

# **CPS104 Computer Organization**

## **Lecture 16**

### **Memory Elements, Datapath Building Blocks**

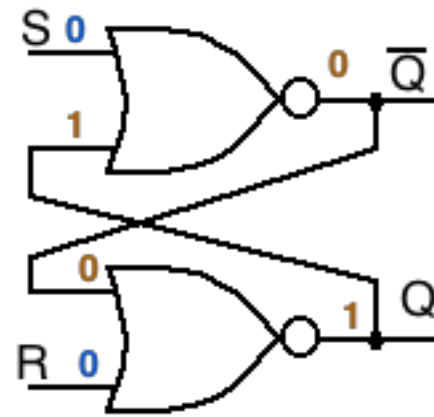
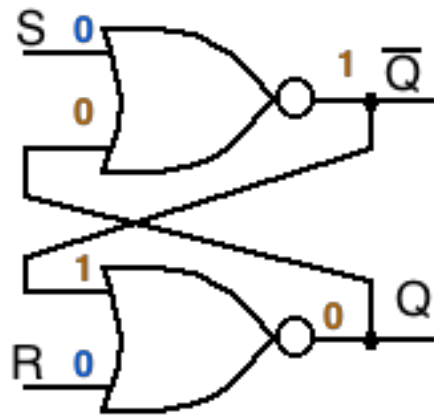
**October 21 , 2009**

**Gershon Kedem**

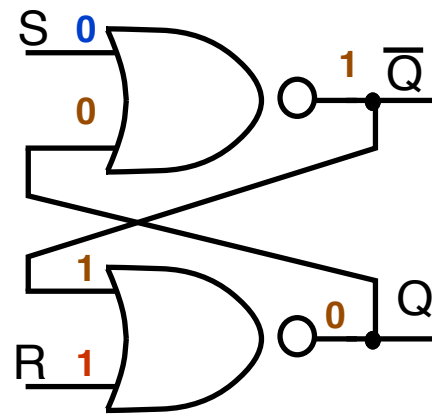
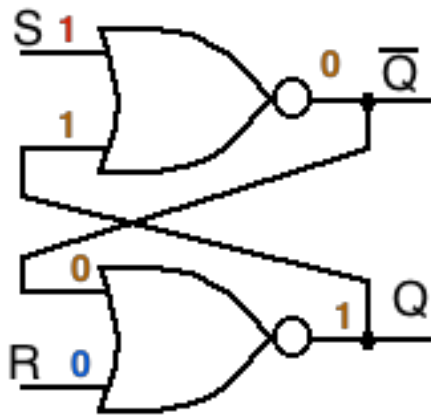
# Administratrivia

- Homework-4 Posted; **Due October 26, in class**
- Extra Credit Homework is posted. **Due: October 26**
- Reading: Chapter 4.
- Reading: Appendix C
  - ◆ Available on: . . . [cps104/Handouts/Appendix-C.pdf](http://cps104/Handouts/Appendix-C.pdf)

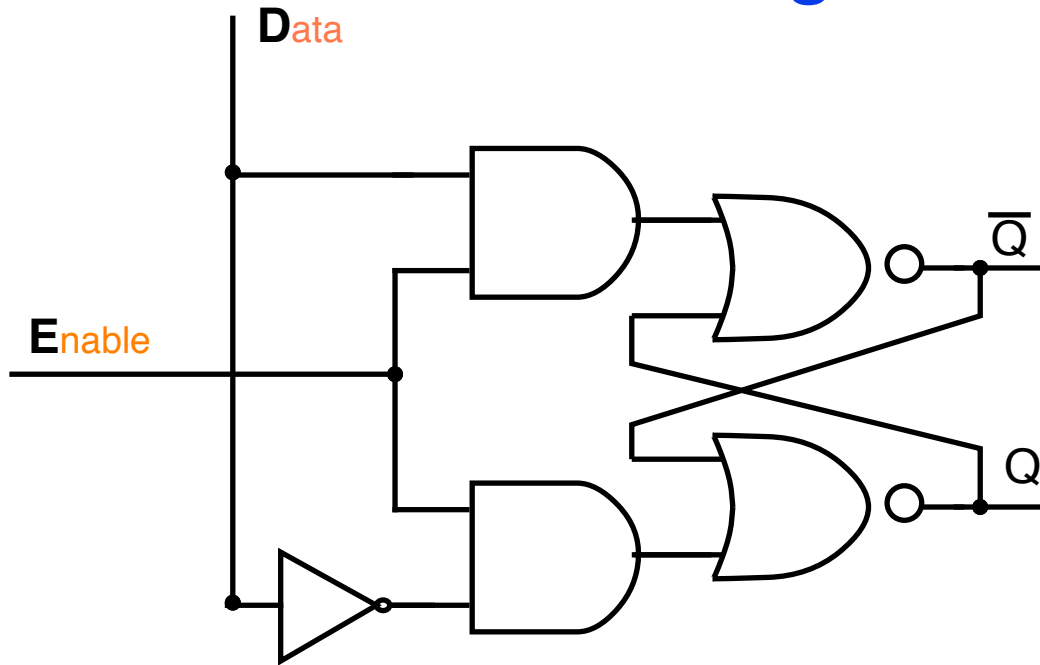
# Review: Rest-Set Latch



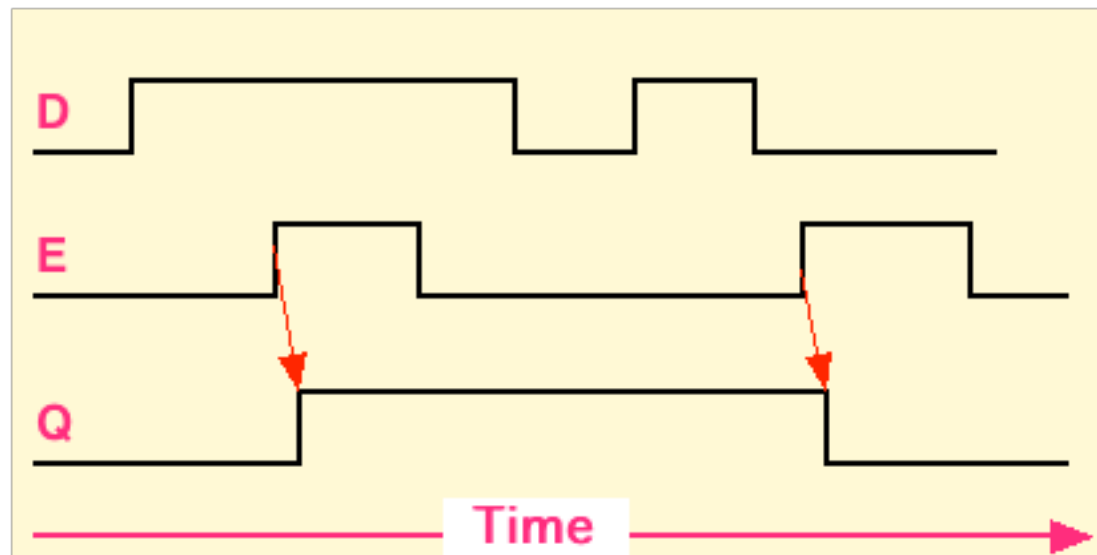
# Review: Rest-Set Latch (cont.)



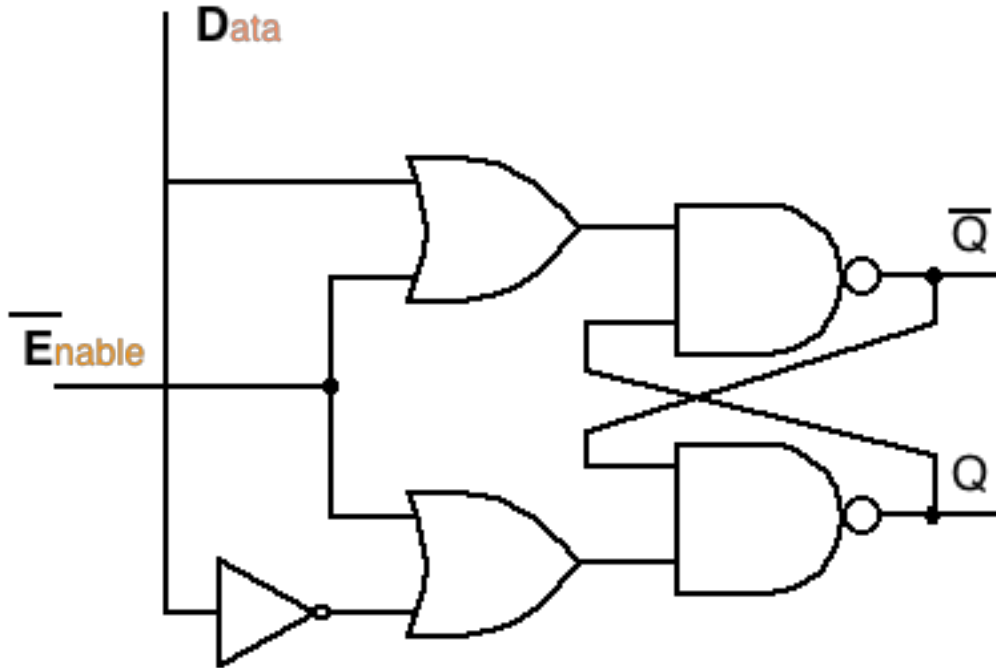
# Review: Positive Edge Data-Latch



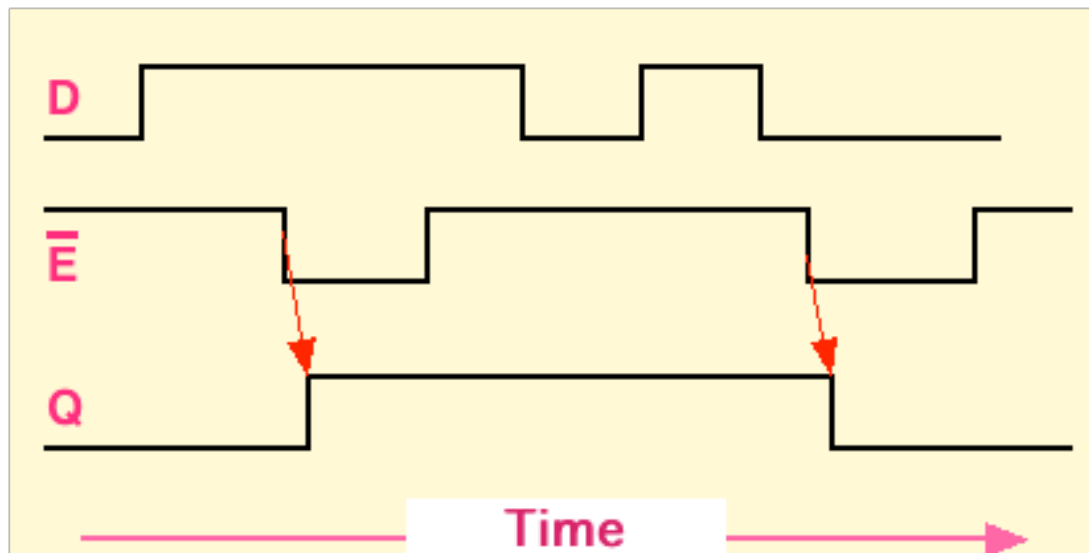
D	E	Q
0	1	0
1	1	1
-	0	Q



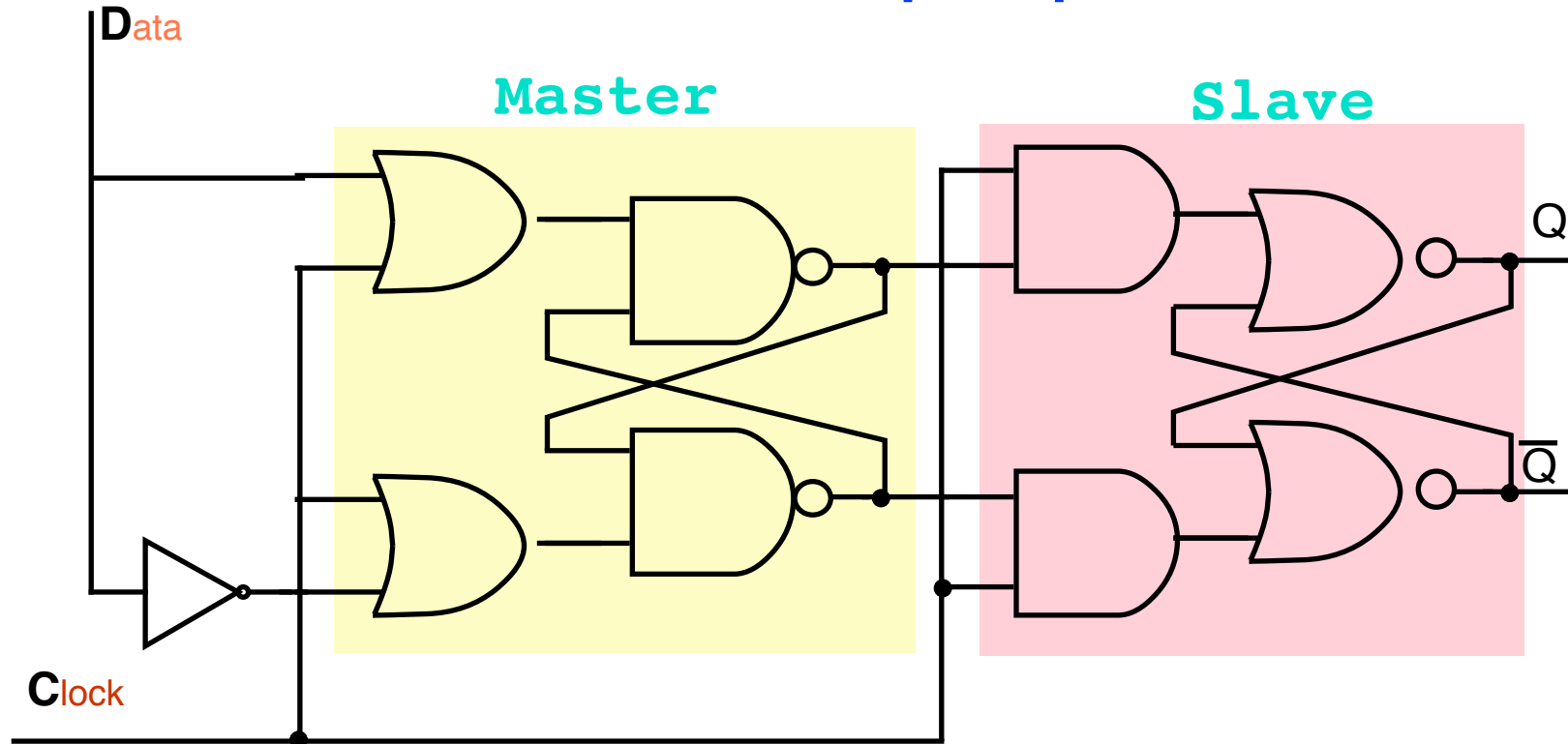
# Review: Negative Edge D-Latch



D	E	Q
0	1	0
1	1	1
-	0	Q

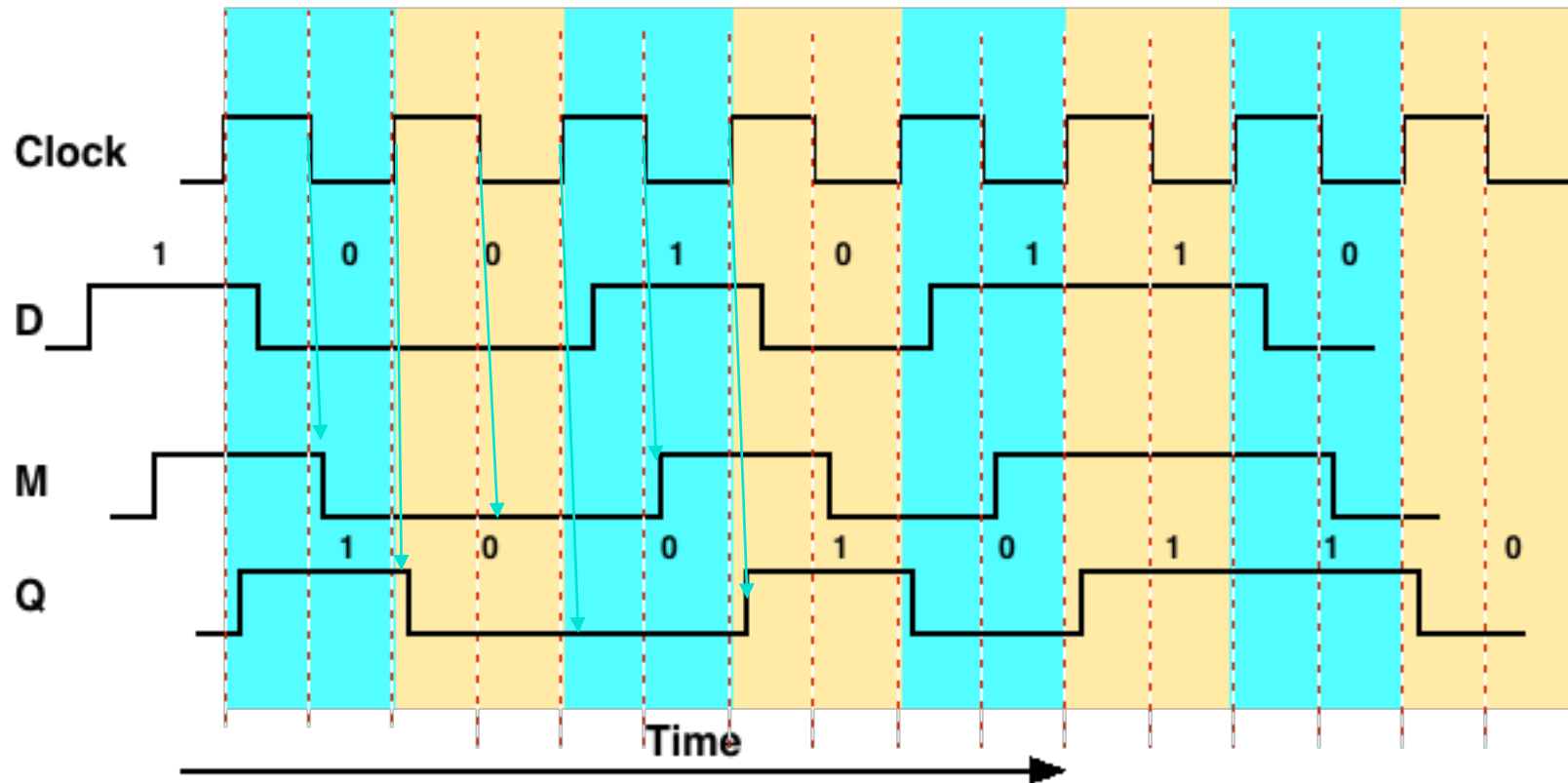
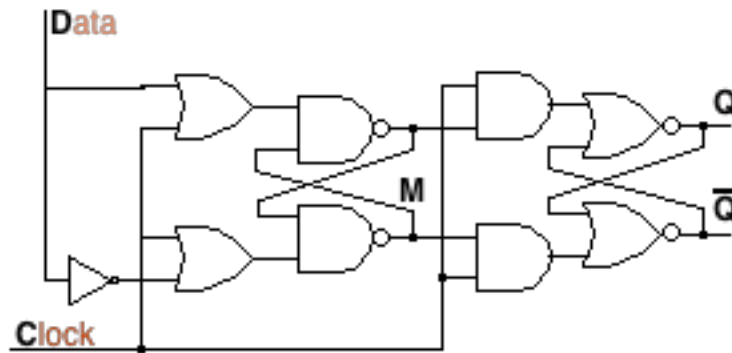


# Review: Master-Slave Data-Flip-Flop



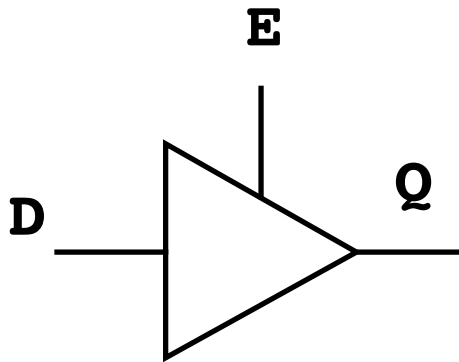
- ◆ On C ↓ D is transferred to the master stage and the slave is stable.
- ◆ On C ↑ the Master stage is transferred into the slave stage (output), and the master stage is stable.

# Review: DFF Timing



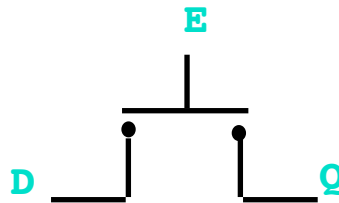
# Review: Tri-State Driver

- The Tri-State driver is like a (one directional) switch:
  - ◆ When the Enable is on ( $E=1$ ) it transfers the input to the output.
  - ◆ When the Enable is off ( $E=0$ ) it disconnects the output.

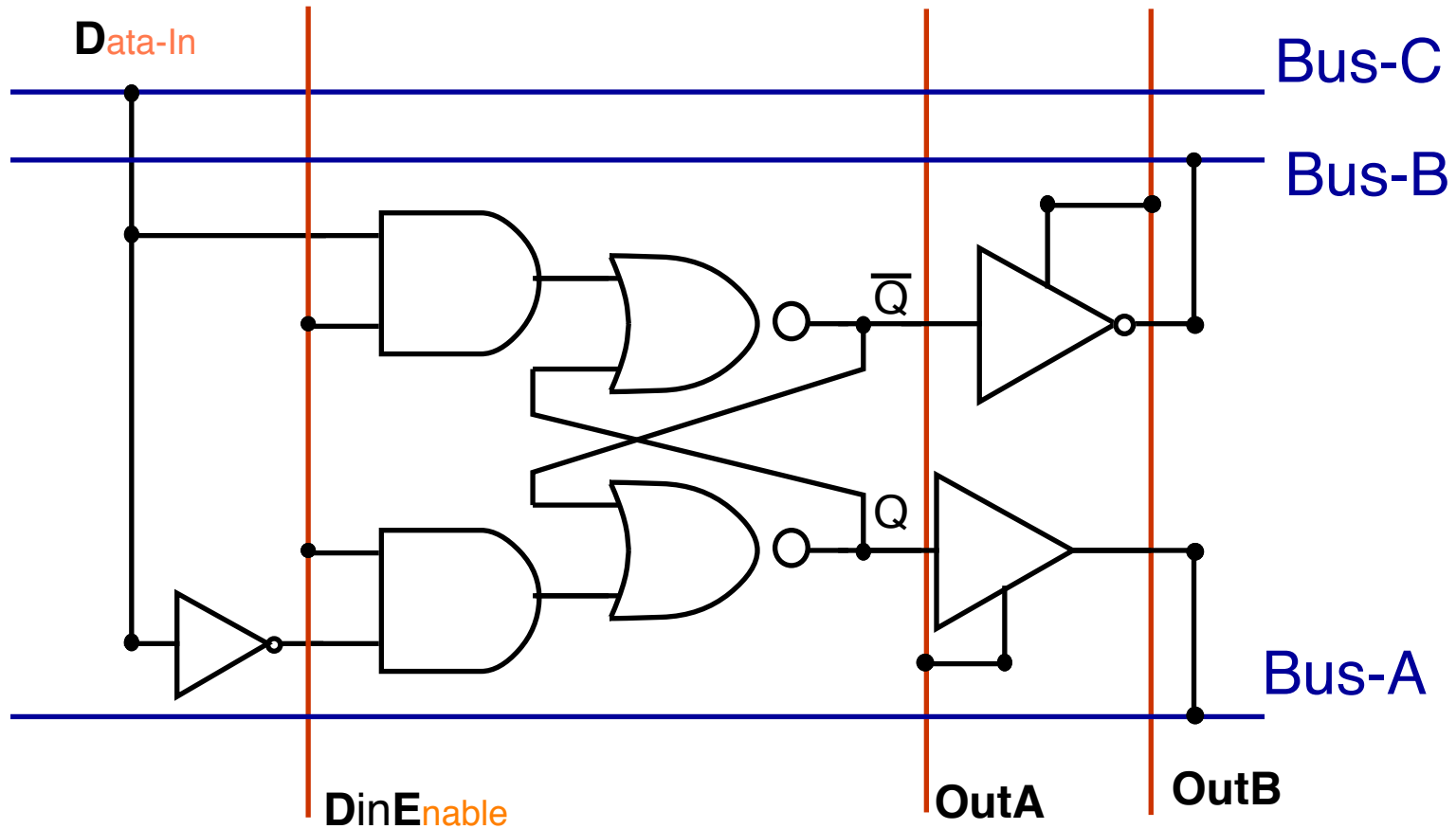


D	E	Q
0	1	0
1	1	1
-	0	Z

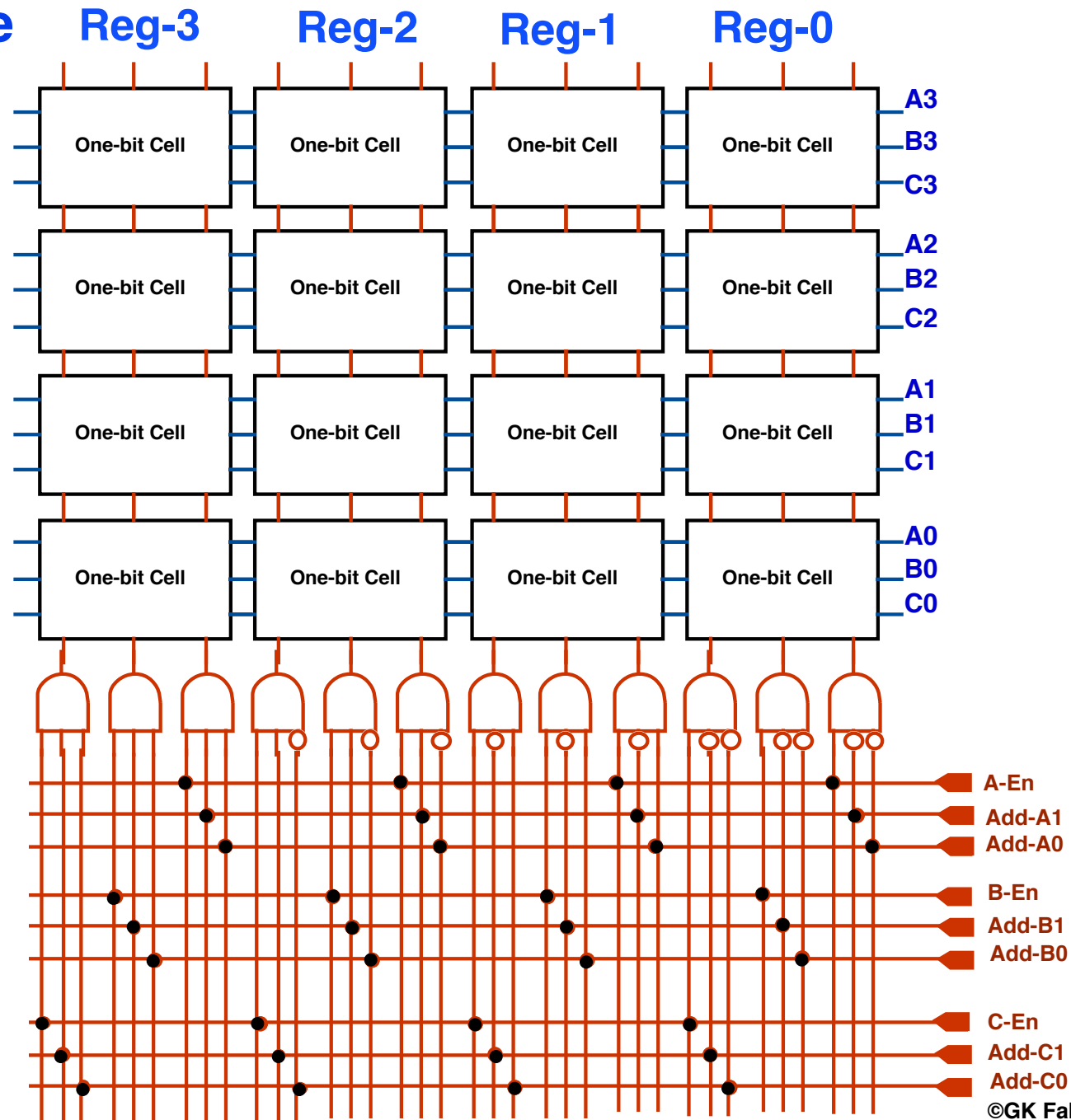
**Z :- High Impedance**



# Review: 3-Port Register Cell



# Register File

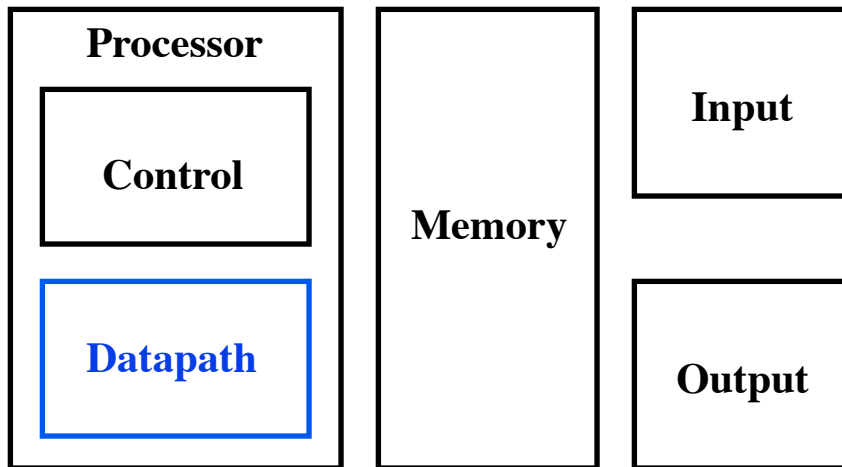


# Summary

- So far we saw how to take a Boolean function and generate a circuit that “realize” the function.
- We learned to construct circuits that can **add** and **subtract**.
- We learned about the **ALU**: a circuit that can **add, subtract, detect overflow, compare**, and do **bit-wise operations (AND, OR, NOT)**
- Saw how to construct a **shifter** circuit.
- Learned about the memory elements: **RS-Latch, D-Latches** and **D-Flip-flops**.
- Learned about **Tri-State** drivers and **BUS** Communication. (many-to many)
- Learned about how to construct a **register file**.
- Saw how **control signals** can modify what the circuit will do with **inputs**.
  - ♦ **Examples:** ALU, Shift, Register read-write, ...

# The Big Picture: Where are We Now?

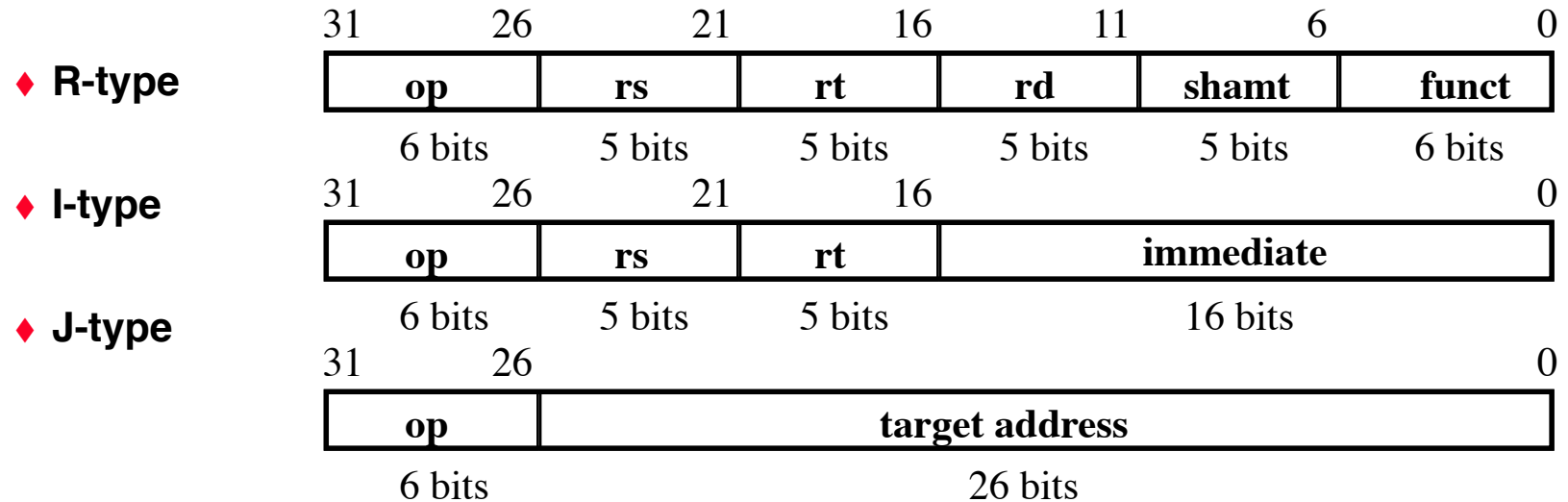
## • The Five Classic Components of a Computer



## Today's Topic: Datapath Design

# Review: The MIPS Instruction Formats

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

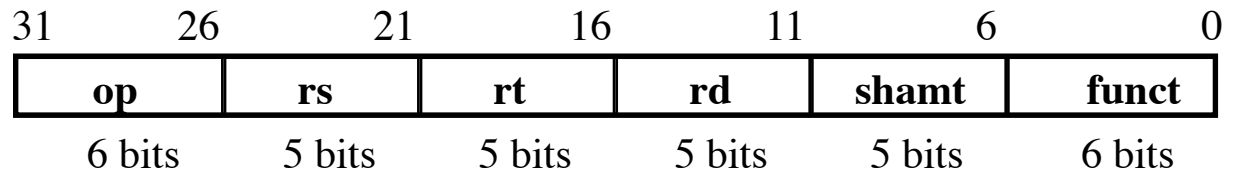


- The different fields are:
  - ♦ **op**: operation of the instruction
  - ♦ **rs, rt, rd**: the source and destination register specifiers
  - ♦ **shamt**: shift amount
  - ♦ **funct**: selects the variant of the operation in the “op” field
  - ♦ **address / immediate**: address offset or immediate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

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

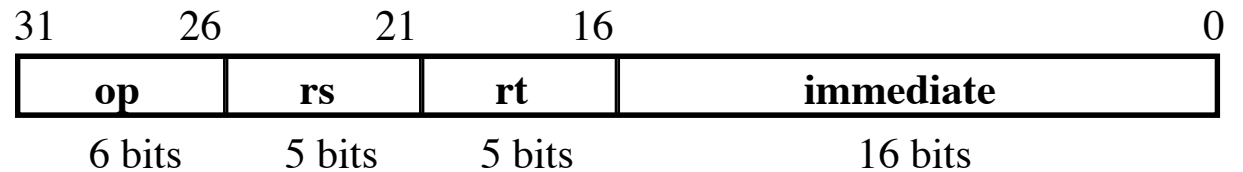
- **ADD and subtract**

- ◆ add rd, rs, rt
- ◆ sub rd, rs, rt



- **OR Immediate:**

- ◆ ori rt, rs, imm16



- **LOAD and STORE**

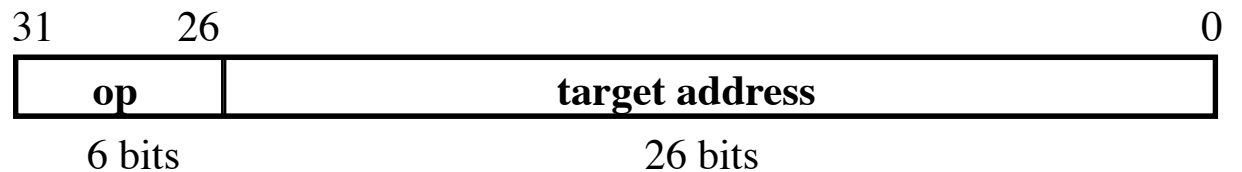
- ◆ lw rt, rs, imm16
- ◆ sw rt, rs, imm16

- **BRANCH:**

- ◆ beq rs, rt, imm16

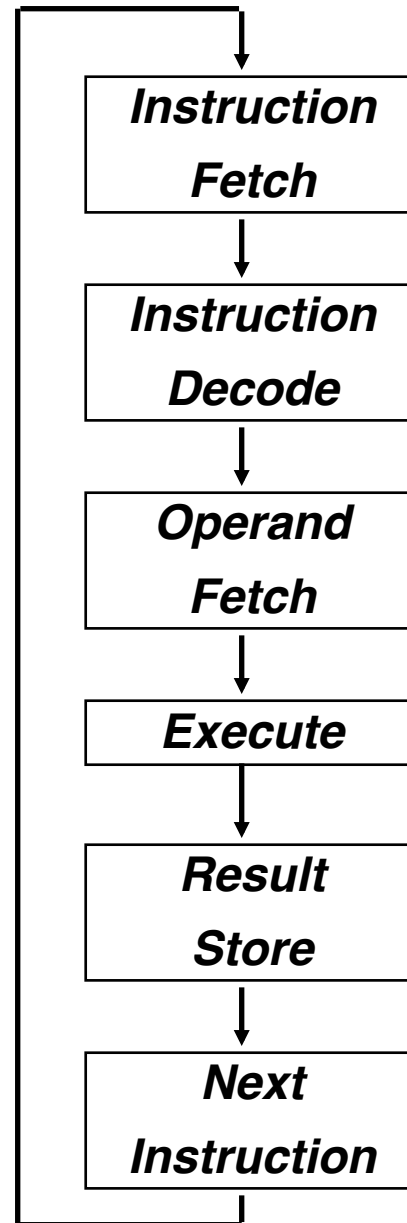
- **JUMP:**

- ◆ j target



# The Hardware “Program”

How does one build hardware that implements the MIPS instructions?



# 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**

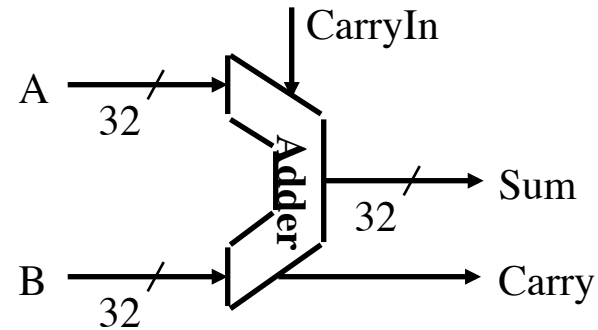
◆ **Addr <- R[rs] + SignExt(imm16)**  
   **Calculate the memory address**

◆ **R[rt] <- Mem[Addr]** **Load the data into the register**

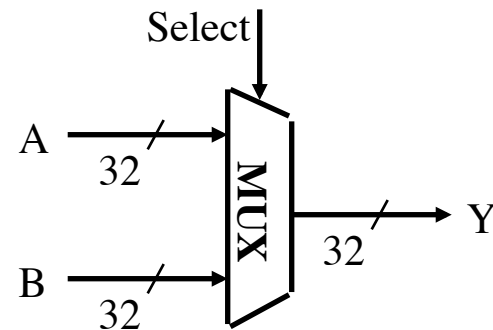
◆ **PC <- PC + 4**                      **Calculate the next instruction's address**

# Combinational-Logic Elements (Basic Building Blocks)

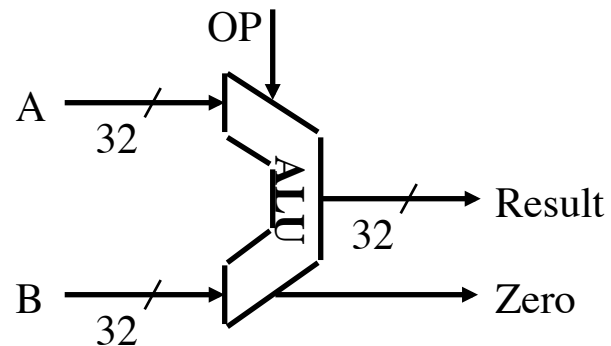
## • Adder



## • MUX



## • ALU



# Storage Element: Register (Basic Building Block)

## • Register

- ◆ Similar to the D Flip Flop except

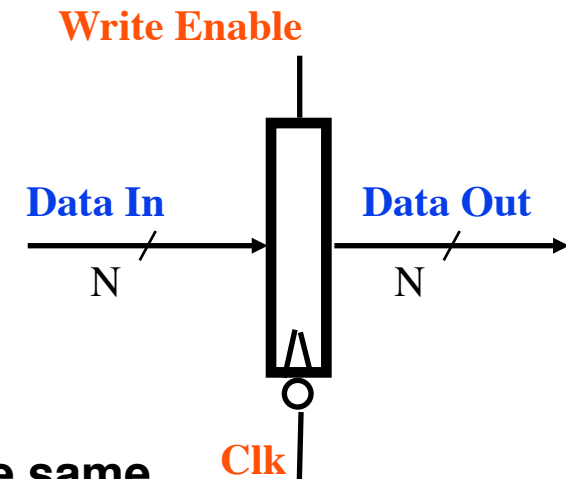
- N-bit input and output

- Shared Write Enable input

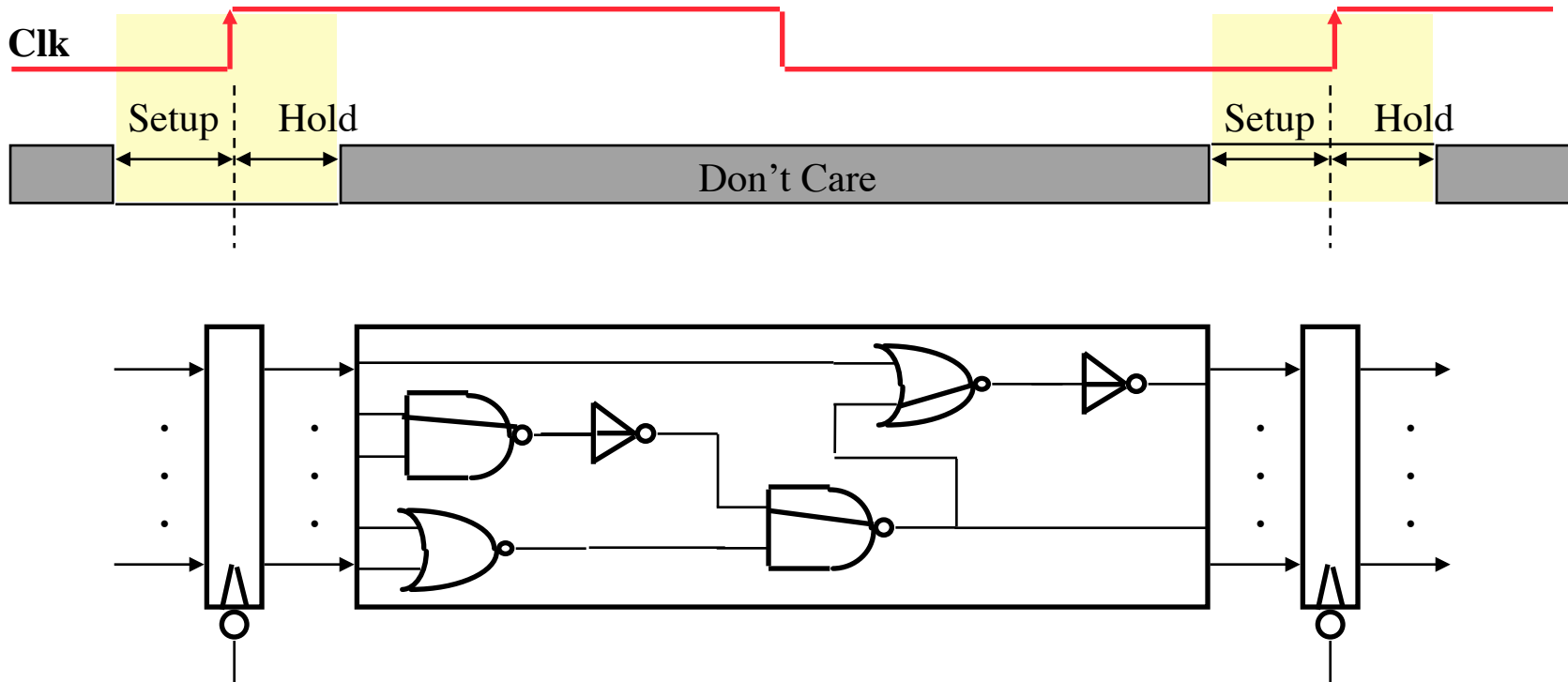
- ◆ Write Enable:

- 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
- $\text{Cycle Time} \geq \text{CLK-to-Q} + \text{Longest Delay Path} + \text{Setup} + \text{Clock Skew}$

# Storage Element: Register File

- Register File consists of 32 registers:

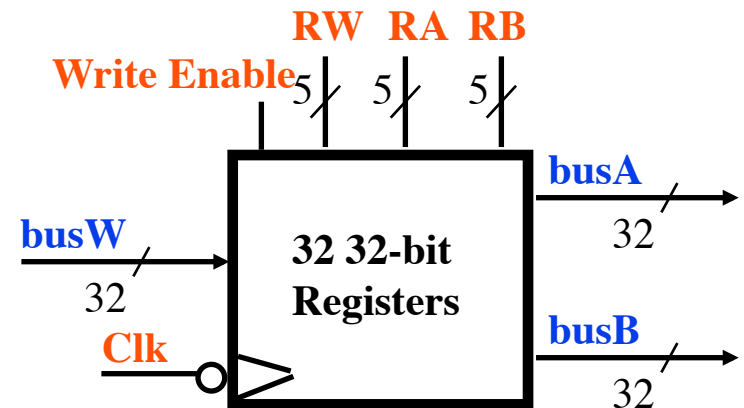
- ◆ Two 32-bit output busses:  
**busA** and **busB**
- ◆ One 32-bit input bus: **busW**

- 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)

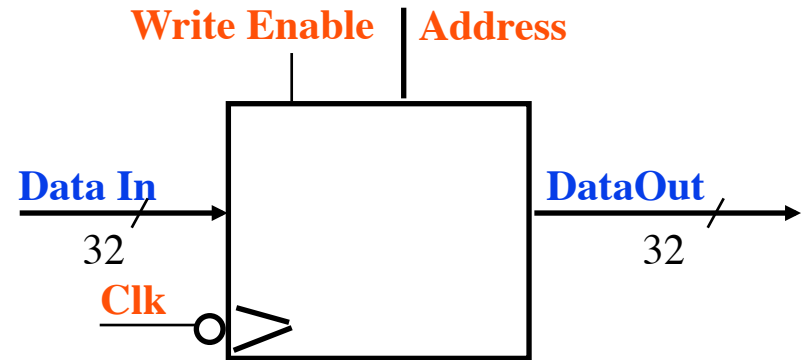
- ◆ The **CLK** input is a factor ONLY during write operation
- ◆ During read operation, behaves as a combinational logic block:
  - **RA** or **RB** valid  $\Rightarrow$  **busA** or **busB** valid after “access time.”



# 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  $\Rightarrow$  **Data Out** valid after “access time.”

# Overview of the Instruction Fetch Unit

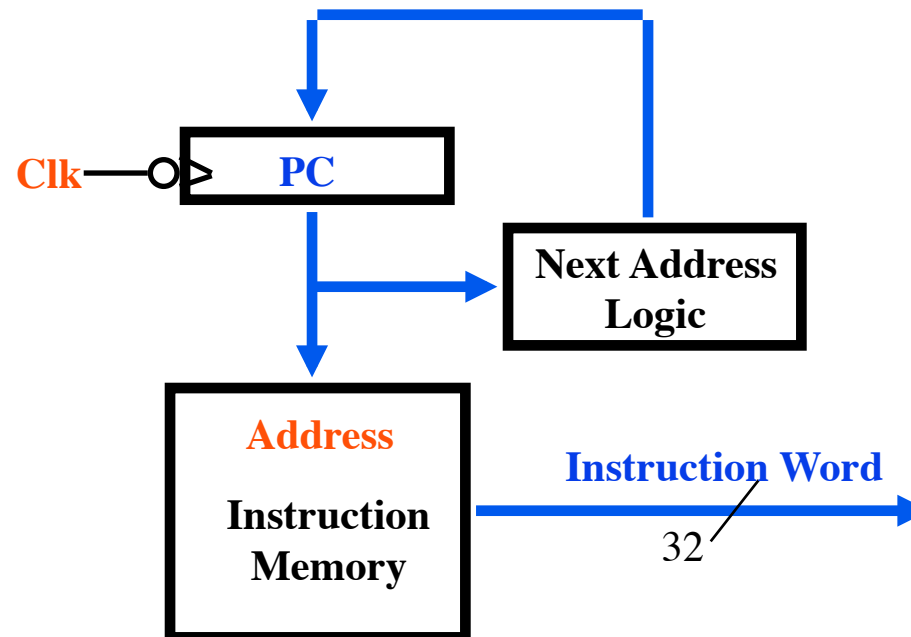
- The common RTL operations

- ◆ Fetch the Instruction:  $\text{mem}[\text{PC}]$

- ◆ Update the program counter:

- Sequential Code:  $\text{PC} \leftarrow \text{PC} + 4$

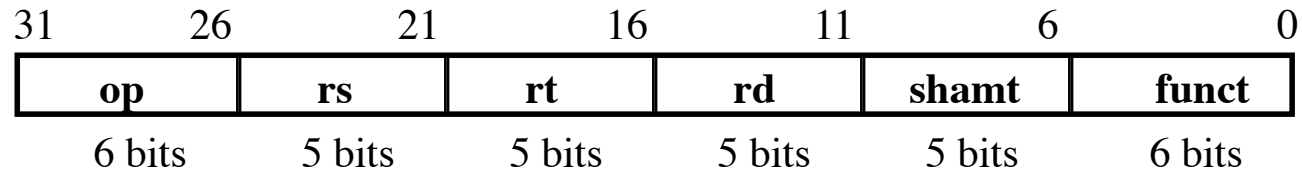
- Branch and Jump:  $\text{PC} \leftarrow$  “something else”



# The Instructions Subset

- **add/sub**      **Typical Register to register Instructions**
- **ori**      **A typical Register & Immediate Instruction**
- **lw**      **Load word**
- **sw**      **Store word**
- **beq**      **A typical branch instruction**
- **j**      **The jump instruction**

# RTL: The ADD Instruction



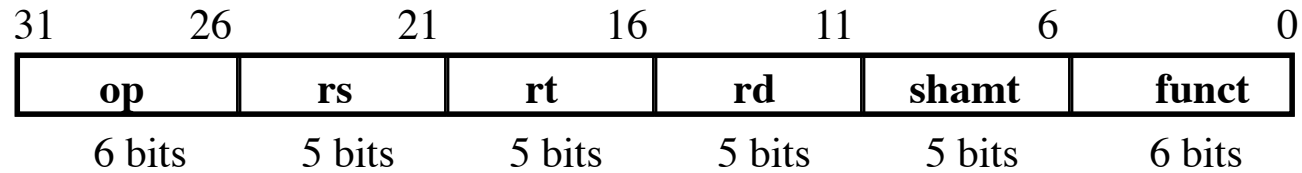
• **add rd, rs, rt**

◆ **mem[PC]**                      **Fetch the instruction from memory**

◆ **R[rd] ← R[rs] + R[rt]**                      **The actual operation**

◆ **PC ← PC + 4**                      **Calculate the next instruction's address**

# RTL: The Subtract Instruction



• **sub rd, rs, rt**

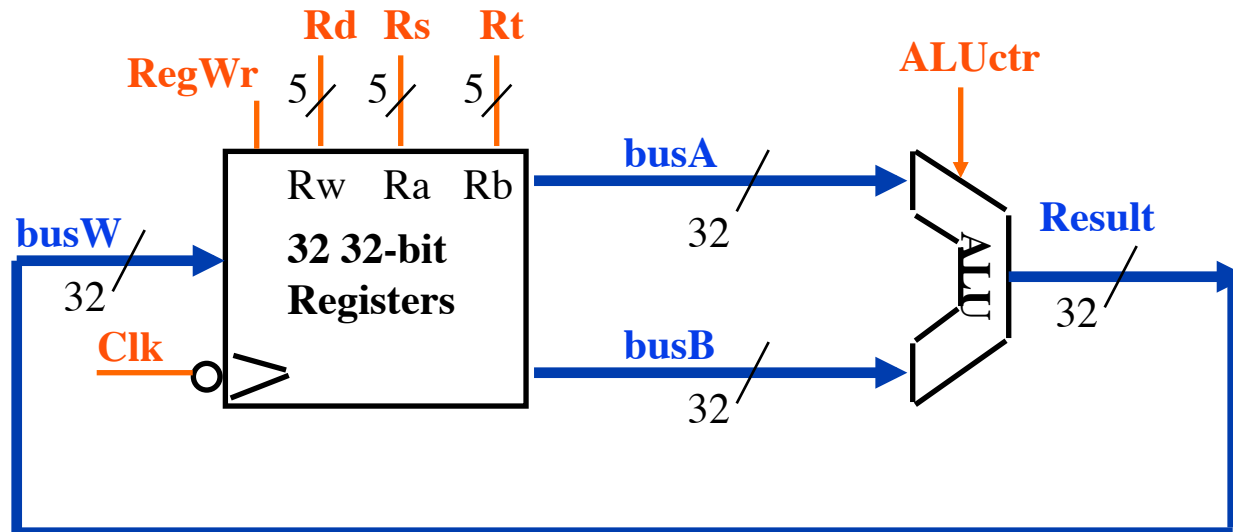
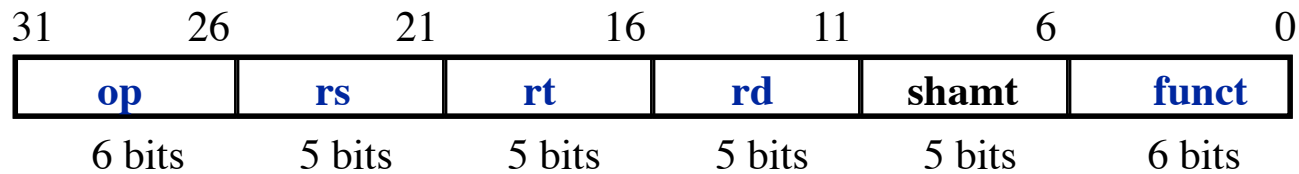
♦ **mem[PC]**                      **Fetch the instruction from memory**

♦ **R[rd] ← R[rs] - R[rt]**                      **The actual operation**

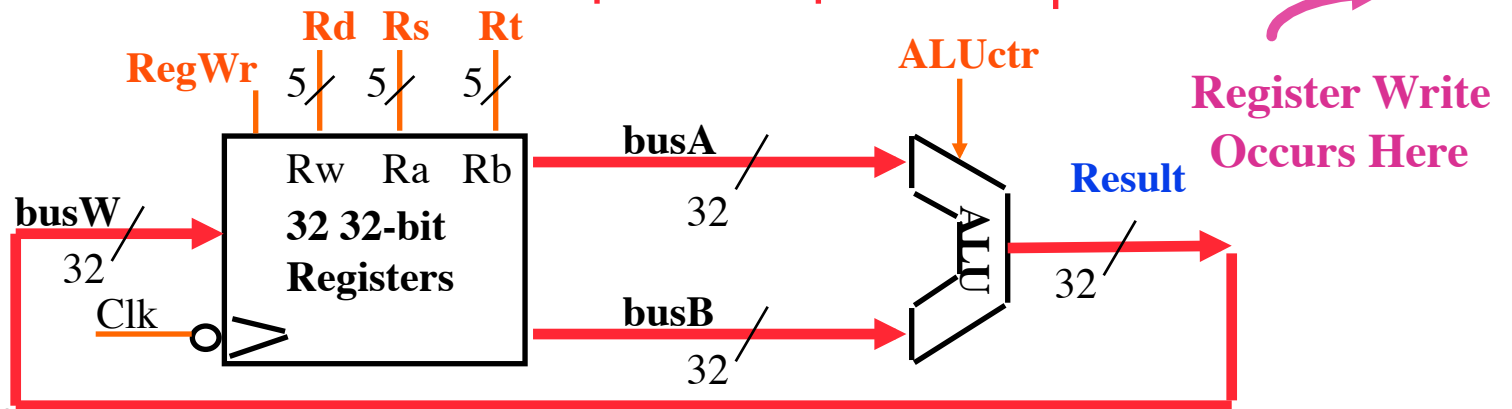
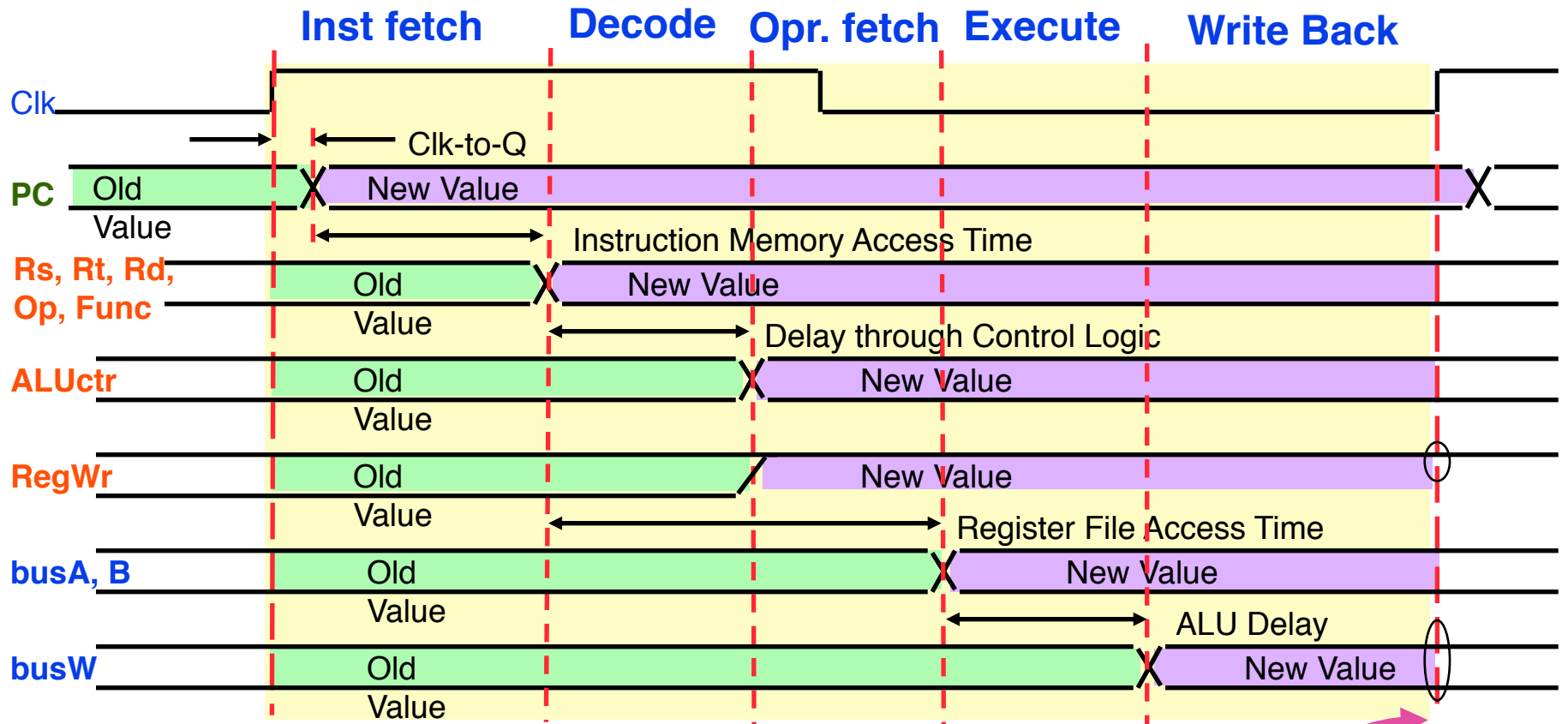
♦ **PC ← PC + 4**                      **Calculate the next instruction's address**

# Datapath for Register-Register Operations

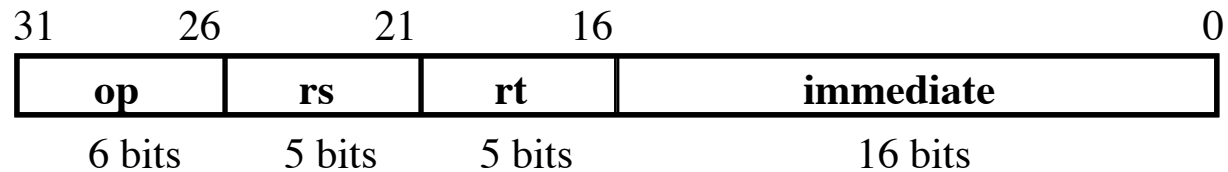
- \*  $R[rd] \leftarrow R[rs] \text{ op } R[rt]$ 
Example: add rd, rs, rt
  - ♦  $Ra$ ,  $Rb$ , and  $Rw$  comes from instruction's  $rs$ ,  $rt$ , and  $rd$  fields
  - ♦  $ALUctr$  and  $RegWr$ : control logic after decoding the instruction fields:  $op$  and  $func$



# Register-Register Timing

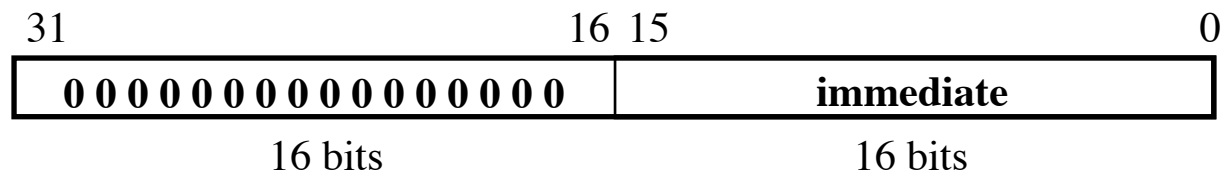


# RTL: The OR Immediate Instruction



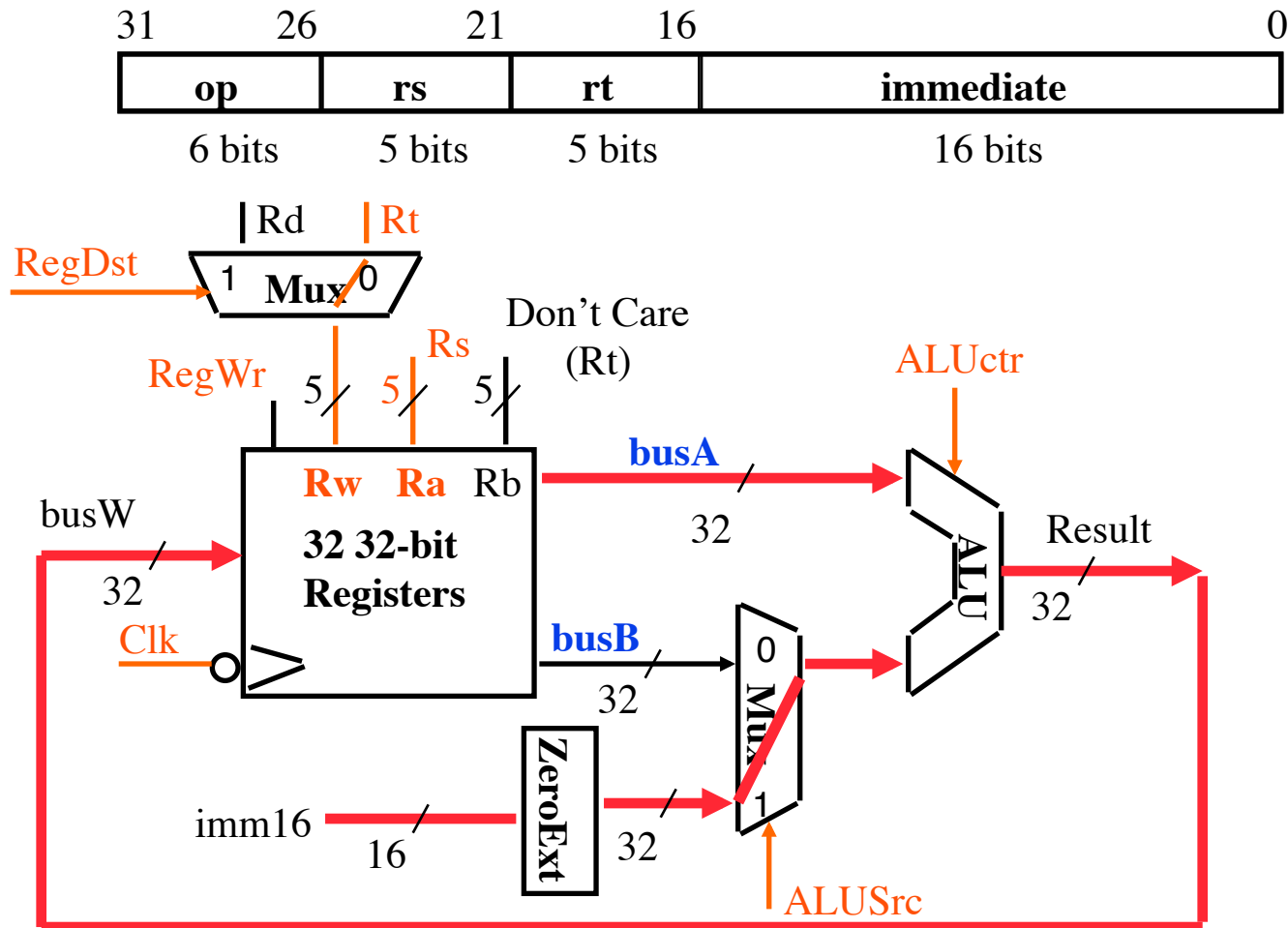
• **ori** `rt, rs, imm16`

- ◆ **mem[PC]**                      **Fetch the instruction from memory**
- ◆ **R[rt] ← R[rs] OR ZeroExt(imm16)**      **The OR operation**
- ◆ **PC ← PC + 4**                      **Calculate the next instruction's address**



# Datapath for Logical Operations with Immediate

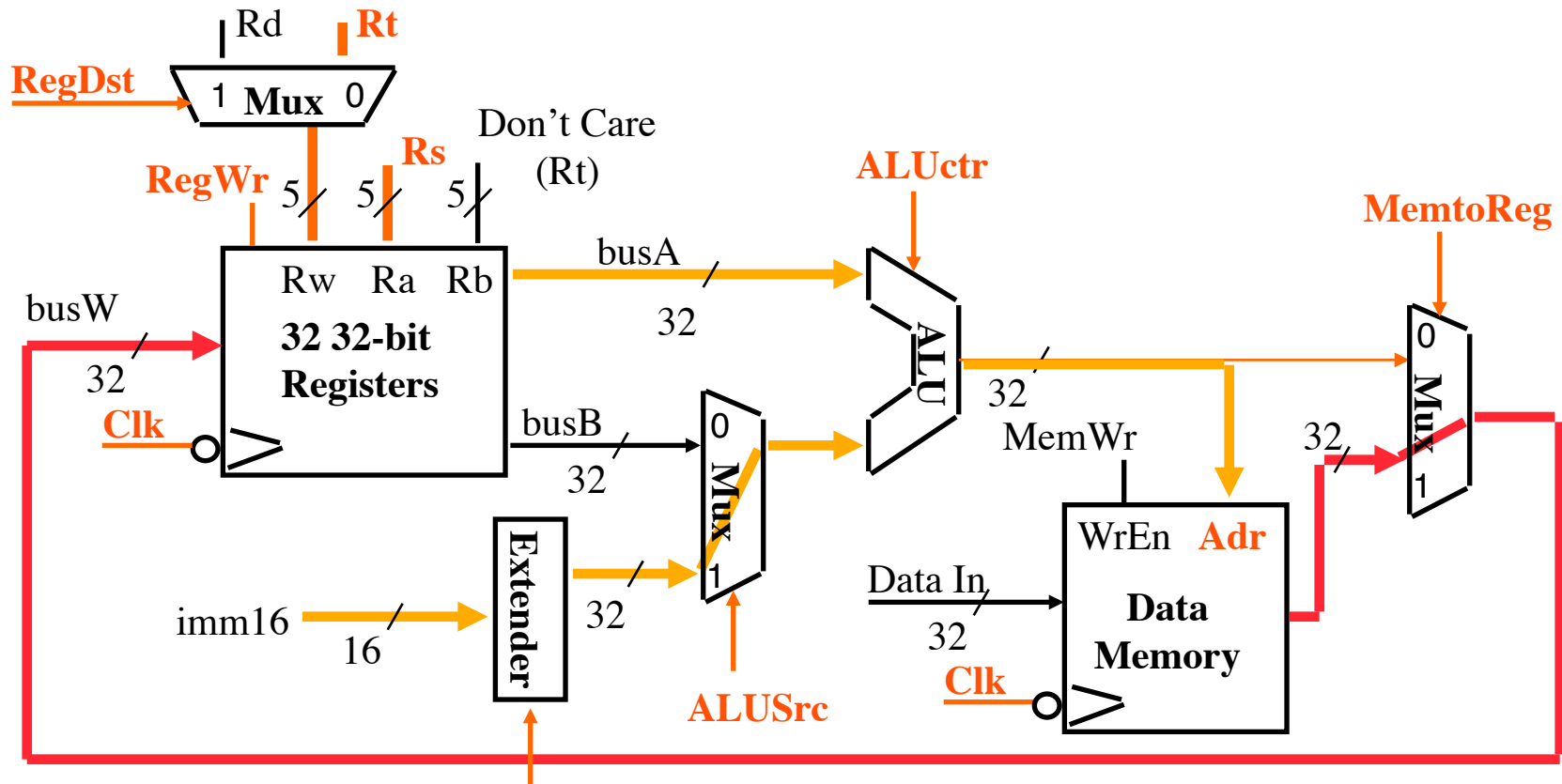
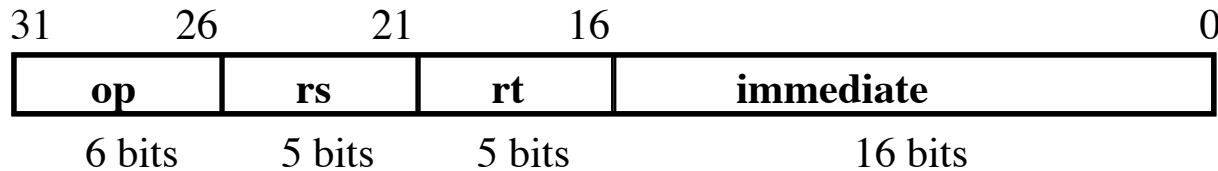
- $R[rt] \leftarrow R[rs] \text{ op ZeroExt}[imm16]$ 
Example: ori rt, rs, imm16



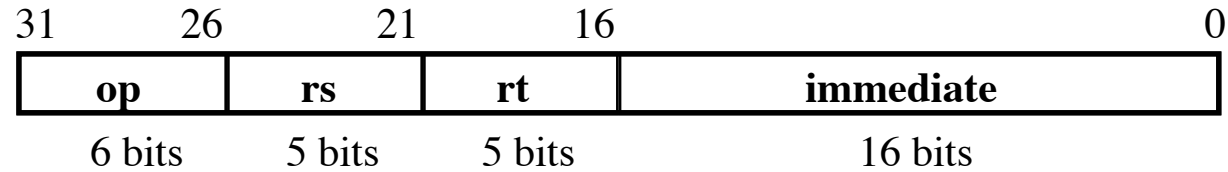


# Datapath for Load Operations

- $R[rt] \leftarrow Mem[R[rs] + SignExt[imm16]]$ 
Example: lw rt, rs, imm16



# RTL: The Store Instruction



## • **sw** **rt, rs, imm16**

◆ **mem[PC]** Fetch the instruction from memory

◆ **Addr** <- **R[rs] + SignExt(imm16)**

Calculate the memory address

◆ **Mem[Addr]** <- **R[rt]** Store the register into memory

◆ **PC** <- **PC + 4**

Calculate the next instruction's address

# Datapath for Store Operations

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

