

CSEE 3827: Fundamentals of Computer Systems, Spring 2022

Lecture 11

Prof. Dan Rubenstein (danr@cs.columbia.edu)

Agenda (P&H 4.1-4.4)

- Single Cycle Datapath
 - Human (serial steps) v. Computer (massive parallelism & discard) approach to interpreting an instruction
 - Computer's approach: Step by step through the parallelism
 - Buildup of Datapath
 - Combinational components
 - Architecture without Jump
 - Some example configurations for various instructions
 - Adding in the jump

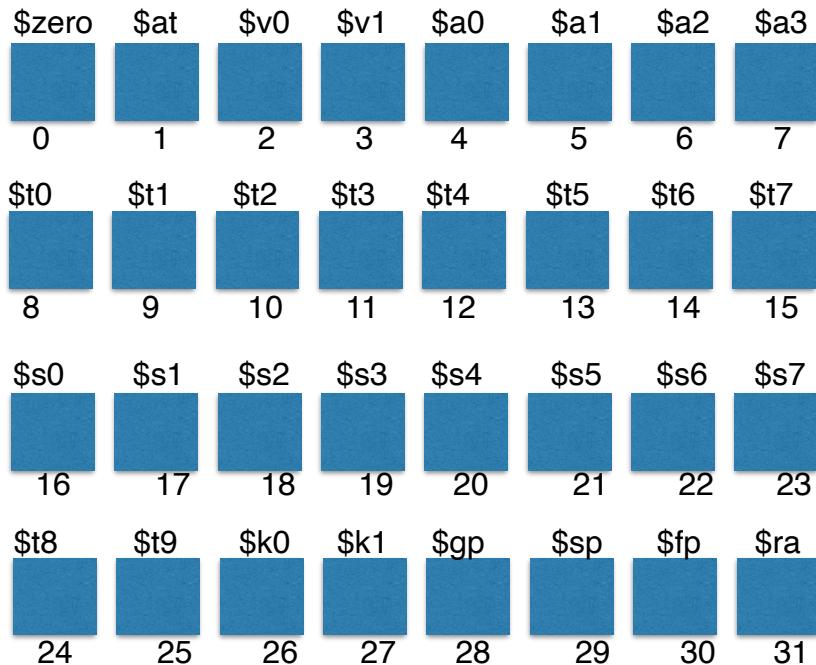
Single Cycle Architecture (SCA): High Level

- Single Cycle Arch (SCA) processes one instruction per clock cycle
- SCA has two types of state: **Data** and **Control**
 - **Data:**
 - 32 **Registers** in **register file**
 - 2^{30} words of **Memory**
 - **Control:** Program Counter (**\$PC**) register (the one reg outside reg file)
 - Control's only jobs:
 - fetch (current) instruction to be processed
 - be told where next instruction is (in memory)

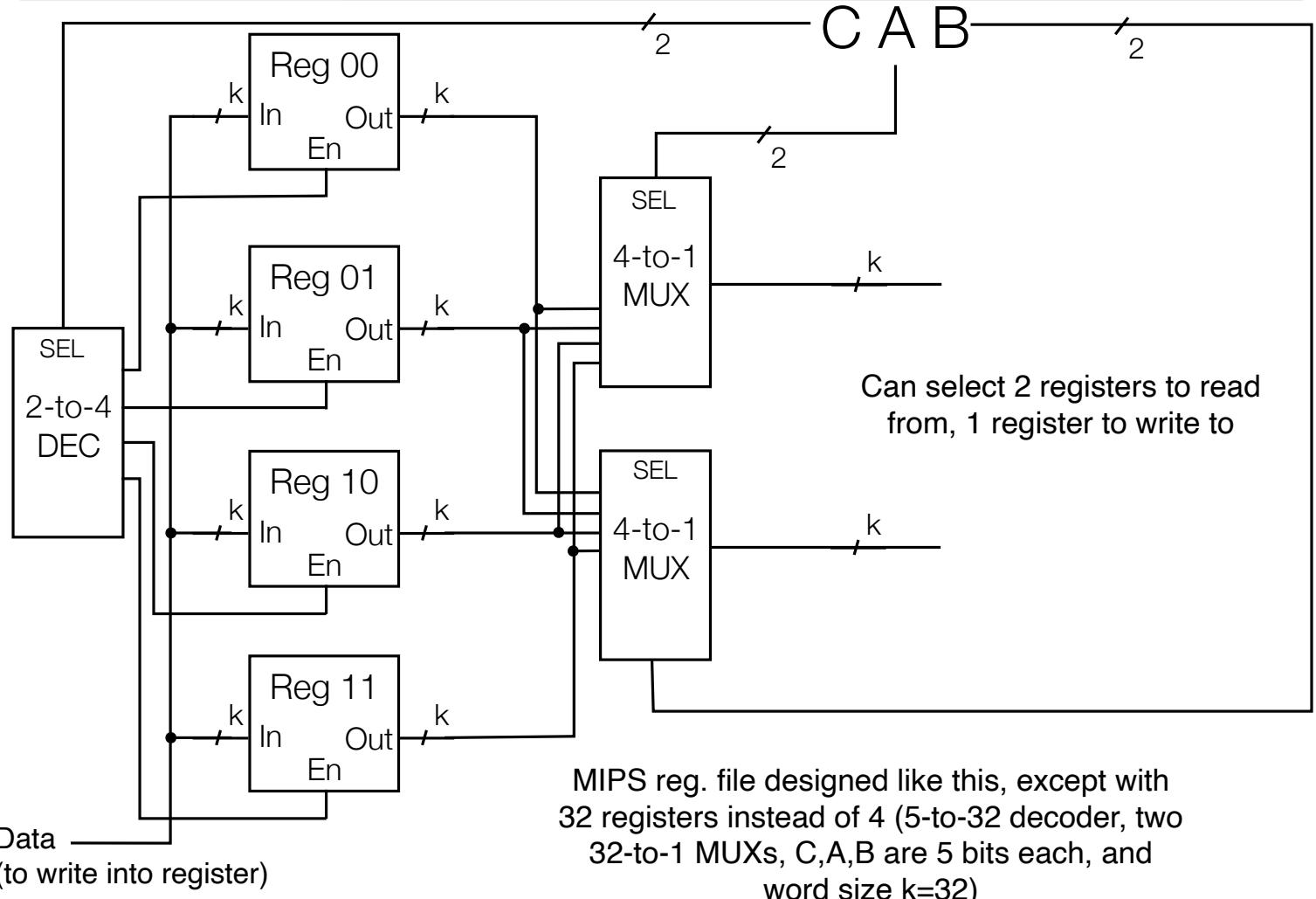
Major Hardware
Components of CSA
(we've already built them)

Recall: Register File & Registers

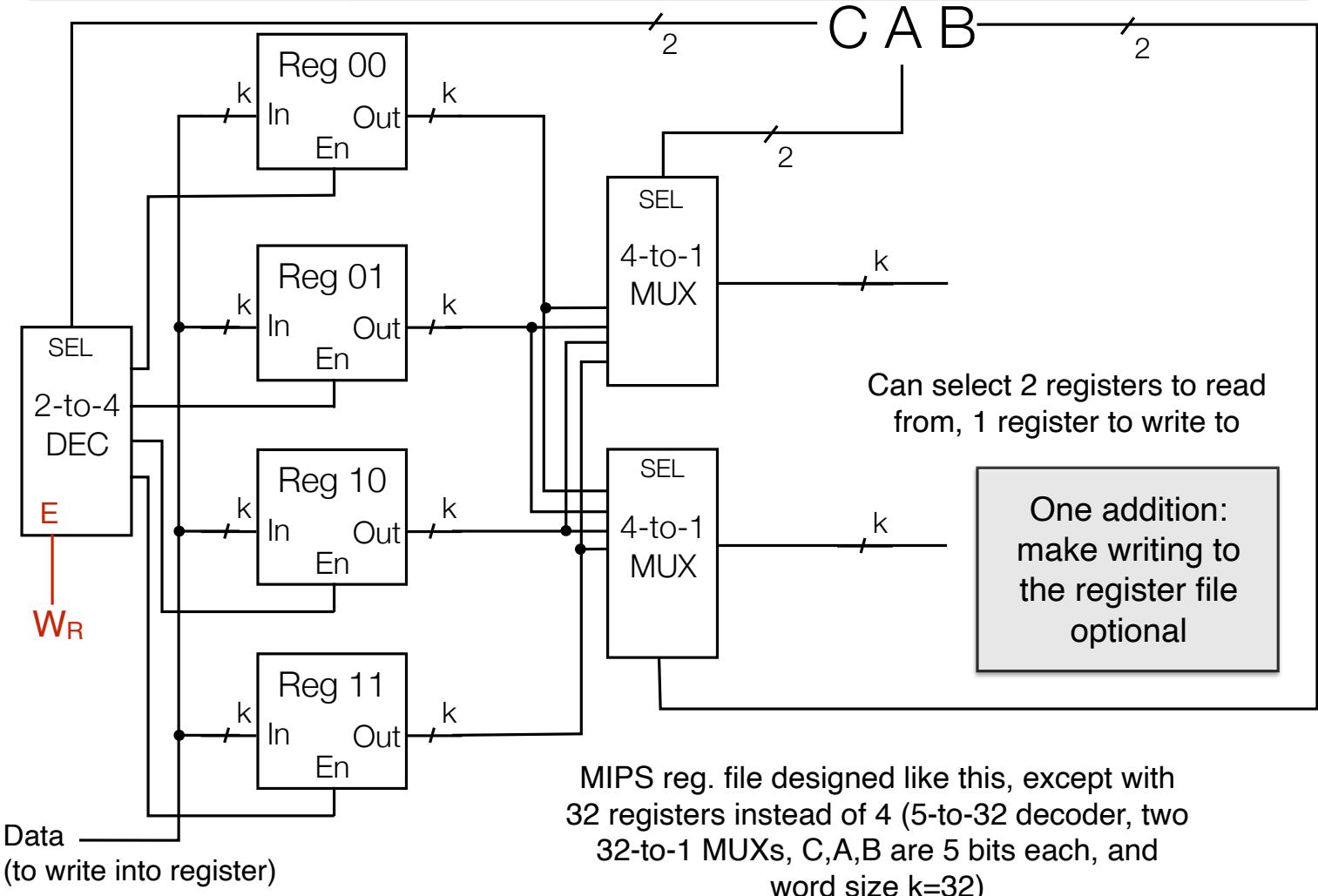
- Each register can hold a word (words size # of bits) of data (over several clock cycles)
- Each register has a unique name (e.g., in MIPS: \$s0, \$s1, \$t1, \$sp, etc.)



Recall: 4-register Register File

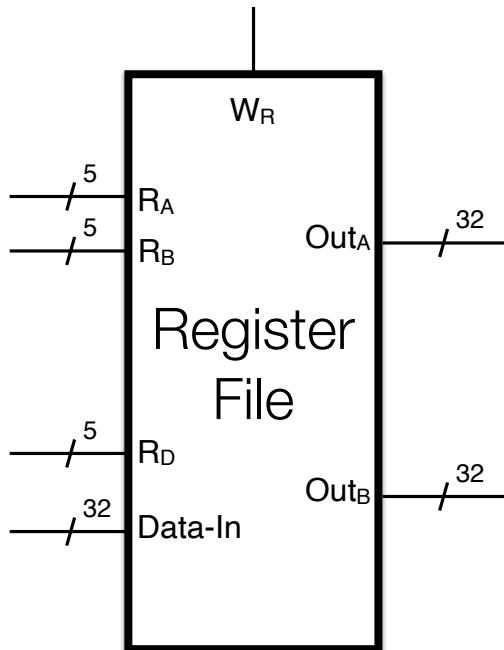


Recall: 4-register Register File

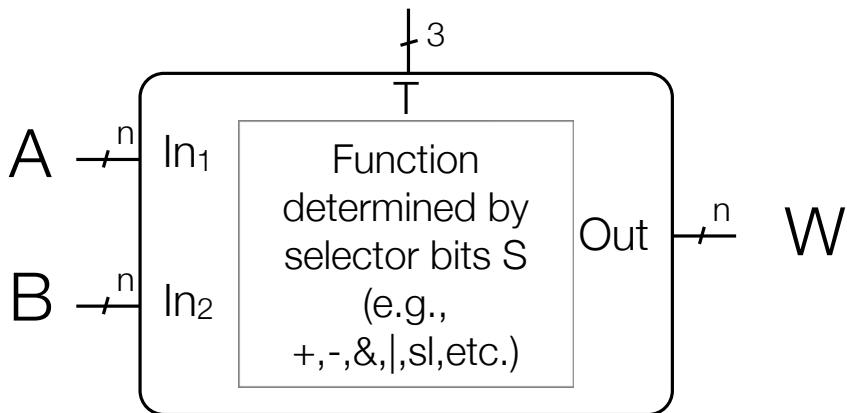


Register File Circuit

- R_A, R_B : 5-bit (unsigned binary) description of registers to read from
- Out_A, Out_B : values in registers described above
- R_D : 5-bit description of a register to write to (if writing)
- W_R : whether or not to write to reg. file this clock cycle (and if so, to R_D)
- Data-In: Data to write into register R_D when $W_R=1$



Recall: (Simplified) MIPS Function Unit (high level)

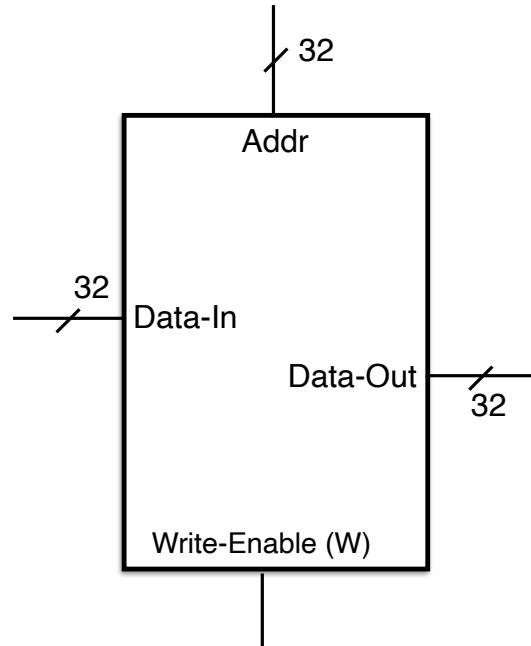


T=XYZ	Output	Comments
"000"	A + B	Add
"001"	A - B	Subtract
"010"	sl A \leftarrow B	Shift left by 5-bit low order bits of
"011"	sr A \leftarrow B	Shift right by 5-bit low order bits of
"100"	A B	logical AND
"101"	A + B	logical OR
"110"	A \oplus B	logical XOR
"111"	A	complement A (ignore B)

- Combinational Circuitry
- OP is the operation determined by 3-bit selector T

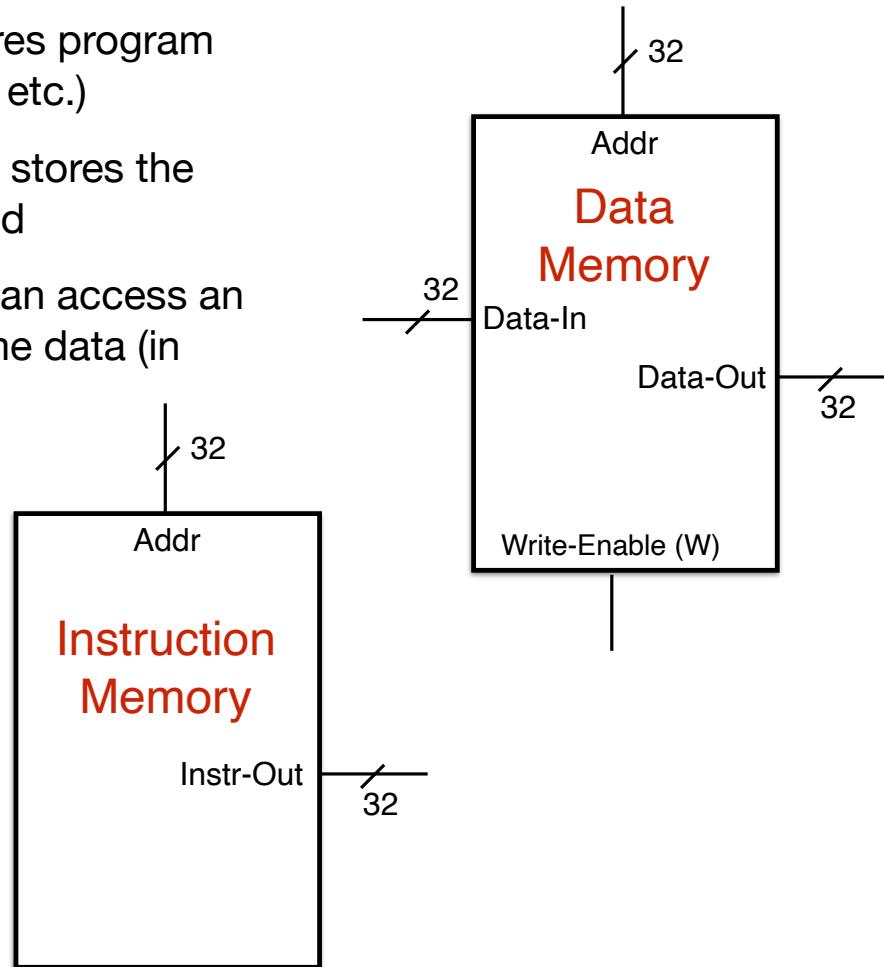
Recall: MIPS Memory

- $k=32$: 32-bit addresses (2^{32} byte-size addressable slots)
- $n=32$: we access memory 32-bits at a time
- 32-bit address always multiple of 4 (2 least significant bits always “00”)



Simplified Arch has 2 memories

- **Data Memory:** stores program data (arrays, stack, etc.)
- **Program Memory:** stores the code to be executed
- Each clock cycle, can access an instruction and some data (in memory) in parallel



High-Level Description

CPU's task in each clock cycle

- In each clock cycle CPU does the following:
 - **Fetch** current instruction
 - located in memory
 - address A_{cur} of current instruction is in \$PC register
 - Determine **default next instruction** (i.e. $A_{cur} + 4$)
 - **Process** the current instruction, which:
 - **Reads** in Data (from register file, memory, or embedded in instruction)
 - **Computes**: uses ALU to perform computation on data read in
 - Maybe **Writes** Data (to either a register in register file or to memory)
 - Maybe **Changes** next instruction (i.e., value in \$PC altered)

The final arch...

- Lots of possible ways data can “flow” through this big (sequential) circuit
 - Clock cycle is fixed. Must be slow enough give enough time for data to flow all the way through the longest (deepest) path
 - Moral: To make the computer run fast (high clock speed), keep the depth of the longest path as short as possible

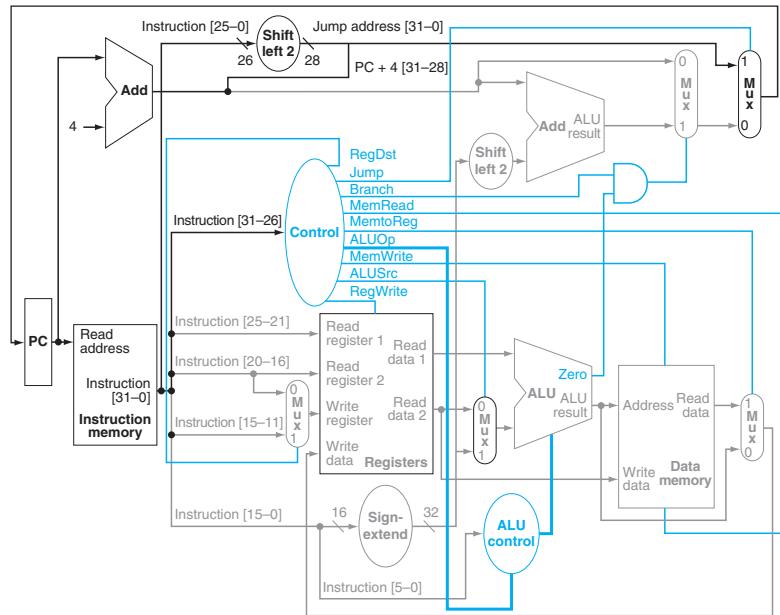


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexer (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexer is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

The final arch - components

- **Program Counter:** register that “points” to current address in instruction memory

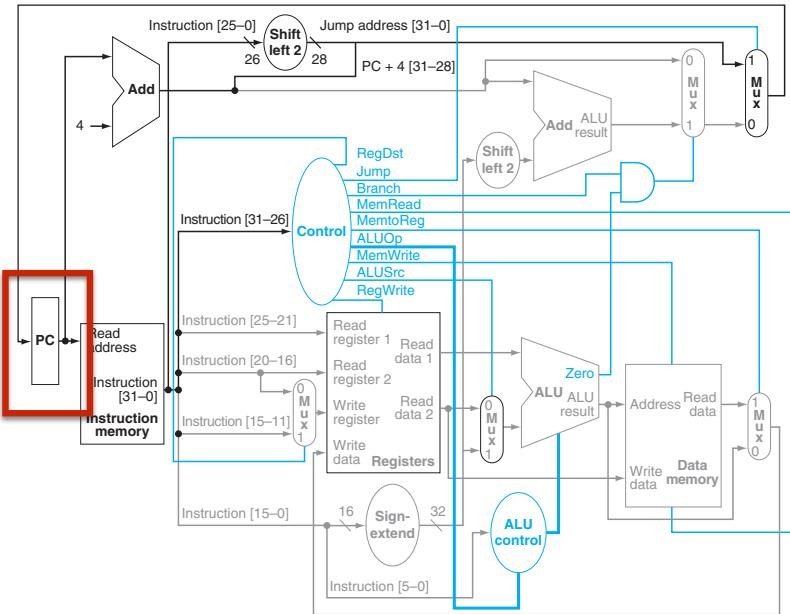


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexer (at the upper right) is used to choose between the jump target or either the branch target or the sequential instruction following this one. This multiplexer is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

The final arch - components

- **Instruction Memory:** memory that holds the program

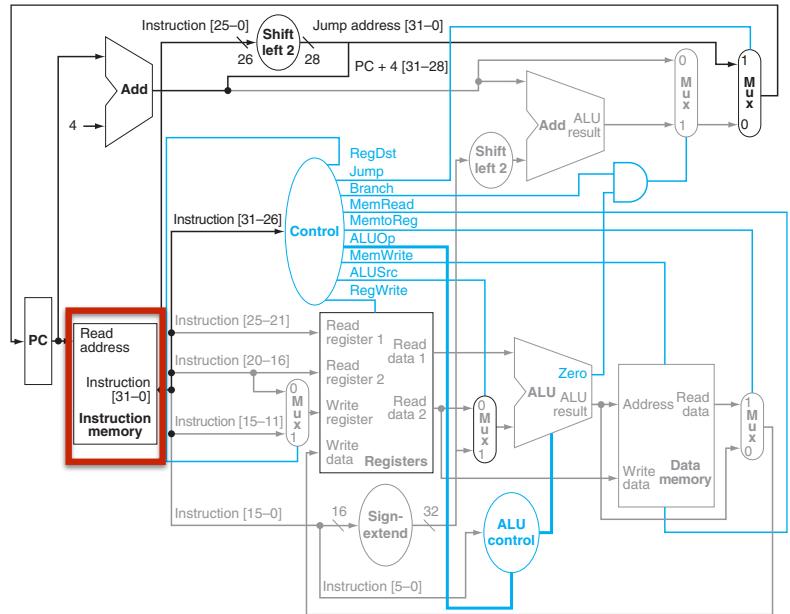


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target or either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

The final arch - components

- **Register File:** holds the registers that are read from and written to

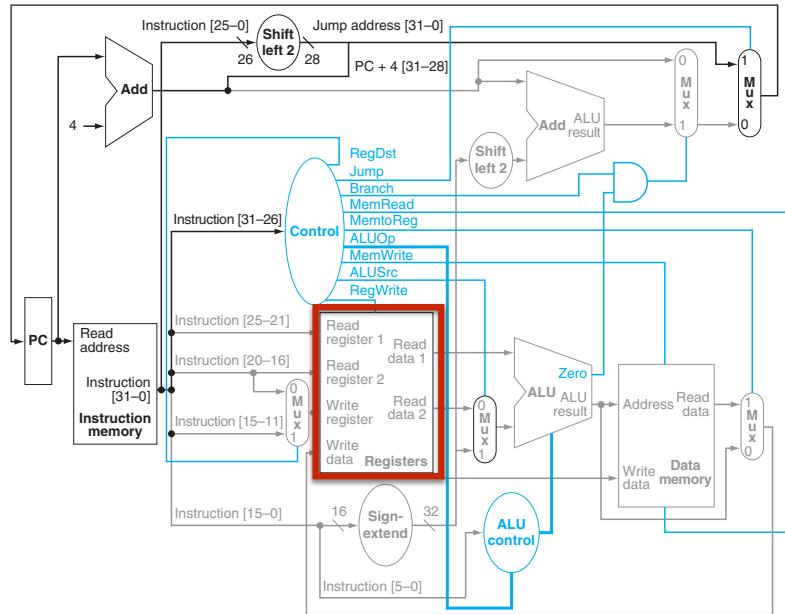


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexer (at the upper right) is used to choose between the jump target or either the branch target or the sequential instruction following this one. This multiplexer is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

The final arch - components

- **Function Unit (ALU):** performs the computation (+,-,&|,shift, etc.)

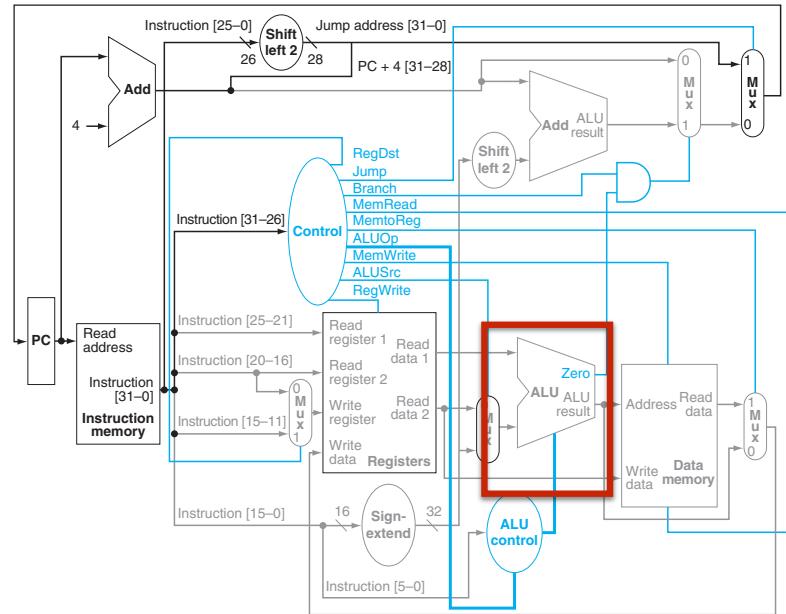


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexer (at the upper right) is used to choose between the jump target or either the branch target or the sequential instruction following this one. This multiplexer is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

The final arch - components

- **Data Memory:** memory where program data (not instructions) are stored

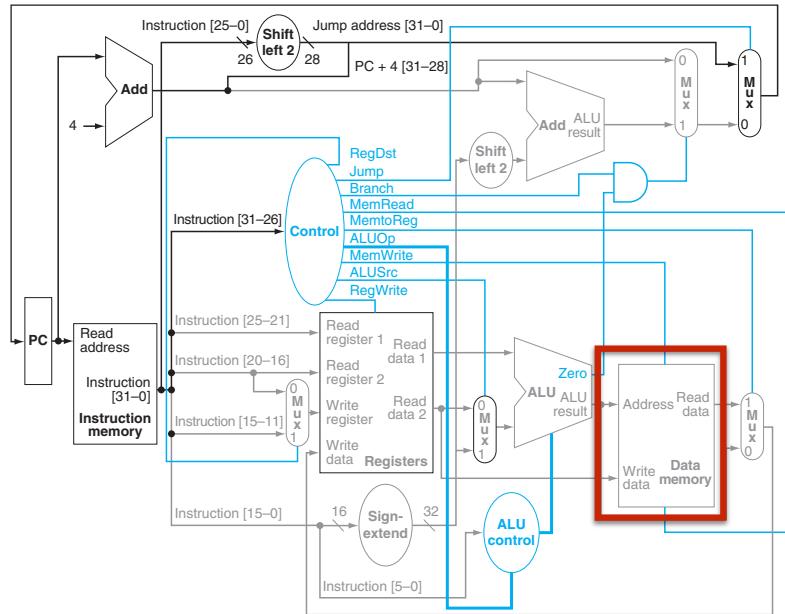


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexer (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexer is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

The final arch - components

- **Instruction-to-Control Word converter:** takes instruction and “figures out” the various control word fields

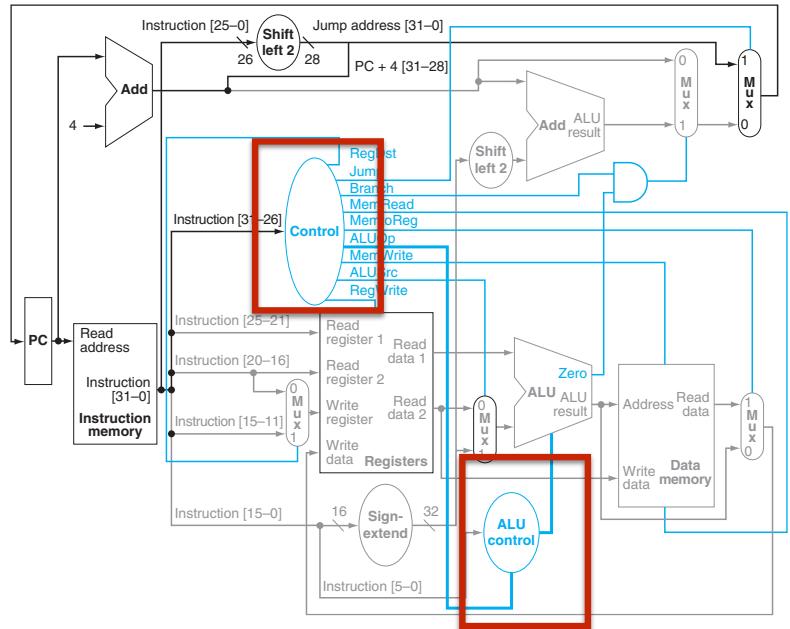


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

The final arch - components

- **Program Counter manipulation:** figures out if a jump or branch should occur (or just +4) and updates PC

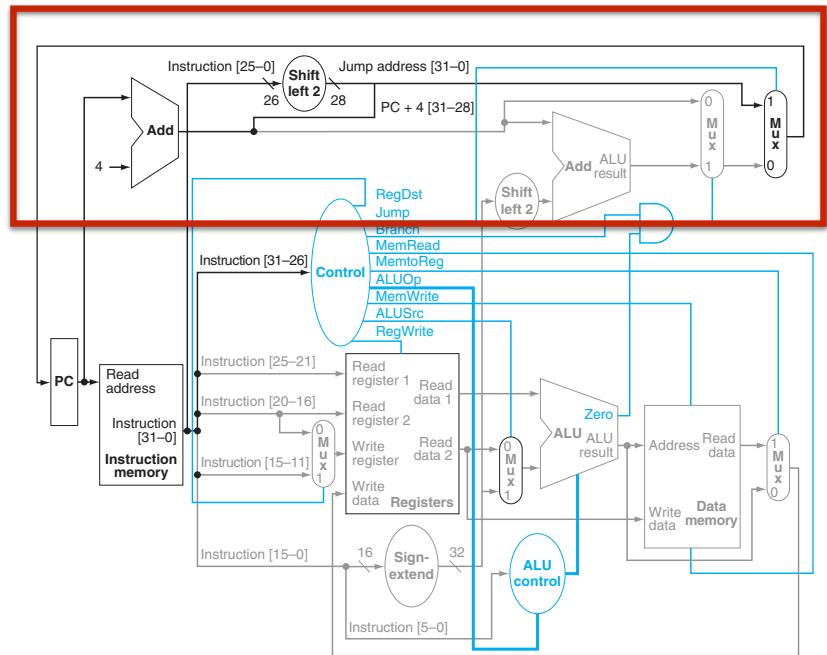


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexer (at the upper right) is used to choose between the jump target or either the branch target or the sequential instruction following this one. This multiplexer is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

The final arch - components

- **MUXs:** Selectors from control word decide what “flows through” the circuit and what gets dropped

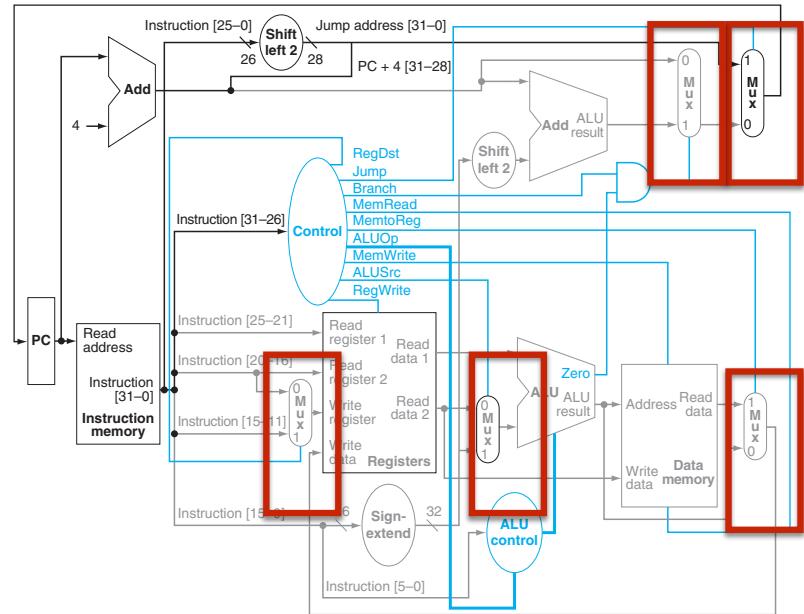


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

The final arch...

- Lots of possible ways data can “flow” through this big (sequential) circuit
- Clock cycle is fixed. Must be slow enough give enough time for data to flow all the way through the longest (deepest) path
- Moral: To make the computer run fast (high clock speed), keep the depth of the longest path as short as possible
- Let’s look at 2 long paths...
 - register 2’s route
 - PC modify route

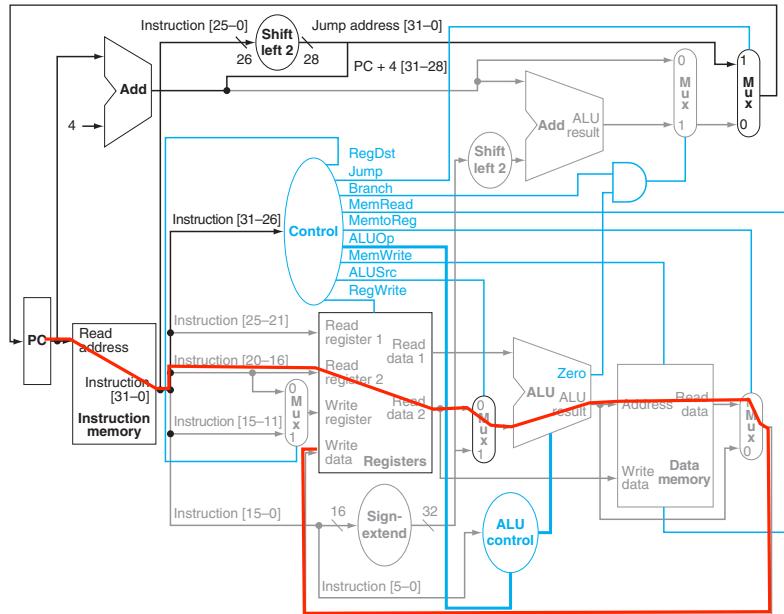


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

The final arch...

- Lots of possible ways data can “flow” through this big (sequential) circuit
- Clock cycle is fixed. Must be slow enough give enough time for data to flow all the way through the longest (deepest) path
- Moral: To make the computer run fast (high clock speed), keep the depth of the longest path as short as possible
- Let’s look at 2 long paths...
 - register 2’s route
 - PC modify route

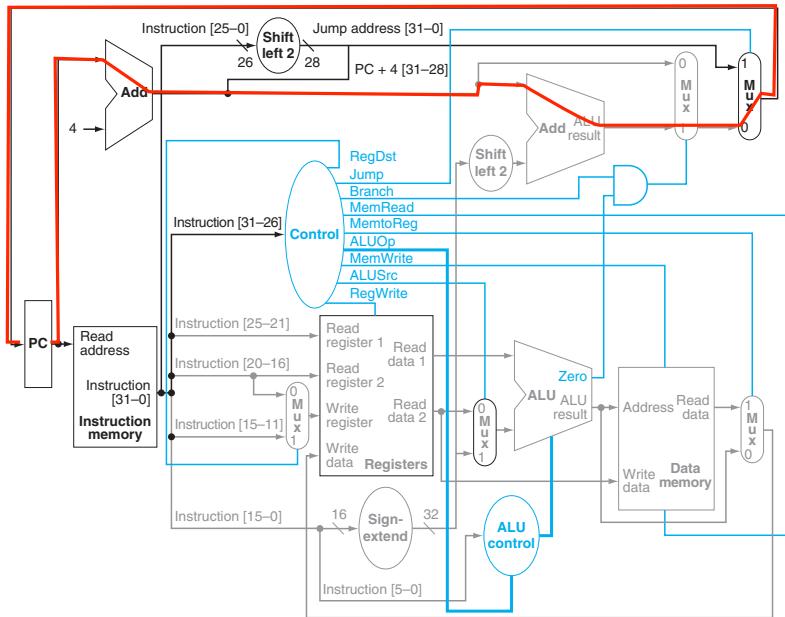


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexer (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexer is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

DataPath

CPU has Data Path and Control Path

Control Path: Determines value of PC in next clock cycle
(we'll talk about this later...)

Data Path: enables the current instruction to user and alter data state

- Instructions can read from state
 - e.g., branch conditional computation
- Instructions might change values in
 - register file
 - memory

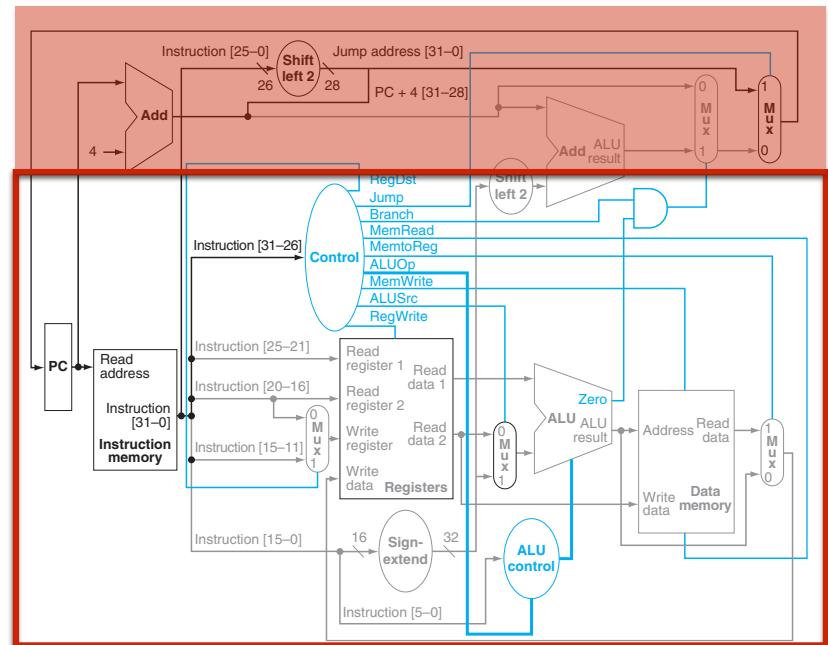


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexer (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexer is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

Data Path

Timeline within a clock cycle

Step I: Fetch Instruction

The final arch - components

- **Program Counter:** register that “points” to current address in instruction memory

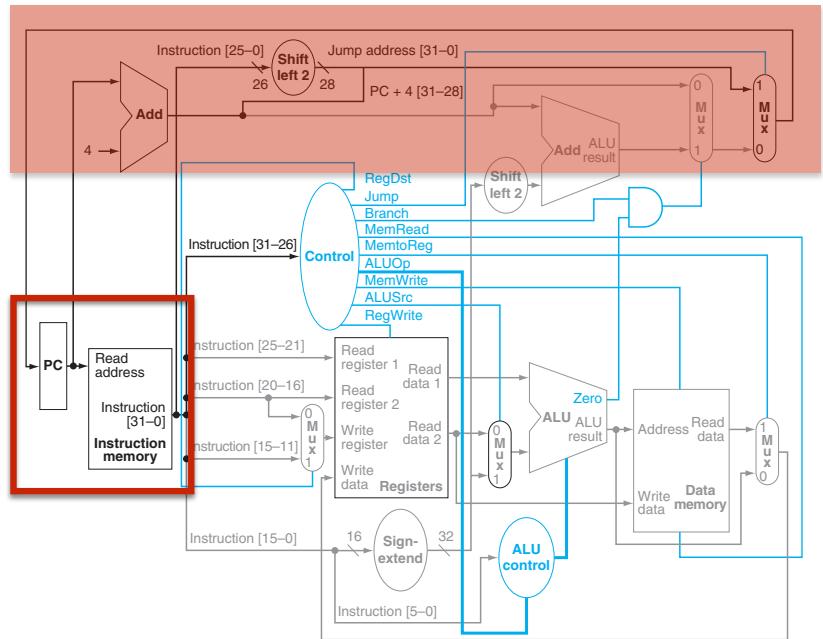


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexer (at the upper right) is used to choose between the jump target or either the branch target or the sequential instruction following this one. This multiplexer is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

Instruction is fetched, PC +=4

- 32 bit instruction retrieved from memory, PC set for next cycle (to next instruction)

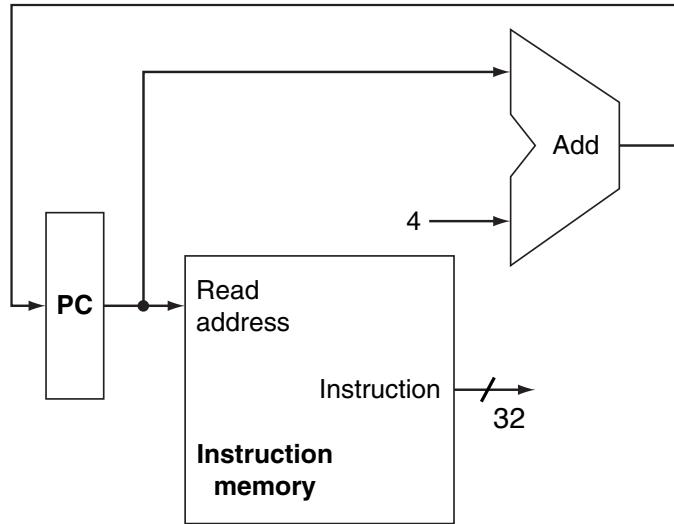


FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath. Copyright © 2009 Elsevier, Inc. All rights reserved.

Instruction is fetched, PC +=4

- 32 bit instruction retrieved from memory, PC set for next cycle (to next instruction)

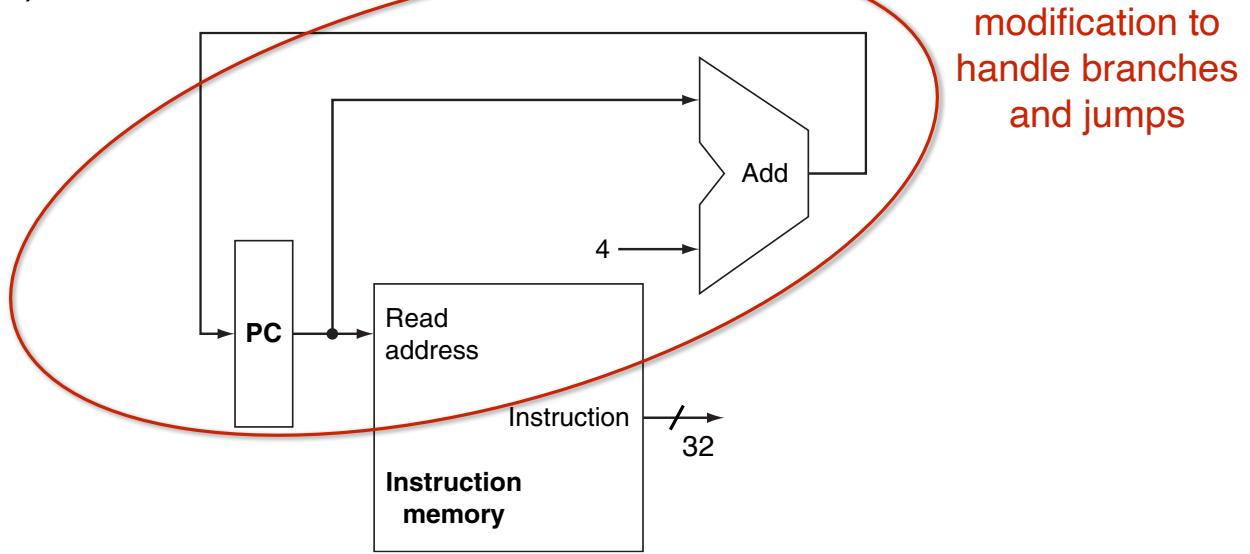


FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath. Copyright © 2009 Elsevier, Inc. All rights reserved.

Step 2: Provide all
parsing of instruction
fields (in parallel)

Computer's Approach (high-level)

- Computer gets 32-bit instruction: does not try to decipher the type of instruction at first
- Instead, processes various possibilities in parallel

XXXXXX10101111000000000101110

- How to process the 32 bits above???

Parse as R-type

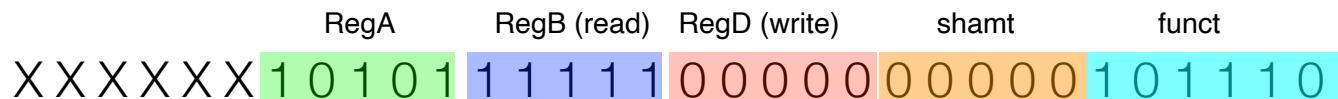
- Assume is **R-type**

RegA	RegB (read)	RegD (write)	shamt	funct
XXXXXX	10101	11111	00000	00000101110

And in Parallel...

Parse as R-type

- Assume is **R-type**



- If is indeed R-type will 1) get values from registers regA and regB, 2) perform funct (if funct is shift, use shamf) on these values, 3) store result in destination reg regD

And in Parallel...

Parse as Branch/I-type/Mem Access

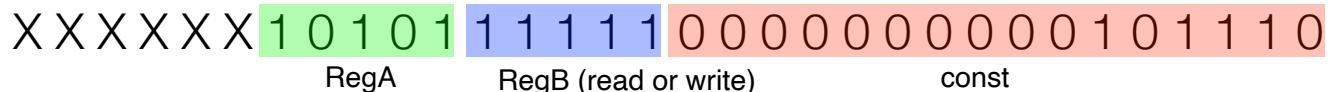
- Assume is I-type, branch, or memory access

XXXXXX 10101 11111 0000000000101110
RegA RegB (read or write) const

And in Parallel...

Parse as Branch/I-type/Mem Access

- Assume is I-type, branch, or memory access



- If is **Branch**: 1) read from both registers, 2) evaluate conditional, 3) fwd conditional result and constant for PC adjustment
- If is **I-type**: 1) read RegA, 2) perform function (specified in opcode) on RegA and constant, store result in RegB
- If is **Mem access**: 2) Get base address (in RegA), add the constant
 - lw**: 3) read from memory at said address, write to dest regB.
 - sw**: 3) write val in regB to memory at specified address

And in Parallel...

Parse as Jump

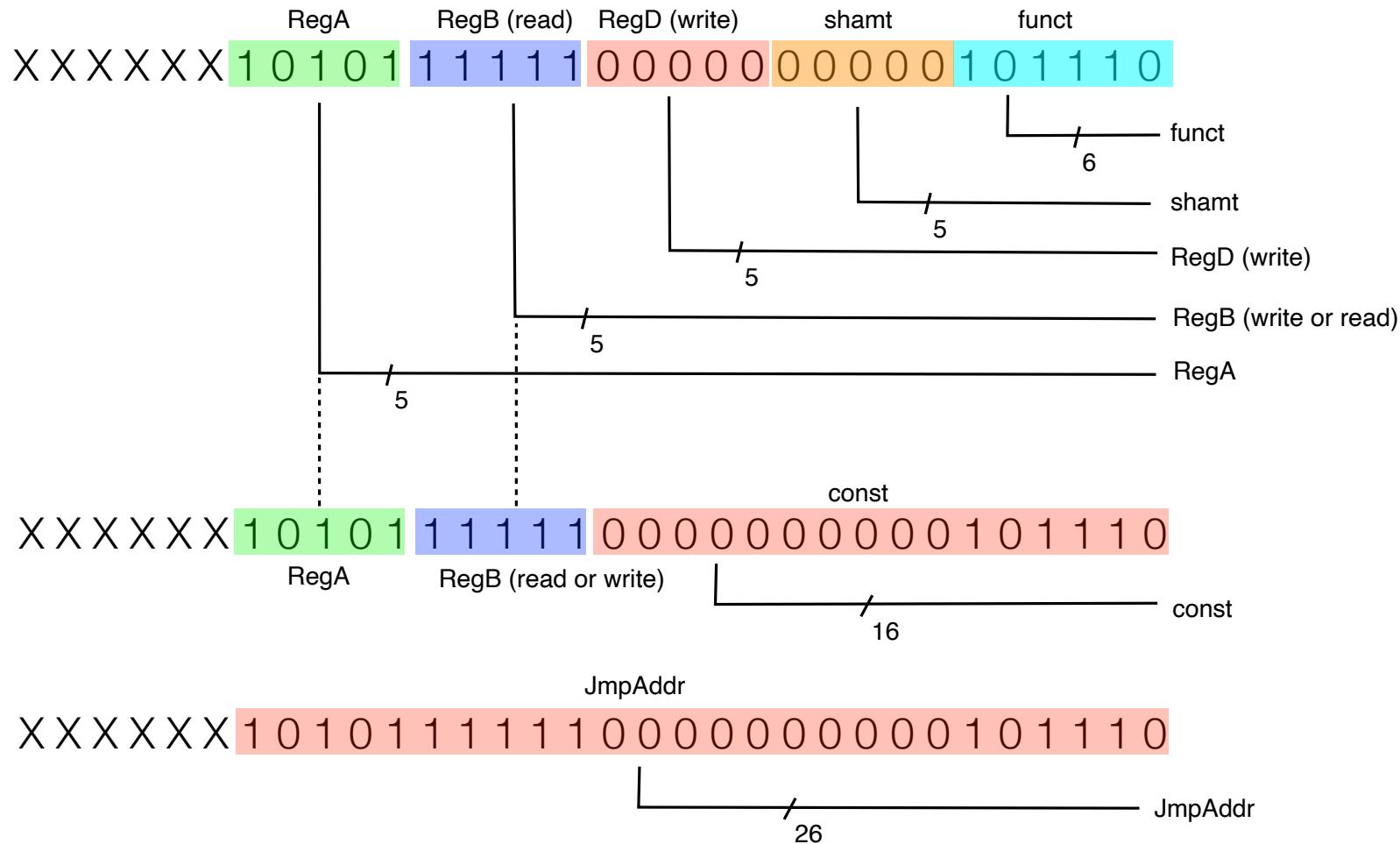
- Assume is **jump**

JmpAddr

XXXXXX **101011110000000000101110**

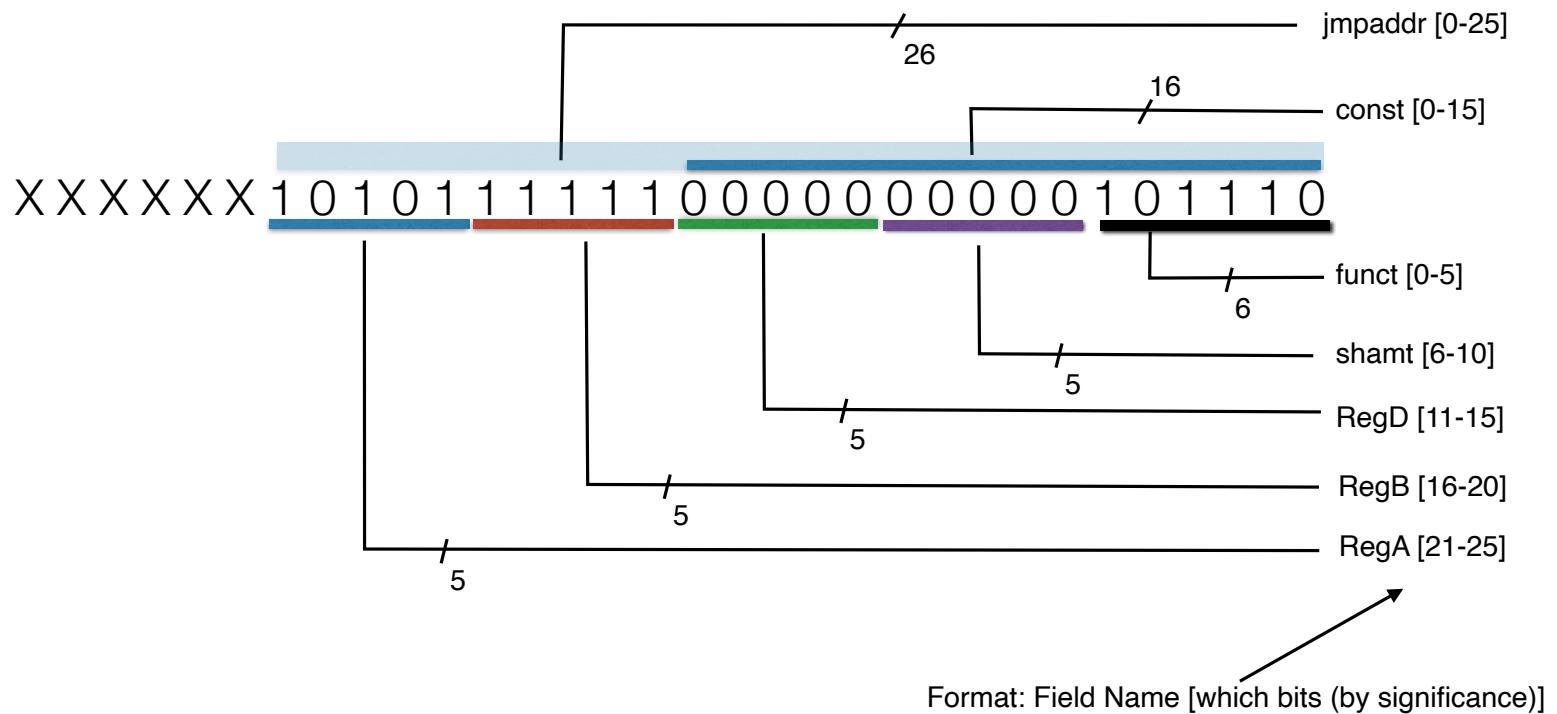
- If is jump, forward JumpAddr to control to determine next address for PC (pseudo-direct)

To Summarize: CPU performs all parsings in parallel



To Summarize: CPU performs all parsings in

Parsings all pulled from same 32-bit copy of instruction



And in Parallel with instruction field parsing...

Opcode Parsing (overloading the term “Control”)

The final arch - components

- **Instruction-to-Control Word converter:** takes instruction and “figures out” the various control word fields

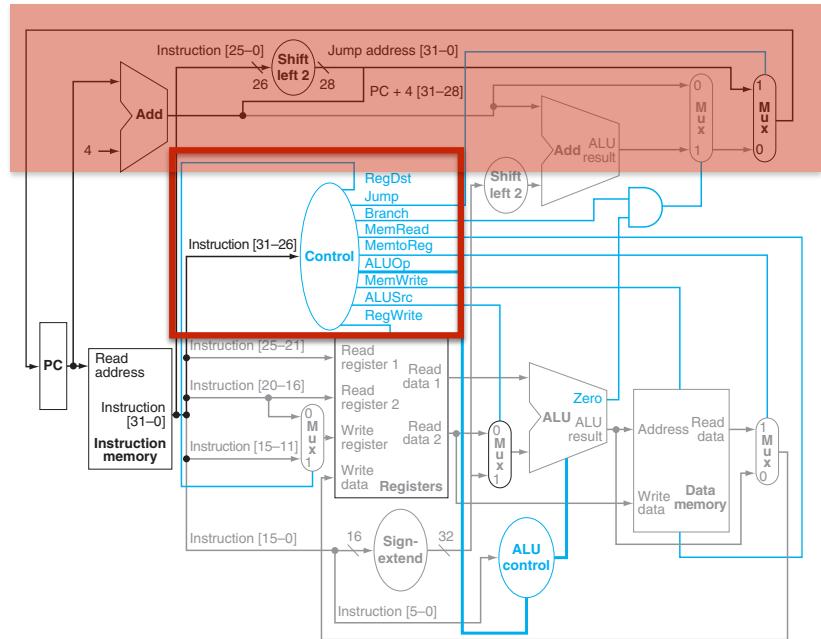
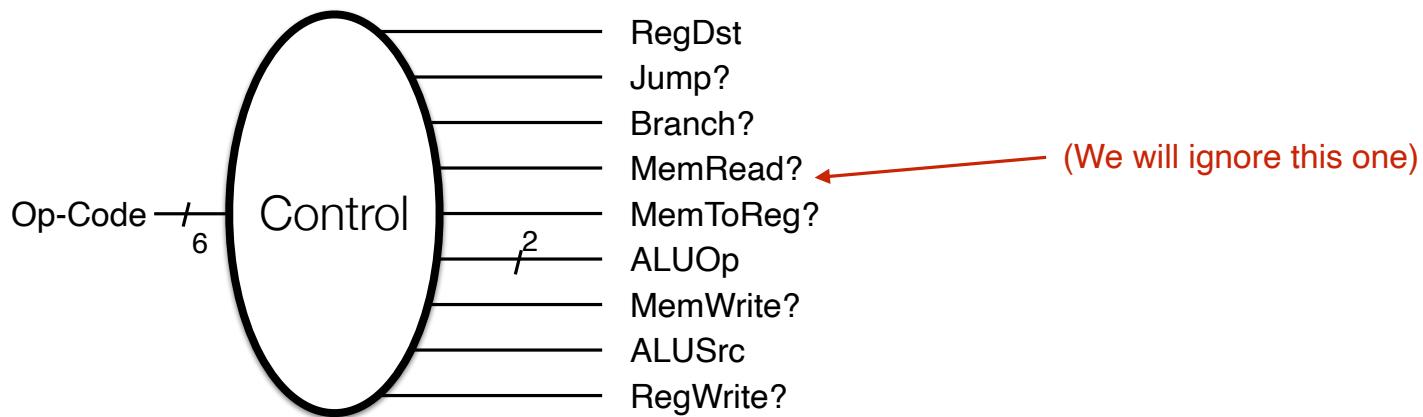


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexer (at the upper right) is used to choose between the jump target or either the branch target or the sequential instruction following this one. This multiplexer is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

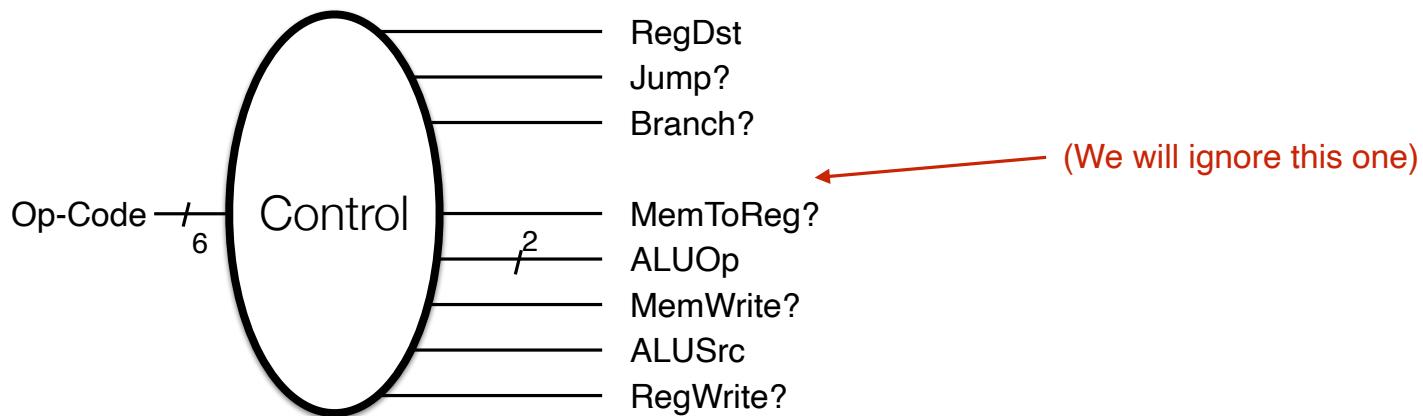
“Control” of the Data Path

- Combinatorial circuitry that maps 6-bit OpCode to various selector fields
- Not to be confused with program “Control” that modifies PC



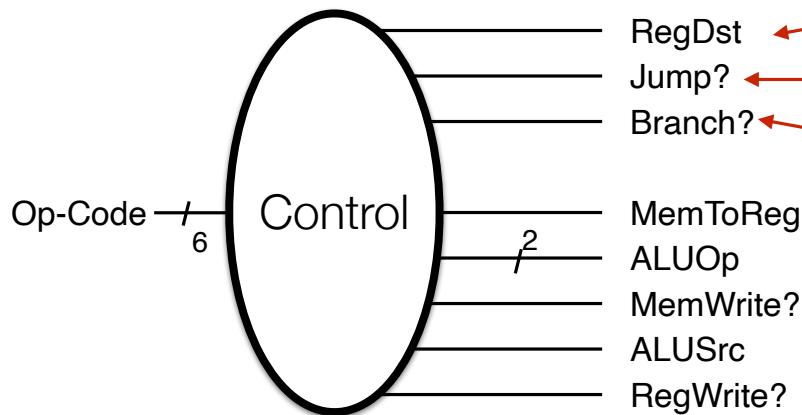
“Control” of the Data Path

- Combinatorial circuitry that maps 6-bit OpCode to various selector fields
- Not to be confused with program “Control” that modifies PC



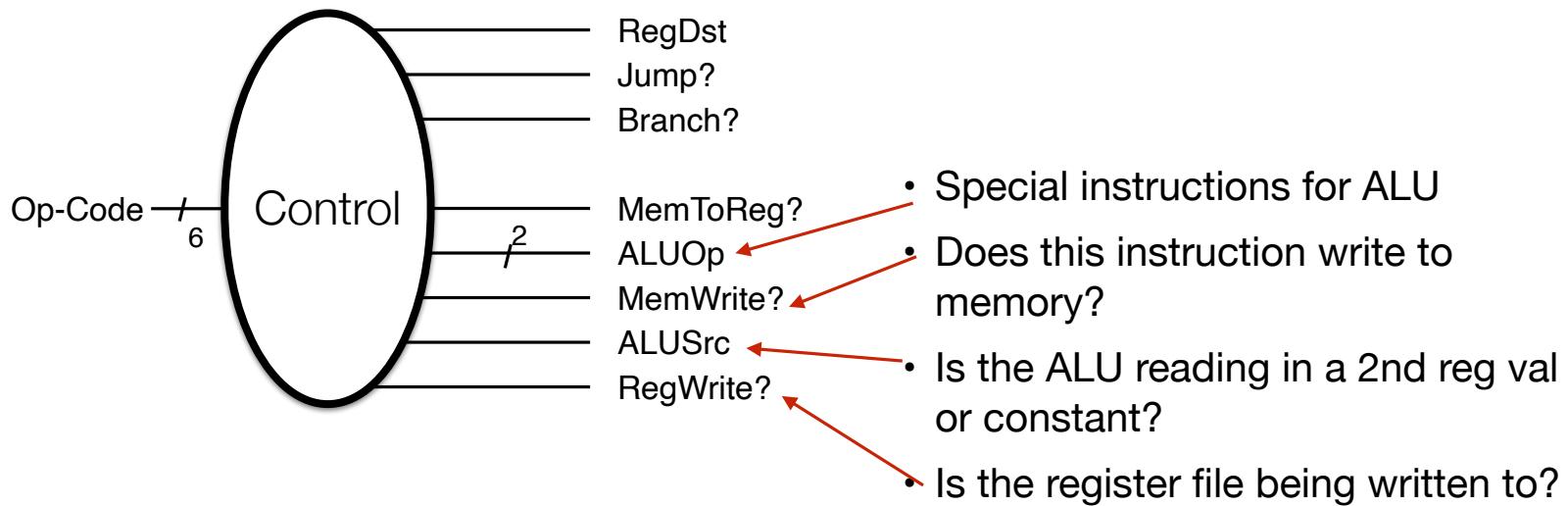
“Control” of the Data Path

- Combinatorial circuitry that maps 6-bit OpCode to various selector fields
- Not to be confused with program “Control” that modifies PC
 - The Opcode determines:
 - Whether RegA or RegD indicates the register to write to
 - Is this a jump instruction?
 - Is this a branch instruction?
 - Will the output of memory be stored in a register (written to reg file)



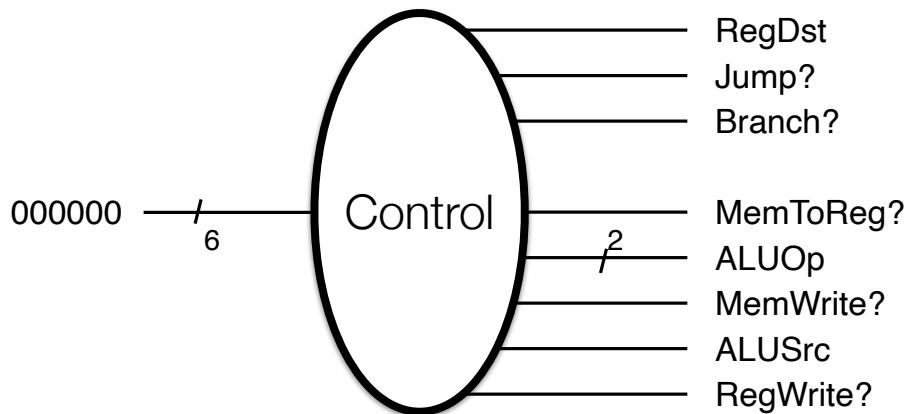
“Control” of the Data Path

- Combinatorial circuitry that maps 6-bit OpCode to various selector fields
- Not to be confused with program “Control” that modifies PC
 - The Opcode determines:



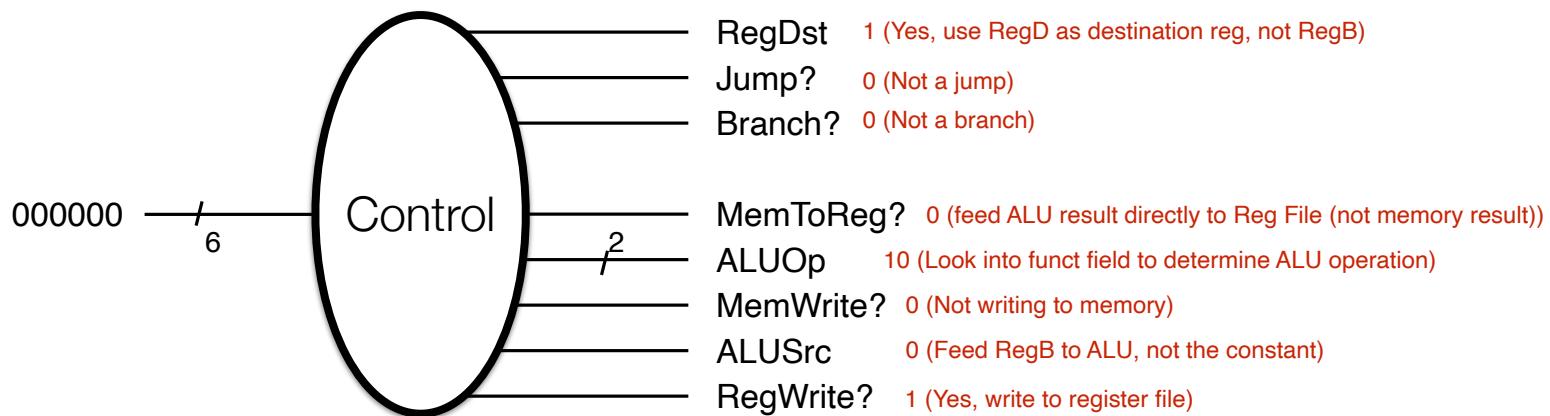
Example 1: R-type instruction

- Recall: Op-code is 000000
- Means read from 2 registers, funct field describes ALU operation, write to RegD register, etc.



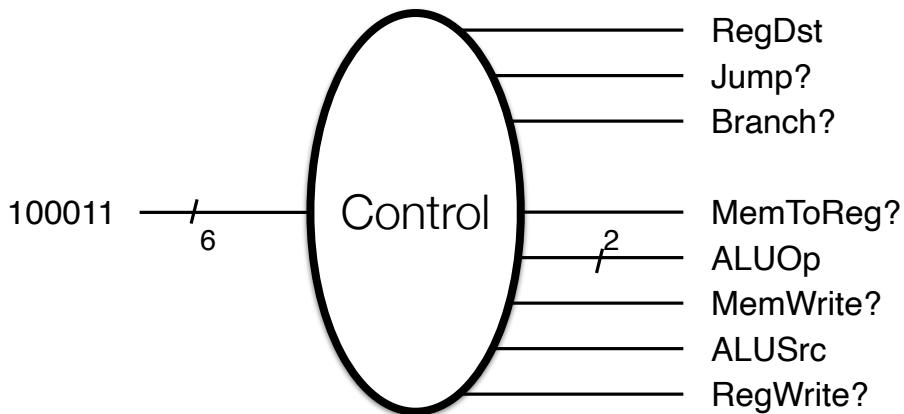
Example 1: R-type instruction

- Recall: Op-code is 000000
- Means read from 2 registers, funct field describes ALU operation, write to RegD register, etc.
- e.g., add \$s1, \$s2, \$s3



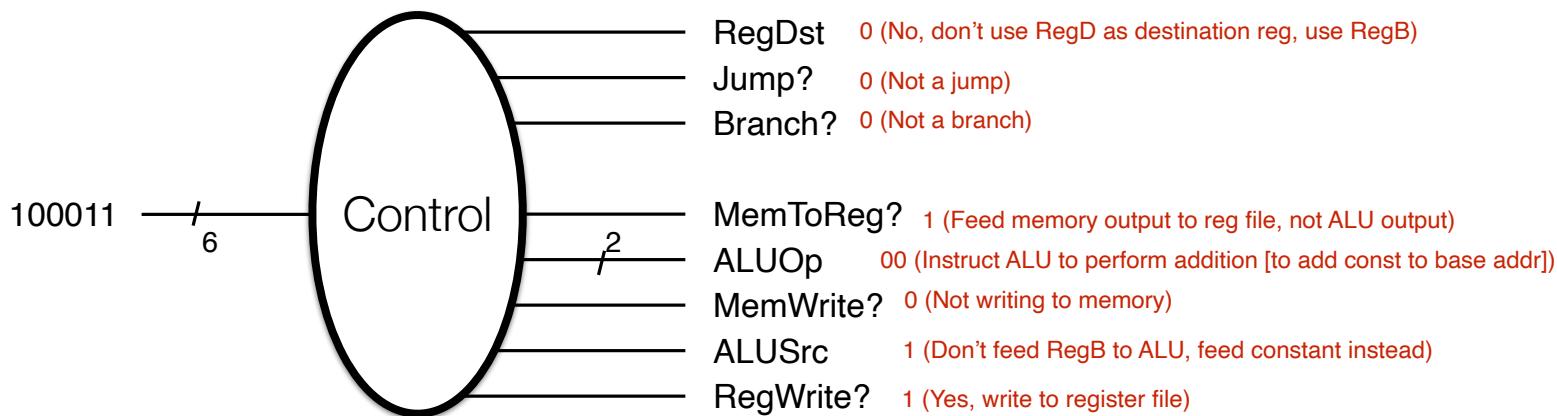
Example 2: lw instruction

- Recall: Op-code is 100011
- Means read from 2 registers, funct field describes ALU operation, write to RegD register, etc.



Example 2: lw instruction

- Recall: Op-code is 000000
- Means read from 2 registers, funct field describes ALU operation, write to RegD register, etc.



Step 3: Read in data
from register file (and
maybe throw away)

The final arch - components

- **Register File:** holds the registers that are read from and written to

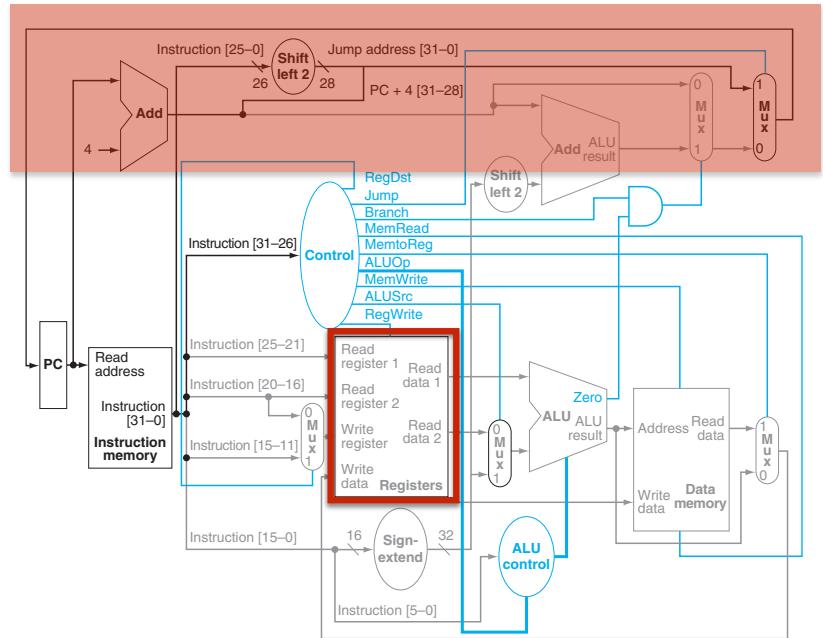


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexer (at the upper right) is used to choose between the jump target or either the branch target or the sequential instruction following this one. This multiplexer is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

Reading from Register File Circuit

- Recall: R_A, R_B : 5-bit (unsigned binary) description of registers to read from

**non-Opcode fields
from instruction**

JmpAddr

RegA

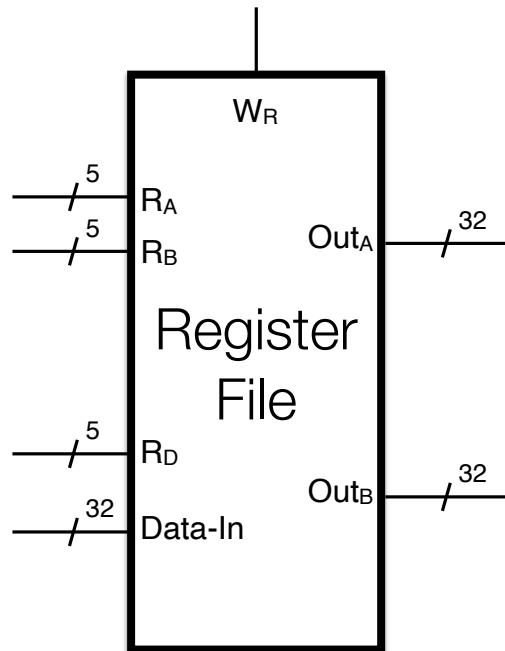
RegB

RegD

shamt

funct

const



Reading from Register File Circuit

- R_A, R_B : 5-bit (unsigned binary) description of registers to read from

**non-Opcode fields
from instruction**

JmpAddr

RegA

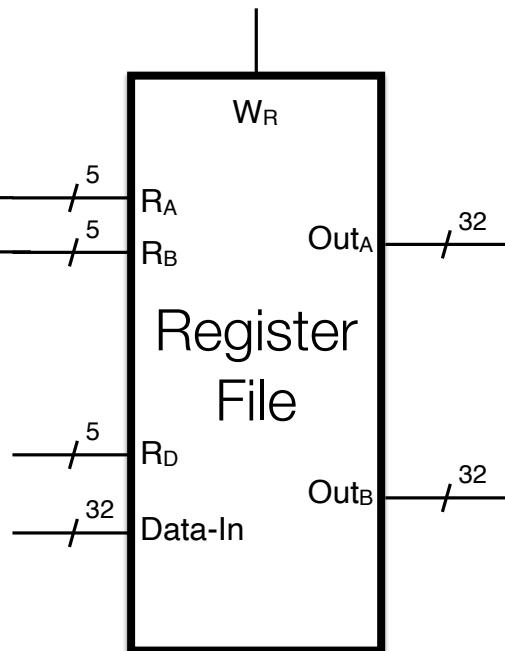
RegB

RegD

shamt

funct

const



Reading from Register File Circuit

- R_A, R_B : 5-bit (unsigned binary) description of registers to read from

**non-Opcode fields
from instruction**

JmpAddr

RegA

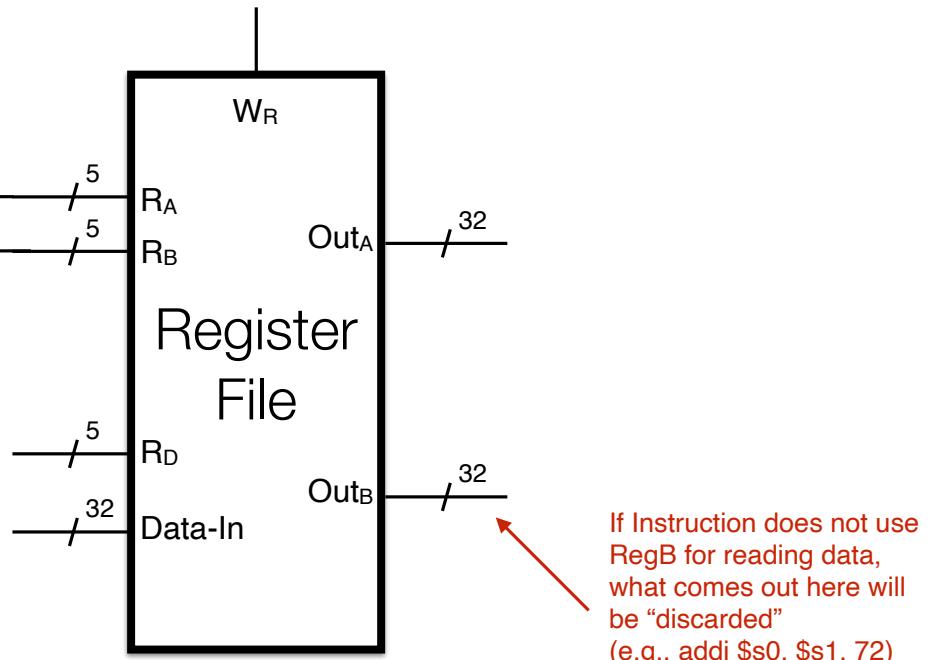
RegB

RegD

shamt

funct

const



If Instruction does not use
RegB for reading data,
what comes out here will
be “discarded”
(e.g., addi \$s0, \$s1, 72)

The final arch - components

- **Register File:** holds the registers that are read from and written to

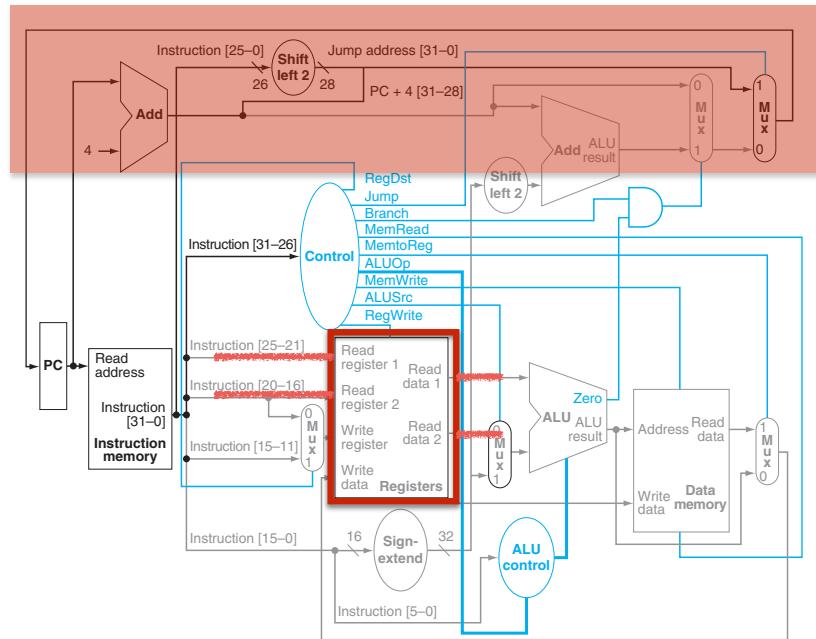


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

Step 4: Apply the ALU

The final arch - components

- **Function Unit (ALU):** performs the computation (+,-,&|,shift, etc.)

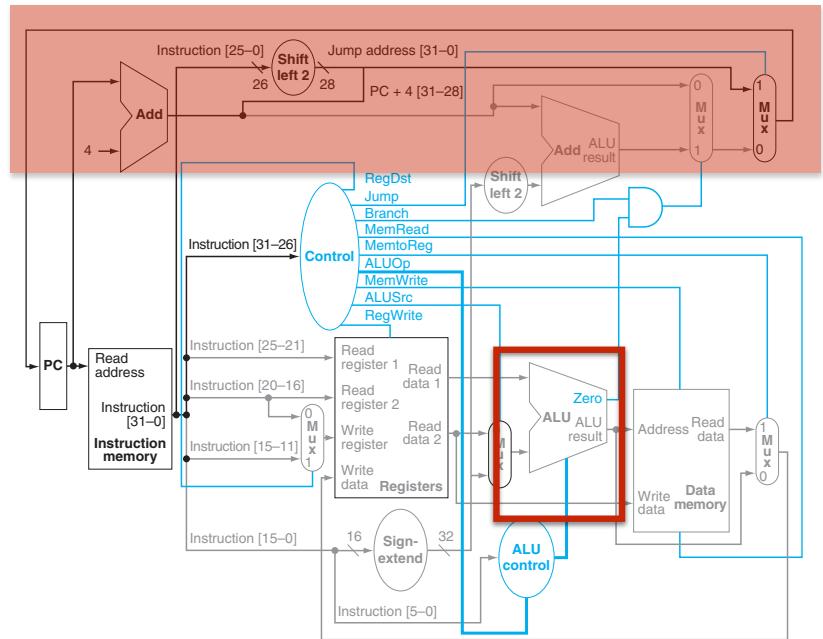
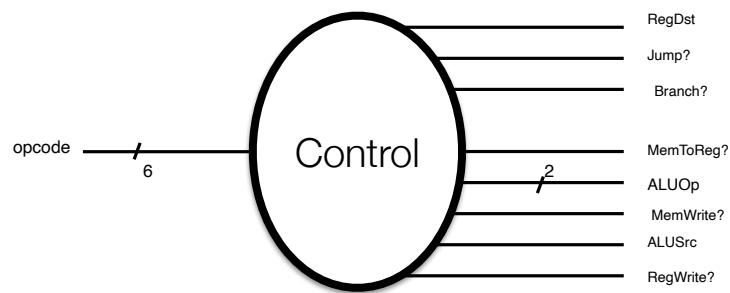
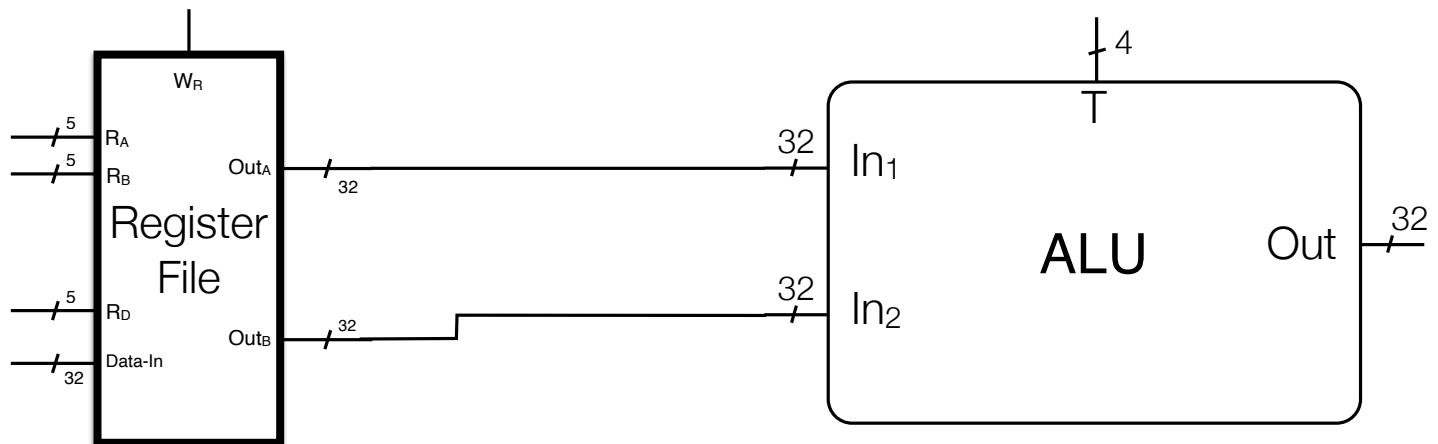


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexer (at the upper right) is used to choose between the jump target or either the branch target or the sequential instruction following this one. This multiplexer is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

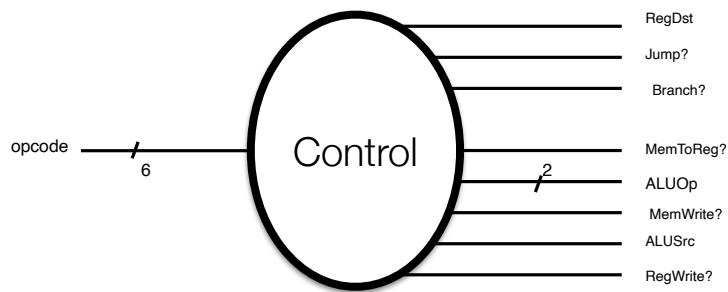
Using the ALU



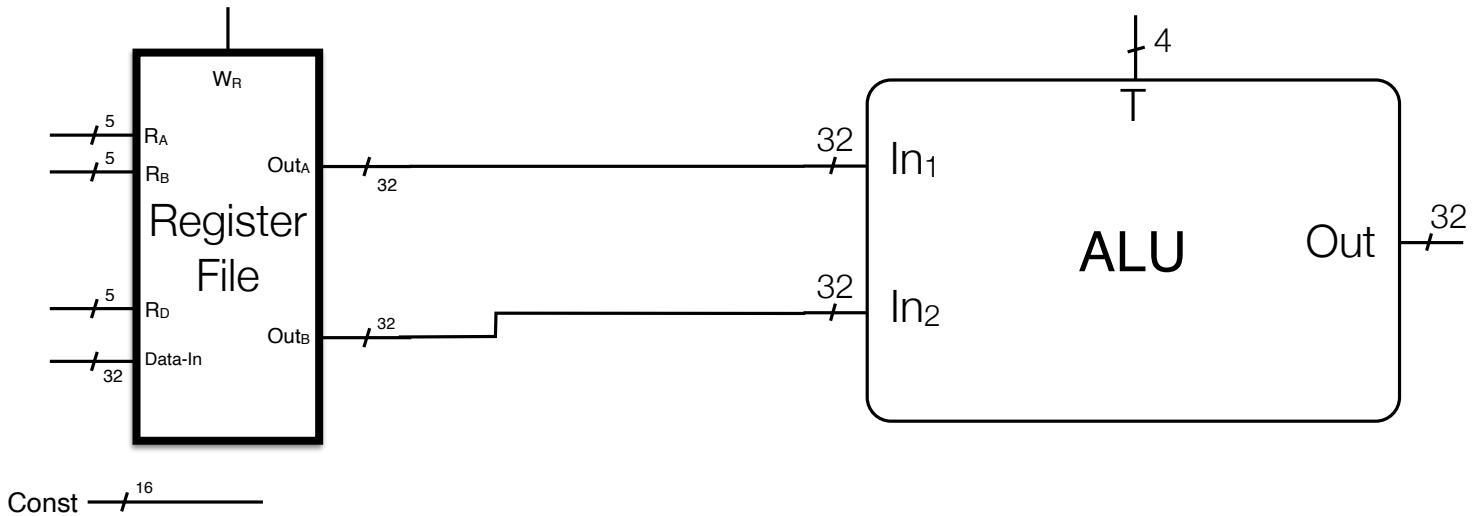
- Some instructions send two register values to the ALU (e.g., R-type, branch)



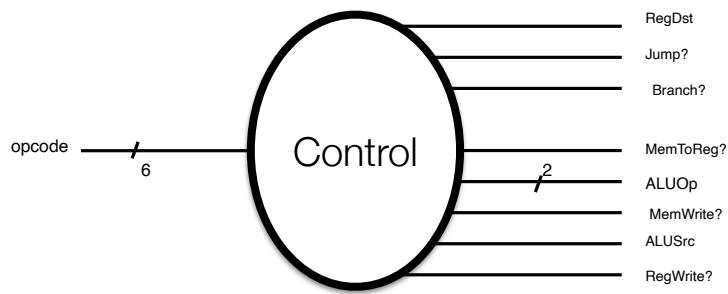
Using the ALU



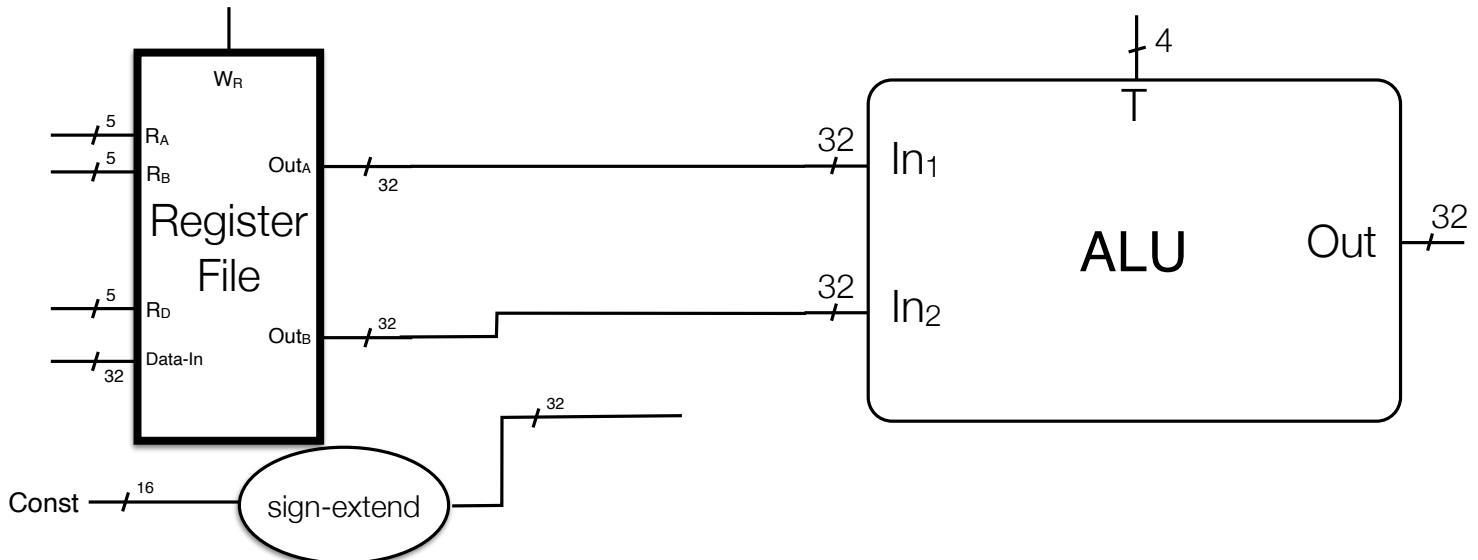
- Some instructions send two register values to the ALU (e.g., R-type, branch)
- Others a register and a constant (e.g., I-type, lw, sw)



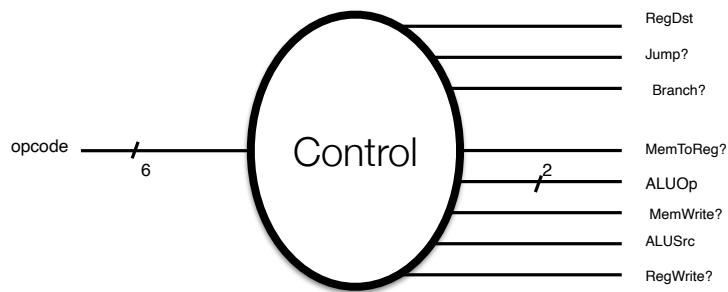
Using the ALU



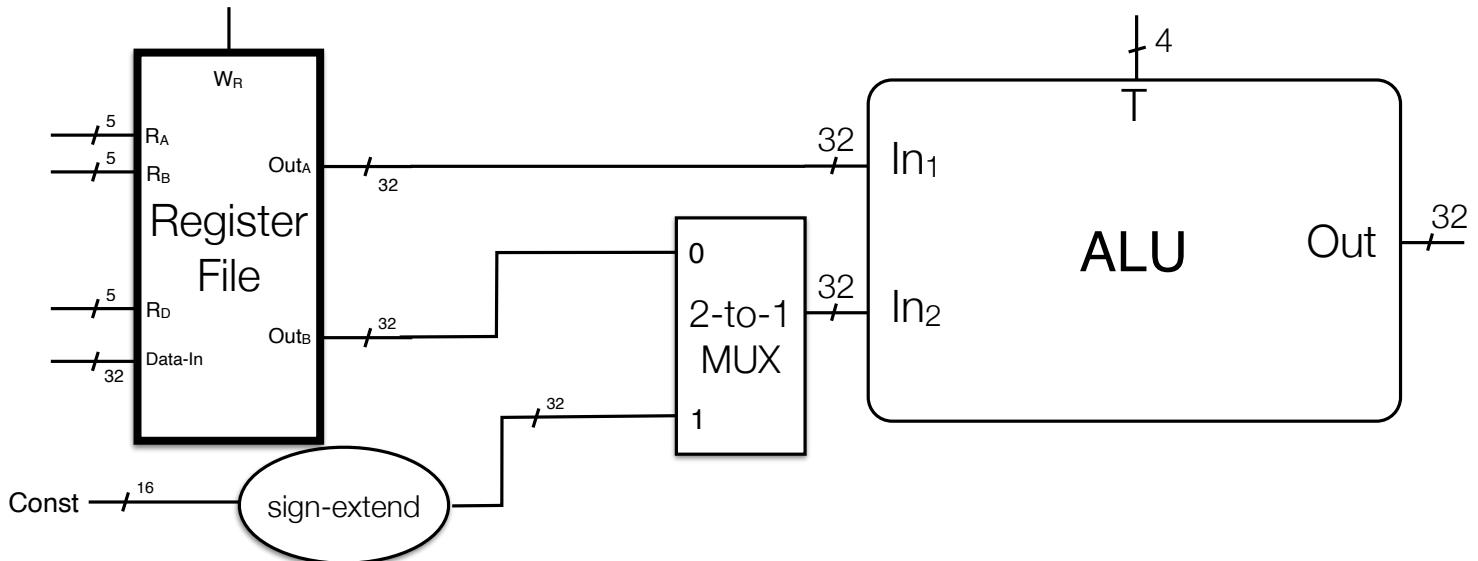
- Some instructions send two register values to the ALU (e.g., R-type, branch)
- Others a register and a constant (e.g., I-type, lw, sw)



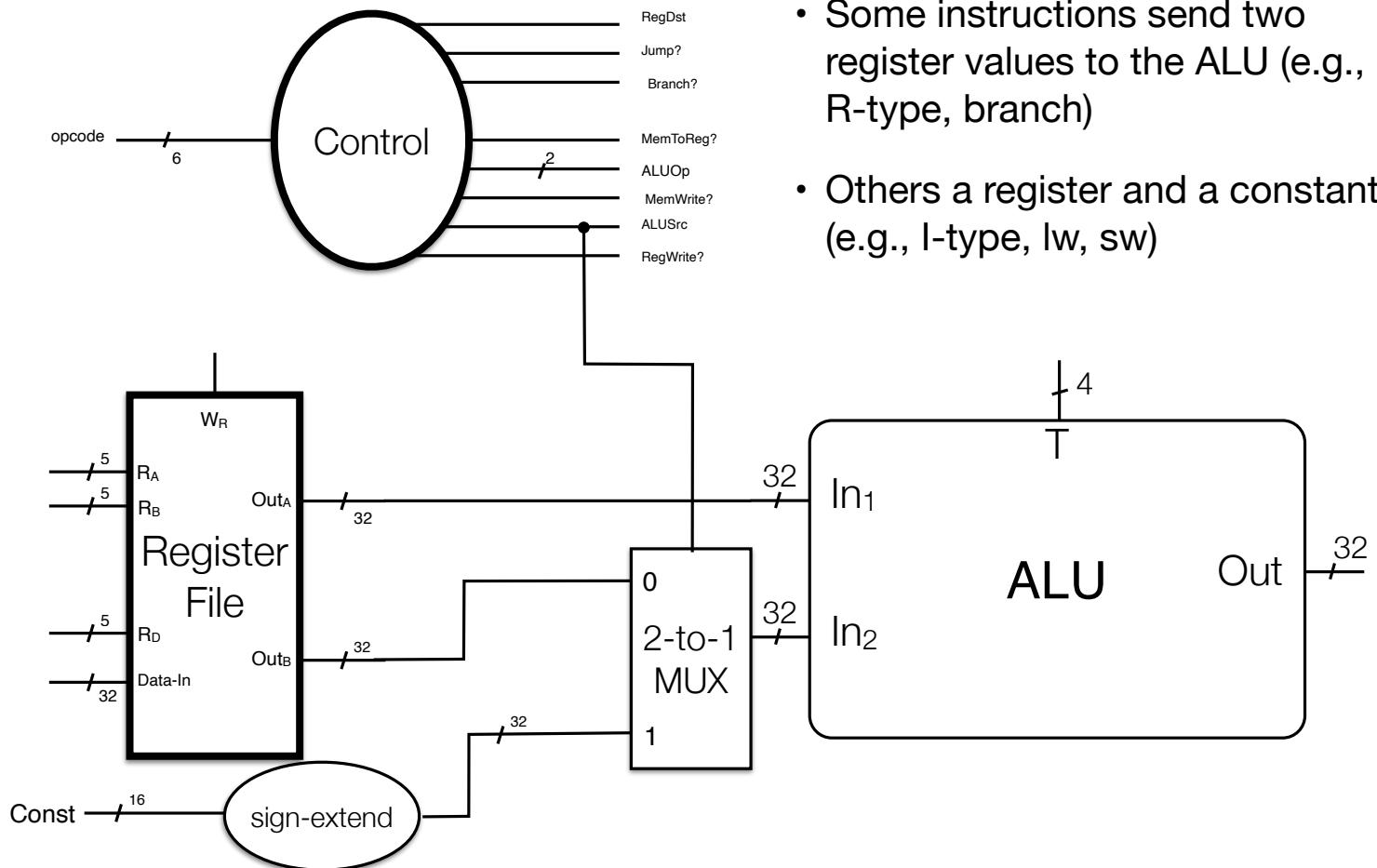
Using the ALU



- Some instructions send two register values to the ALU (e.g., R-type, branch)
- Others a register and a constant (e.g., I-type, lw, sw)

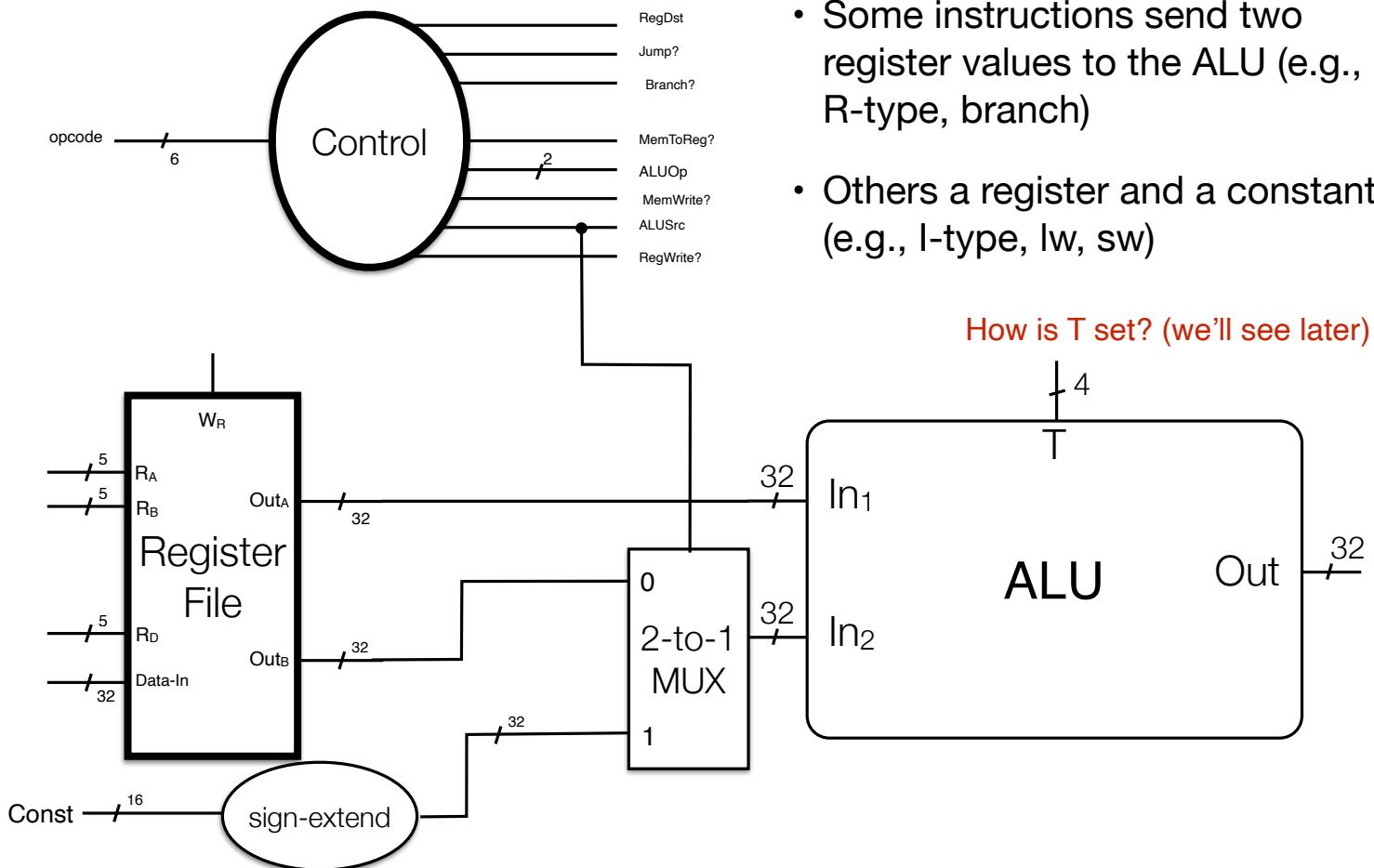


Using the ALU



- Some instructions send two register values to the ALU (e.g., R-type, branch)
- Others a register and a constant (e.g., I-type, lw, sw)

Using the ALU



The final arch - components

- **Function Unit (ALU):** performs the computation (+,-,&|,shift, etc.)

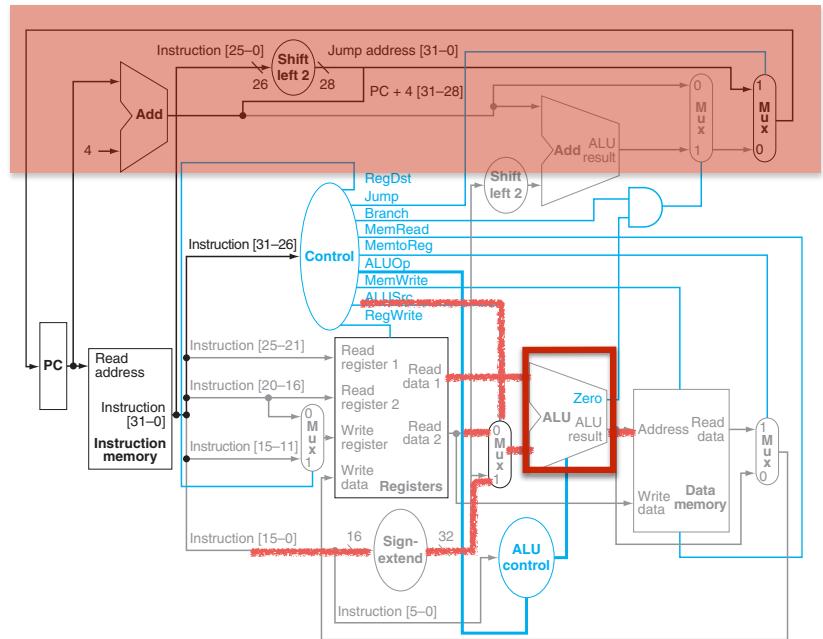


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexer (at the upper right) is used to choose between the jump target or either the branch target or the sequential instruction following this one. This multiplexer is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

No change to state
yet...

Observation

- An instruction only “does something” if it changes state:
 - changes value in a register
 - writes to memory
 - outputs to other device (screen, disk, etc.) - we don’t worry about this
- So until instruction “writes” to reg file or memory, any values read, computations performed have no effect on machine state

Step 5:Access Memory

The final arch - components

- **Data Memory:** memory where program data (not instructions) are stored

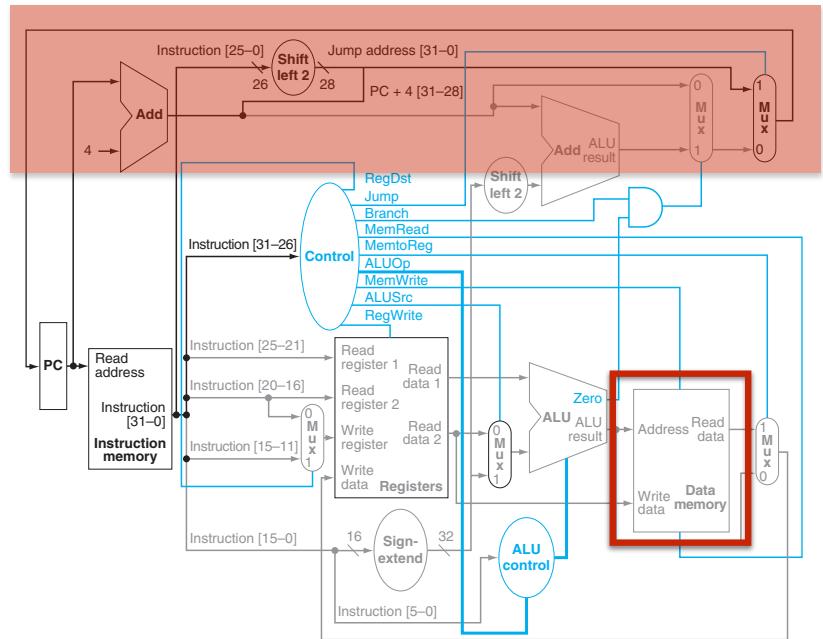
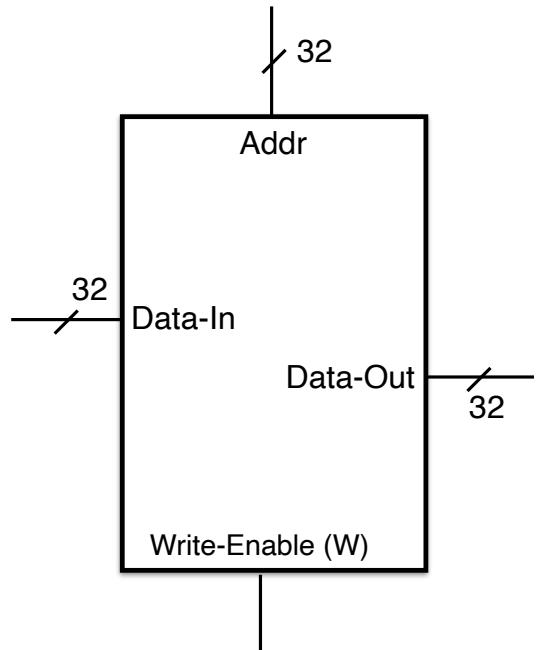


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target or either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

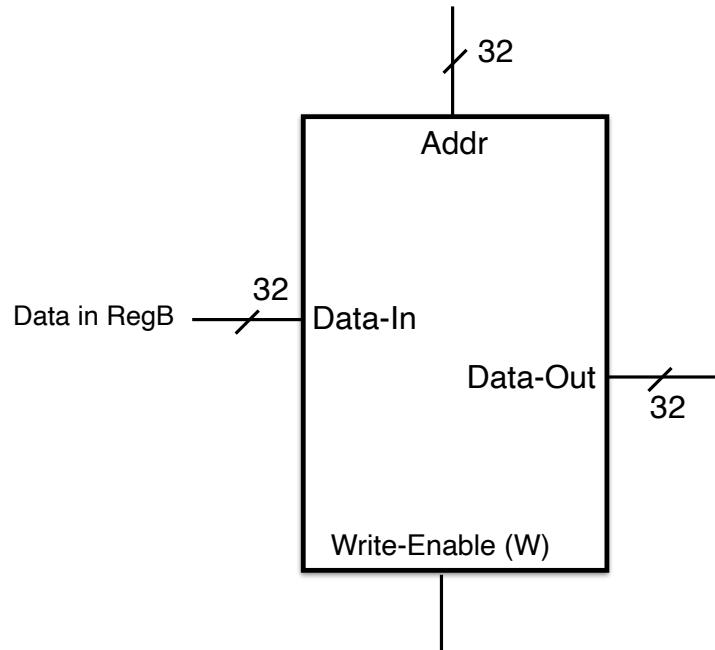
Something useful done here only if lw or sw

- If not lw or sw, memory result will be ignored



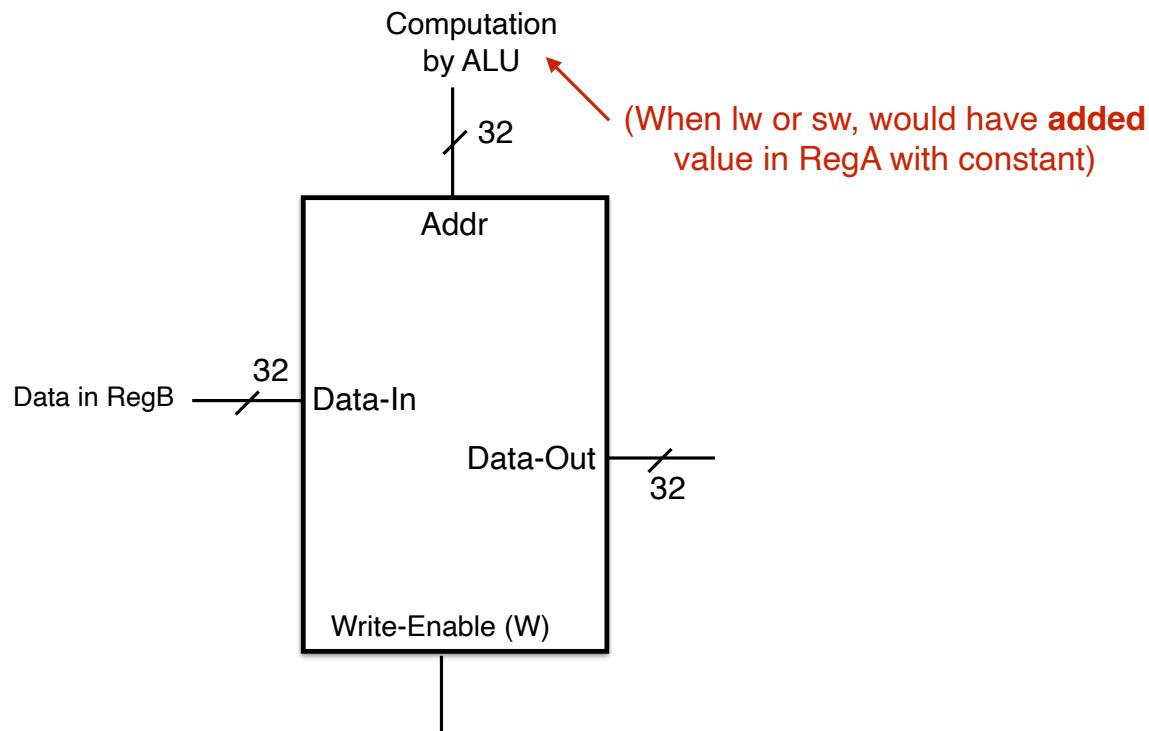
Something useful done here only if lw or sw

- If not lw or sw, memory result will be ignored



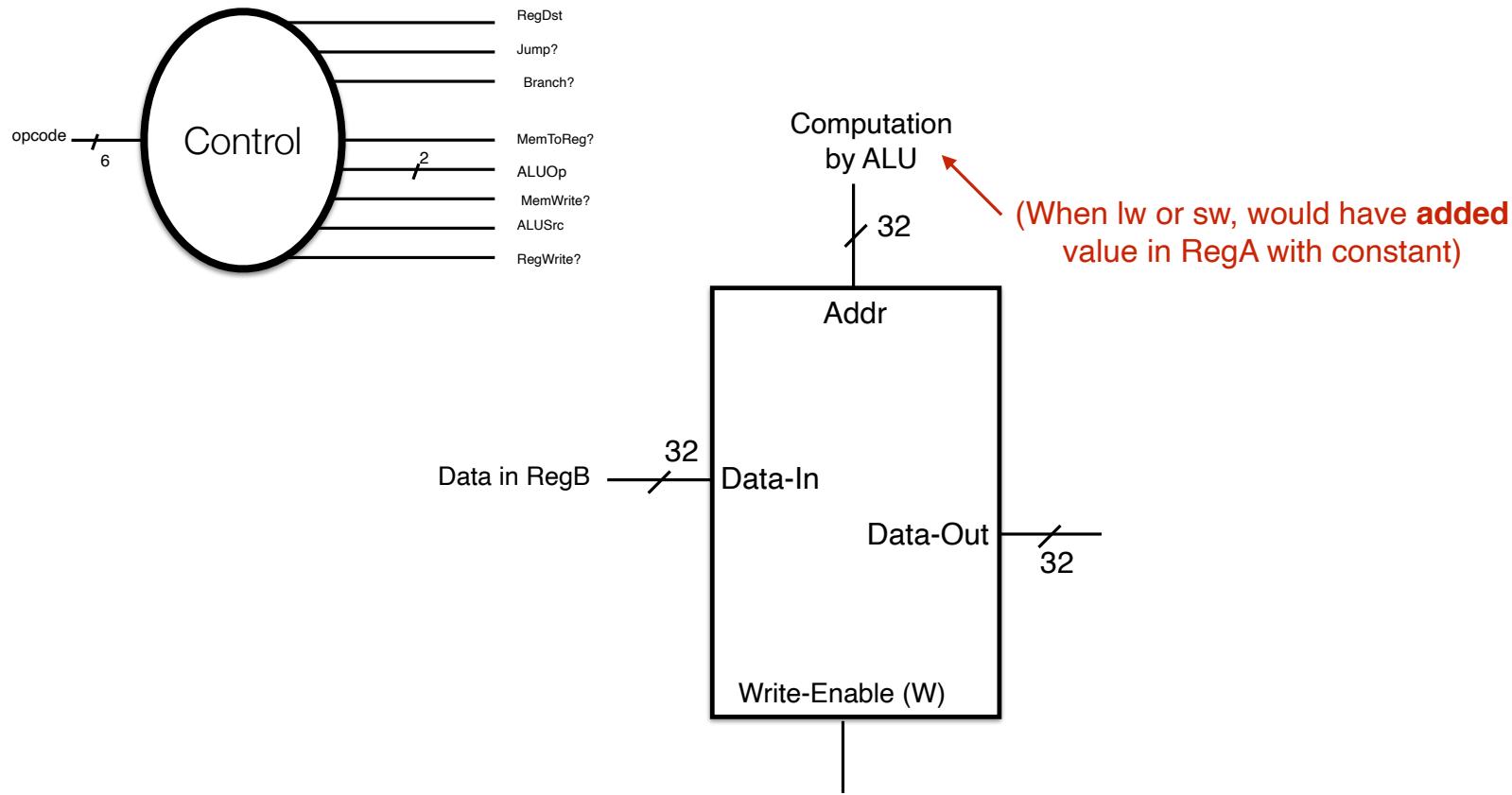
Something useful done here only if lw or sw

- If not lw or sw, memory result will be ignored



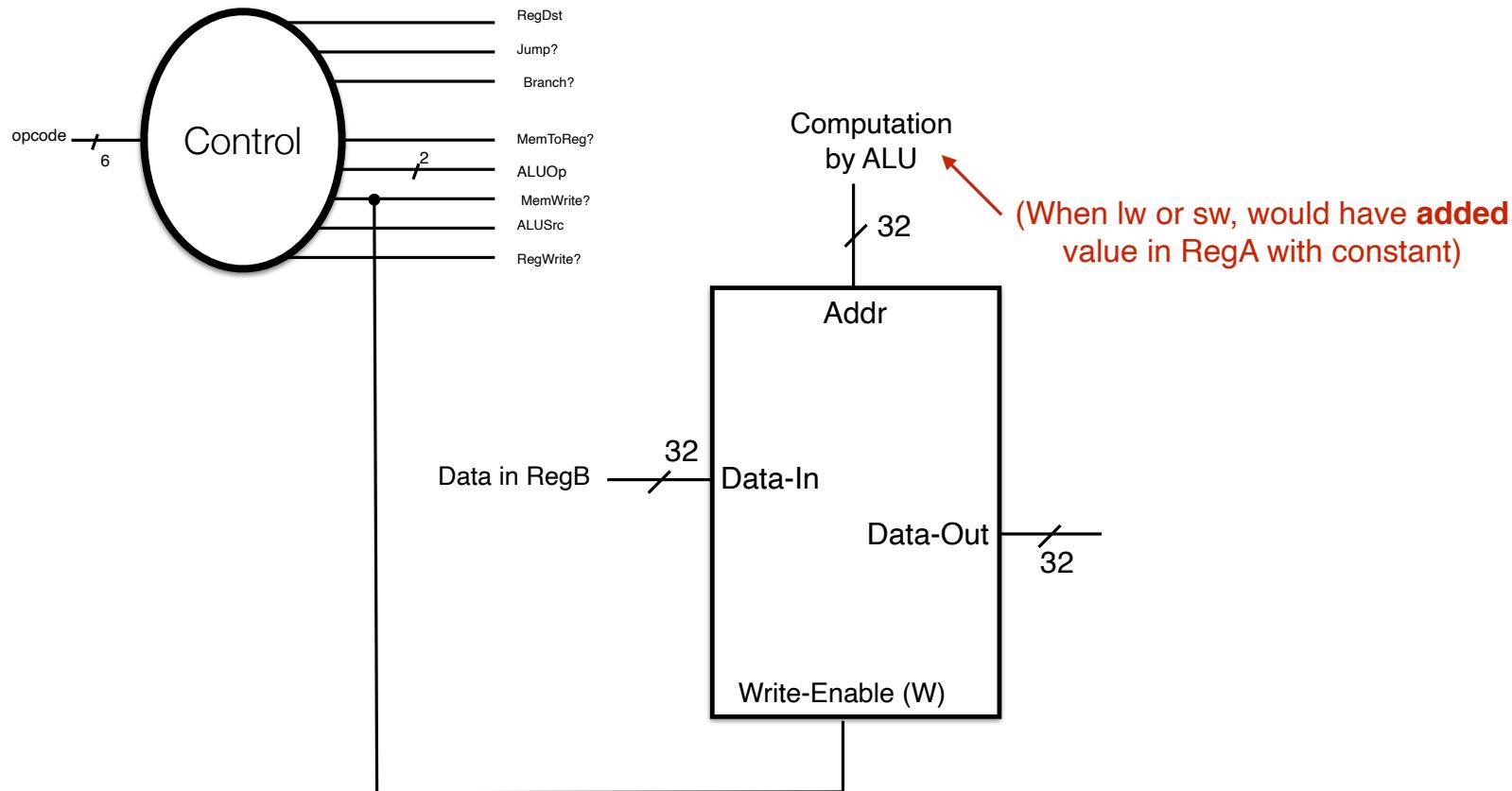
Something useful done here only if lw or sw

- If not lw or sw, memory result will be ignored



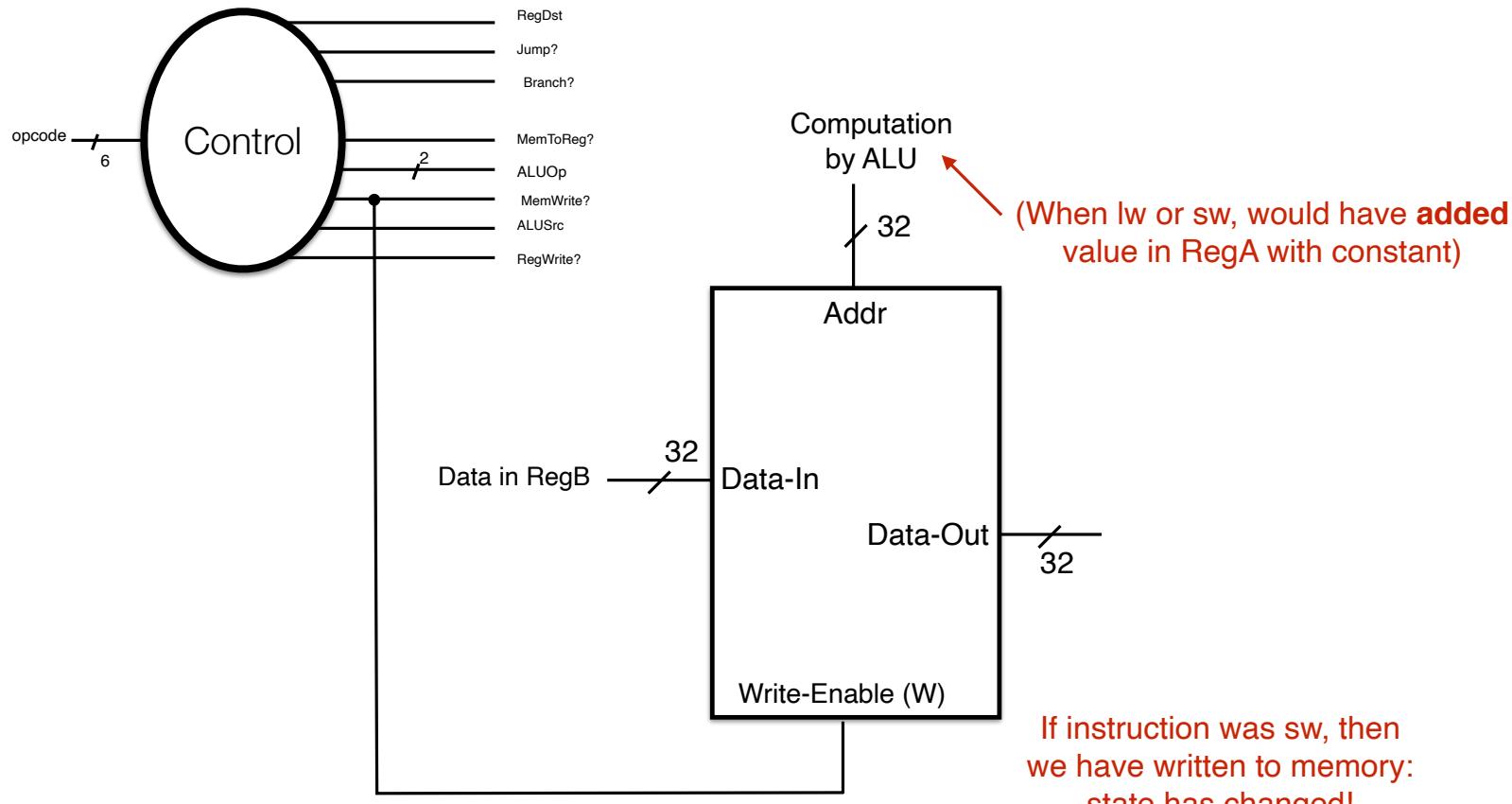
Something useful done here only if lw or sw

- If not lw or sw, memory result will be ignored



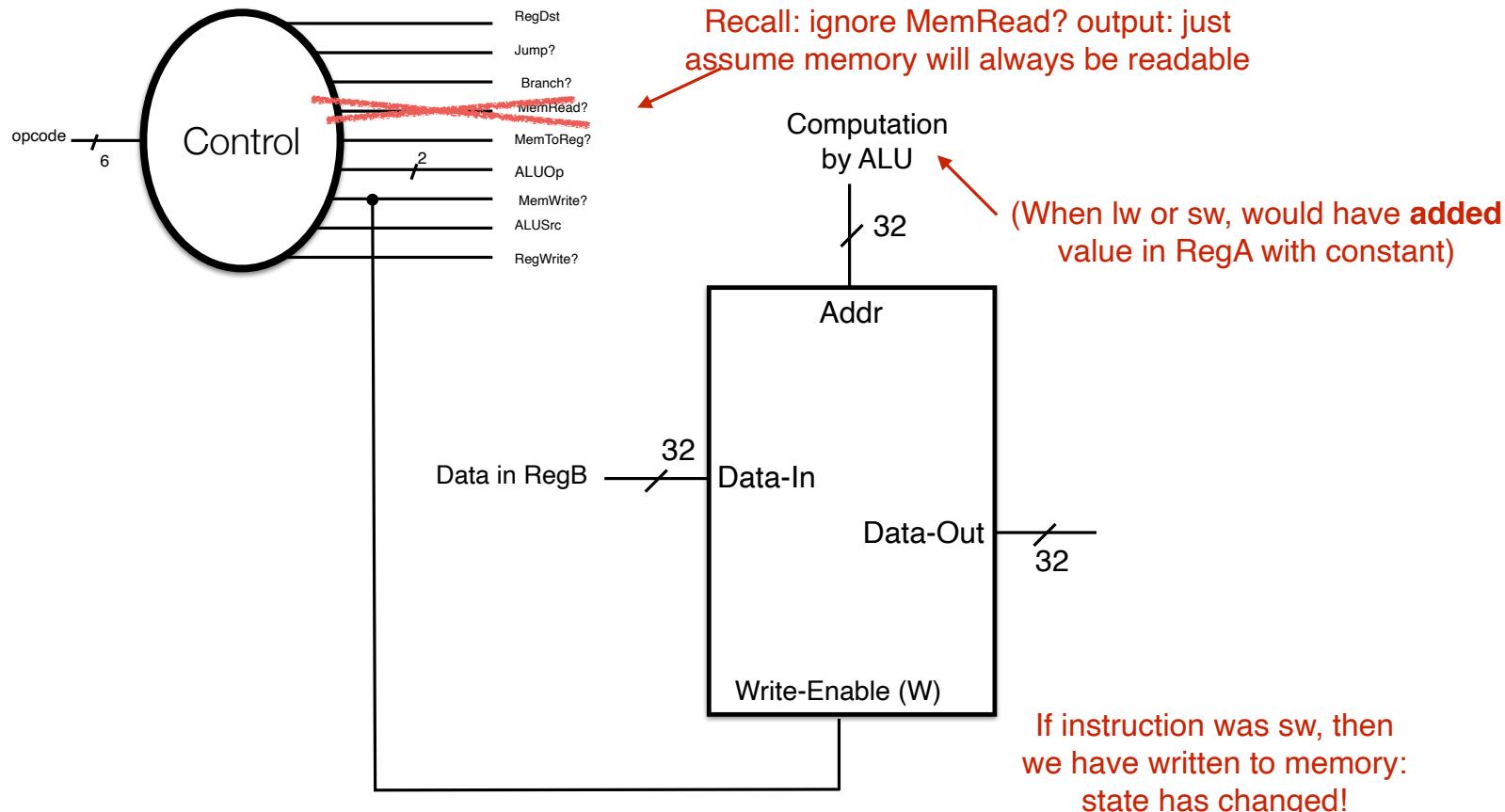
Something useful done here only if lw or sw

- If not lw or sw, memory result will be ignored



Something useful done here only if lw or sw

- If not lw or sw, memory result will be ignored



The final arch - components

- **Data Memory:** memory where program data (not instructions) are stored

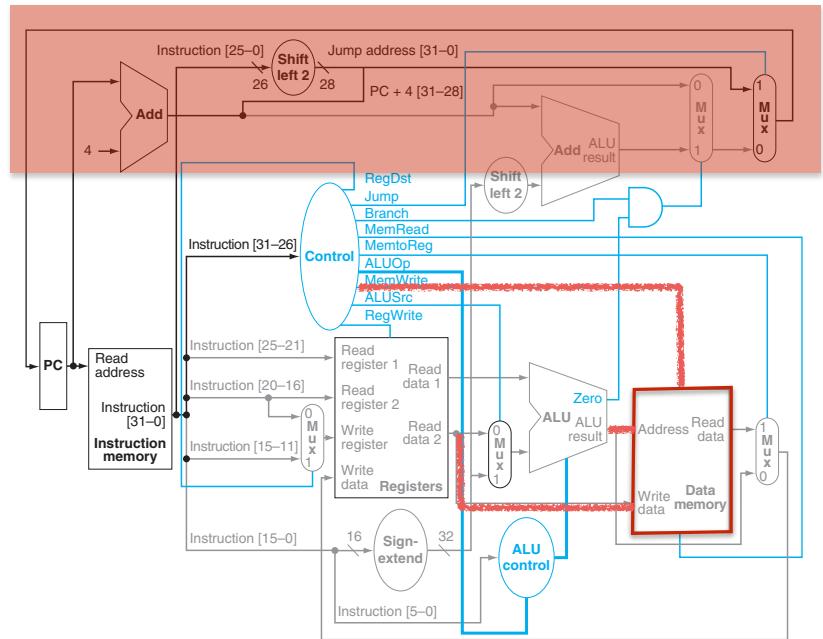


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target or either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

**Step 6:Writing back to
register file**

The final arch - components

- **Register File:** holds the registers that are read from and written to

- We looked earlier at the reading phase of the register file

- When the cycle ends, the register file might be written to

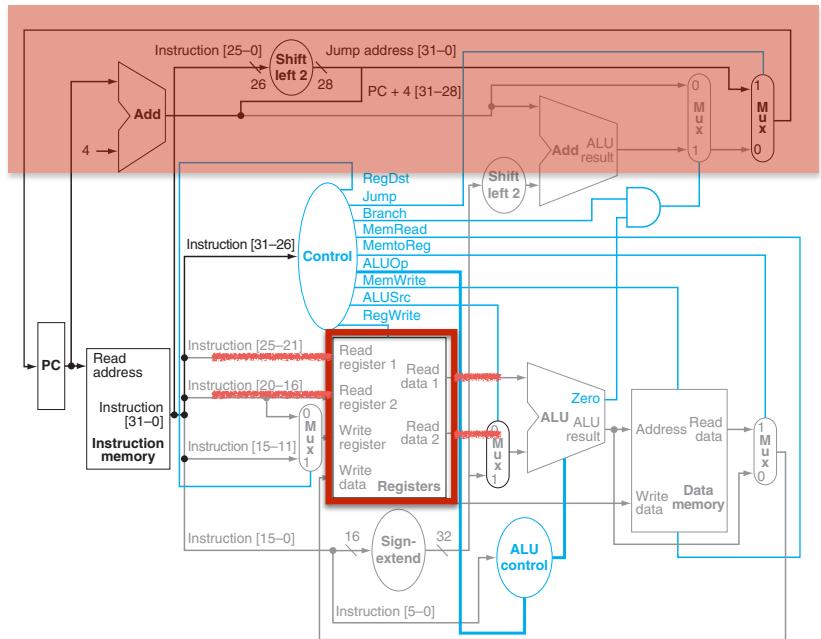
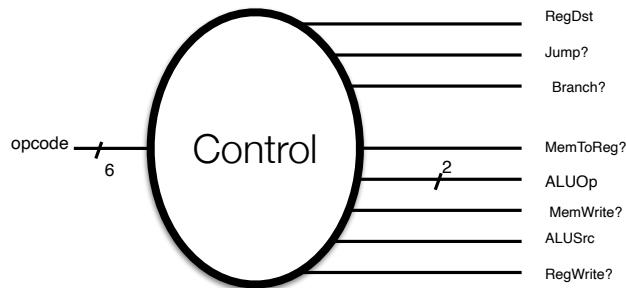
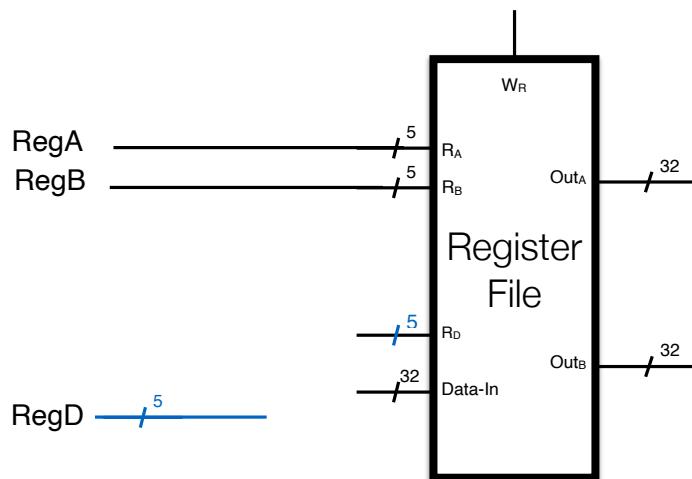


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexer (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexer is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

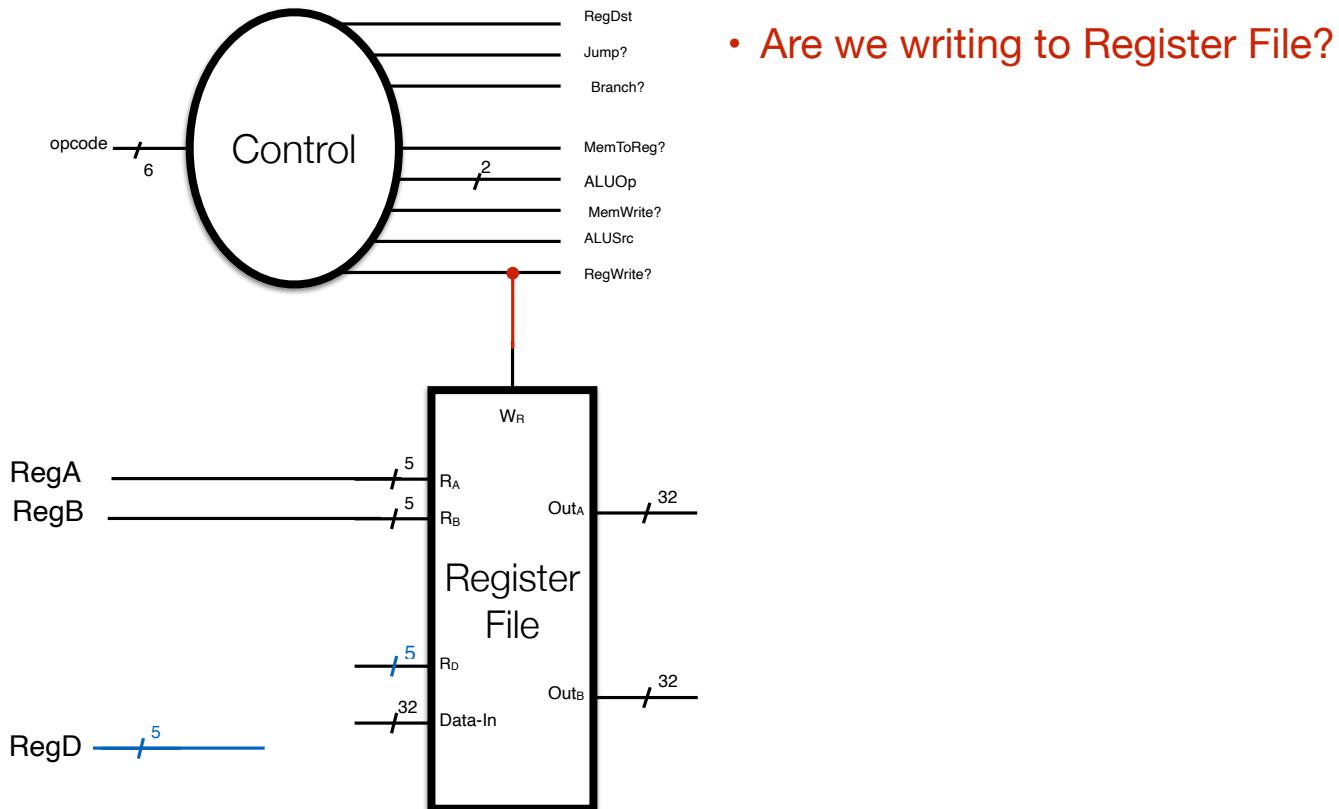
Writing to Register File Circuit



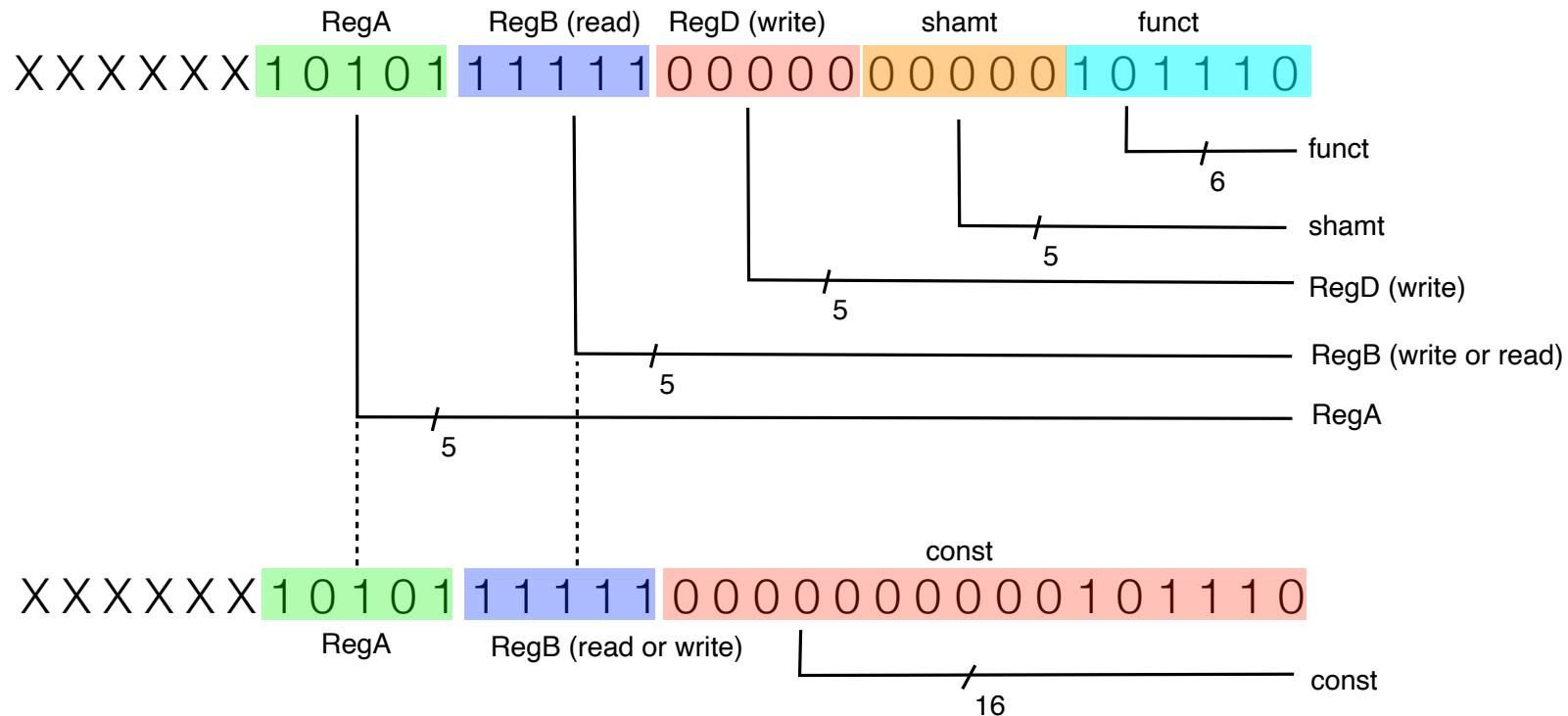
- Are we writing to Register File?



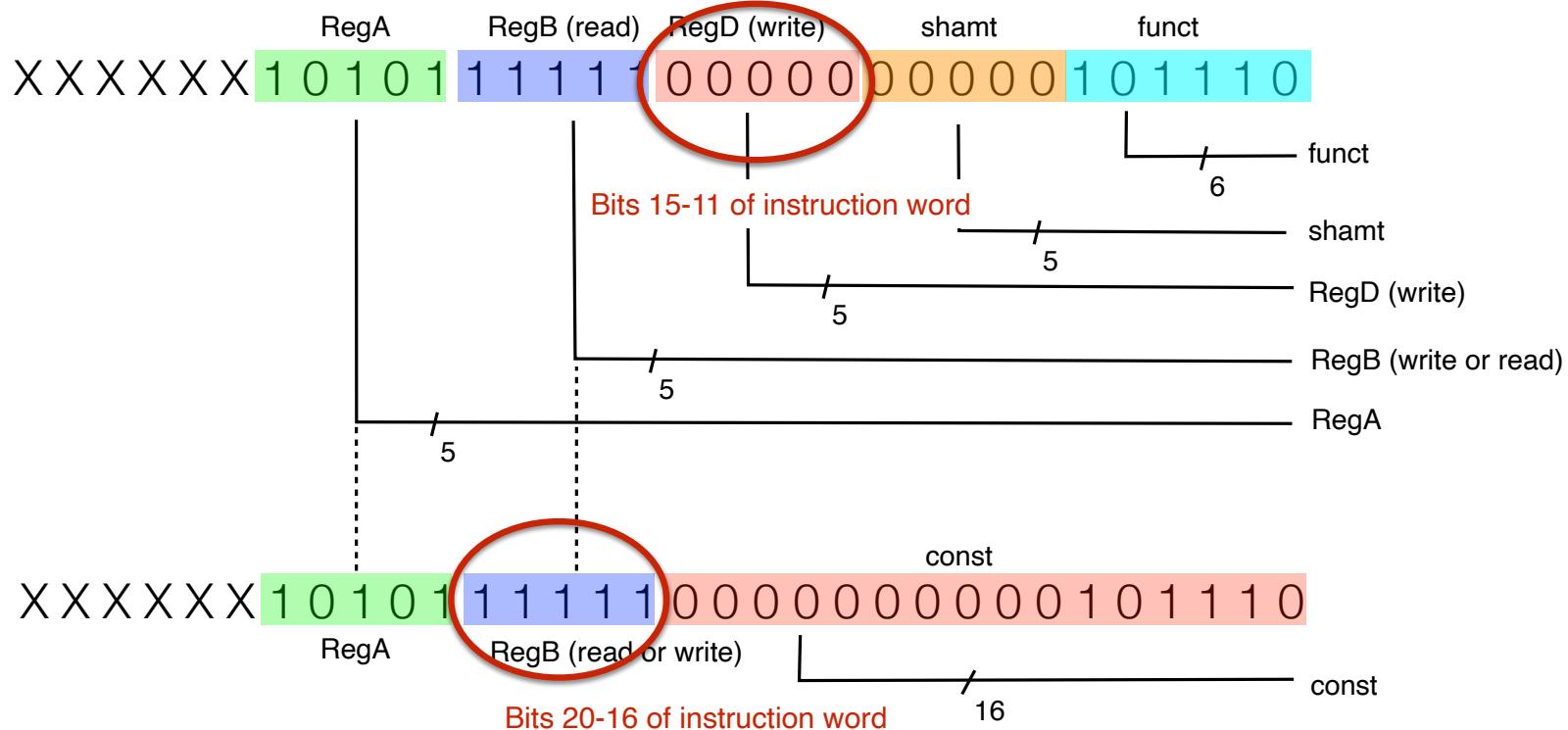
Writing to Register File Circuit



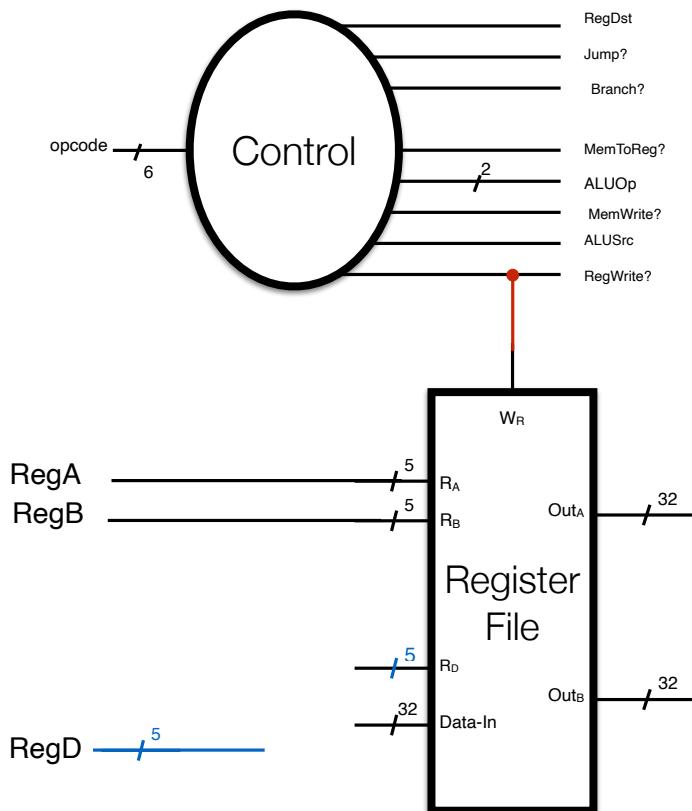
Where is description of register to write to?



Where is description of register to write to?

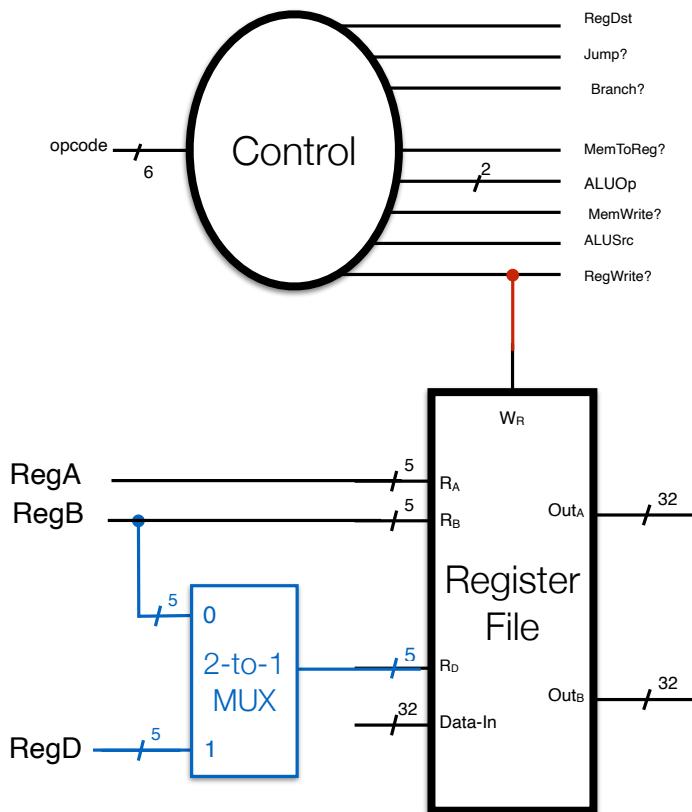


Writing to Register File Circuit



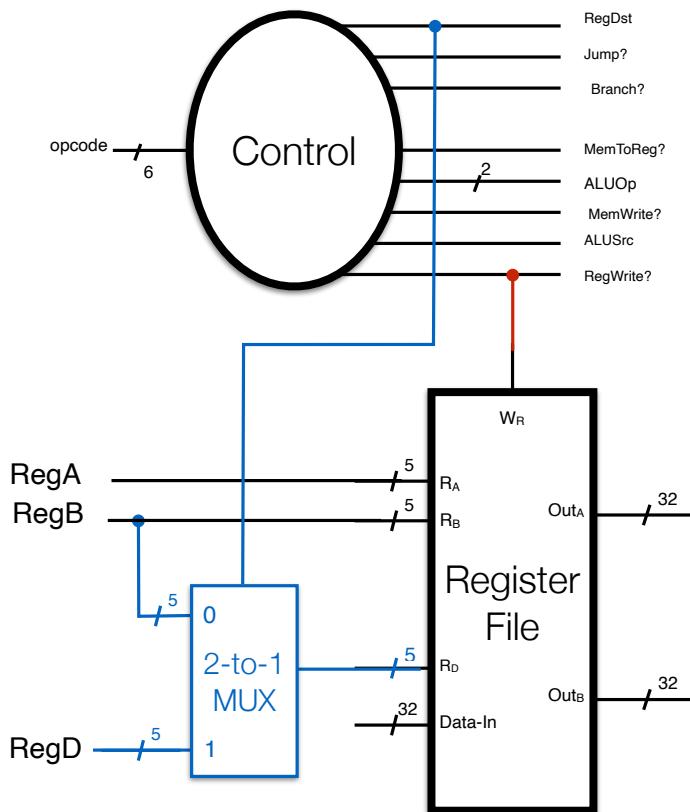
- Are we writing to Register File?
- Register to write to (if writing):
 - R-type: RegD
 - Otherwise: RegB

Writing to Register File Circuit



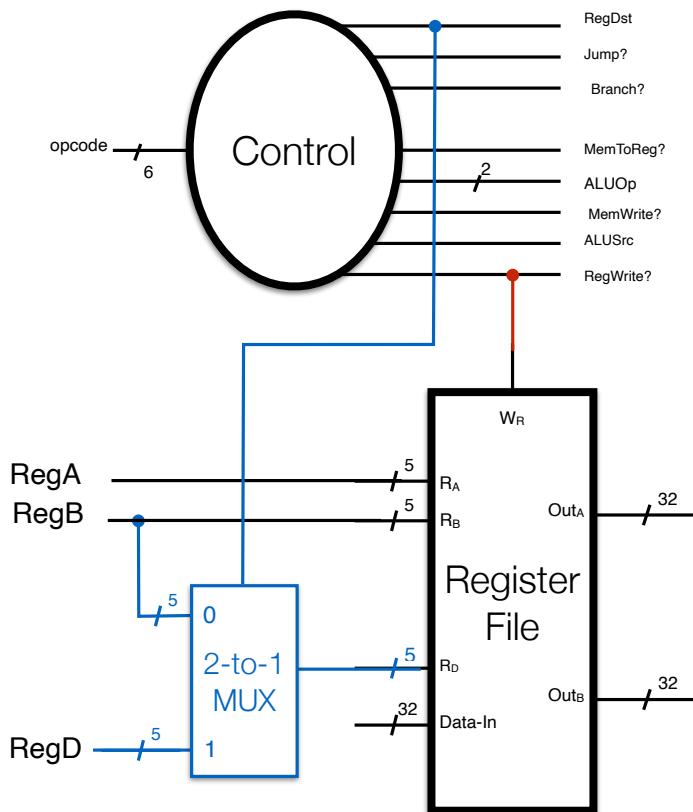
- Are we writing to Register File?
- Register to write to (if writing):
 - R-type: RegD
 - Otherwise: RegB

Writing to Register File Circuit



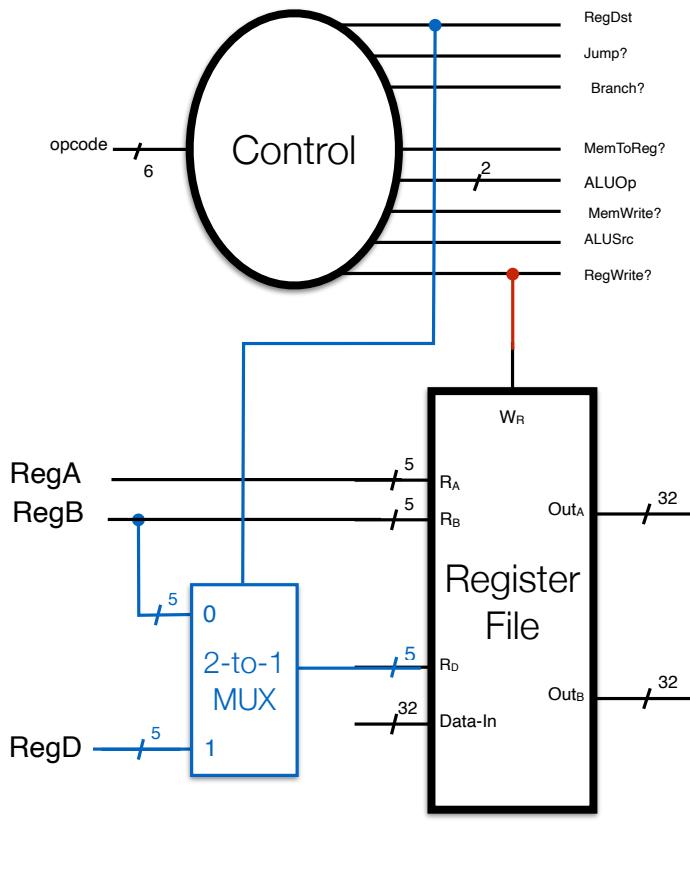
- Are we writing to Register File?
- Register to write to (if writing):
 - R-type: RegD
 - Otherwise: RegB

Writing to Register File Circuit

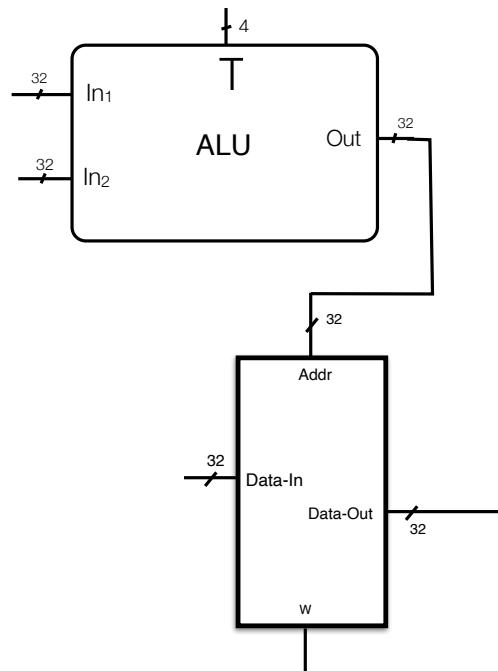


- Are we writing to Register File?
- Register to write to (if writing):
 - R-type: RegD
 - Otherwise: RegB
- Data to write from ALU (R-type, I-type) or memory (sw)?

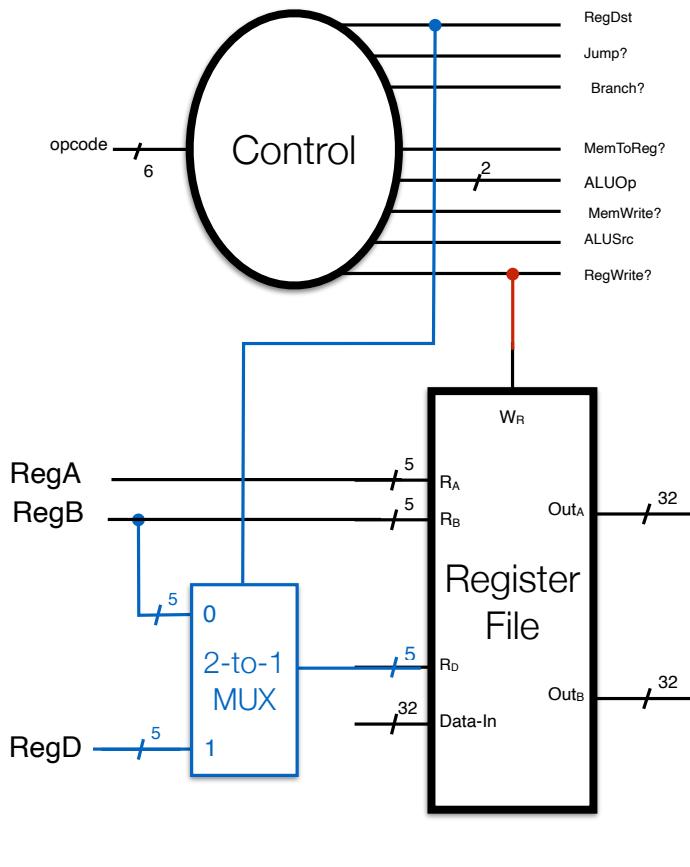
Writing to Register File Circuit



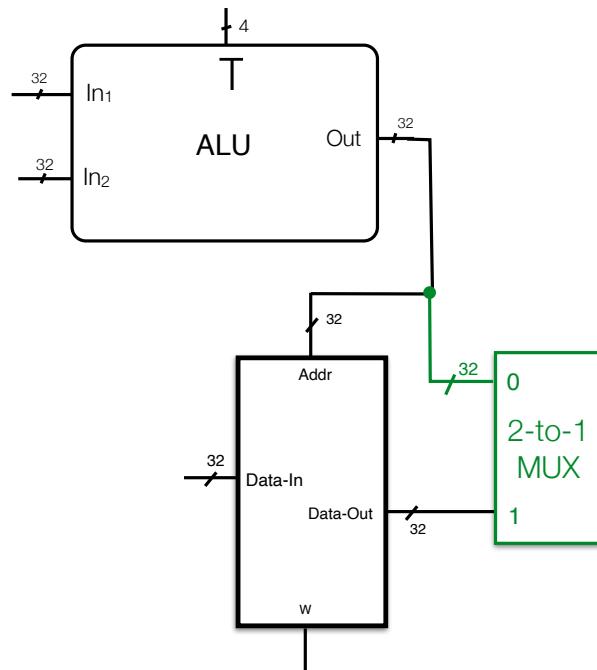
- Data to write from ALU (R-type, I-type) or memory (sw)?



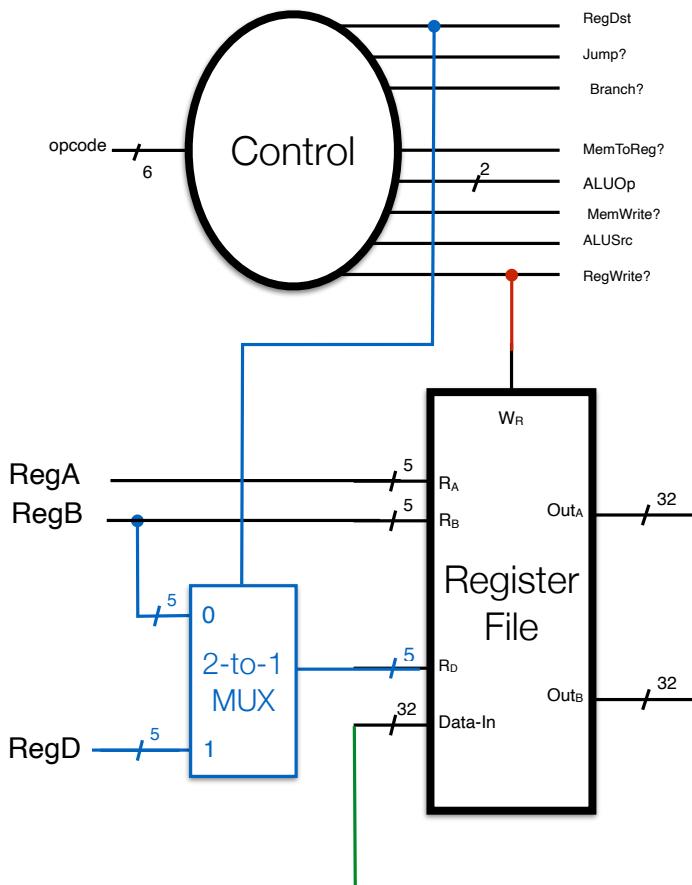
Writing to Register File Circuit



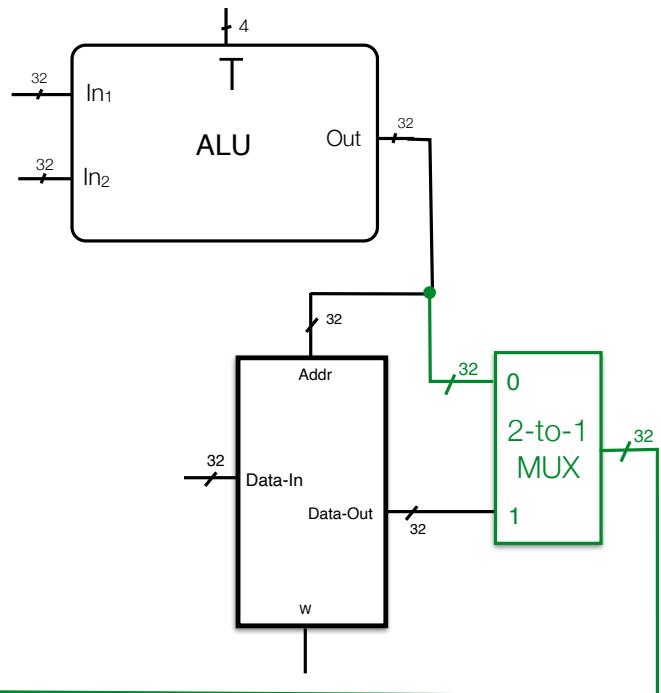
- Data to write from ALU (R-type, I-type) or memory (sw)?



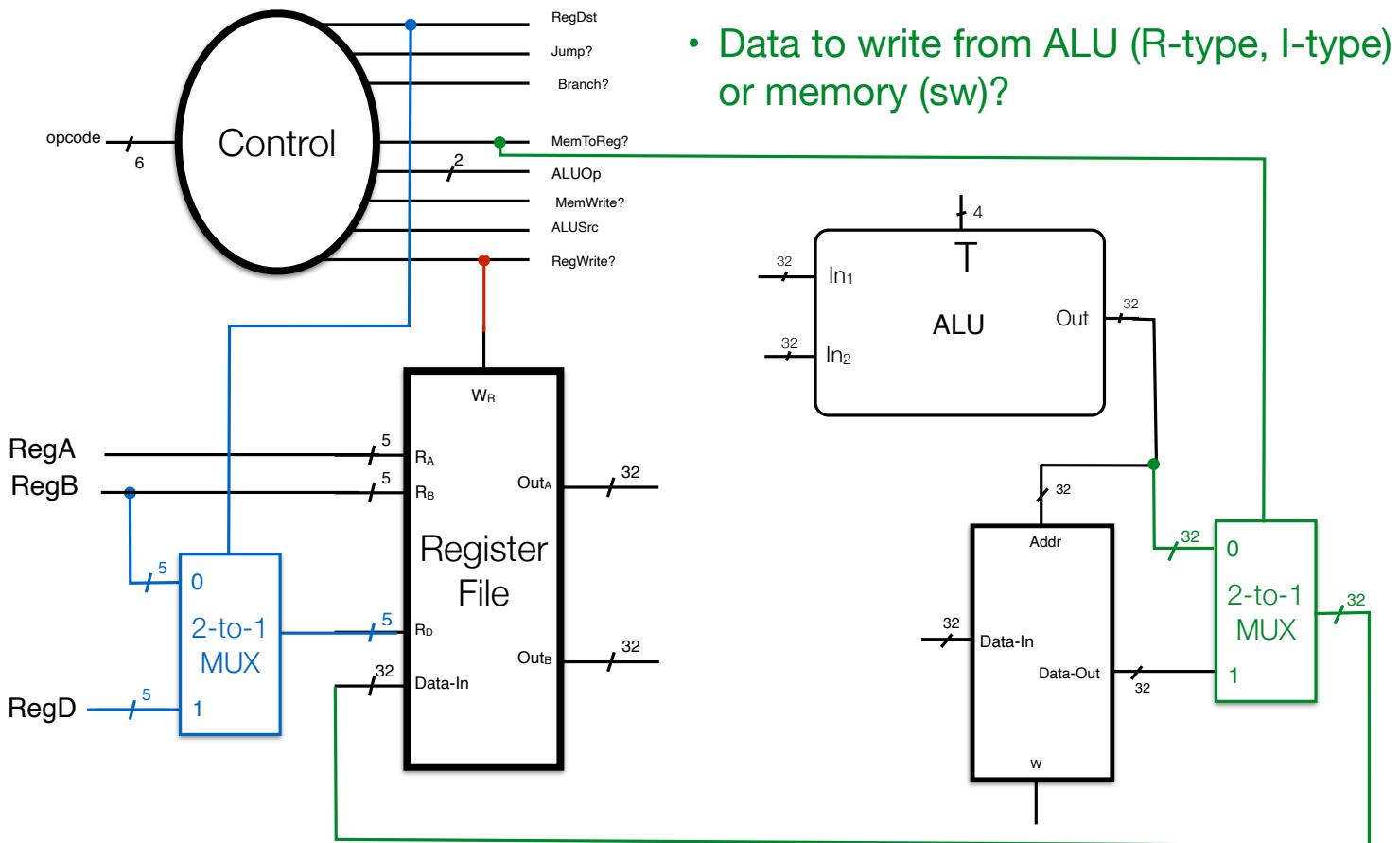
Writing to Register File Circuit



- Data to write from ALU (R-type, I-type) or memory (sw)?



Writing to Register File Circuit



The path of data

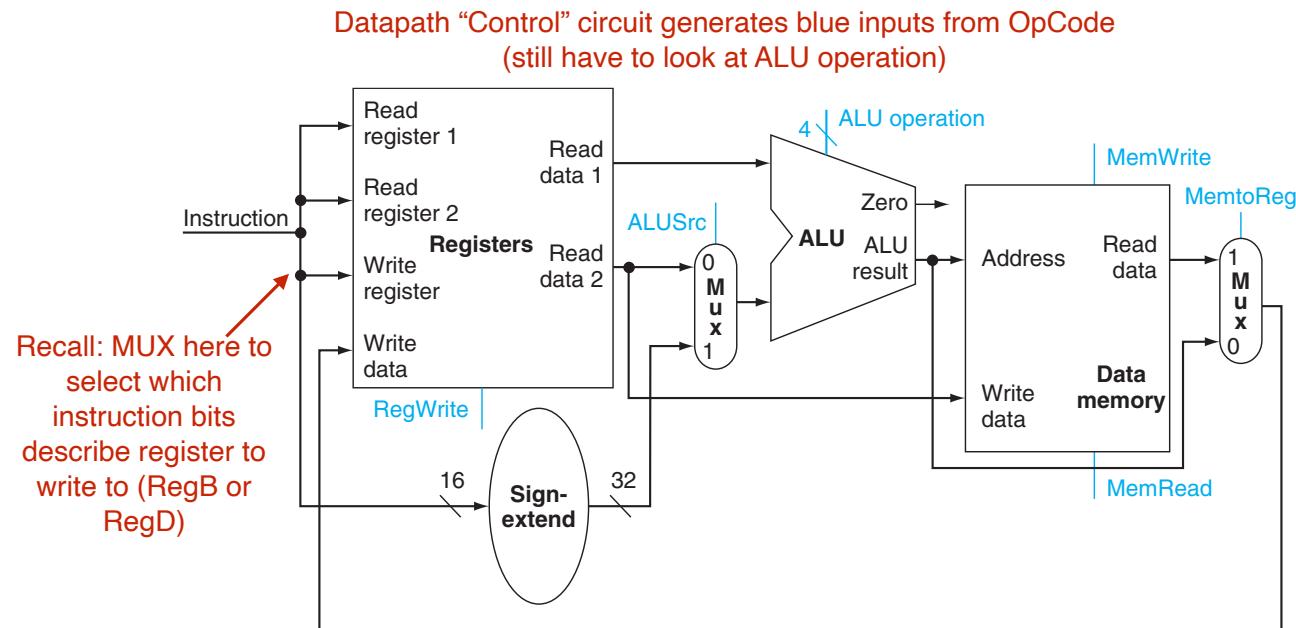


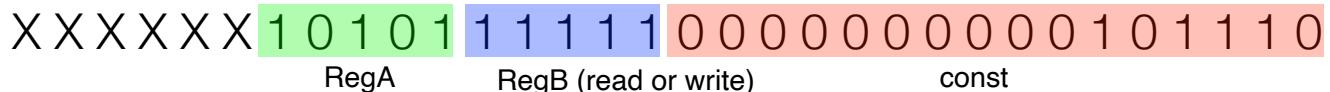
FIGURE 4.10 The datapath for the memory instructions and the R-type instructions. This example shows how a single datapath can be assembled from the pieces in Figures 4.7 and 4.8 by adding multiplexors. Two multiplexors are needed, as described in the example. Copyright © 2009 Elsevier, Inc. All rights reserved.



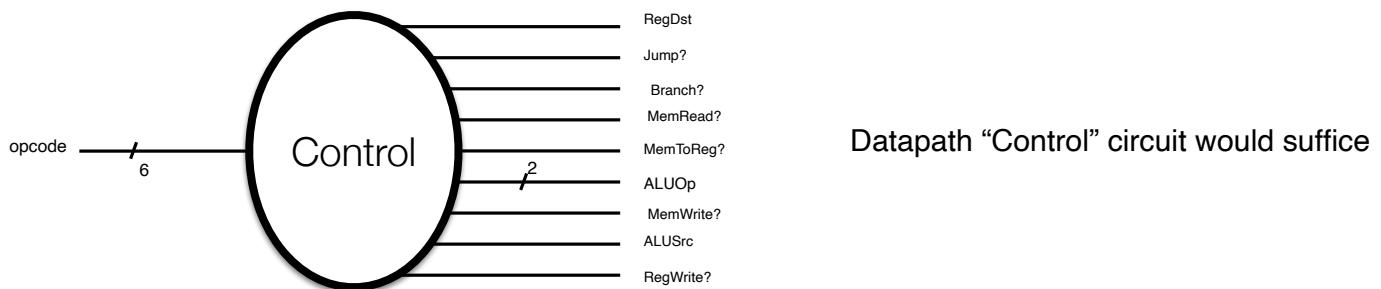
Determining ALU function to perform

Often in the OpCode

- Assume is I-type, branch, or memory access



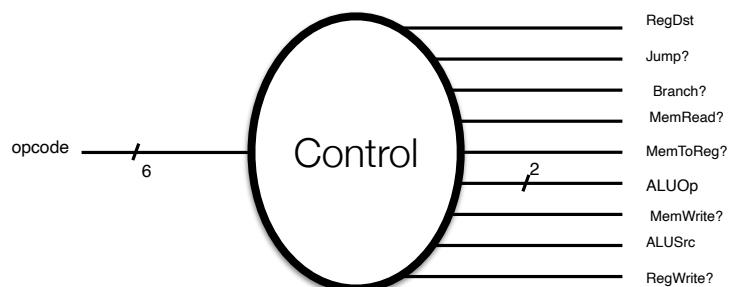
- If is **Branch** (e.g., `beq`, `bne`): will use ALU to compare equality (**subtract**)
- If is **I-type**: Opcode specifies the function (add, subtract, OR, AND, etc.)
- If is **Mem access**: ALU used to **add** base address (in register) to constant



R-type: not in the op-code



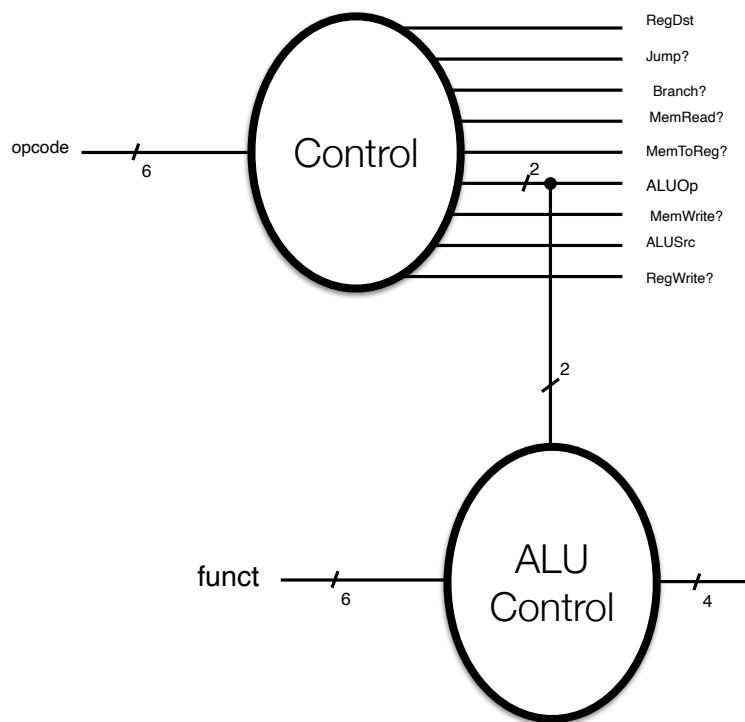
- For R-type: OpCode is 000000, need to also look at funct code



R-type: not in the op-code



- For R-type: OpCode is 000000, need to also look at funct code
- Designed as follows:
 - “Control” parses Op-code and passes 2-bit signal to an “ALU control” circuit that will set the ALU
 - ALU control reads in both ALUOp and funct. These two inputs determine how to set ALU



Datapath Control Scheme

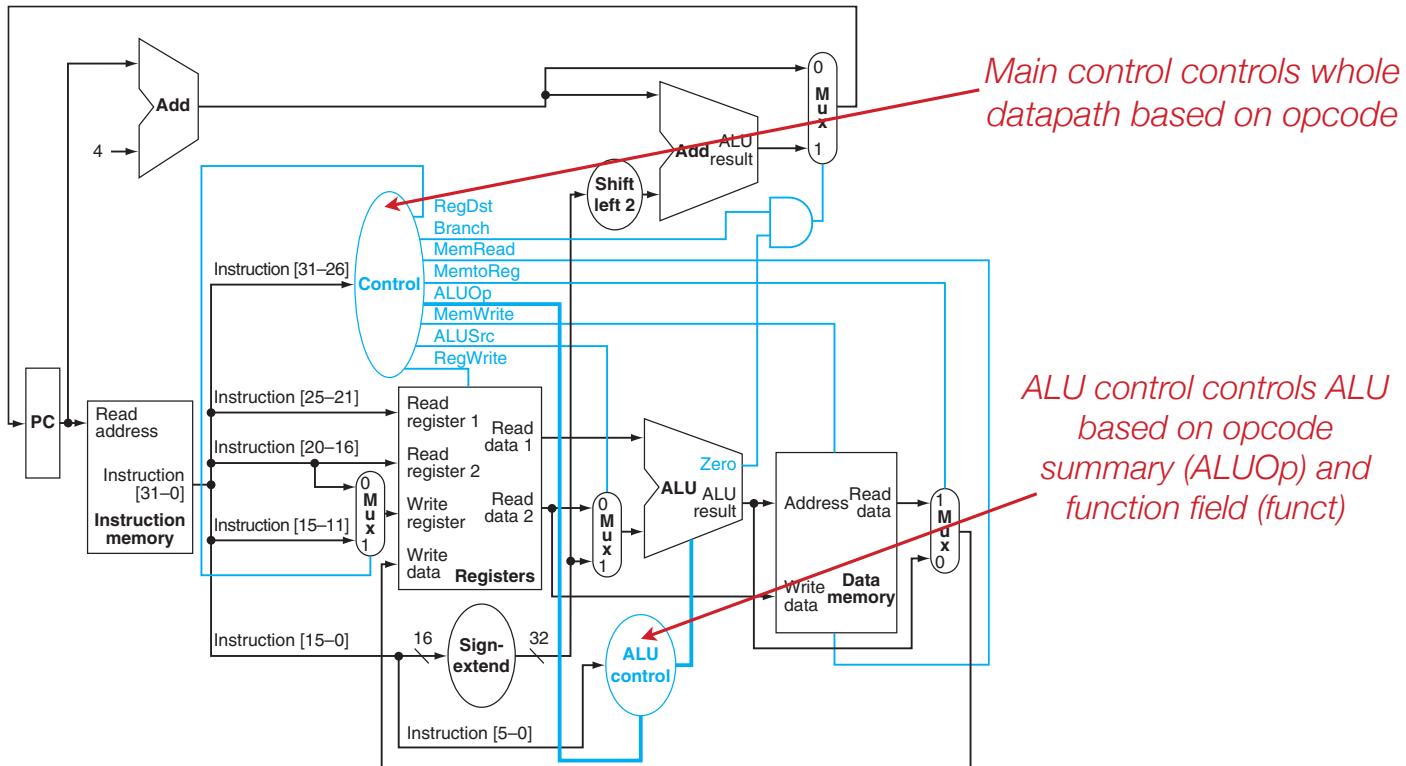
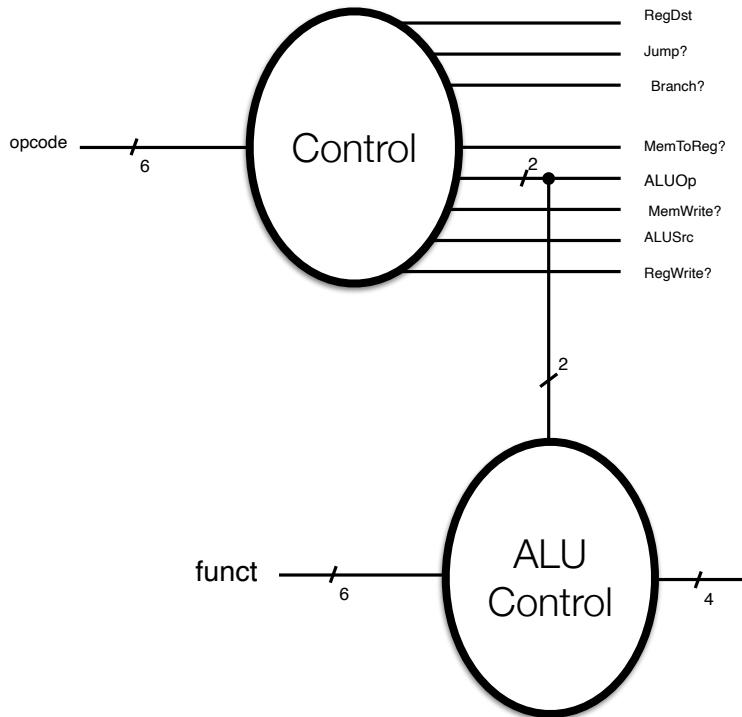


FIGURE 4.17 The simple datapath with the control unit. The input to the control unit is the 6-bit opcode field from the instruction. The outputs of the control unit consist of three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Thus, we drop the signal name in subsequent figures. Copyright © 2009 Elsevier, Inc. All rights reserved.



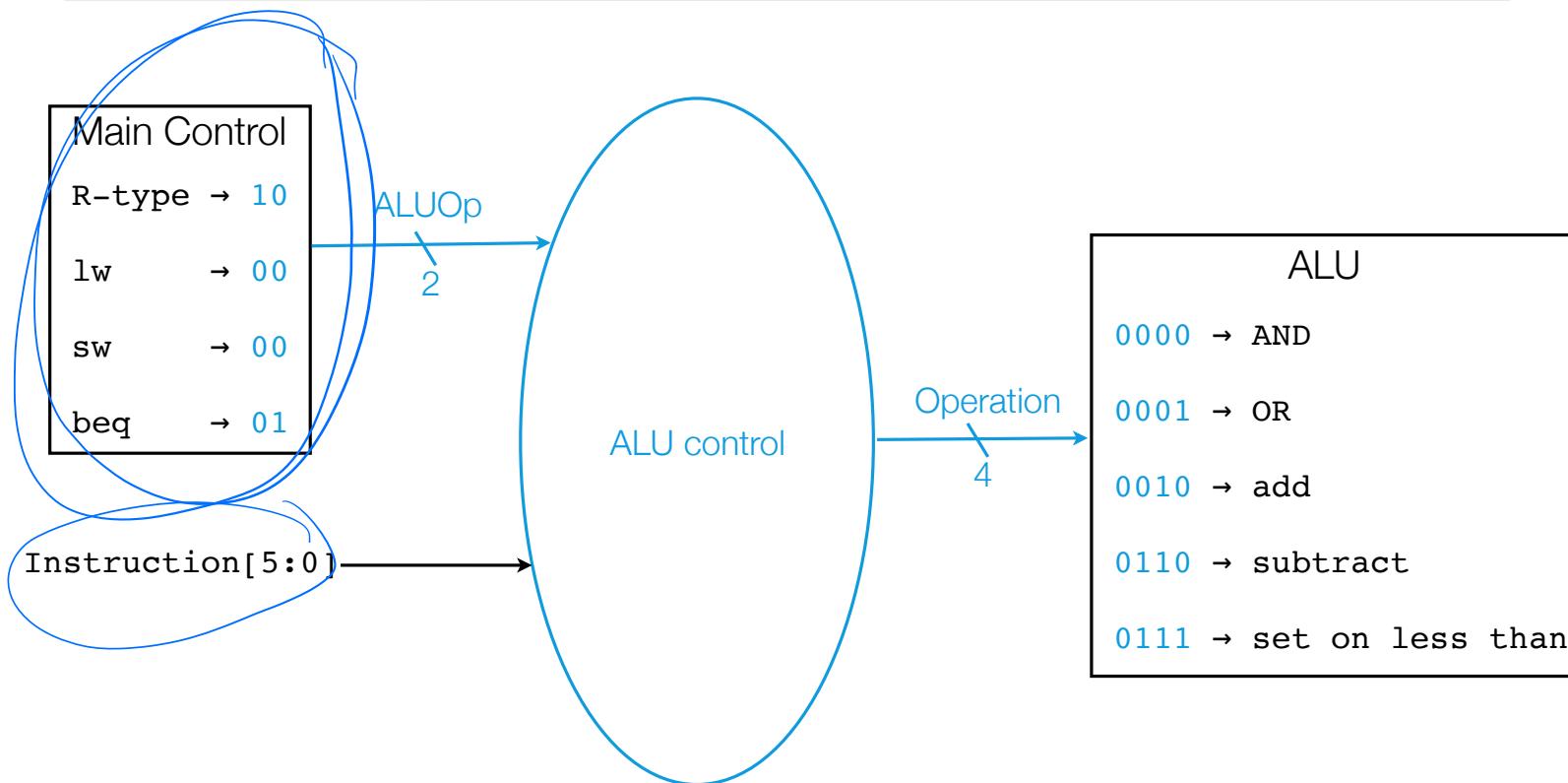
Mapping opcode & funct to 4-bit T



ALUOp	Function & using	T
00	add (lw or sw)	0010
01	subtract (beq, bne)	0110
10	R-type	depends on funct
11		UNUSED

- ALUOp can
 - indicate (to ALU control) the specific operation to perform (e.g., ALUOp = 00 or 01)
 - indicate (to ALU control) to look at funct code (ALUOp = 10)

ALU Control Inputs/Outputs



Note: I-type logic (e.g., ORI, ANDI) instructions not included in what is shown here

ALU Control Implementation

opcode	Instruction[5:0]	ALUOp from main control	Operation
lw	xxxxxx	→ load word	→ add → 0010
sw	xxxxxx	→ store word	→ add → 0010
beq	xxxxxx	→ branch equal	→ subtract → 0110
R-type	100000	→ add	→ add → 0010
R-type	100010	→ subtract	→ subtract → 0110
R-type	100100	→ AND	→ AND → 0000
R-type	100101	→ OR	→ OR → 0001
R-type	101010	→ set on less than	→ set on less than → 0111

ALU Control Truth Table

ALUOp from main control	Instruction[5:0]	Operation
00	xxxxxx	0010
00	xxxxxx	0010
01	xxxxxx	0110
10	100000	0010
10	100010	0110
10	100100	0000
10	100101	0001
10	101010	0111

The final arch - components

- Function Unit (ALU): performs the computation (+,-,&,|,shift, etc.)

- Red lines: Data flow
- Green lines: setting ALU function

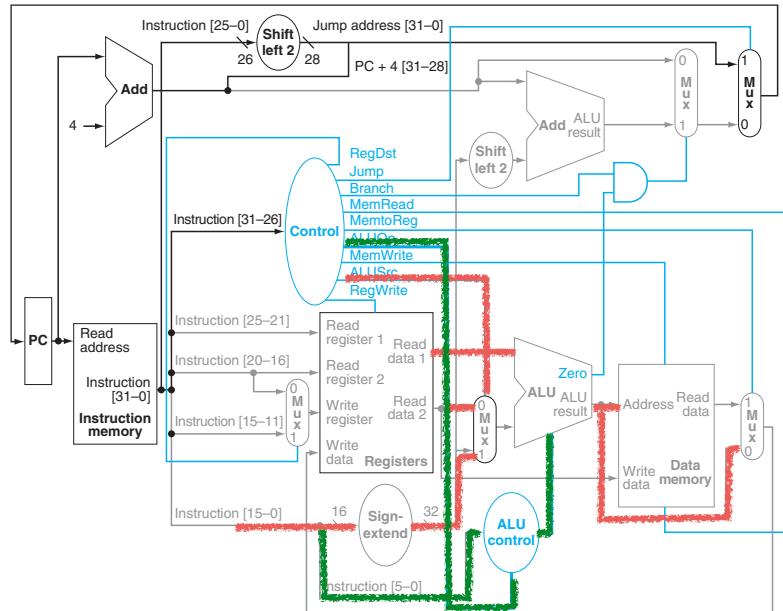
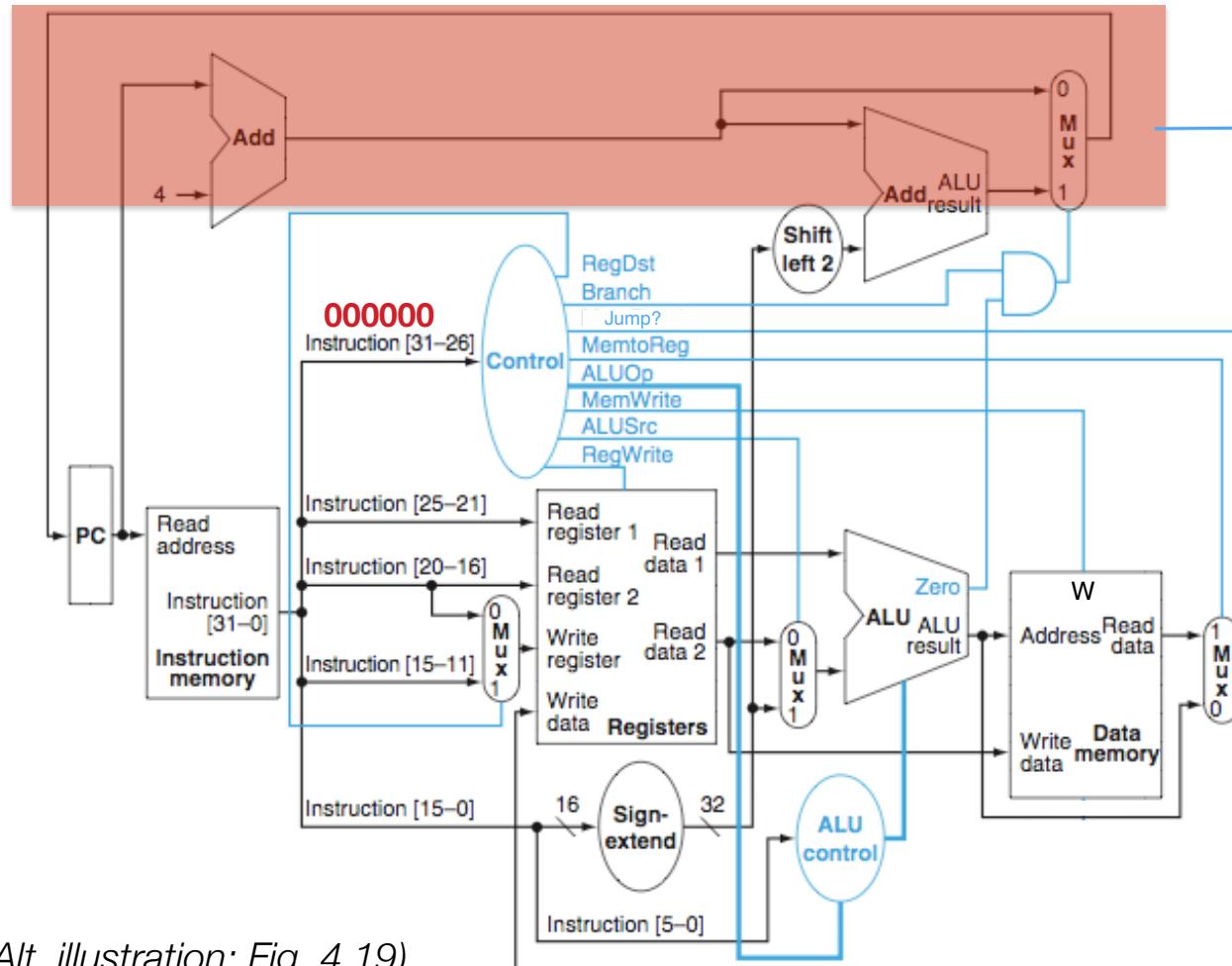


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexer (at the upper right) is used to choose between the jump target or either the branch target or the sequential instruction following this one. This multiplexer is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

How various
instructions' OpCodes
control data flow

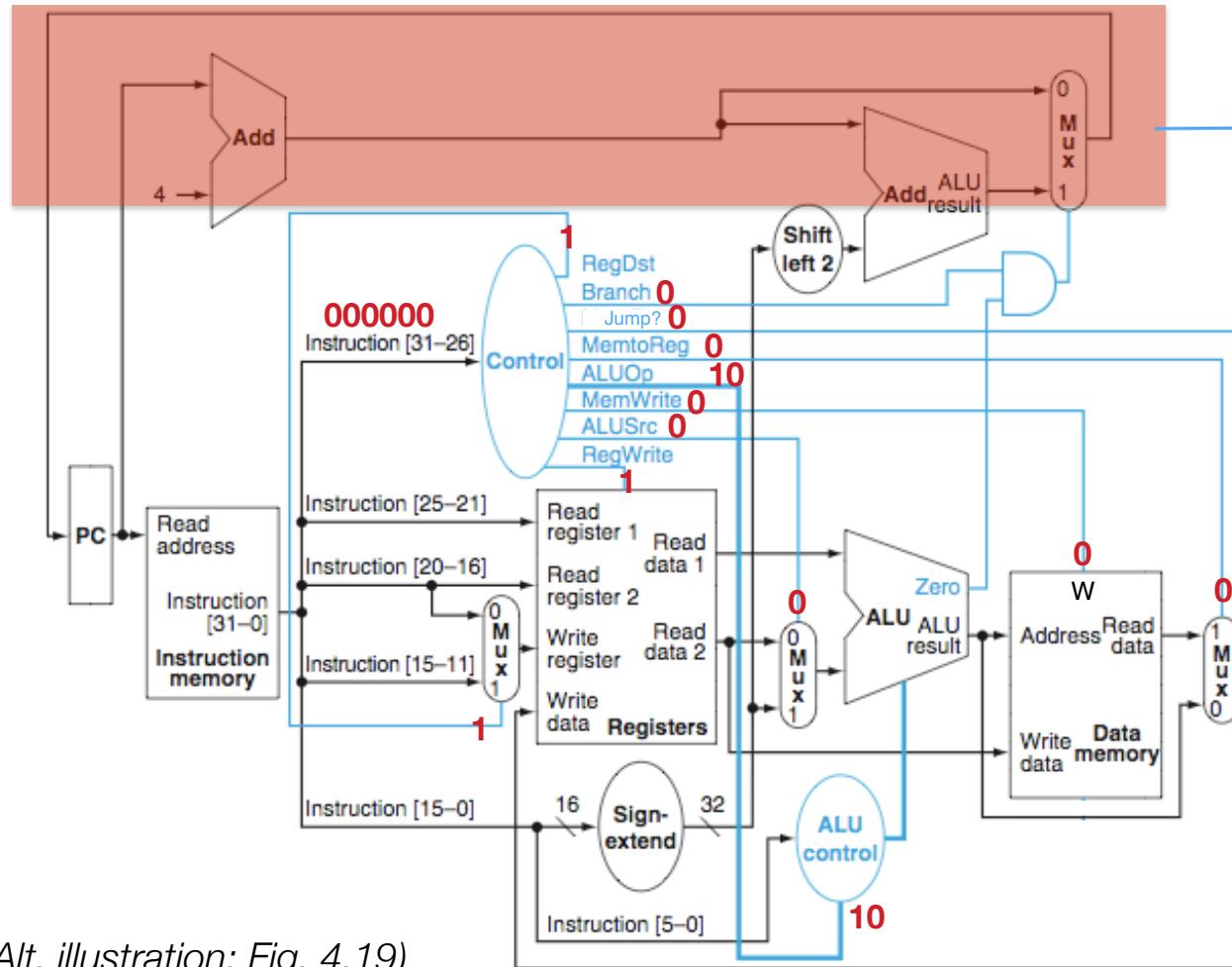
R-Type Instruction (generic)



(Alt. illustration: Fig. 4.19)



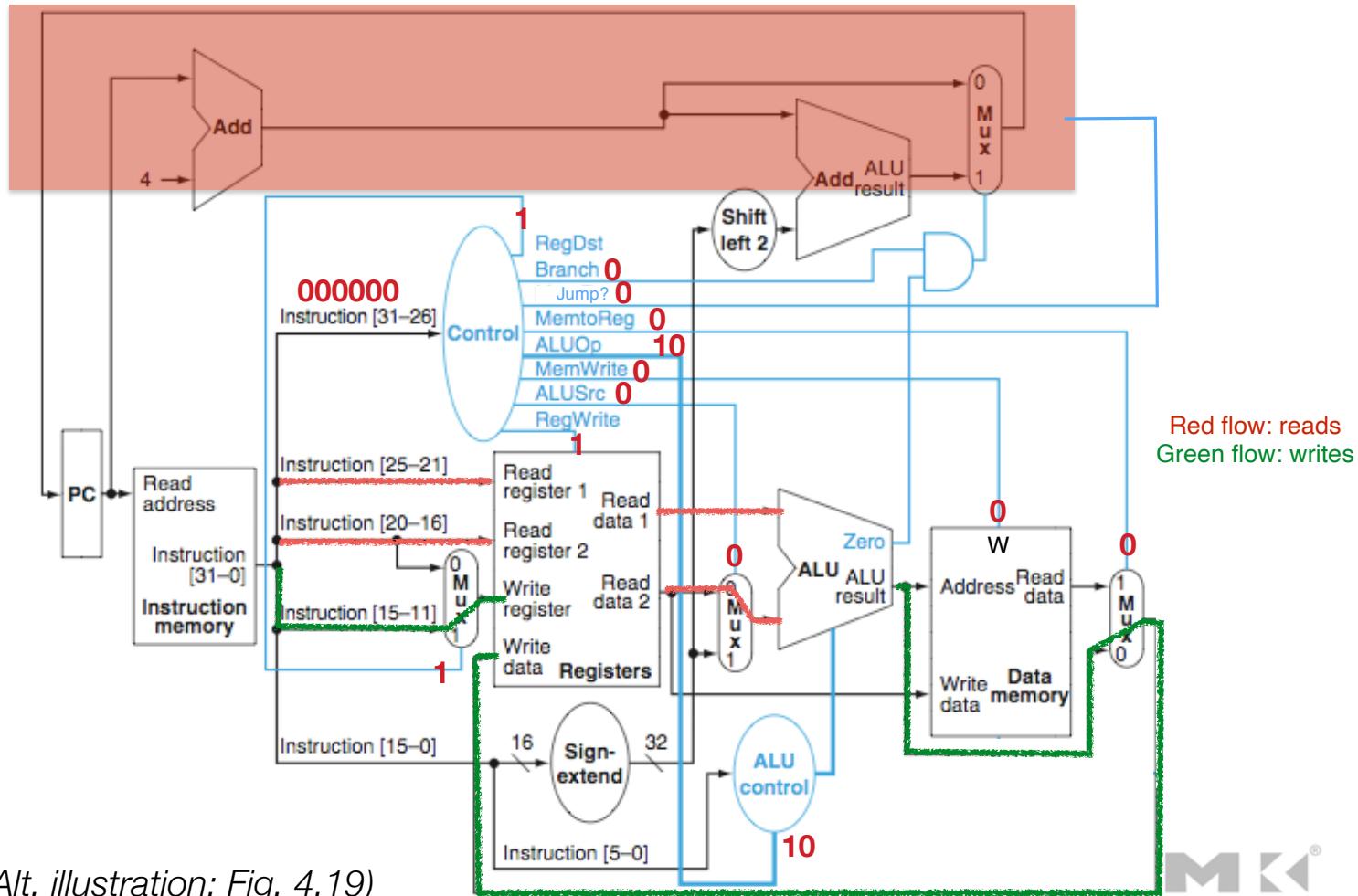
R-Type Instruction (generic)



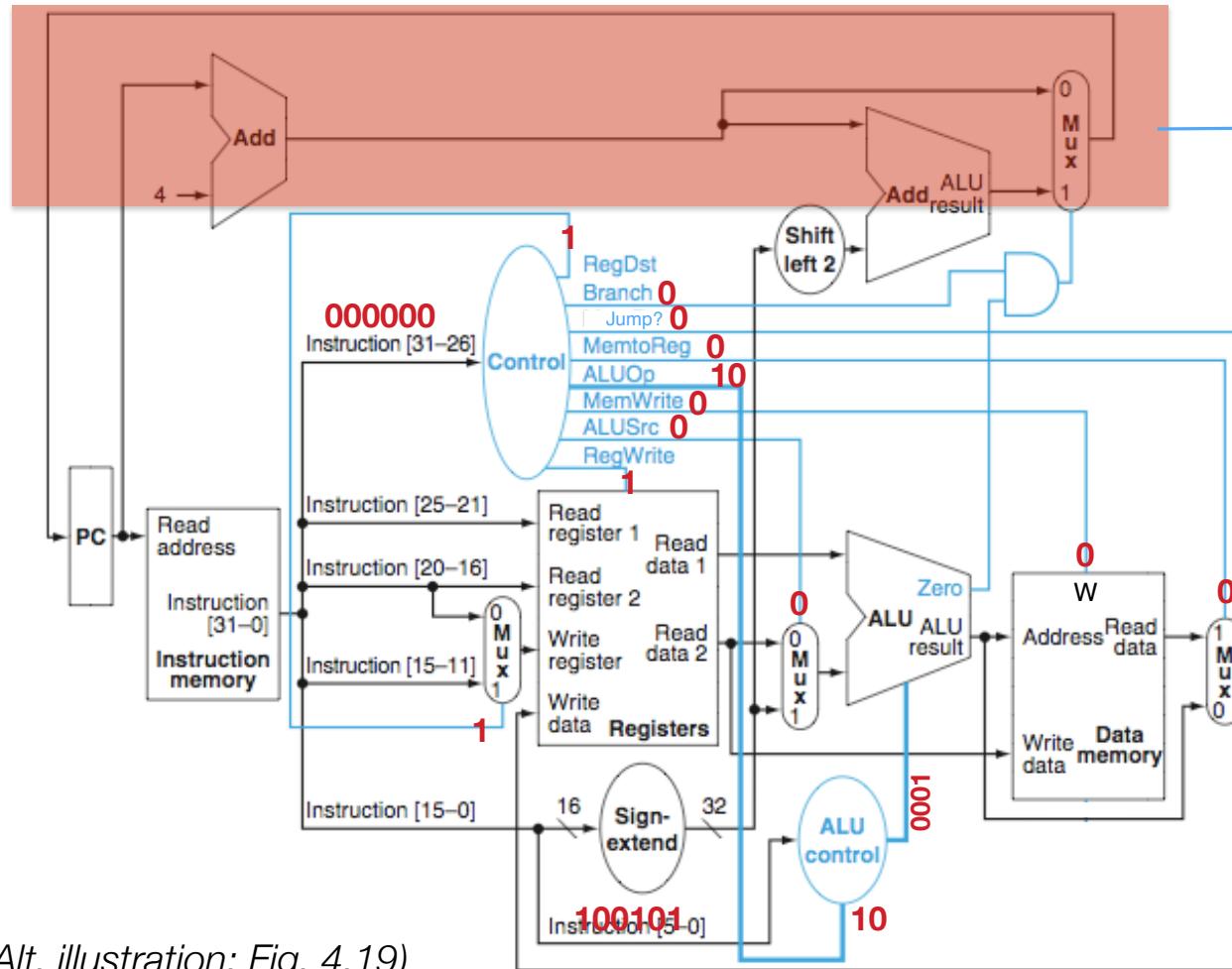
(Alt. illustration: Fig. 4.19)



R-Type Instruction (generic): Used Data Flow



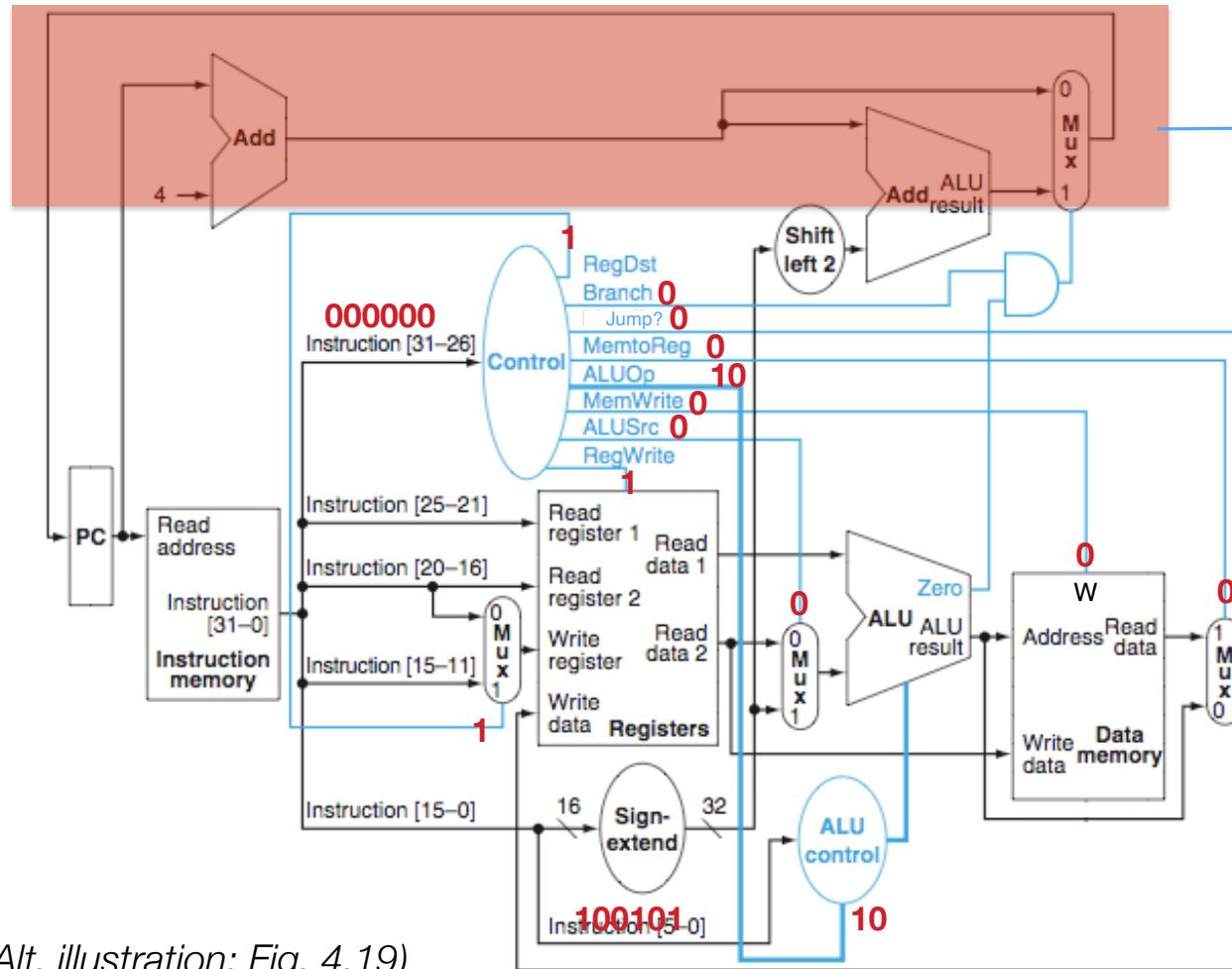
R-Type Instruction (OR)



(Alt. illustration: Fig. 4.19)



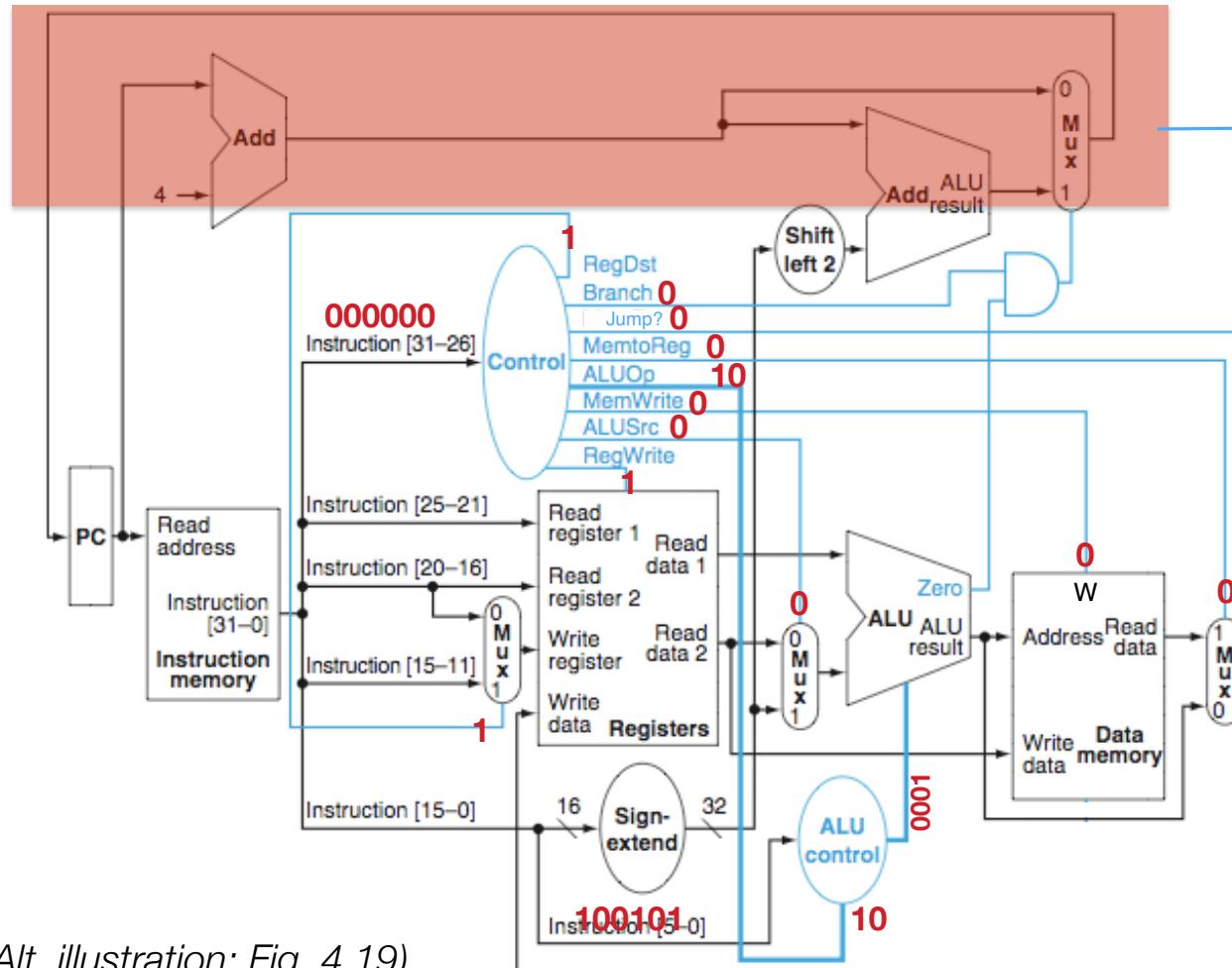
R-Type Instruction (OR)



(Alt. illustration: Fig. 4.19)



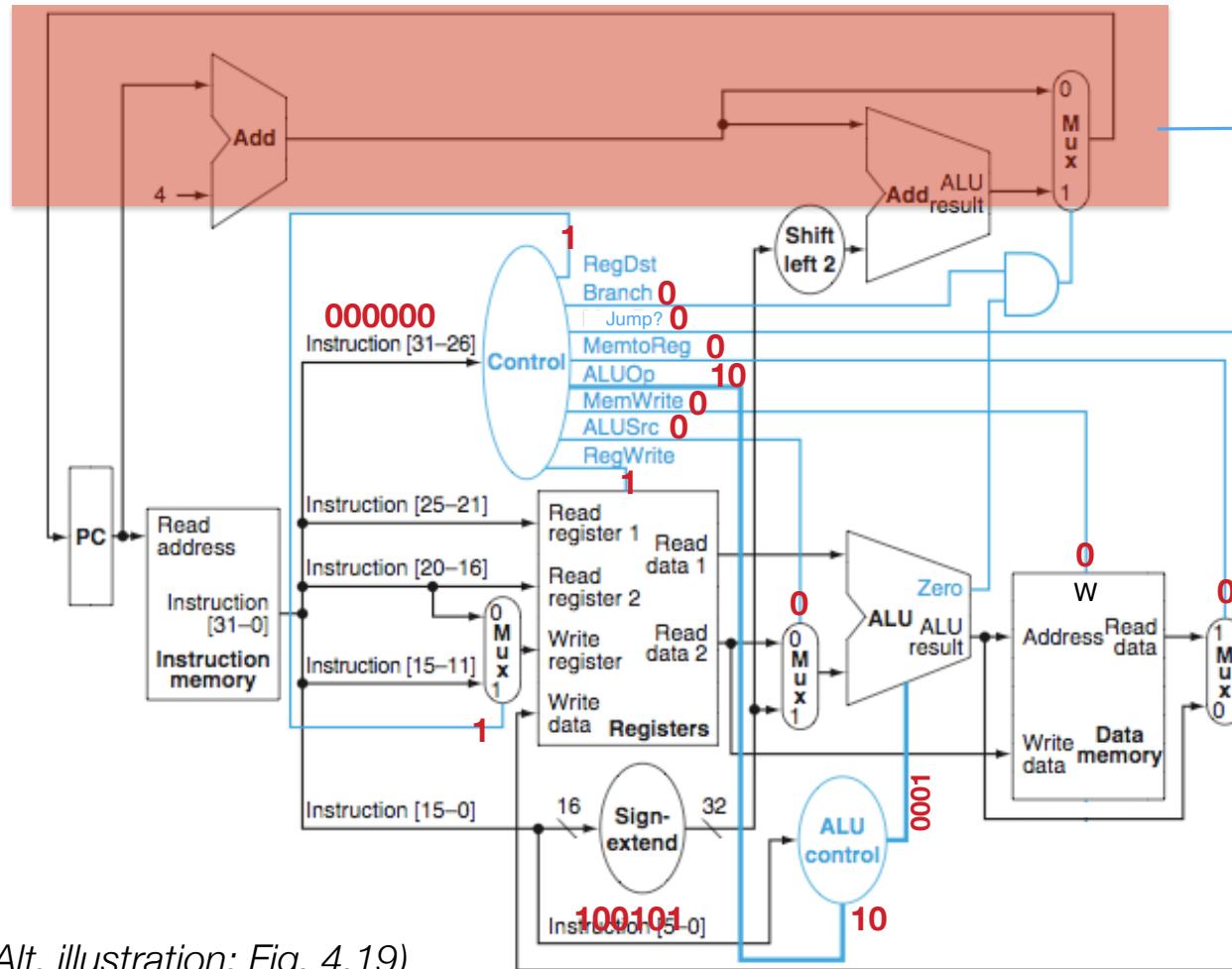
R-Type Instruction (OR)



(Alt. illustration: Fig. 4.19)



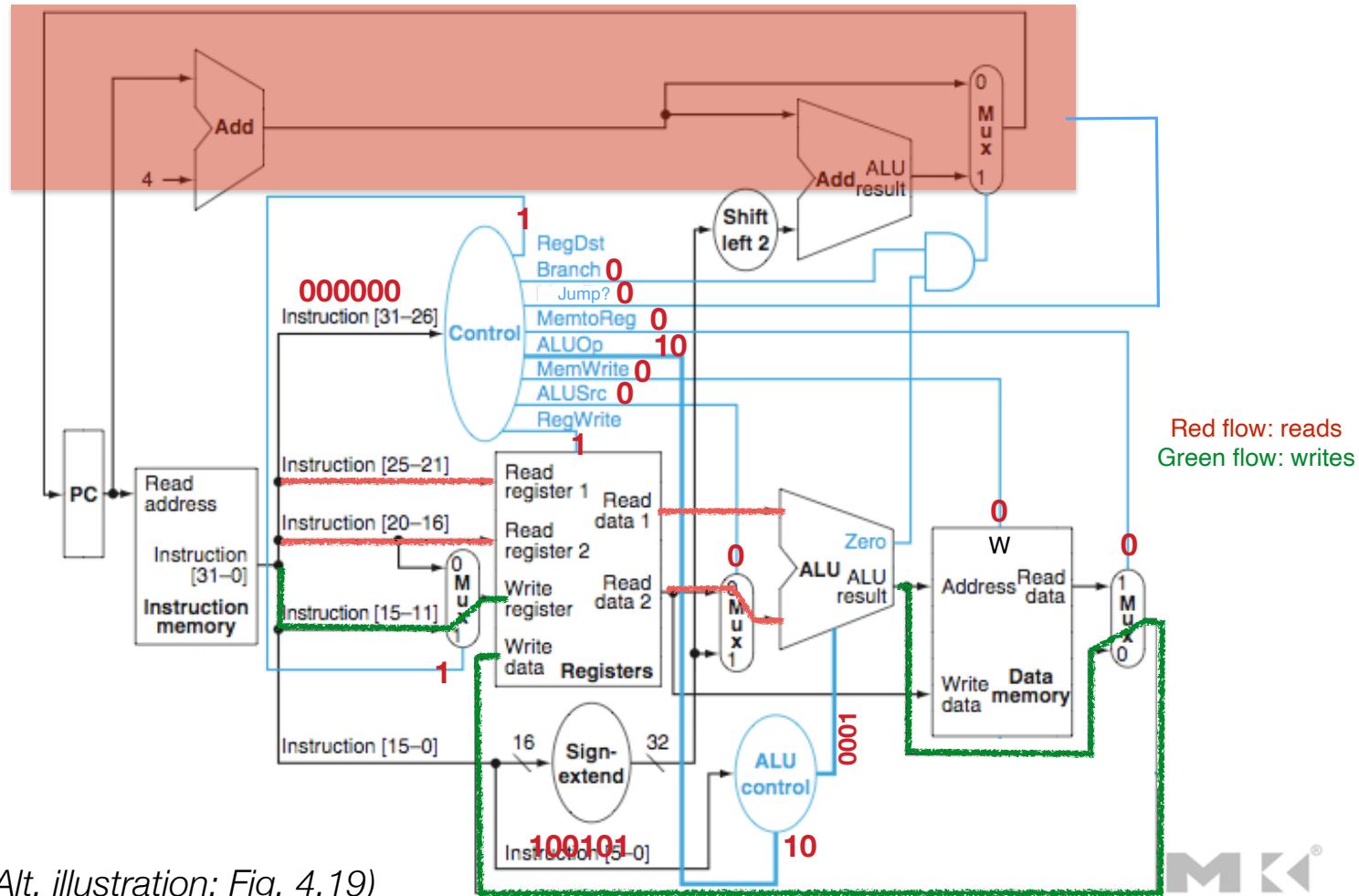
R-Type Instruction (OR)



(Alt. illustration: Fig. 4.19)



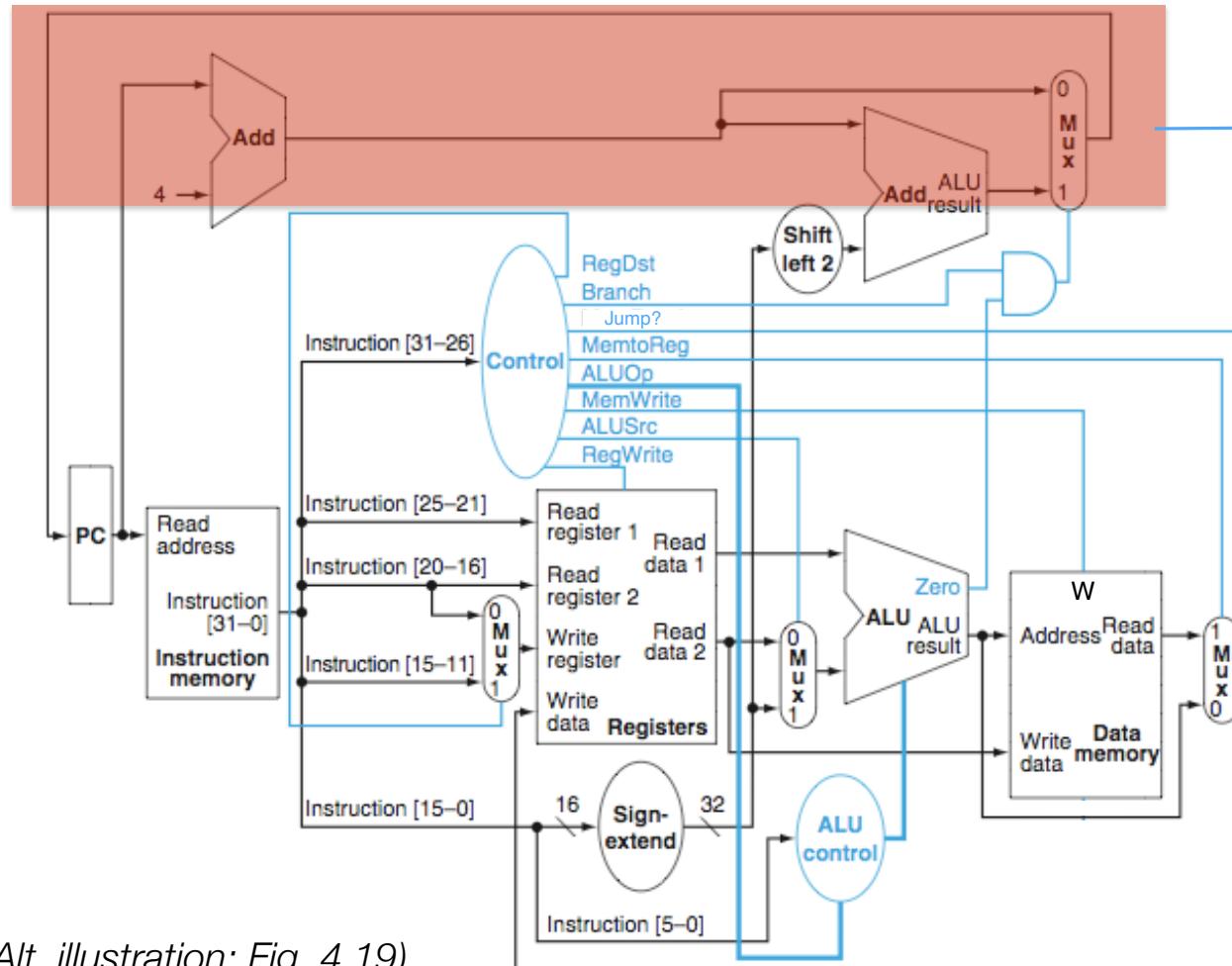
R-Type Instruction (OR): Used Data Flow



(Alt. illustration: Fig. 4.19)



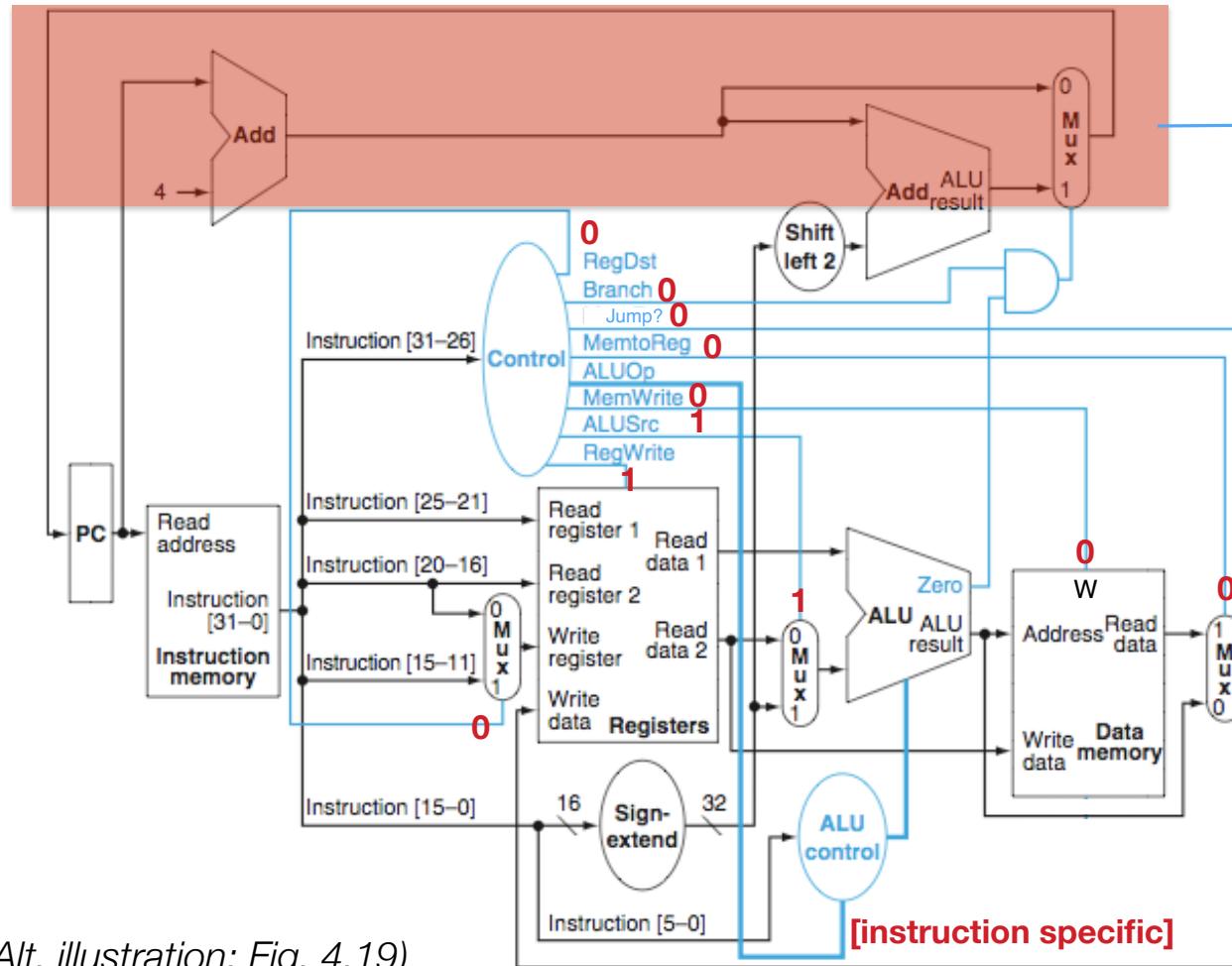
I-Type Instruction (generic)



(Alt. illustration: Fig. 4.19)



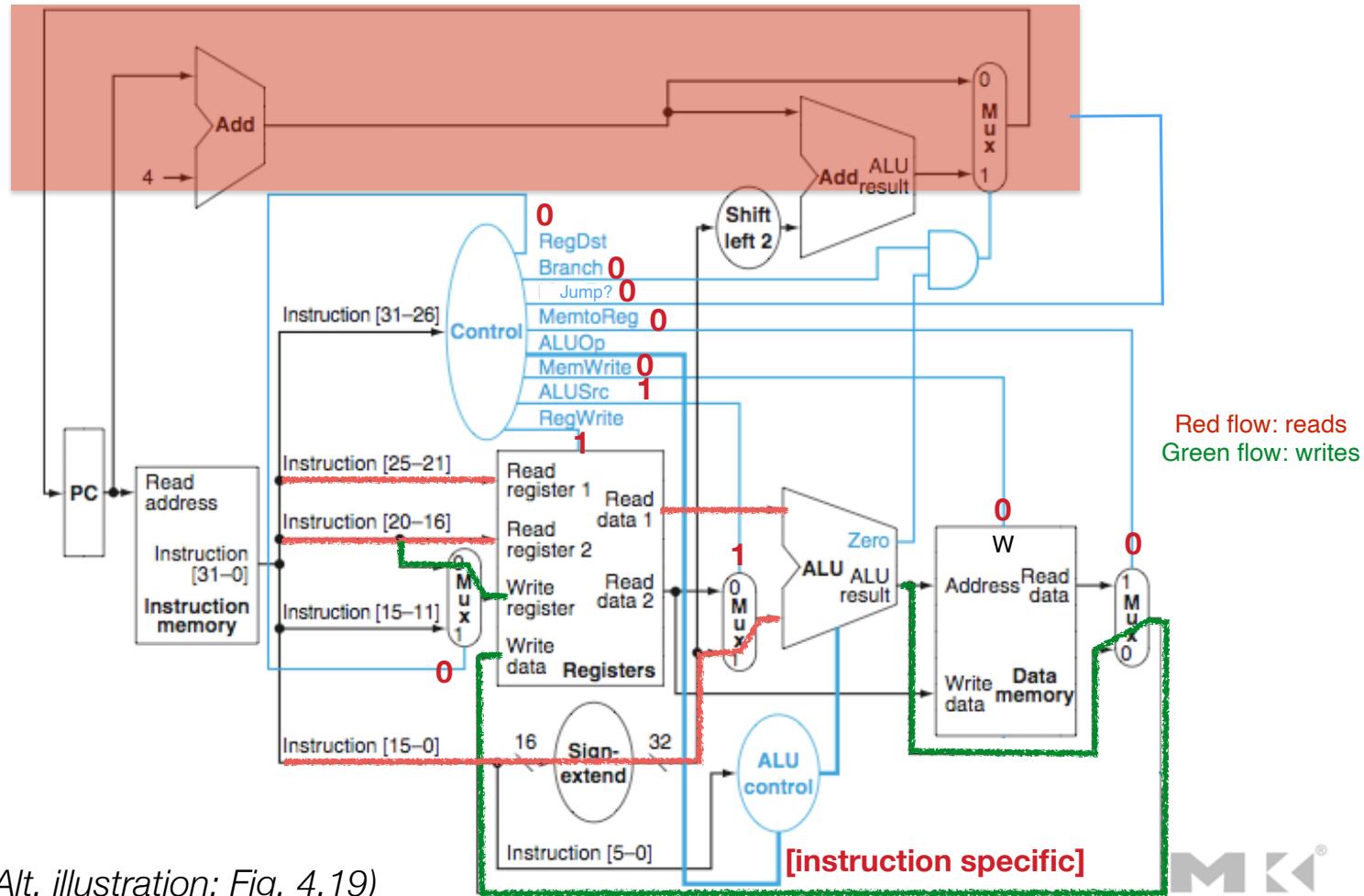
I-Type Instruction (generic)



(Alt. illustration: Fig. 4.19)



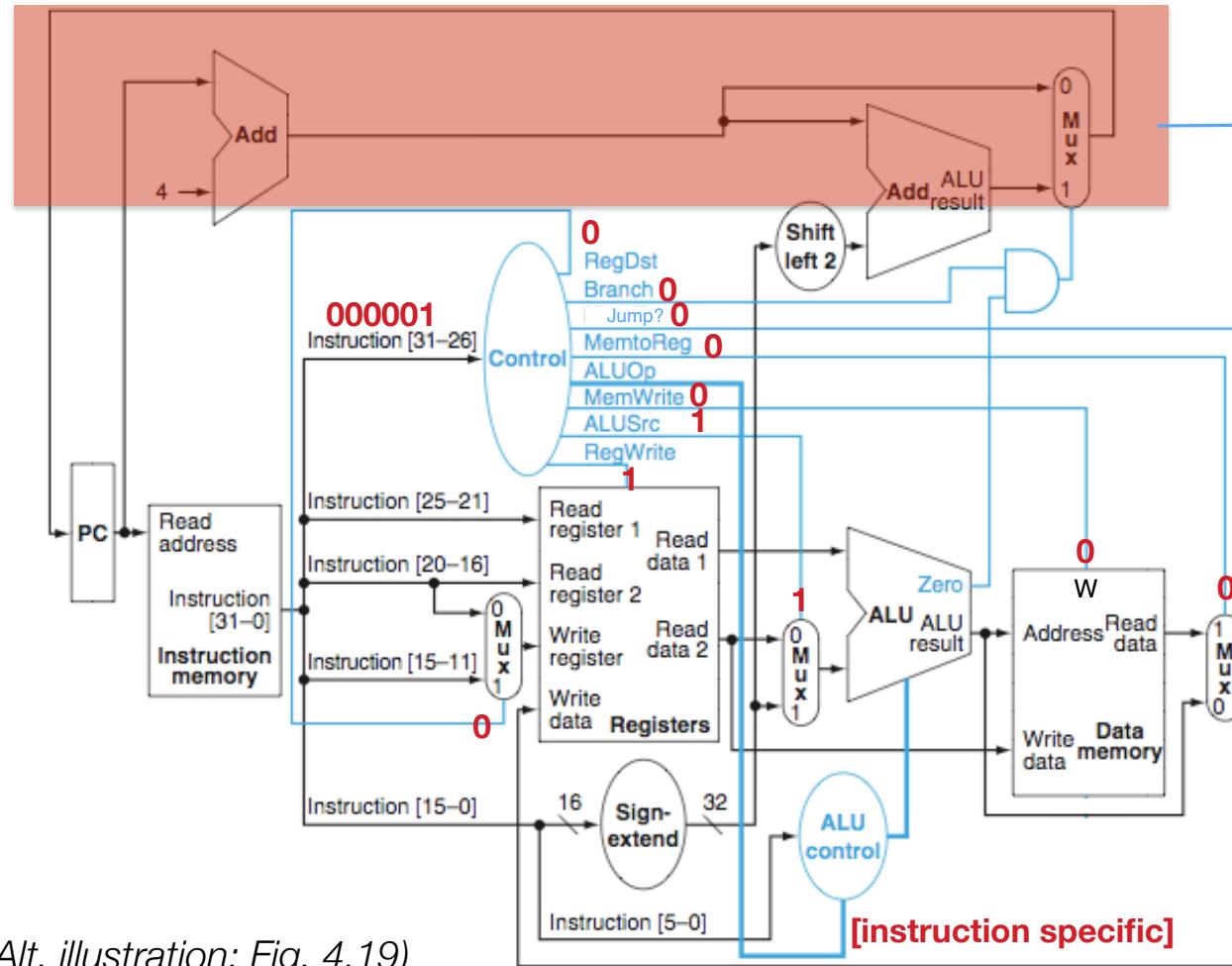
I-Type Instruction (generic): Used Data Flow



(Alt. illustration: Fig. 4.19)



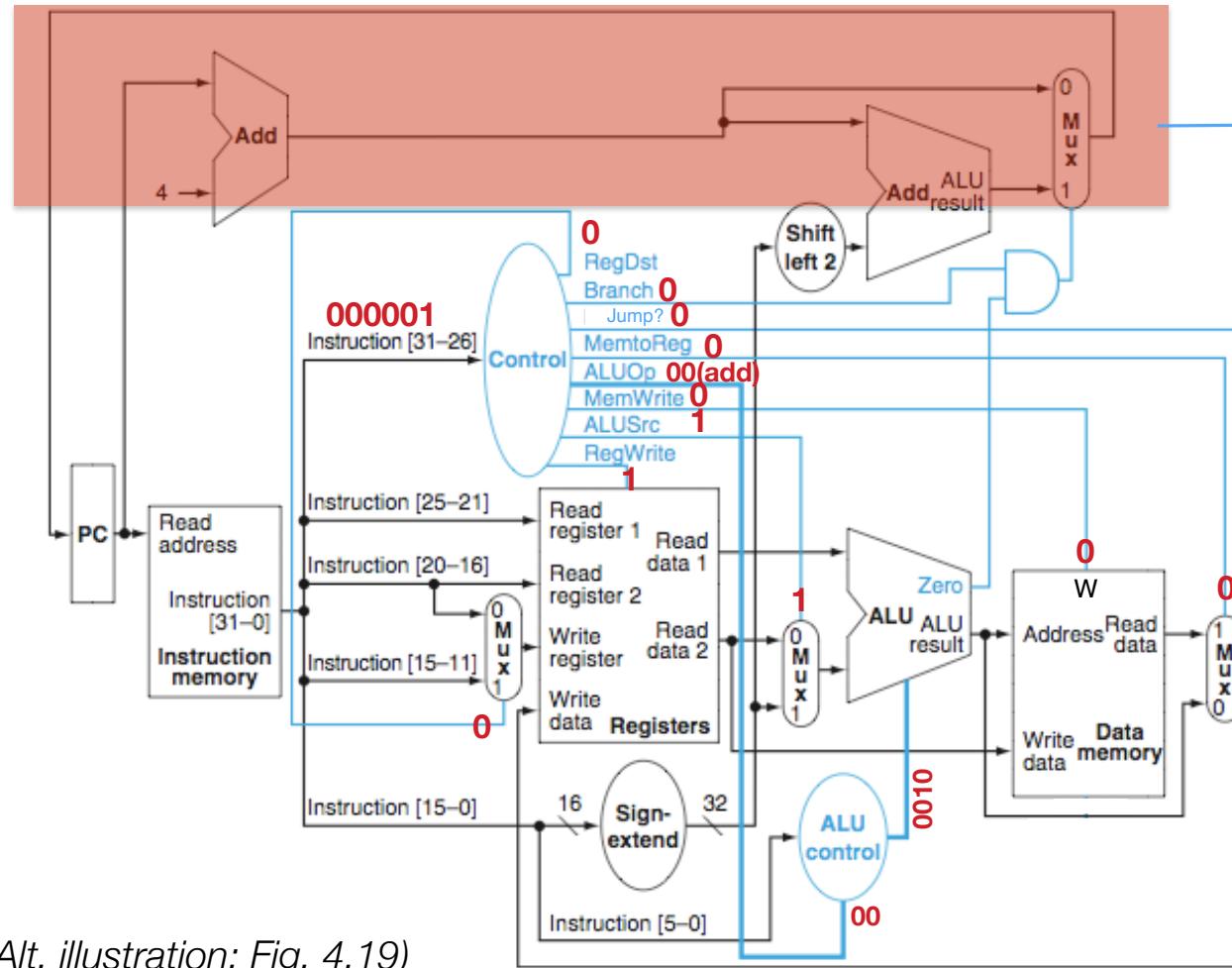
I-Type Instruction: addi



(Alt. illustration: Fig. 4.19)



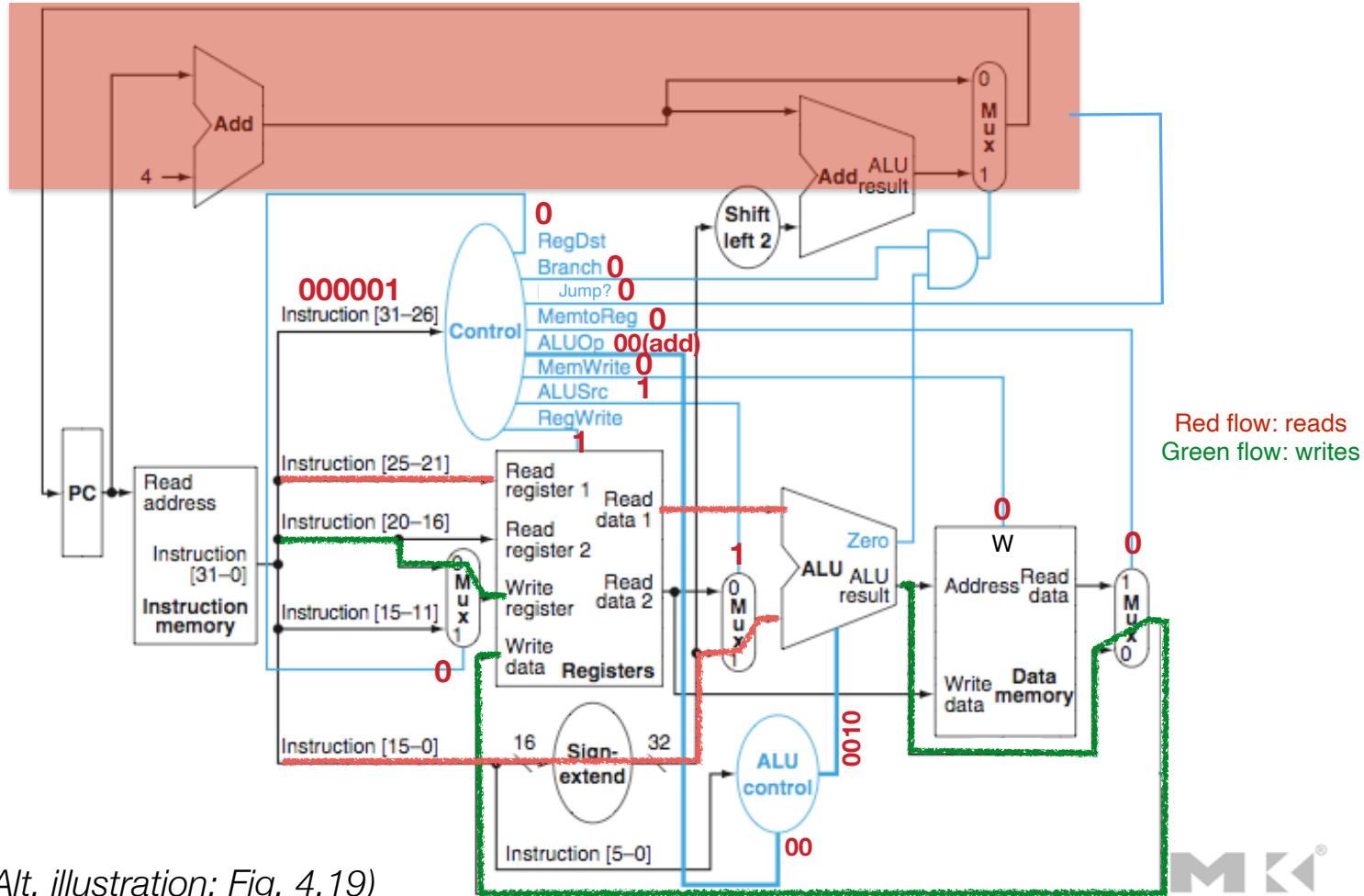
I-Type Instruction: addi



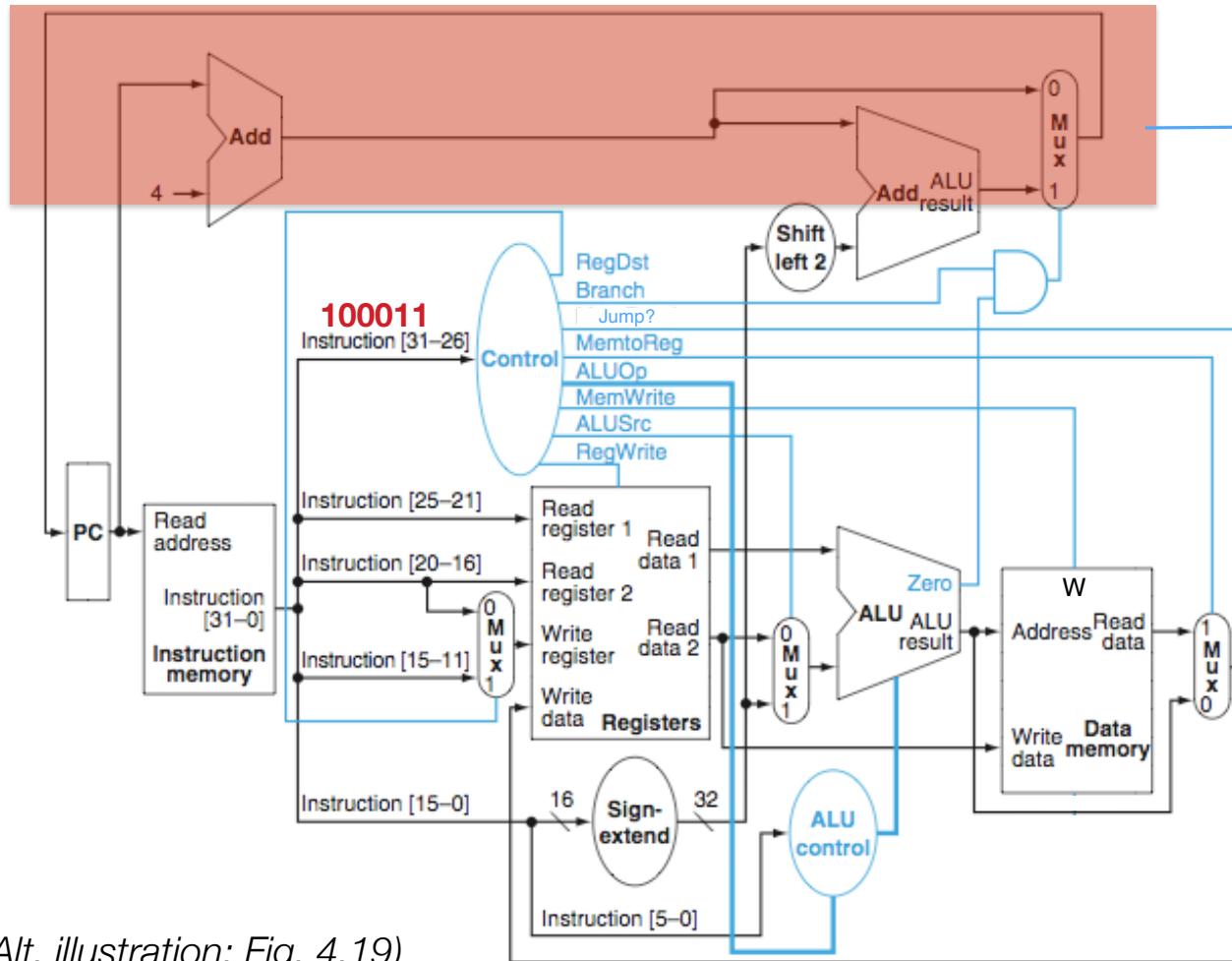
(Alt. illustration: Fig. 4.19)



I-Type Instruction: addi: Used Data Flow



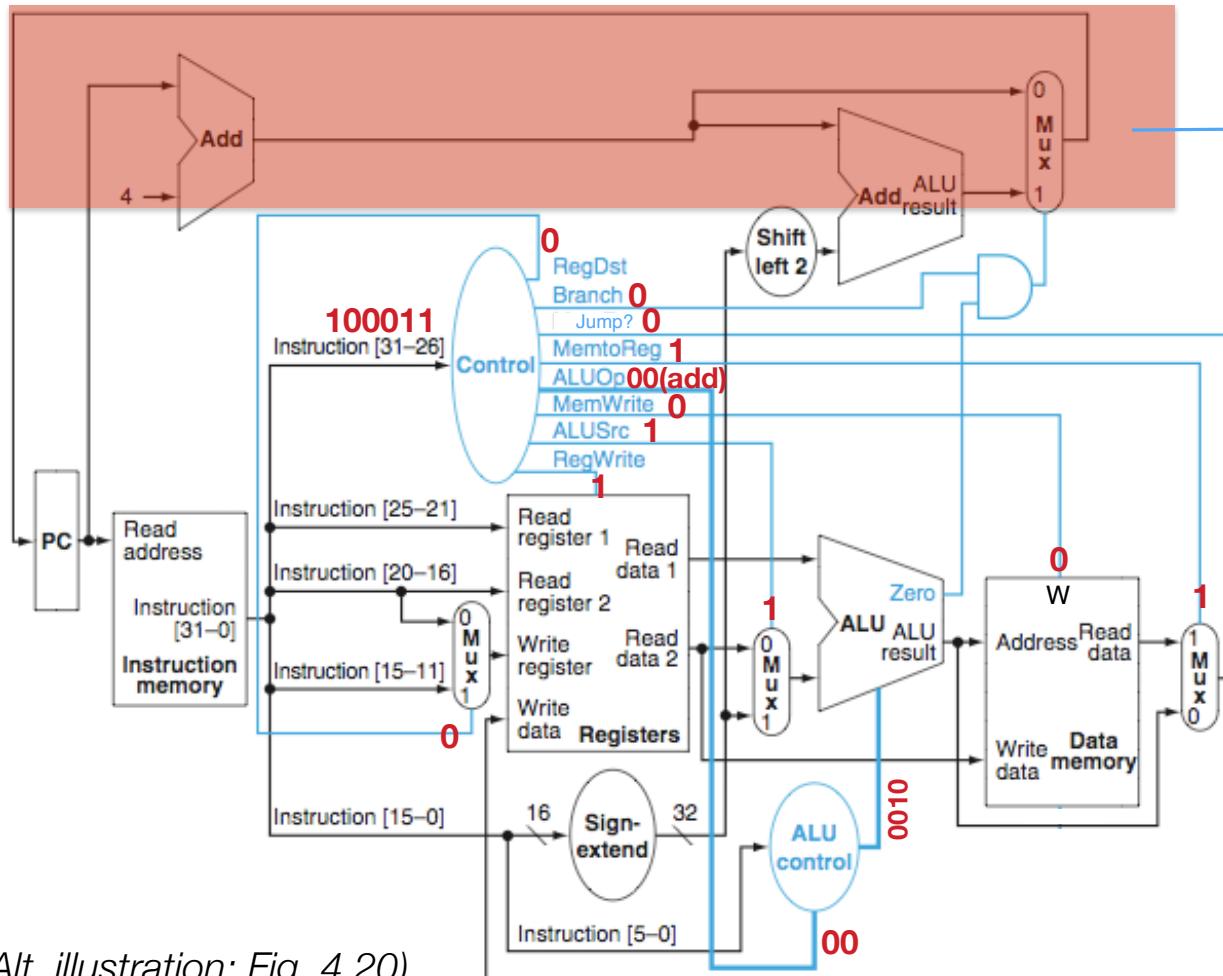
lw instruction (read from memory)



(Alt. illustration: Fig. 4.19)



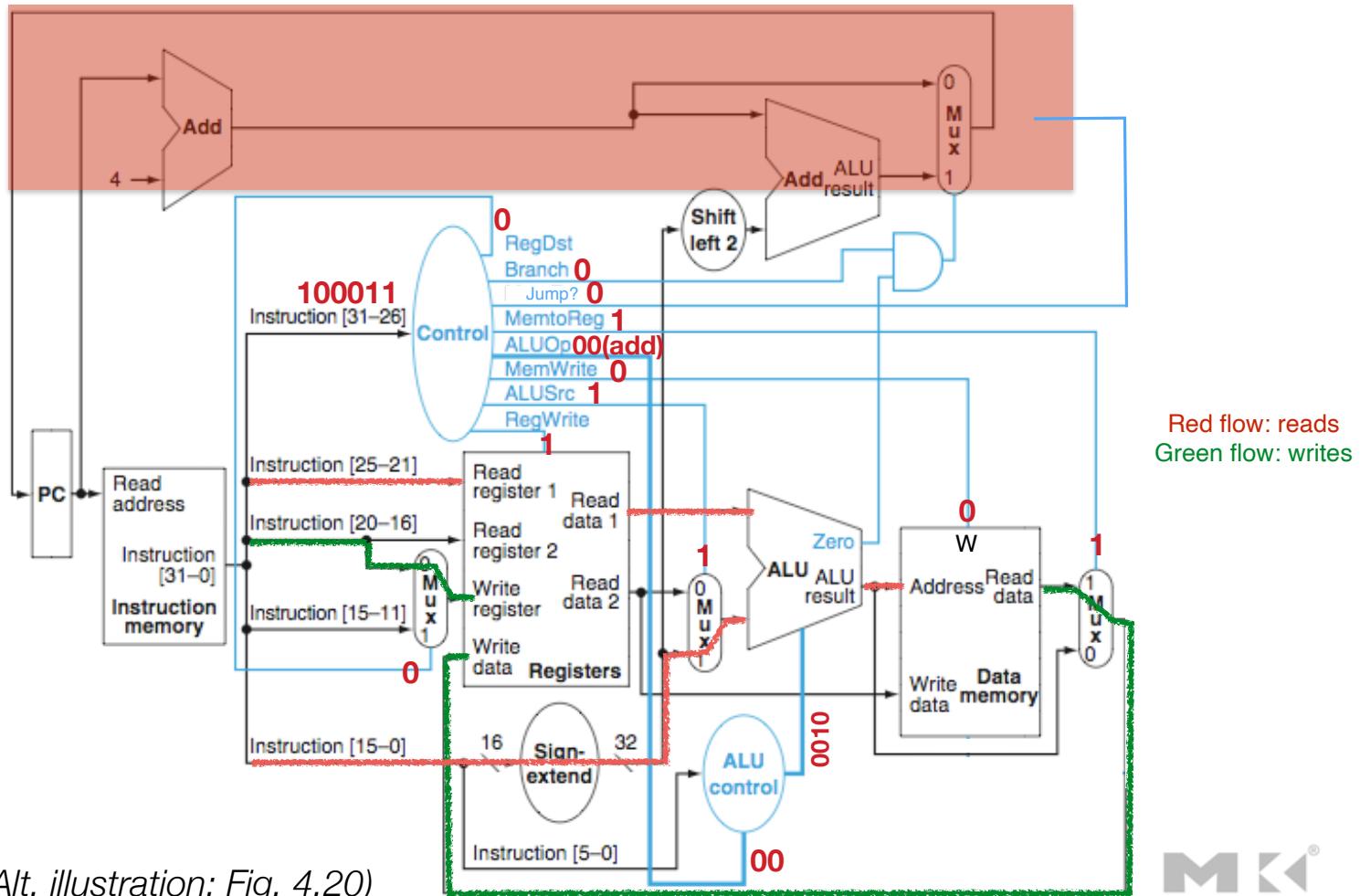
lw instruction (read from memory)



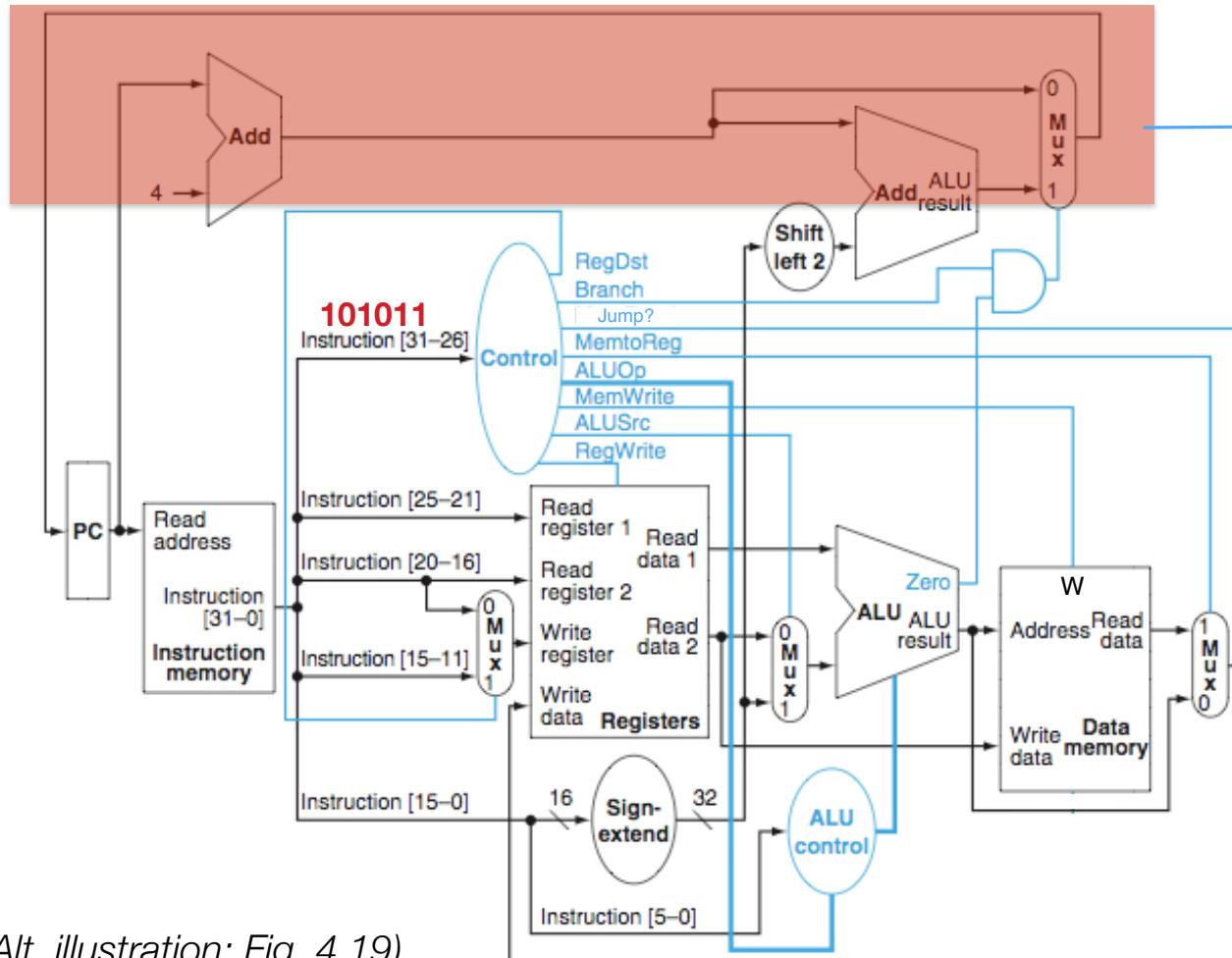
(Alt. illustration: Fig. 4.20)



lw instruction (read from memory): Used Data Flow



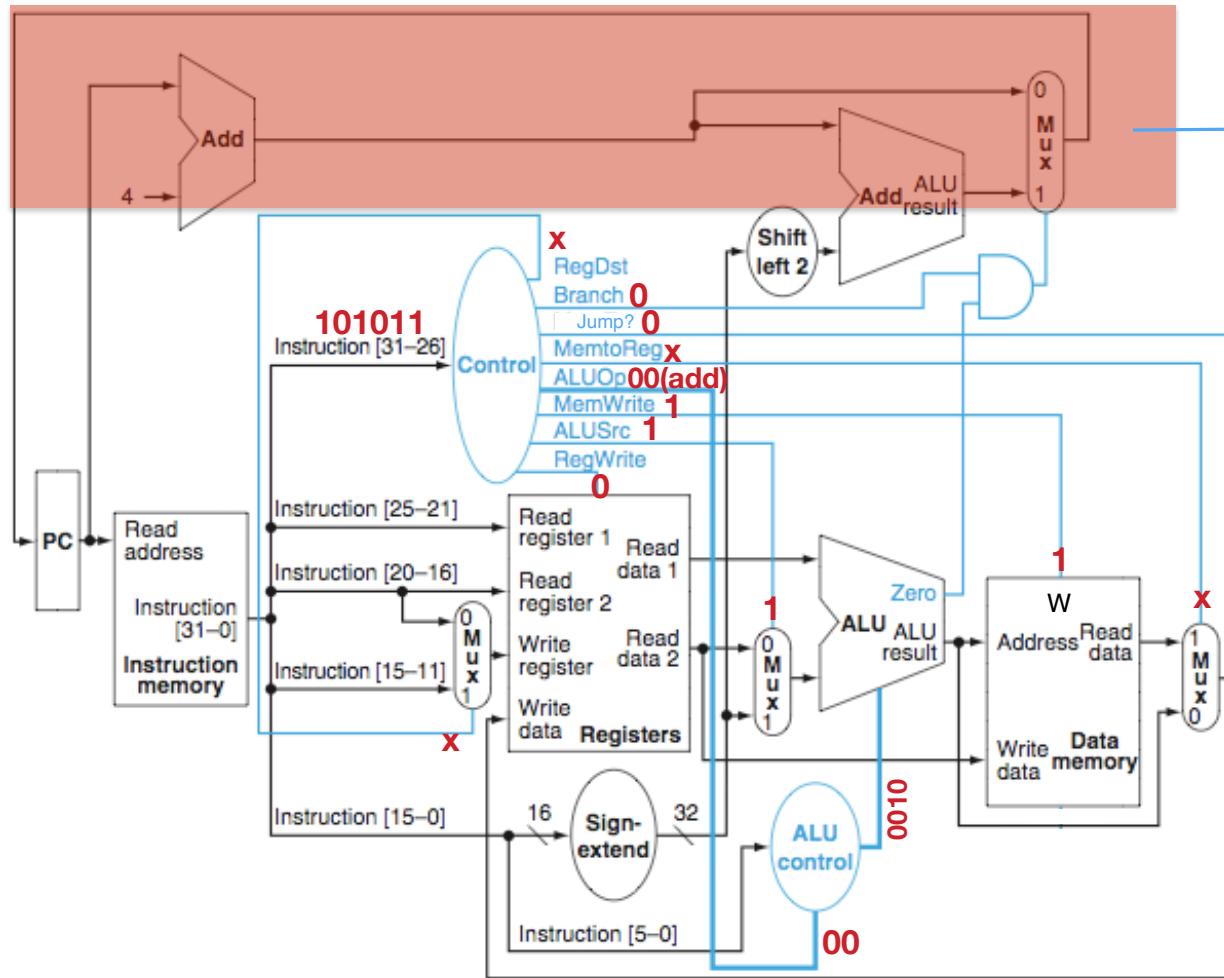
sw instruction (write to memory)



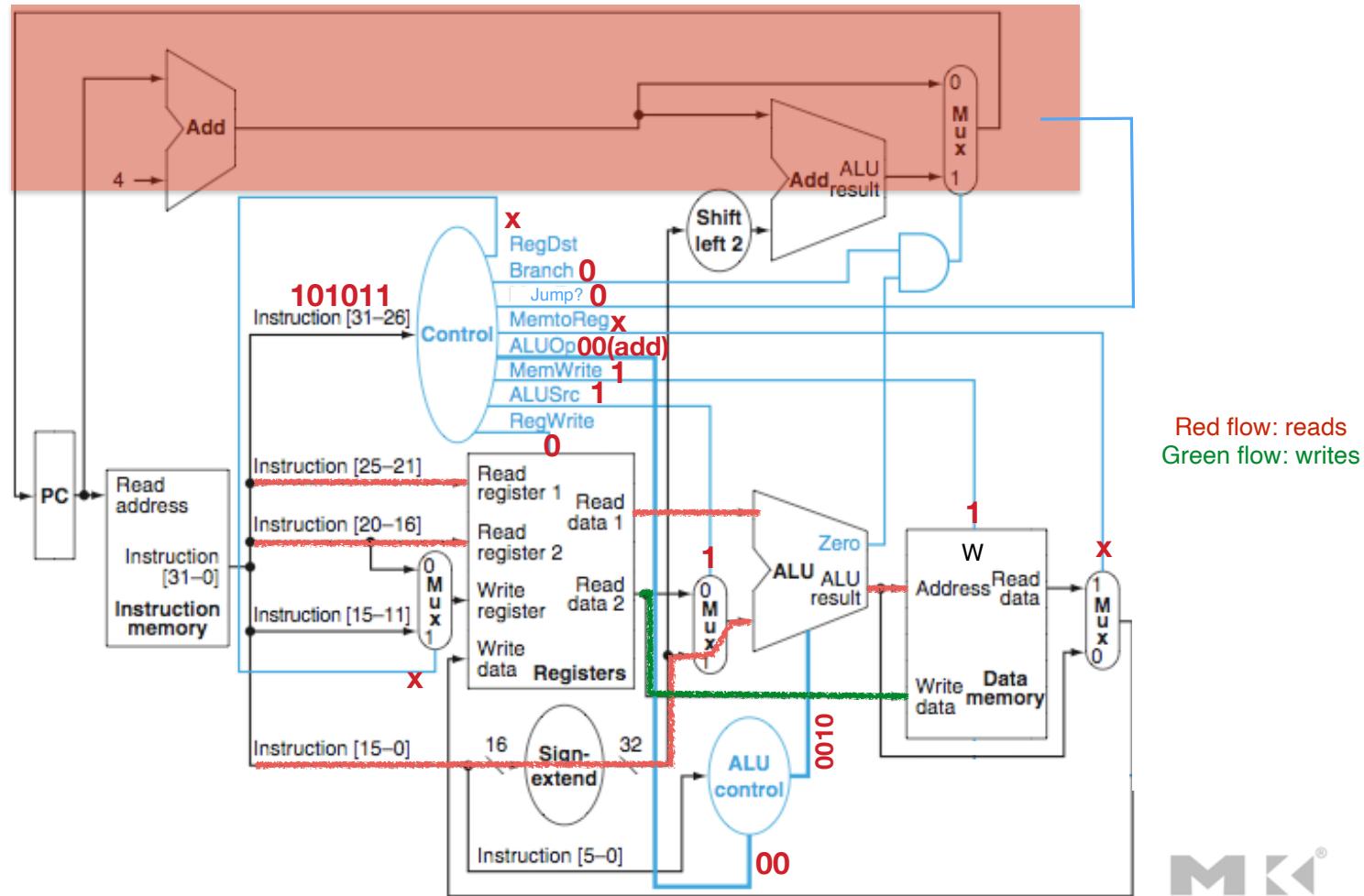
(Alt. illustration: Fig. 4.19)



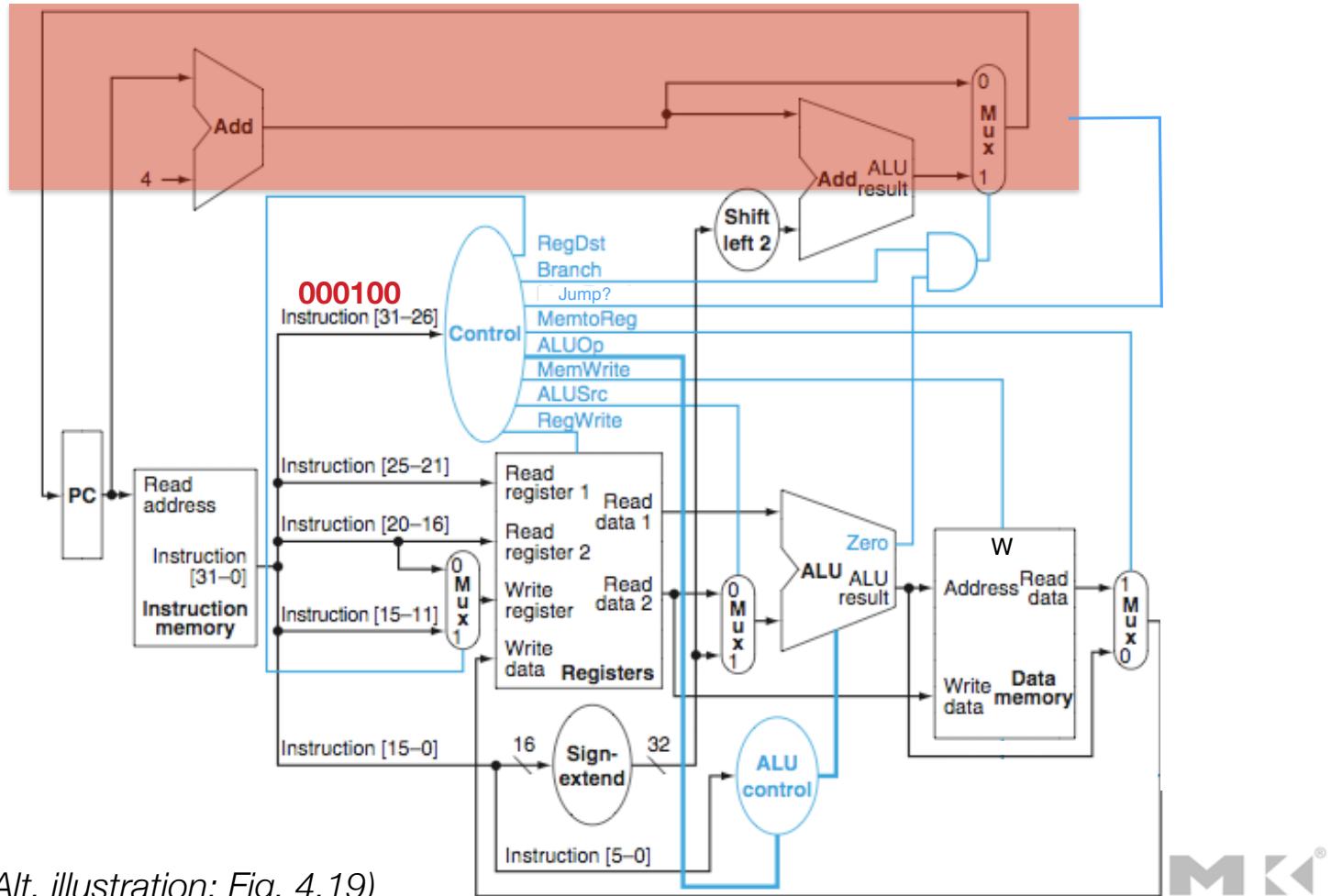
sw instruction (write to memory)



sw instruction (write to memory): Used Data Flow



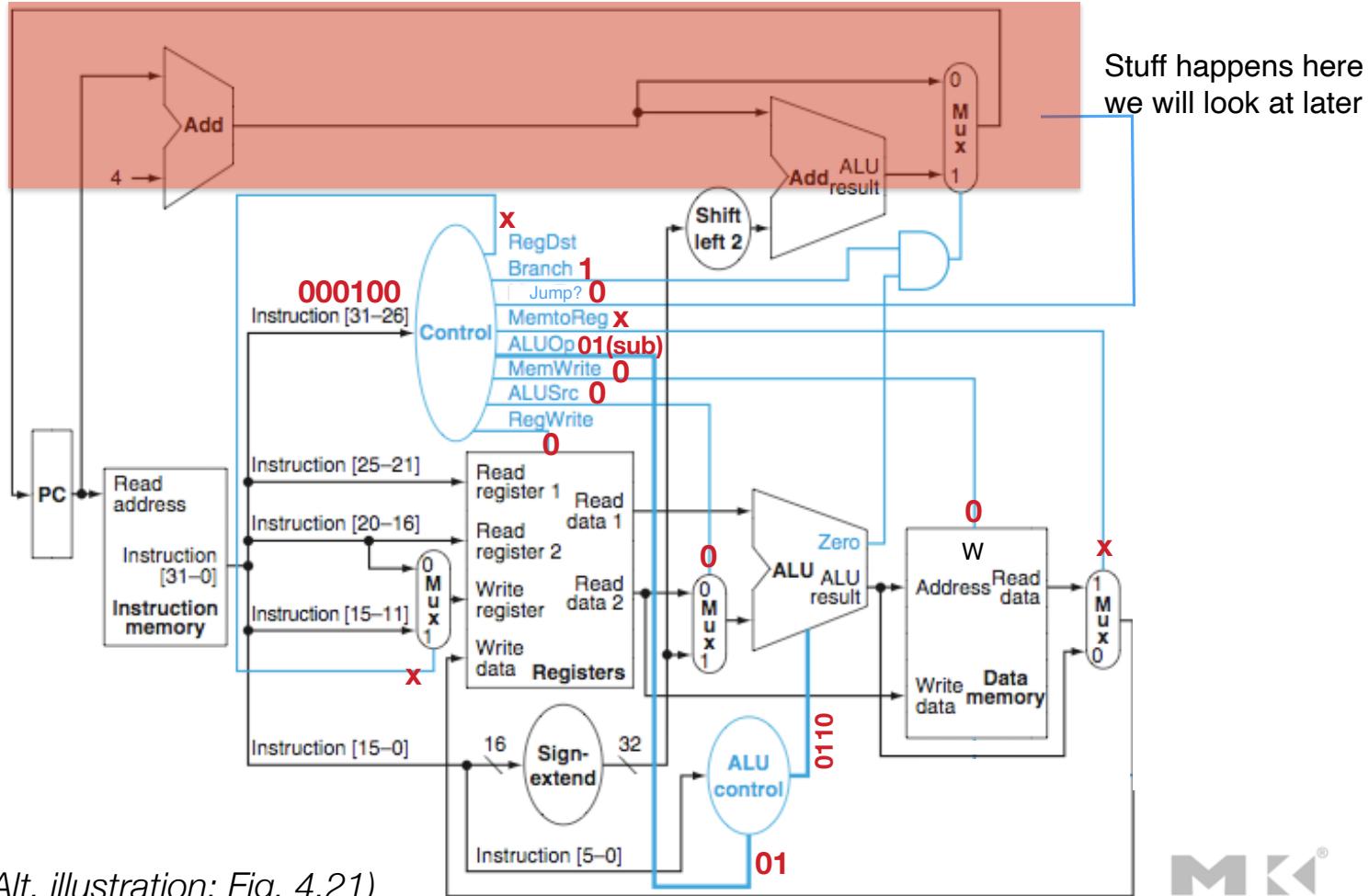
beq instruction



(Alt. illustration: Fig. 4.19)



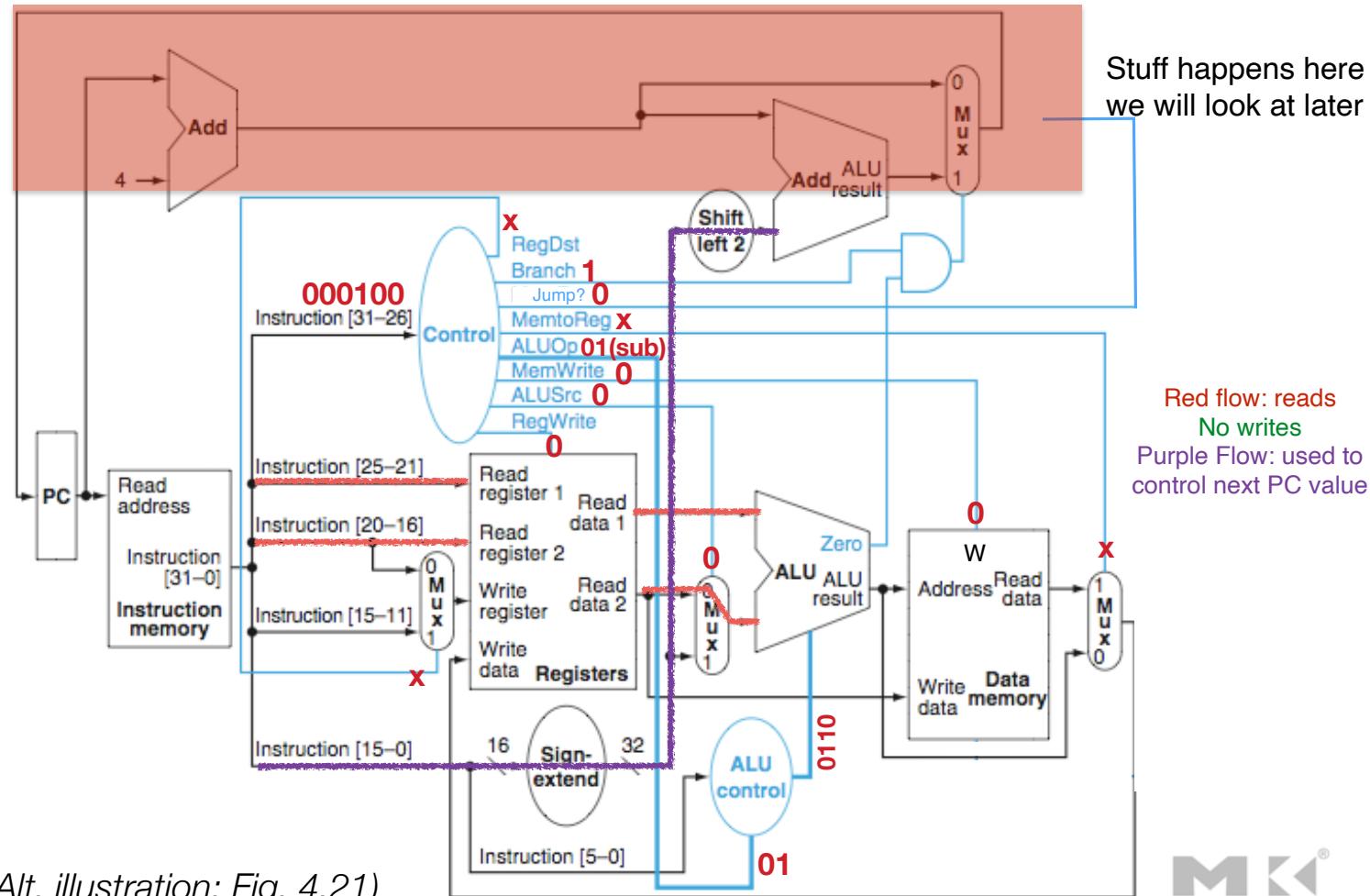
beq instruction



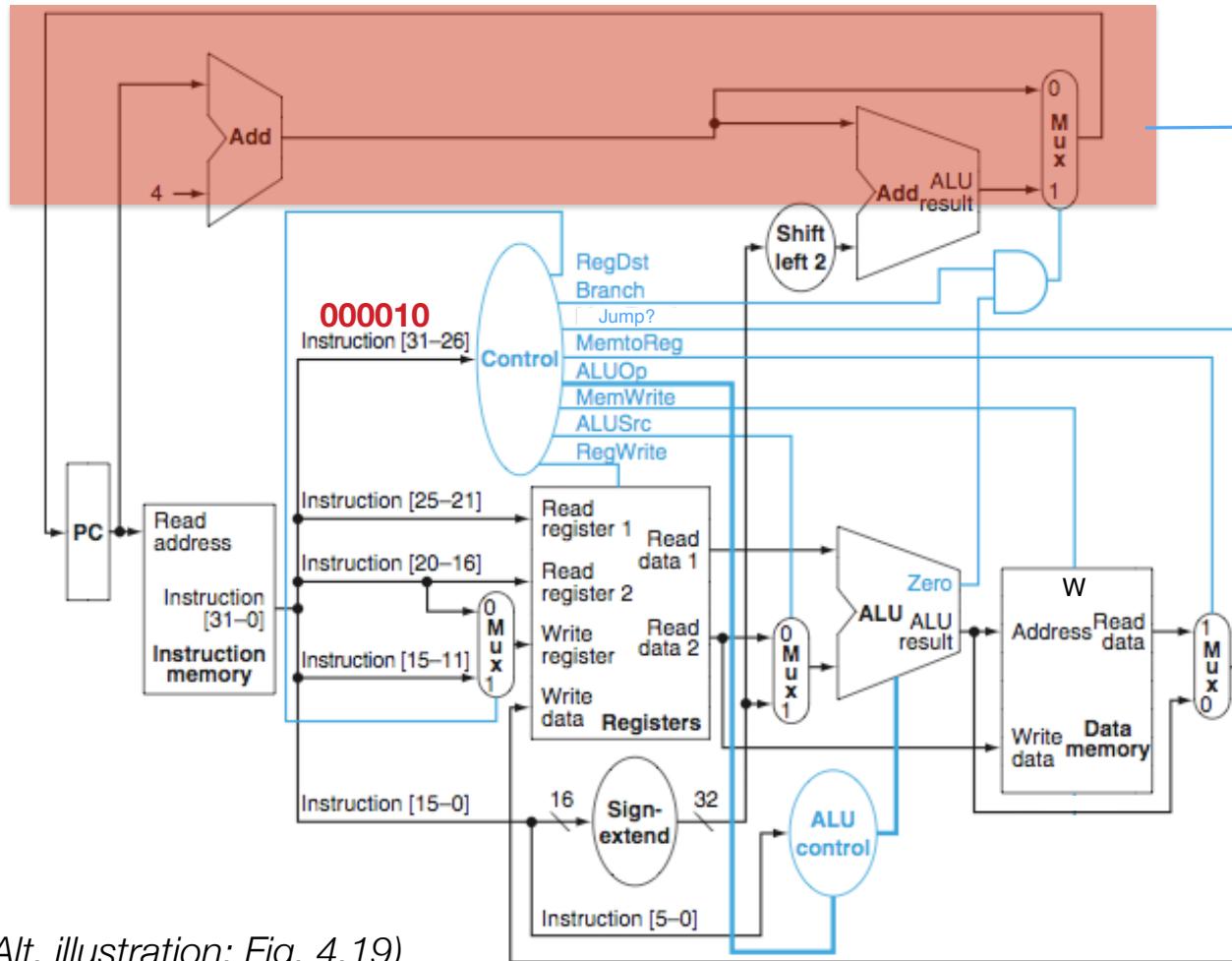
(Alt. illustration: Fig. 4.21)



beq instruction: Used Data Flow



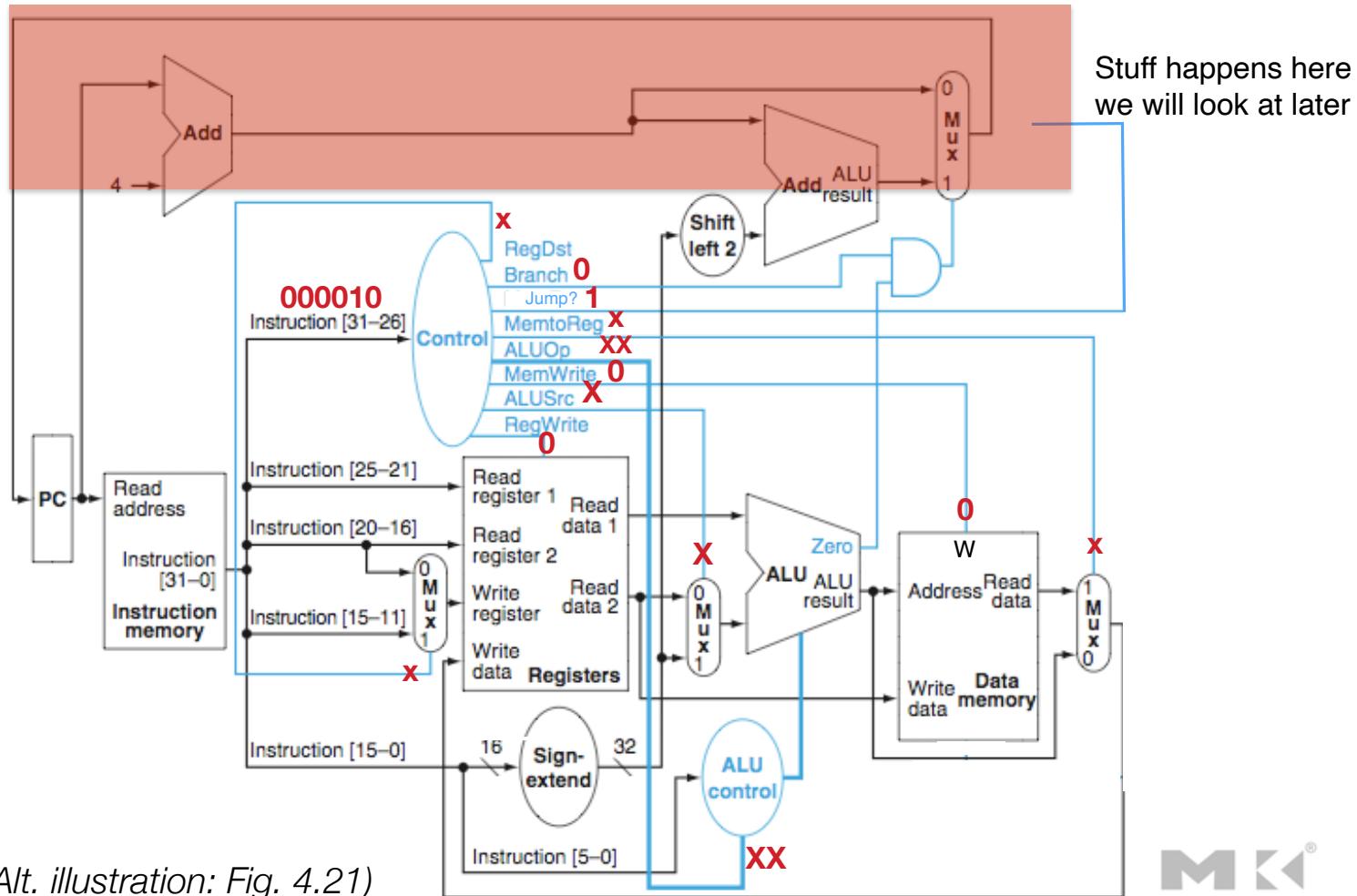
j instruction



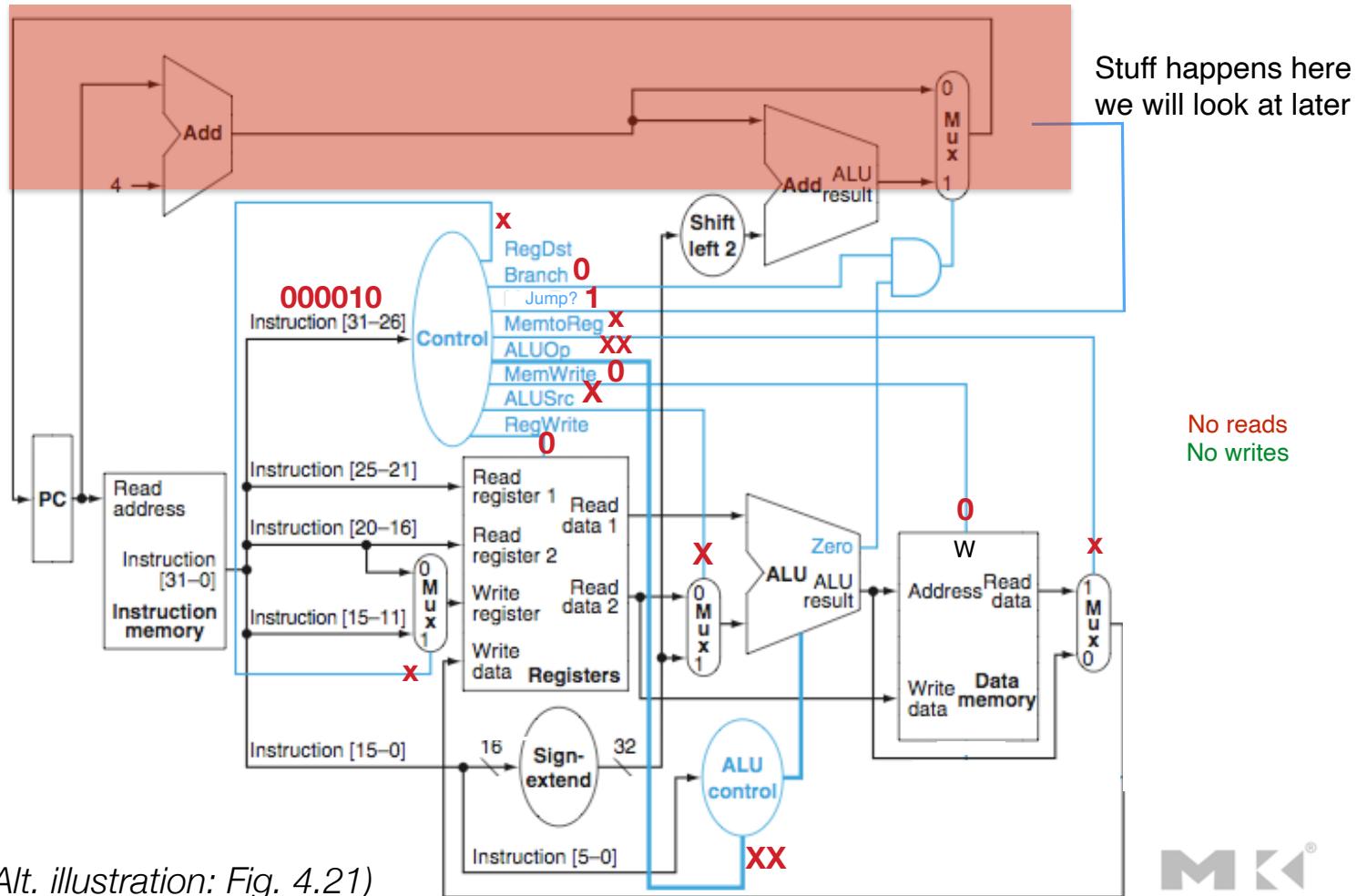
(Alt. illustration: Fig. 4.19)



j instruction



j instruction: Used Data Flow



Control
(i.e., modifying the PC)

Control Portion of CPU

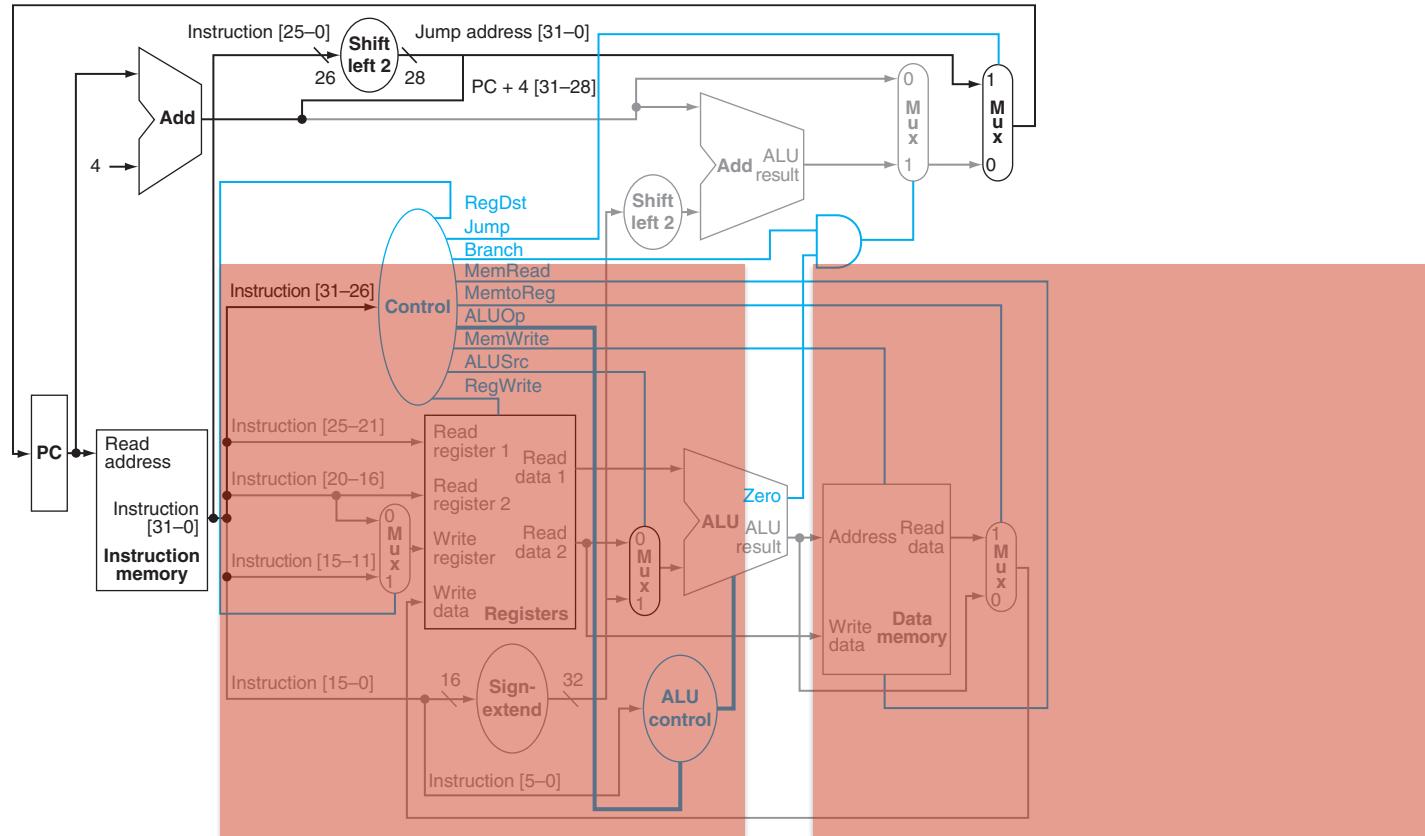


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.



Program Counter

- **Program Counter:** register that “points” to current address in instruction memory

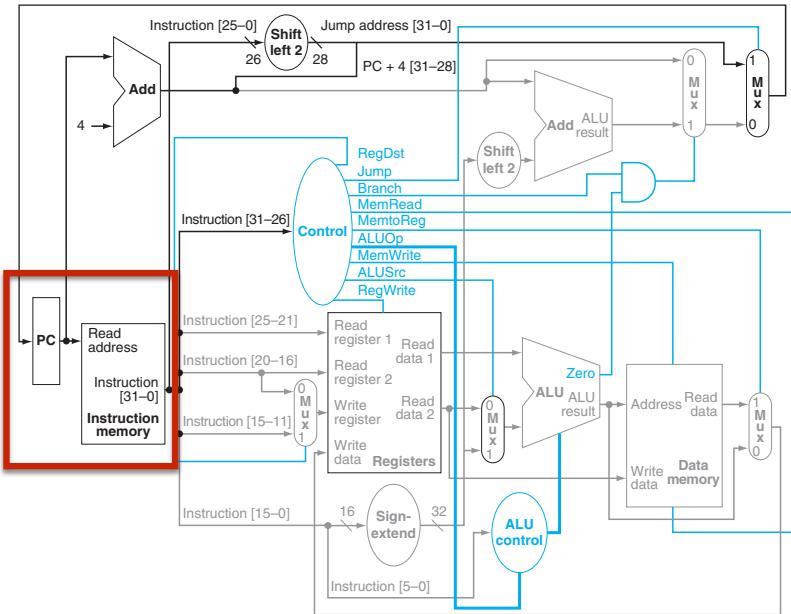
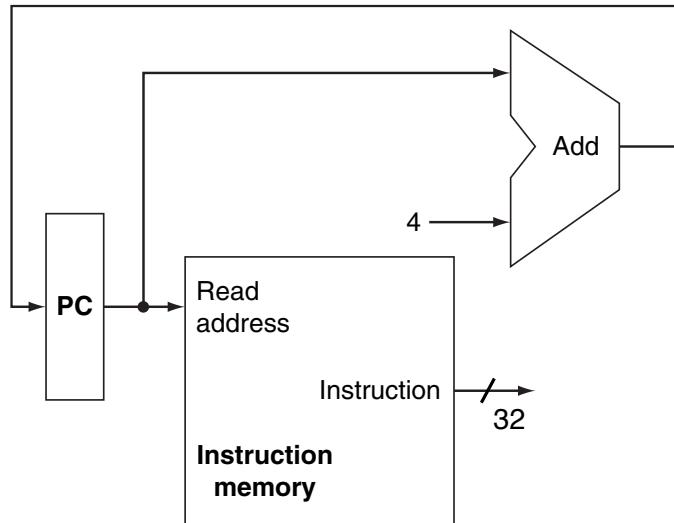


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target or either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of **PC + 4** as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.

Instruction is fetched, PC $\text{+=}4$

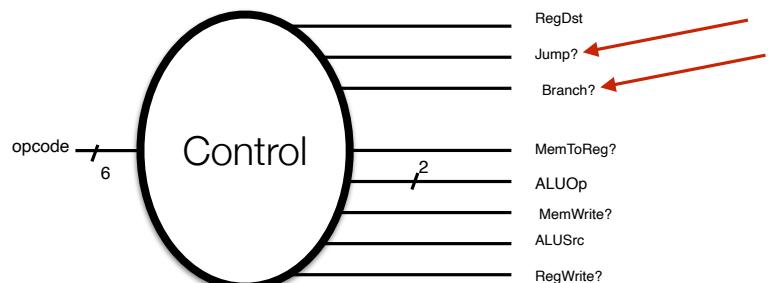
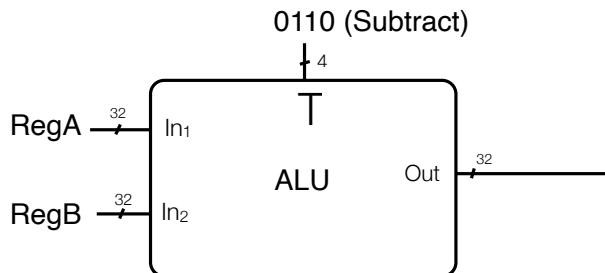
- 32 bit instruction retrieved from memory, PC set for next cycle (to next instruction)



- Without branching or jumps, $\text{PC} \text{ += } 4$ is control
- But these instructions exist, so let's build them in...
- We will build in **j** and **beq**, leaving out others...

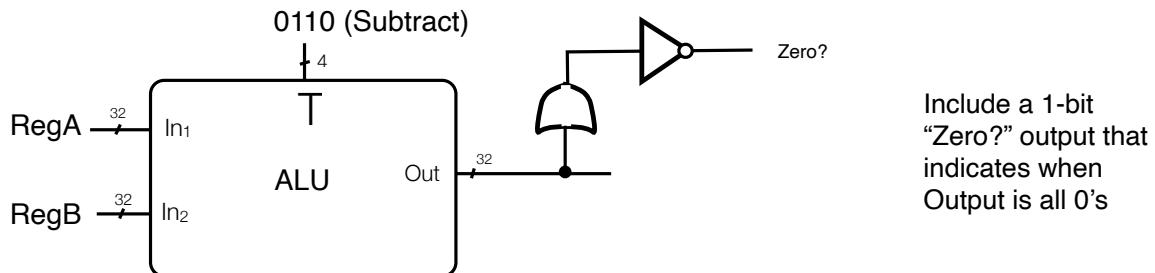
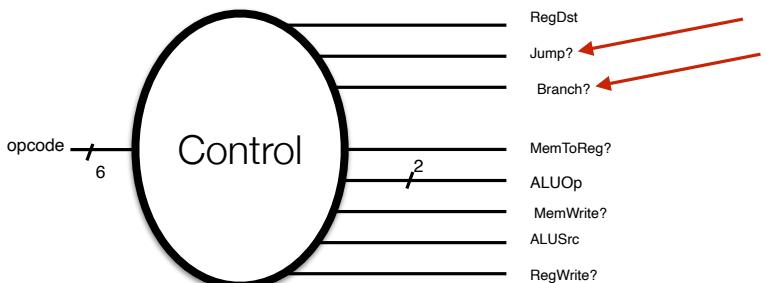
Datapath contributions to control

- Recall that Data path provides some support for control:
 - Pulled from 32-bit instruction:
 - 26-bit **JmpAddr** field
 - 16-bit **const** field sign-extended to 32 bits
 - OpCode “Control” provided info about whether instruction was a jump or branch
 - For branch instructions, ALU used to evaluate conditional



Datapath contributions to control

- Recall that Data path provides some support for control:
 - Pulled from 32-bit instruction:
 - 26-bit **JmpAddr** field
 - 16-bit **const** field sign-extended to 32 bits
 - OpCode “Control” provided info about whether instruction was a jump or branch
 - For branch instructions, ALU used to evaluate conditional



Review of Jump and Branch

- Jump: After PC += 4, set PC to [4 highest order bits of PC] . JumpAddr . 00
 - Branch: If conditional is true, then $\text{PC} = (\text{PC} + 4) + 4 * \text{const}$
 - Recall: the constant represents the **# of instructions to skip** if the branch is taken

```
beq $t0, $zero, L1  
.....  
addi $v0, $zero, 1  
addi $sp, $sp, 8  
jr $ra  
.....  
L1: addi $a0, $a0, -1  
.....  
L2: addi $a0, $a0, -1  
beq $a0, $zero, L2  
.....
```

const = 3

const = -2

PC Adjustment pseudocode

```
PC += 4;  
  
If (Branch? && Zero?) {  
  
    PC += (const<<2);  
  
}  
  
If (Jump) {  
  
    PC = PC[31-28] . JmpAddr . 00  
  
}
```

- **Always** increment PC by 4 (next instruction) before making branch or jump adjustments
- For branch:
 - why Zero? must be true?
 - why const<<2?

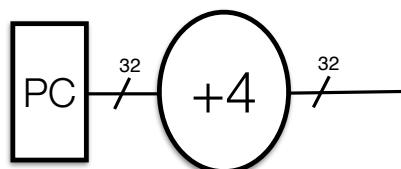
PC Adjustment pseudocode

```
PC += 4;  
  
If (Branch? && Zero?) {  
  
    PC += (const<<2);  
  
}  
  
If (Jump) {  
  
    PC = PC[31-28] . JmpAddr . 00  
  
}
```

- **Always** increment PC by 4 (next instruction) before making branch or jump adjustments
- For branch:
 - why Zero? must be true?
 - conditional must be true (compared values from registers equal)
 - why const<<2?
 - Need to shift by words of memory (i.e., 4 x # bytes)

PC Adjustment Circuitry

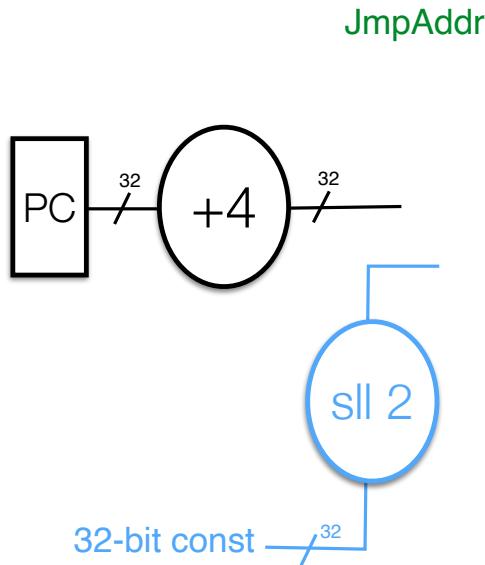
JmpAddr



32-bit const

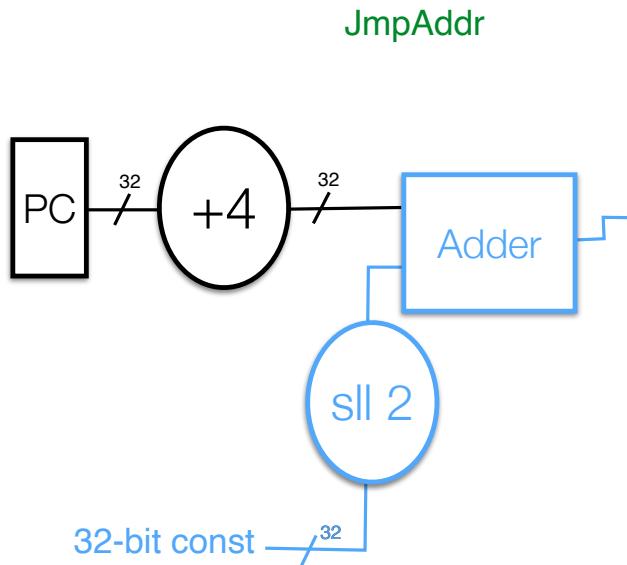
- JumpAddr: 26-bit pseudo-direct add pulled from instruction
- 32-bit const: converted from the 16-bit const pulled from instruction

PC Adjustment Circuitry



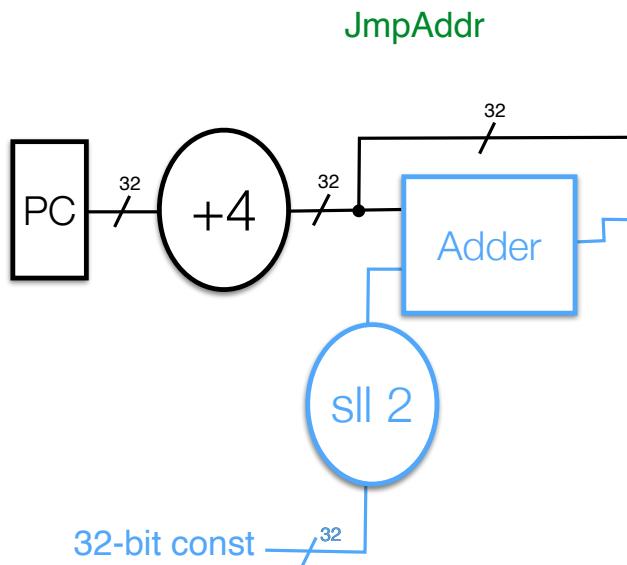
- Determine branch address as $(PC+4)+4*const$

PC Adjustment Circuitry



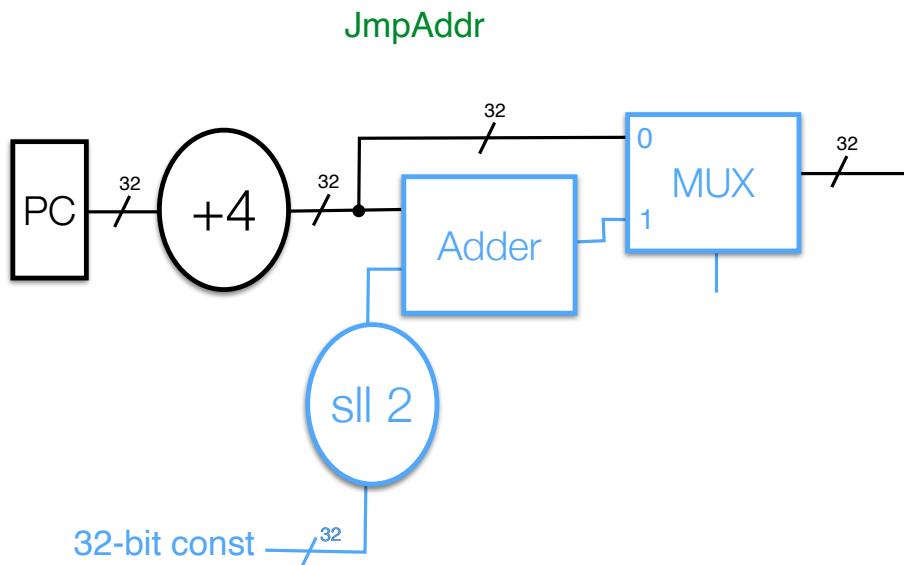
- Determine branch address as $(PC+4)+4*const$

PC Adjustment Circuitry



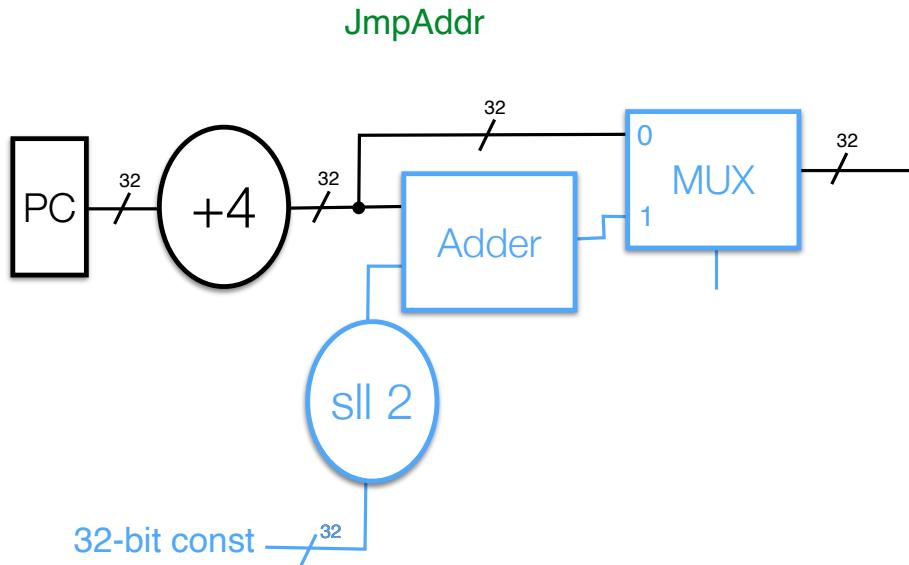
- Enable choice between PC+4 and branch address

PC Adjustment Circuitry

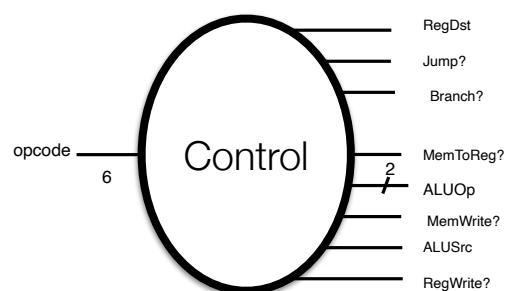
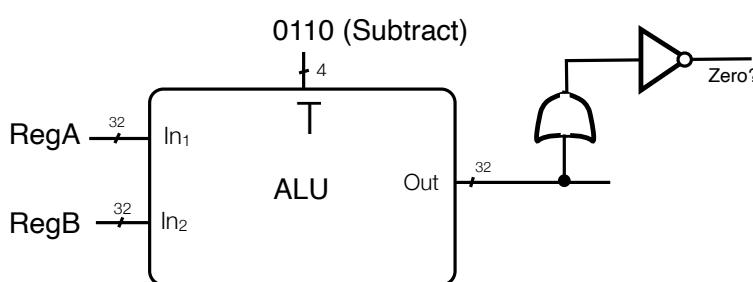


- Enable choice between PC+4 and branch address

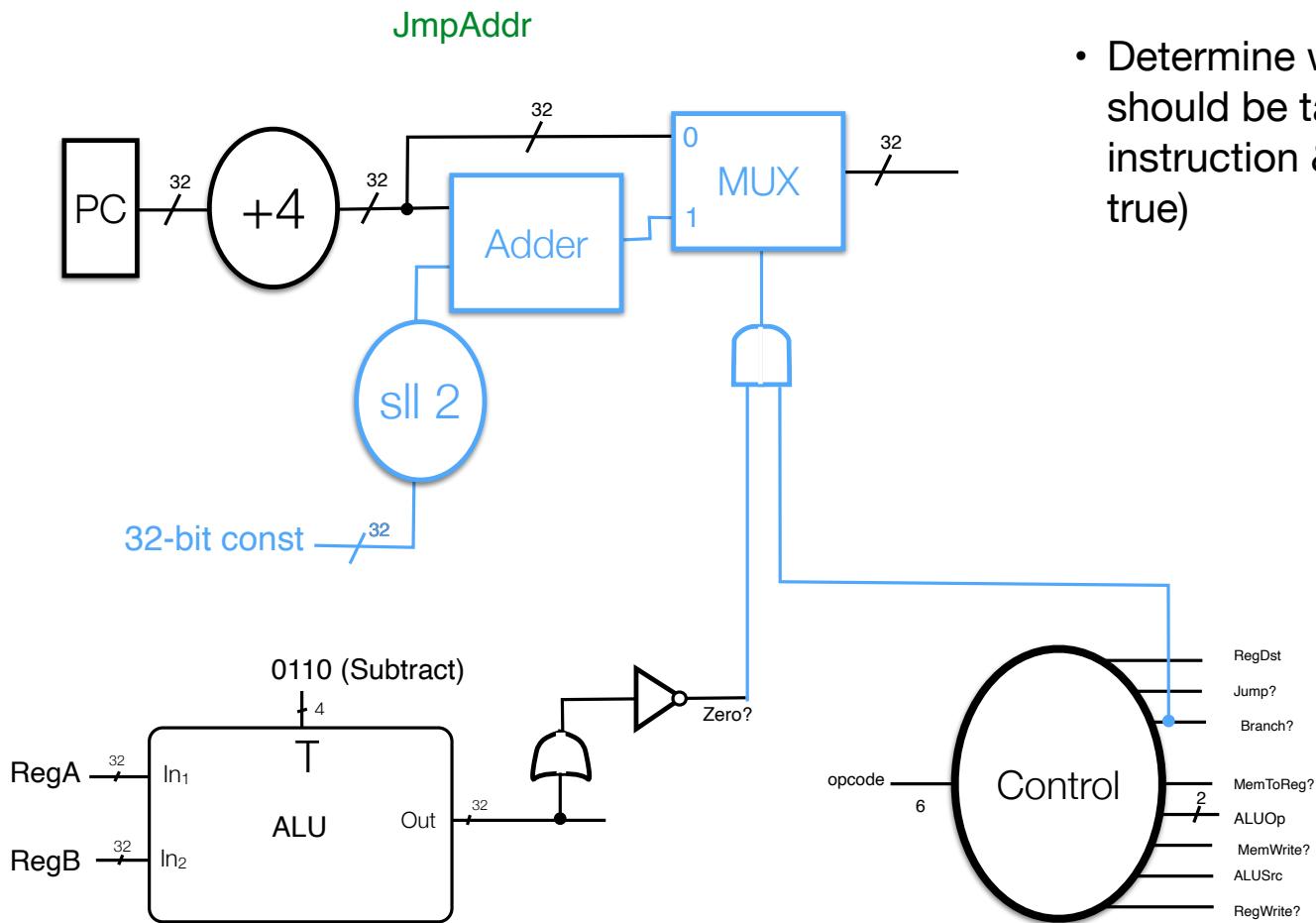
PC Adjustment Circuitry



- Determine when Branch should be taken (branch instruction & conditional true)

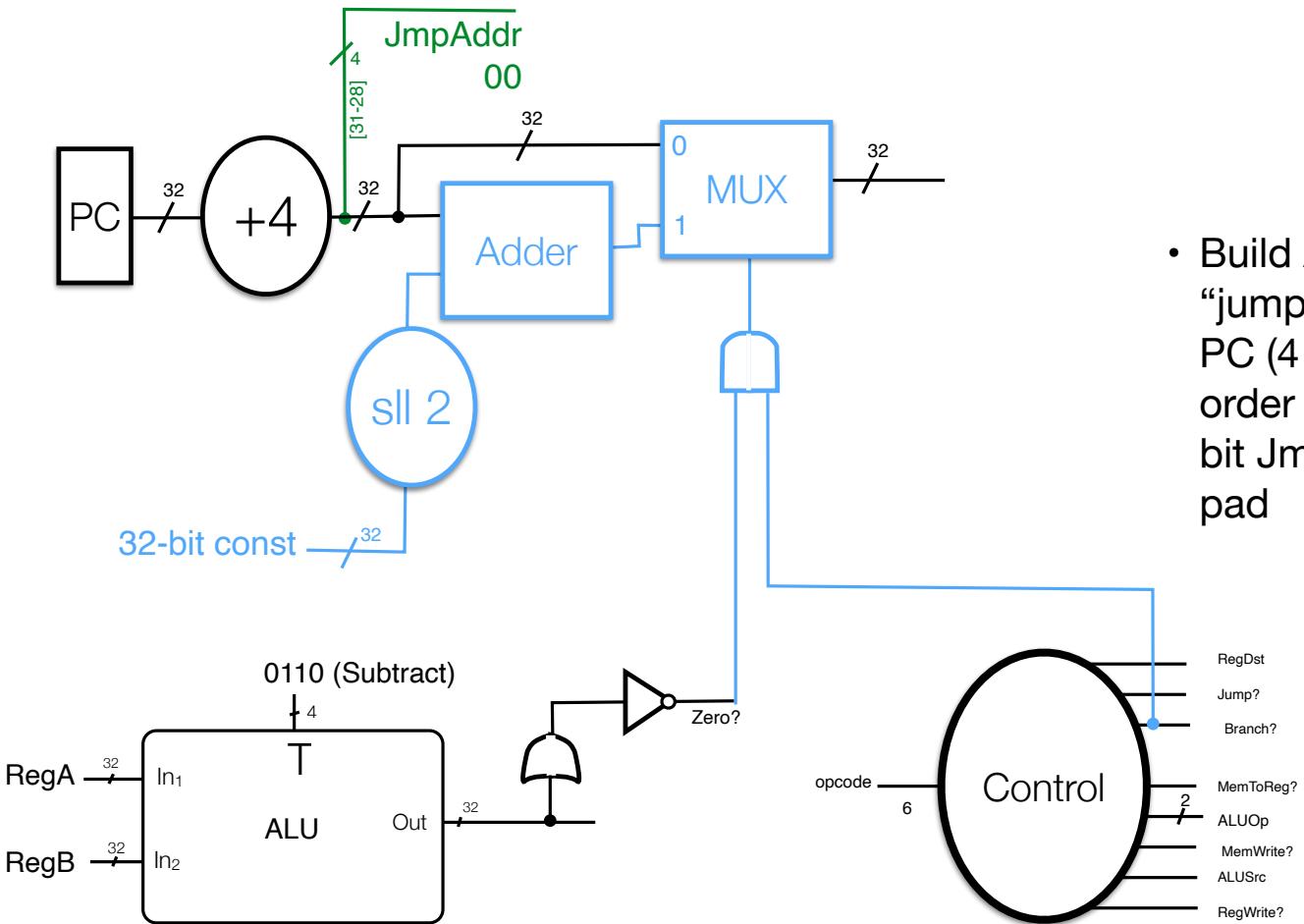


PC Adjustment Circuitry



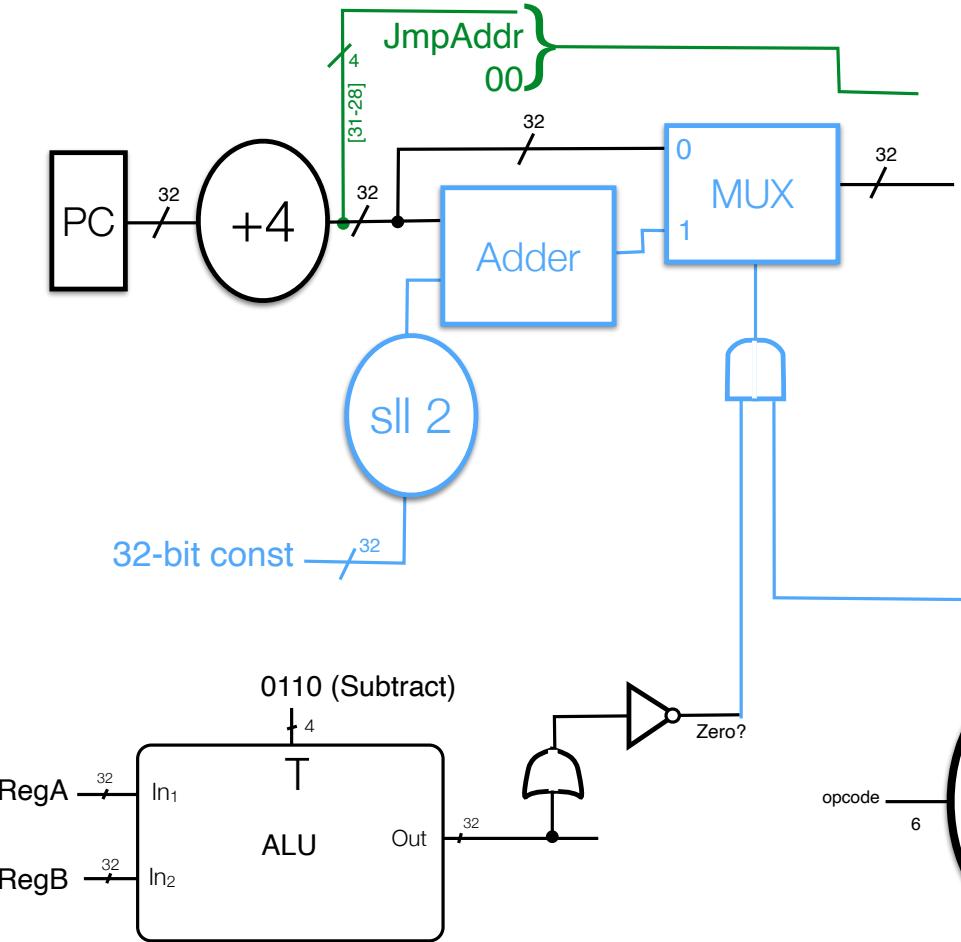
- Determine when Branch should be taken (branch instruction & conditional true)

PC Adjustment Circuitry

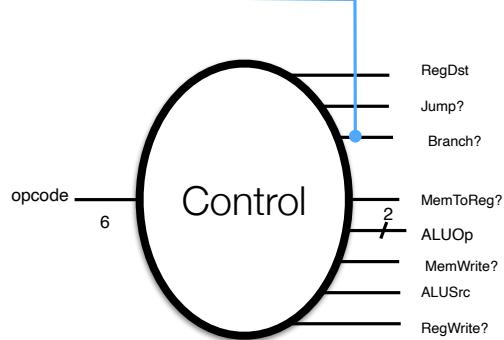


- Build Address to “jump” to from PC (4 highest order bits), 26-bit JmpAddr, 00 pad

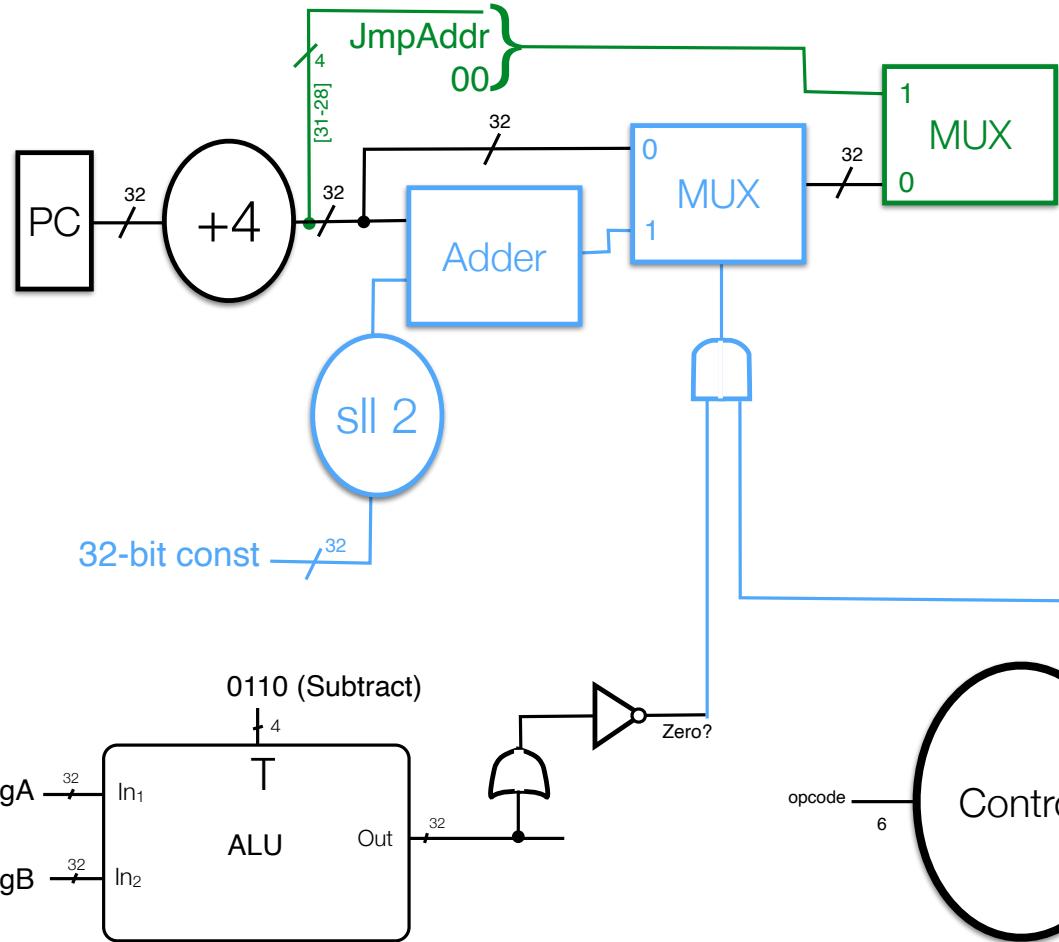
PC Adjustment Circuitry



- Build Address to “jump” to from PC (4 highest order bits), 26-bit JmpAddr, 00 pad

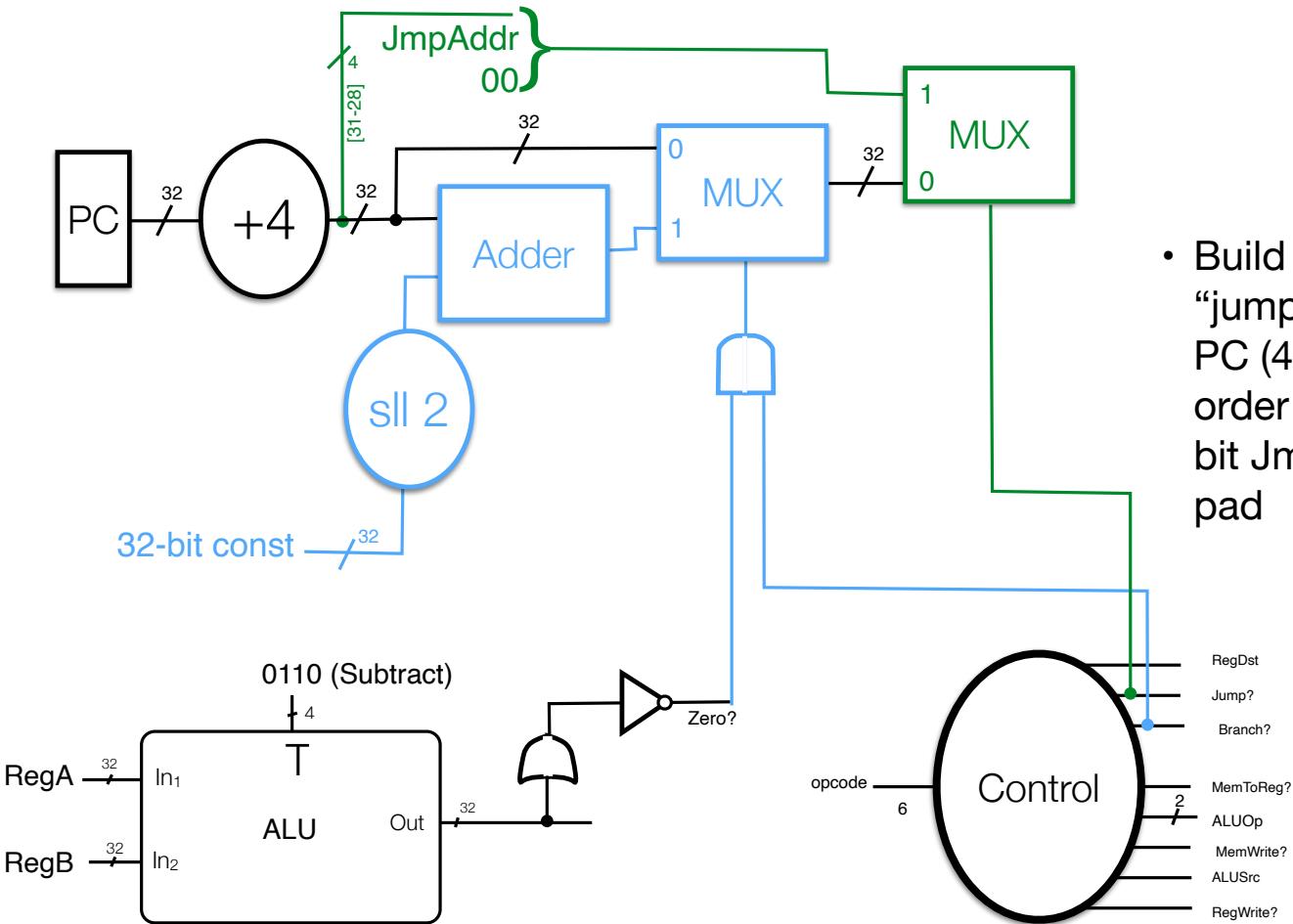


PC Adjustment Circuitry



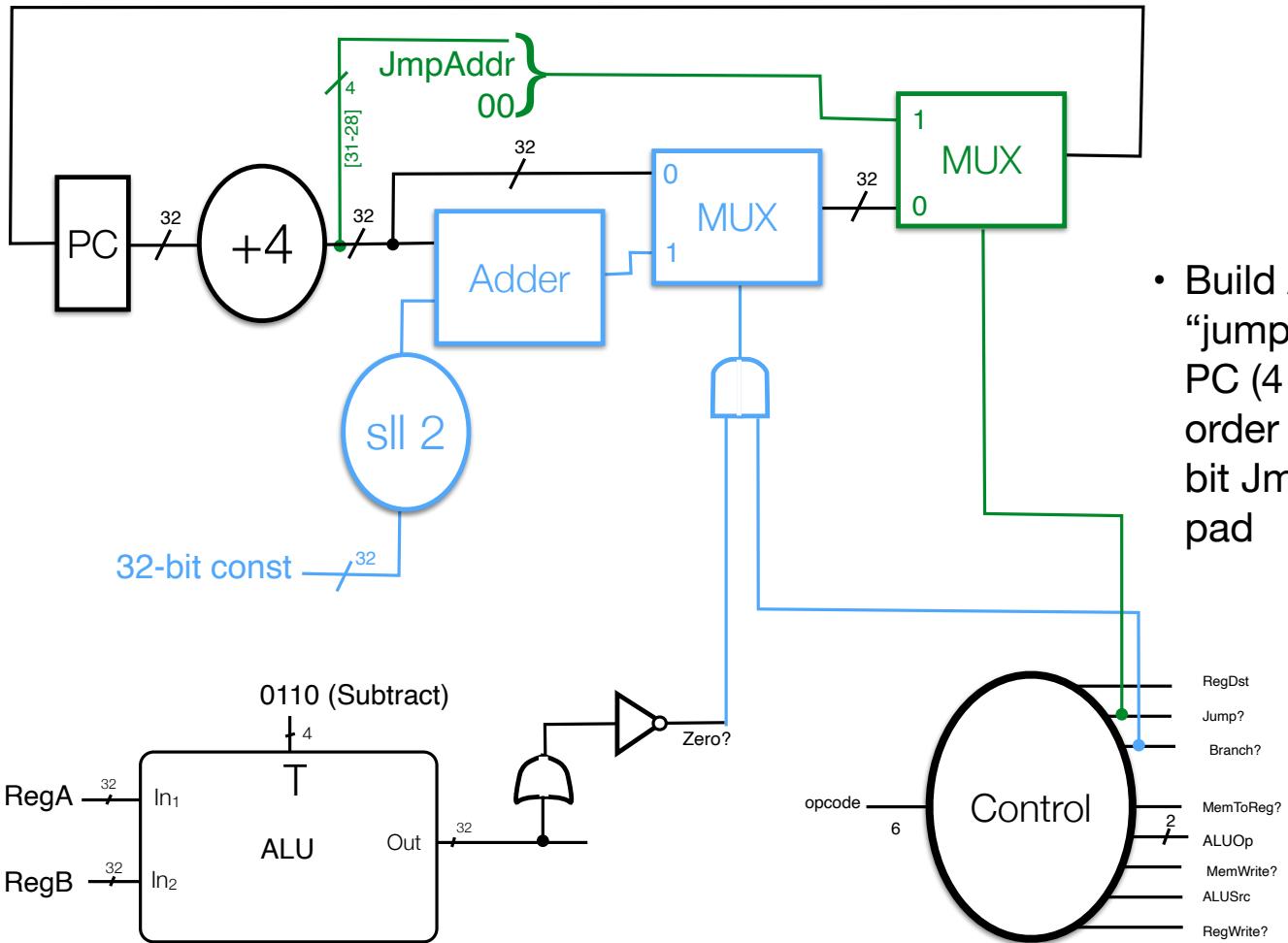
- Build Address to “jump” to from PC (4 highest order bits), 26-bit JmpAddr, 00 pad

PC Adjustment Circuitry



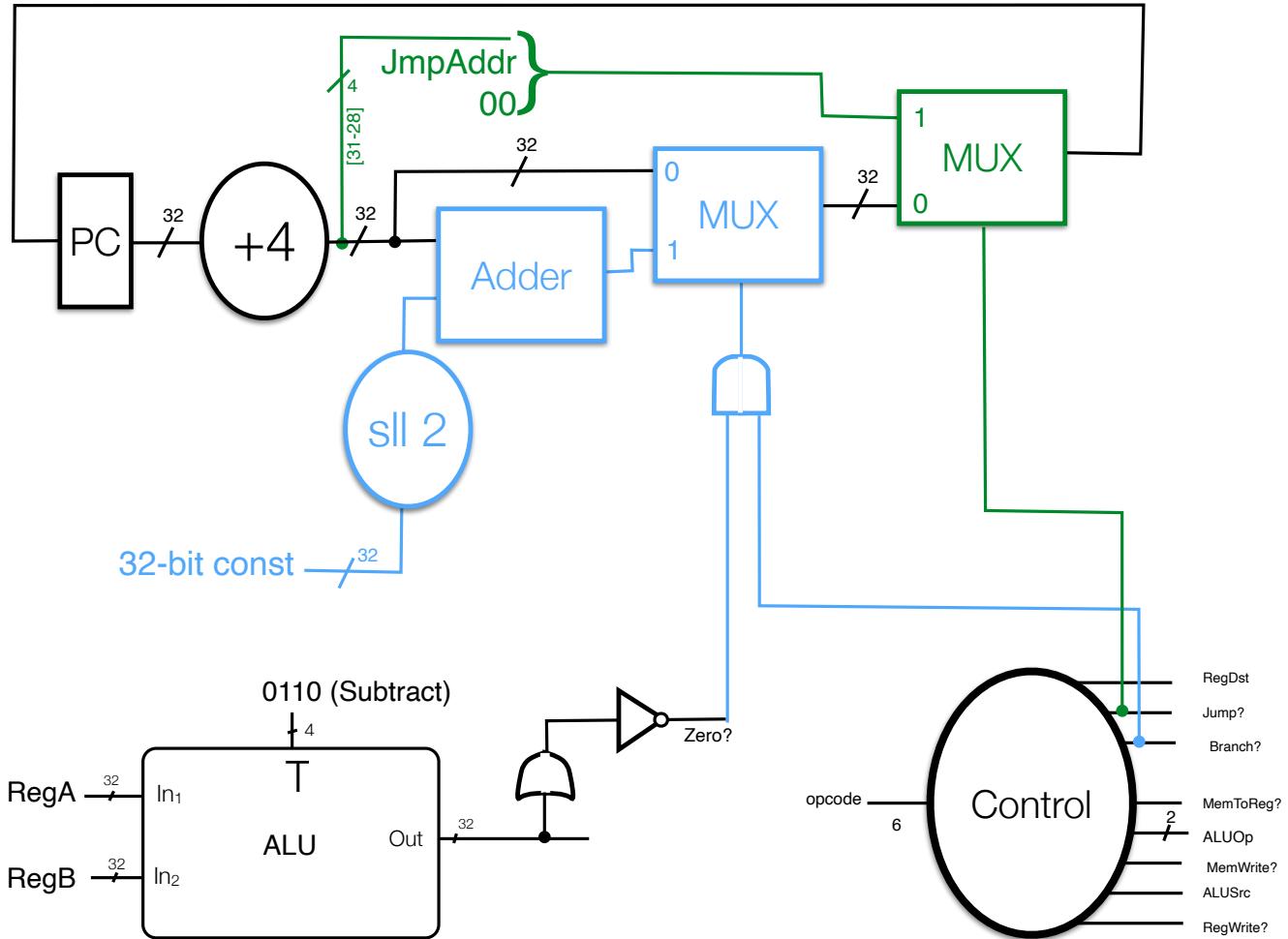
- Build Address to “jump” to from PC (4 highest order bits), 26-bit JmpAddr, 00 pad

PC Adjustment Circuitry

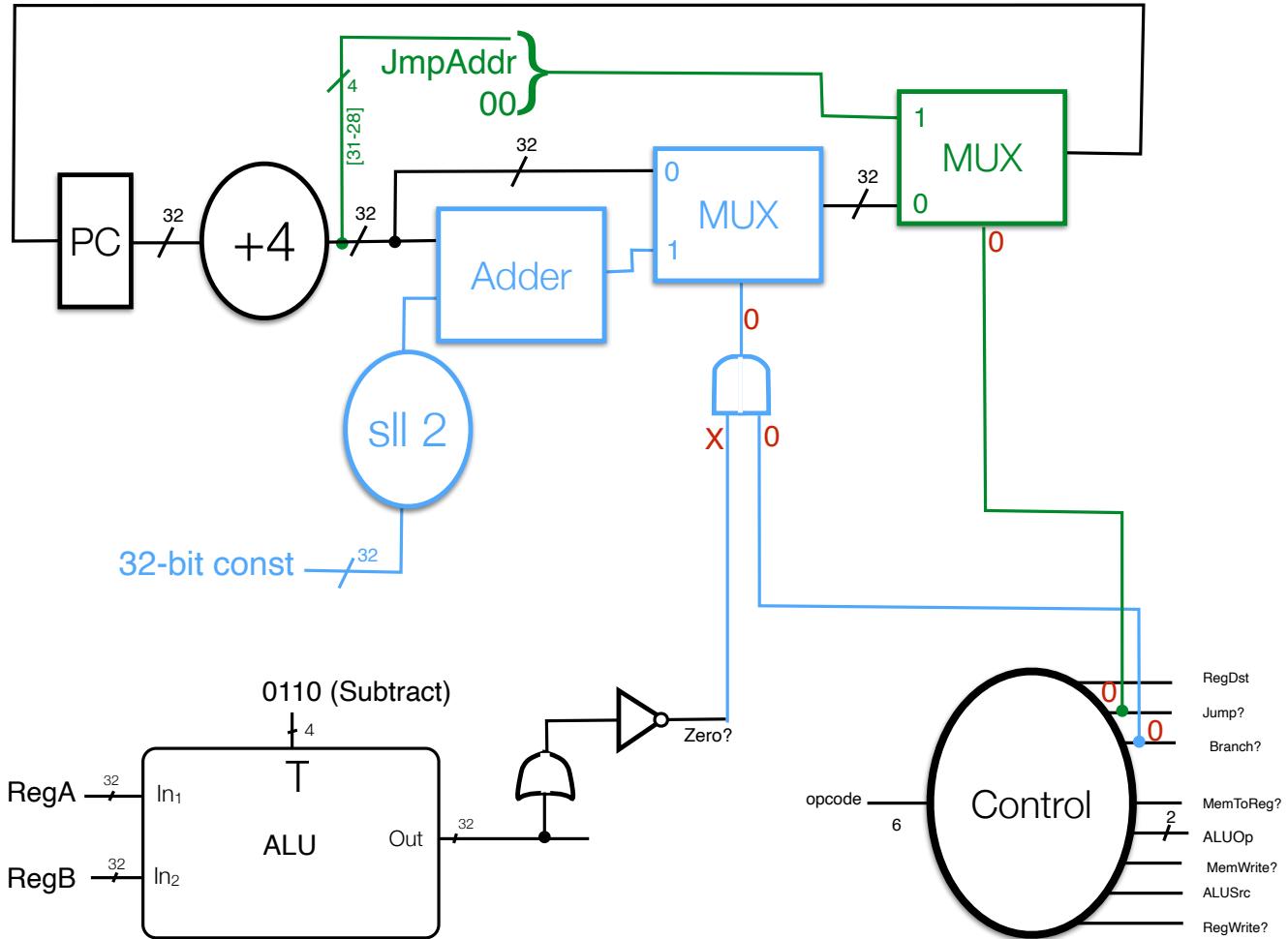


Examples

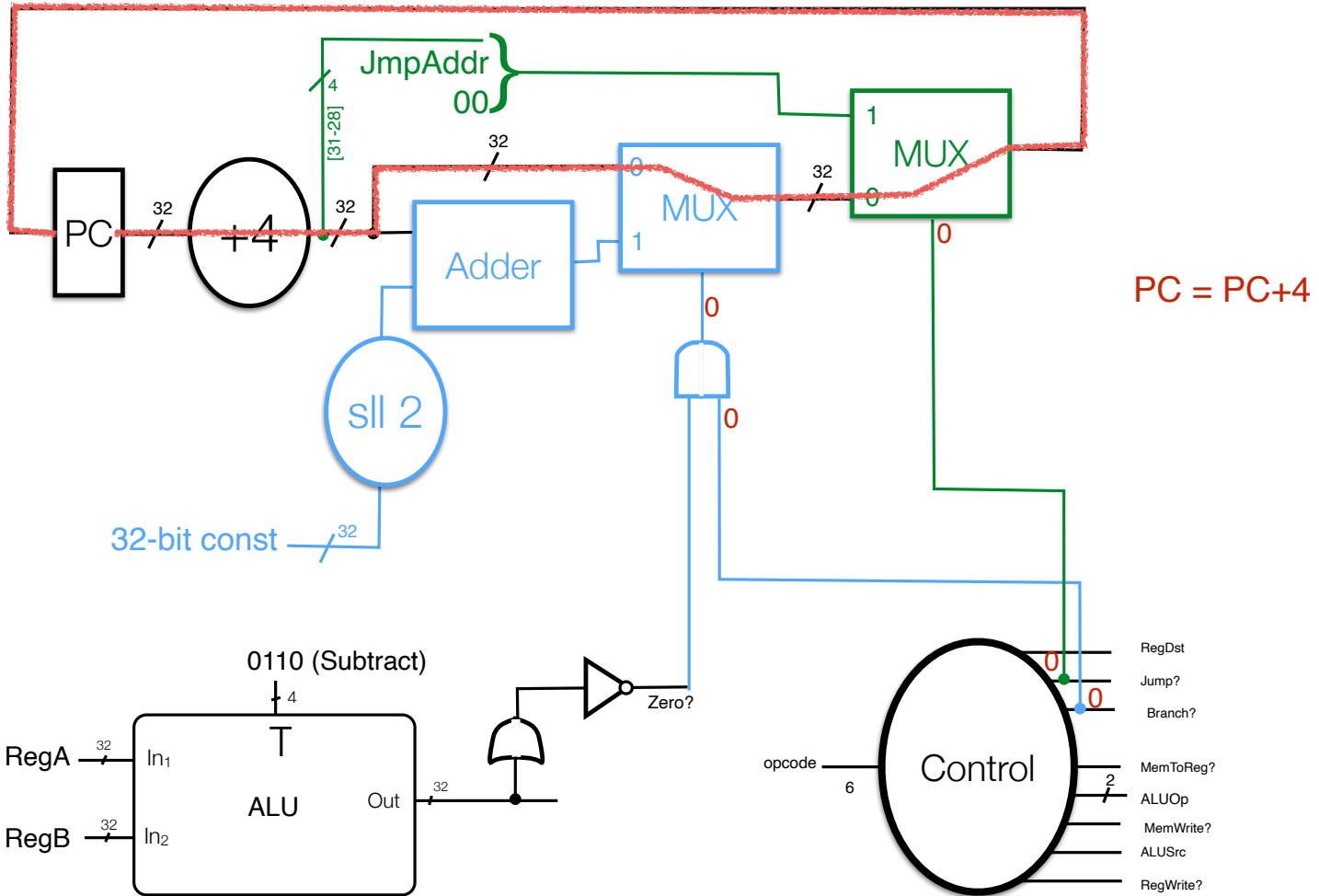
Not Branch or Jump



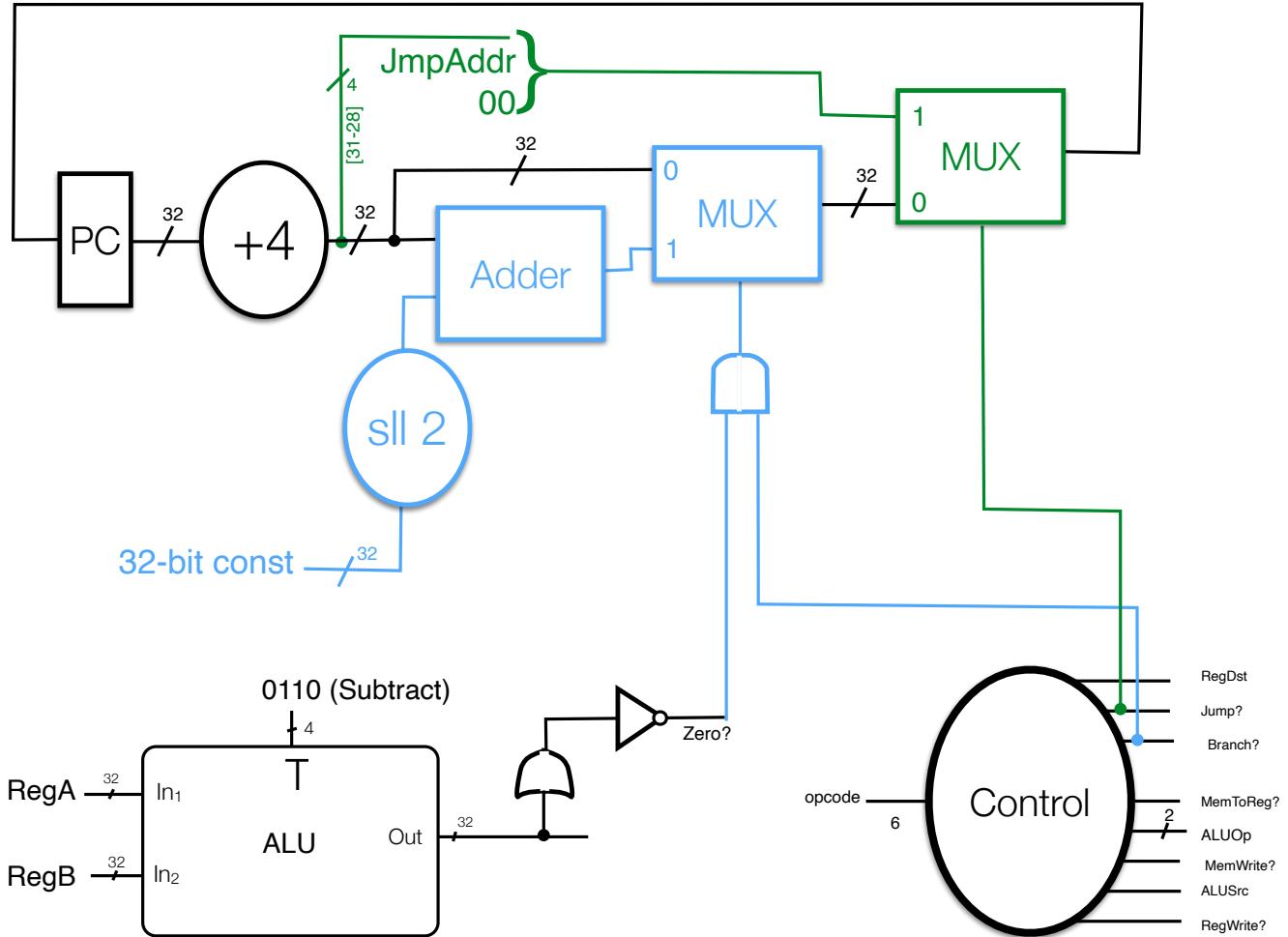
Not Branch or Jump



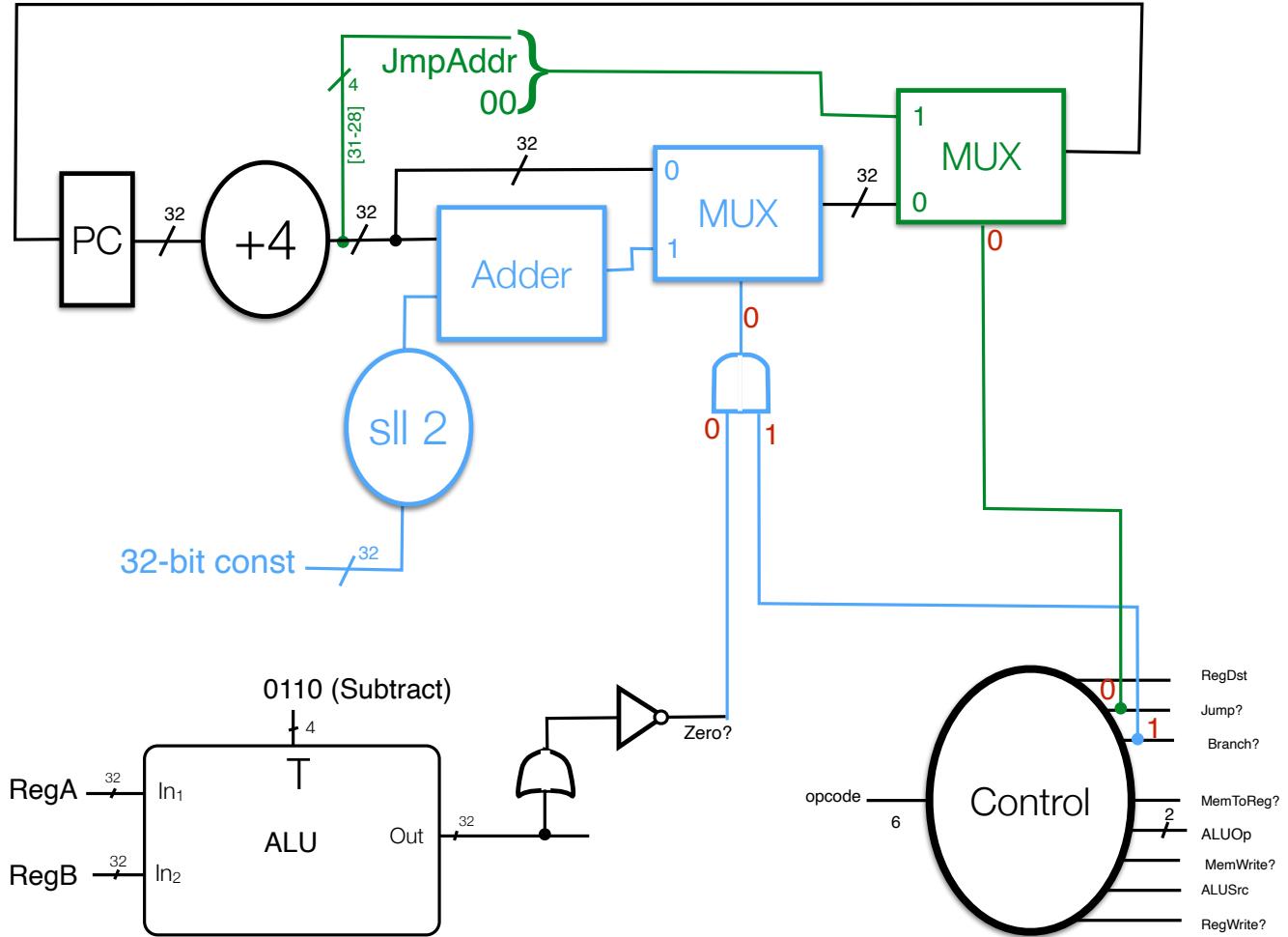
Not Branch or Jump



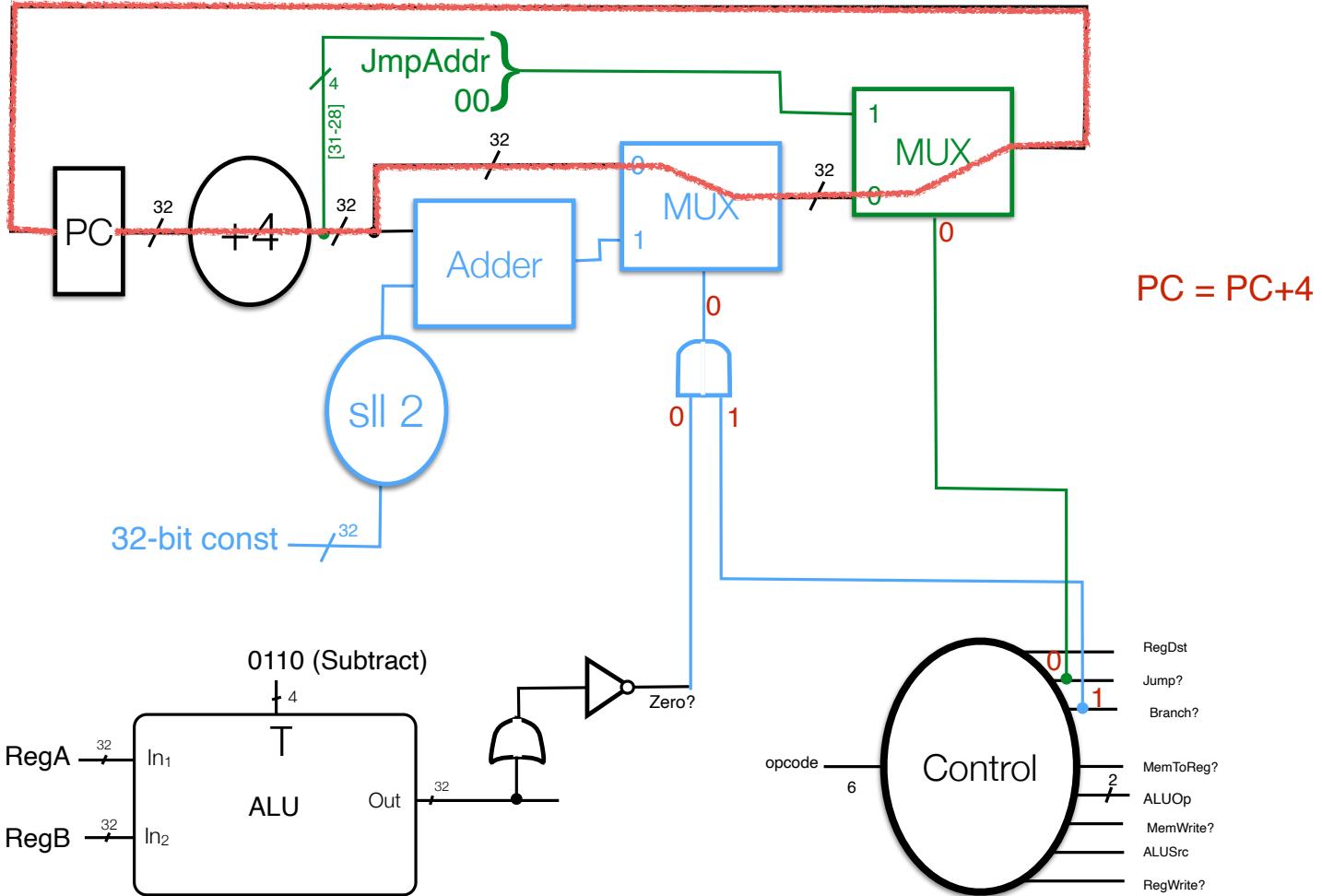
Branch but Conditional False



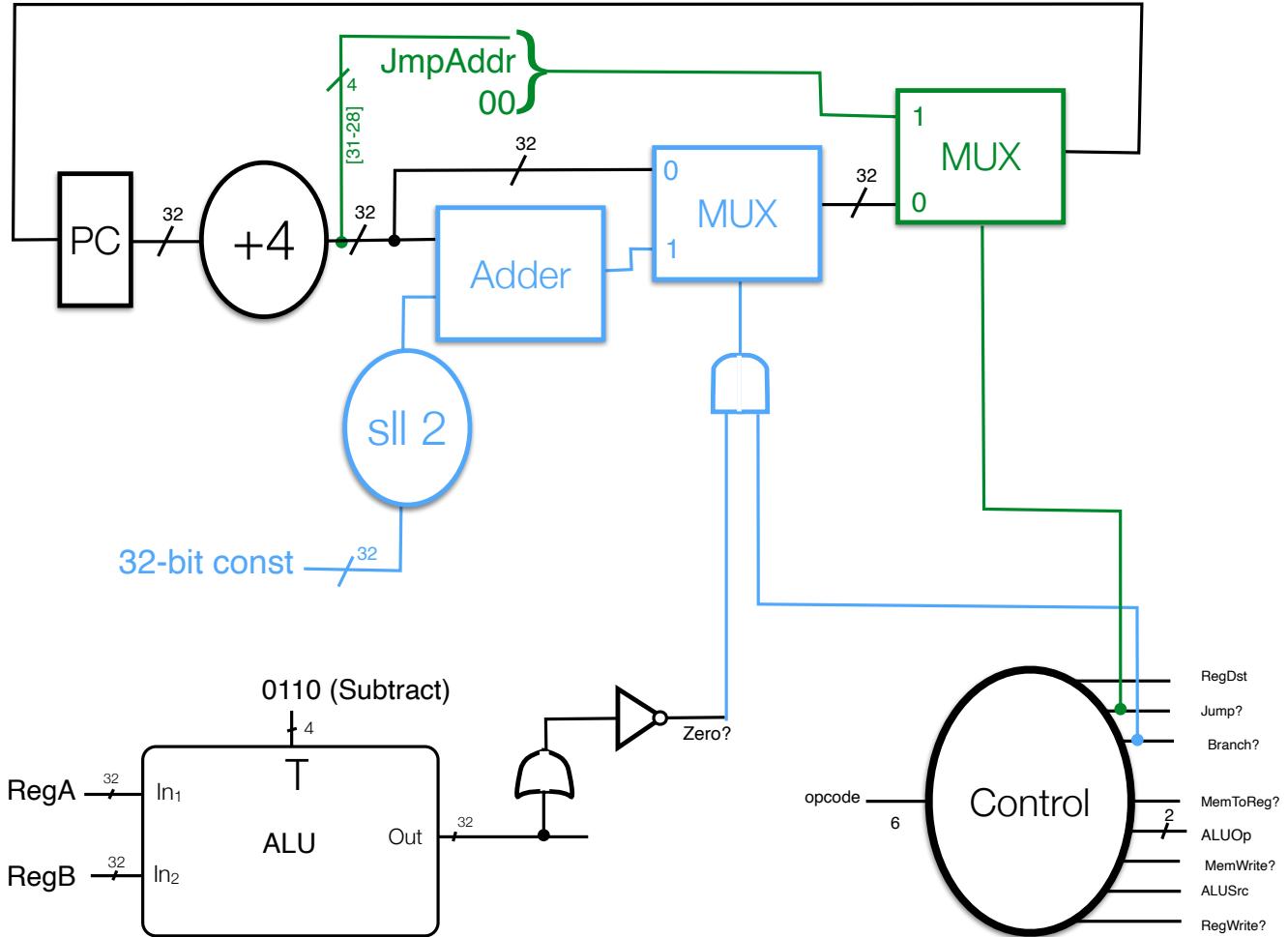
Branch but Conditional False



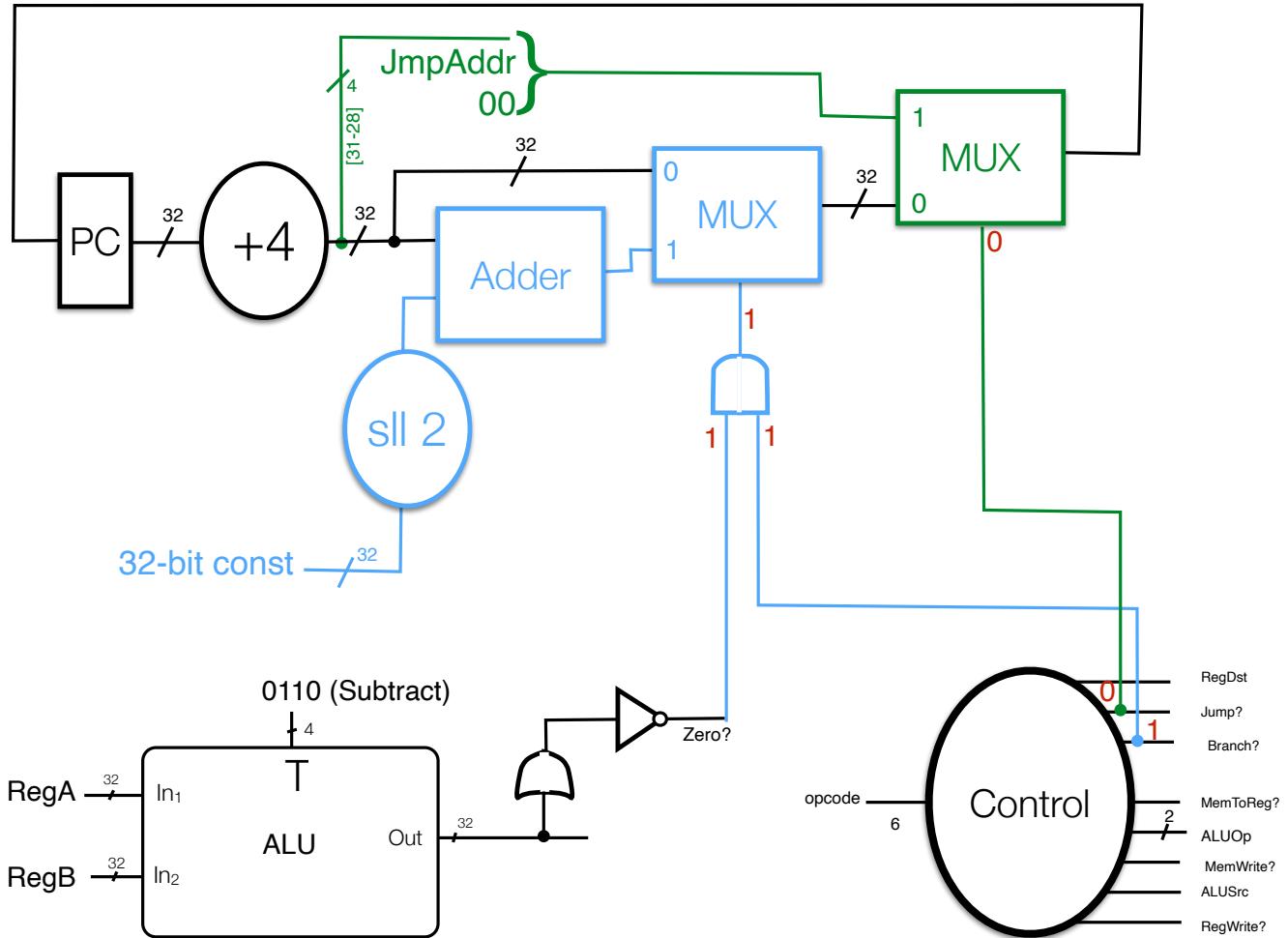
Branch but Conditional False



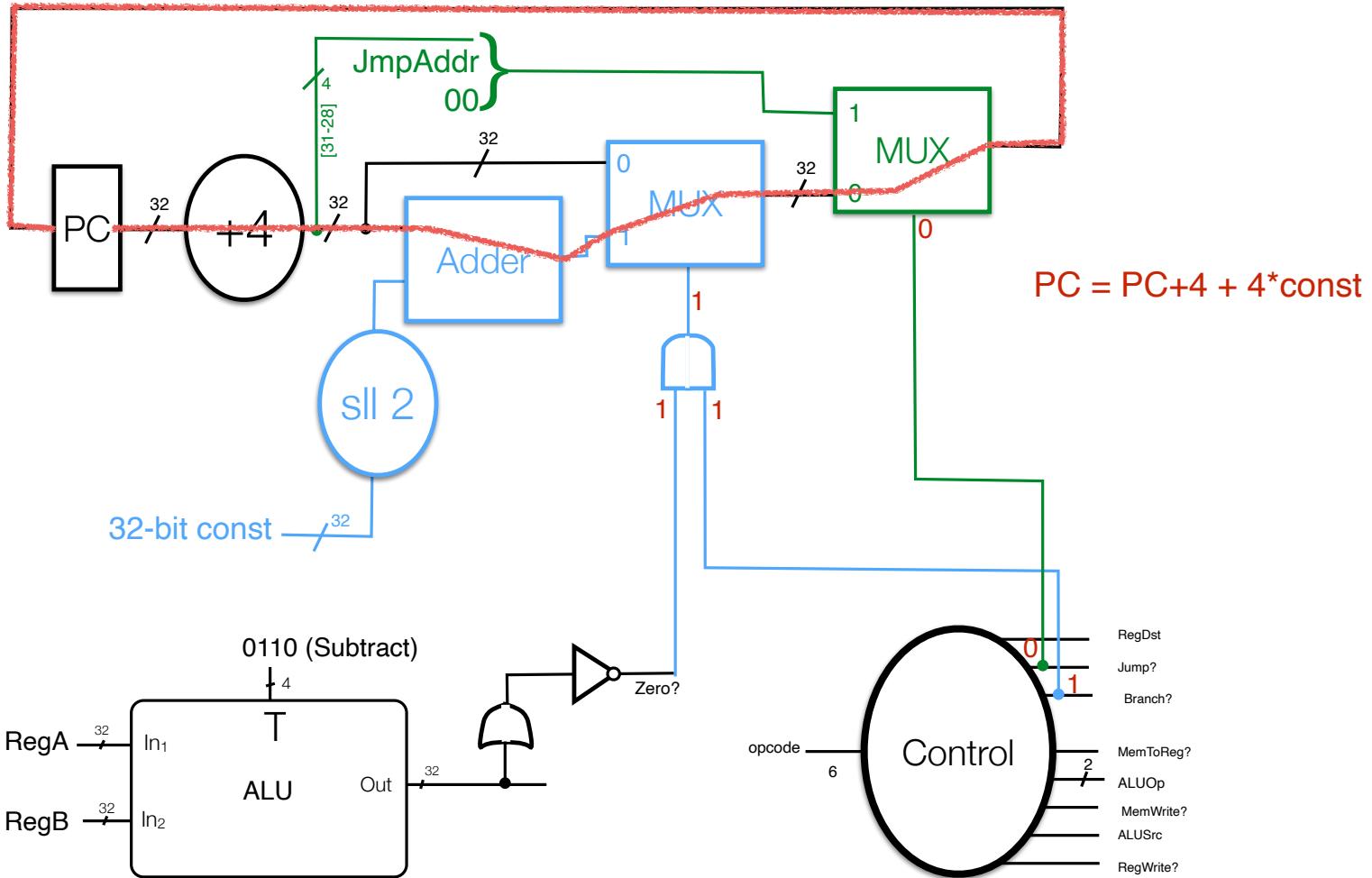
Branch and Conditional True



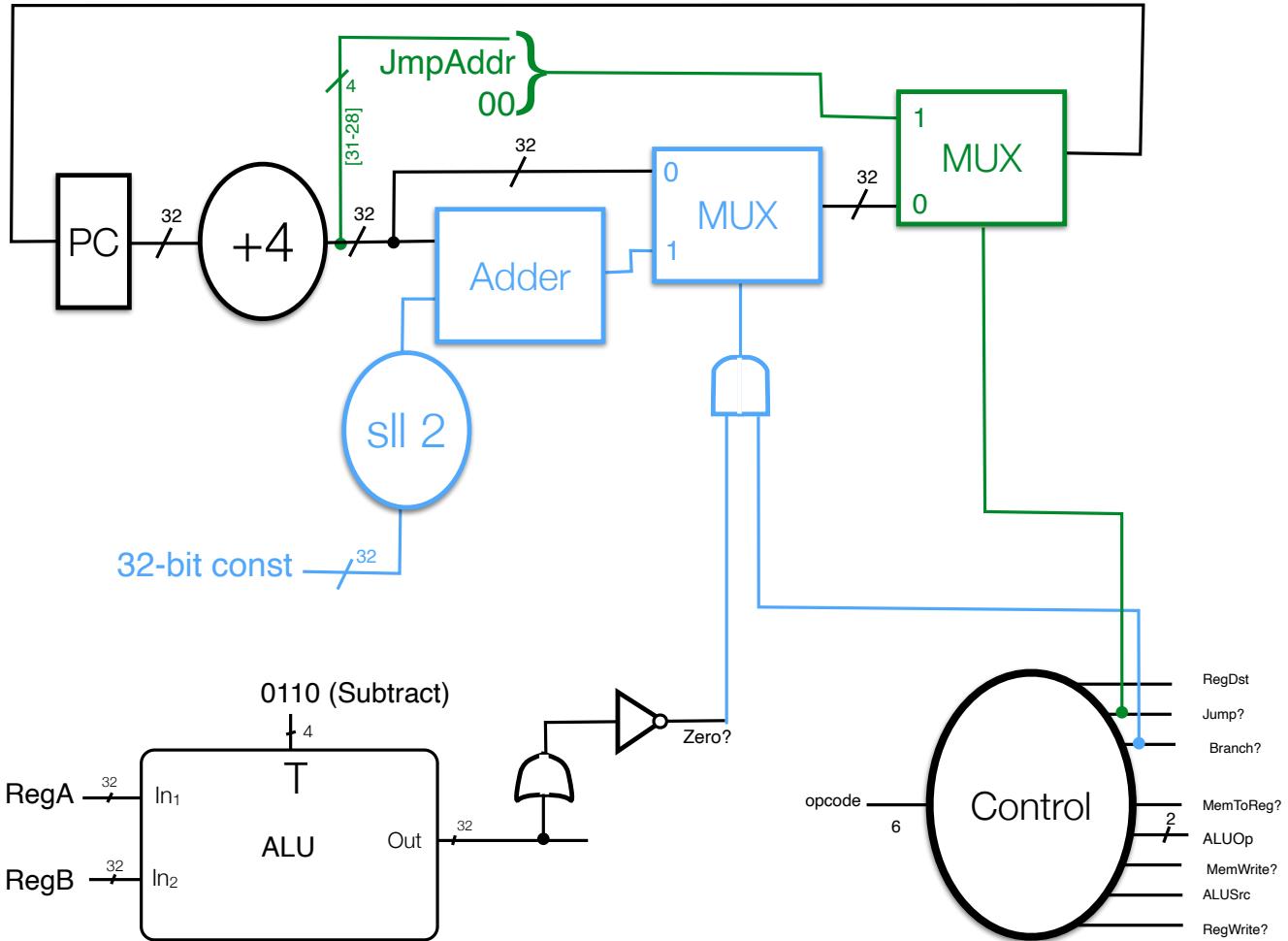
Branch and Conditional True



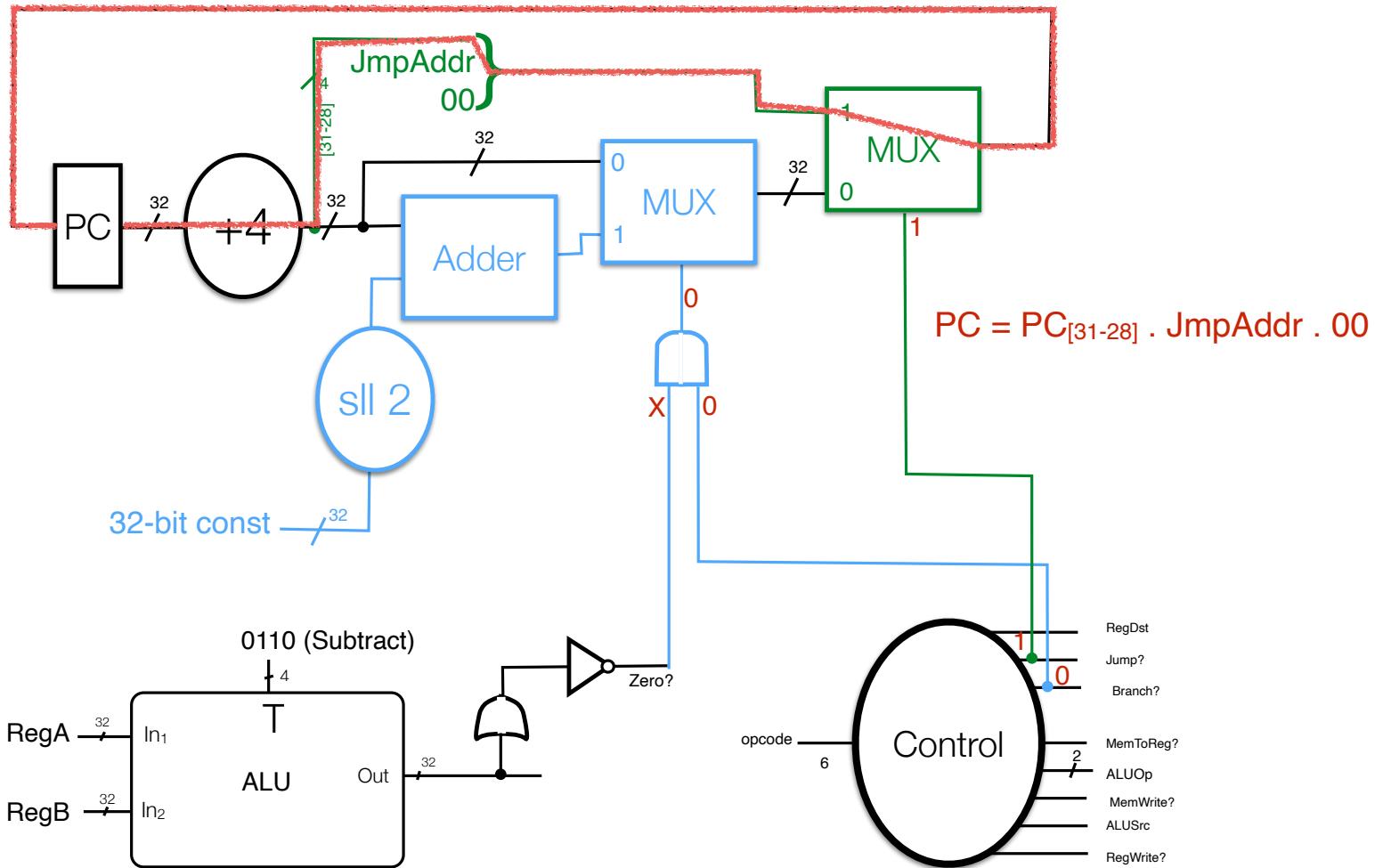
Branch and Conditional True



Jump



Jump



Implementing the jump instruction -- SOLN

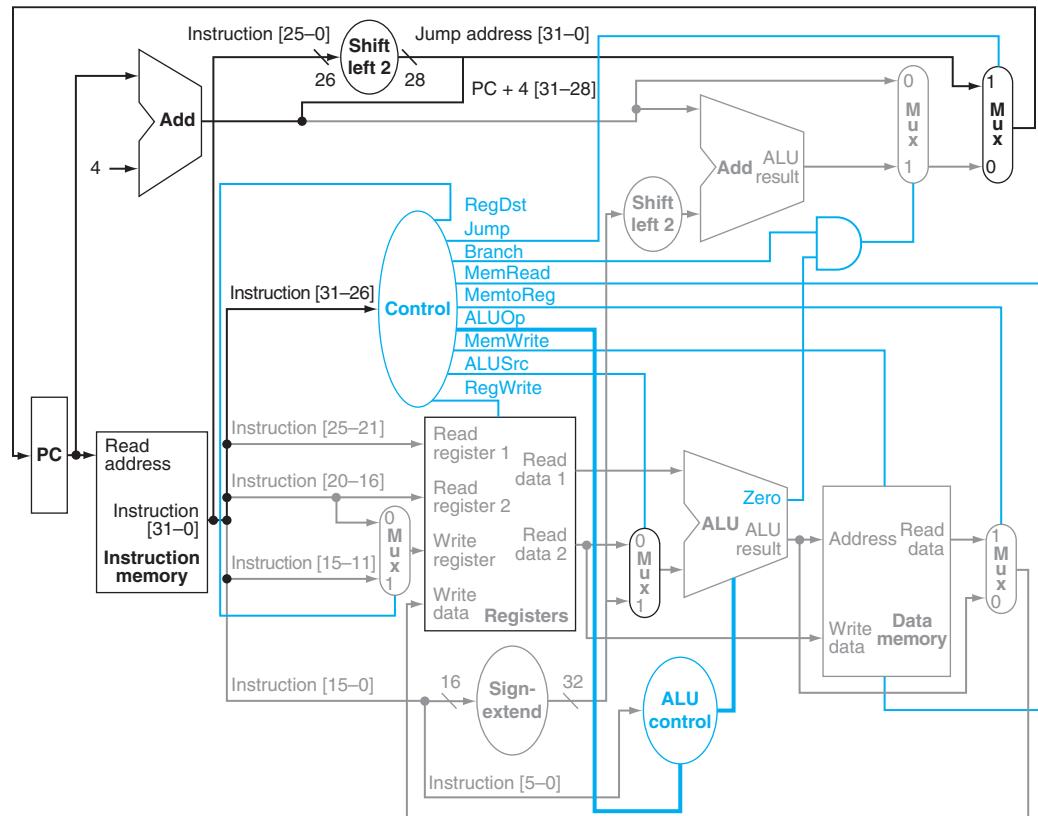


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address. Copyright © 2009 Elsevier, Inc. All rights reserved.



Final Thoughts

- Instruction performed in a single cycle
 - Inputs from instruction fed in, values read out from Reg File, ALU, Memory
 - When enough time has passed so all data feeds are stable, clock cycle will end, any writes that are supposed to happen do so (true in reality for reg file, a bit more complex for memory writes)
 - If nothing is written to memory or reg file (data state), then instruction has no affect on the (data) state: only control state changes (PC).
- That entire schematic can be broken down into ANDs, ORs and NOTs (or just NORs or just NANDs)!!!