WWW.MWFTR.COM

# EECE 417 Computer Systems Architecture

Department of Electrical and Computer Engineering
Howard University

Charles Kim

Spring 2007

# Computer Organization and Design ($3^{rd}$ Ed)

## -The Hardware/Software Interface

by

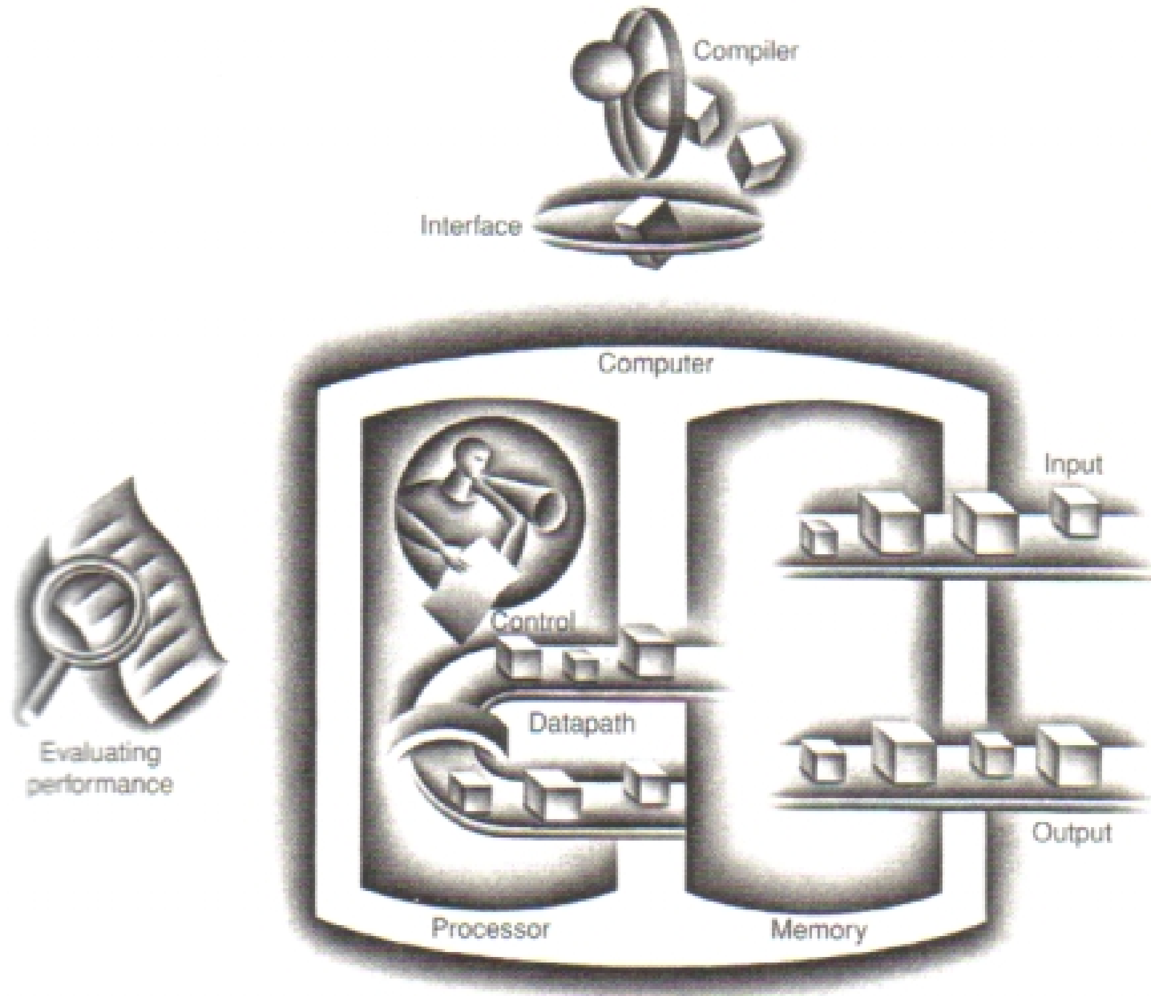
David A. Patterson
John L. Hennessy

# Chapter Five

# The Processor: Detapath and Control

# Part A

# Five Classic Components of a Computer



# Overview

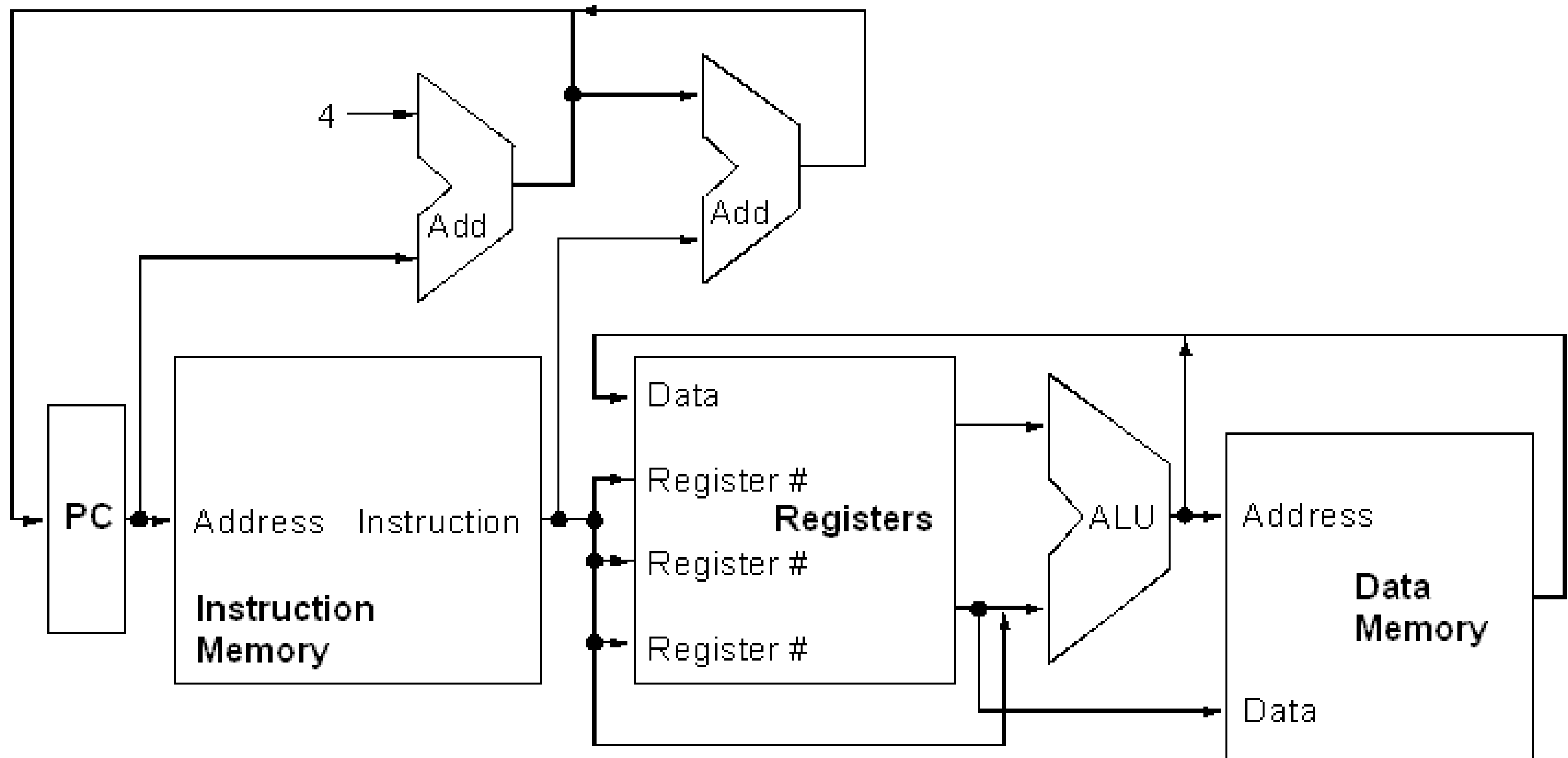
- Performance Key Factors
  - Instruction count
  - Clock cycle time
  - CPI
- What determines the Instruction Count
  - Compiler
  - instruction set
- What determines the Clock Cycle Time and CPI
  - Implementation of the processor
- Focus of the chapter
  - Implementation principle and technique of MIPS
  - Datapath and control

# The Processor: Datapath & Control

- **Implementation of the MIPS - Simplified version**
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`
- **Generic Implementation (for all three types):**
  - Send PC to Memory that contains the code
  - Fetch the instruction from that memory
  - Read registers using the fields of the instruction
  - Execute the Instruction depending on the instruction class
- **Common Actions (across different instructions)**
  - Step 1:
    - **Use ALU for operation execution after reading registers**
    - **Use ALU for address calculation (for memory-reference instr)**
  - Step 2
    - **Memory access to load or store**
    - **write back to register**

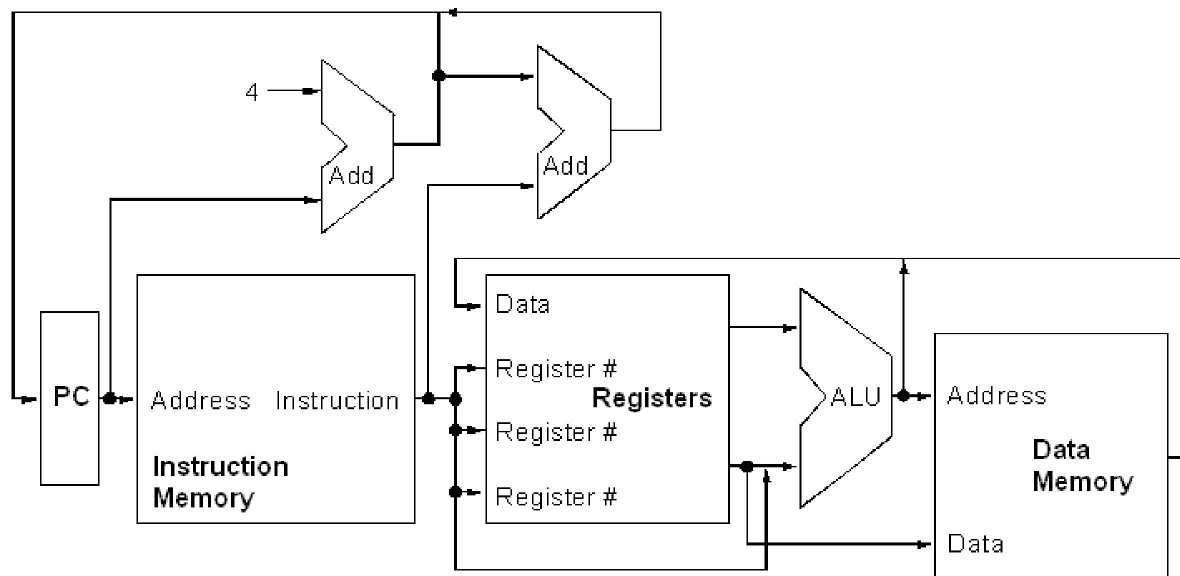
# High Level view of MIPS Implementation

- Abstract / Simplified View:
- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)

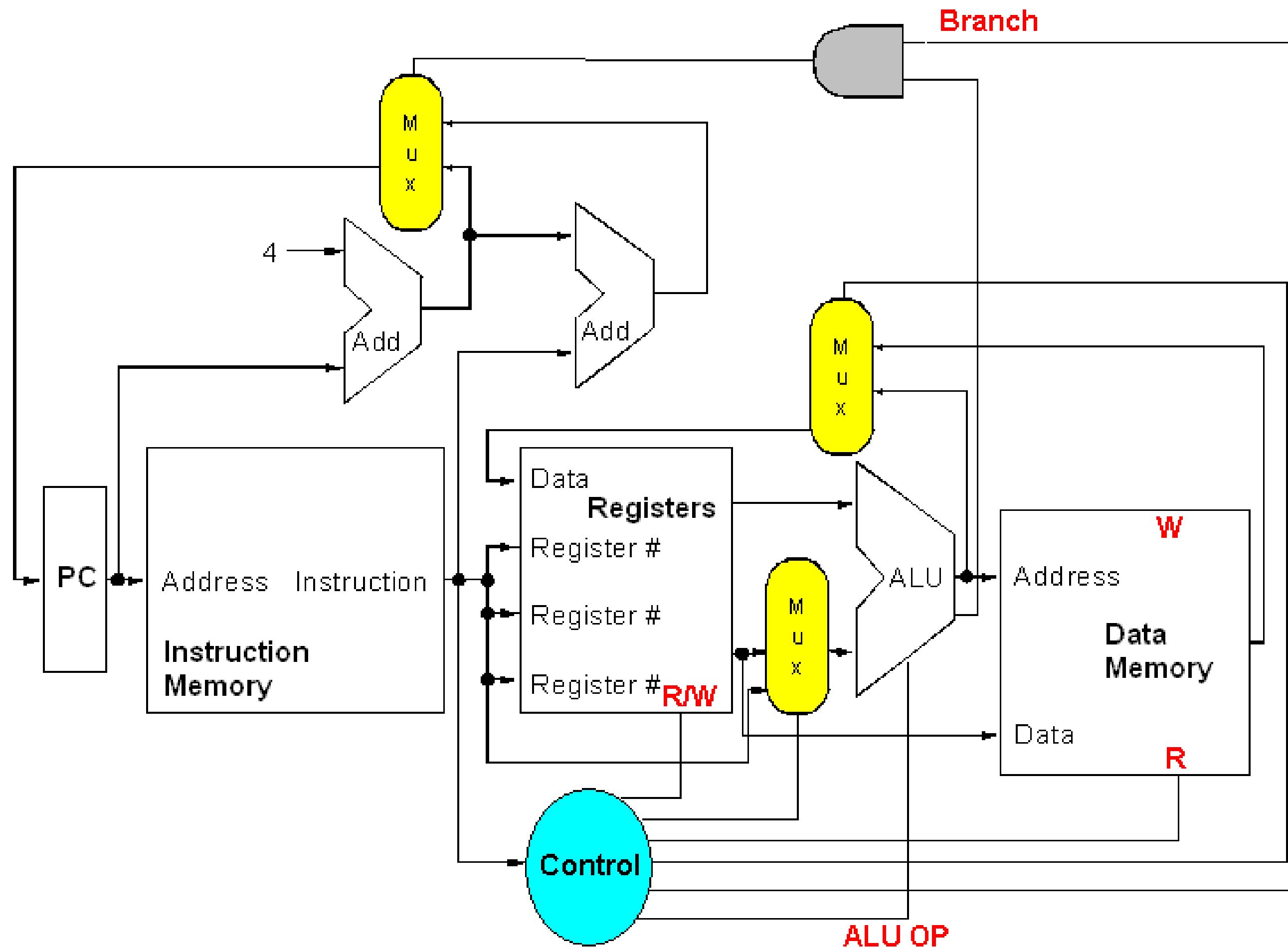


# What's missing here in the High Level view of MIPS Implementation

- No multiplexer (or data selector)
  - Data lines to PC
  - Data lines to Registers
  - Cannot be wired together
- No Control
  - Data memory read (on load) or write (on store)
  - read from and write to register



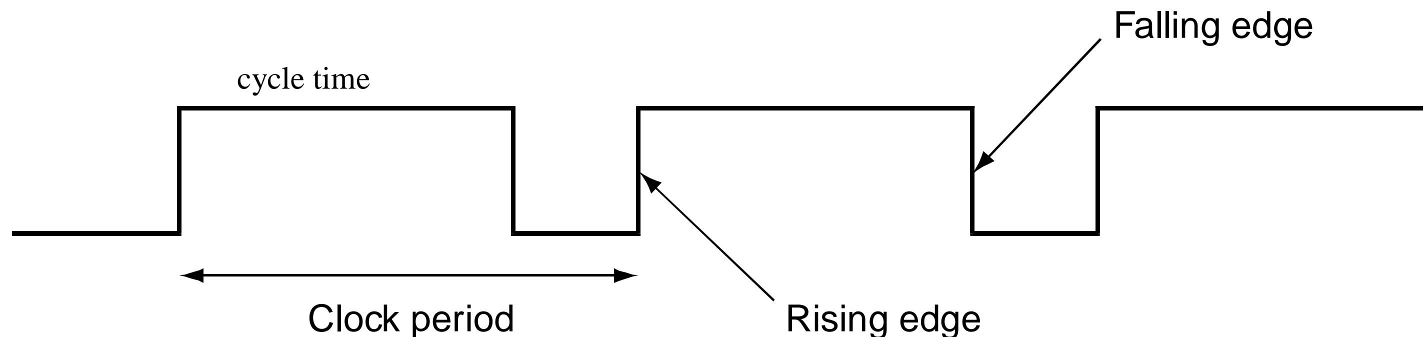
# MIPS Implementation with necessary elements




- What comes next?
  - Element and Components
  - Building Datapath with major components
  - Control

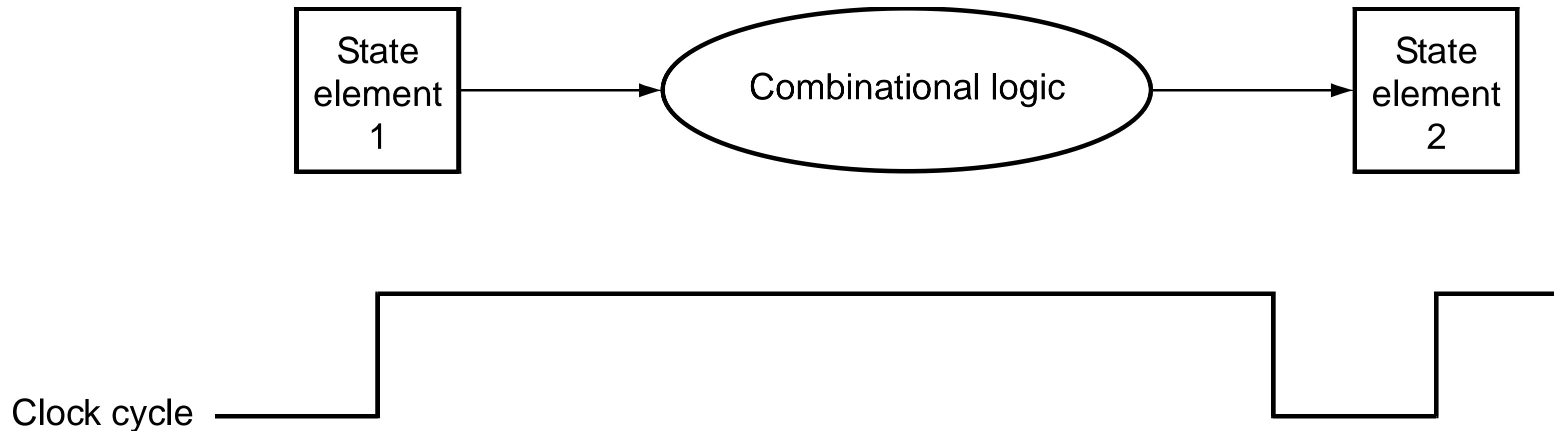
# Logic Design Convention

- State Elements
  - Internal Storage to indicate a state (sequential)
  - 2 inputs (data and clock) and 1 output
  - Unclocked vs. Clocked
  - Clocks used in synchronous logic
    - **when should an element that contains state be updated?**



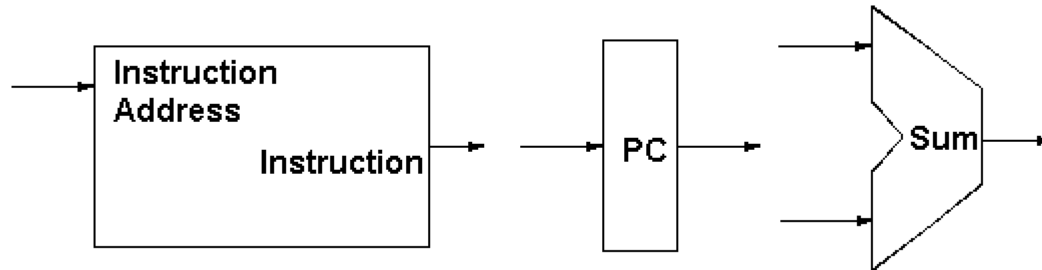
# Synchronous Digital System

- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements

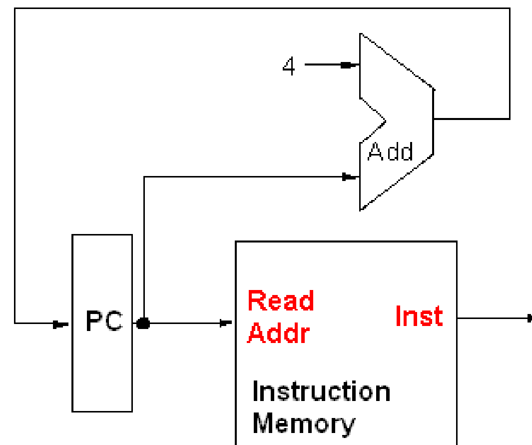


# Datapath Elements - for instruction

- Components required to execute each class of MIPS instruction
- Instruction Memory - instruction store
- PC - hold address of the current instruction
- Adder - Increment the PC to the address of next instruction

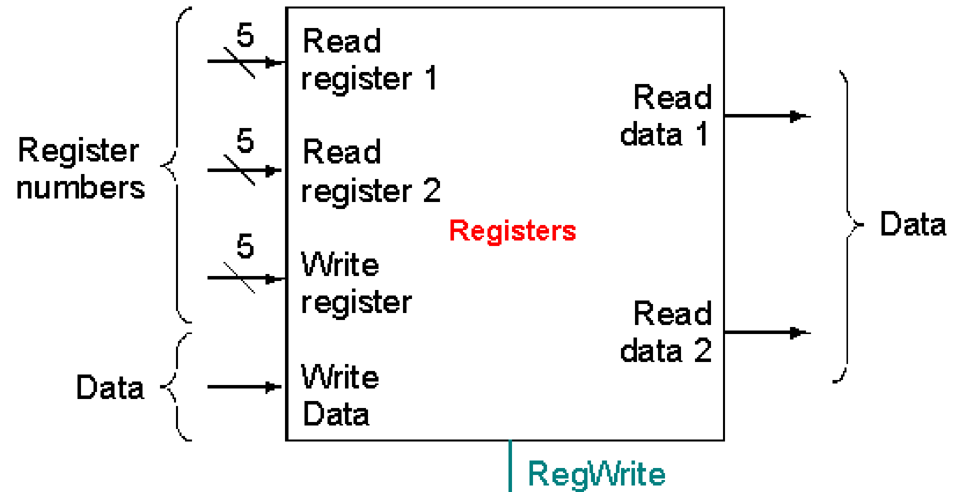


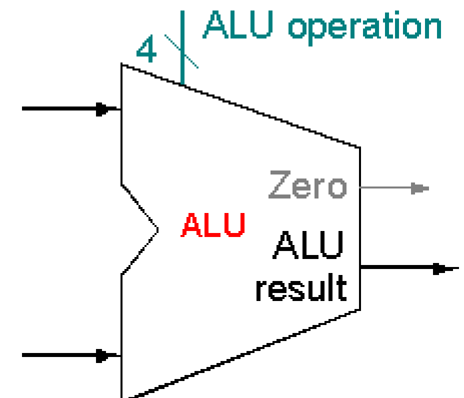

- Fetching Instructions and Incrementing PC by 4



# Elements for R-type Instructions

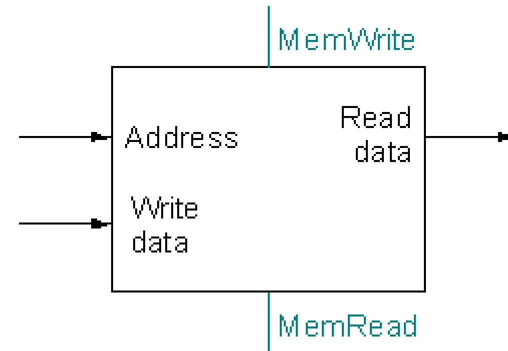
- R-type instructions
  - `add, sub, and, or,` etc
  - Read two registers
  - perform ALU operation
  - Write the result
- Needed Elements
  - Register File
    - 32 registers
    - 2 read ports
    - 1 write port
  - ALU
    - Operation on the values of read from registers
    - two 32-bit inputs
    - one 32-bit output
    - 1-bit signal if output is zero
    - 4-bit ALU control signal

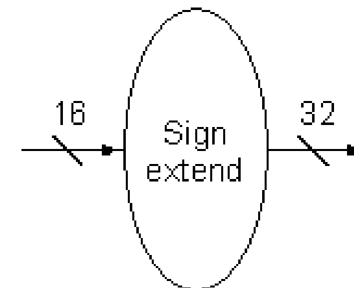


# Elements for Load and Store

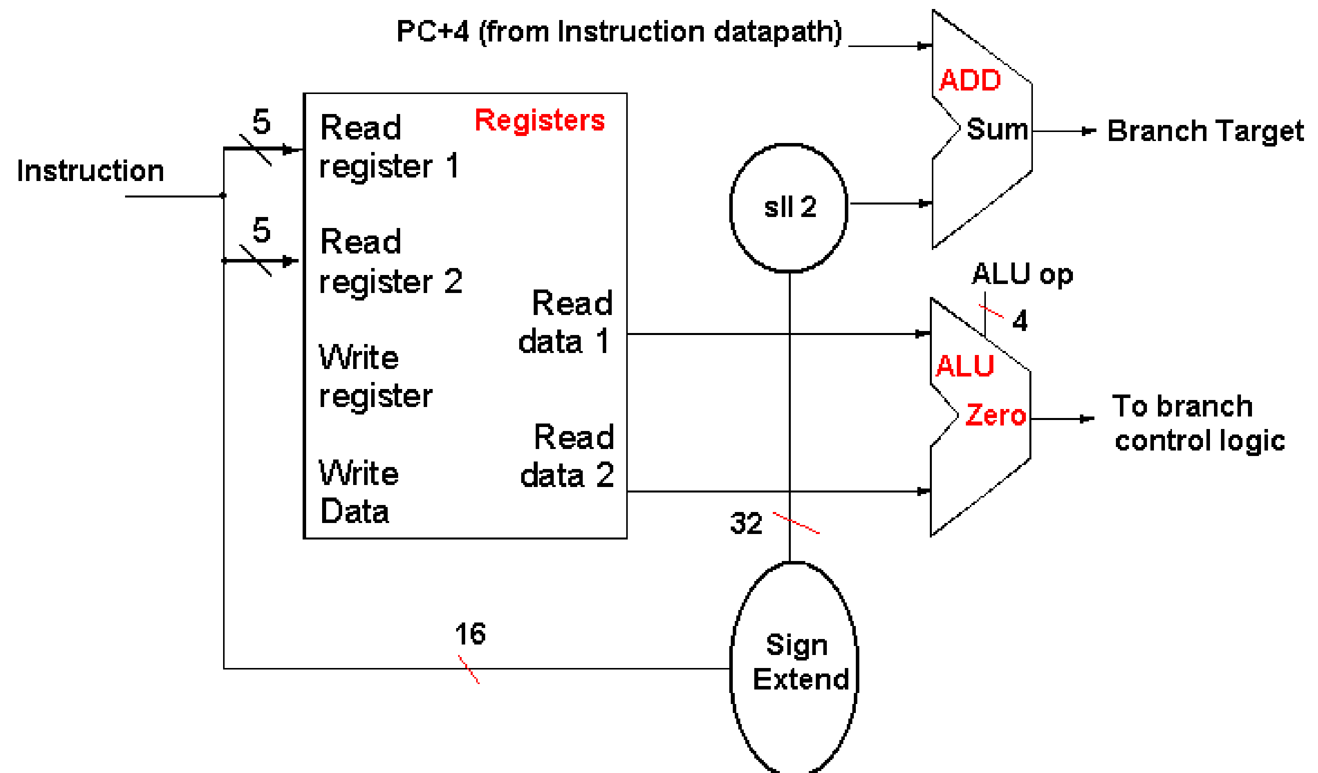
- Load and Store
  - `lw $t1, offset($t2)`
  - Memory address = base_addr [$t2] + 16-bit signed offset
  - Registers are involved
- Elements Needed
  - Data Memory
  - Register File
  - ALU
  - Sign-Ext unit
    - **extension of 16-bit offset filed into 32-bit signed value**



# Datapath for Branch Target Address

- `beq $t1, $t2, offset` : branch to PC+offset if [$t1]=[$t2]
  - 3 operands
  - 2 registers
  - 16-bit offset
  - Byte offset into Word offset (x4)



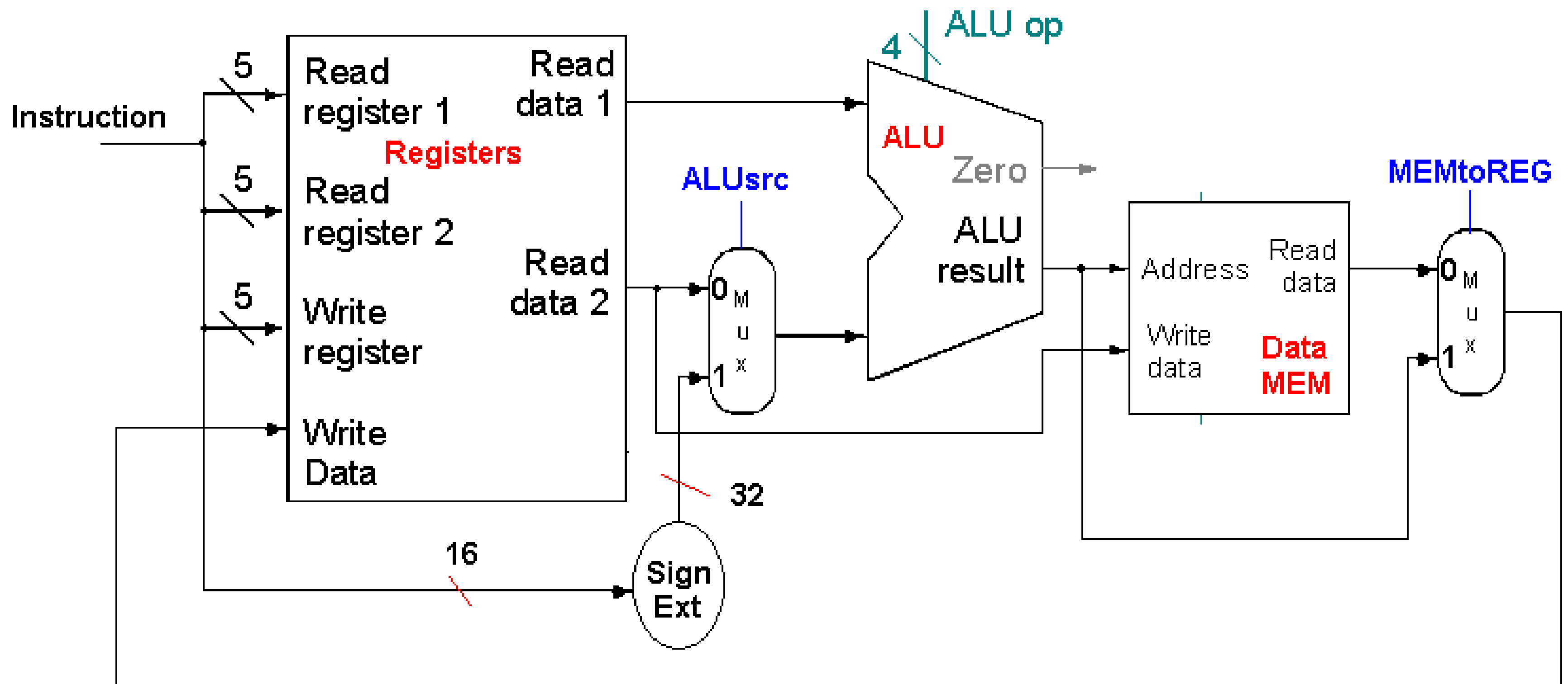
# Building a Datapath

- Combination of datapath components into a single datapath
- Add control
- Attempt to execute all instructions in one clock cycle
  - no datapath resource can be used more than once per instruction
  - any element needed more than once must be duplicated
  - Separation of instruction memory from data memory
- Share of datapath elements for different instruction flows
  - multiplexor
  - control signal

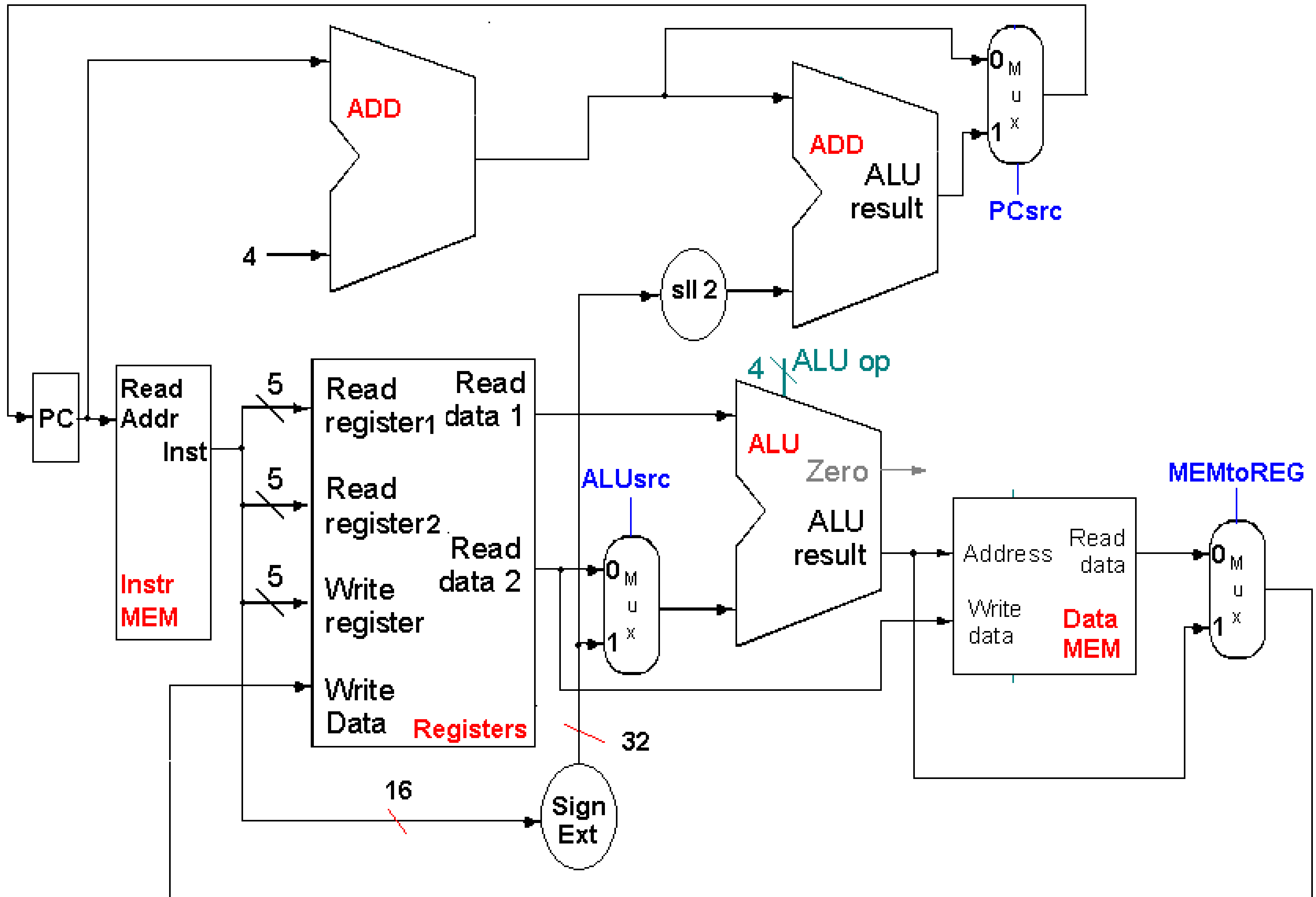
# Building a Datapath - Example

- Q: Build a datapath for the operational portion of the memory reference and arithmetic-logical instructions that uses a single register file and a single ALU to handle both types of instructions, adding any necessary mux.
- Analysis:
  - ALU
    - **R-type -- for two registers**
    - **Mem - add calculation (second input is sign-ext)**
  - Destination Reg
    - **T-type -- from ALU**
    - **Mem -- from Mem**
- Approach
  - 2 different sources for (2nd) ALU input ---->MUX
  - 2 different sources for data stored into register file --->MUX

# Datapath for MEM instr and R-type instr



# Datapath for MEM, R-type, and Branch Instr



# A Simple Implementation Scheme

- Simplest possible implementation of MIPS subset using the datapath we discussed before plus simple control function
- Covers load (`lw`), store (`sw`), branch equal(`beq`), add (`add`), sub (`sub`), and (`and`), or(`or`), set on less than (`slt`) only --"**subset**"
- ALU Control (with 4 control inputs)

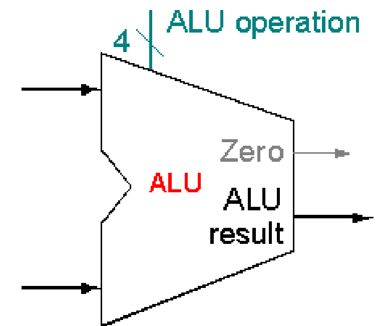
```
0000   and
0001   or
0010   add
0110   sub
0111   slt
1100   nor
```

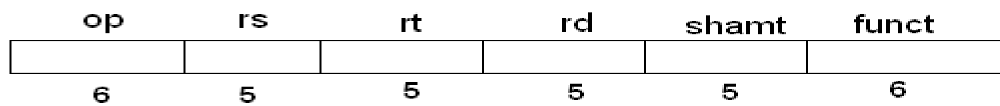
| ALU control lines | Function |
|---|---|
| 0000 | AND |
| 0001 | OR |
| 0010 | add |
| 0110 | subtract |
| 0111 | set on less than |
| 1100 | NOR |

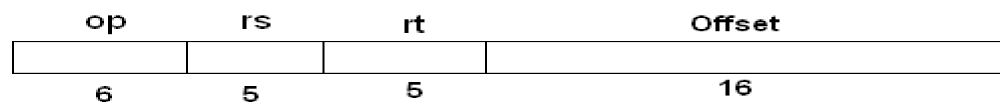
- Generation of 4-bit ALU control is not simple--what's involving?
  - 2-bit ALUop ("control" bit) --- Instruction classes
  - 6-bit function filed (in machine code)
  - 4-bit info from 8 bits?

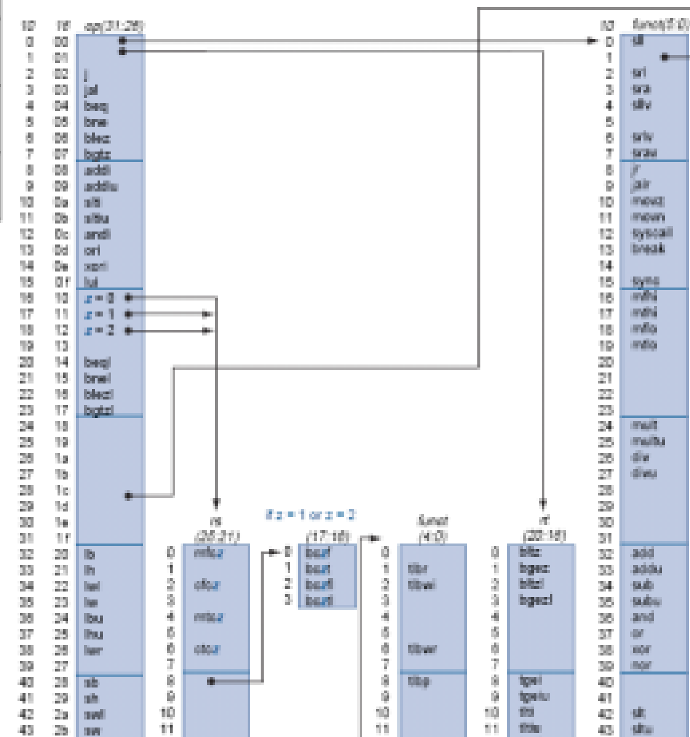
# ALUop and Function Fields

| Instruction Opcode | ALUop | Instruction Operation | Function Field | Desired ALU action | ALU control Input |
|---|---|---|---|---|---|
| LW | 00 | load word | xxxxxx | add | 0010 |
| SW | 00 | store word | xxxxxx | add | 0010 |
| Branch Equal | 01 | branch equal | xxxxxx | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| R-type | 10 | subtract | 100010 | subtract | 0110 |
| R-type | 10 | AND | 100100 | and | 0000 |
| R-type | 10 | OR | 100101 | or | 0001 |
| R-type | 10 | set on less than | 101010 | set on less than | 0111 |



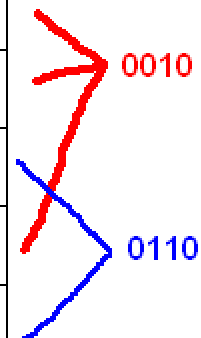





# Truth Table for ALU control bits

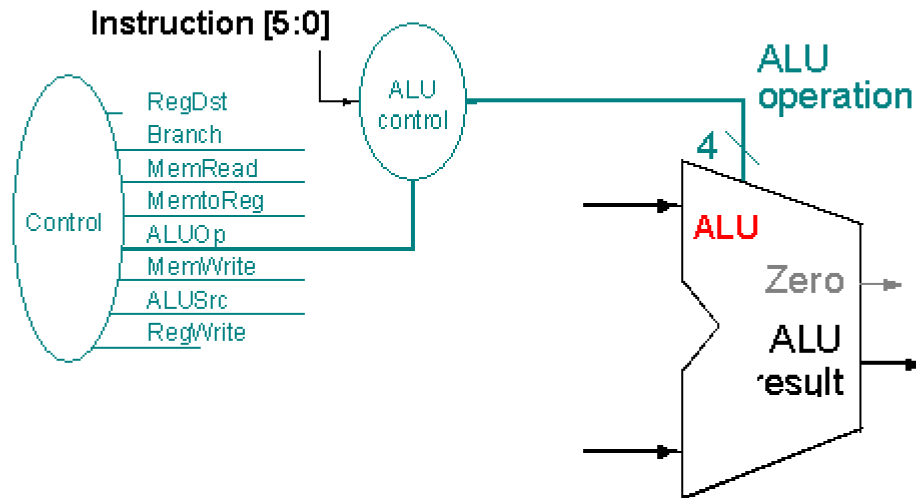
| Instruction Opcode | ALUop | Instruction Operation | Function Field | Desired ALU action | ALU control Input |
|---|---|---|---|---|---|
| LW | 00 | load word | XXXXXX | add | 0010 |
| SW | 00 | store word | XXXXXX | add | 0010 |
| Branch Equal | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| R-type | 10 | subtract | 100010 | subtract | 0110 |
| R-type | 10 | AND | 100100 | and | 0000 |
| R-type | 10 | OR | 100101 | or | 0001 |
| R-type | 10 | set on less than | 101010 | set on less than | 0111 |



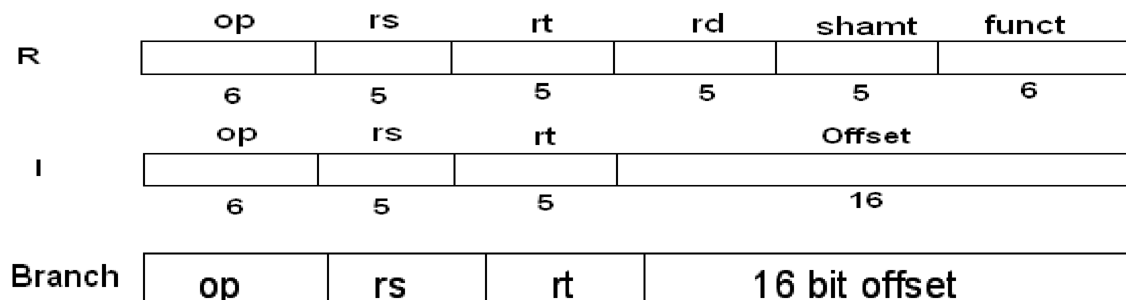

| ALUOp | | Funct field | | | | | | Operation |
|---|---|---|---|---|---|---|---|---|
| ALUOp1 | ALUOp0 | F5 | F4 | F3 | F2 | F1 | F0 | |
| 0 | 0 | X | X | X | X | X | X | 0010 |
| X | 1 | X | X | X | X | X | X | 0110 |
| 1 | X | X | X | 0 | 0 | 0 | 0 | 0010 |
| 1 | X | X | X | 0 | 0 | 1 | 0 | 0110 |
| 1 | X | X | X | 0 | 1 | 0 | 0 | 0000 |
| 1 | X | X | X | 0 | 1 | 0 | 1 | 0001 |
| 1 | X | X | X | 1 | 0 | 1 | 0 | 0111 |

# Main Control Unit

- Schematic for ALU operation

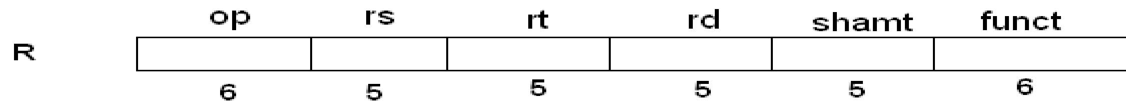



- Design of Main Control Unit
  - Check the instruction formats first

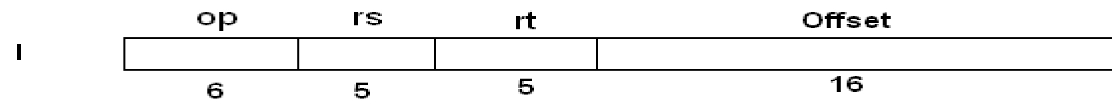


# Main Control Unit Datapath

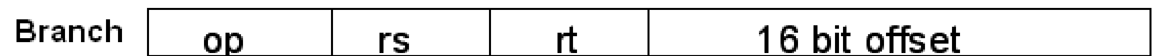
- Observation of the Instruction Formats
  - R type
    - Opcode: 0
    - Sources: rs and rt
    - Destination: rd



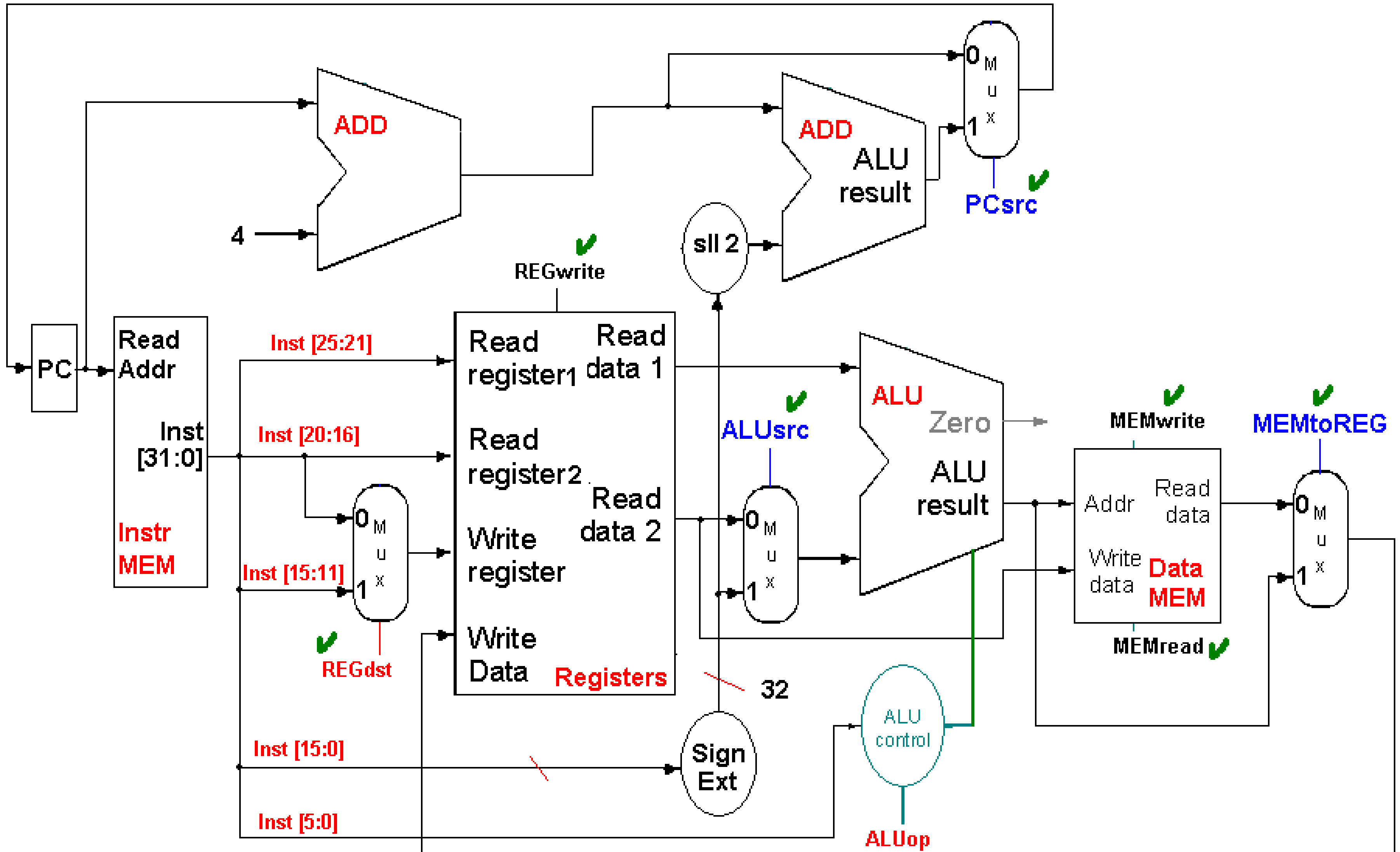

  - Load/Store Type
    - Opcode: 35 (load) or 43 (store)
    - Base addr: rs
    - Destination: rt (load)
    - Source: rs(store)



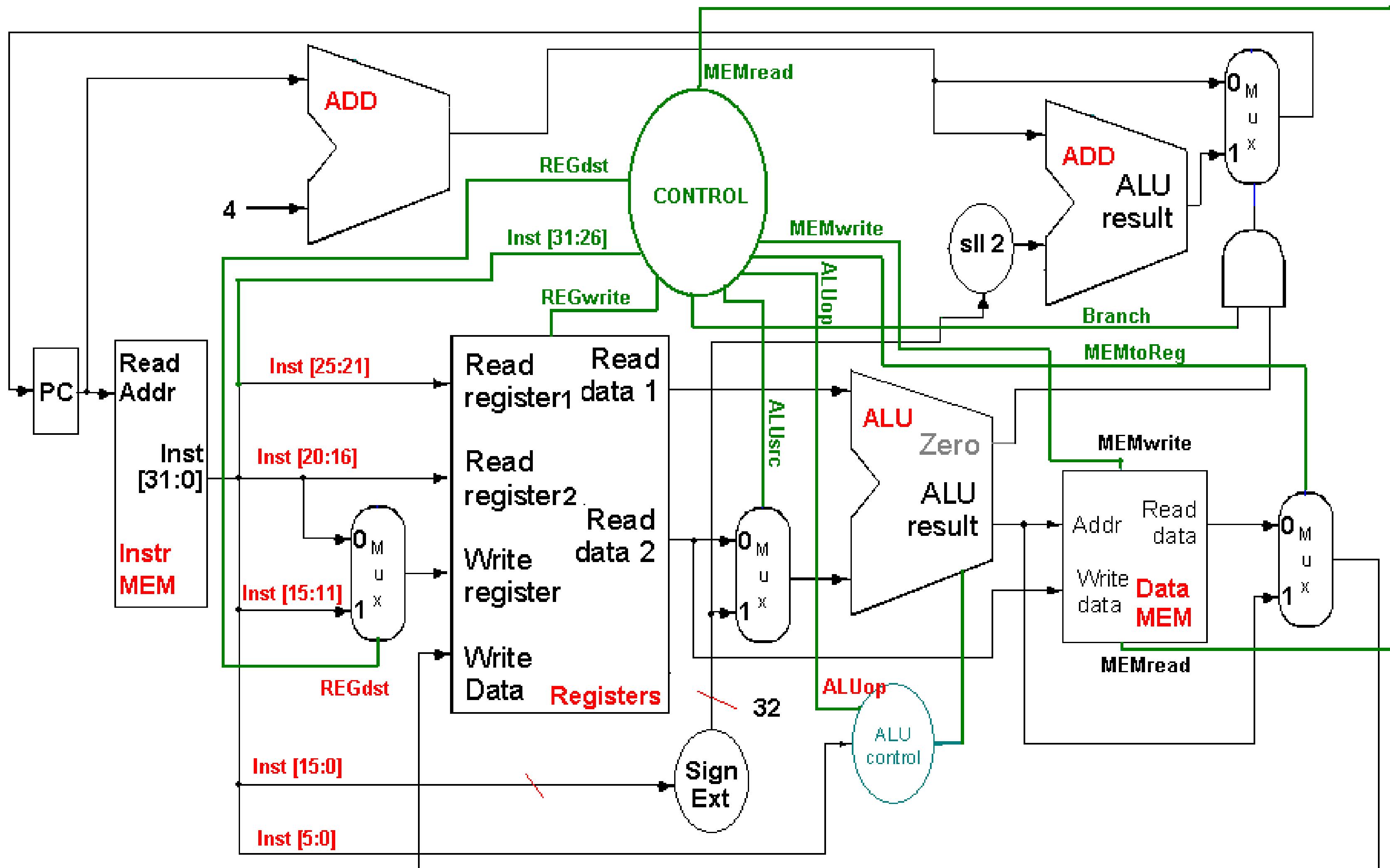

  - Branch Type
    - Opcode: 4
    - Base addr: rs
    - Destination: rt



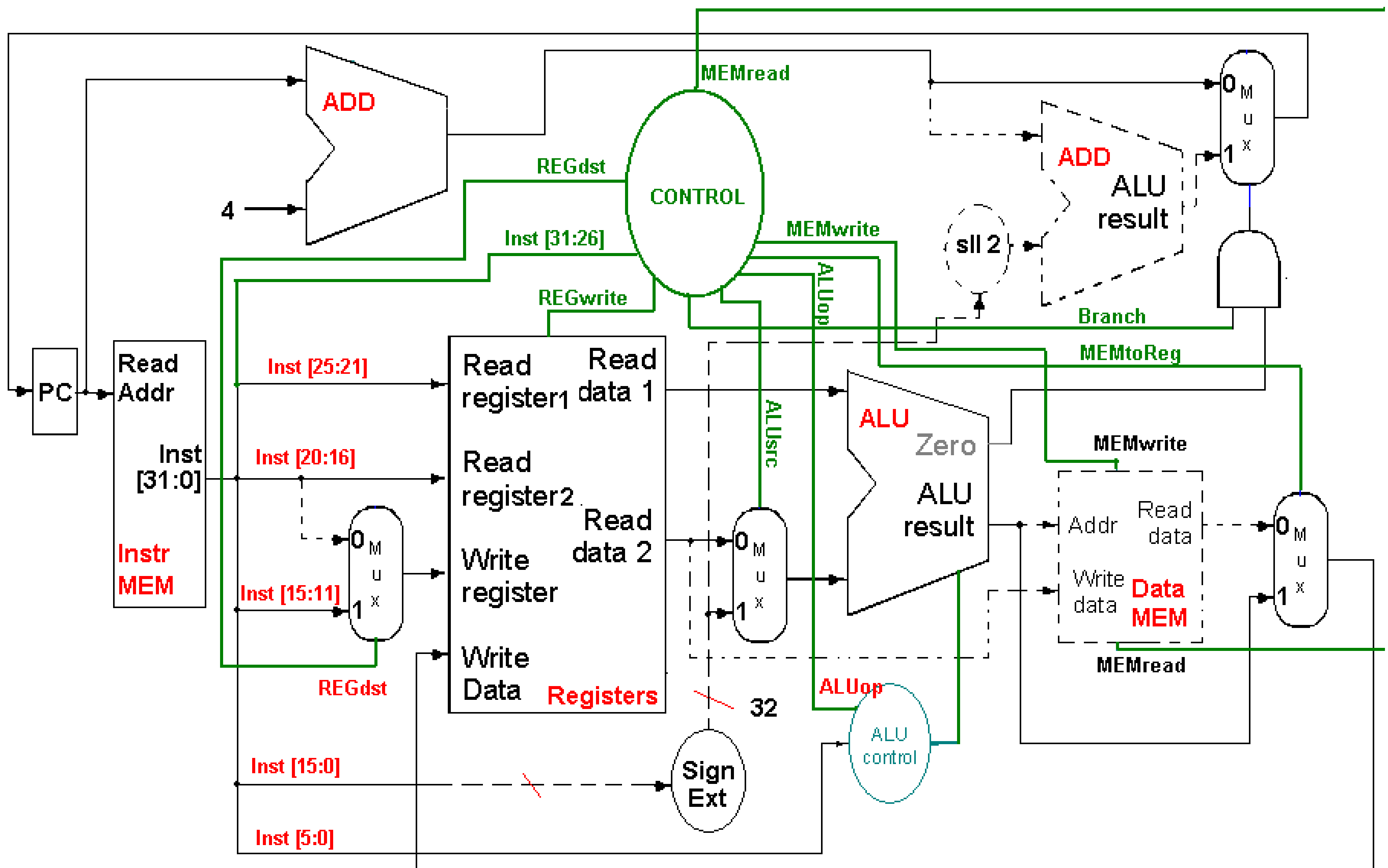
# Control Lines



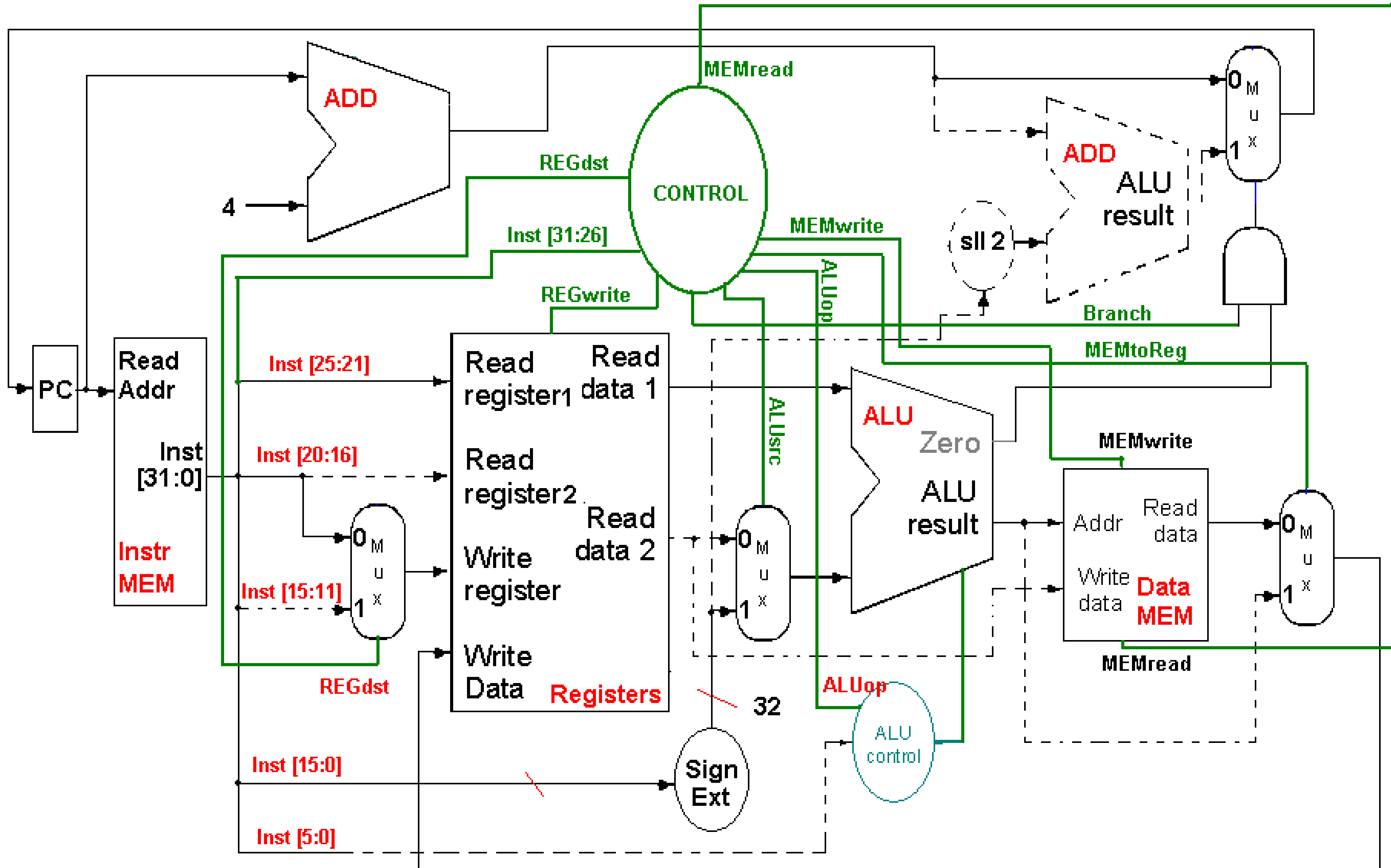
# Control Lines (also for beq)



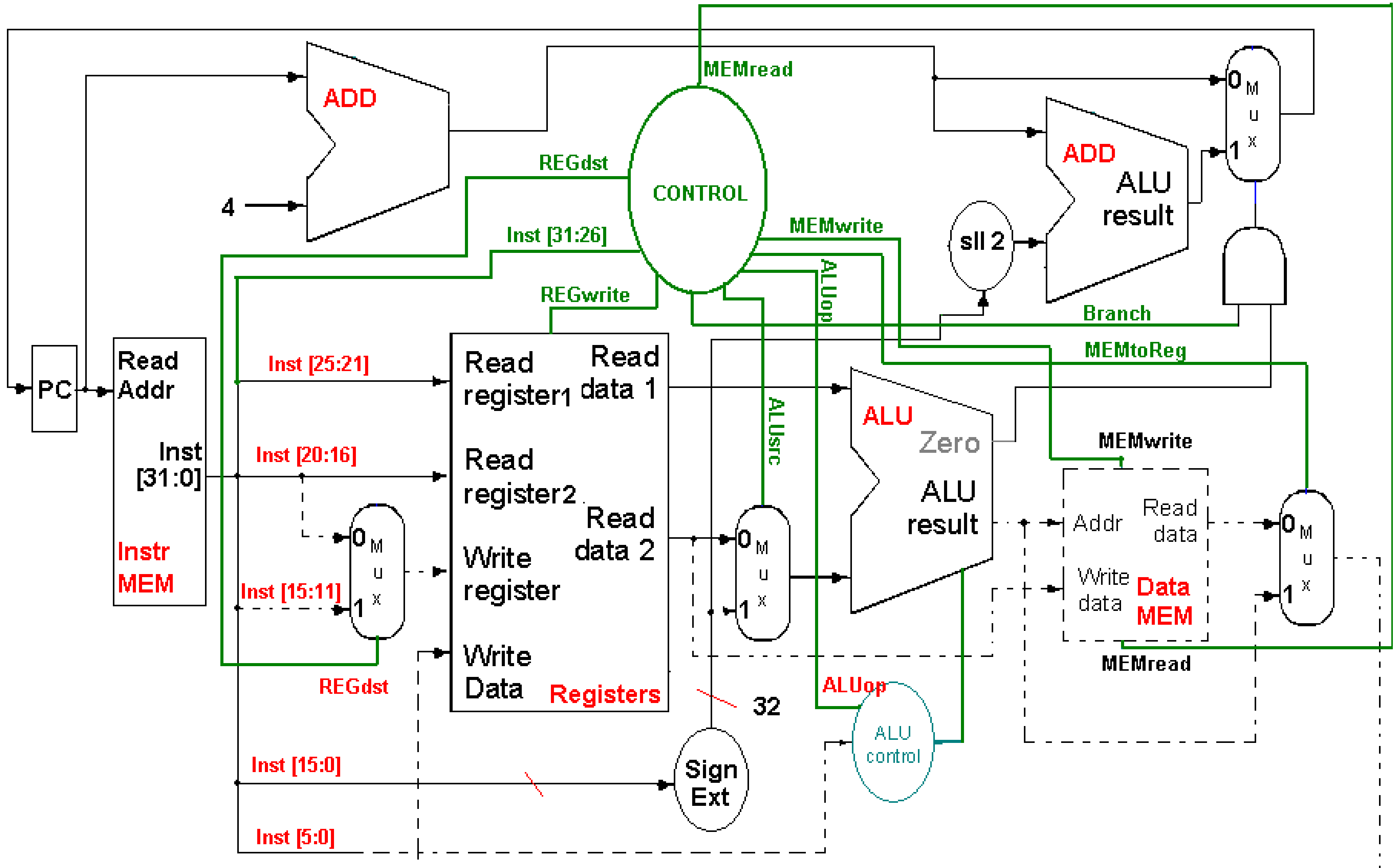
# Operation of Datapath for `add $t1, $t2, $t3`



# Operation of Datapath for `lw $t1, offset($t2)`



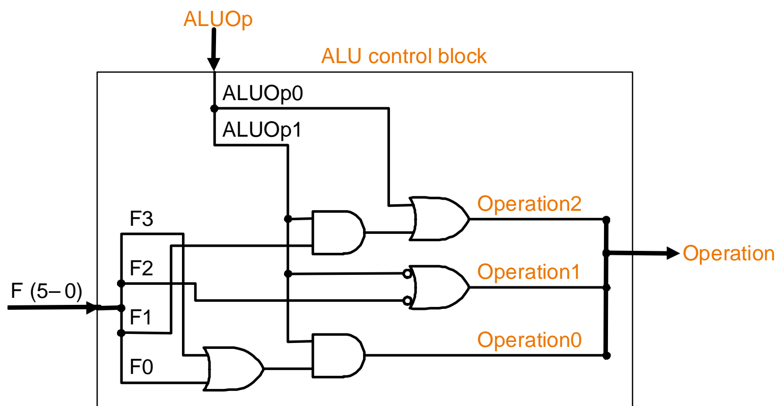
# Operation of Datapath for beq $t1, $t2,offset

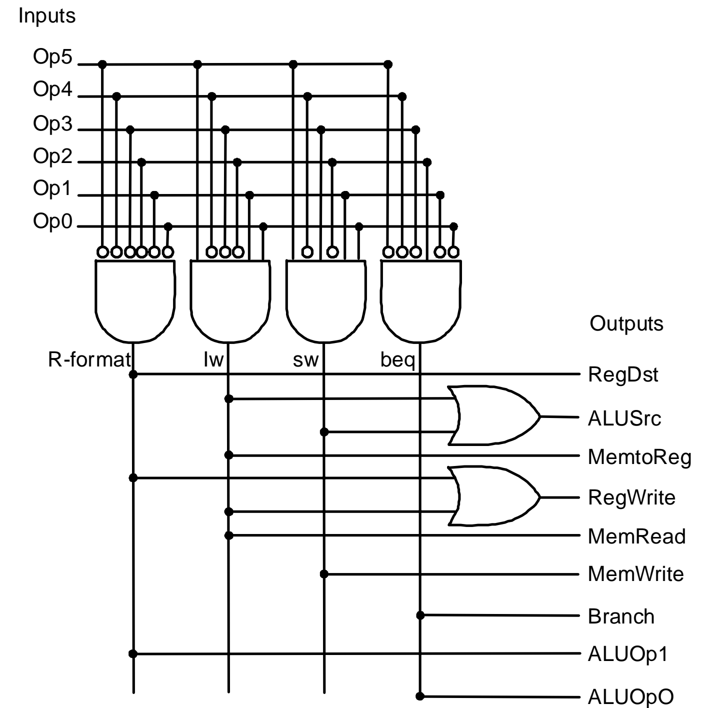


# Control Implementation

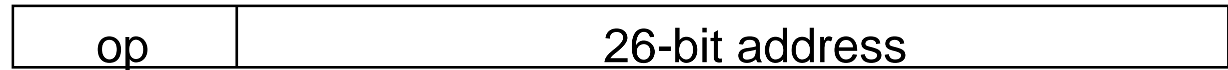
| Instruction | RegDst | ALUSrc | Memto-Reg | Reg Write | Mem Read | Mem Write | Branch | ALUOp1 | ALUp0 |
|---|---|---|---|---|---|---|---|---|---|
| R-format | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| `lw` | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| `sw` | X | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 |
| `beq` | X | 0 | X | 0 | 0 | 0 | 1 | 0 | 1 |

- Simple combinational logic (truth tables)

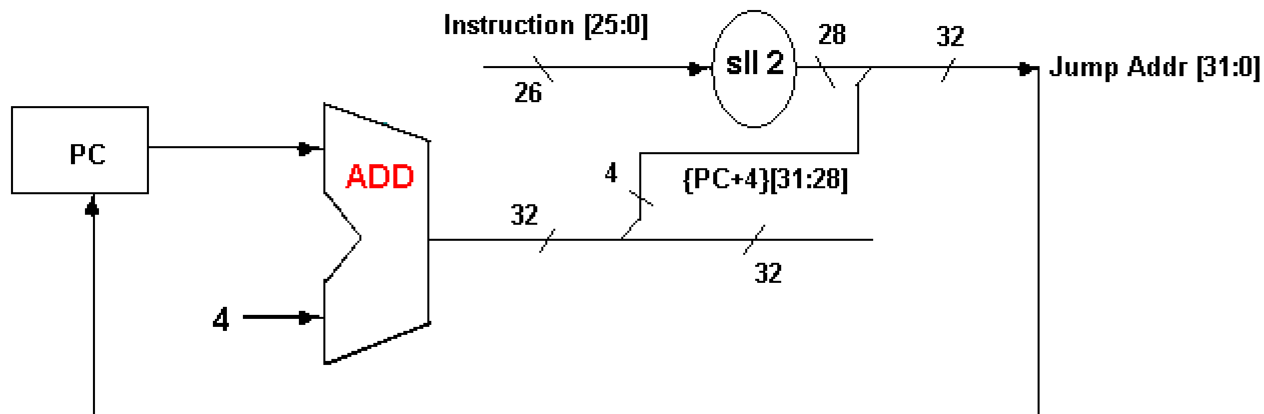


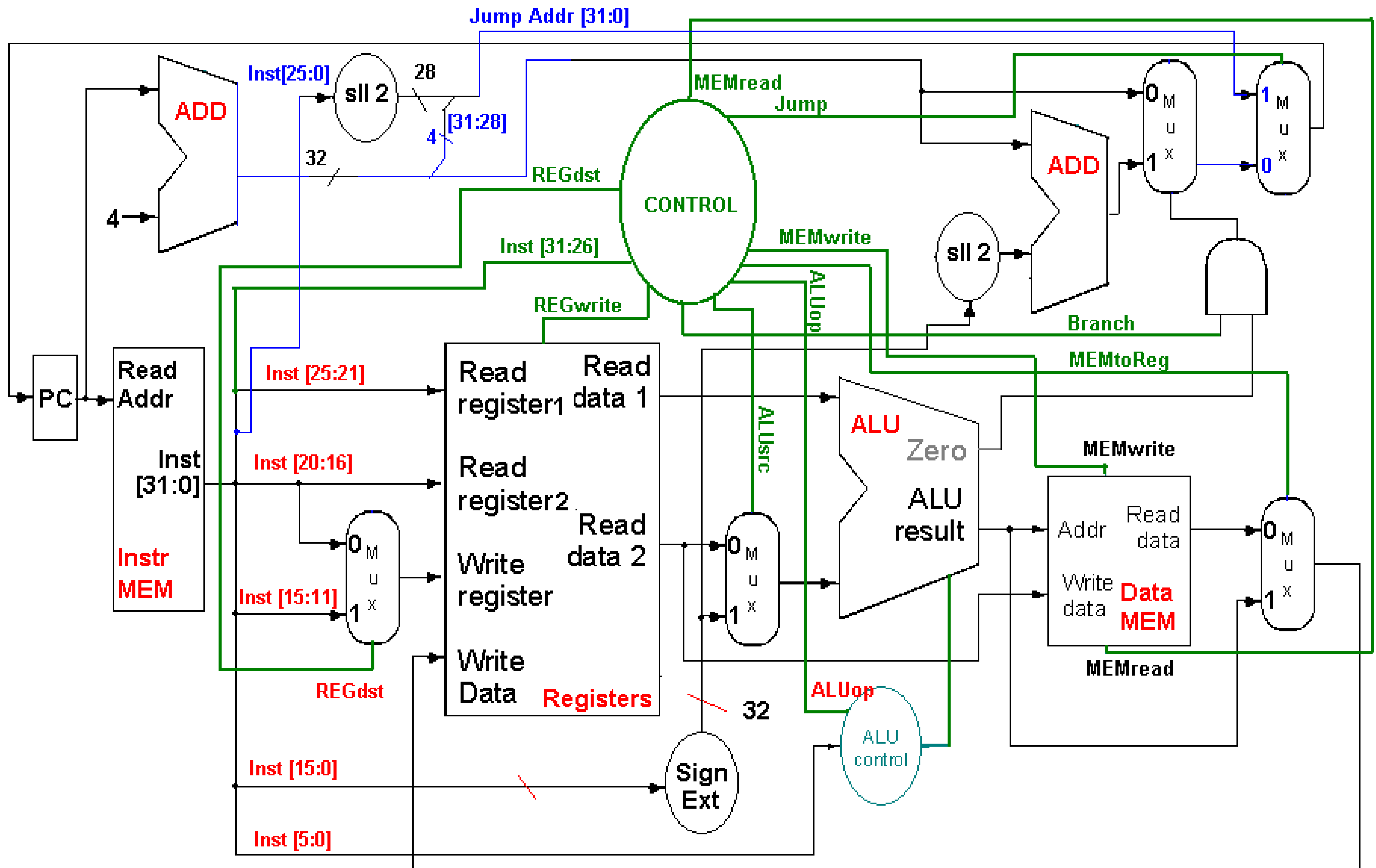
# Addition Jump Instruction to Datapath



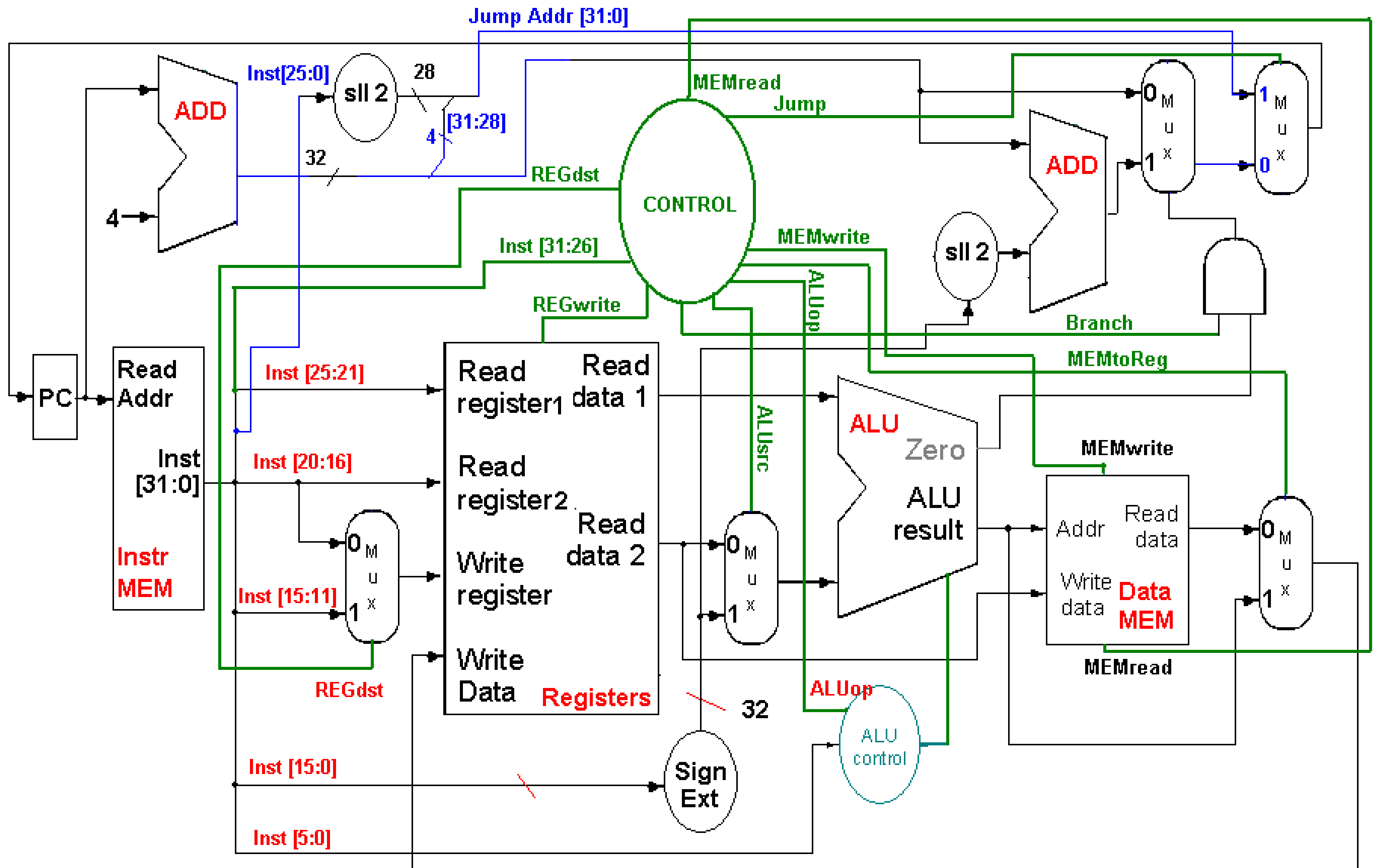

- Lower 2 bits of jump address is always 00 (as in branch) as the last two bits. Why? Word address --> Byte address
- 26-bit address from immediate field in the middle
- Upper 4 bits of the current (PC+4) occupies the upper 4 bits of the address
- Summary for jump address [31:0]
  - jump addr[31:0]= {PC+4} [31:28]//Instr[25:0]//00



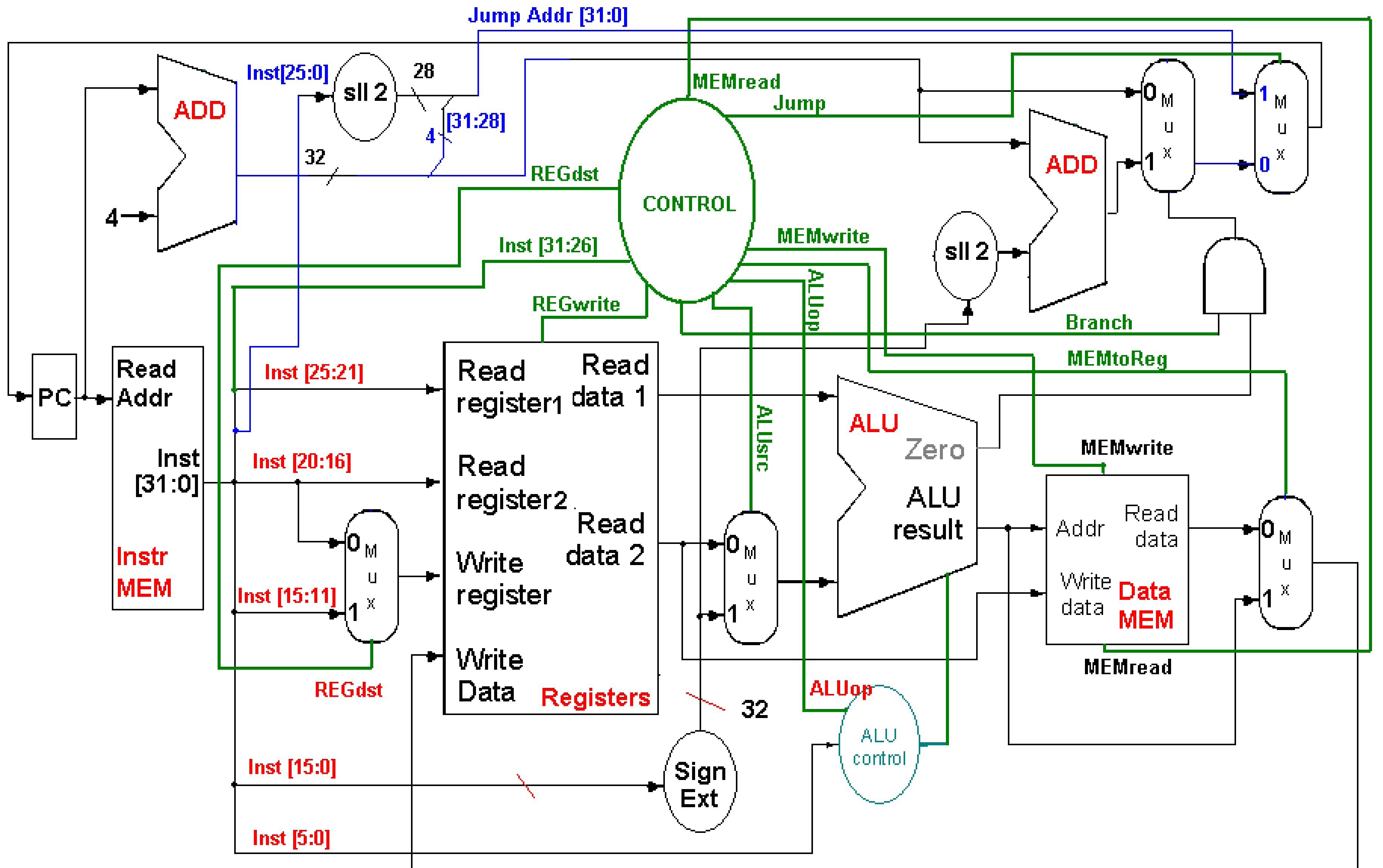
# Final Datapath Including R, L, Branch, and Jump



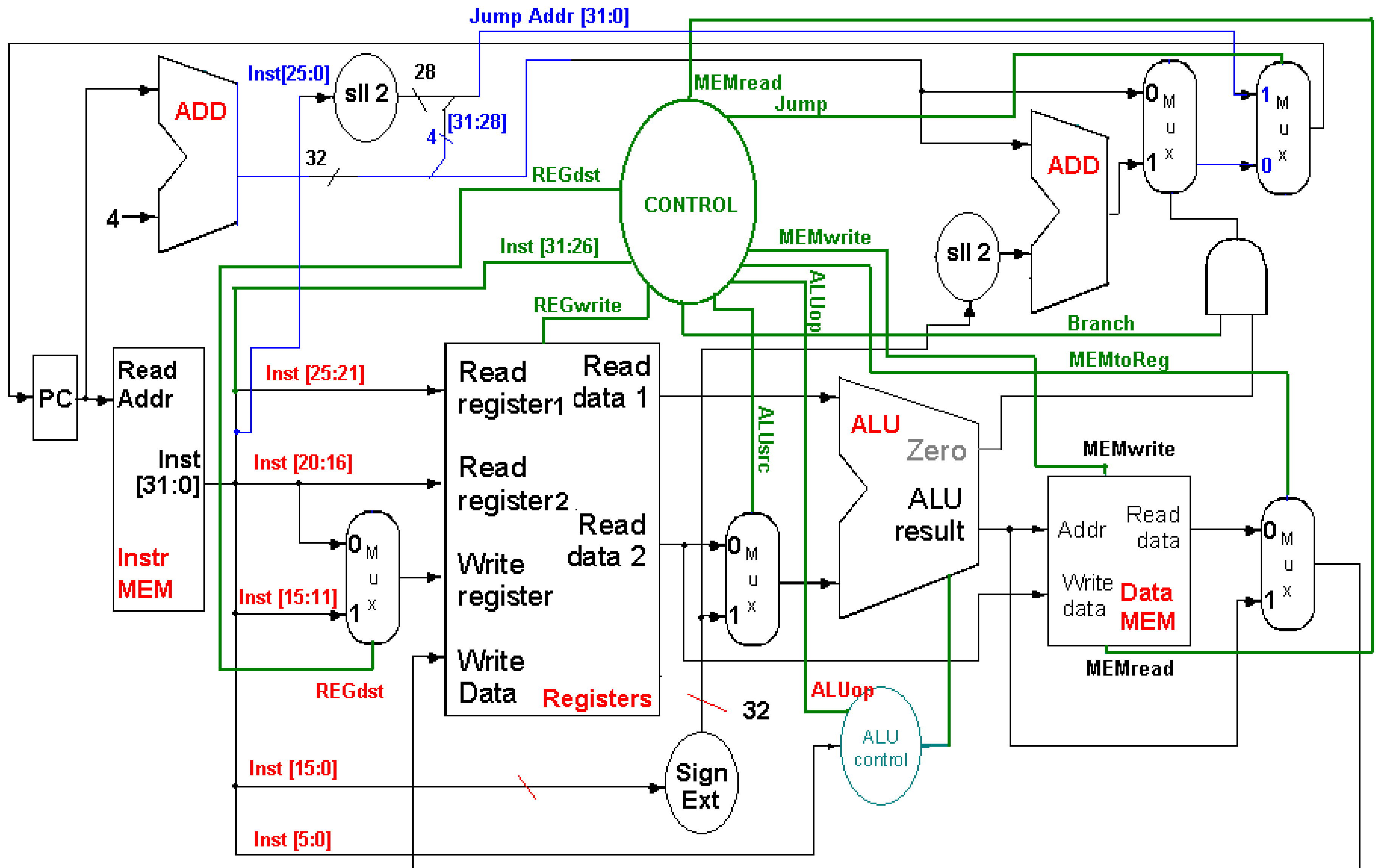
# R-type instruction Flow



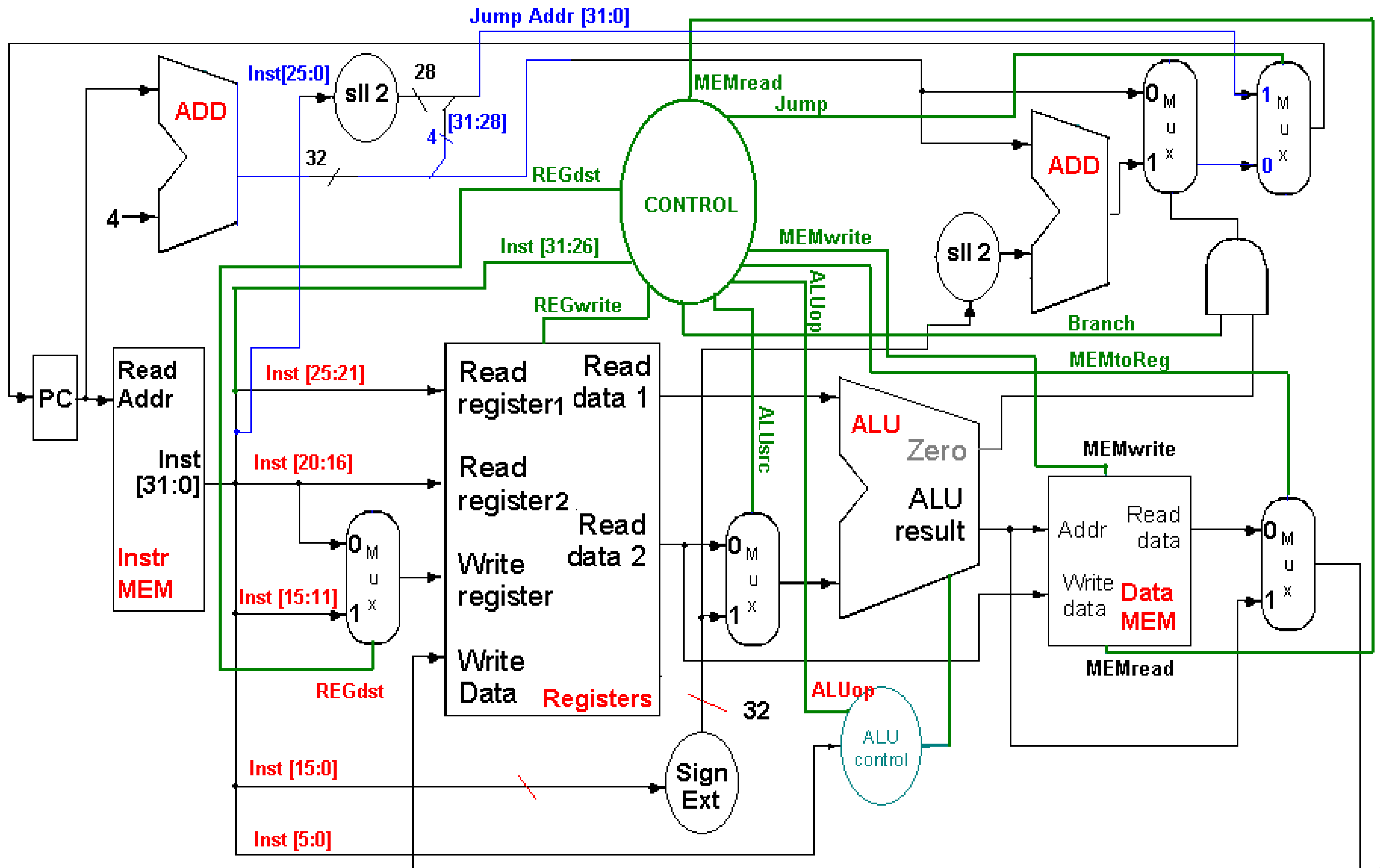
# Load Word Flow



# Branch Instruction Flow



# Jump Instruction Flow



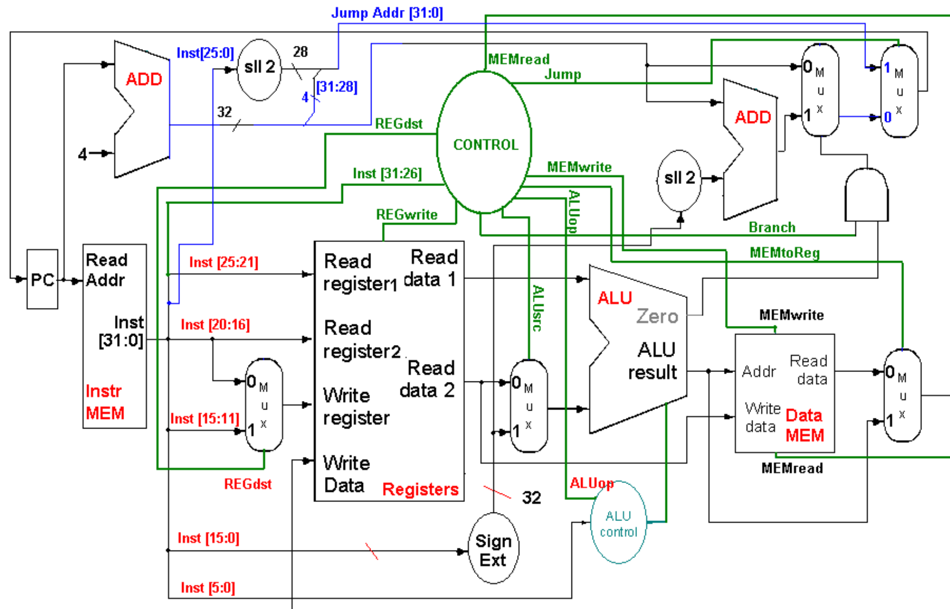
# Single Cycle Implementation--Why Not Used

- So far our implementation is single-cycle design
- Why not used in modern design? Inefficiency!
  - Clock cycle length must be the same for every instruction
  - CPI =1 for every instruction
  - Clock cycle is determined by the longest possible path in the machine
  - Load instruction uses five functional units in series
    - **Instruction Memory**
    - **Register File**
    - **ALU**
    - **Data Memory**
    - **Register File**
  - Several other instruction classes could run in a shorter cycle.
- Each functional unit can be used only once per clock ---some functional units must be duplicated. ---Hardware cost!

# Performance of Single Cycle Machine

- Performance comparison of two approaches for a program run
  - A: Every instruction operates in 1 clock cycle of fixed length
  - B: Every instruction executes in 1 clock cycle using a variable-length clock (only as long as it needs to be)
- Conditions
  - Operational times
    - memory (200ps)
    - ALU and adders (100ps)
    - Register file access (50ps)
    - All others (Negligible)
  - A program has the following instruction mix:
    - load (25%)
    - store (10%)
    - ALU instructions (45%)
    - Branch (15%)
    - Jump (5%)

# Performance Comparison




| Execution | Instruction Fetch | Register Read | ALU Opr | Memory Access | Register Write |
|---|---|---|---|---|---|
| Unit | Instruction Memory | Register File | ALU | Data Memory | Register File |
| R-type | 200 | 50 | 100 | 0 | 50 |
| Load Word | 200 | 50 | 100 | 200 | 50 |
| Store Word | 200 | 50 | 100 | 200 | |
| Branch | 200 | 50 | 100 | 0 | |
| Jump | 200 | | | | |

# Comparison, Problem, and an Alternative

- Approach A:
  - single clock cycle time= 600ps
  - Total execution time = 600px 1.0 = 600 ps
- Approach B:
  - Variable clock cycle times
  - Total execution time

    = 600x0.25 + 550x0.1 + 400x0.45 + 350x0.15 + 200x0.5

    = 447.5 ps
- Performance Comparison
  - 600/447.5=1.34
- Approach B is faster, but
- Implementation of variable clock cycle length for every instruction is extremely difficult
- Solution: Multi-cycle Implementation
  - shorter clock cycle for less work class
  - number of clock cycles for different class