

**CPS104**  
**Computer Organization and programming**  
**Lecture 13: Designing a Single Cycle Datapath**

**Oct 18, 1999**

**Dietolf (Dee) Ramm**

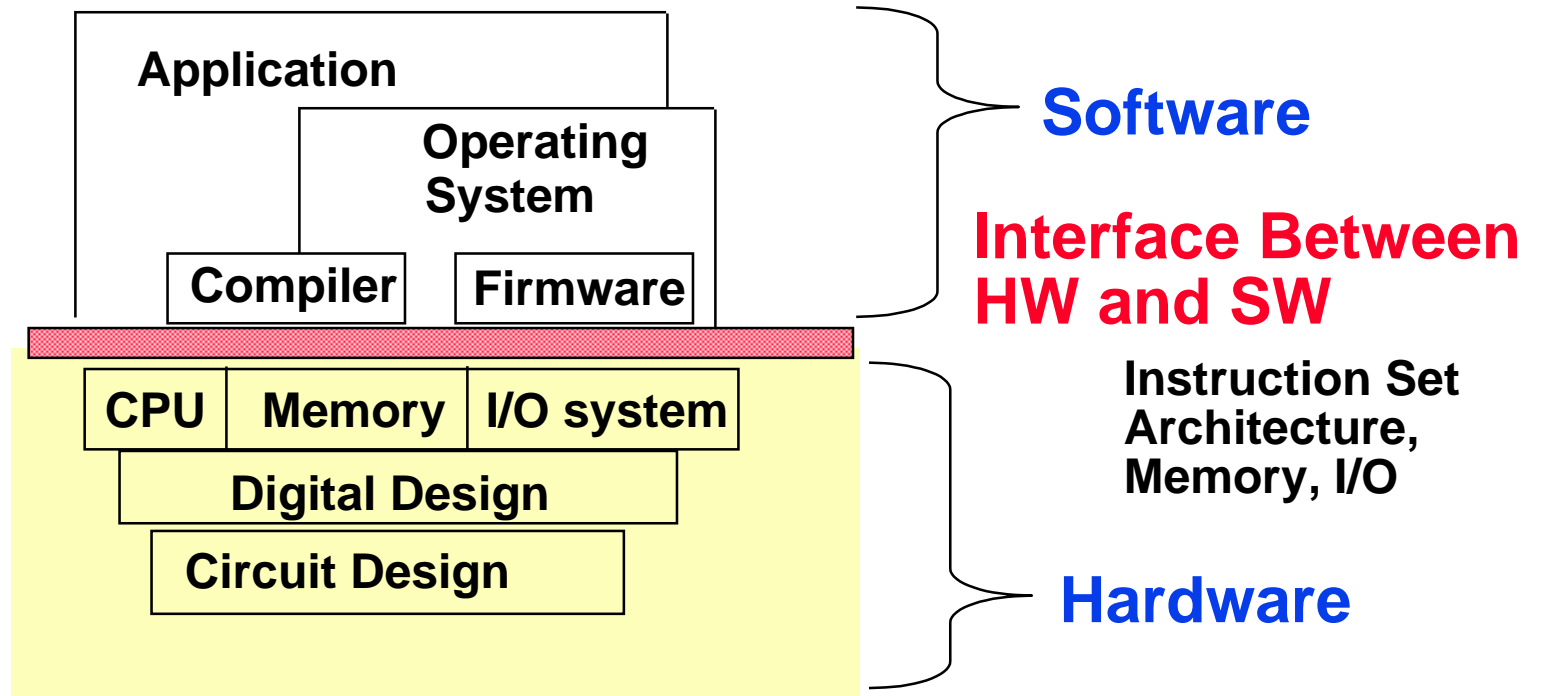
**<http://www.cs.duke.edu/~dr/cps104.html>**

# Outline of Today's Lecture

- **Where are we with respect to the BIG picture?**
- **The Steps of Designing a Processor**
- **Datapath components.**
- **Datapath and timing for Reg-Reg Operations**
- **Datapath for Logical Operations with Immediate**
- **Datapath for Load and Store Operations**
- **Datapath for Branch and Jump Operations**

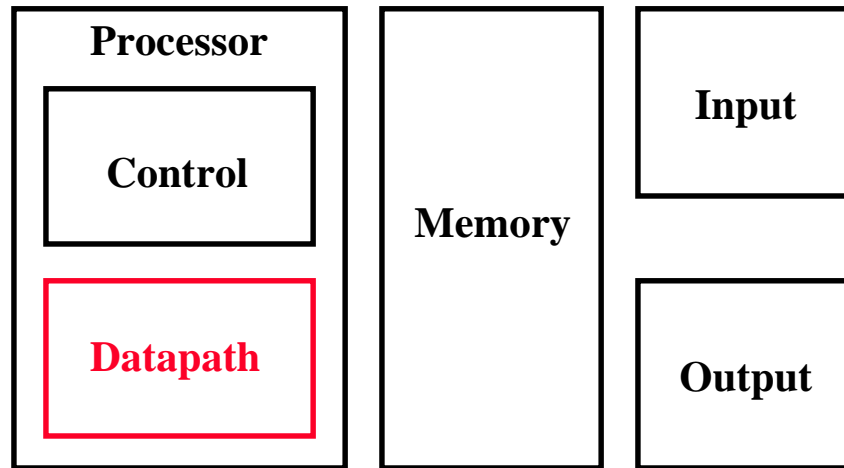
# What is Computer Architecture?

- Coordination of levels of abstraction



# The Big Picture: Where are We Now?

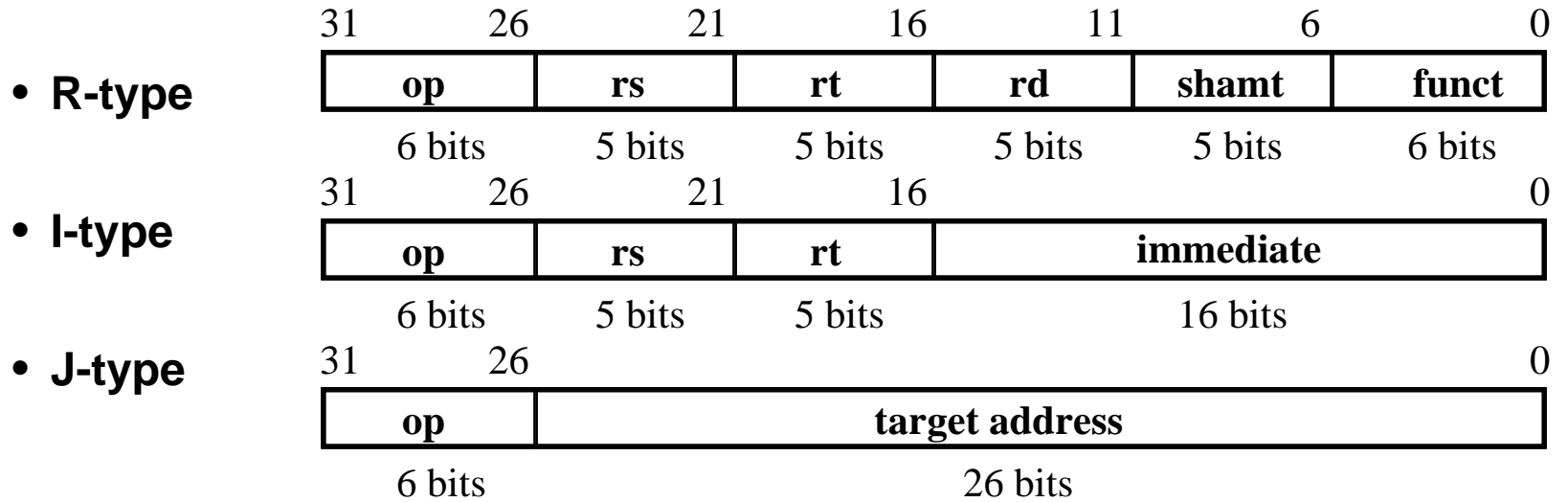
- The Five Classic Components of a Computer



**Today's Topic: Datapath Design**

# The MIPS Instruction Formats

◦ All MIPS instructions are 32 bits long. The three instruction formats:



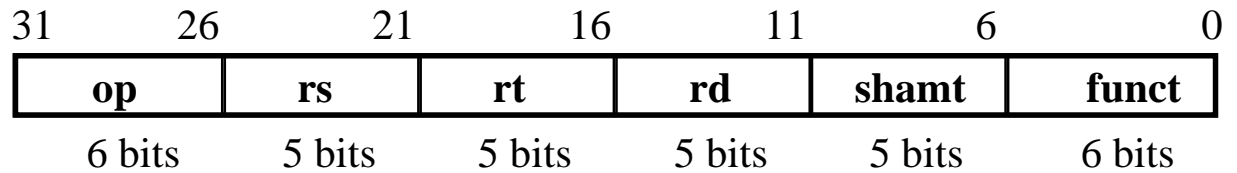
◦ The different fields are:

- **op**: operation of the instruction
- **rs, rt, rd**: the source and destination register specifiers
- **shamt**: shift amount
- **funct**: selects the variant of the operation in the “op” field
- **address / immediate**: address offset or immediate value
- **target address**: target address of the jump instruction

## The MIPS Subset (We can't do them all!)

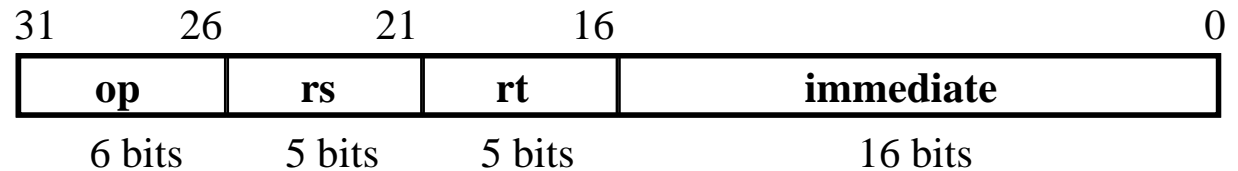
- **ADD and subtract**

- add rd, rs, rt
- sub rd, rs, rt



- **OR Immediate:**

- ori rt, rs, imm16



- **LOAD and STORE**

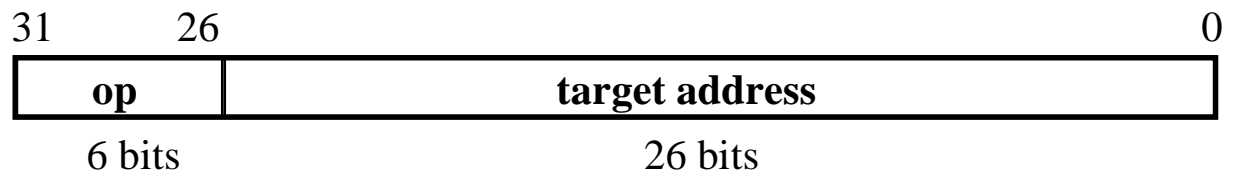
- lw rt, rs, imm16
- sw rt, rs, imm16

- **BRANCH:**

- beq rs, rt, imm16

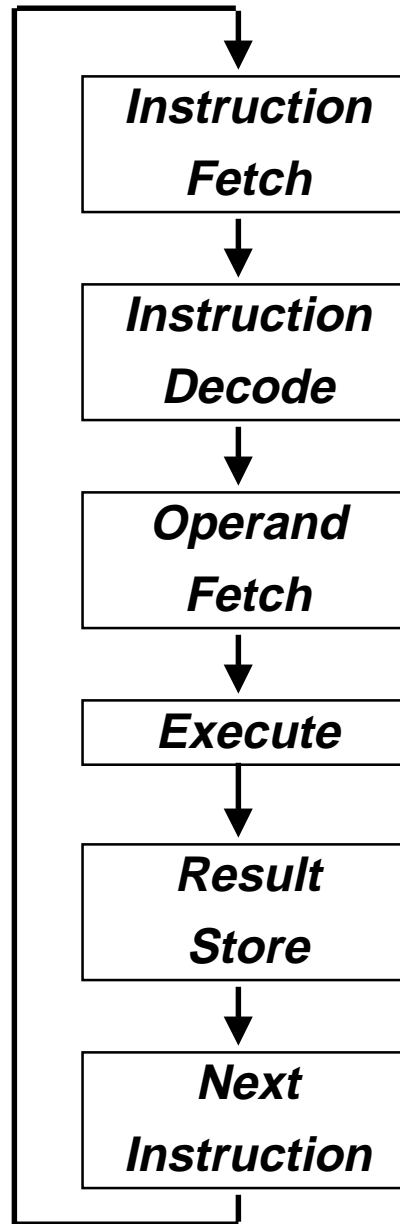
- **JUMP:**

- j target

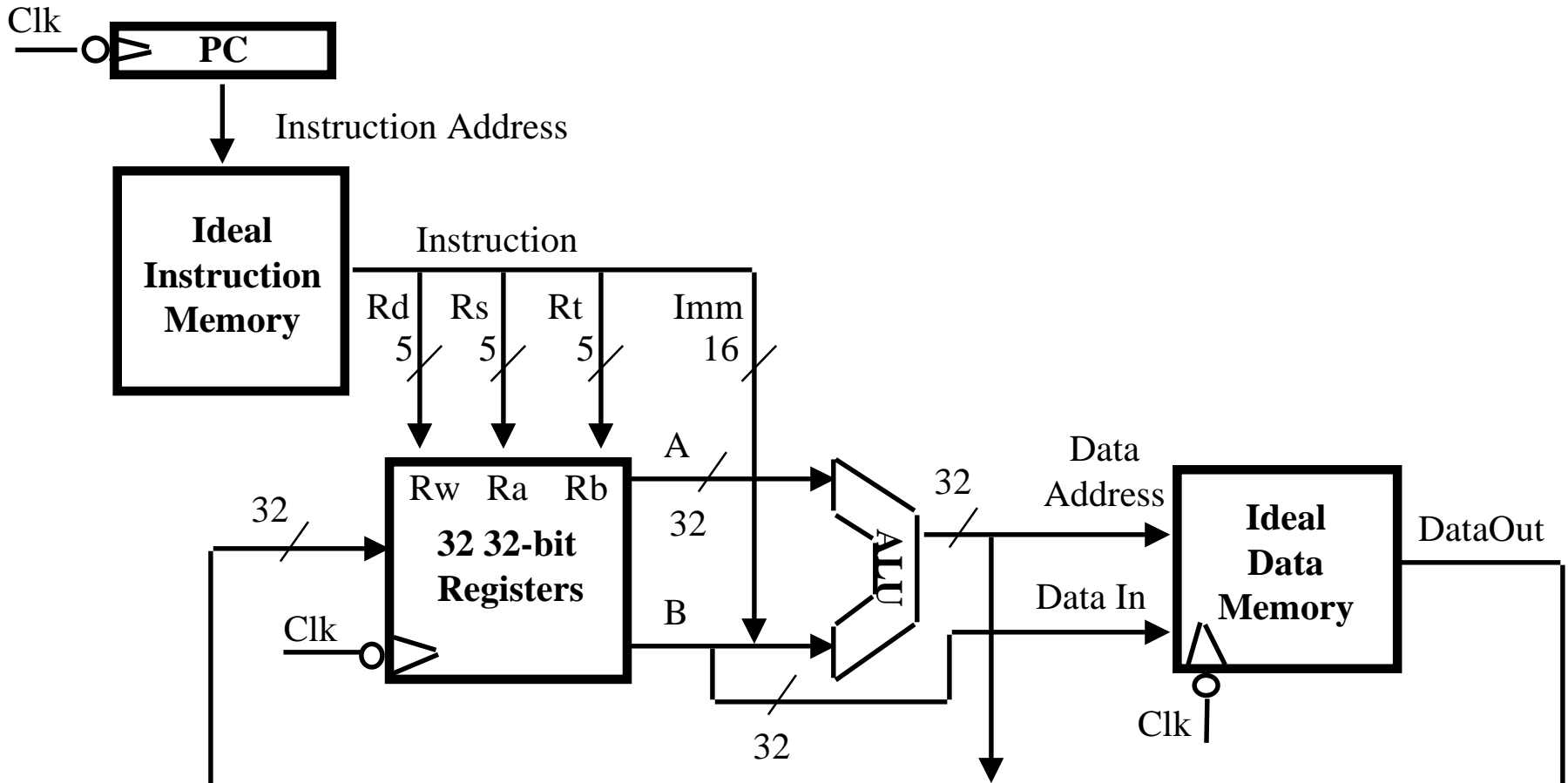


# The Hardware “Program”

How does one build hardware that implements the MIPS instructions?

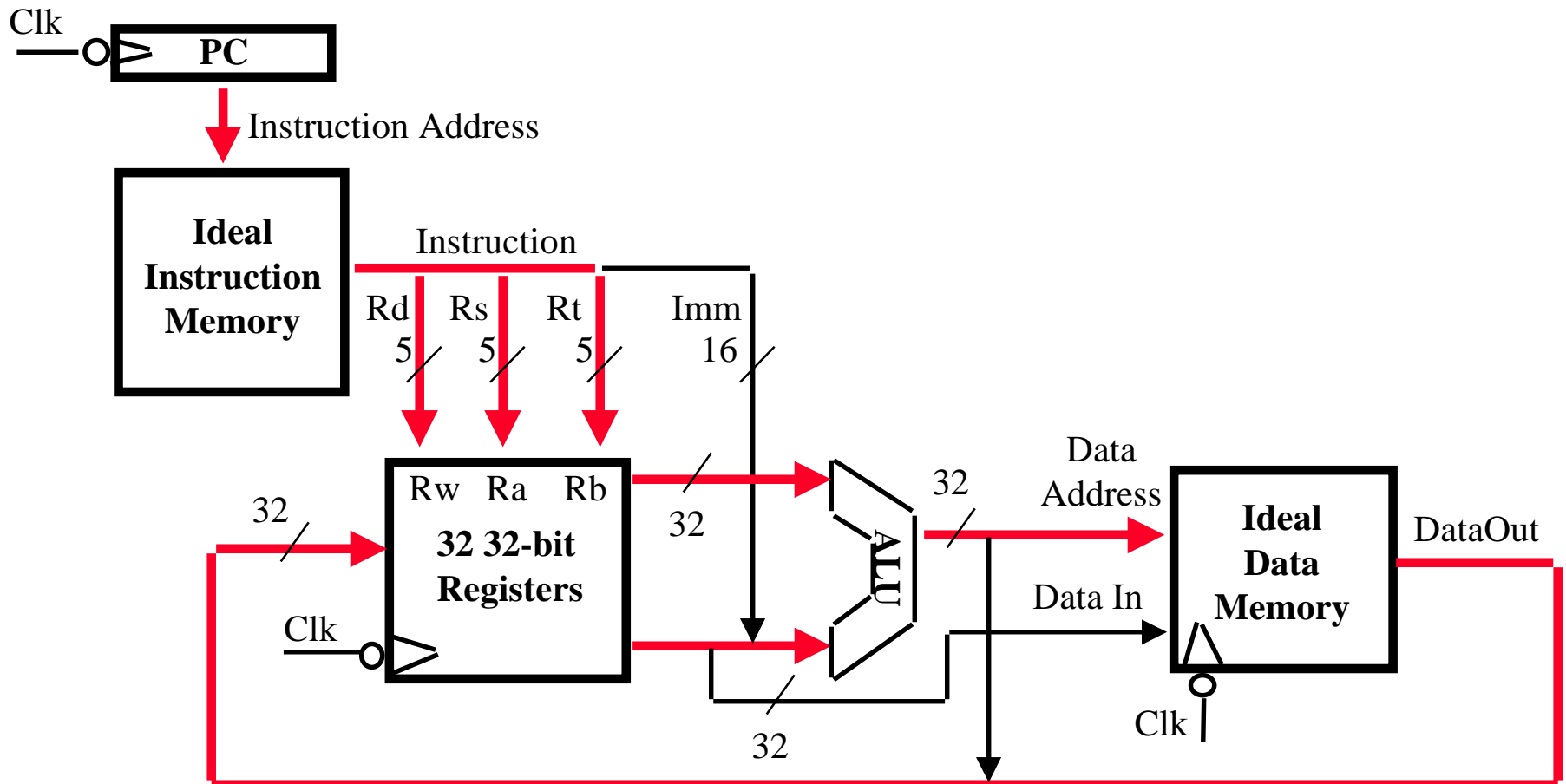


# An Abstract View of the Implementation



# An Abstract View of the Critical Path

- Register file and ideal memory:
  - The CLK input is a factor **ONLY** during write operation
  - During read operation, behave as combinational logic:
    - Address valid => Output valid after “access time.”



# The Steps of Designing a Processor

- **Instruction Set Architecture => Register Transfer Language**
- **Register Transfer Language =>**
  - **Datapath components**
  - **Datapath interconnect**
- **Datapath components => Control signals**
- **Control signals => Control logic**

# RTL: The ADD Instruction

◦ **add rd, rs, rt**

- **mem[PC]**                      **Fetch the instruction from memory**
- **$R[rd] \leftarrow R[rs] + R[rt]$**                       **The ADD operation**
- **$PC \leftarrow PC + 4$**                       **Calculate the next instruction's address**

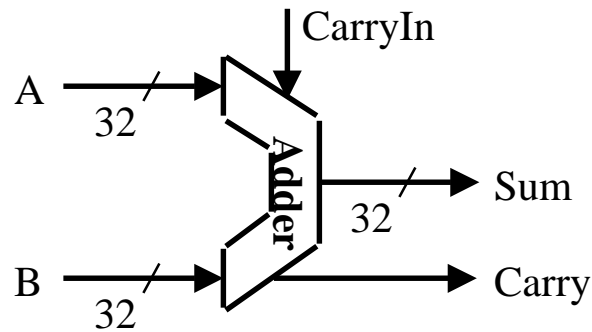
# RTL: The Load Instruction

◦ **lw** **rt, rs, imm16**

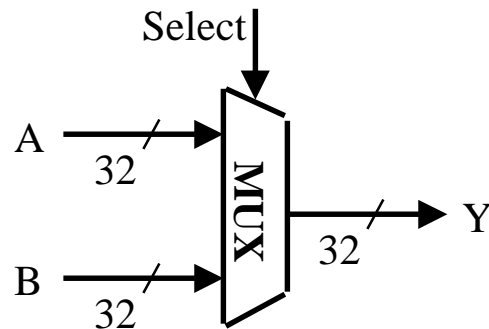
- **mem[PC]** **Fetch the instruction from memory**
- **Addr  $\leftarrow$  R[rs] + SignExt(imm16)** **Calculate the memory address**
- **R[rt]  $\leftarrow$  Mem[Addr]** **Load the data into the register**
- **PC  $\leftarrow$  PC + 4** **Calculate the next instruction's address**

# Combinational Logic Elements (Basic Building Blocks)

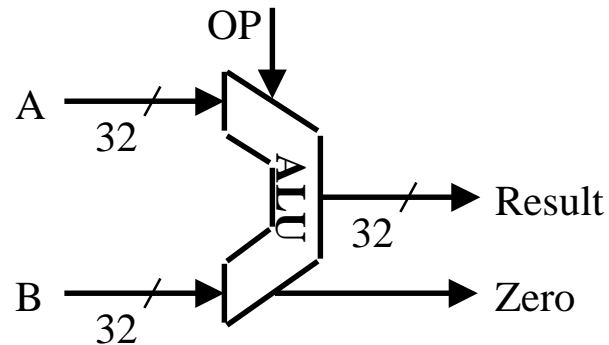
- **Adder**



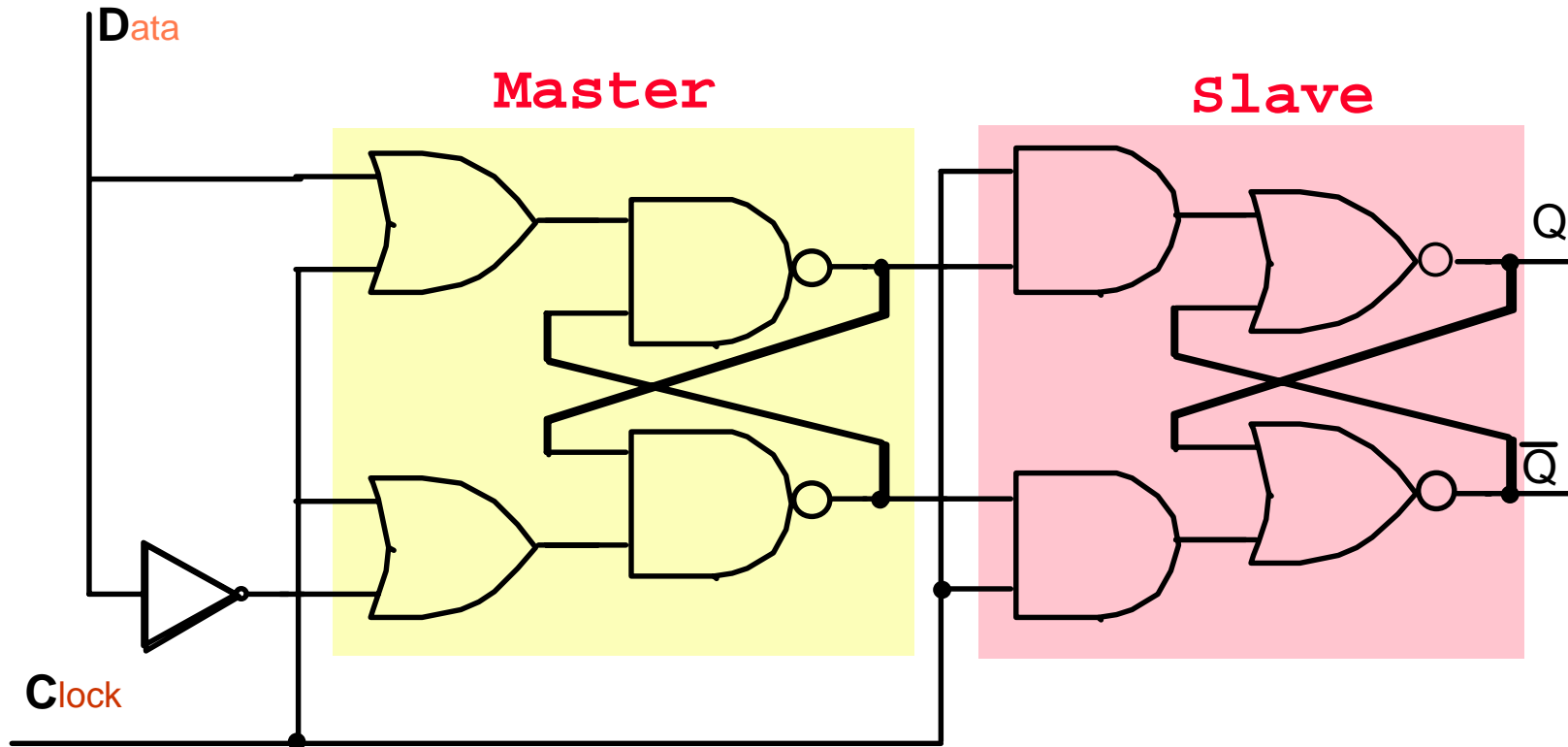
- **MUX**



- **ALU**

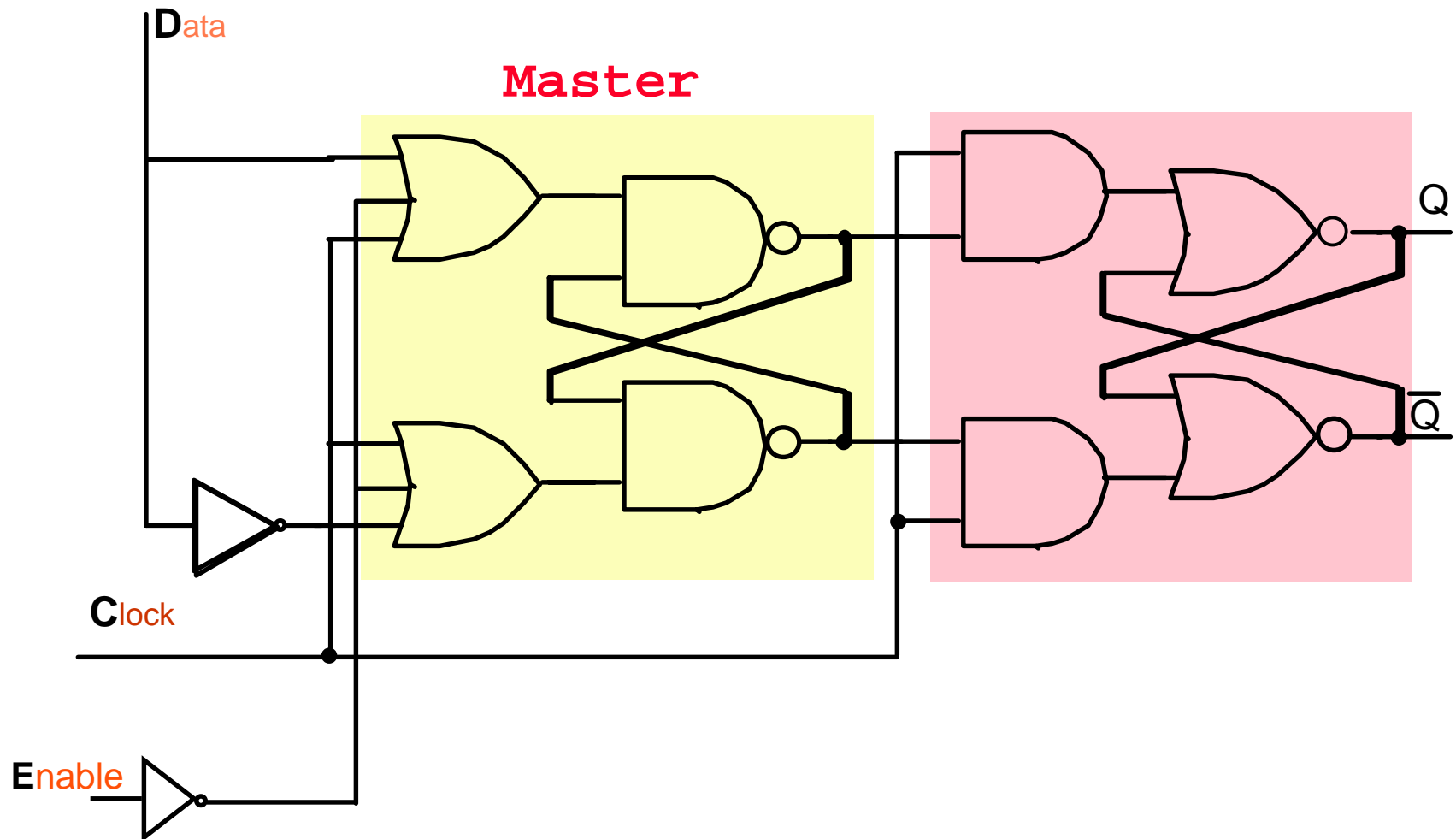


# Master-Slave Data-Flip-Flop



- On C ↓ D is transferred to the master stage and the slave is stable.
- On C ↑ the Master stage is transferred into the slave stage (output), and the master stage is stable.

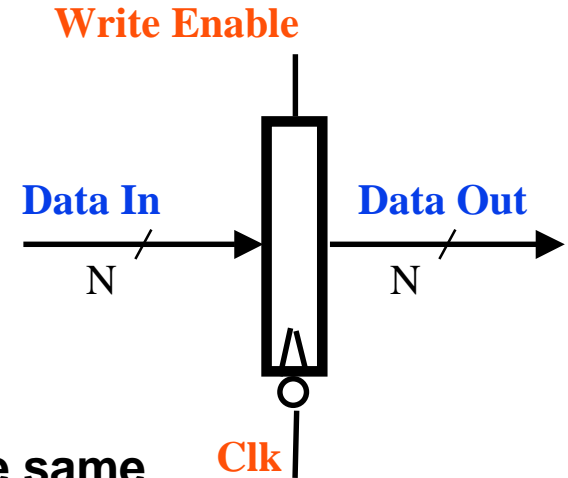
# DFF with Enable



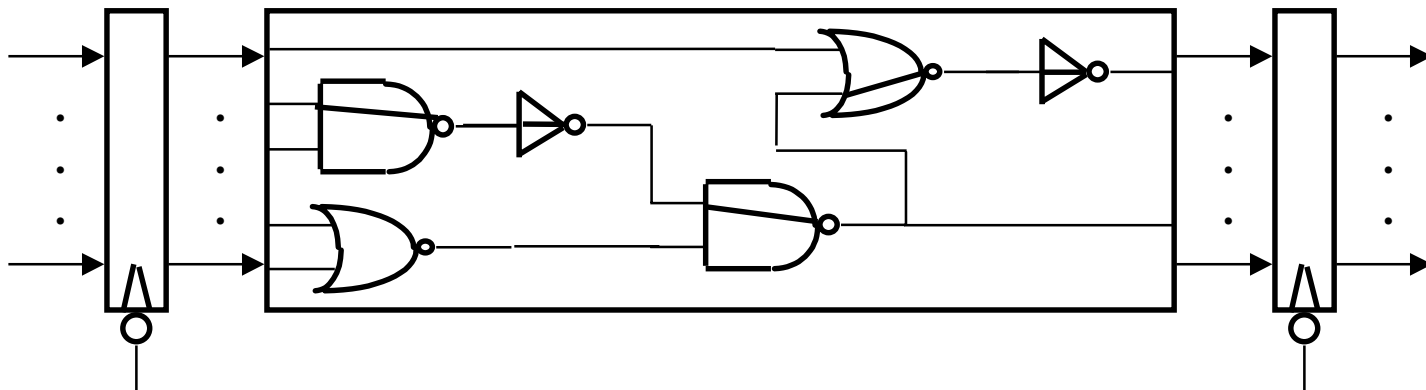
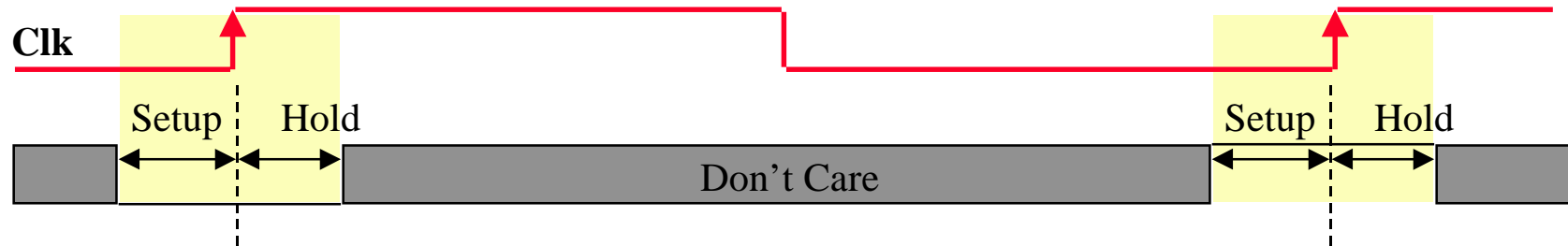
# Storage Element: Register (Basic Building Block)

## ◦ Register

- Similar to the D Flip Flop except
  - N-bit **input** and **output**
  - **Write Enable** input
- **Write Enable:**
  - **negated (0): Data Out** will not change
  - **asserted (1): Data Out** will become the same as **Data In**.



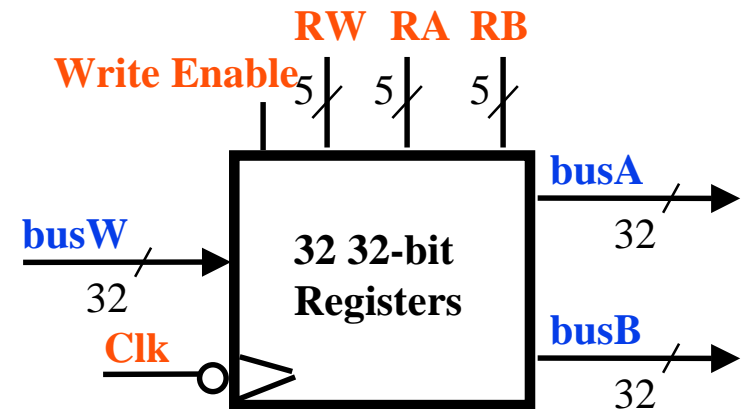
# Clocking Methodology



- All storage elements are clocked by the same clock edge
- Cycle Time  $\geq$  CLK-to-Q + Longest Delay Path + Setup + Clock Skew

# Storage Element: Register File

- 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** selects the register to put on **busA**
  - **RB** selects the register to put on **busB**
  - **RW** selects the register to be written via **busW** when **Write Enable** is 1
- Clock input (CLK)
  - The **CLK** input is a factor **ONLY** during write operation
  - During read operation, behaves as a combinational logic block:
    - **RA** or **RB** valid => **busA** or **busB** valid after “access time.”



# Storage Element: Idealized Memory

- Memory (idealized)

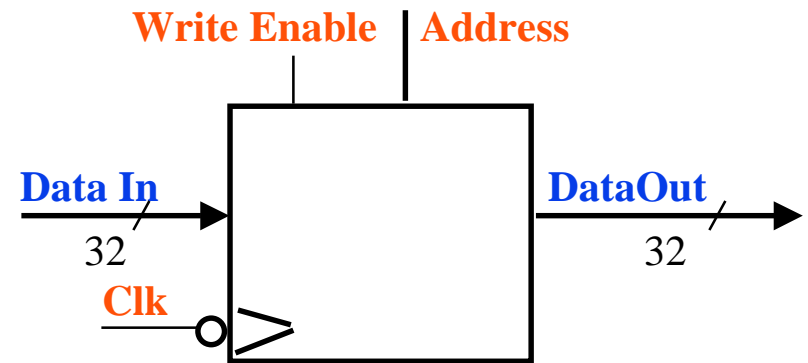
- One input bus: **Data In**
- One output bus: **Data Out**

- Memory word is selected by:

- **Write Enable** = 0: **Address** selects the word to put on the **Data Out** bus
- **Write Enable** = 1: **Address** selects the memory word to be written via the **Data In** bus

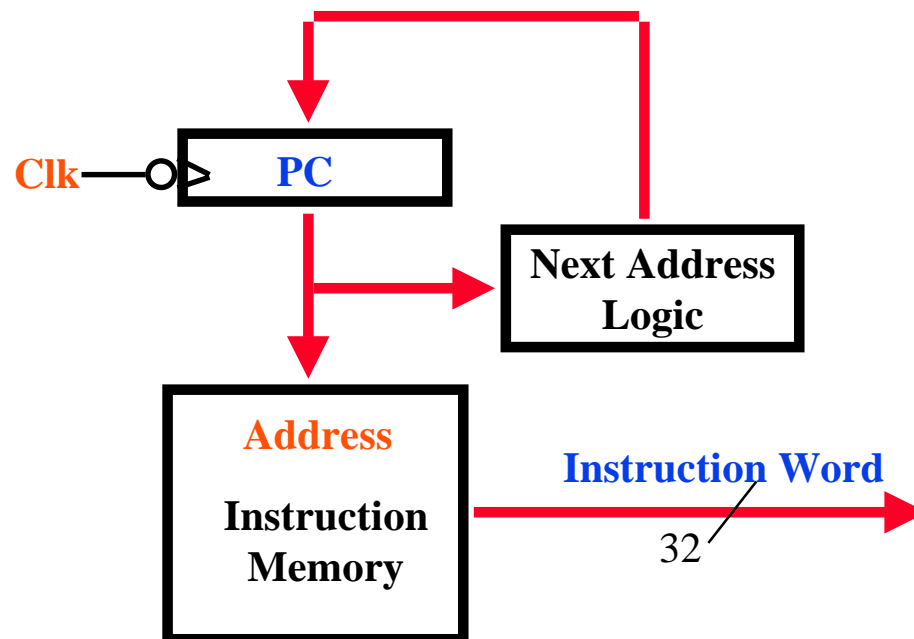
- Clock input (CLK)

- The **CLK** input is a factor **ONLY** during write operation
- During read operation, behaves as a combinational logic block:
  - **Address** valid => **Data Out** valid after “access time.”



# Overview of the Instruction Fetch Unit

- The common RTL operations
  - Fetch the Instruction:  $\text{mem}[\text{PC}]$
  - Update the program counter:
    - Sequential Code:  $\text{PC} \leftarrow \text{PC} + 4$
    - Branch and Jump:  $\text{PC} \leftarrow$  “something else”





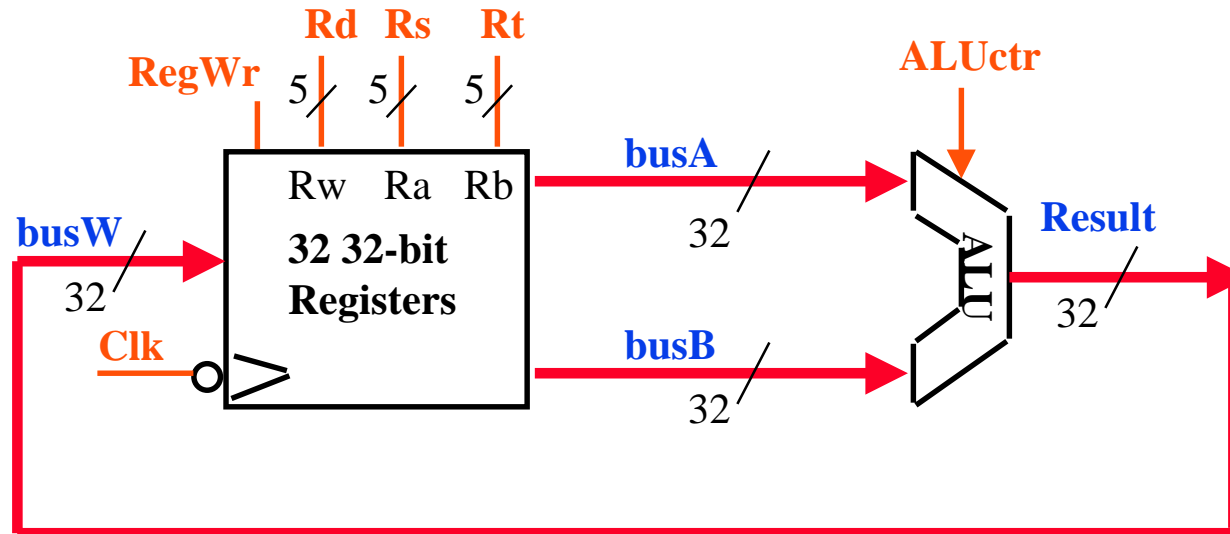
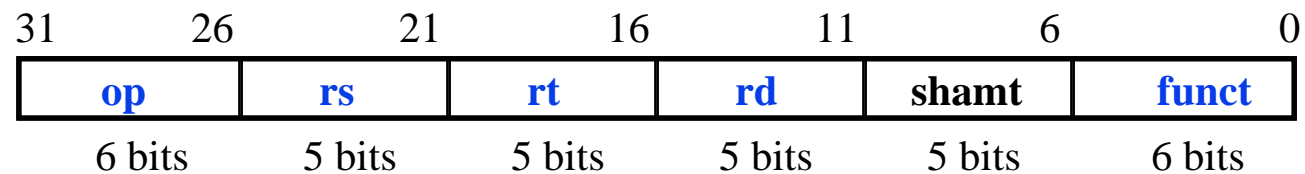


# Datapath for Register-Register Operations

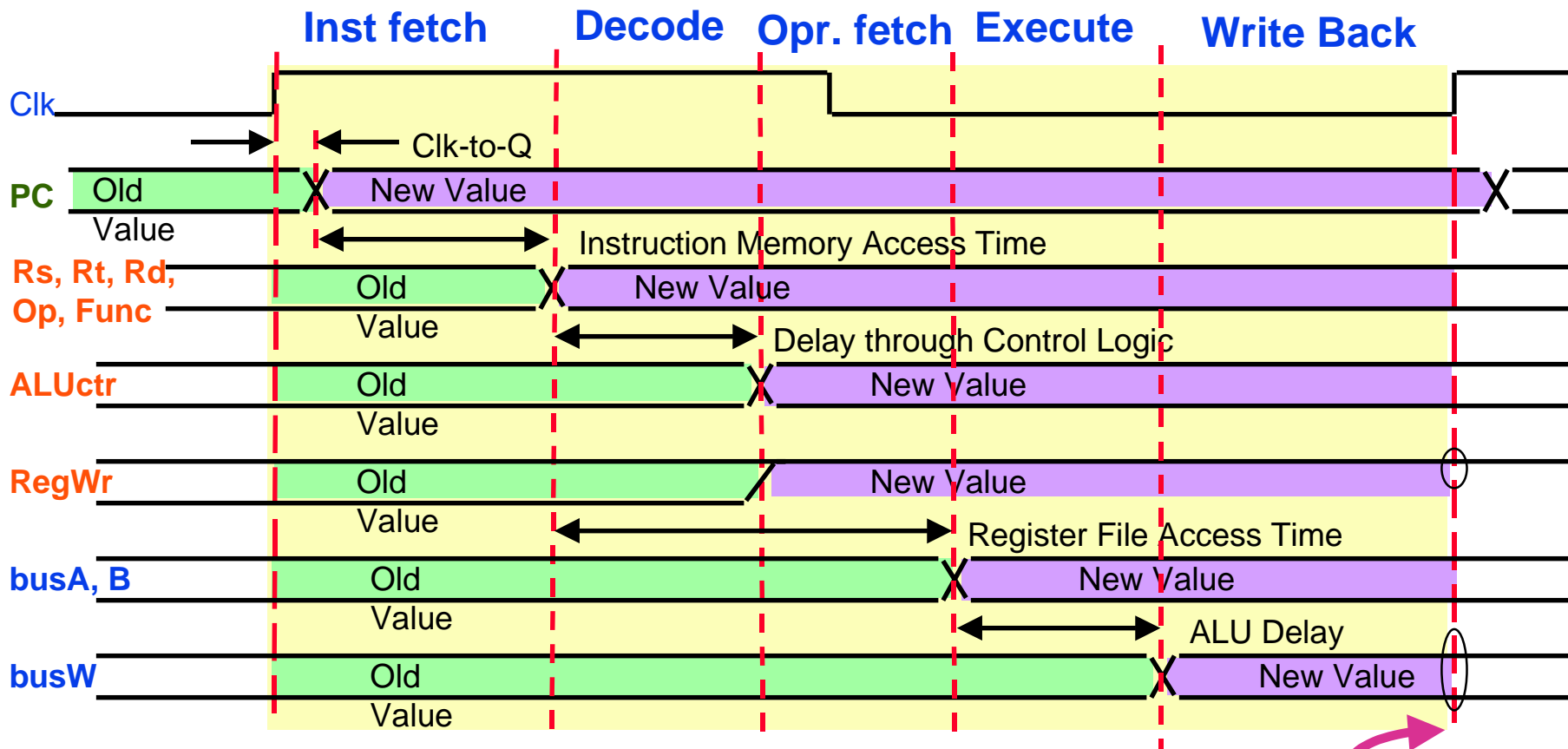
◦  $R[rd] \leftarrow R[rs] \text{ op } R[rt]$

Example: `add rd, rs, rt`

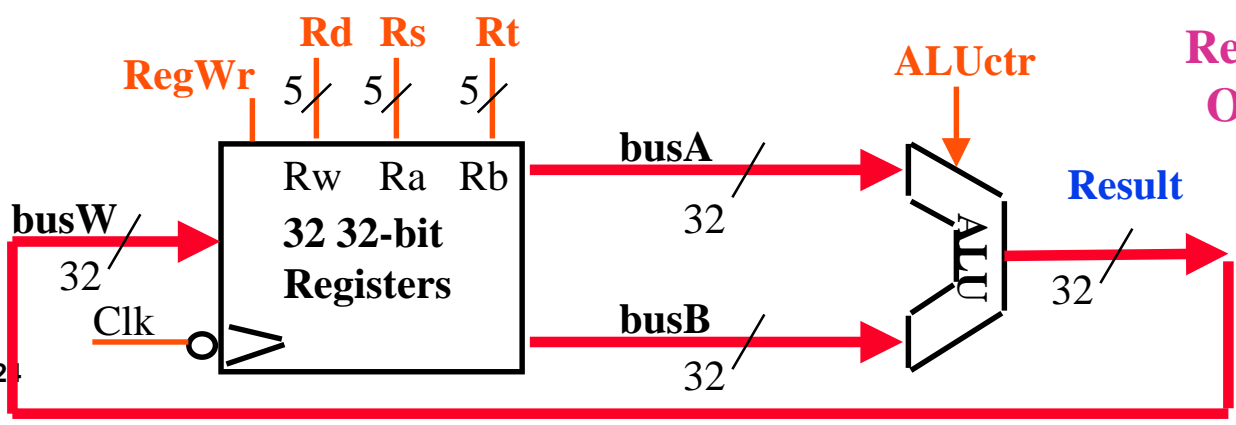
- **Ra**, **Rb**, and **Rw** comes from instruction's **rs**, **rt**, and **rd** fields
- **ALUctr** and **RegWr**: control logic after decoding the instruction fields: **op** and **func**



# Register-Register Timing



Register Write Occurs Here

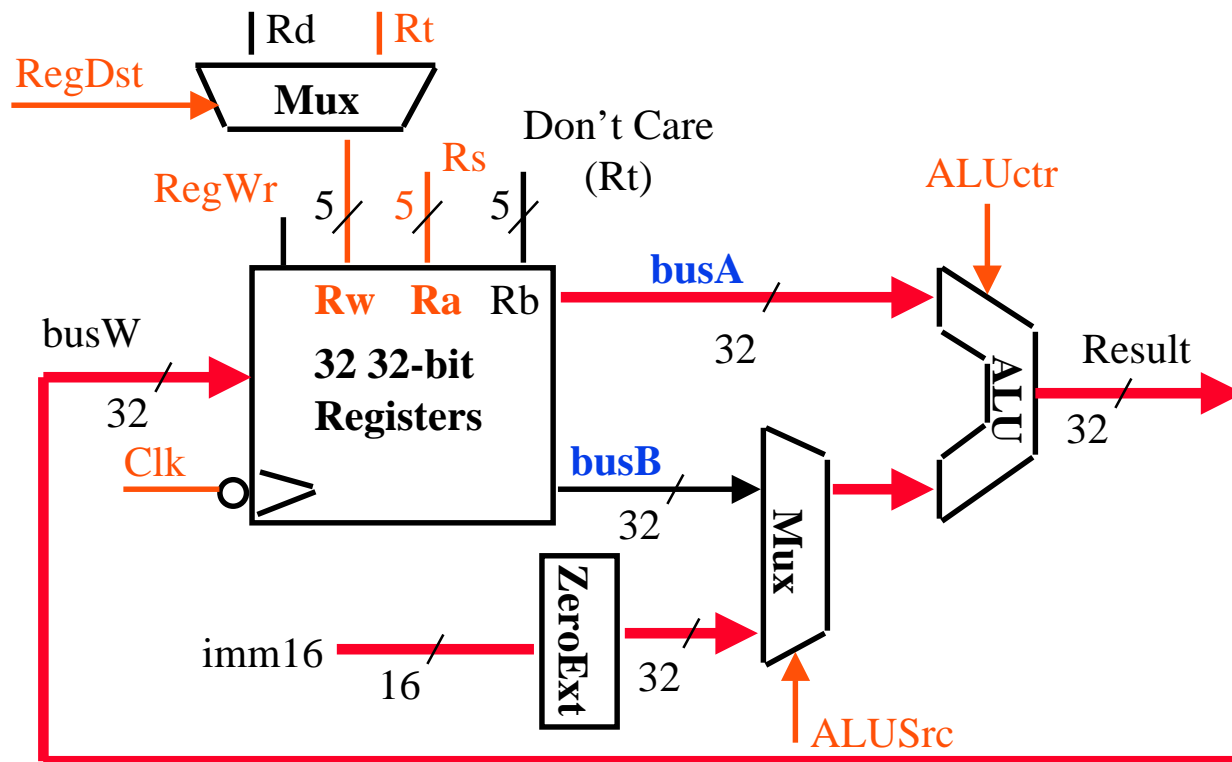
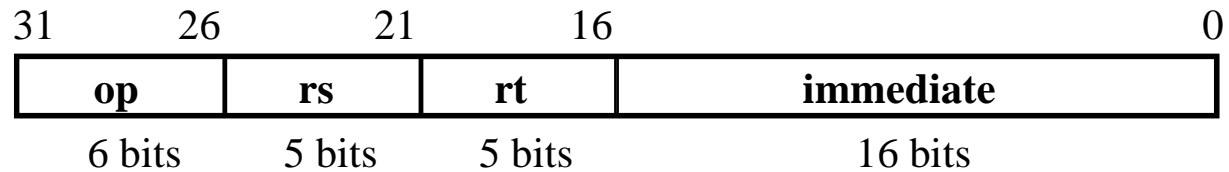




# Datapath for Logical Operations with Immediate

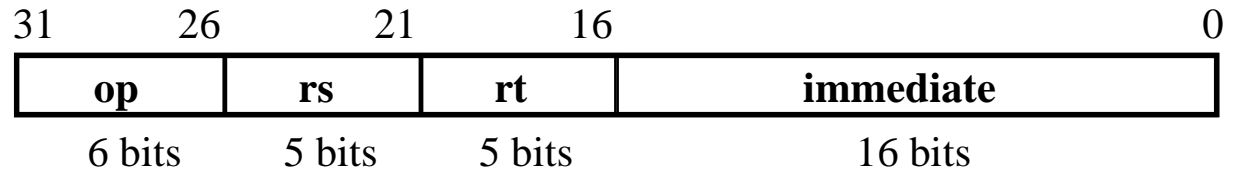
°  $R[rt] \leftarrow R[rs] \text{ op ZeroExt}[imm16]$

Example: `ori rt, rs, imm16`

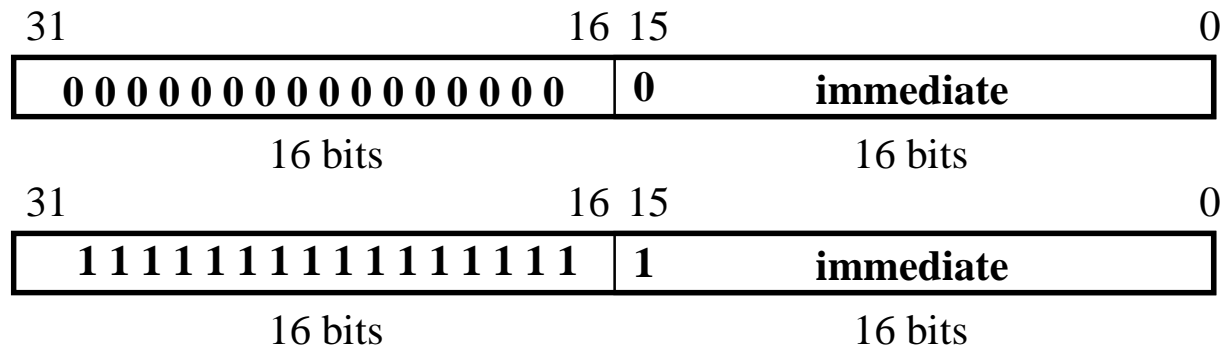


# RTL: The Load Instruction

◦ lw rt, rs, imm16



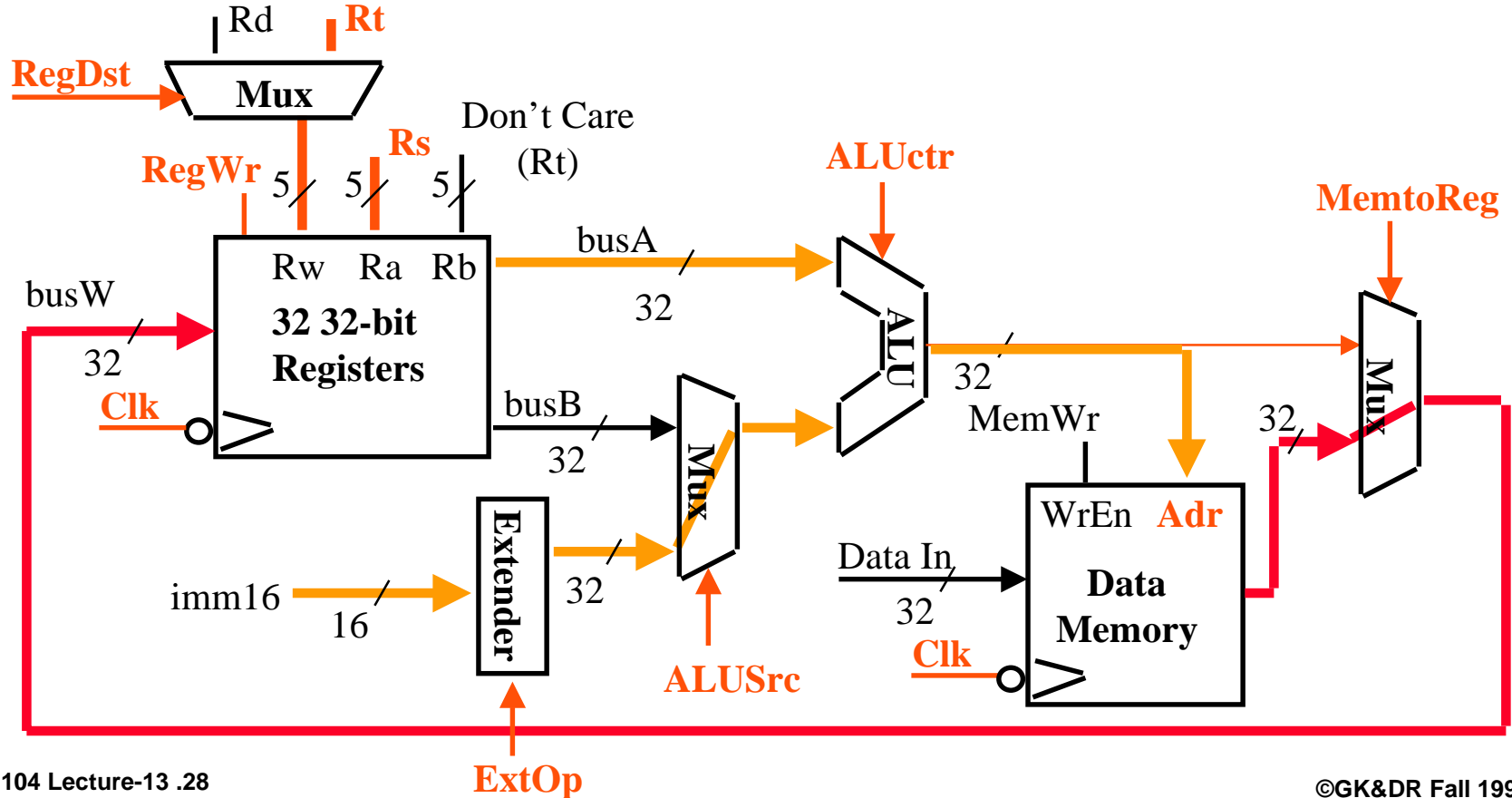
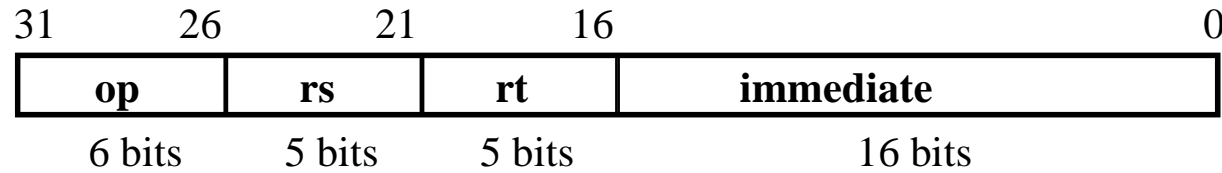
- mem[PC] Fetch the instruction from memory
  
- Addr ← R[rs] + SignExt(imm16) Calculate the memory address  
R[rt] ← Mem[Addr] Load the data into the register
  
- PC ← PC + 4 Calculate the next instruction's address



# Datapath for Load Operations

◦  $R[rt] \leftarrow \text{Mem}[R[rs] + \text{SignExt}[\text{imm16}]]$

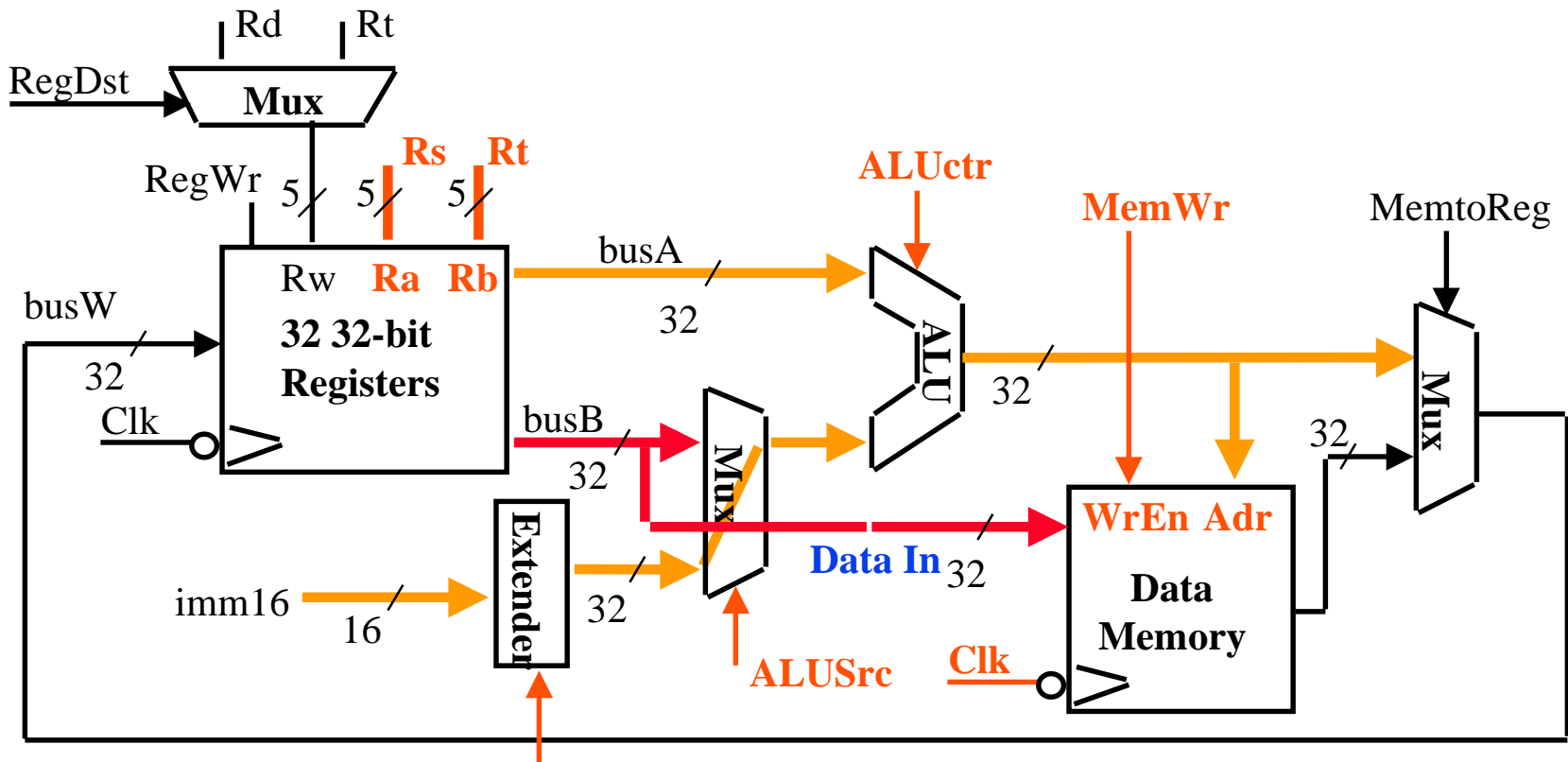
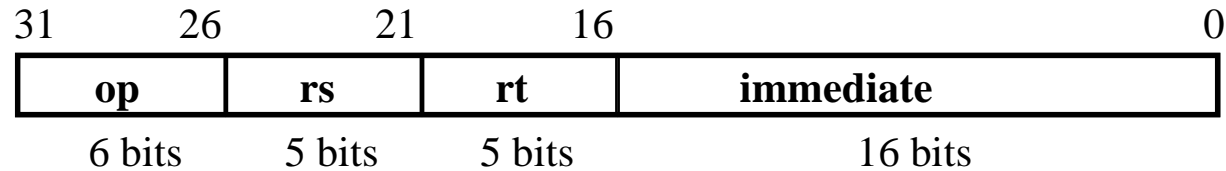
Example: **lw** *rt, rs, imm16*





# Datapath for Store Operations

- $\text{Mem}[\text{R}[\text{rs}] + \text{SignExt}[\text{imm16}] \leftarrow \text{R}[\text{rt}]]$     Example: **sw**    **rt, rs, imm16**

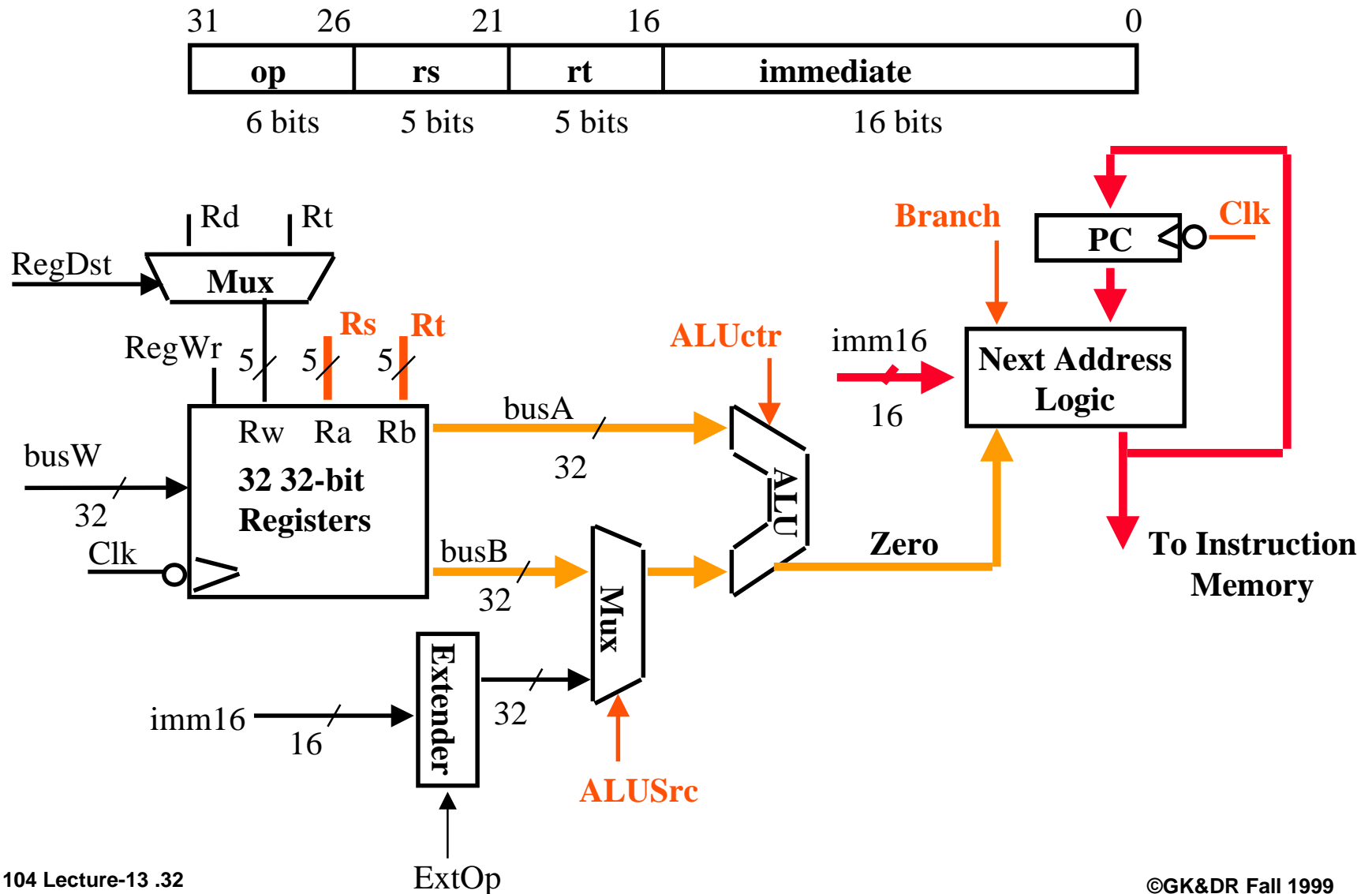




# Datapath for Branch Operations

° **beq rs, rt, imm16**

We need to compare Rs and Rt!

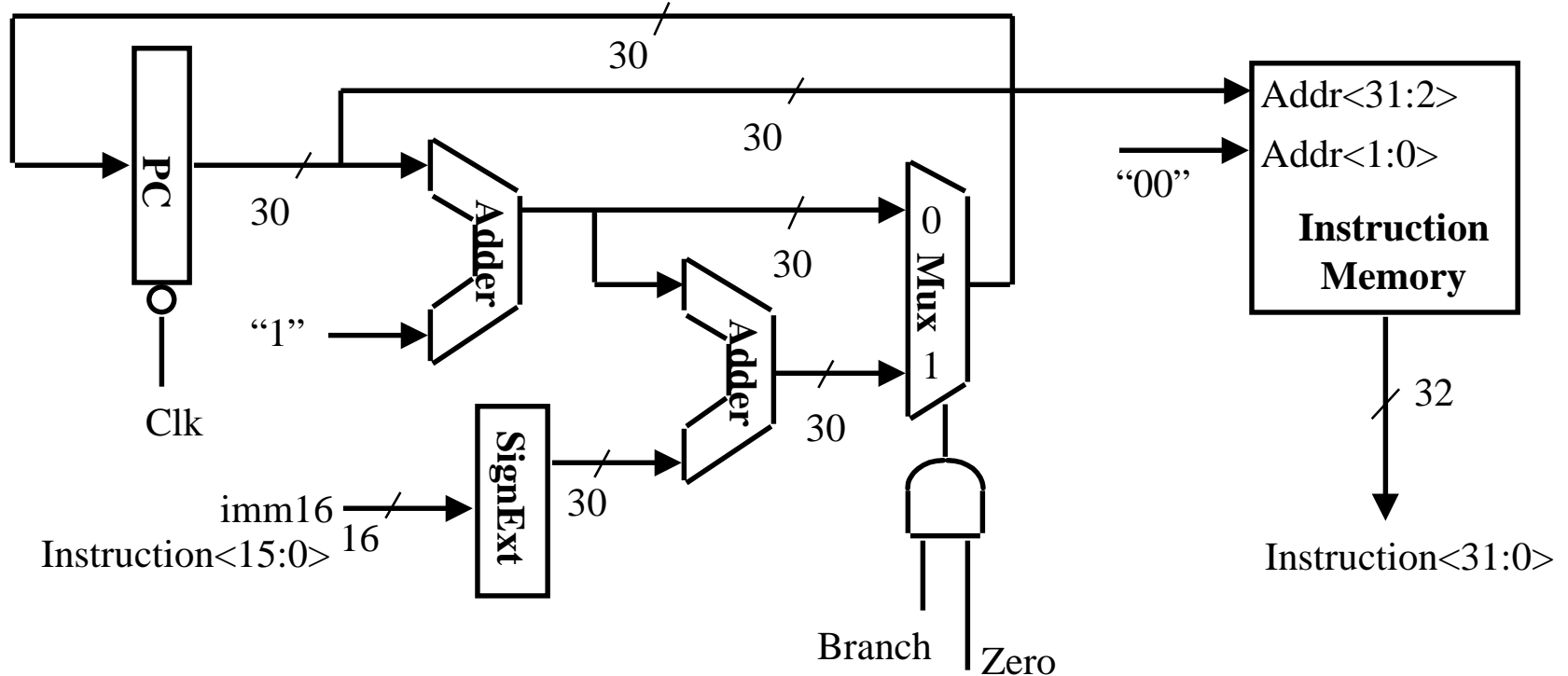


# Binary Arithmetic for the Next Address

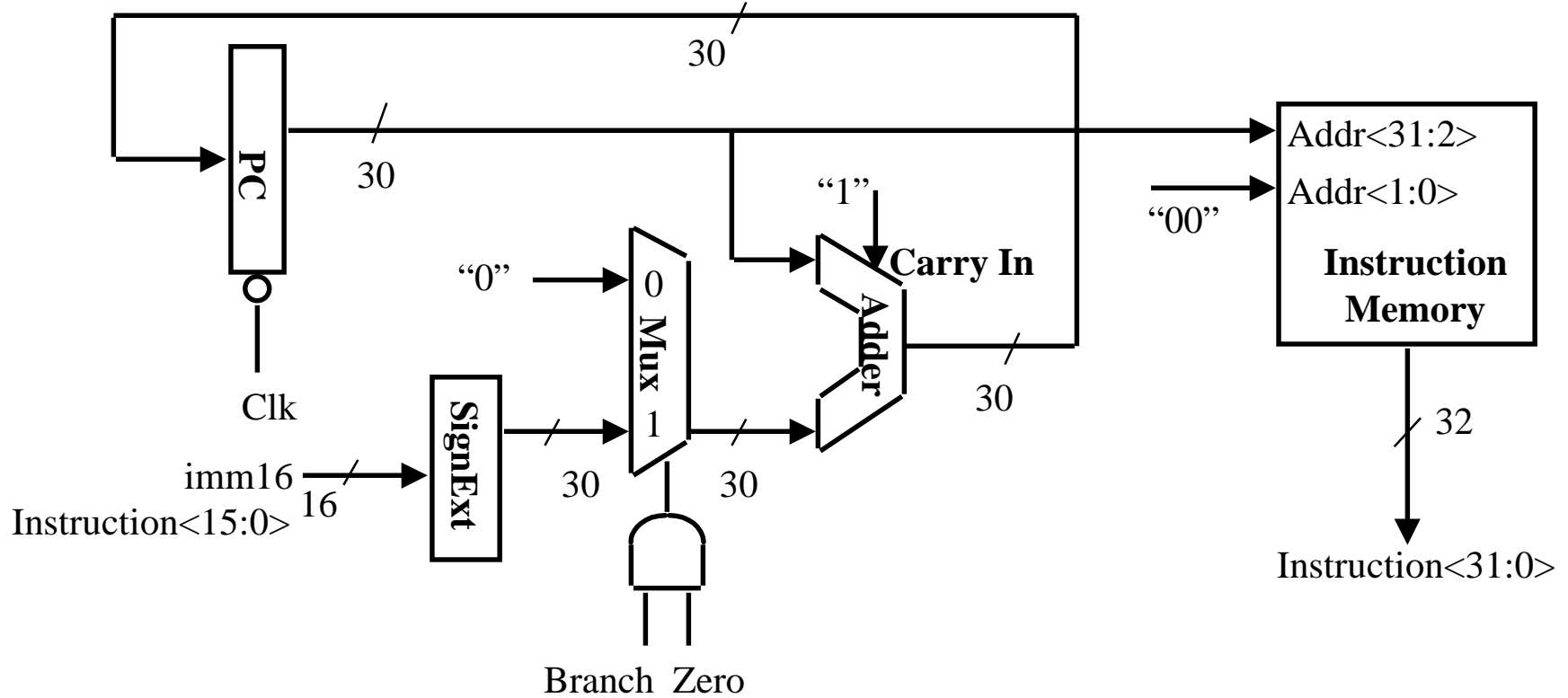
- In theory, the PC is a 32-bit byte address into the instruction memory:
  - Sequential operation:  $PC\langle 31:0 \rangle = PC\langle 31:0 \rangle + 4$
  - Branch operation:  $PC\langle 31:0 \rangle = PC\langle 31:0 \rangle + 4 + \text{SignExt}[\text{Imm16}] * 4$
- The magic number “4” always comes up because:
  - The 32-bit PC is a byte address
  - And all our instructions are 4 bytes (32 bits) long
- In other words:
  - The 2 LSBs of the 32-bit PC are always zeros
  - There is no reason to have hardware to keep the 2 LSBs
- In practice, we can simplify the hardware by using a 30-bit  $PC\langle 31:2 \rangle$ :
  - Sequential operation:  $PC\langle 31:2 \rangle = PC\langle 31:2 \rangle + 1$
  - Branch operation:  $PC\langle 31:2 \rangle = PC\langle 31:2 \rangle + 1 + \text{SignExt}[\text{Imm16}]$
  - In either case:  $\text{Instruction-Memory-Address} = PC\langle 31:2 \rangle \text{ concat } \text{“00”}$

# Next Address Logic: Expensive and Fast Solution

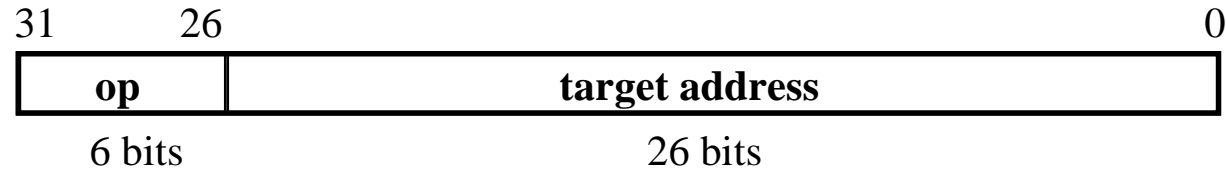
- Using a 30-bit PC:
  - Sequential operation:  $PC\langle 31:2 \rangle = PC\langle 31:2 \rangle + 1$
  - Branch operation:  $PC\langle 31:2 \rangle = PC\langle 31:2 \rangle + 1 + \text{SignExt}[\text{Imm16}]$
  - In either case:  $\text{Instruction-Memory-Address} = PC\langle 31:2 \rangle \text{ concat } "00"$



# Next Address Logic



# RTL: The Jump Instruction



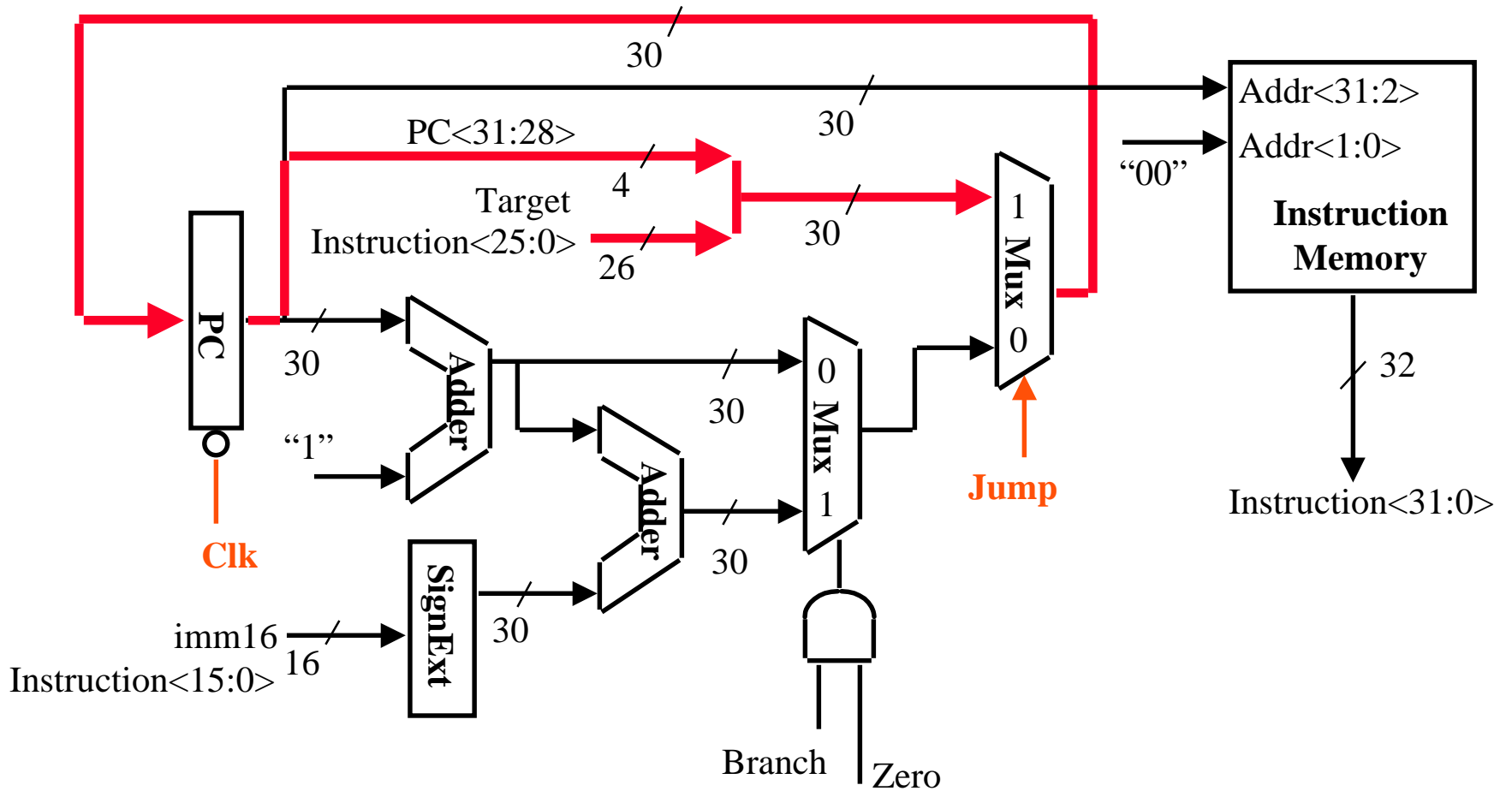
◦ **j        target**

- **mem[PC]**    **Fetch the instruction from memory**
- **PC<31:2> <- PC<31:28> concat target<26:0> concat <00>**  
**Calculate the next instruction's address**

# Instruction Fetch Unit

◦ **j target**

- $PC\langle 31:2 \rangle \leftarrow PC\langle 31:28 \rangle \text{ concat target}\langle 25:0 \rangle$



# Putting it All Together: A Single Cycle Datapath

- We have everything except **control signals**.

