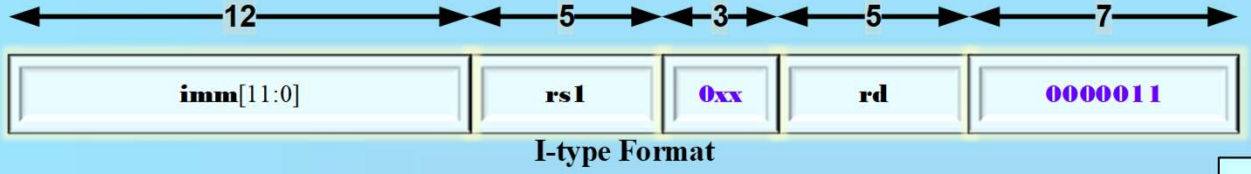
We have now introduced one more control signal, ExuSrc, into the design so that the datapath can execute both **R format** and **I format** instructions.

I-Format Data Memory "Read" Access

SFU

Memory Instructions: (7 load and 4 store instructions)

- Load instructions are **I-Format** instructions. **ld** $R_D \leftarrow mem(R_{S1} + imm)$

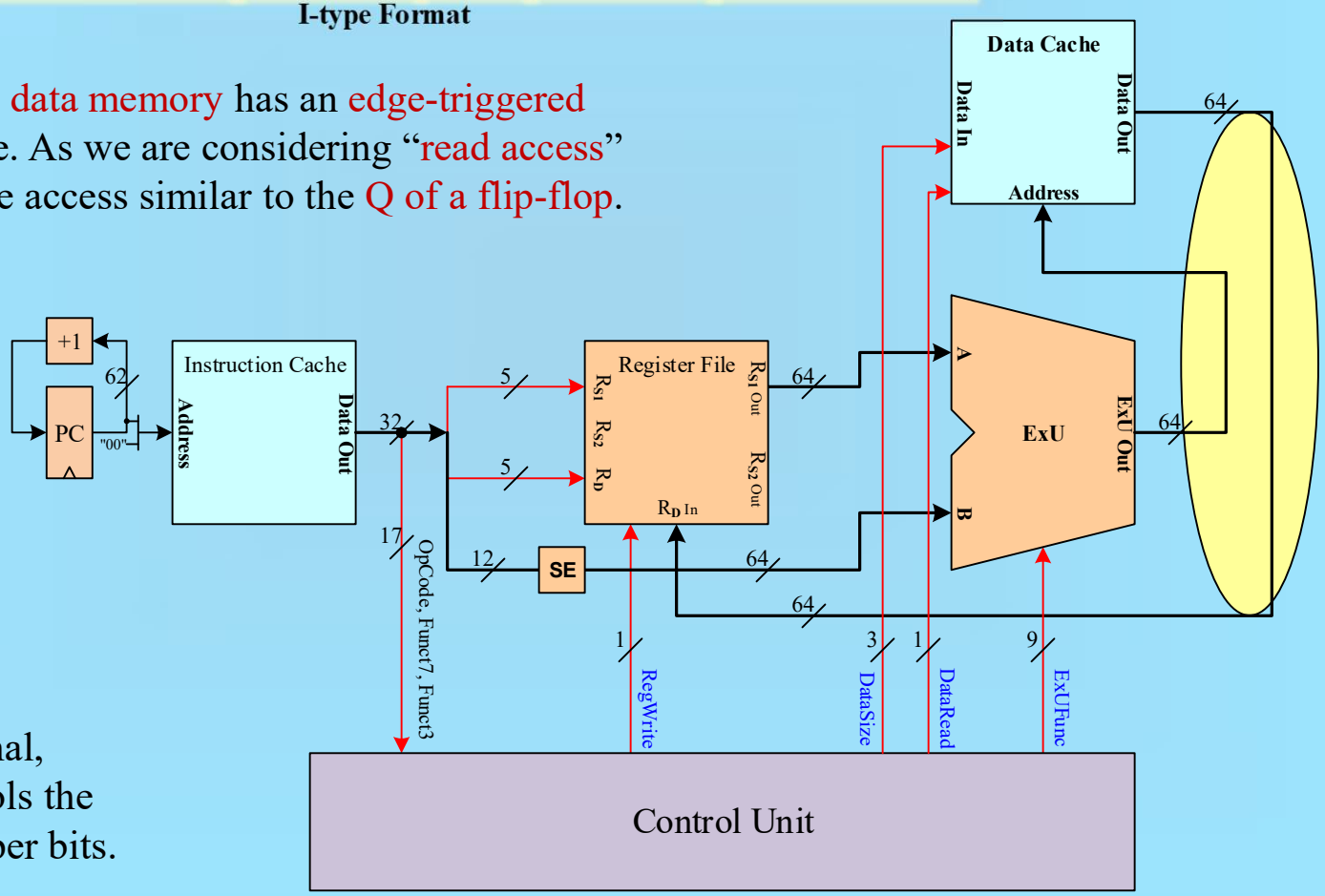


I-type Format

Assume that the data memory has an edge-triggered clocking scheme. As we are considering "read access" we may view the access similar to the Q of a flip-flop.

DataSize			Read Op
0	0	0	Ubyte
0	0	1	Uhalf
0	1	0	Uword
0	1	1	double
1	0	0	Sbyte
1	0	1	Shalf
1	1	0	Sword
1	1	1	double

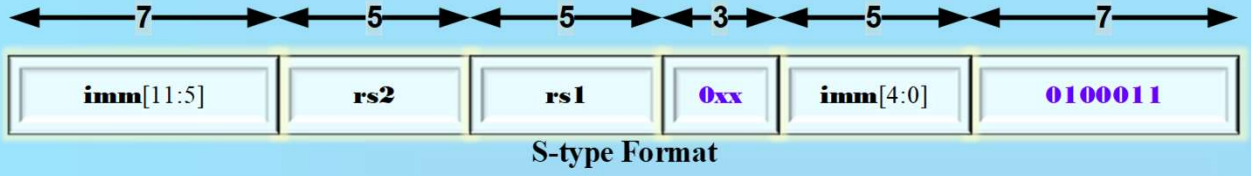
A new 3-bit signal, DataSize, controls the extension of upper bits.



S-Format Data Memory "Write" Access

Memory Instructions: (7 load and 4 store instructions)

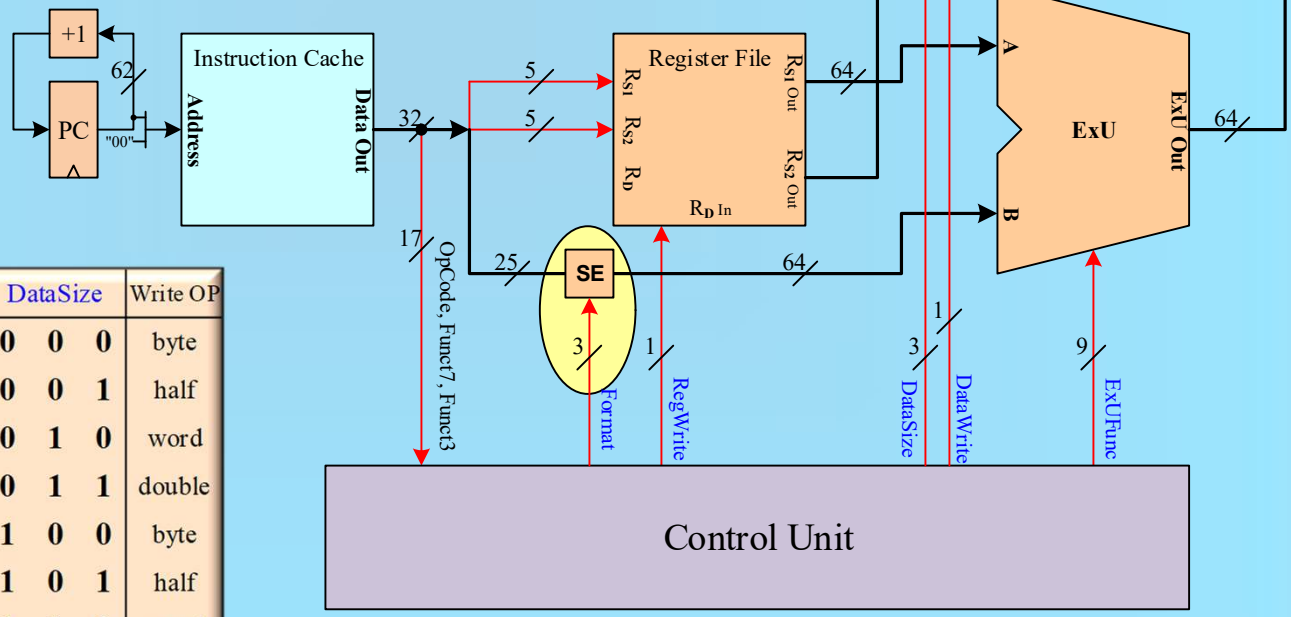
- Store instructions are **S-Format** instructions. **sd** mem(R_{S1} + imm) ← R_{S2}



S-type Format

A new 3-bit signal, Format, controls formation of the immediate constant .

Format			operation
0	0	0	R-type
0	0	1	I-type
0	1	0	S-type
0	1	1	SB-type
1	0	0	U-type
1	0	1	UJ-type
1	1	0	xx
1	1	1	xx

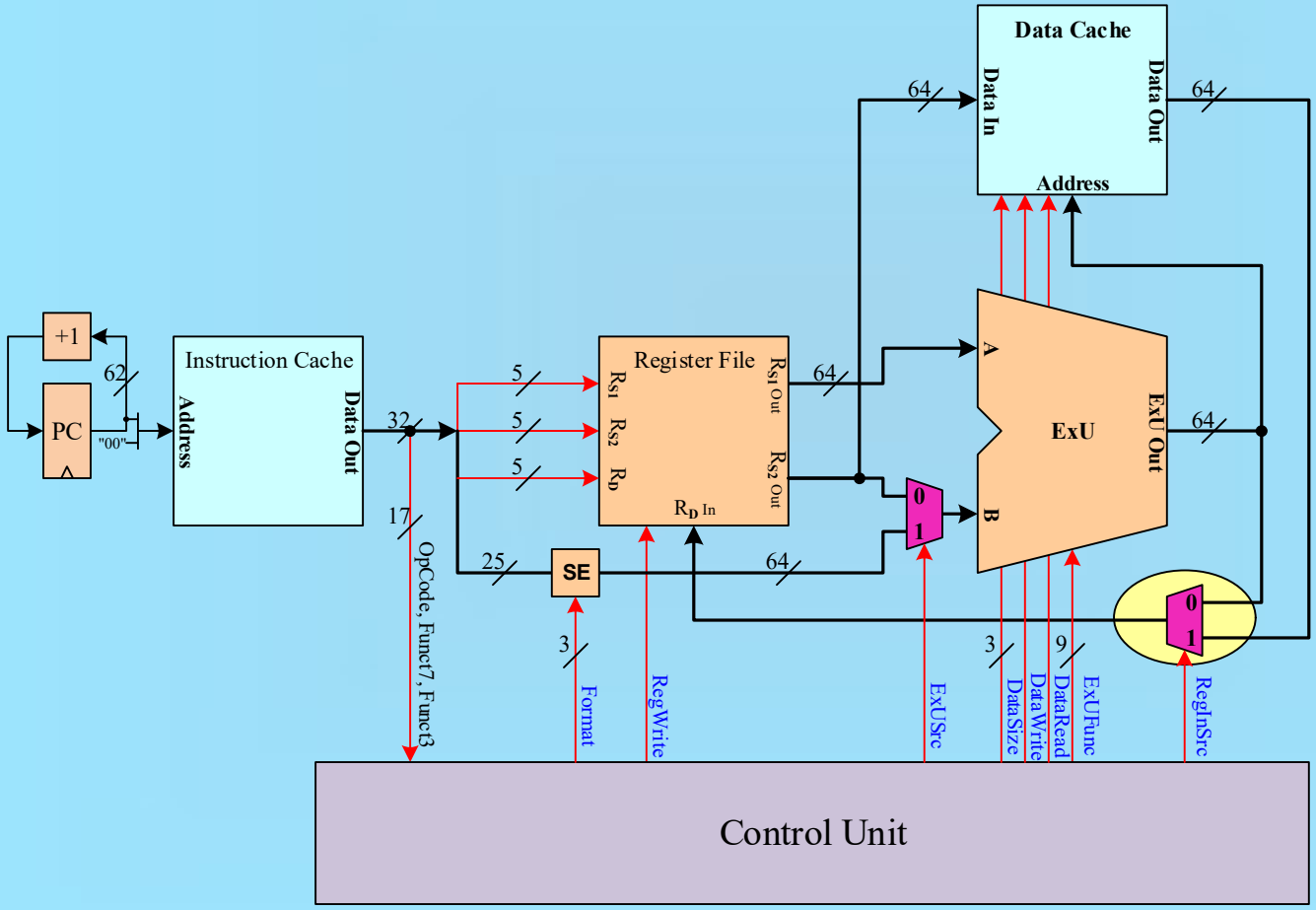


DataSize			Write OP
0	0	0	byte
0	0	1	half
0	1	0	word
0	1	1	double
1	0	0	byte
1	0	1	half
1	1	0	word
1	1	1	double

Assume that the Data Memory Unit ignores the **msb** of DataSize for data write operations.

Adding The Destination Register MUX

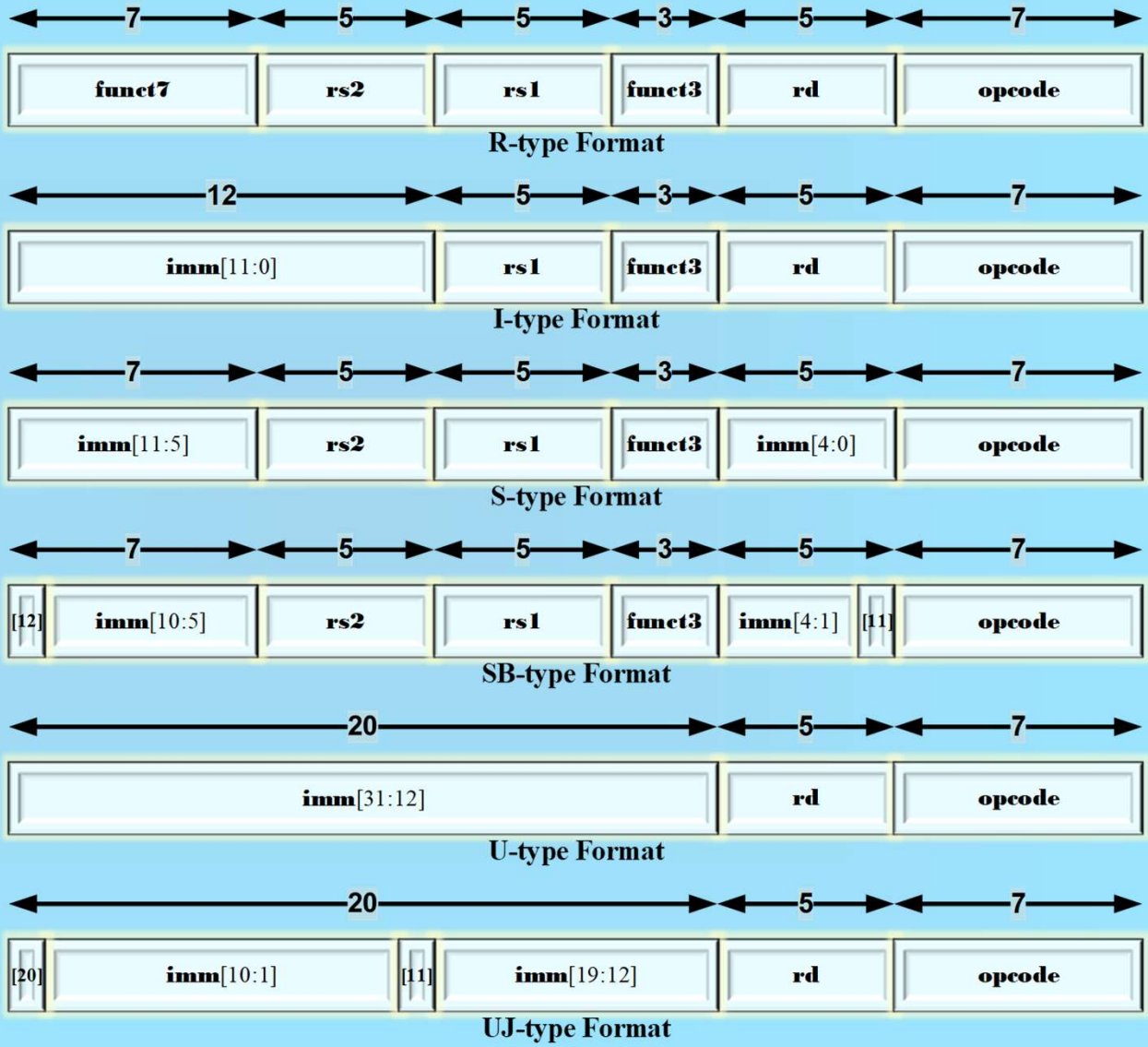
The Data Memory Operations can be combined with the Execution Operations by inserting a MUX that selects the source for storing to the Register File.
This MUX is controlled by a control signal called RegInSrc.



The RV64I Instruction Formats

SFU

Of the 6 instruction - formats, 5 of them contain immediate constants. The bits of the immediate constant must be unpacked to produce an appropriate 64-bit double-word.



Flow Control Instructions

SFU

So Far we have 28/30 of the Execution Instructions and all 11 of the Memory Instructions.
The Flow Control Instructions are comprised of 2 (unconditional) Jumps and 6 conditional Branches.

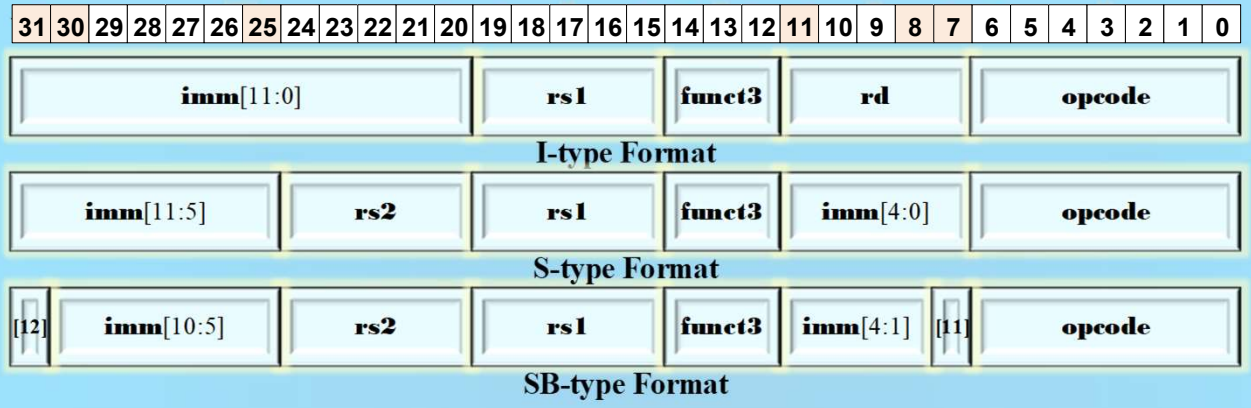
Instruction	Operation	Format	Application
jal	PC ← PC+(2*imm20) R_D ← PC+4	UJ	Must always use for unconditional branches. (To avoid polluting Branch Prediction Tables) To implement subroutine calls.
jalr	PC ← ((R_{S1}+imm12)&(~1)) R_D ← PC+4	I	Used to implement Branch Tables. (Case/Switch Statements) Used to perform return from subroutines. The lsb of the function pointer may be used as a flag.
beq, (**bne**)	IF (R_{S1} = R_{S2}) PC ← PC+(2*imm12)	SB	Used for IF-Then-Else type short branching. Used for loops, either at the top or the bottom. Relative distance to target instruction $\pm 2^{12}$.
blt, (**bge**)	IF (R_{S1} < R_{S2}) PC ← PC+(2*imm12)	SB	
bltu, (**bgeu**)	IF (R_{S1} < R_{S2}) PC ← PC+(2*imm12)	SB	

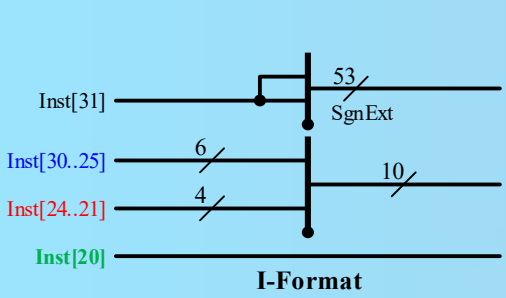
All **branch** and **jal** instructions use PC-relative addressing to support position-independent code.
The address of the target instruction is formed by appending a '0' to the (12-bit/20-bit) immediate, sign-extending to 64-bits and then adding to the PC.
(containing the address of the branch instruction)

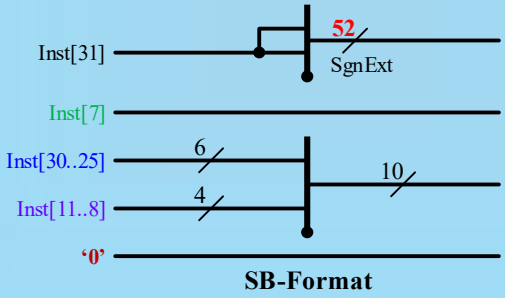
Absolute jumps with an address range of 2^{32} are implemented using **lui** and then **jalr**.
Relative jumps with an address range of 2^{32} are implemented using **auipc** and then **jalr**.

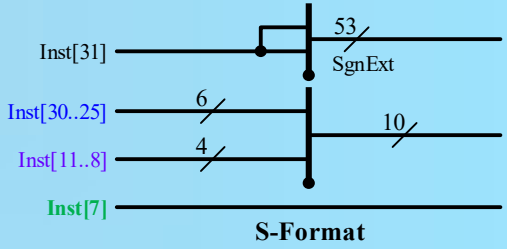
Unpacking the Immediate Constant

The Instruction Formats with immediate constants have been chosen to overlap as much as possible. The circuit that unpacks and sign-extends the immediate is quite straight forward.



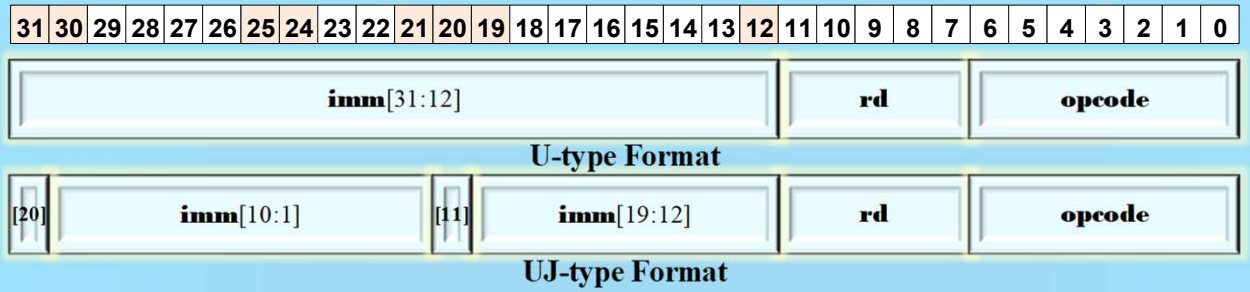


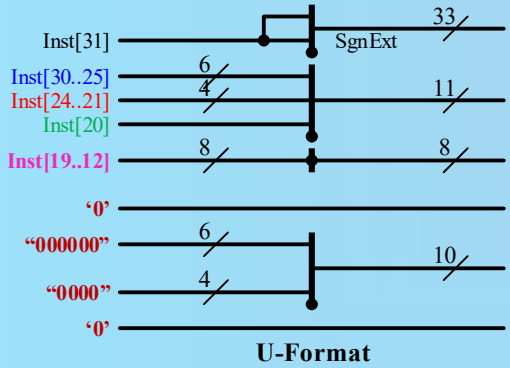


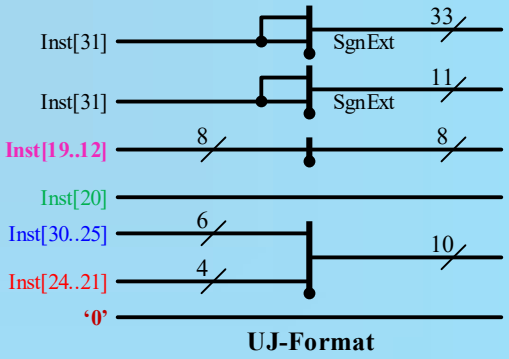


Unpacking the Immediate Constant

The U-Formats also have overlapping fields.



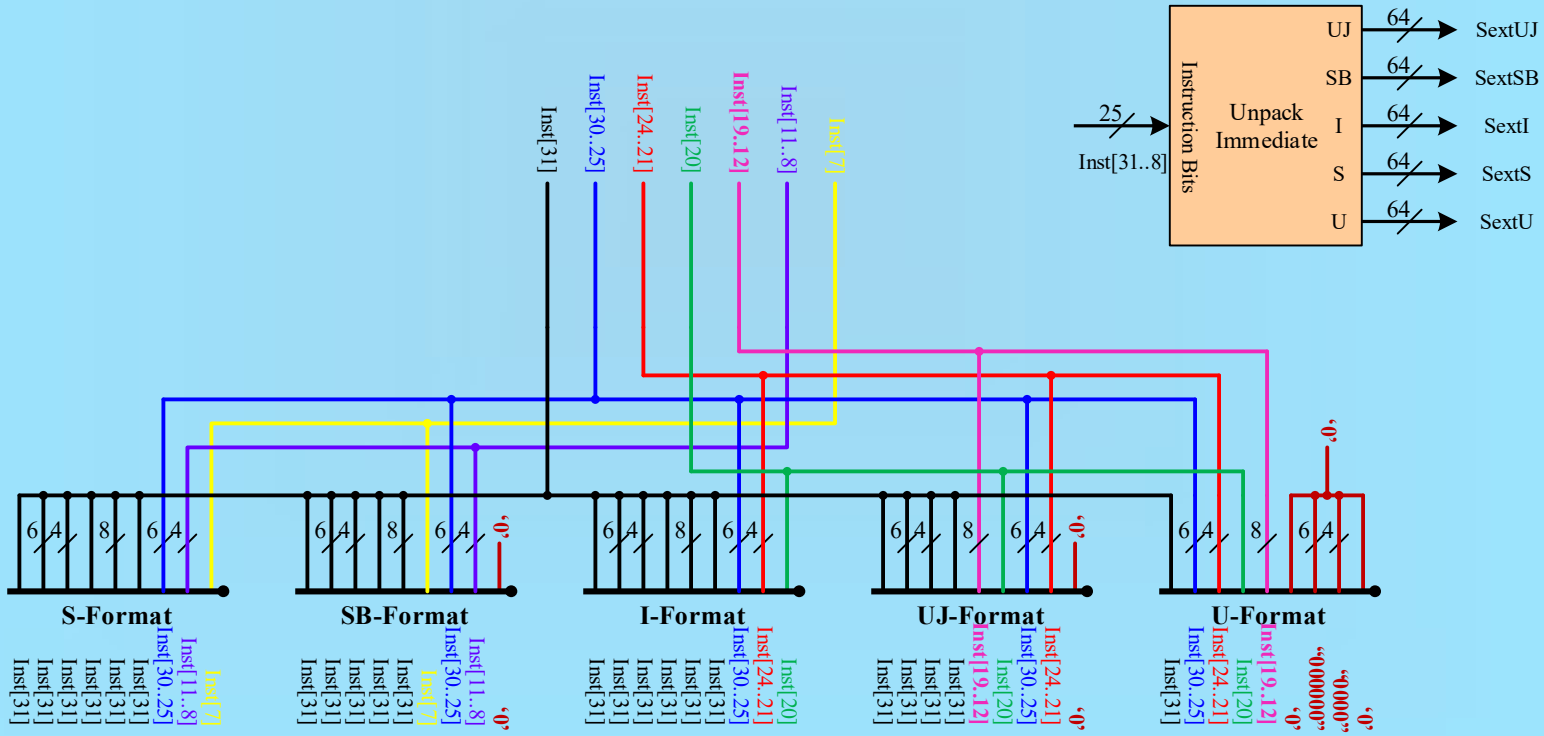




UJ and SB Instruction Formats

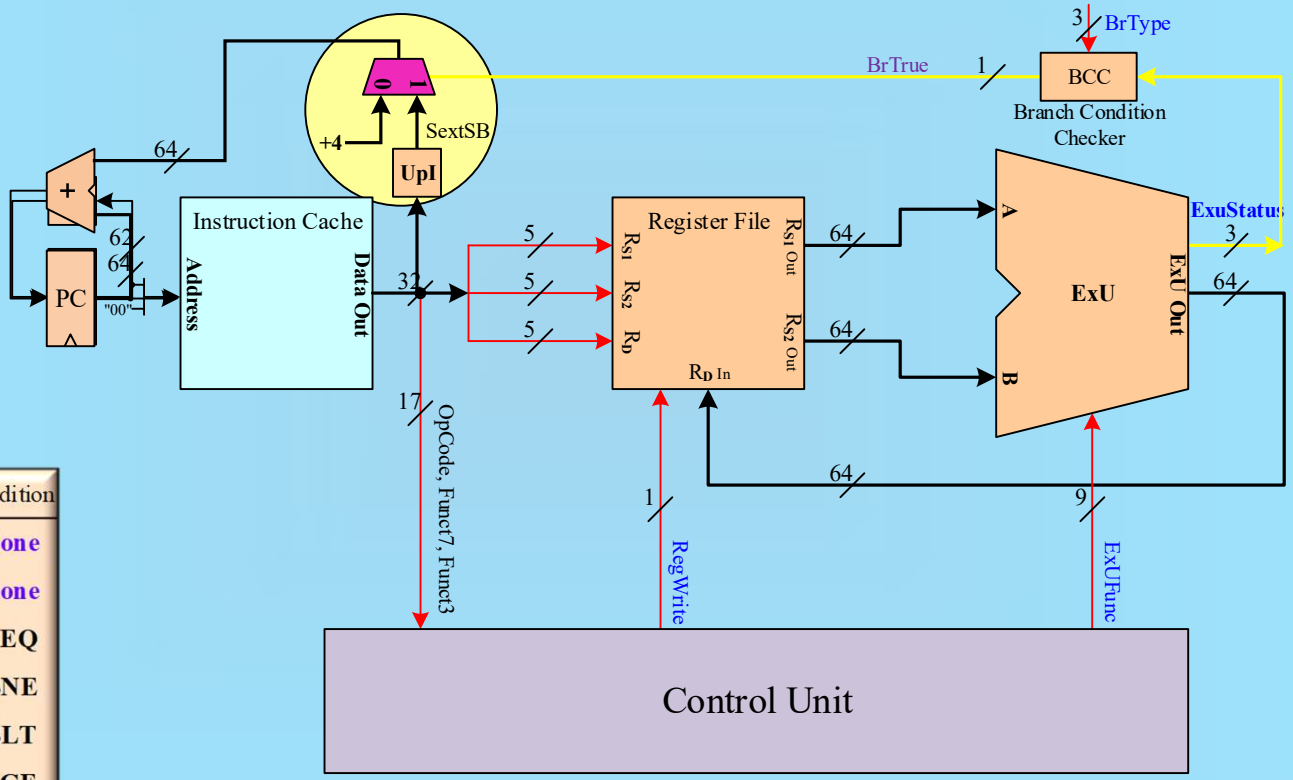
To simplify the diagrams we shall assume that the five outputs of the Unpack Immediate circuit are all 64-bits. (assume that zeros have been inserted in the lower bits when required) The labelling of the outputs is shown below.

The control signal "Format" is unnecessary. The outputs Sext**I**, Sext**S** & Sext**U** are all directed to the Execution Unit and thus may be selected by the ExUSrcB control lines.
The Sext**SB** and Sext**UJ** outputs are connected to the Next Address Logic, a circuit that calculates the address of the next instruction for flow control instructions.



Branch Instruction PC Calculation

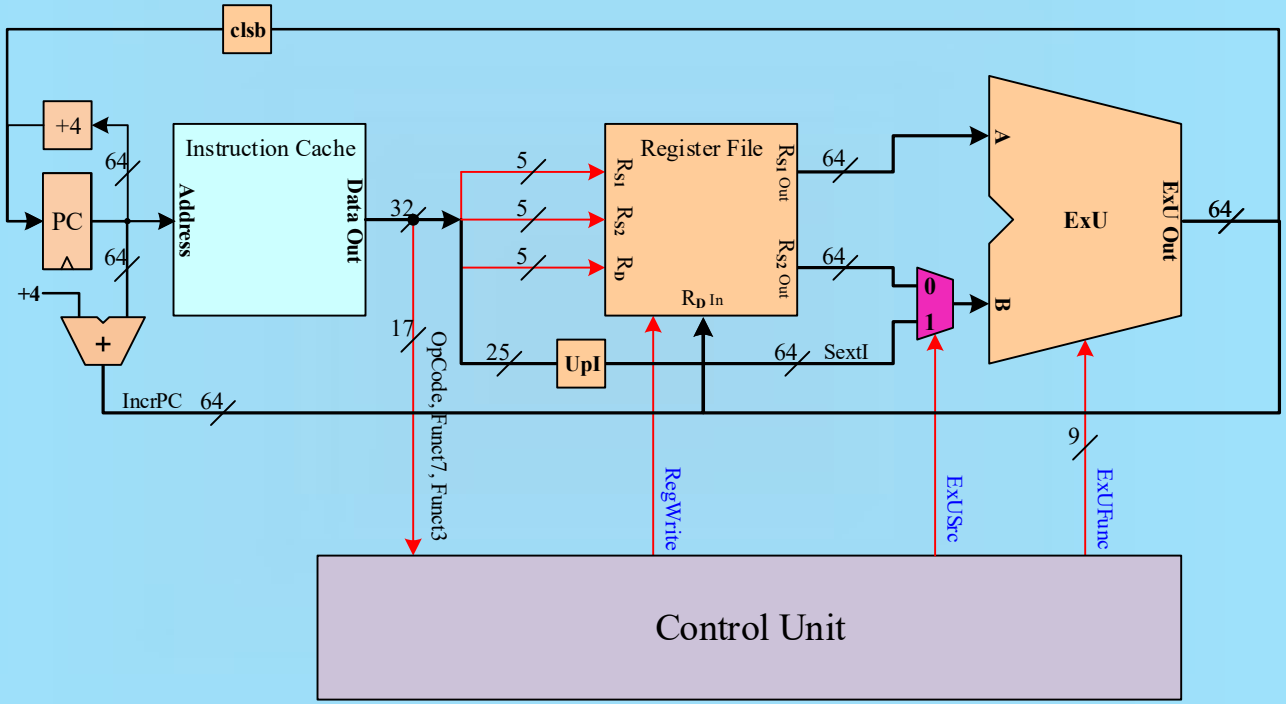
A branch instruction evaluates a condition using the contents of $\mathbf{R_{S1}}$ and $\mathbf{R_{S2}}$. The resulting branch condition can be determined from the status outputs from the Execution Unit, **Zero**, **Cout** & **Ovfl**. (We already have **AltB** and **AltBu** but they only require an inverter and an xor to reconstruct.) Now we introduce a 3-bit control signal, BrType, that indicates the type of Branch Condition.



BrType			condition
0	0	0	None
0	0	1	None
0	1	0	BEQ
0	1	1	BNE
1	0	0	BLT
1	0	1	BGE
1	1	0	BLTU
1	1	1	BGEU

Jump & Link PC Calculation

The **`jalr`** instruction is an **I-Format** instruction. The 12-bit immediate constant is added to the value in $\mathbf{R_{S1}}$ exactly as it would for an instruction like **`addi`**. The lsb of the result is cleared and then the result is transferred to the PC. PC + 4 is stored in the destination register.



The difference between **`jalr`** and an execution-class instruction is that the sources for the destination register and PC have swapped.

IncrPC and **ExUOut** have swapped their destinations, **PC** and $\mathbf{R_D in}$.

Jump & Link PC Calculation

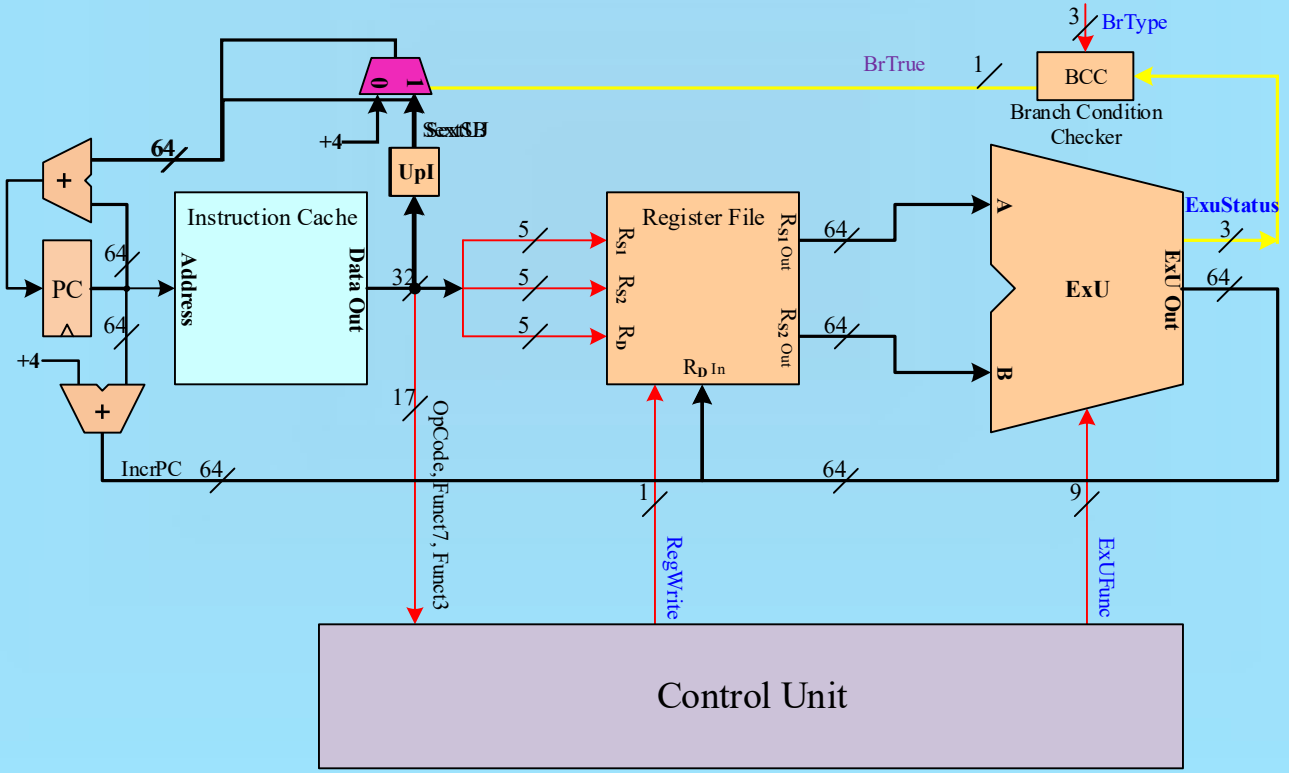
The **`jal`** instruction is an **UJ-Format** instruction.

The 20-bit immediate constant is unpacked by appending a '0' at the lsb and then sign-extending to 64-bits. This value is added to the PC. PC + 4 is stored in the destination register.

As this addressing mode is PC-relative, let's start by modifying the circuit for a branch instruction.

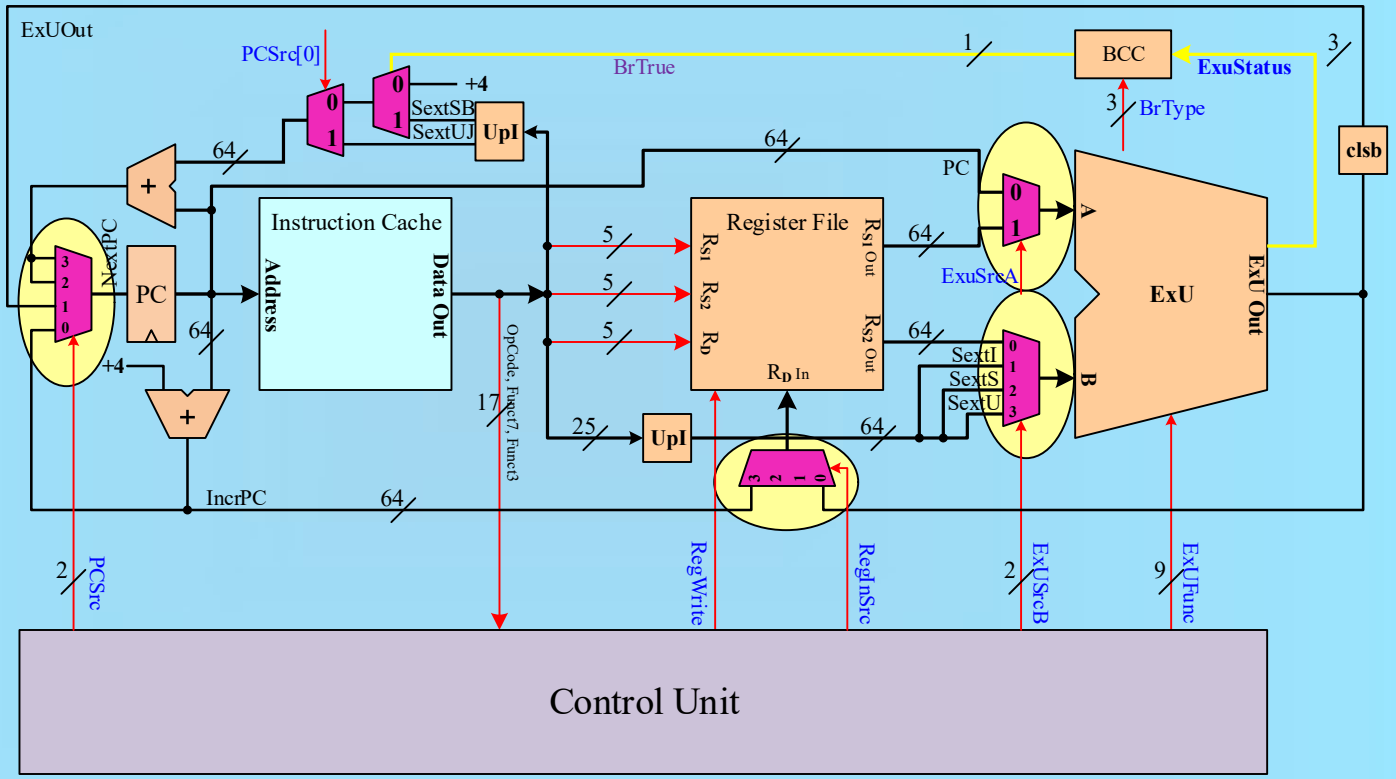
The values **`jta`** and **incrPC** both need to be calculated and thus we need two addition resources. The execution unit and source registers are not being used so we could direct the **PC** to **ExuInA**.

Since be have included an additional adder for branch instructions, let's try using this same adder.



Combining the Previous Circuits

We may now combine all these datapaths by inserting MUXes and introducing suitable control signals. The Source to the B input of ExU has been expanded to include more immediate formats. A MUX is inserted before the **PC** to select the value to be stored that targets the next instruction. (NextPC)



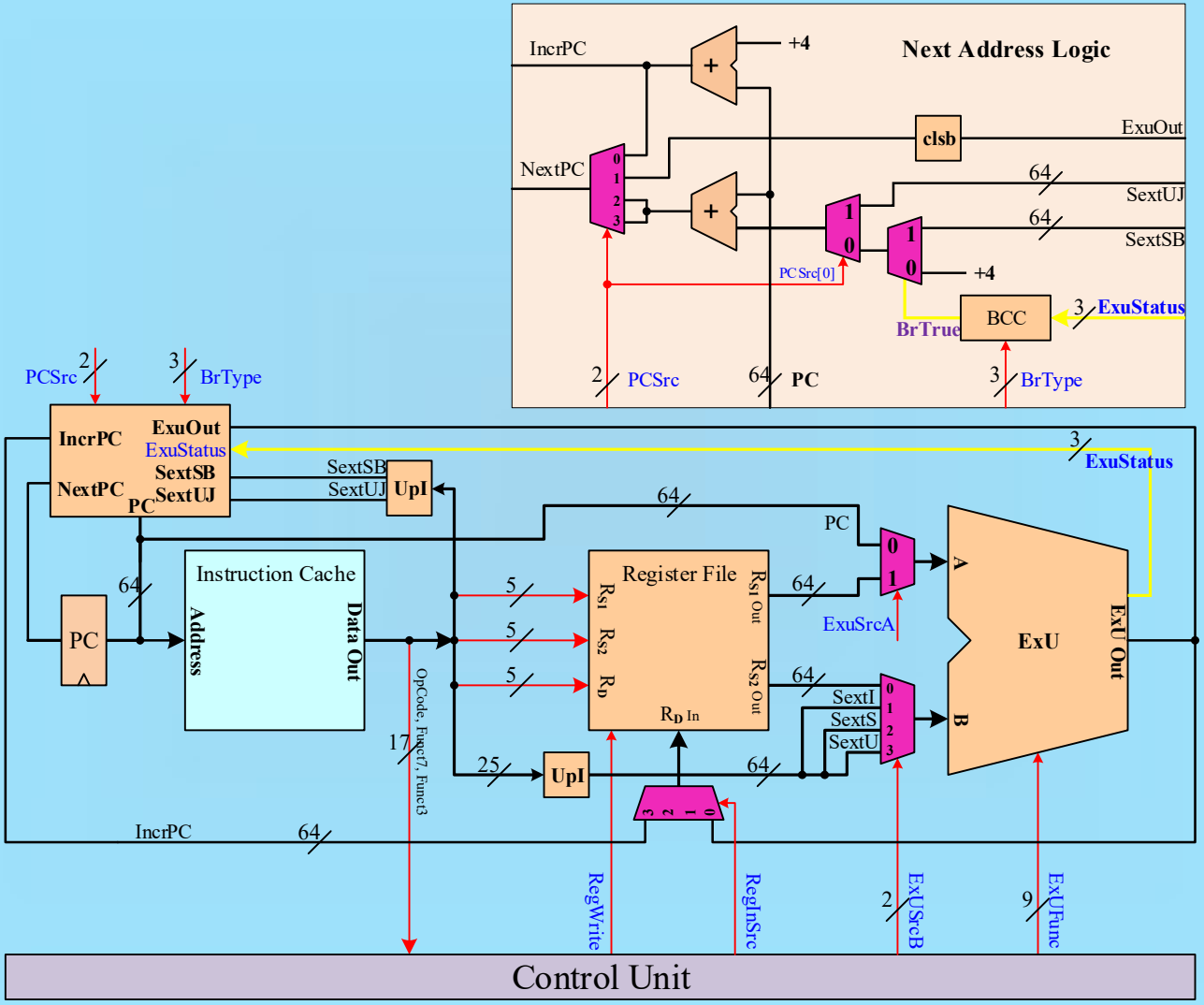
PCSrc		NextPC
0	0	PC+4
0	1	ExuOut
1	0	Branch
1	1	PC+2*imm20

ExuSrcB		Source
0	0	R_{S2} Out
0	1	SextI
1	0	SextS
1	1	SextU

The instructions **`lui`** and **`auipc`** are easily included by directing **PC** and **SextU** to the execution unit.

The Next-Address Logic

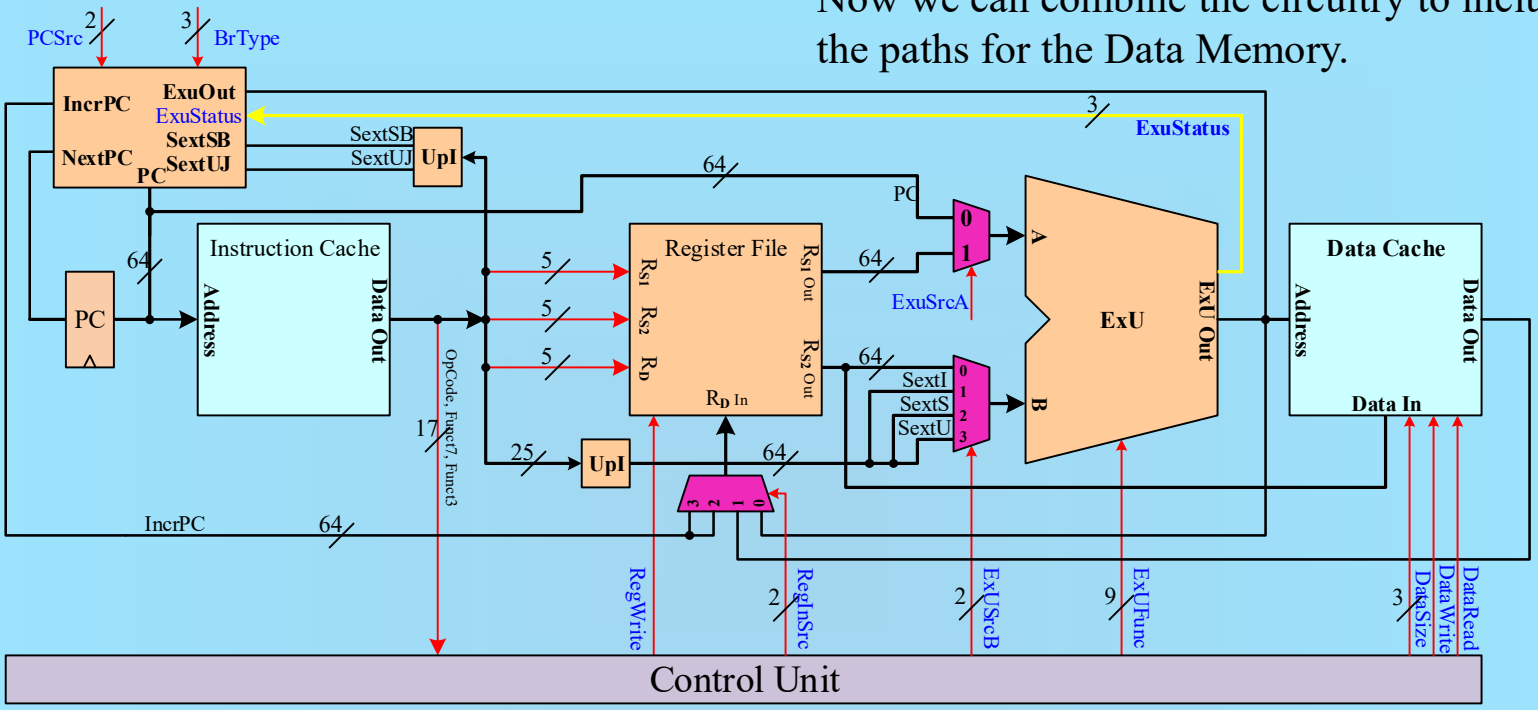
It is convenient to contain most of this new circuitry in a sub-circuit called the Next-Address Logic.



Adding the Data Memory Unit

SFU

Now we can combine the circuitry to include the paths for the Data Memory.



FuncClass		operation	LogicFn		operation	ShiftFN		operation
0	0	sltu	0	0	Lui	0	0	arith
0	1	slt	0	1	A xor B	0	1	sll
1	0	shift/arith	1	0	A or B	1	0	srl
1	1	logic	1	1	A and B	1	1	sra

PCSrc		NextPC
0	0	PC+4
0	1	ExuOut
1	0	Branch
1	1	PC+2*imm20

NotA | FuncClass | LogicFN | ShiftFN | Addn Sub | Ext Word | Cout | Ovfl | Zero

BrType			condition
0	0	0	None
0	0	1	None
0	1	0	BEQ
0	1	1	BNE
1	0	0	BLT
1	0	1	BGE
1	1	0	BLTU
1	1	1	BGEU

DataSize			Write OP	Read Op
0	0	0	byte	Ubyte
0	0	1	half	Uhalf
0	1	0	word	Uword
0	1	1	double	double
1	0	0	byte	Sbyte
1	0	1	half	Shalf
1	1	0	word	Sword
1	1	1	double	double

Building the Control Unit

SFU

The Control Unit is a **combinational circuit** that has 17 inputs (Opcode, Funct3, Funct7) and 25 outputs (**Control Signals**). The Circuit may be implemented as a two-stage circuit. The First stage is a decoder that asserts a unique signal for each of the 51 instructions. The second stage is a bank of 25 OR gates that produce the 25 control signals from the instruction signals.

3→RegWrite, $RegInSrc_{[1,0]}$
5→ DataRead, DataWrite, $DataSize_{[2,0]}$
12→ $ExUSrcB_{[1,0]}$, ExUSrcA, $ExUFunc_{[8,0]}$
5→ $PCSrc_{[1,0]}$, $BrType_{[2,0]}$

Using the previous diagram, we can determine the appropriate values for all 25 control signals.