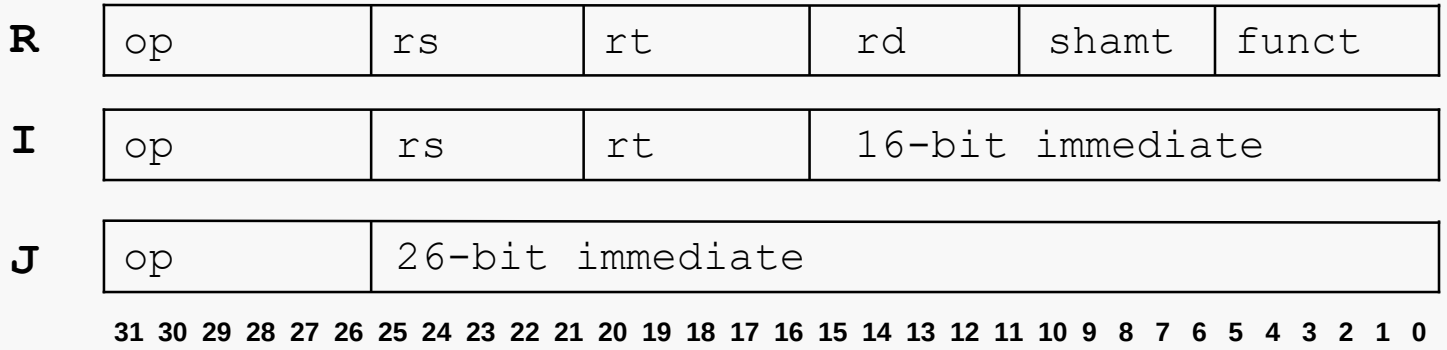
Introduction

We will examine a simplified MIPS implementation first, and then produce a more realistic pipelined version.

A simple, representative subset of machine instructions, shows most aspects:

- Memory reference: `lw`, `sw`
- Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
- Transfer of control: `beq`, `j`



Instruction Execution

I PC → instruction memory, fetch instruction

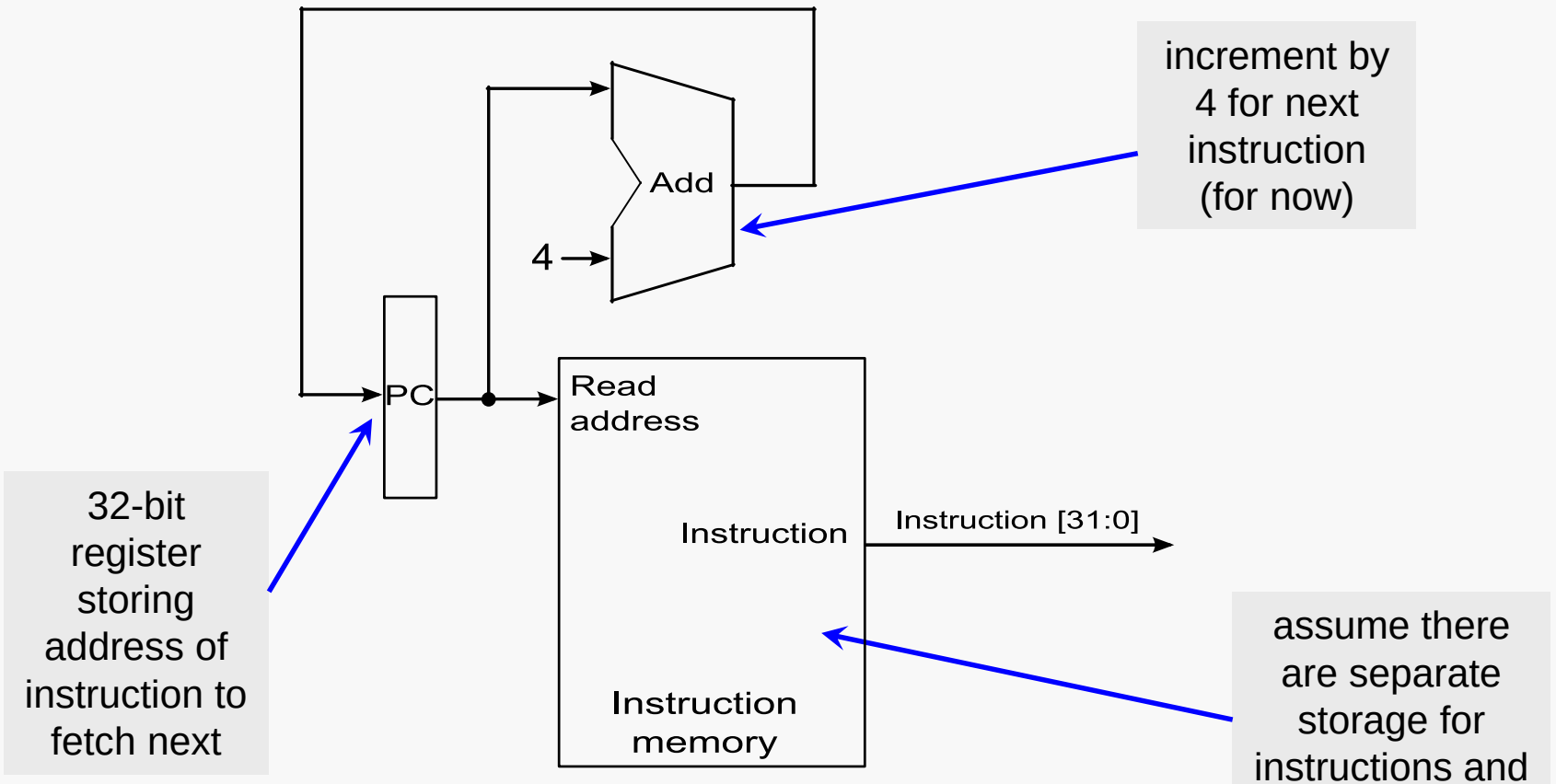
II Register numbers → register file, read registers to get operand(s)

III Depending on instruction class

- May use ALU to calculate needed value
  - R-type: need result of specified operation
  - Load/store: need memory address to be read from/written to
  - Branch: need to compare registers AND need the branch target address
- May access data memory
  - Load/store: access data memory to read/write value
- Set address for next instruction fetch: PC ← branch target OR PC + 4 OR jump target

Executing Instruction Fetch

Here's what we need for sequential fetches (no `beq` or `j`):



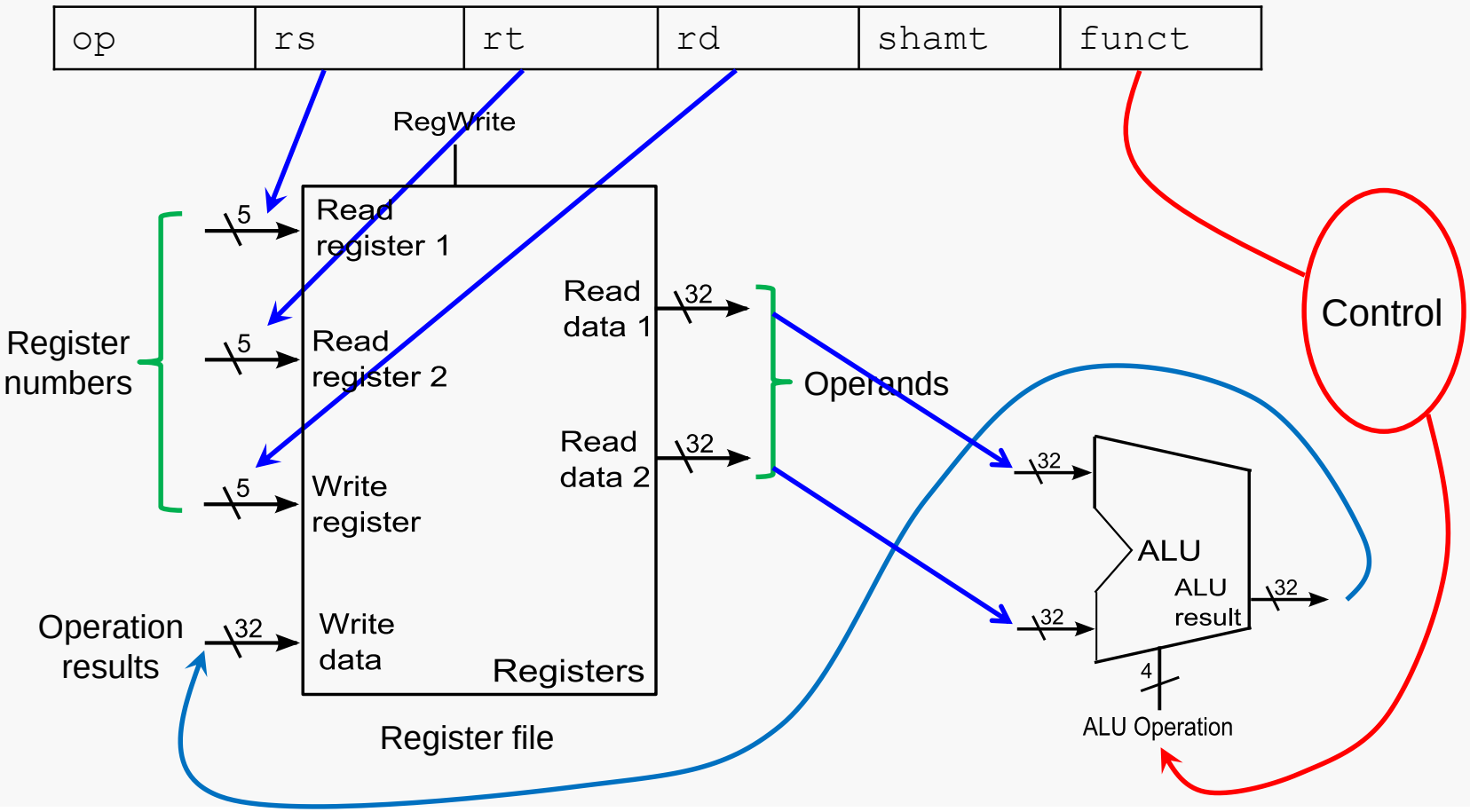
Executing R-Format Instructions

Read two operands from register file

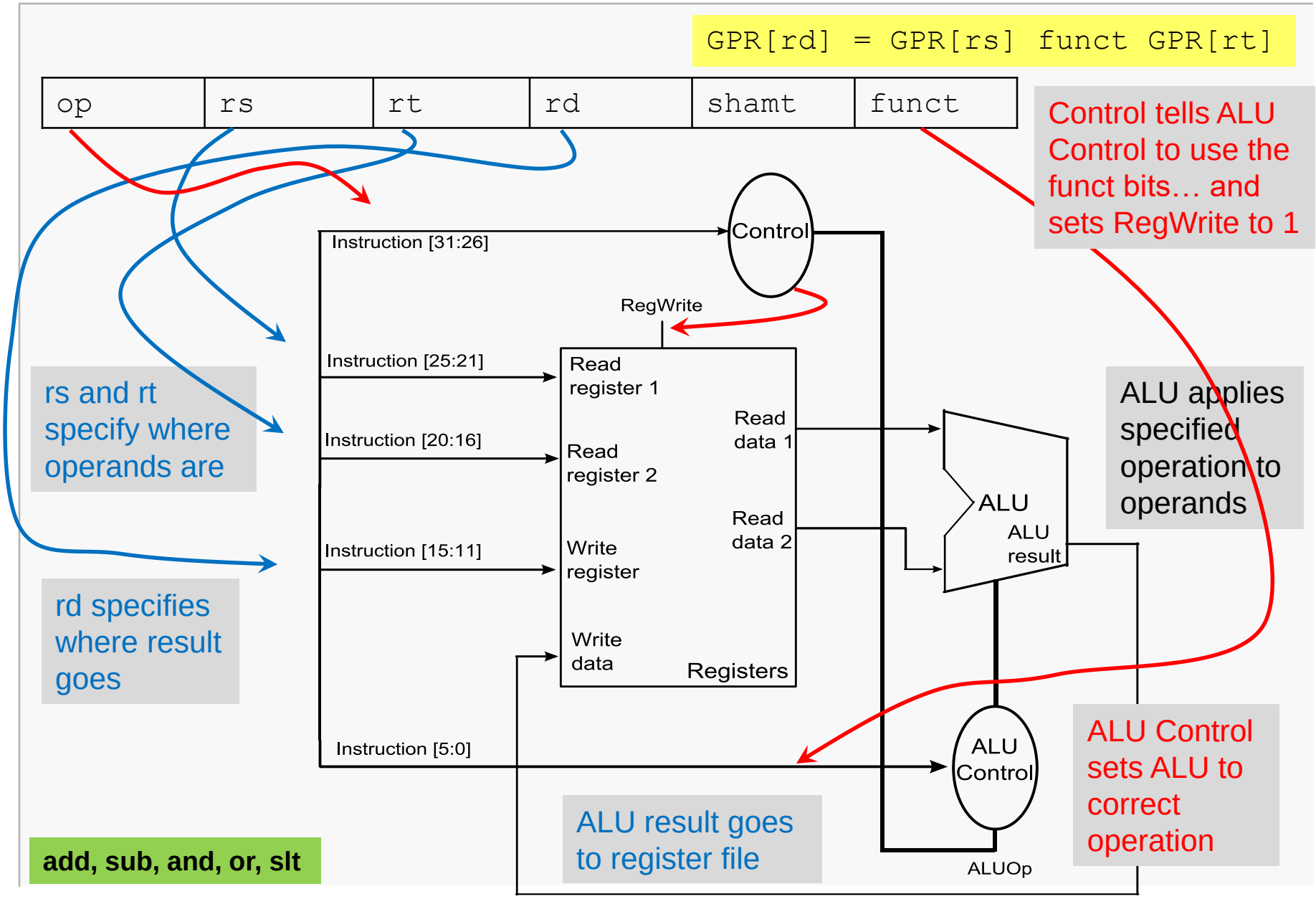
```
GPR[rd] = GPR[rs] funct GPR[rt]
```

Use ALU to perform the arithmetic/logical operation

Write the result to a register



Executing R-Format Instructions



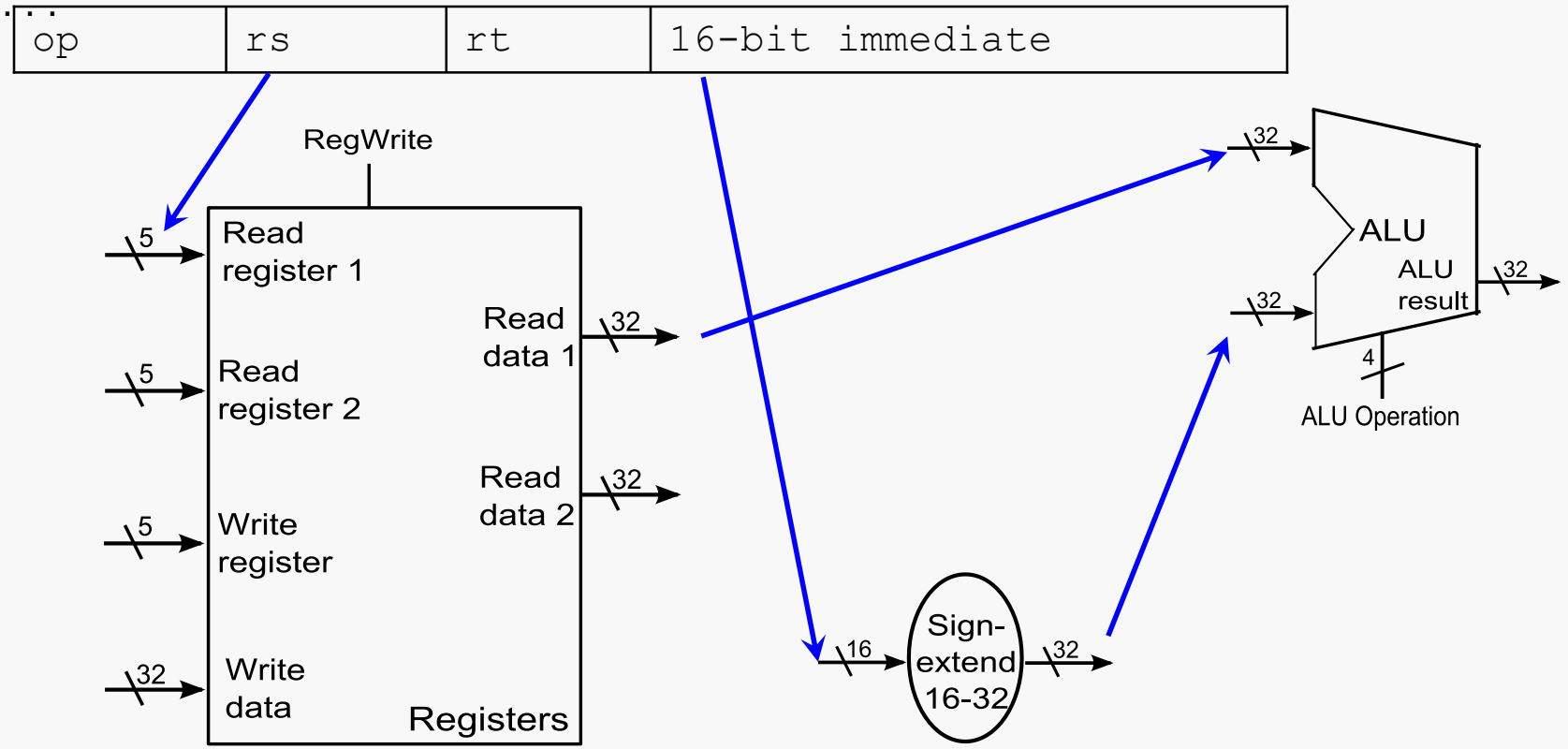
Executing Load Instructions

Read register operand

```
GPR[rt] = Mem[GPR[rs] + imm]
```

Calculate the address to be read using register operand and 16-bit offset from instruction

- Use ALU, but sign-extend offset

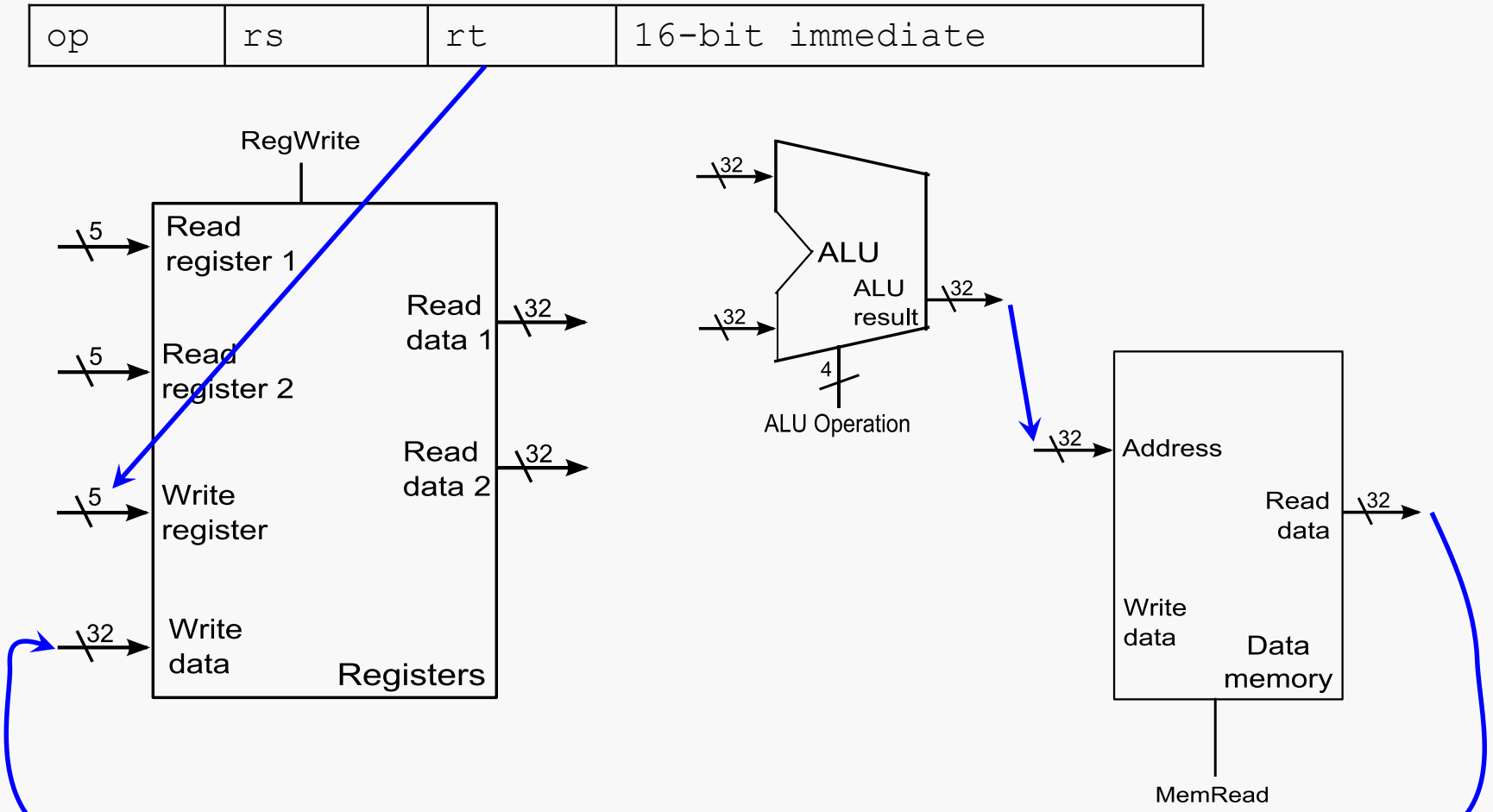


Executing Load Instructions

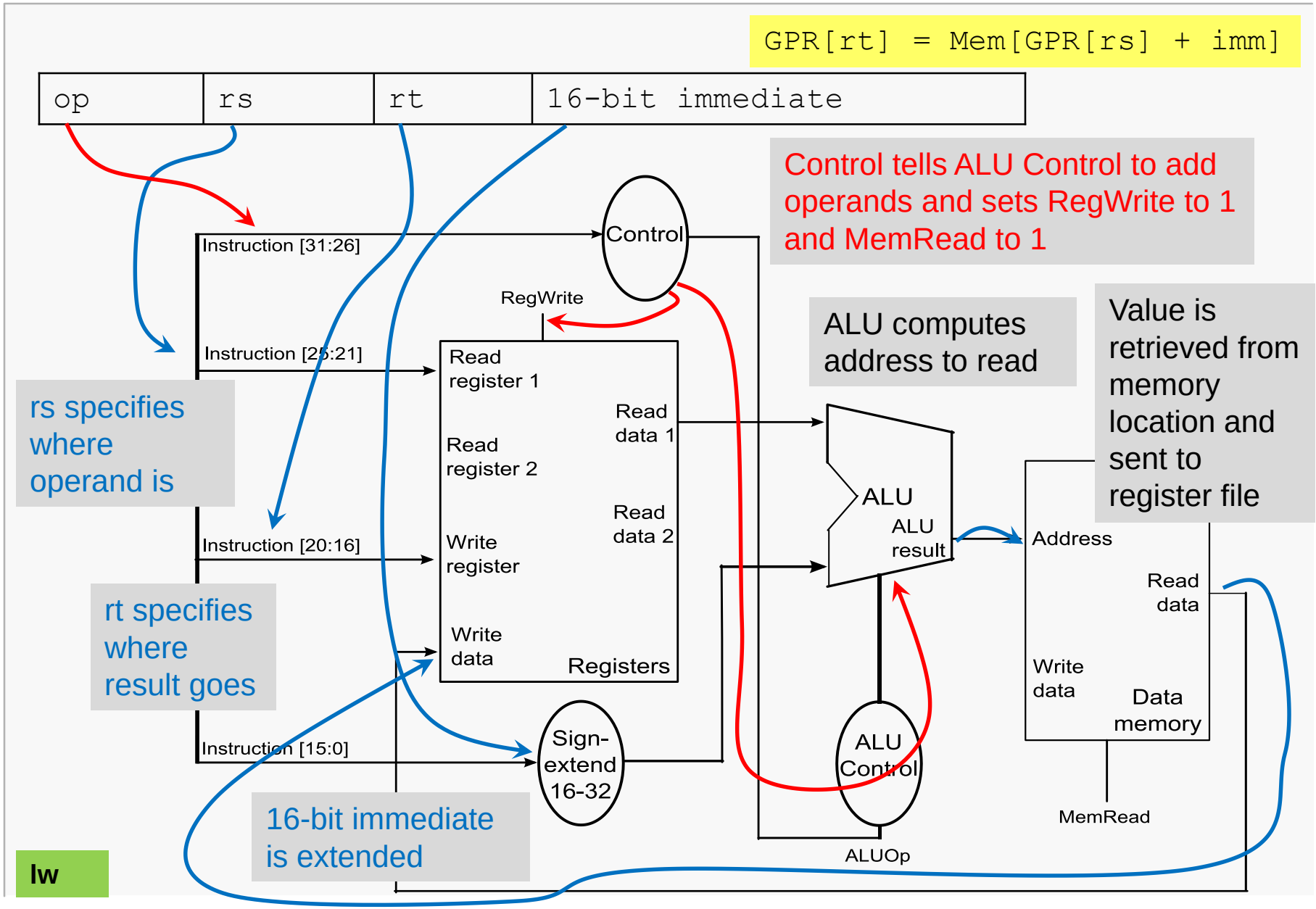
```
GPR[rt] = Mem[GPR[rs] + imm]
```

. . .

Read memory and update register

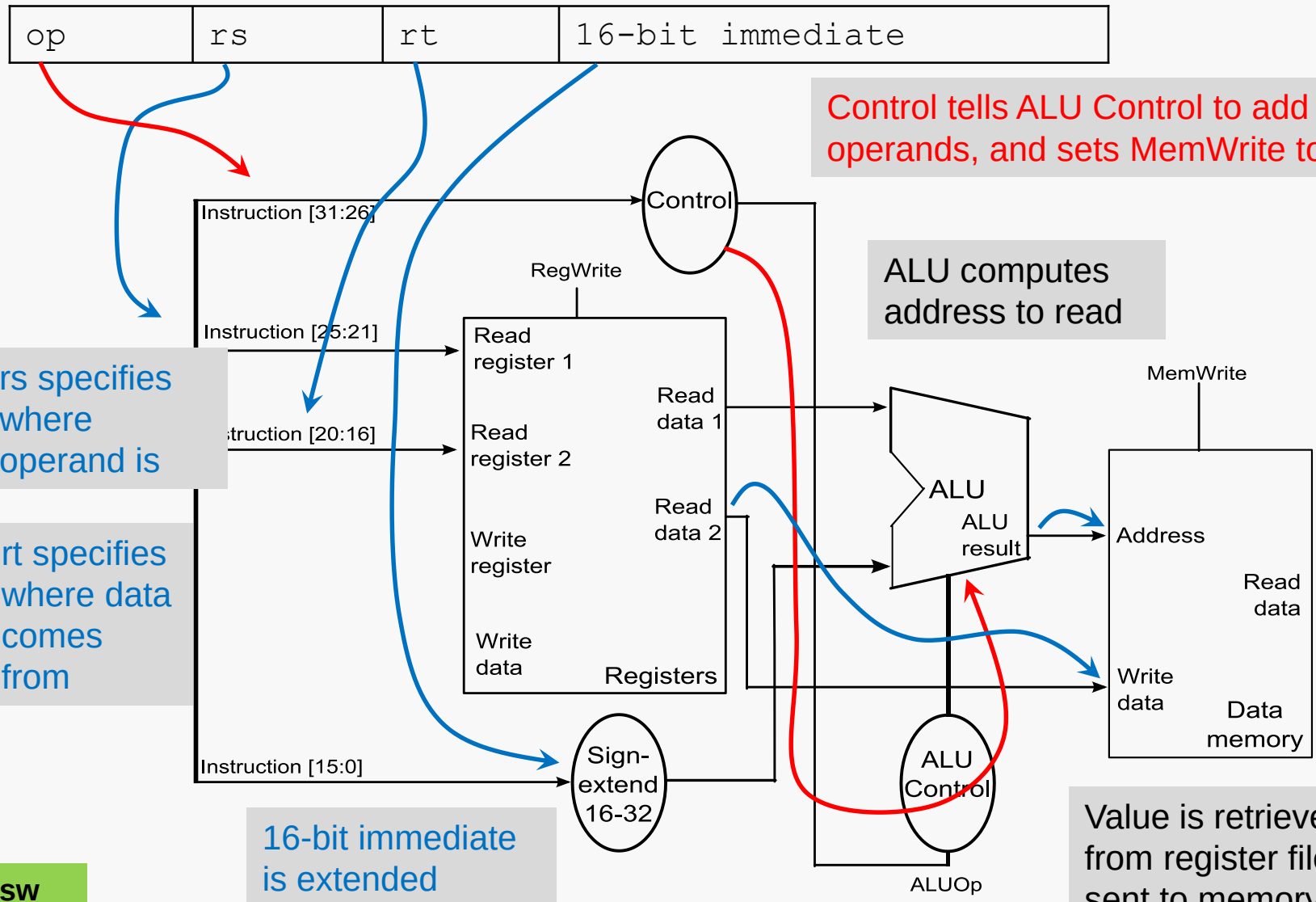


Executing Load Instructions



Executing Store Instructions

```
Mem[GPR[rs] + imm] = GPR[rt]
```



Unifying the Designs

In order to produce a complete datapath design, we must identify and deal with any conflicts.

First, consider the specification of the register numbers supplied to the register file.

They come from the current instruction, but using which bits?

	Read reg 1	Read reg 2	Write reg
R-type	25:21	20:16	15:11
load	25:21	not used	20:16
store	25:21	20:16	not used
	OK	**OK**	**Conflict**

We have a conflict regarding the write register.

To resolve the conflict, we must be able to select one set of bits for R-type instructions and a different set of bits for the load instructions… how do we make a selection in hardware?

Unifying the Designs

We also have a conflicts regarding the source of the write data and the source of the right (lower) operand to the ALU:

	Write data source	Right operand source
R-type	ALU output	register file
load	Data memory	sign-extend
store	not used	sign-extend
	Conflict	**Conflict**

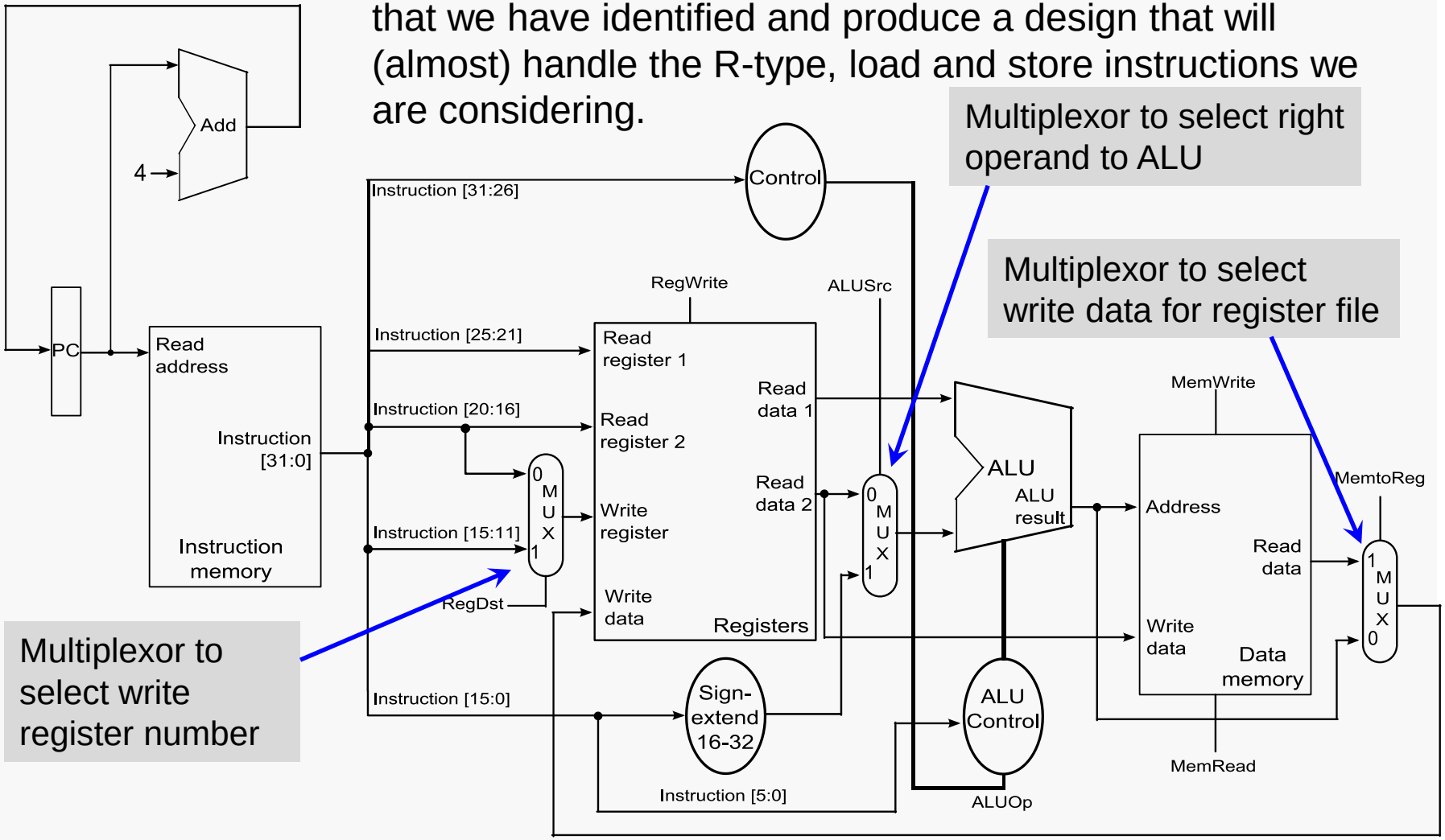
To resolve these conflicts, we must be able to:

- send the ALU output to the register file for R-type instructions, but send the data read from the memory unit to the register file for load instructions

- send the data read from the register file to the ALU for R-type instructions, but send the output from the sign-extender to the ALU for load and store instructions

Unified R-type/Load/Store Datapath

By adding three multiplexors, we can resolve the conflicts that we have identified and produce a design that will (almost) handle the R-type, load and store instructions we are considering.



add, sub, and, or, slt, lw, sw

Analyzing the Control Signals

We've identified quite a few necessary control signals in our design.

Which ones are two-valued? Do any require more than two values?

How should they be set?

The datapath cannot operate correctly unless every control signal is managed properly.

Two things to remember:

- Every line always carries a value. We may not know (or care) what it is in some cases. But there is always a value there.

- It doesn't matter what value a line carries if that value is never stored or used to make a decision. (Hence, we will find that we have *don't-care conditions*.)

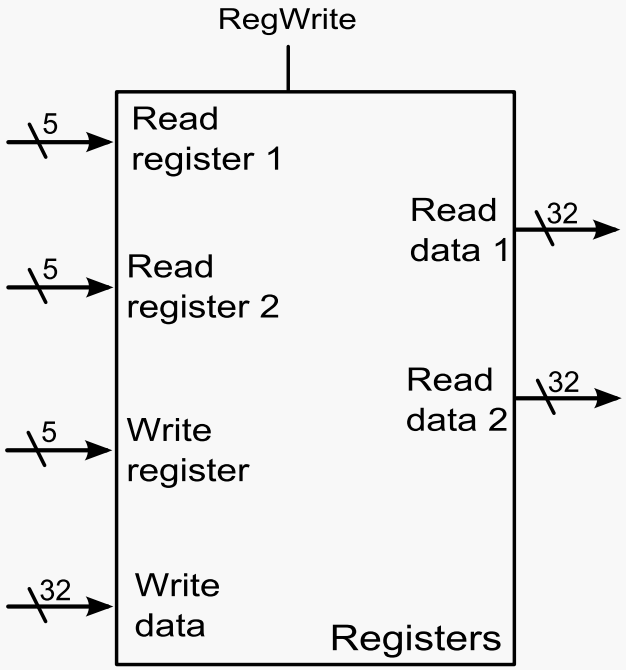
Register File Control

R-type and load instructions require writing a value into a register, but the store instruction does not.

Writing a value modifies the state of the system, so it is *not a don't-care*.

So, we must manage the RegWrite signal accordingly:

	RegWrite
R-type	1
load	1
store	0



Value at Write data is written to the Write register iff RegWrite is 1.

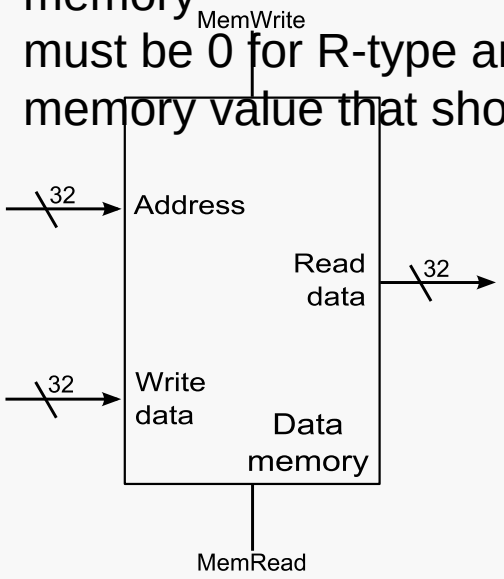
Data Memory Control

MemRead

- must be 1 for load instructions, since they copy a value from memory to a register
- might be a don't-care for R-type and store instructions… why? If not, should be 0.

MemWrite

- must be 1 for store instructions, since they copy a value from a register to memory
- must be 0 for R-type and load instructions; otherwise they could modify a memory value that should nc



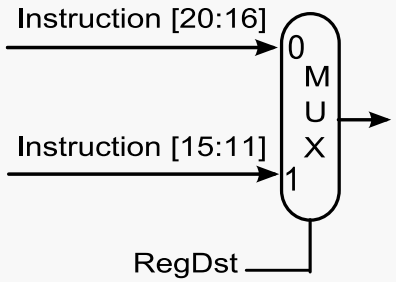
	MemRead	MemWrite
R-type	?	0
load	1	0
store	?	1

Multiplexor Control

RegDst

- must be 0 for load instructions
- must be 1 for R-type instructions
- don't-care for store instructions (why?)

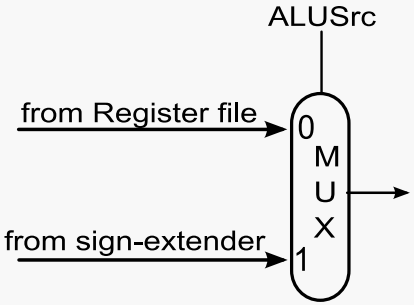
see slide 10



ALUSrc

- must be 0 for R-type instructions
- must be 1 for load and store instructions

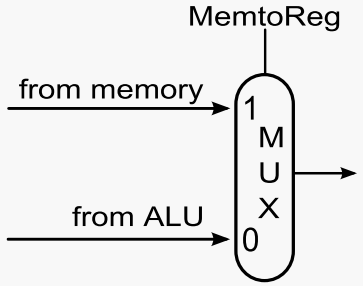
see slide 11



MemtoReg

- must be 1 for load instructions
- must be 0 for R-type instructions
- don't-care for store instructions (why?)

see slide 15



ALU Control

There are a lot of control signals, even in our simple datapath.

At this point, almost all of them are single-bit signals (i.e., they make a choice between two alternate actions).

The ALU control needs to be different because there are more than two choices for what it will actually do:

	ALU
R-type **add** **sub** **and** **or** **slt**	 add subtract and or slt
load	add
store	add

So, the ALU will require a multi-bit control signal… why? How many bits?

ALU Control

This suggests separating the control logic into two modules:

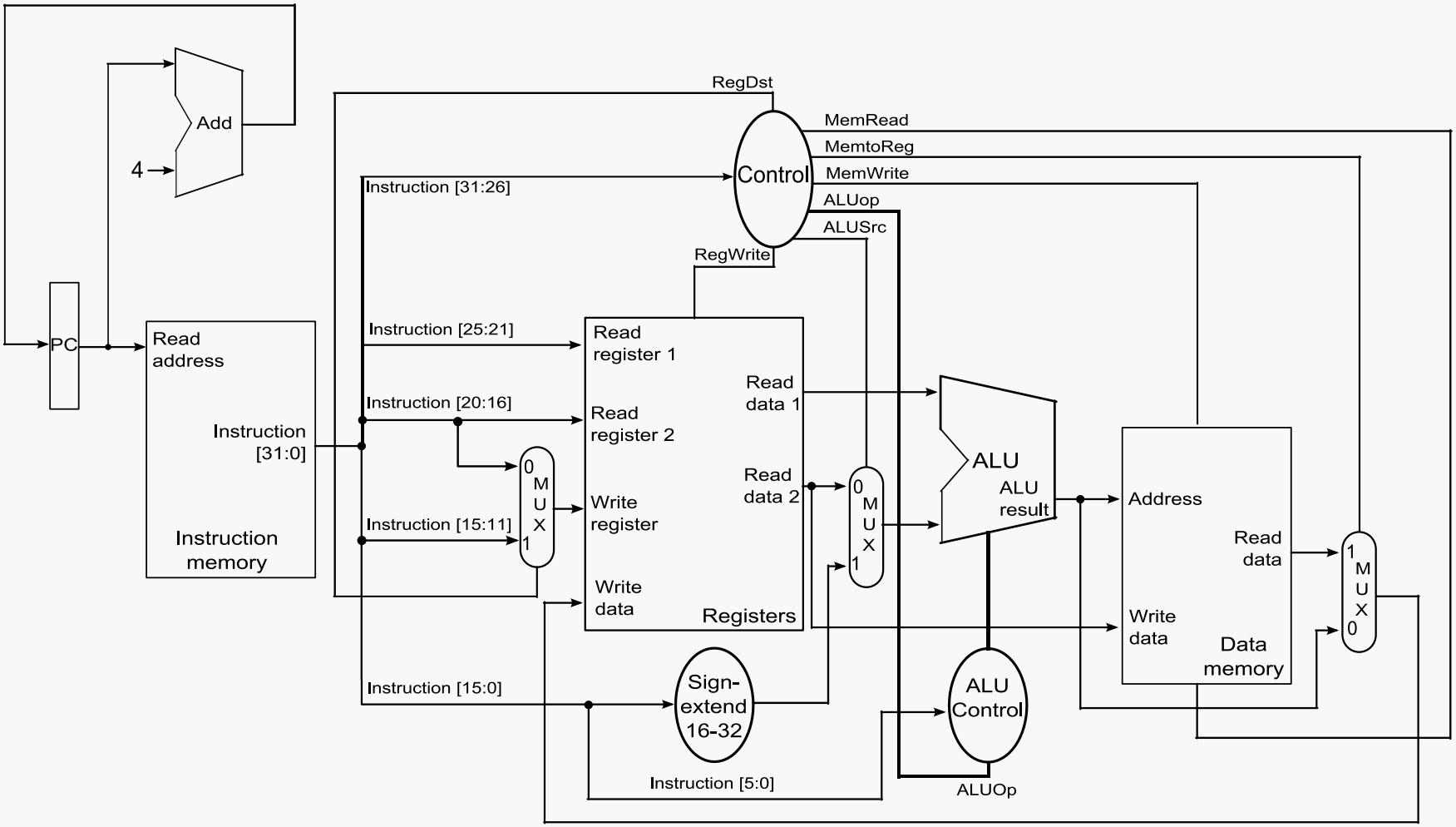
- a master control module that determines the type of instruction being executed and sets the non-ALU control signals
- a secondary control module that manages the interface of the ALU itself

The master module will send :

- a specific selector pattern for the ALU if the instruction is not R-type; e.g., it sends the ADD selector pattern for `lw` and `sw` instructions
- a flag telling the secondary module to analyze the `funct` bits if the instruction is R-type

We'll fill in the details of the two modules later, but for now we do know what each must do, at a high level.

Unified Datapath Design with Control



add, sub, and, or, slt, lw, sw

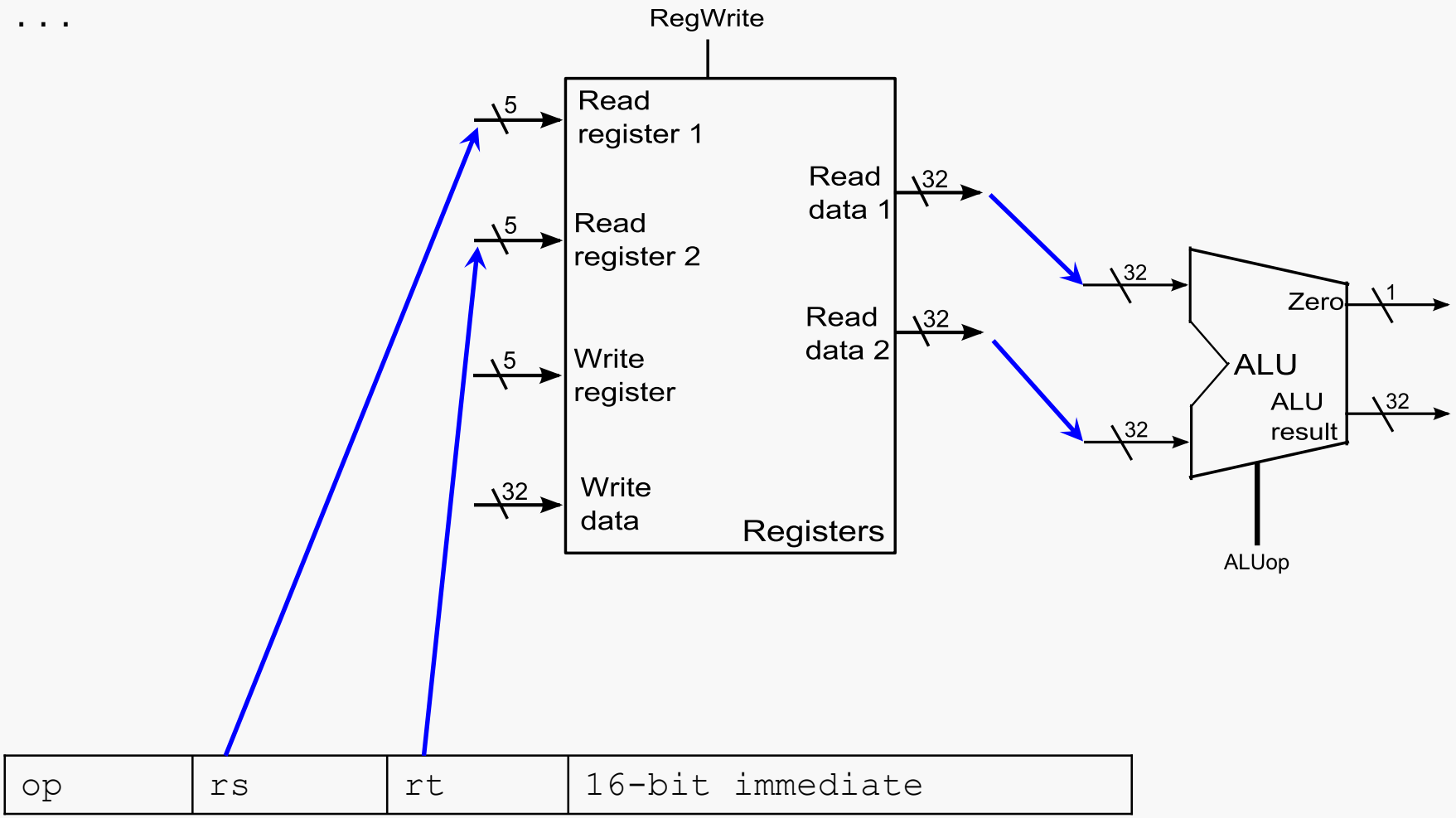
Executing Branch Instructions

```
if GPR[rs] == GPR[rt] then
    PC = PC + 4 + (imm << 2)
```

Read two operands from the register file

Use the ALU to compare the operands: subtract and check Zero signal

. . .



Executing Branch Instructions

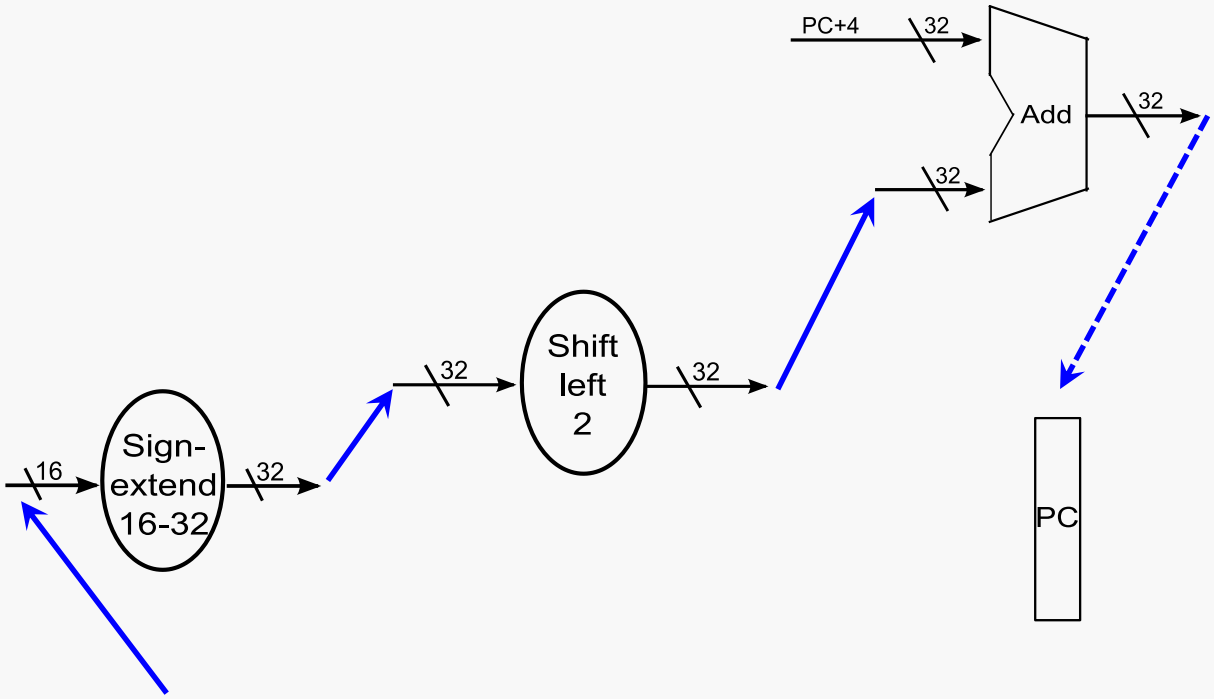
```
if GPR[rs] == GPR[rt] then
    PC = PC + 4 + (imm << 2)
```

. . .

Calculate the branch target address:

- Sign-extend displacement (immediate from instruction)
- Shift left 2 places (MIPS uses word displacement – why?)
- Add to PC + 4 (already calculated PC + 4 during the instruction fetch)

Send computed branch target address to the PC (if taken)



op	rs	rt	16-bit immediate

Aside: Branch Target Address

Examine the display below of a short MIPS assembly program, and the addresses at which the instructions will be loaded into memory when the program is executed:

If taken, a conditional branch causes the PC to be set to the address of the instruction that's the target of the branch:

```
 address           assembly source code
-------------------------------------------
                            .text
0x00400000     main:         addi  $s0, $zero, 10
0x00400004                   addi  $s1, $zero, 20
0x00400008                   j     check

0x0040000C     repeat:       sub   $s3, $s1, $s0
0x00400010                   sra   $s3, $s3, 1
0x00400014                   beq   $s3, $zero, exit
0x00400018                   add   $s0, $s0, $s3
0x0040001C     check:        bne   $s0, $s1, repeat

0x00400020     exit:       li    $v0, 10
0x00400024                   syscall
```

Aside: Branch Target Address

```
 address           assembly source code
-----------------------------------------
                            .text
0x00400000     main:        addi  $s0, $zero, 10
0x00400004                  addi  $s1, $zero, 20
0x00400008                  j     $check

0x0040000C     repeat:      sub   $s3, $s1, $s0
0x00400010                  sra   $s3, $s3, 1
0x00400014                  beq   $s3, $zero, exit
0x00400018                  add   $s0, $s0, $s3
0x0040001C     check:       bne   $s0, $s1, repeat

0x00400020     exit:       li    $v0, 10
0x00400024                  syscall
```

First of all, note that all the addresses are multiples of 4*.

Therefore, the "distance" between two instructions is always a multiple of 4.

***QTP: why isn't this surprising?**

Aside: Branch Target Address

The immediate we store for a `beq` instruction is determined by the "distance" from the `beq` instruction to the instruction that is the target of the branch.

```
 address           assembly source code
-----------------------------------------
. . .
0x0040000C     repeat:   sub   $s3, $s1, $s0
. . .
0x0040001C      check:   bne   $s0, $s1, repeat
0x00400020       exit:   li    $v0, 10
. . .
```

However, while `bne` is being fetched:

- we have already computed PC + 4
- which is the address of the next instruction in memory
- so we need to compute the distance relative to the next instruction

And note that this computation is done by the assembler when it creates the machine code representation of the instruction, not at runtime.

Aside: Branch Target Address

```
 address           assembly source code
-----------------------------------------
. . .
0x0040000C     repeat:   sub   $s3, $s1, $s0
. . .
0x0040001C      check:   bne   $s0, $s1, repeat
0x00400020       exit:   li    $v0, 10
. . .
```

Here, that "distance" is 0x0C – 0x20 = – 0x14.

In 16-bit 2's complement, 0x14 represented as: `00000000 00010100`

If the "distance" always ends in 00, there's no reason to store that 00 in the instruction…

Aside: Branch Target Address

Do we gain anything if we don't store those two 0's in the instruction?

```
 address           assembly source code
-------------------------------------------
. . .
0x0040000C     repeat:   sub   $s3, $s1, $s0
. . .
0x0040001C      check:   bne   $s0, $s1, repeat
0x00400020       exit:   li    $v0, 10
. . .
```

Yes. We can store a 16-bit "distance" in the instruction, but effectively use an 18-bit "distance" at runtime.

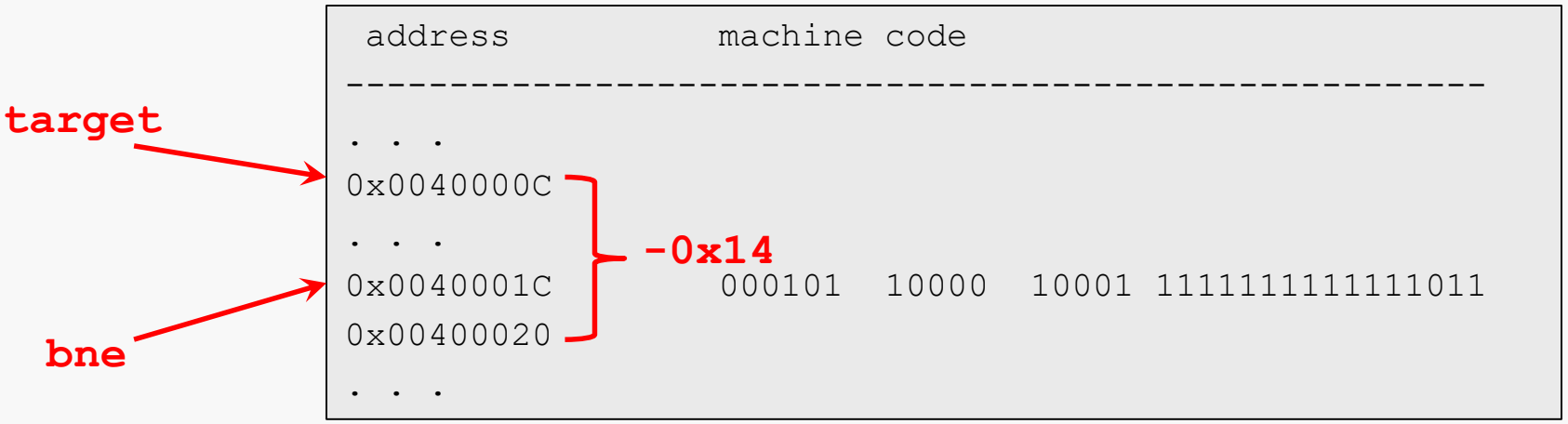
That means that a conditional branch can "branch" farther… which is a gain.

16-bit 2's complement range: -2^{15} to $2^{15}-1$

18-bit 2's complement range: -2^{17} to $2^{17}-1$

Aside: Branch Target Address

Now, consider the machine code for our example:

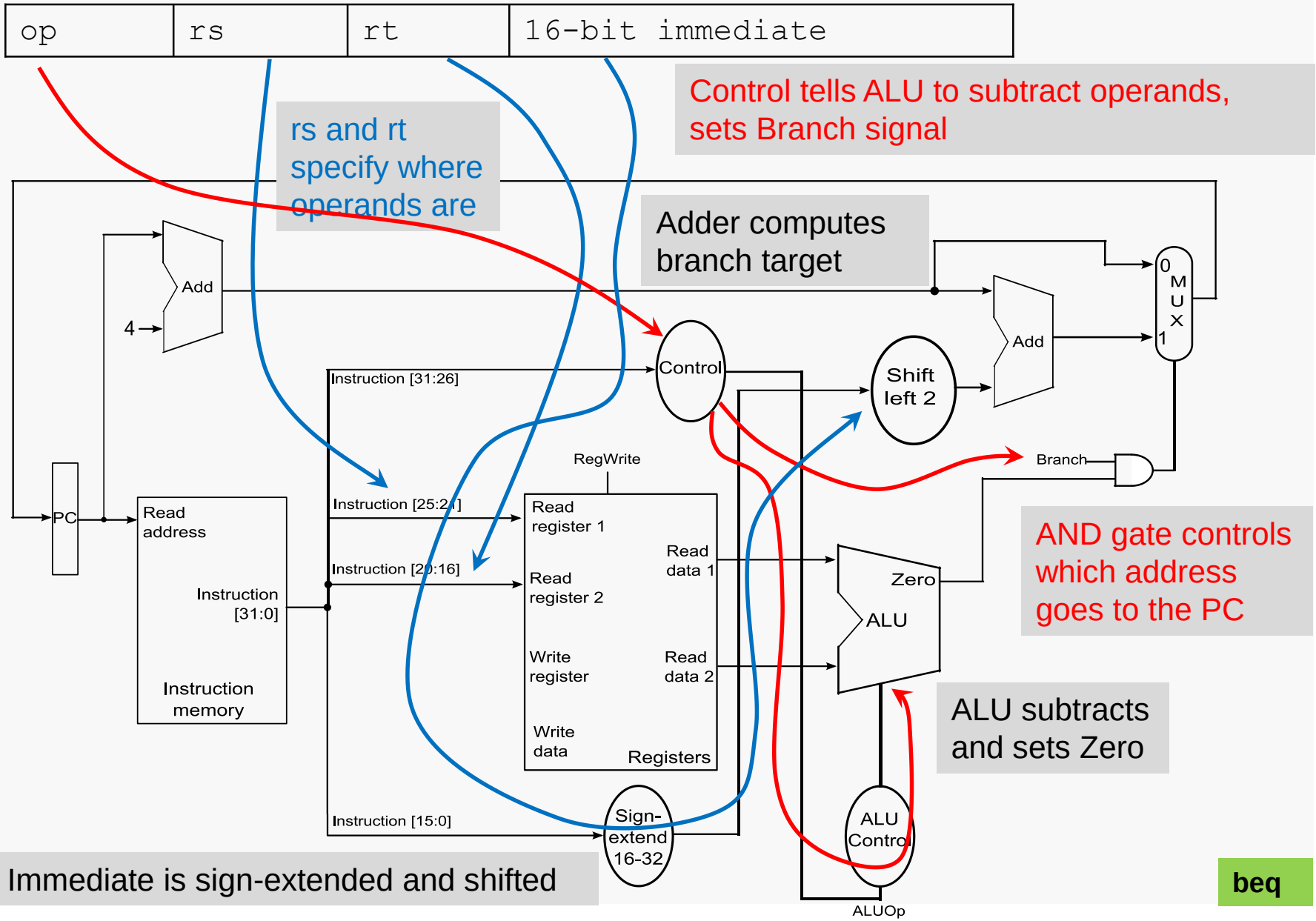


The immediate in the `beq` instruction is: `1111111111111011`

That's -5. If we shift that left by 2 bits (multiply by 4), we get -20 (0x14).

If we add -0x14 to the address `0x00400020`, we get `0x0040000C`, which is the target address we need.

Executing Branch Instructions



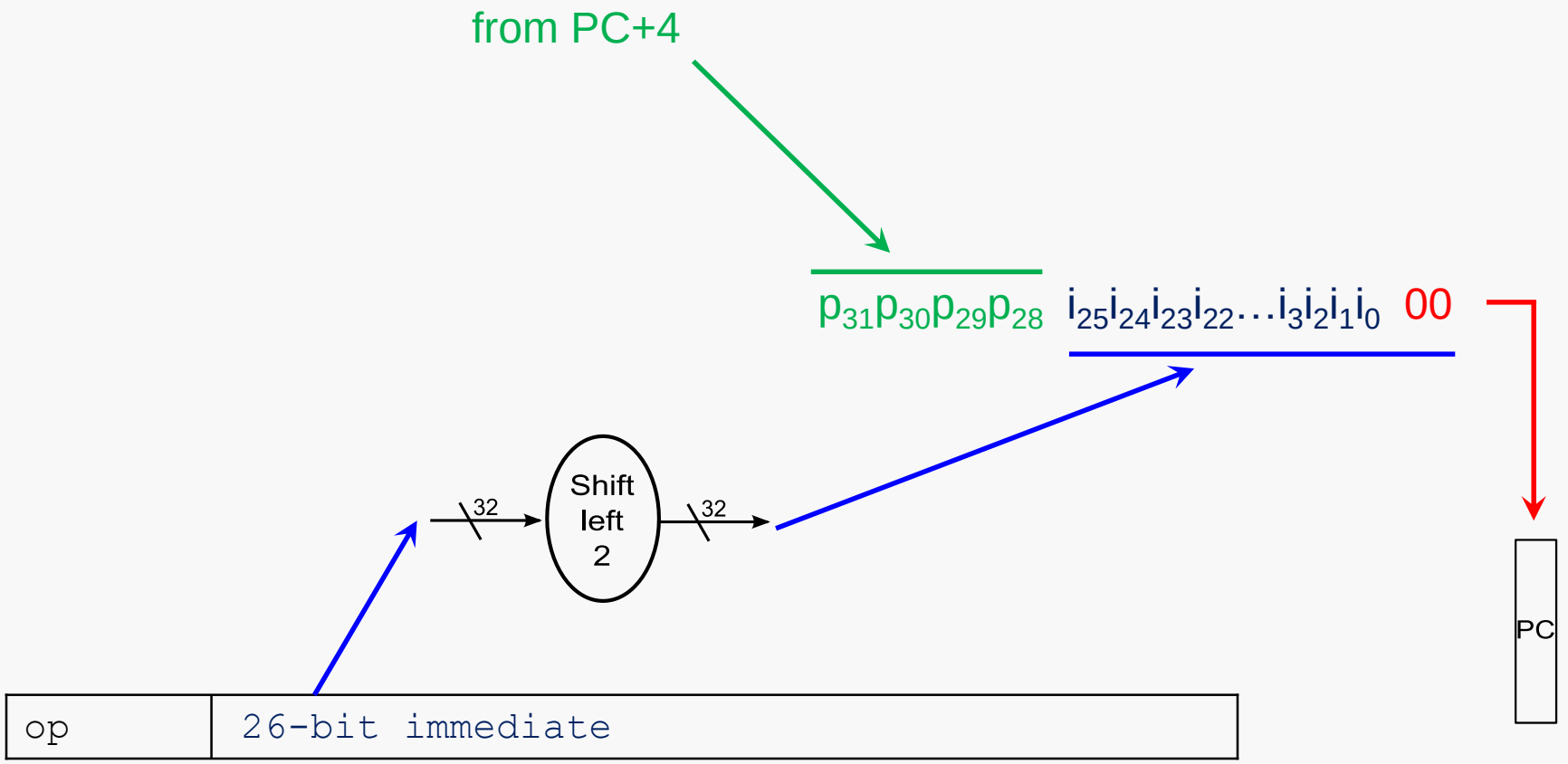
Executing Jump Instructions

Calculate the jump target address:

```
PC = (PC+4)[31:28] | (IR[25:0] << 2)
```

- Shift left 2 places (just as with branch target address calculation)
- Concatenate with PC + 4[31:28] (already calculated PC + 4 during the instruction fetch)

Send computed jump target address to the PC



Aside: Jump Target Address

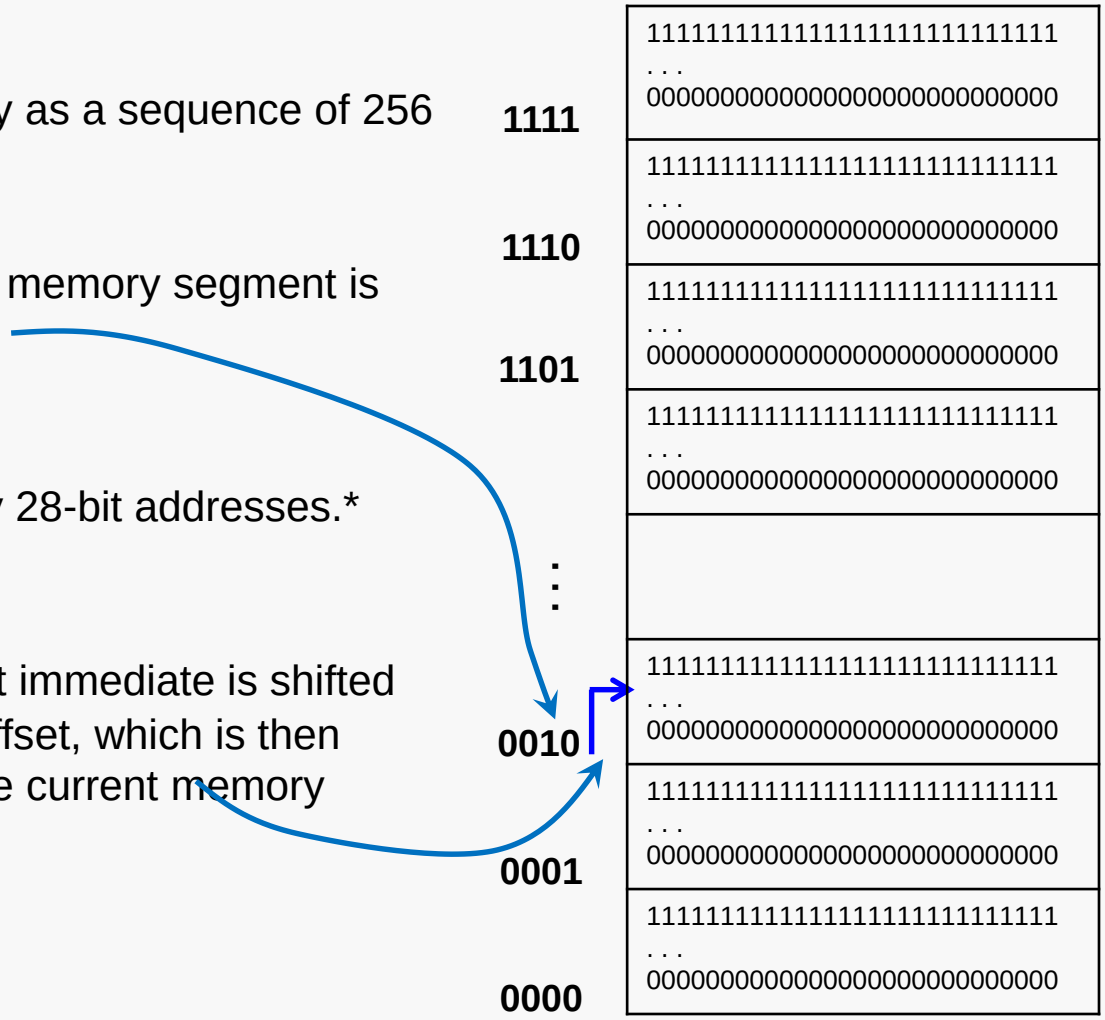
The calculation of the jump target address is similar to the calculation of the branch target address (`beq`).

The MIPS model is to view memory as a sequence of 256 MB segments.

The starting address of the current memory segment is given by the high 4 bits of PC + 4.*

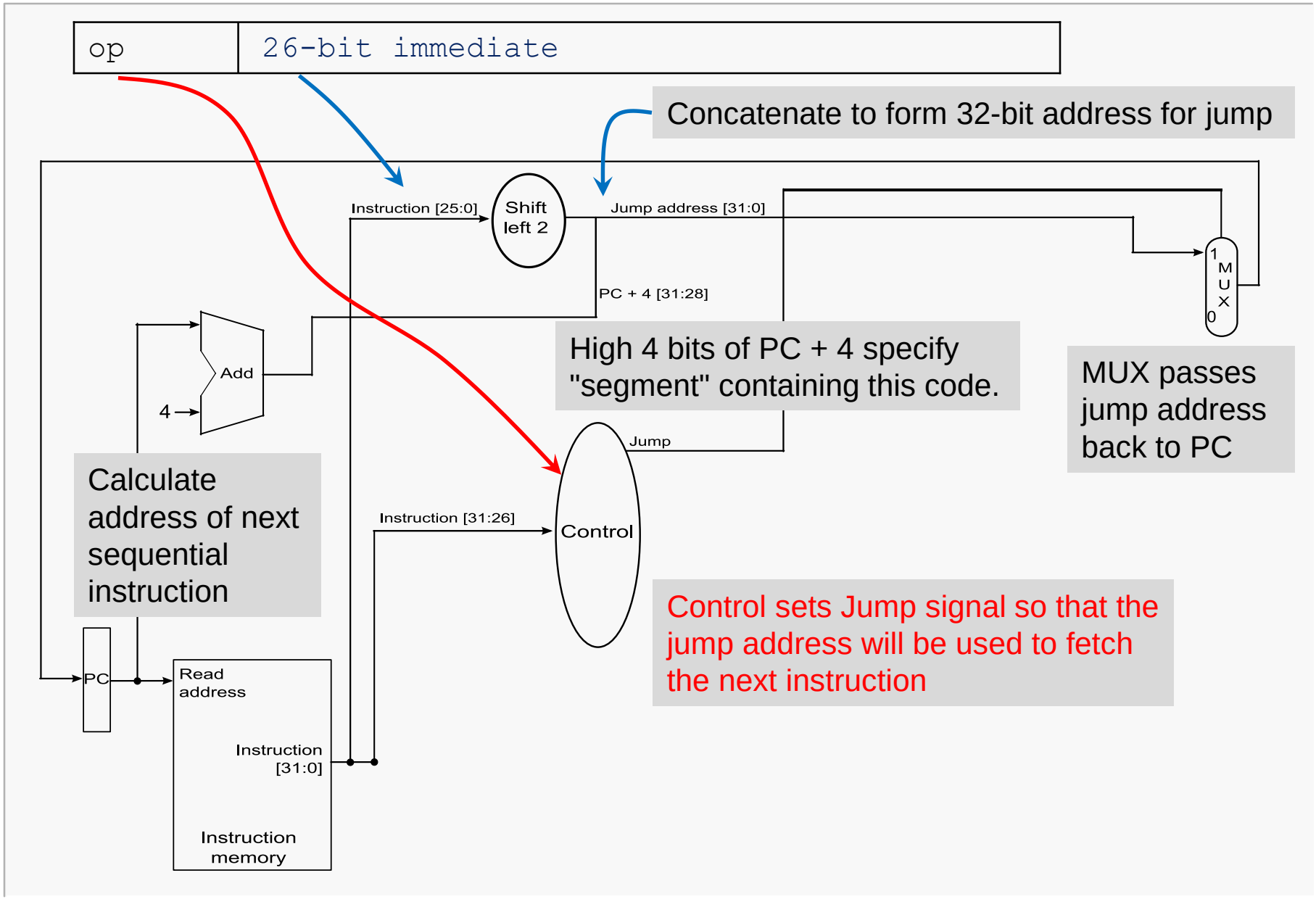
Now, 256 MB can be addressed by 28-bit addresses.*

For jump (`j`) instructions, the 26-bit immediate is shifted 2 bits to the left, yielding a 28-bit offset, which is then added to the starting address of the current memory segment.

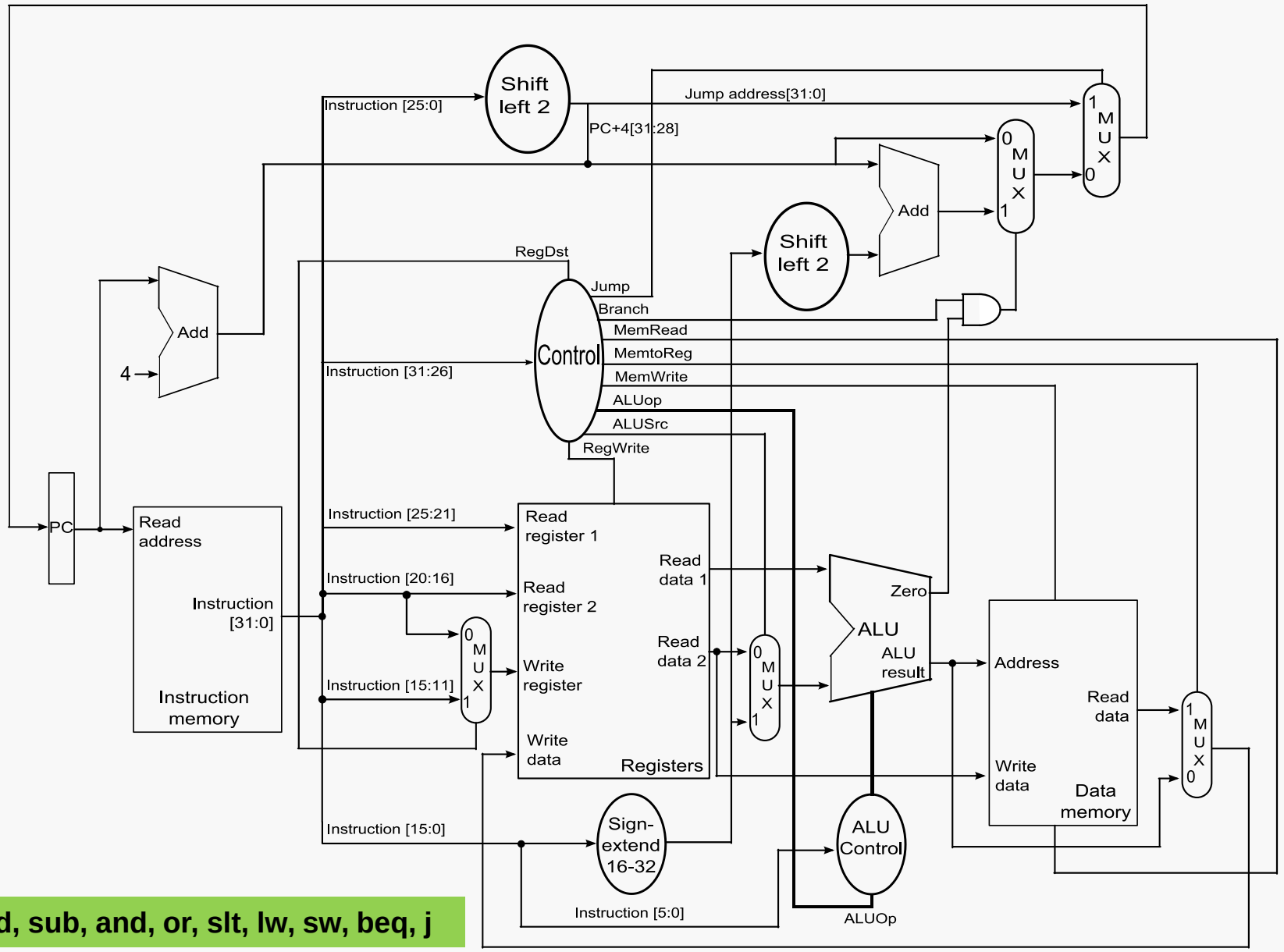


***QTP: why?**

Executing Jump Instructions



Unified Datapath



add, sub, and, or, slt, lw, sw, beq, j

Summary

The unified datapath that we have designed:

- illustrates many of the logical issues that must be solved in designing any datapath
- can be extended to support additional instructions (easily for some, less so for others)
- is fundamentally unsatisfactory in that it requires a single clock cycle be long enough for every path within the datapath to stabilize before the next instruction is fetched

We may explore the second issue in exercises.

The third issue can only be dealt with by transforming the design to incorporate a pipeline.