CS510 Computer Architecture

Lecture 04: MIPS CPU Design II

Soontae Kim

Spring 2017

School of Computing, KAIST

Overview of MIPS Instruction Micro-operations

- **All instructions go through these common steps:**
  - Send program counter to instruction memory and fetch the instruction. (*fetch)* Instruction ← Mem[PC]
  - Update the program counter to point to next instruction PC ← PC + 4
  - Read one or two registers, using instruction fields. (*decode*)
- **Additional instruction execution actions (*execution*) depend on the instruction in question, but similarities exist:**
  - All instruction classes (except J type) use the ALU after reading the registers:
    - **Memory reference instructions use it for address calculation.**
    - **Arithmetic and logic instructions (R-Type), use it for the specified operation.**
    - **Branches use it for comparison.**
- **Additional execution steps where instruction classes differ:**
  - Memory reference instructions: Access memory for a load or store and write to a register for load.
  - Arithmetic and logic instructions: Write ALU result back in register.
  - Branch instructions: Change next instruction address based on comparison.

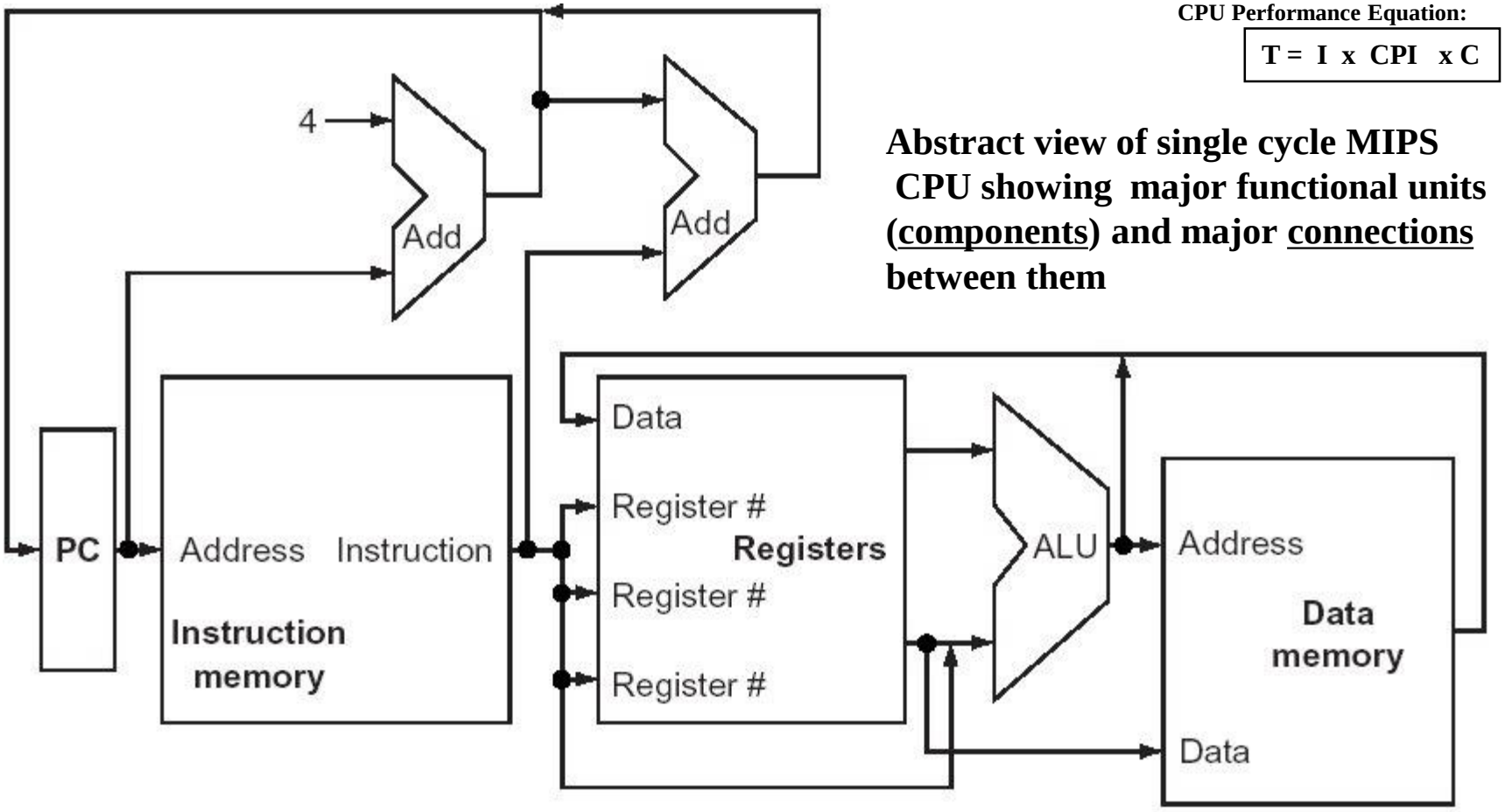
A Single Cycle MIPS CPU Design

Design target: **A single-cycle per instruction MIPS CPU design**

All micro-operations of an instruction are to be carried out in a single CPU clock cycle. Cycles Per Instruction = CPI = 1

CPU Performance Equation:

$$T = I \times CPI \times C$$

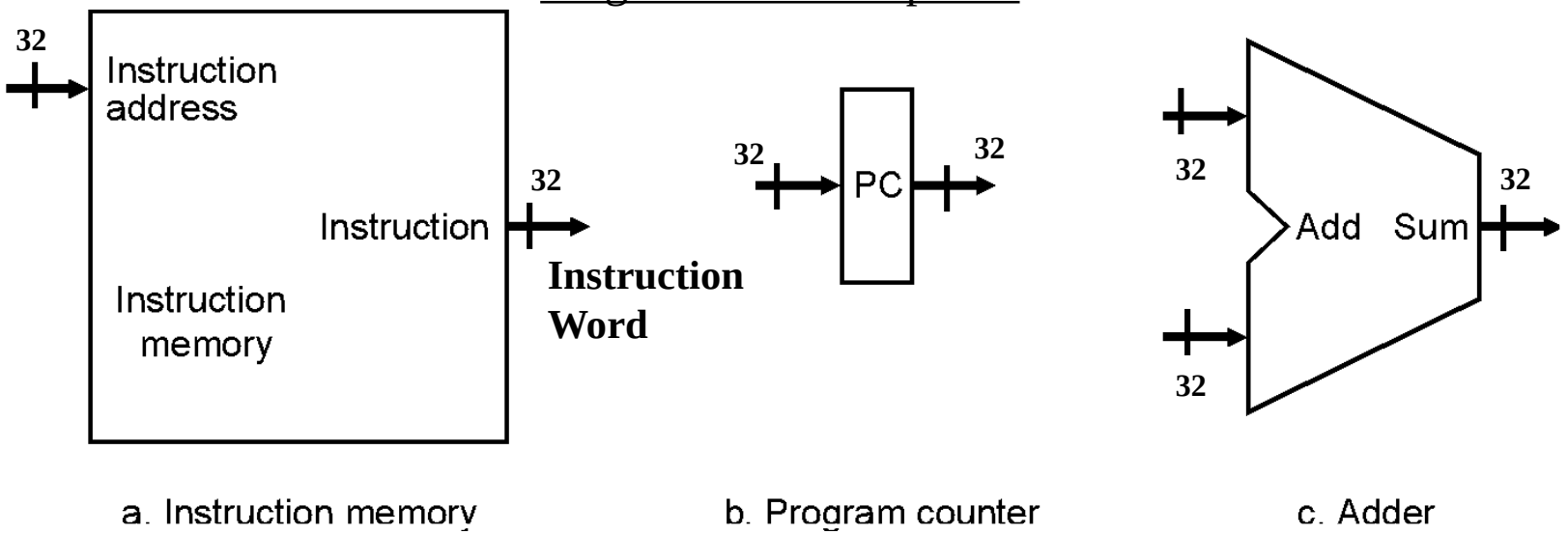


Abstract view of single cycle MIPS CPU showing major functional units (components) and major connections between them

Initial Datapath Components

Three components needed by: Instruction Fetch: **Instruction ← Mem[PC]**

Program Counter Update: **PC ← PC + 4**



Two state elements (memory) needed to store and access instructions:

1 **Instruction memory:**

- **Only read access (by user code). No read control signal needed.**

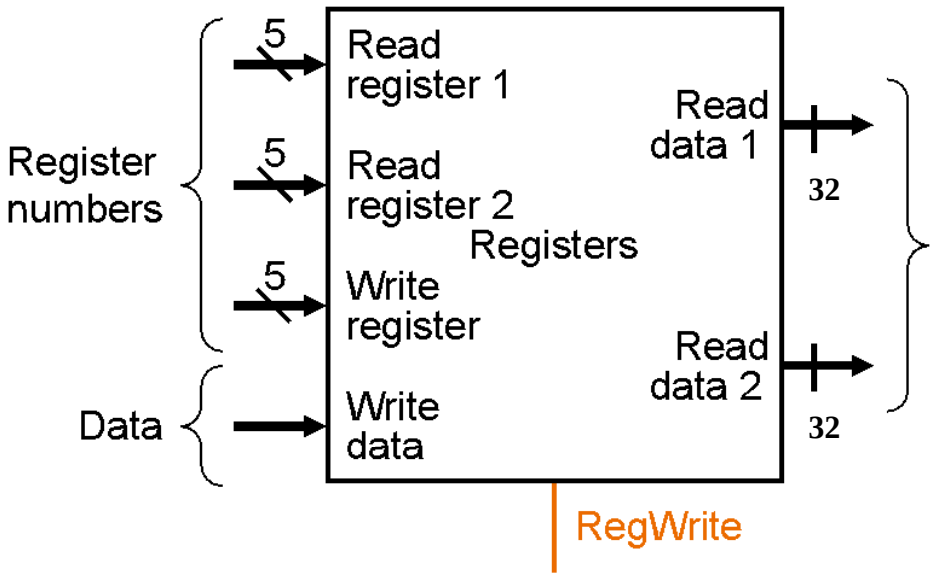
2 **Program counter (PC): 32-bit register.**

- **Written at end of every clock cycle (edge-triggered) : No write control signal.**

3 **32-bit Adder: To compute the the next instruction address (PC + 4).**

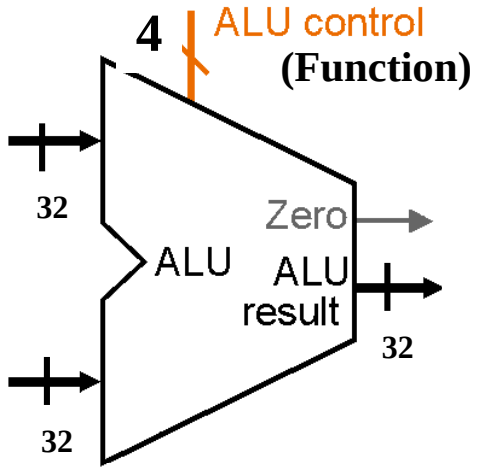
More Datapath Components

ISA Register File



a. Registers

Main 32-bit ALU



32-bit Arithmetic and Logic Unit (ALU)

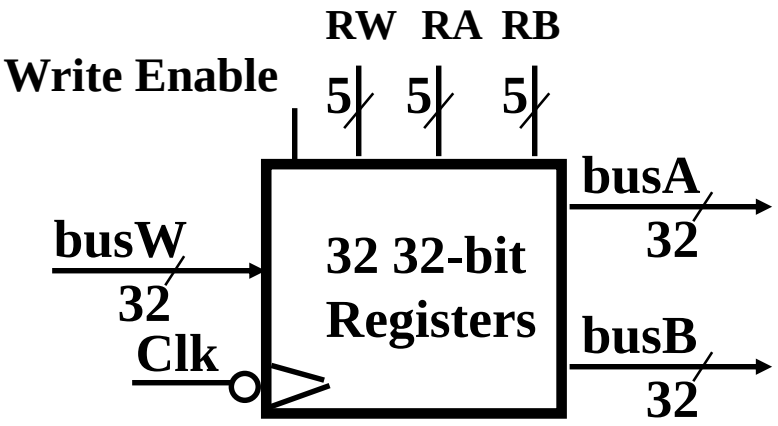
b. ALU

Zero = Zero flag = 1
When ALU result equals zero

Register File:

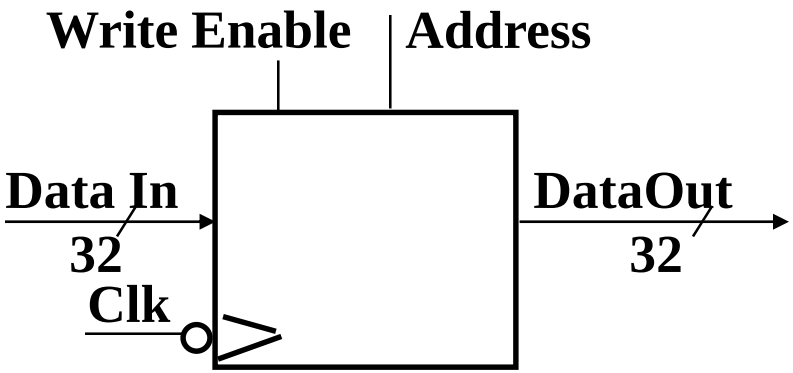
- **Contains all ISA registers.**
- **Two read ports and one write port.**
- **Register writes by asserting write control signal**
- **<u>Clocking Methodology:</u> <u>Writes are edge-triggered</u>.**
  - **Thus can read and write to the same register in the same clock cycle.**

Register File Details



- **Register File consists of 32 registers:**
  - Two 32-bit output busses:
    busA and busB
  - One 32-bit input bus: busW
- **Register is selected by:**
  - RA (number) selects the register to put on busA (data):
    busA = R[RA]
  - RB (number) selects the register to put on busB (data):
    busB = R[RB]
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1
    Write Enable: R[RW] ← busW
- **Clock input (CLK)**
  - The CLK input is a factor ONLY during write operations.
  - During read operation, it behaves as a combinational logic block:
    - **RA or RB valid => busA or busB valid after "access time."**

Idealized Memory

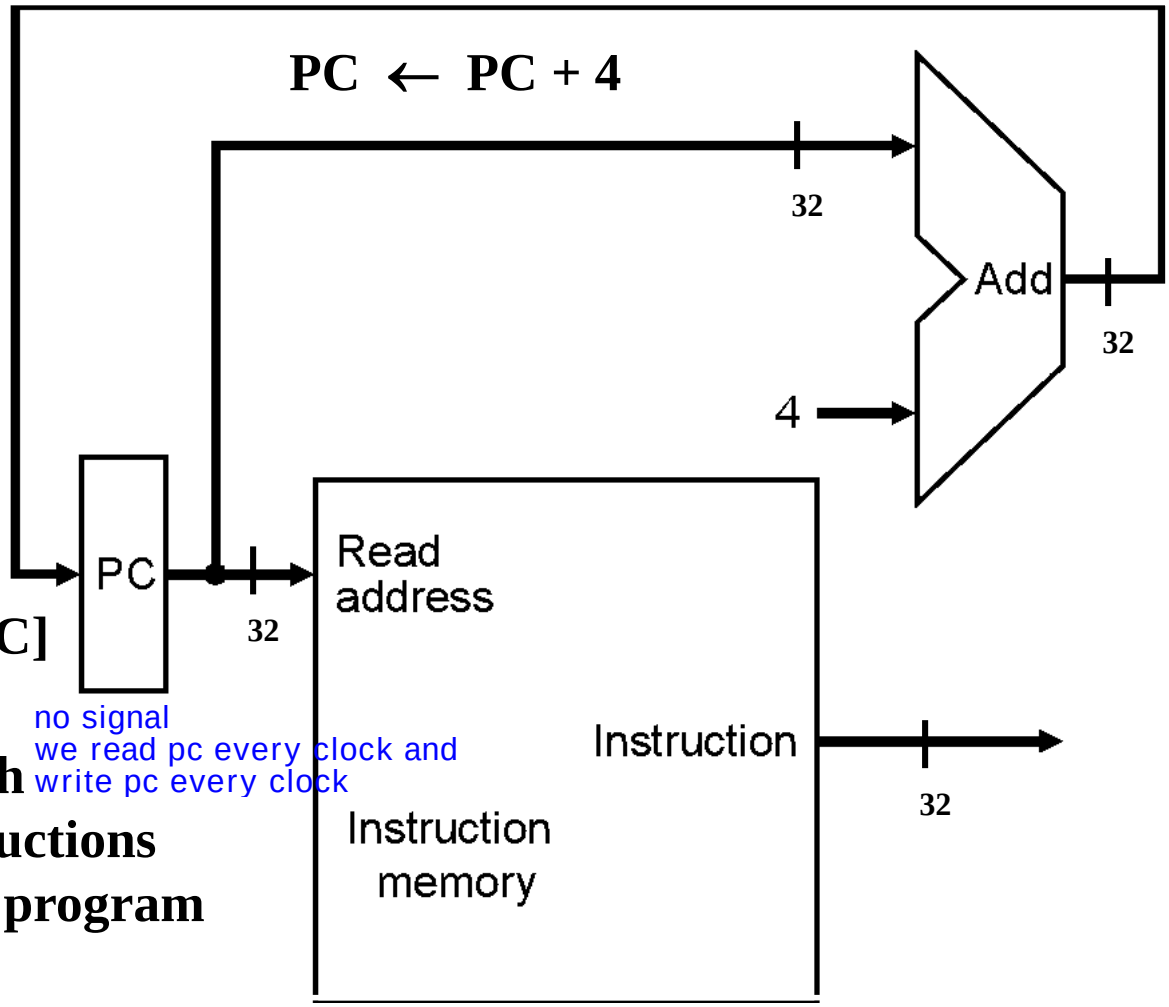


- **Memory (idealized)**
  - One input bus: Data In.
  - One output bus: Data Out.
- **Memory word is selected by:**
  - Address selects the word to put on Data Out bus.
  - Write Enable = 1: address selects the memory word to be written via the Data In bus.
- **Clock input (CLK):**
  - The CLK input is a factor ONLY during write operation,
  - During read operation, this memory behaves as a combinational logic block:
    - **Address valid => Data Out valid after "access time."**
- **Ideal Memory = Short access time.**

Compared to other components

Building The Datapath

Instruction Fetch & PC Update:



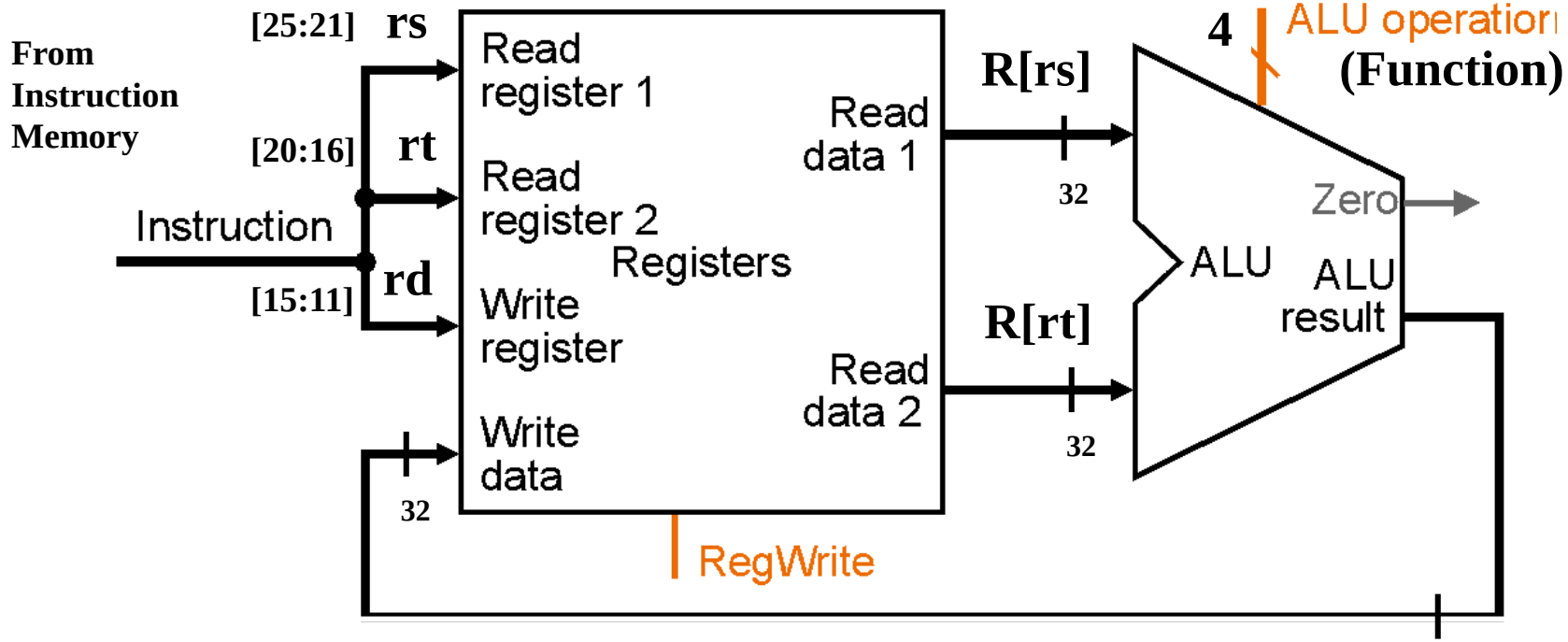
Instruction ← Mem[PC]

Portion of the datapath used for fetching instructions and incrementing the program counter.

PC write or update is edge triggered at the end of the cycle

Clock input to PC, memory not shown

Simplified Datapath For MIPS R-Type Instructions



Components and connections as specified by RTN statement

$$R[rd] \leftarrow R[rs] + R[rt]$$

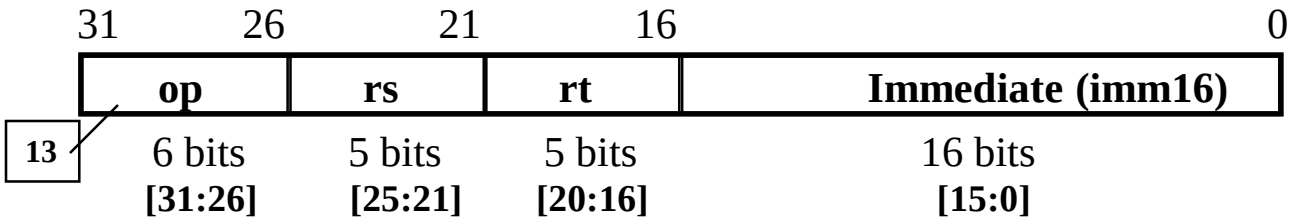
Destination register R[rd] write or update is edge triggered at the end of the cycle

i.e Funct = function =add

Logical Operations with Immediate Example:

Micro-Operation Sequence For ORI

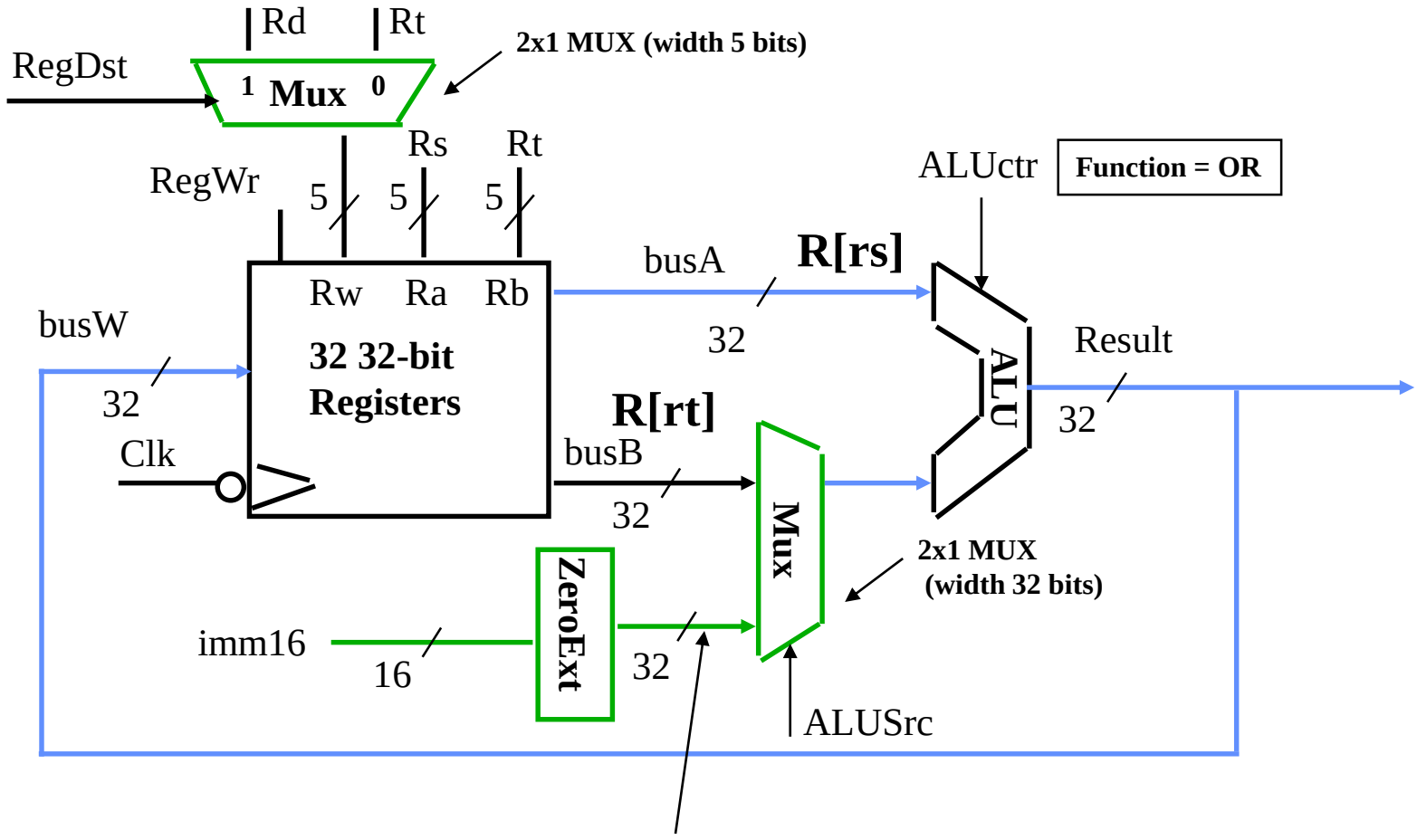
ori rt, rs, imm16



Instruction Word ← Mem[PC]	**Fetch the instruction**
PC ← PC + 4	**Increment PC**
R[rt] ← R[rs] OR ZeroExt[imm16]	**OR register rs with immediate field zero extended to 32 bits, result in register rt**

Done by Main ALU

Datapath For Logical Instructions With Immediate

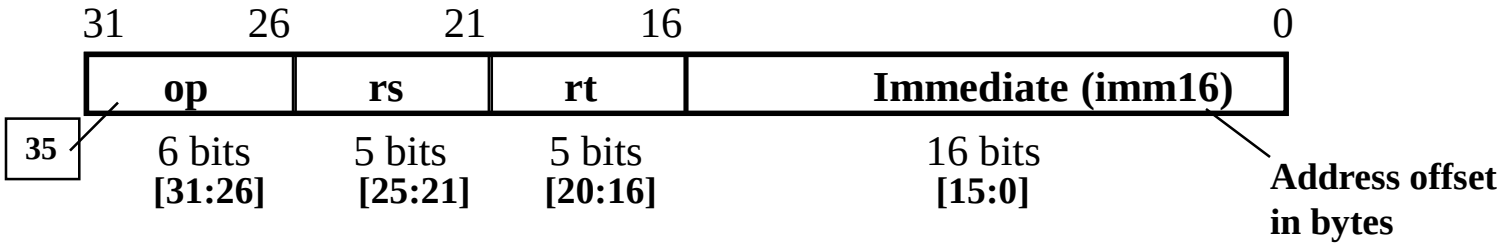


R[rt] ← R[rs] OR ZeroExt[imm16]

Load Operations Example:

Micro-Operation Sequence For LW

lw rt, rs, imm16



Instruction Word ← Mem[PC] — **Fetch the instruction**

PC ← PC + 4 — **Increment PC**

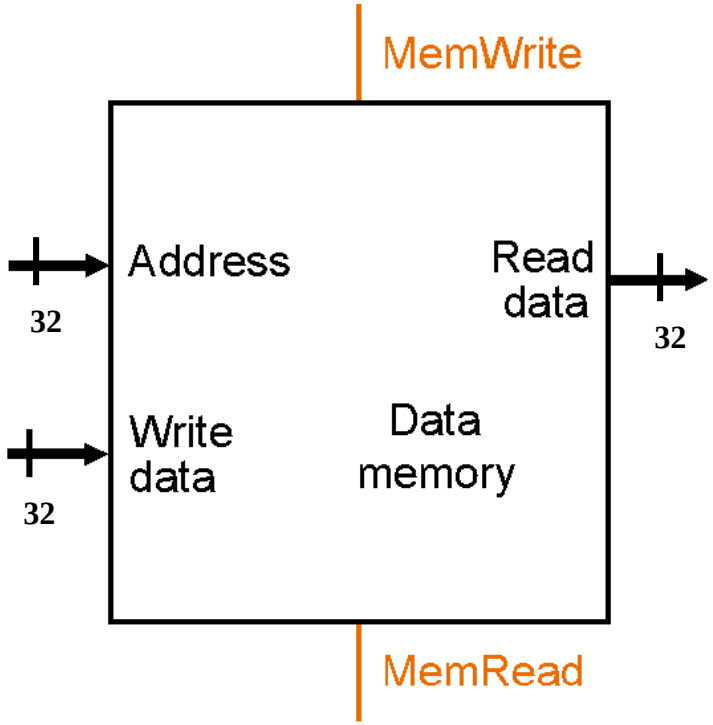
Instruction Memory

R[rt] ← Mem[R[rs] + SignExt[imm16]] — **Immediate field sign extended to 32 bits and added to register rs to form memory load address, write word at effective address to register rt**

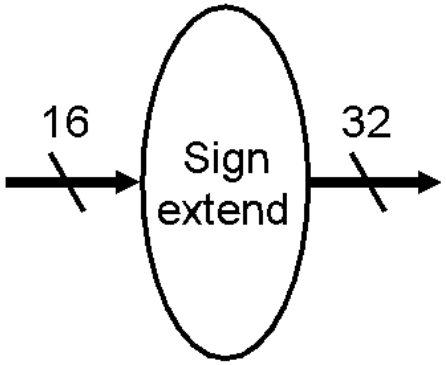
Effective Address

Data Memory

Additional Datapath Components For Loads & Stores



a. Data memory unit



For SignExt[imm16]

b. Sign-extension unit

Inputs:

for address and write (store) data

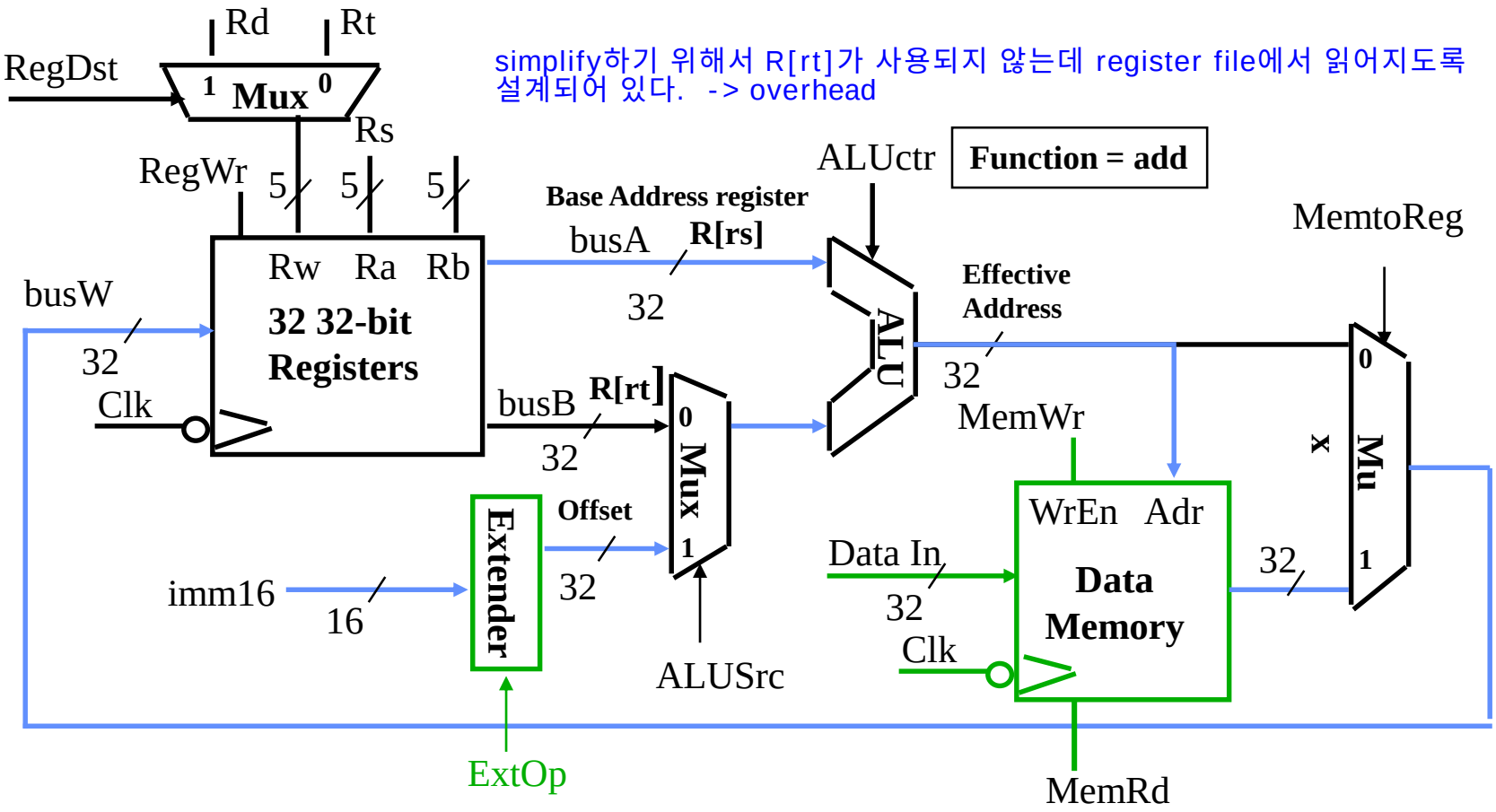
Output

for read (load) data

16-bit input sign-extended into a 32-bit value at the output

Data memory write or update is edge triggered at the end of the cycle (clocking methodology)

Datapath For Loads



R[rt] ← Mem[R[rs] + SignExt[imm16]]

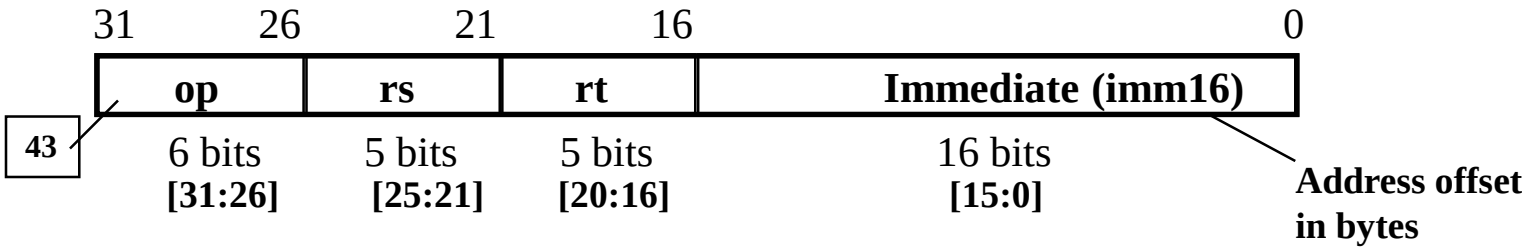
Data Memory

Effective Address

Store Operations Example:

Micro-Operation Sequence For SW

sw rt, rs, imm16

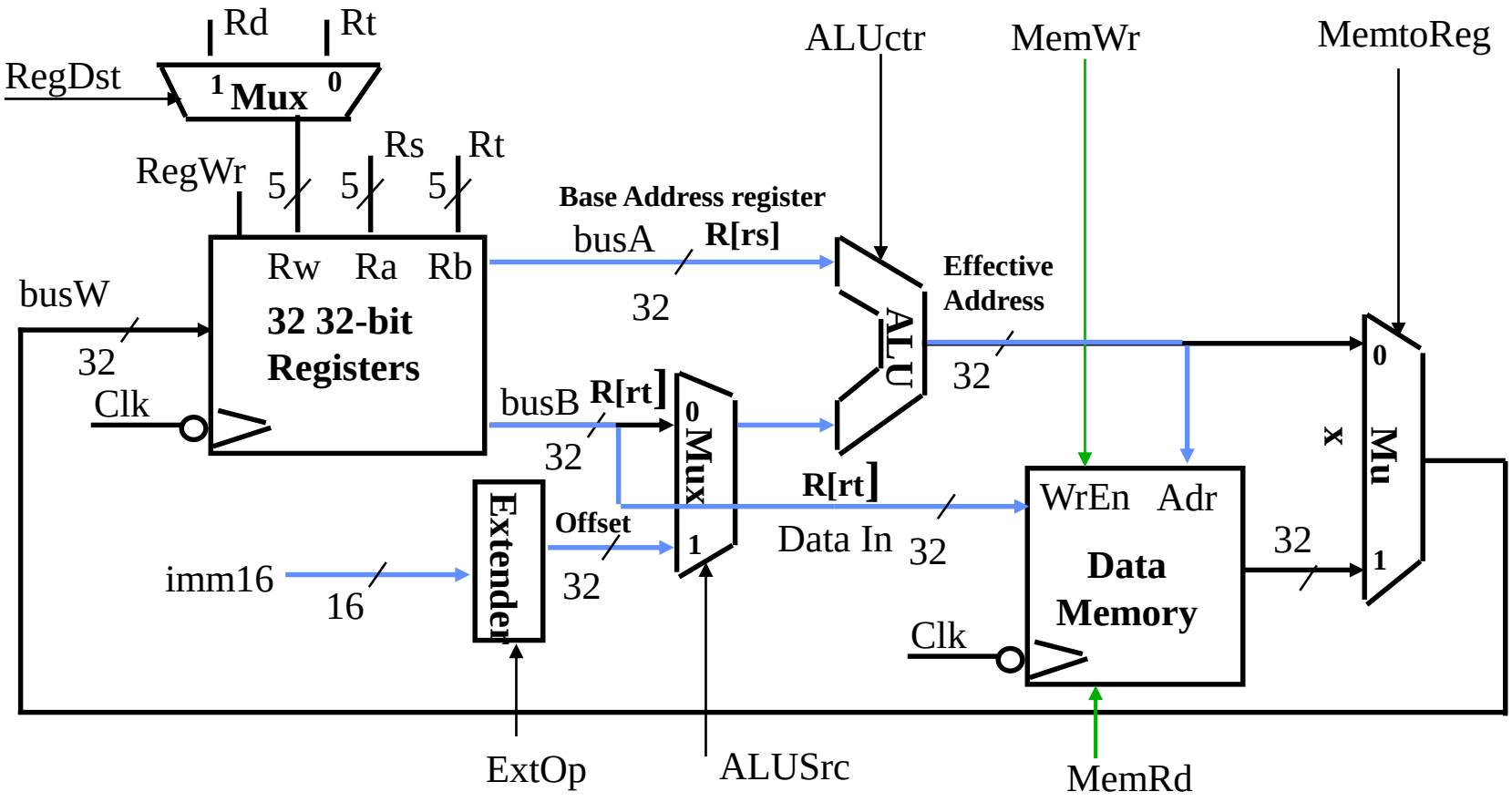


Instruction Word ← Mem[PC]	**Fetch the instruction**
PC ← PC + 4	**Increment PC**
Mem[R[rs] + SignExt[imm16]] ← R[rt]	**Immediate field sign extended to 32 bits and added to register rs to form memory effective address, register rt written to memory at store effective address.**

Effective Address

Data Memory

Datapath For Stores



Mem[R[rs] + SignExt[imm16]] ← R[rt]

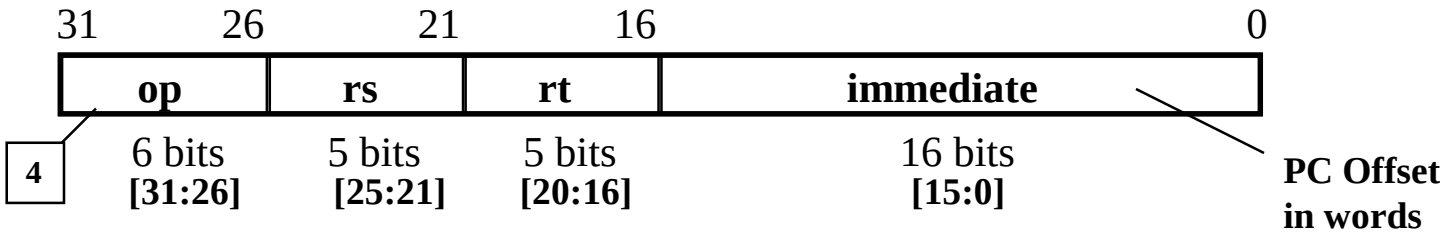
Data Memory

Effective Address

Conditional Branch Example:

Micro-Operation Sequence For BEQ

beq rs, rt, imm16



Instruction Word ← Mem[PC]	**Fetch the instruction**
PC ← PC + 4	**Increment PC**
Zero ← R[rs] - R[rt]	**Calculate the branch condition R[rs] == R[rt] (i.e R[rs] - R[rt] = 0) Then Zero = 1**
Zero : PC ← PC + 4 + (SignExt(imm16) x 4)	**Calculate the next instruction's PC address**

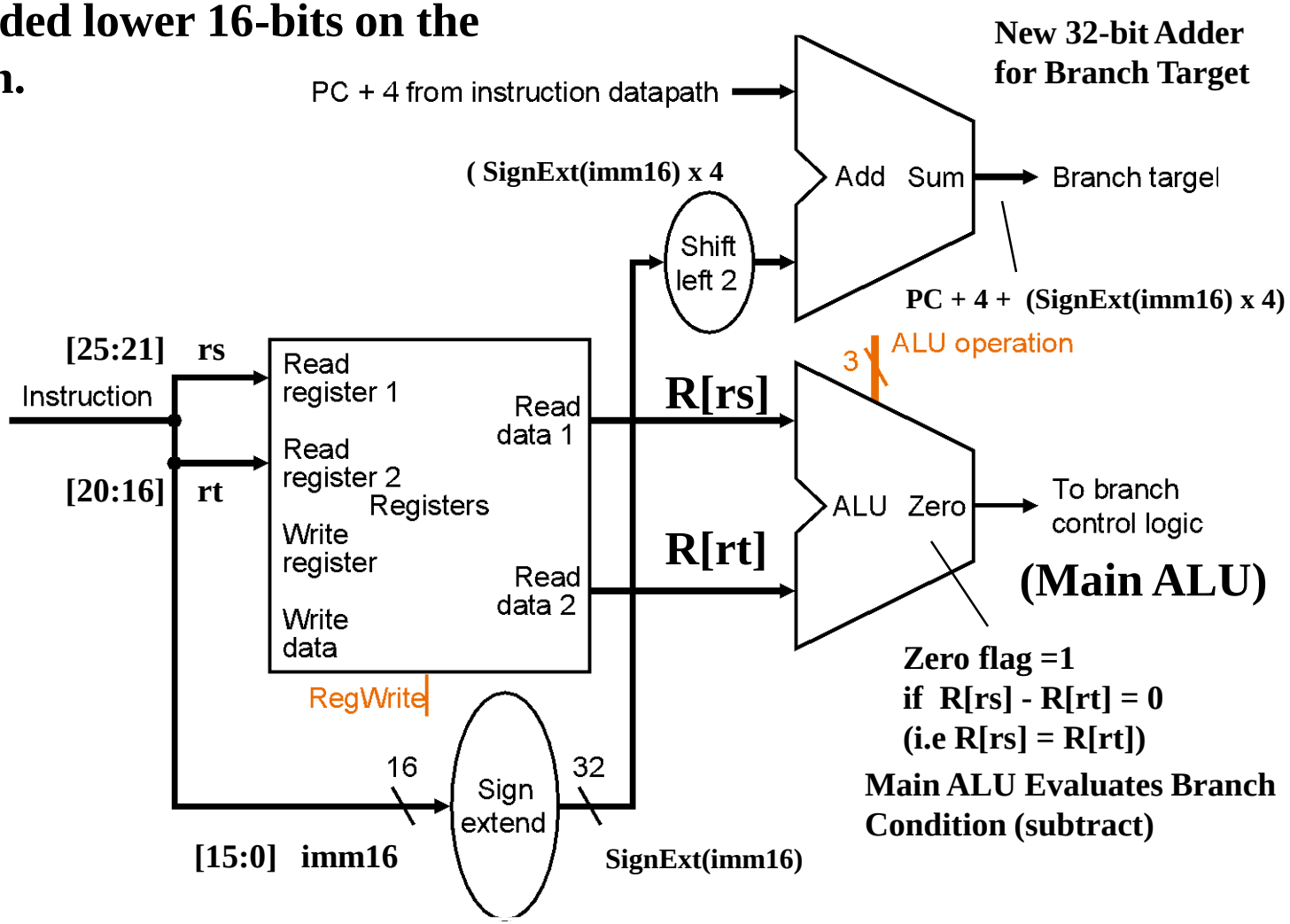
Condition **Action** **Branch Target**

"Zero" is zero flag of main ALU

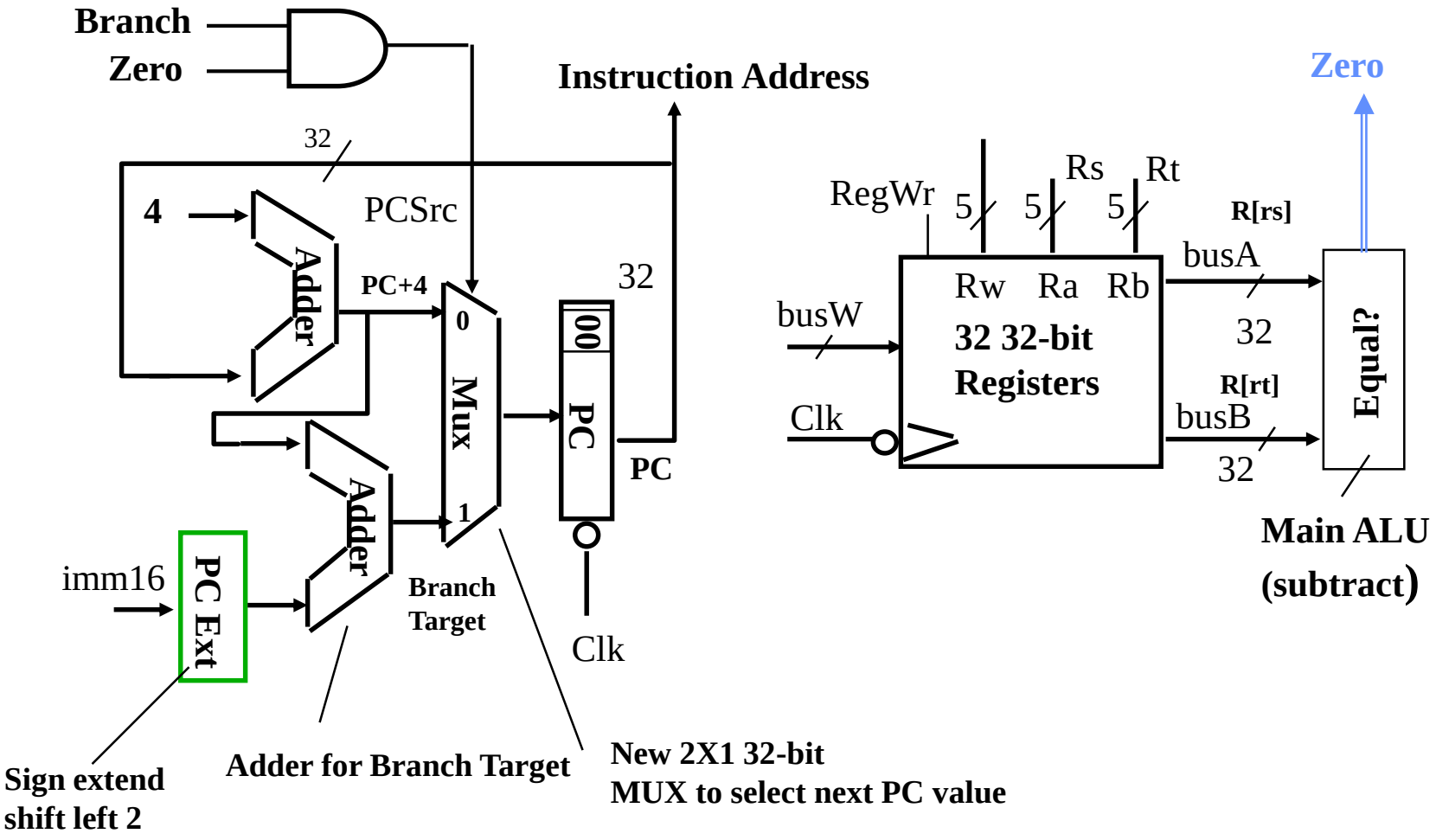
Datapath For Branch Instructions

Main ALU evaluates branch condition
New adder to compute branch target:

- **Sum of incremented PC and the sign-extended lower 16-bits on the instruction.**



More Detailed Datapath For Branch Operations

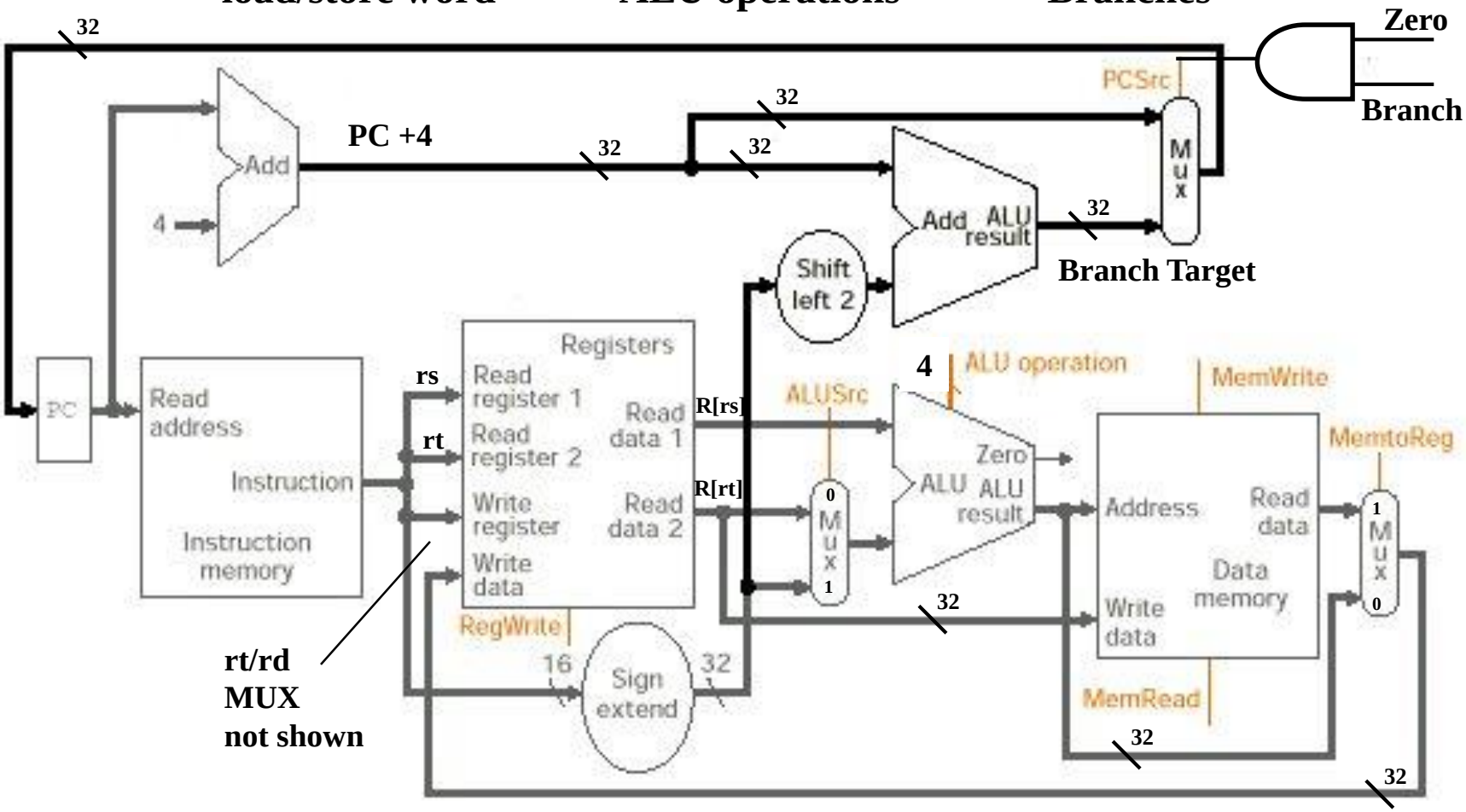


A Simple Datapath For The MIPS Architecture

Datapath of branches and a program counter multiplexor are added.

Resulting datapath can execute in a single cycle the basic MIPS instruction:

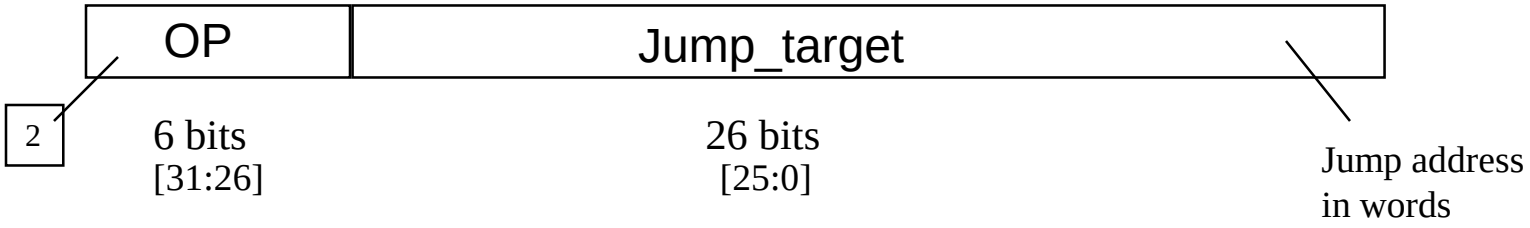
- load/store word - ALU operations - Branches



Adding Support For Jump:

Micro-Operation Sequence For Jump: J

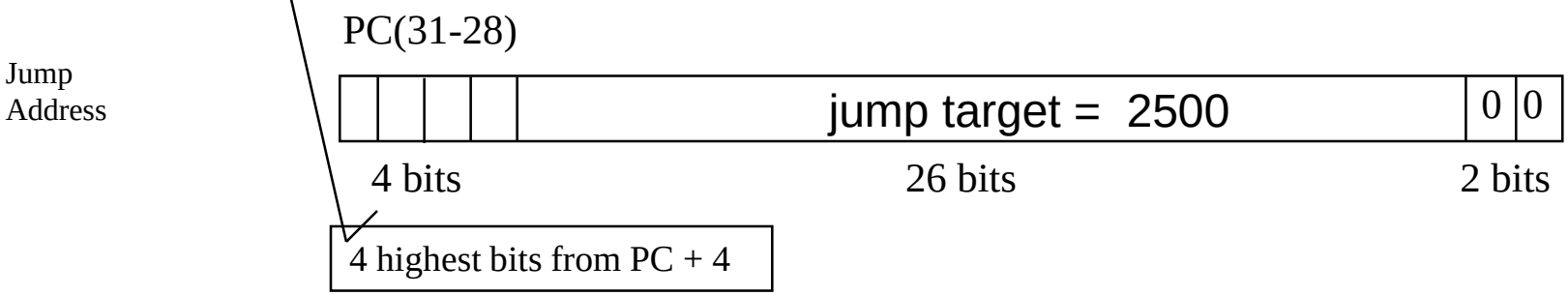
j jump_target



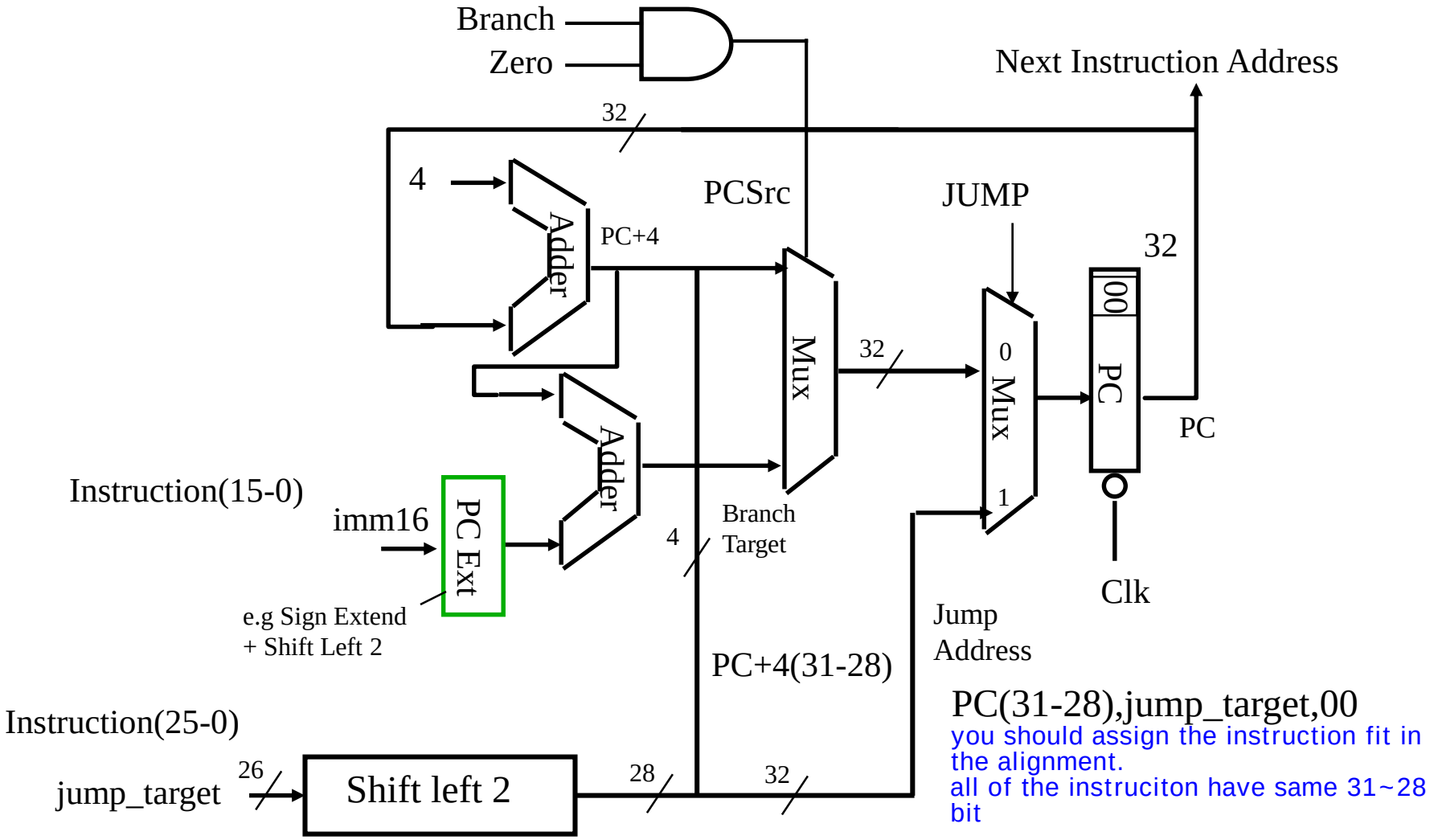
Instruction Word ← Mem[PC] — **Fetch the instruction**

PC ← PC + 4 — **Increment PC**

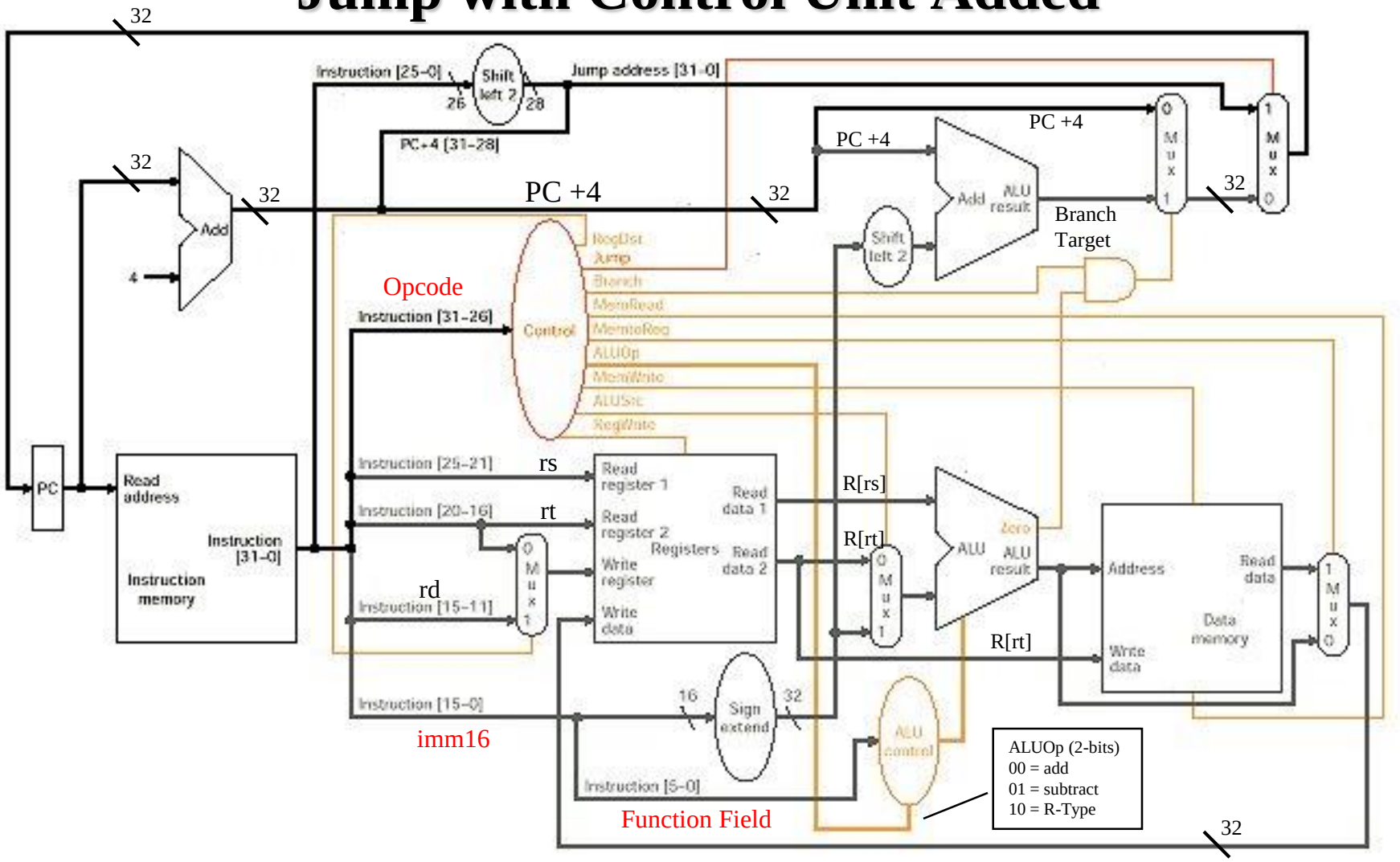
PC ← PC(31-28),jump_target,00 — **Update PC with jump address**



Datapath For Jump



Single Cycle MIPS Datapath Extended To Handle Jump with Control Unit Added



This is book version ORI not supported, no zero extend of immediate needed