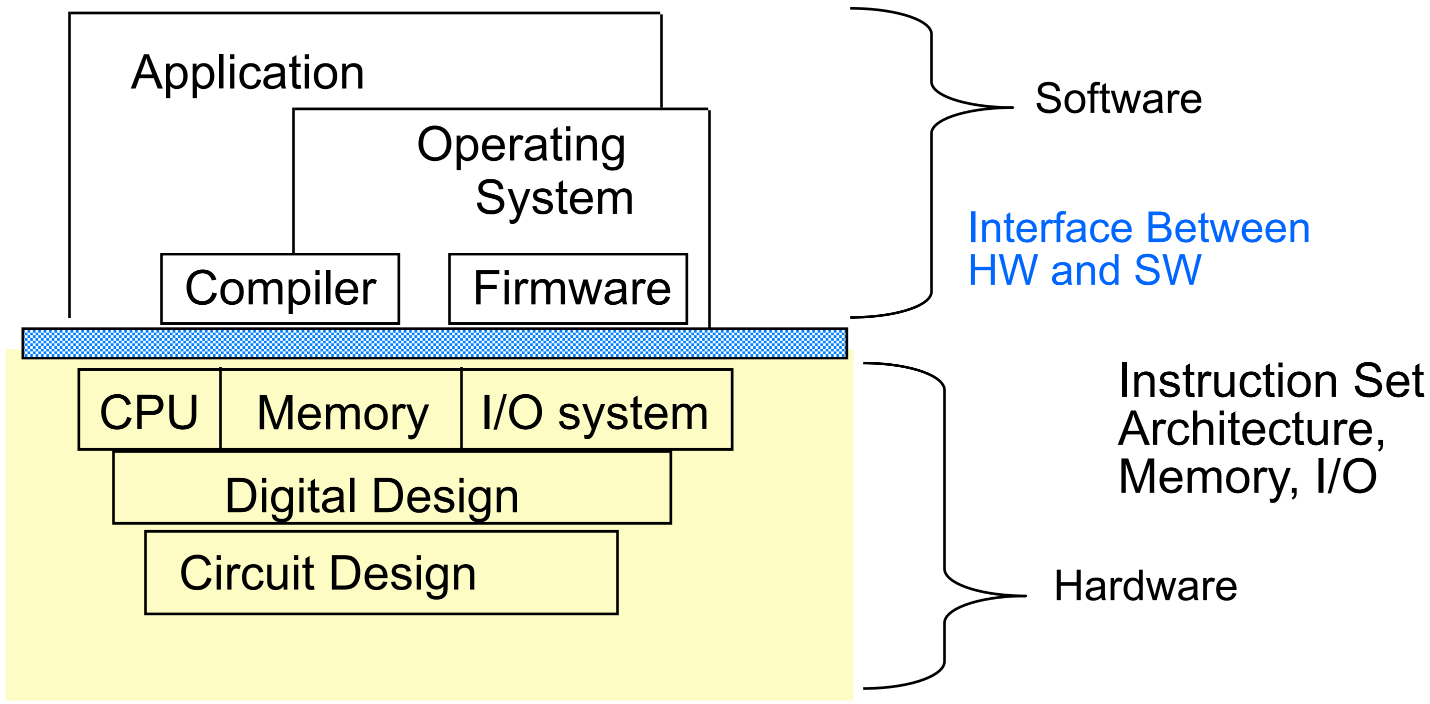
CS/ECE 250: Computer Architecture

Designing a Single Cycle Datapath

Duke University

What is Computer Architecture?

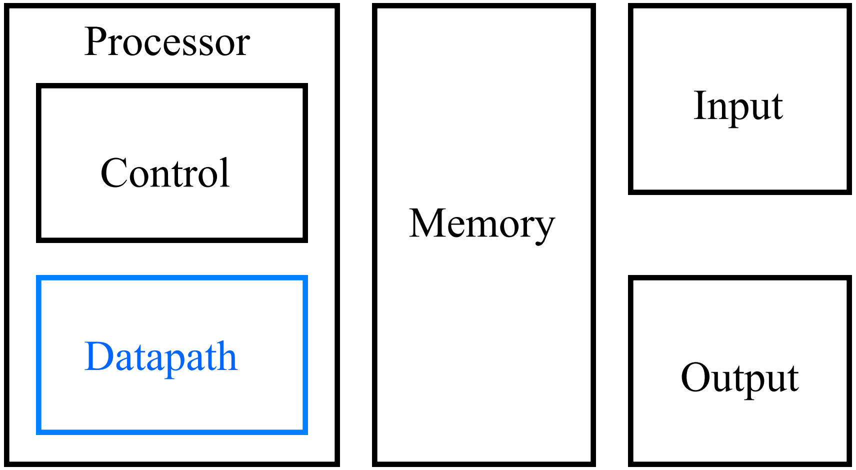
- Coordination of levels of abstraction



- Under a set of rapidly changing *technology Forces*

The Big Picture: Where are We Now?

- **The Five Classic Components of a Computer**

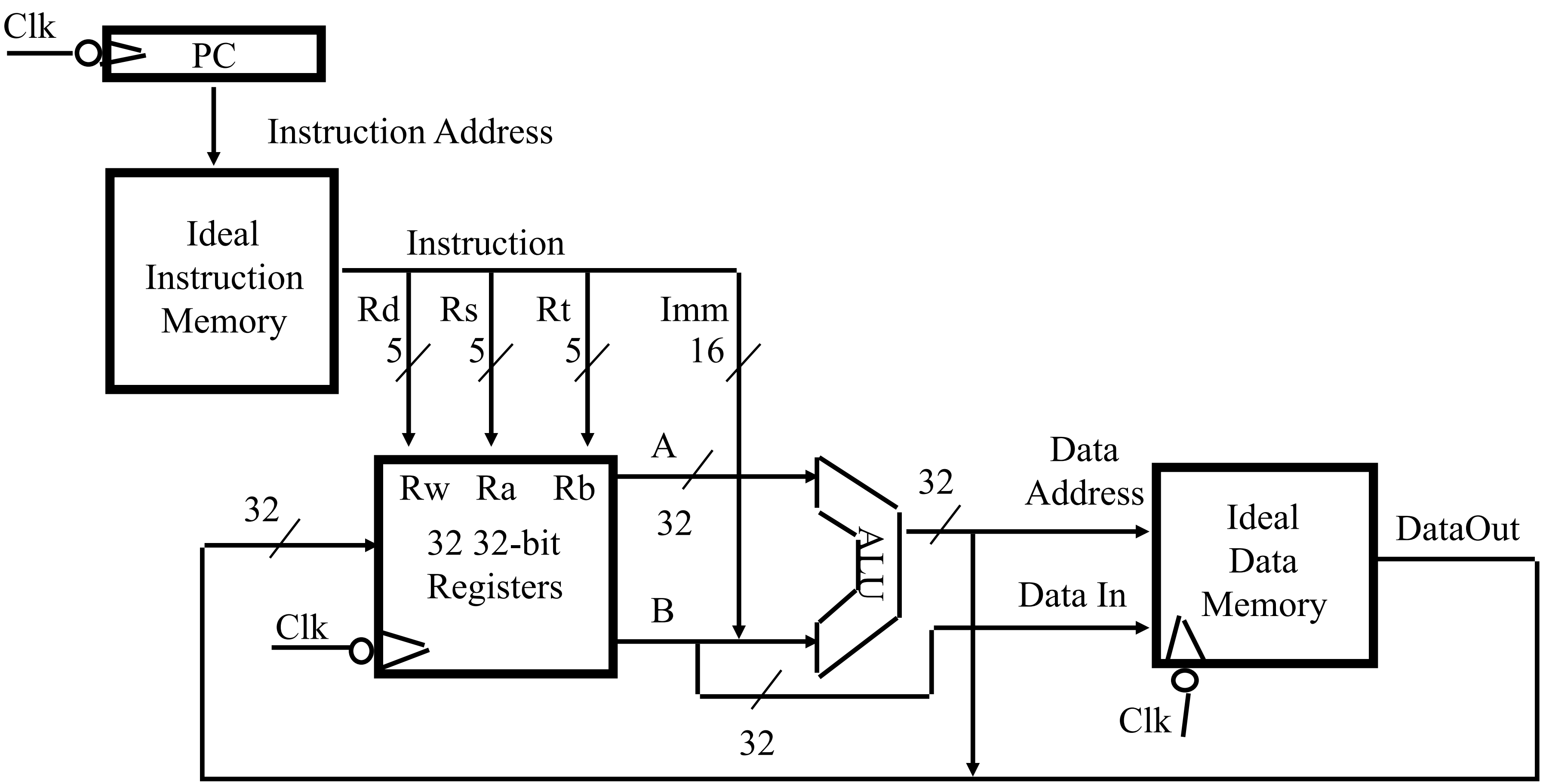


Today's Topic: Datapath Design

Datapath Design

- How do we build hardware to implement the MIPS instructions?
- Add, LW, SW, Beq, Jump

An Abstract View of the Implementation



The MIPS Instruction Formats

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

31–26	25–21	20–16	15–11	10–6	5–0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

  - **I-type**

31–26	25–21	20–16	15–0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

  - **J-type**

31–26	25–0
op	target address
6 bits	26 bits

- **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 implement them all!)

- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt

31–26	25–21	20–16	15–11	10–6	5–0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16

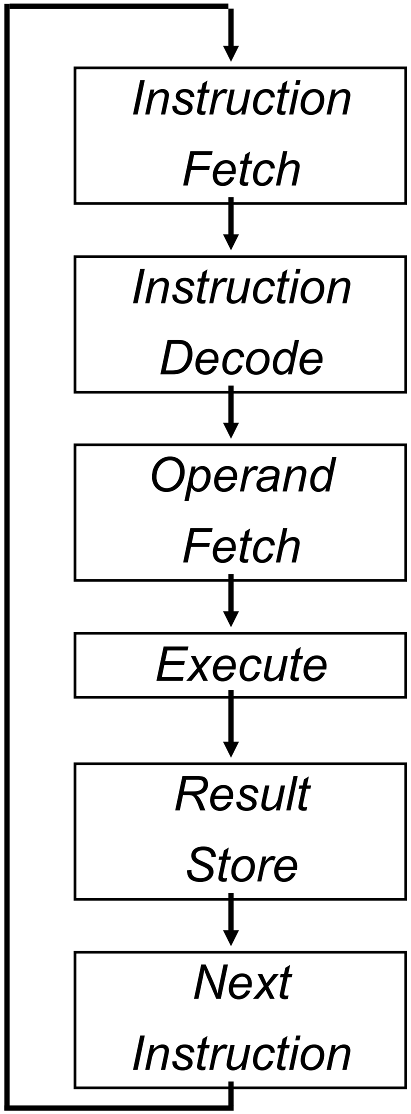
31–26	25–21	20–16	15–0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target

31–26	25–0
op	target address
6 bits	26 bits

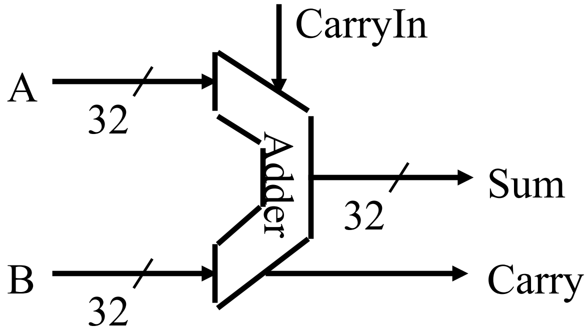
The Hardware "Program"

How do we build the hardware to implement the MIPS instructions and their sequencing?

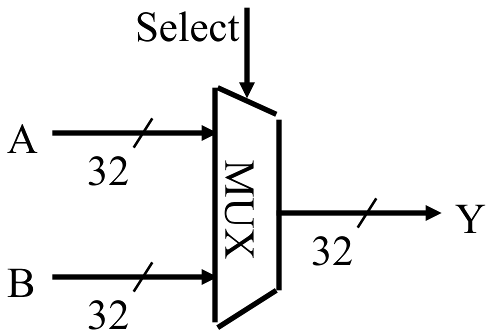


Combinational Logic (Basic Building Blocks)

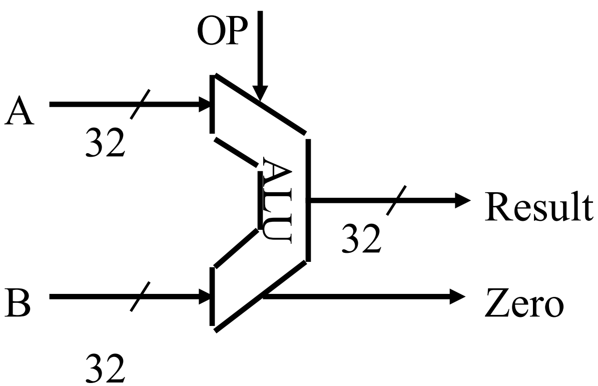
- **Adder**



- **MUX**

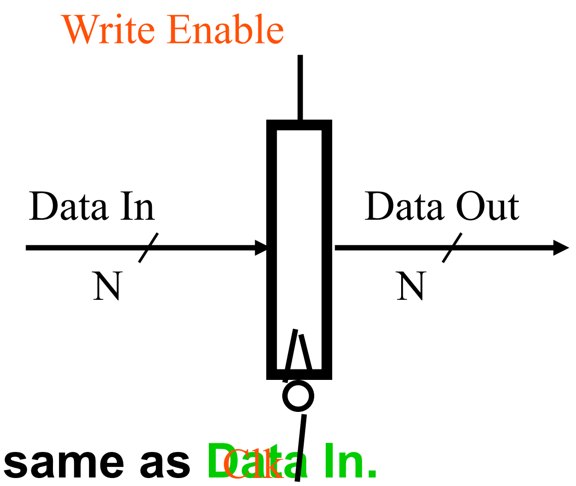


- **ALU**

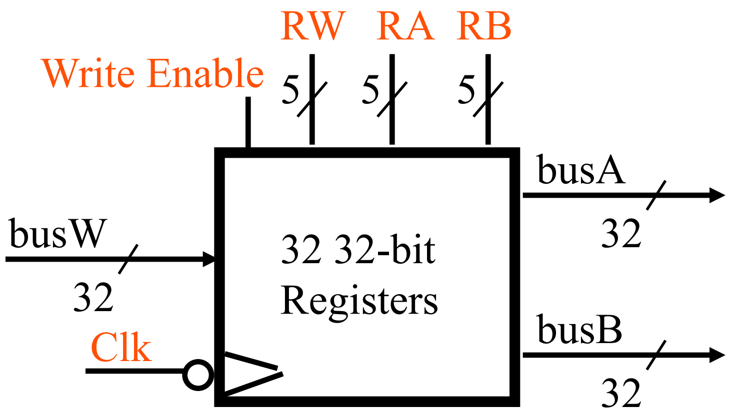


Storage Element: Register (Basic Building Block)

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

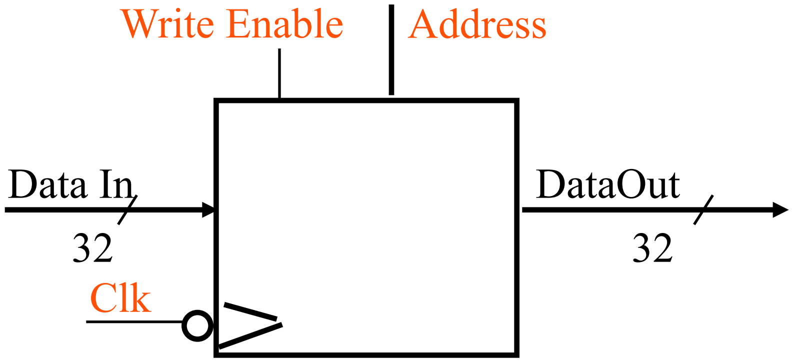


Storage Element: Register File



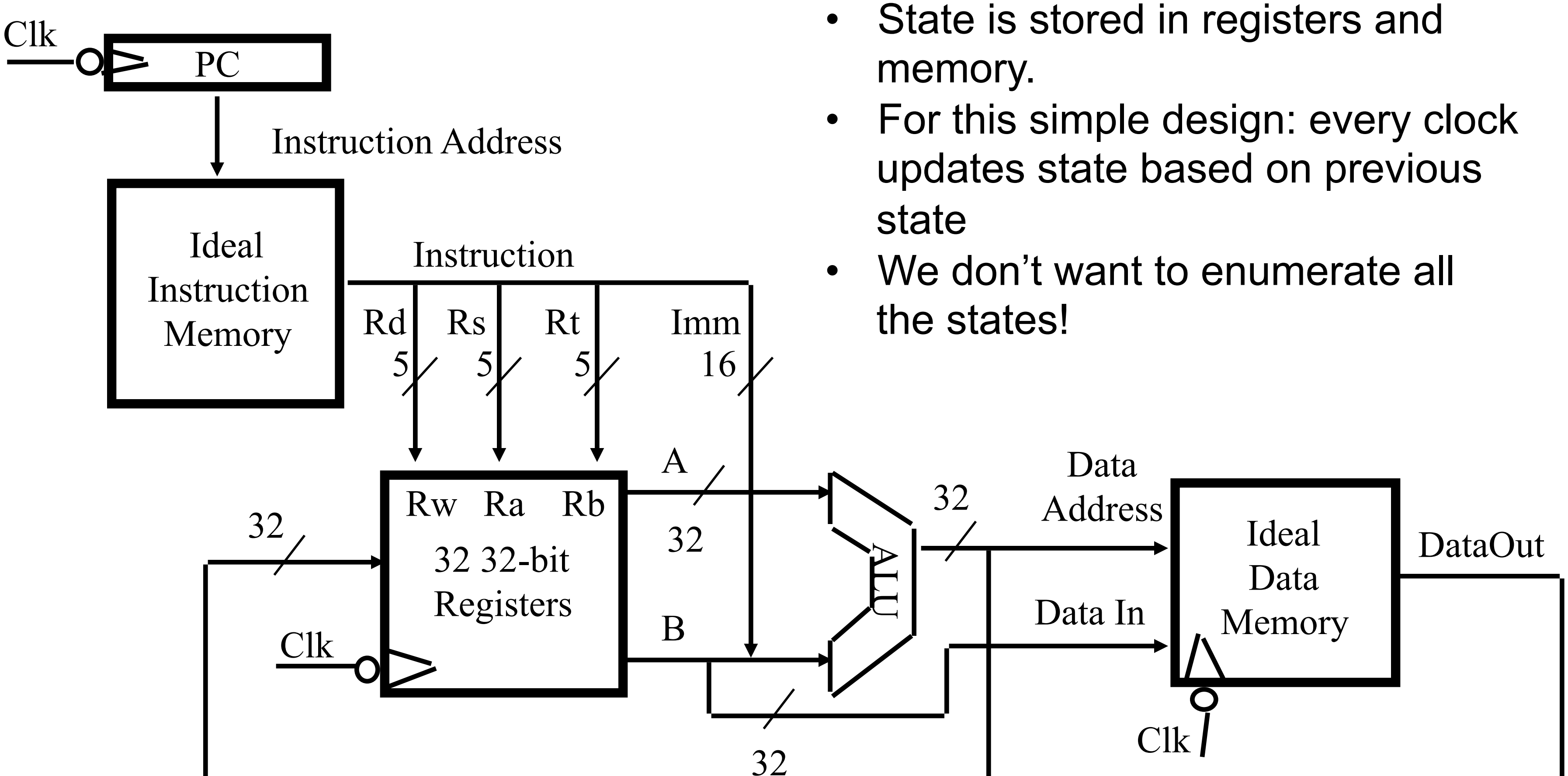
- Register File consists of 32 registers:
  - Two 32-bit output busses:
    busA and busB
  - One 32-bit input bus: busW
- Register is selected by:
  - RA selects the register to put on busA
  - RB selects the register to put on busB
  - RW selects the register to be written via busW when Write Enable 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



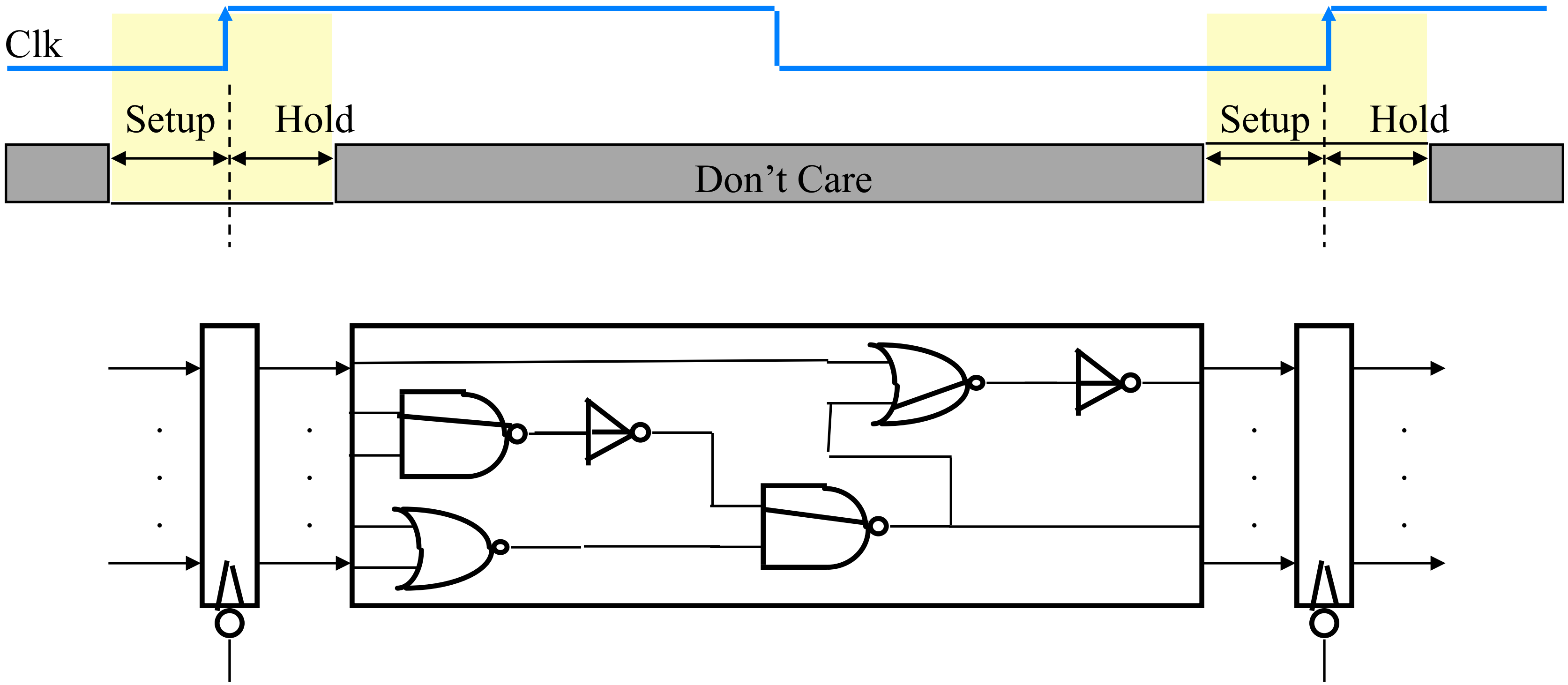
- Memory (idealized)
  - One read/write port
    - On access per cycle (read or write)
  - One input Data bus: Data In
  - One output Data bus: Data Out
  - One Address bus: Address
- Memory word is selected by:
  - Write Enable = 0: Address selects the word to put on the Data Out bus
  - Write Enable = 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.”

An Abstract View of the Implementation



- Can think of computer as large finite state machine
- State is stored in registers and memory.
- For this simple design: every clock updates state based on previous state
- We don't want to enumerate all the states!

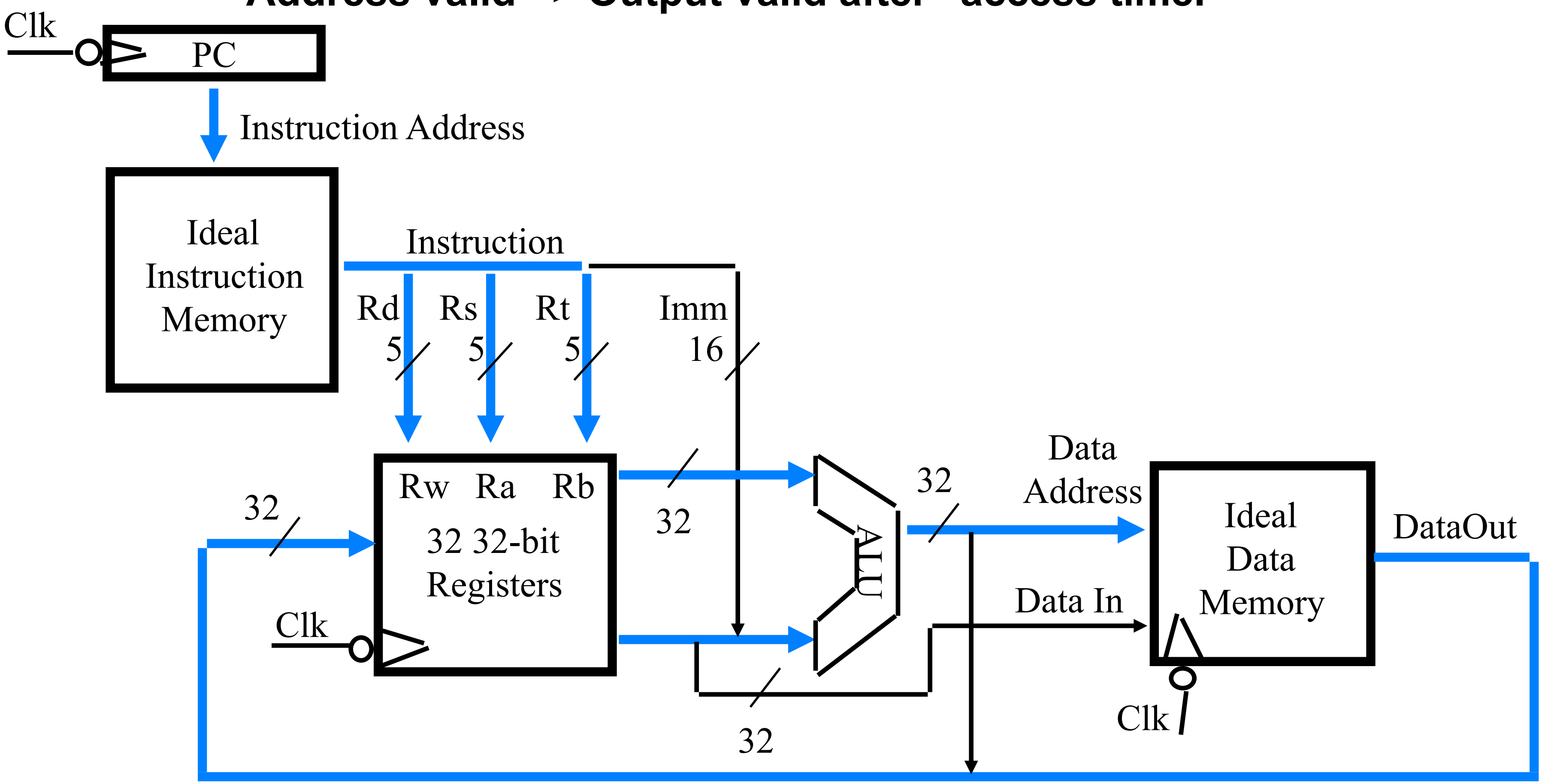
Clocking Methodology



- **All storage elements are clocked by the same clock edge (ignore NOT on clock)**
- **Cycle Time >= CLK-to-Q + Longest Delay Path + Setup + Clock Skew**
- **Longest delay path = critical path**

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."**

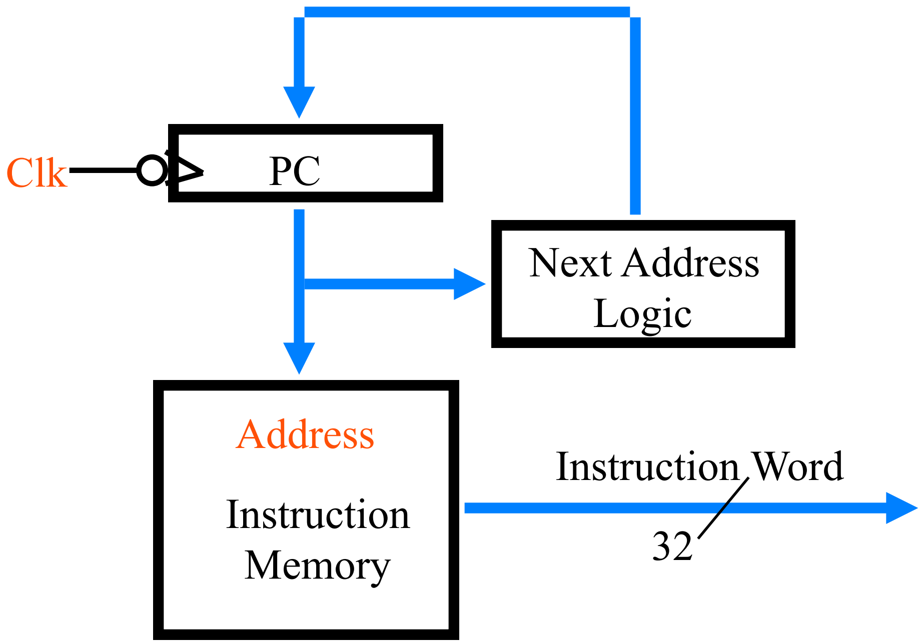


The Steps of Designing a Processor

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

Overview of the Instruction Fetch Unit

- The common RTL operations
  - Fetch the Instruction: mem[PC]
  - Update the program counter:
    - Sequential Code: PC <- PC + 4
    - Branch and Jump: PC <- "something else"



RTL: The ADD Instruction

- **add   rd, rs, t**

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

RTL: The Load Instruction

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

RTL: The ADD Instruction

31 — 26	26 — 21	21 — 16	16 — 11	11 — 6	6 — 0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- **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

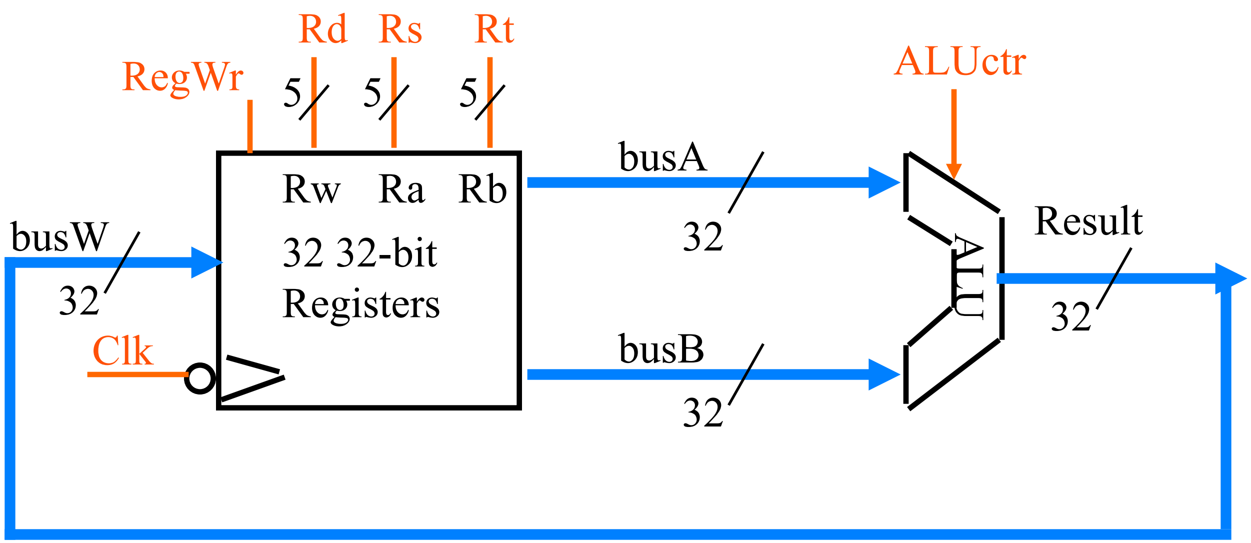
31 26	21	16	11	6	0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- **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**

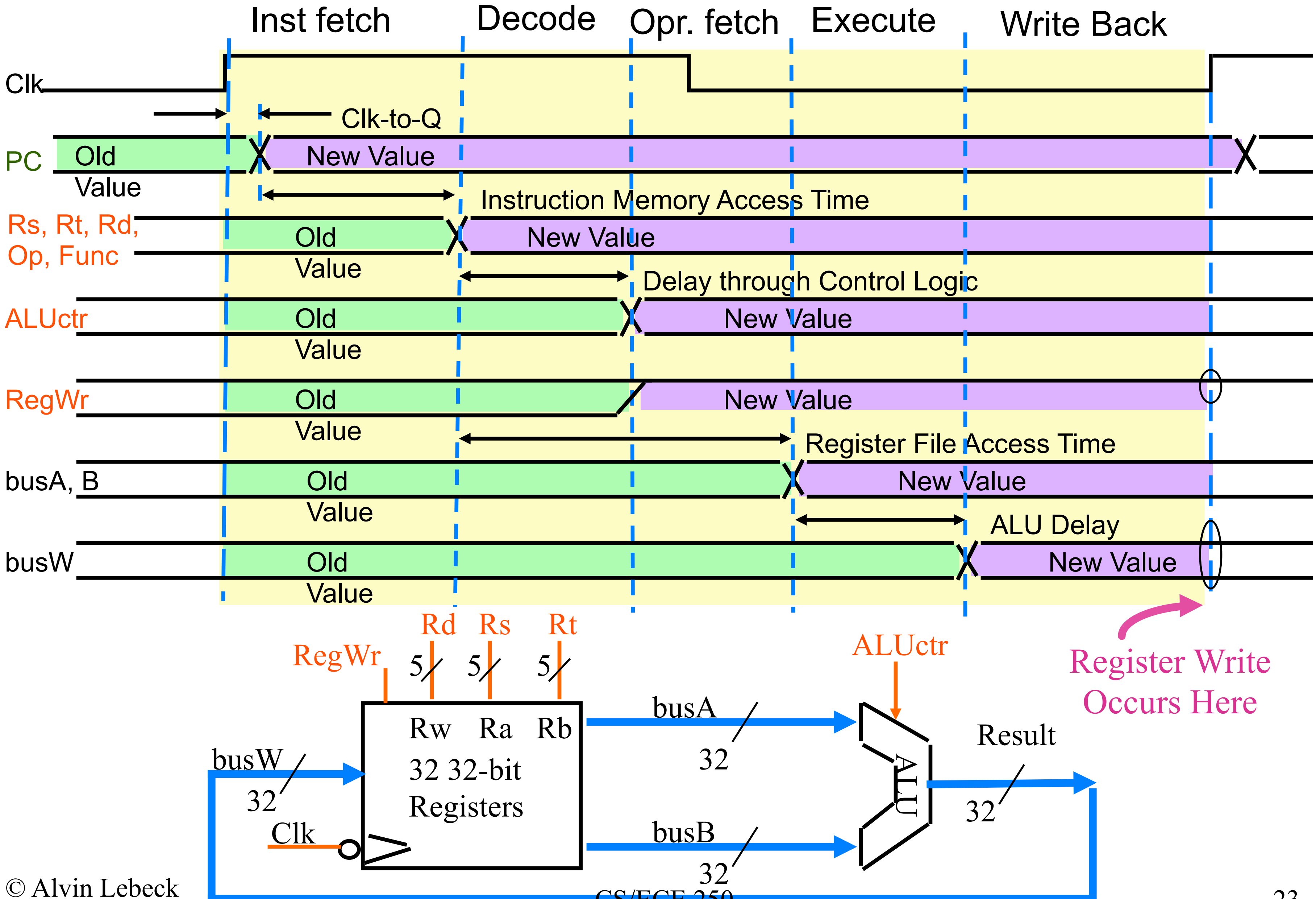
Datapath for Register-Register Operations

- R[rd] <- R[rs] 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

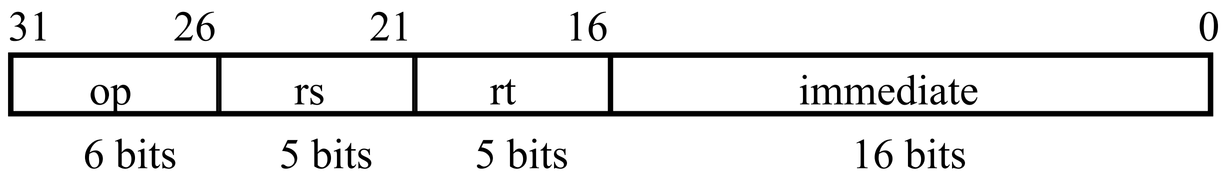
31–26	25–21	20–16	15–11	10–6	5–0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits



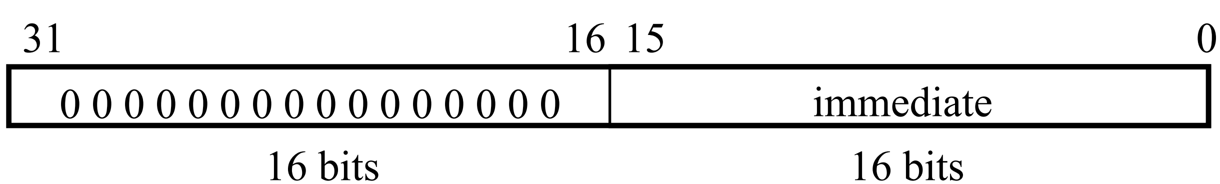
Register-Register Timing



RTL: The OR Immediate Instruction



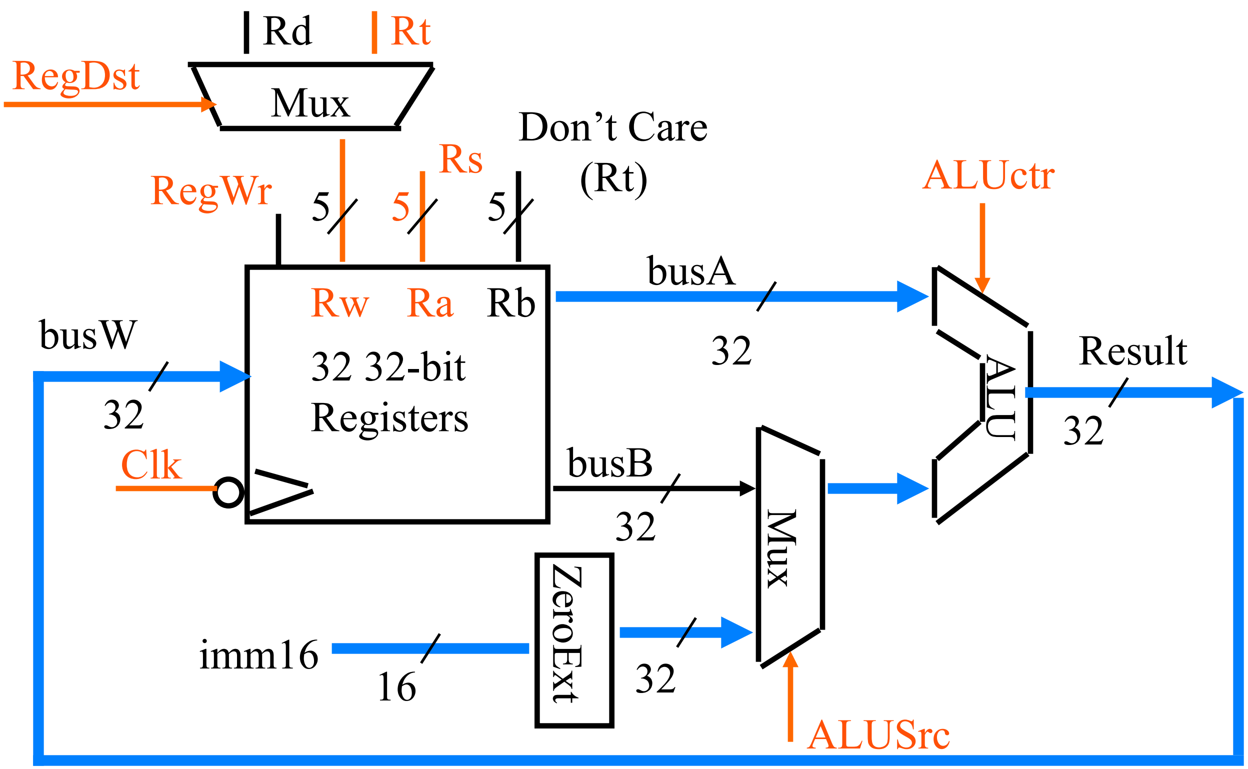
- **ori rt, rs, imm16**
  - **mem[PC]** **Fetch the instruction from memory**
  - **R[rt] <- R[rs] or ZeroExt(imm16)** **The OR operation**
  - **PC <- PC + 4** **Calculate the next instruction's address**



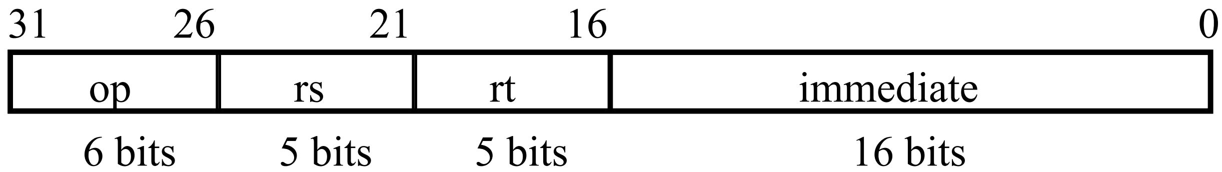
Datapath for Logical Operations with Immediate

- R[rt] <- R[rs] op ZeroExt[imm16]   Example: ori rt, rs, imm16

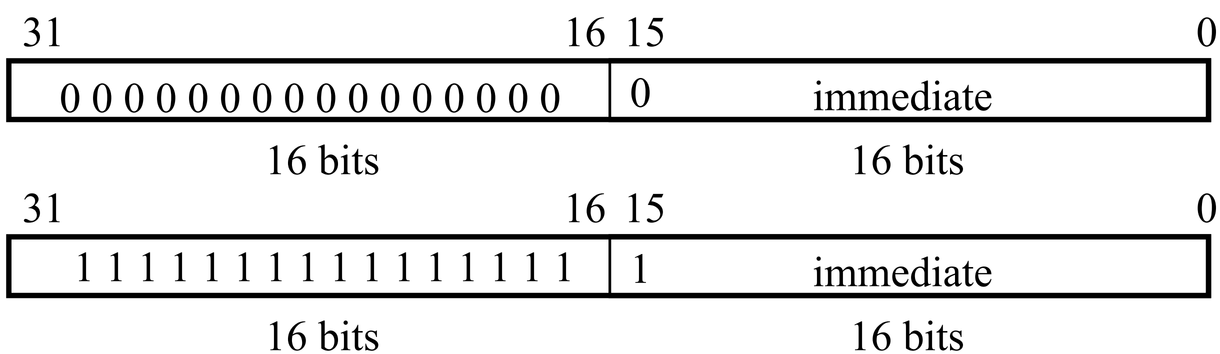
31 … 26	21	16	0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits



RTL: The Load Instruction



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

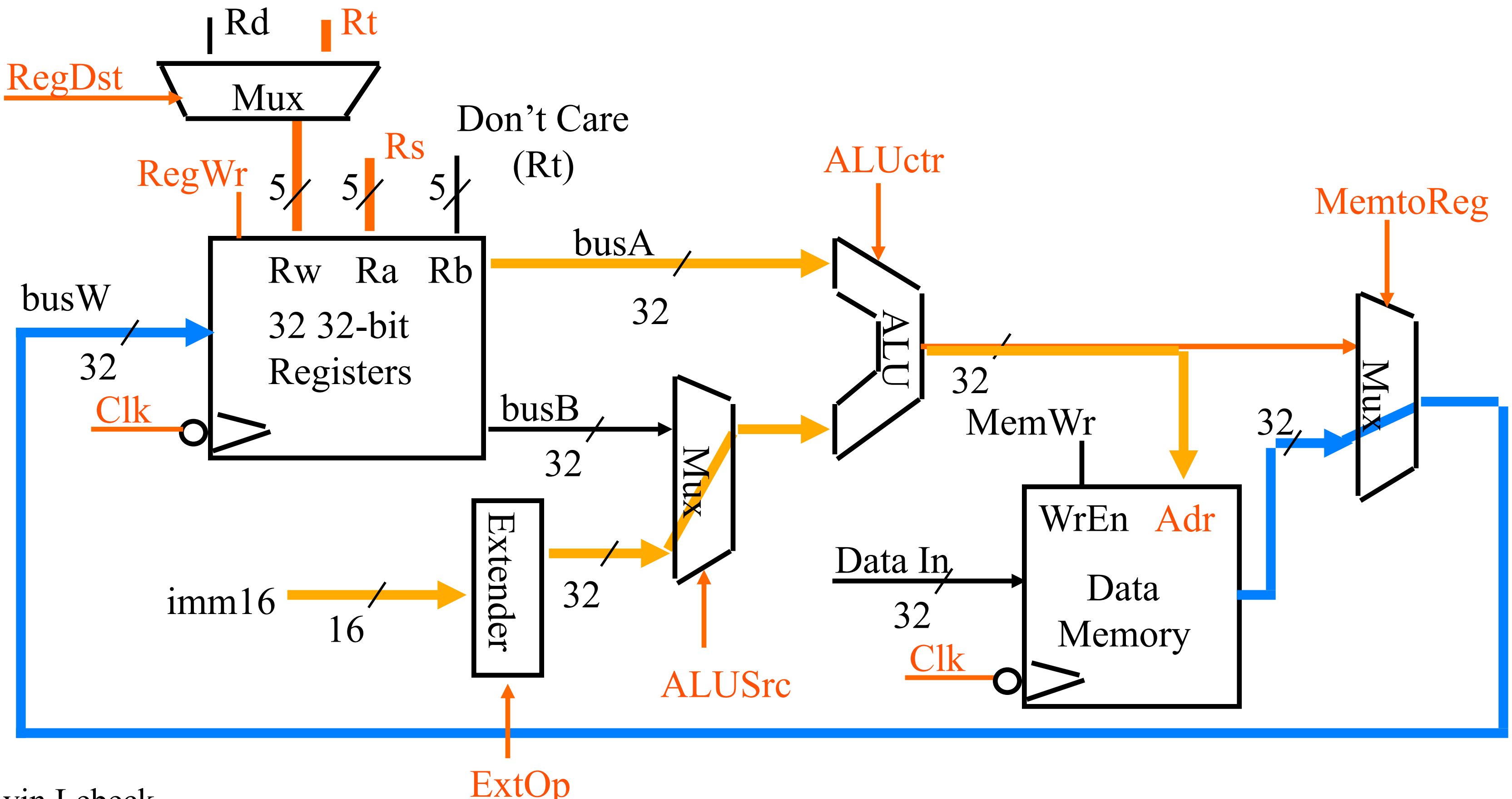


Datapath for Load Operations

- R[rt] <- Mem[R[rs] + SignExt[imm16]]

Example: lw rt, rs, imm16

31 op (6 bits) 26	rs (5 bits) 21	rt (5 bits) 16	immediate (16 bits) 0



RTL: The Store Instruction

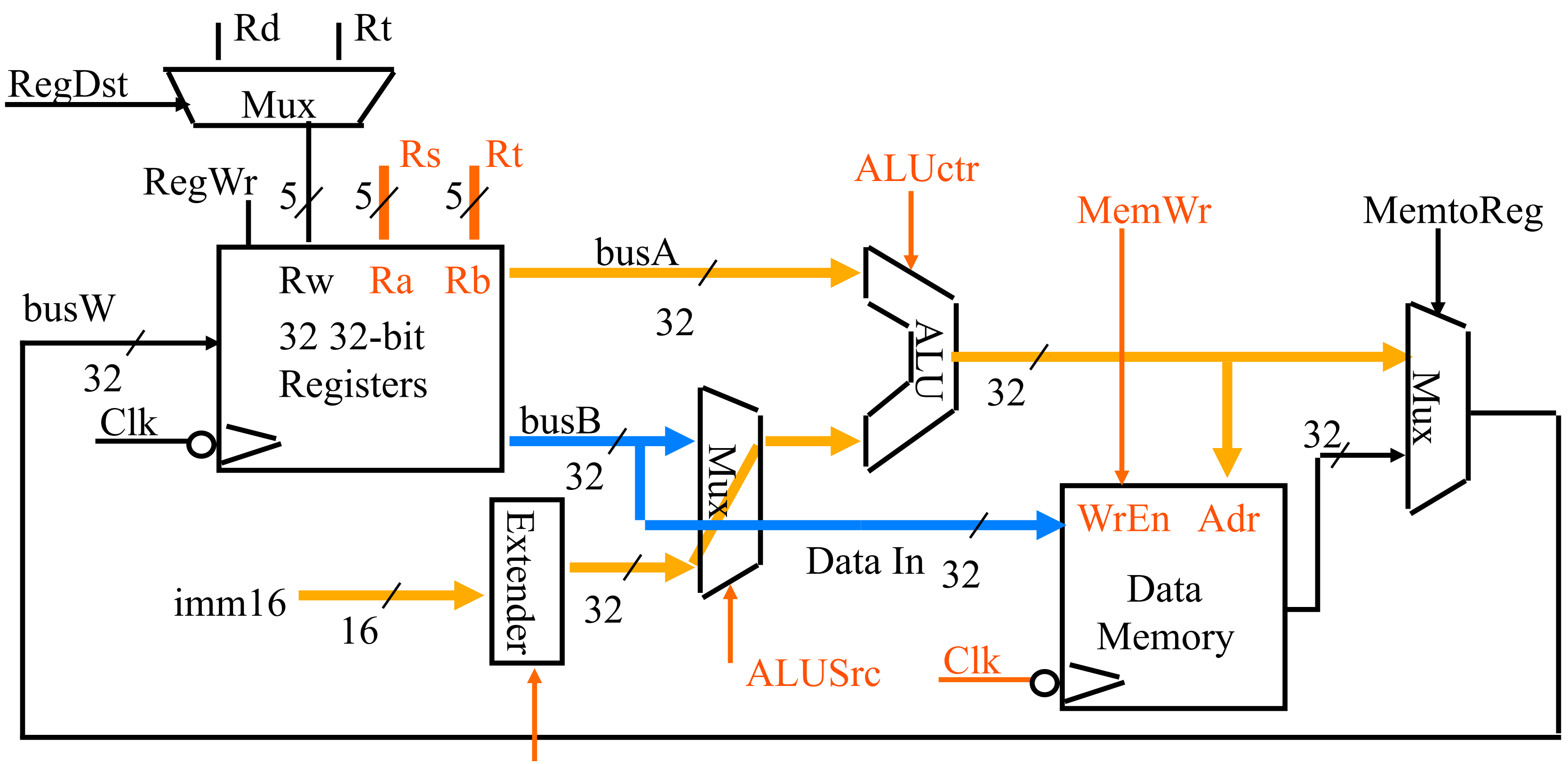
31	26	21	16	0
op	rs	rt	immediate	
6 bits	5 bits	5 bits	16 bits	

- **sw rt, rs, imm16**
  - mem[PC] — **Fetch the instruction from memory**
  - Address <- R[rs] + SignExt(imm16) — **Calculate the memory address**
  - Mem[Address] <- R[rt] — **Store the register into memory**
  - PC <- PC + 4 — **Calculate the next instruction's address**

Datapath for Store Operations

- Mem[R[rs] + SignExt[imm16] <- R[rt]]   Example: sw   rt, rs, imm16

31 – 26	26 – 21	21 – 16	16 – 0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits



RTL: The Branch Instruction

31–26	25–21	20–16	15–0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- **beq rs, rt, imm16**
  - **mem[PC]** — **Fetch the instruction from memory**
  - **Cond <- R[rs] - R[rt]** — **Calculate the branch condition**
  - **if (COND eq 0)** — **Calculate the next instruction's address**

    PC <- PC + 4 + (SignExt(imm16) x 4)

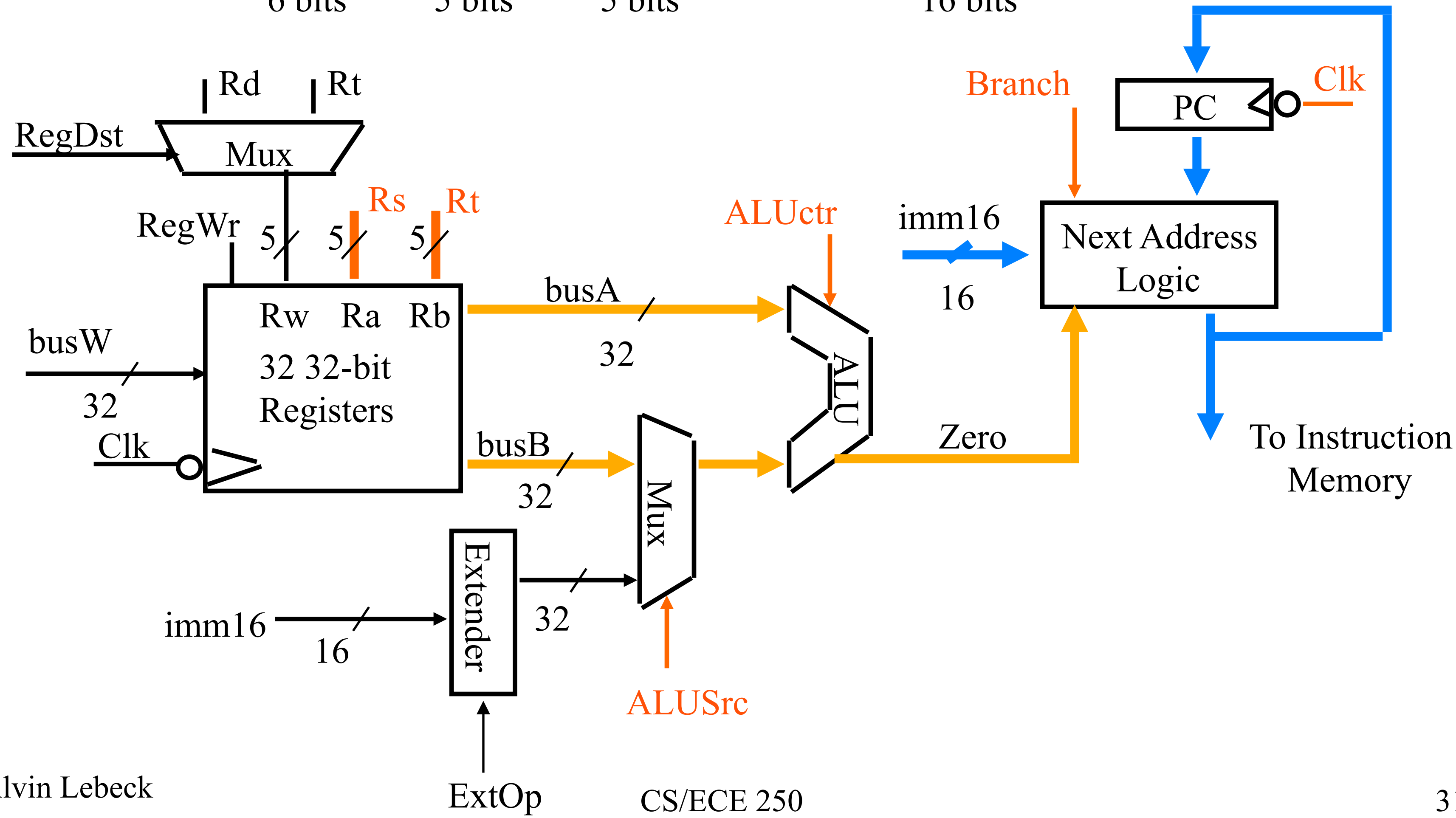
    else

    PC <- PC + 4

Datapath for Branch Operations

- **beq rs, rt, imm16** **We need to compare Rs and Rt!**

31 op (6 bits) 26	rs (5 bits) 21	rt (5 bits) 16	immediate (16 bits) 0

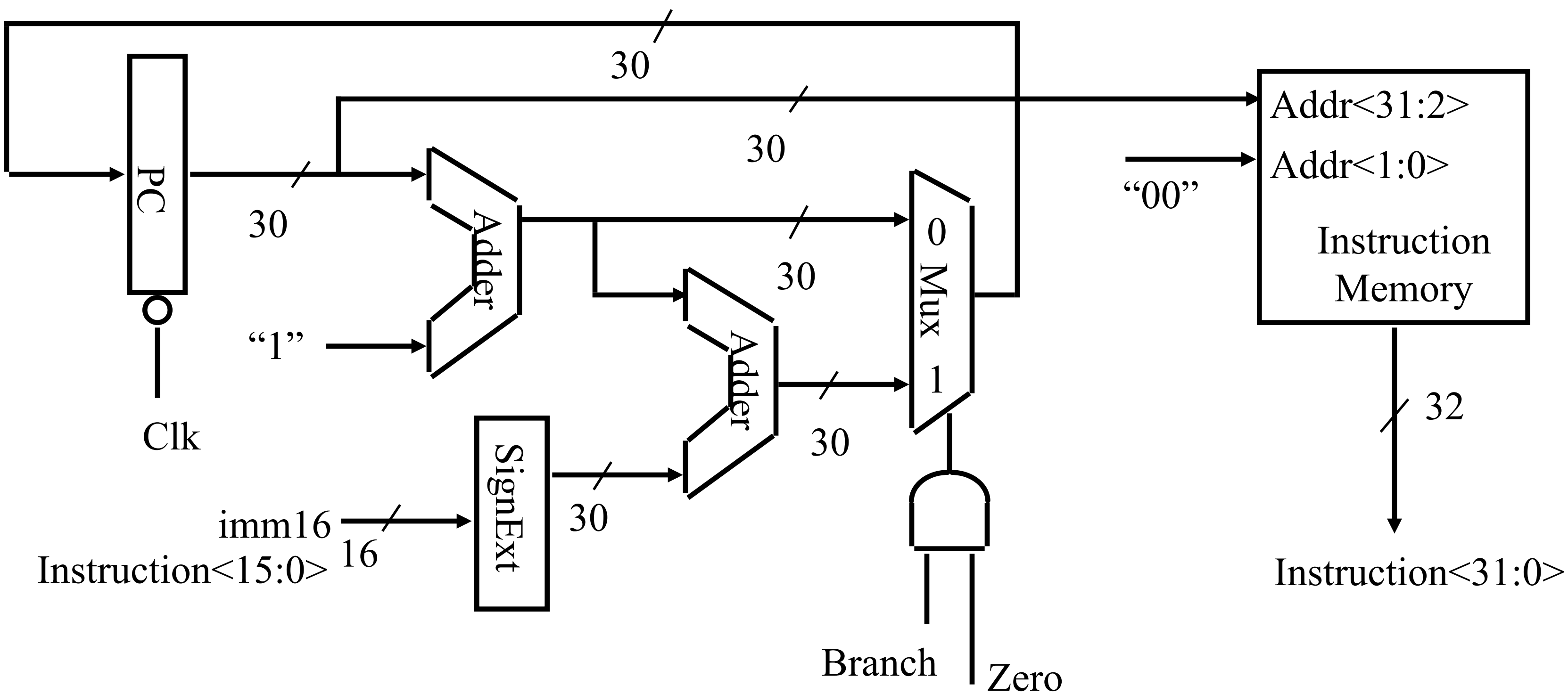


Binary Arithmetic for the Next Address

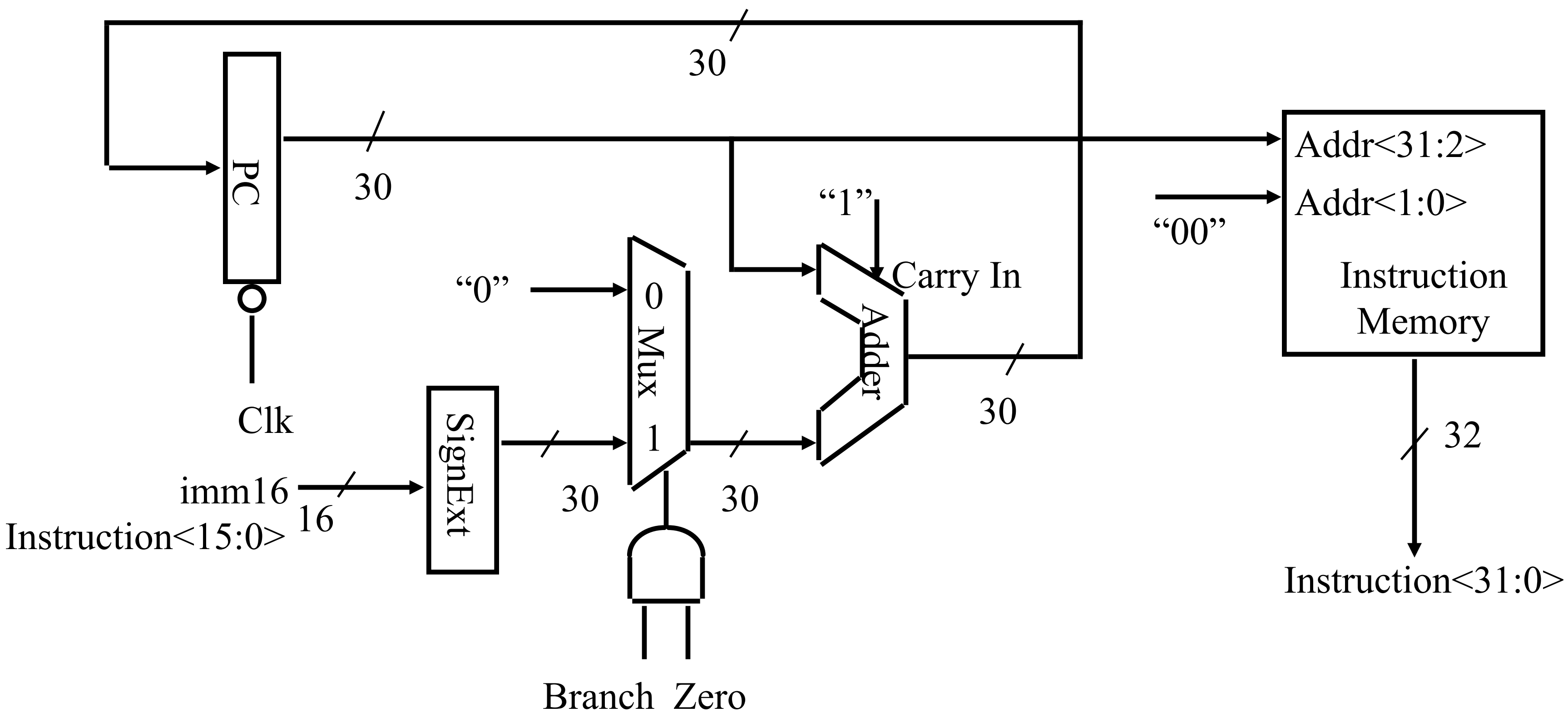
- In theory, the PC is a 32-bit byte address into the instruction memory:
  - Sequential operation: PC<31:0> = PC<31:0> + 4
  - Branch operation: PC<31:0> = PC<31:0> + 4 + SignExt[Imm16] * 4
- The magic number "4" always comes up because:
  - The 32-bit PC is a byte address
  - And all our instructions 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 keep the 2 LSBs
- In practice, we can simplify the hardware by using a 30-bit PC<31:2>:
  - Sequential operation: PC<31:2> = PC<31:2> + 1
  - Branch operation: PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]
  - In either case: Instruction-Memory-Address = PC<31:2> concat "00"
    - Concat means join using a splitter backward…

Next Address Logic: Expensive and Fast Solution

- **Using a 30-bit PC:**
  - **Sequential operation:** PC<31:2> = PC<31:2> + 1
  - **Branch operation:** PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]
  - **In either case:** Instruction-Memory-Address = PC<31:2> concat “00”



Next Address Logic



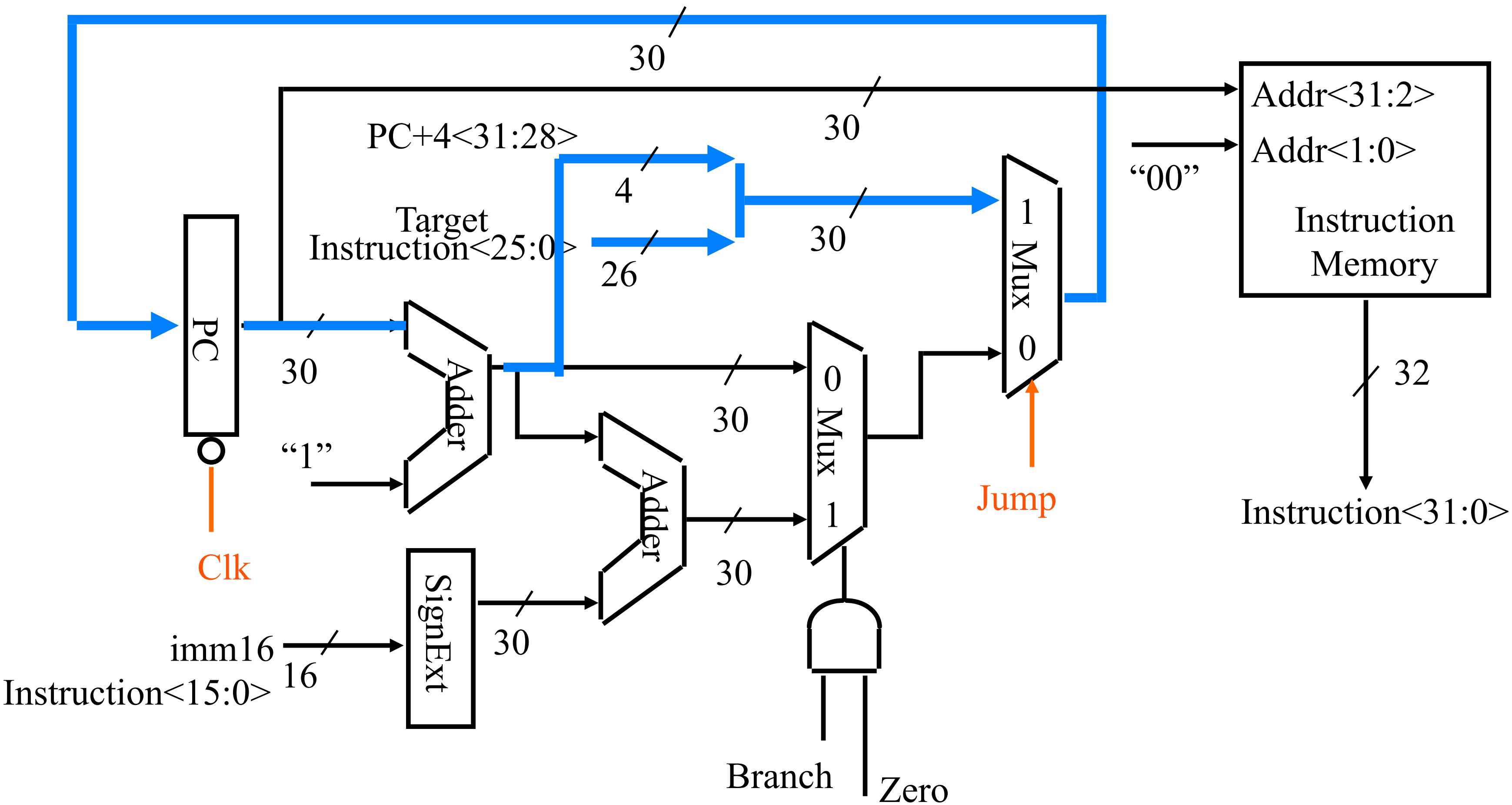
RTL: The Jump Instruction

31 26	0
op	target address
6 bits	26 bits

- **j target**
  - **mem[PC]** **Fetch the instruction from memory**
  - **PC <- PC+4<31:28> concat target<25:0> concat <00>**
    Calculate the next instruction's address

Instruction Fetch Unit

- **j target**
  - PC<31:2> <- PC+4<31:28> concat target<25:0>



Putting it All Together: A Single Cycle Datapath

- We have everything but the **control signals.**