inst.eecs.berkeley.edu/~cs61c

# CS61C : Machine Structures

## Lecture 27 – Single Cycle CPU Datapath, with Verilog II

**2004-11-01**

**Lecturer PSOE Dan Garcia**

www.cs.berkeley.edu/~ddgarcia

**Another shutout for Cal!** ⇒

**Unbelievable! The #4 Bears were dominant in beating ASU 27-0. JJ Arrington shatters Cal records w/his 7th-straight 100yd game, becoming the fastest Cal player ever to reach 1,000 yds. It's ASU's 1st shutout loss in 9 yrs & our first time in the top 5 in 52 years!! OU next Sat…**

calbears.com

# Why is it "memArray[address[9:2]]"?

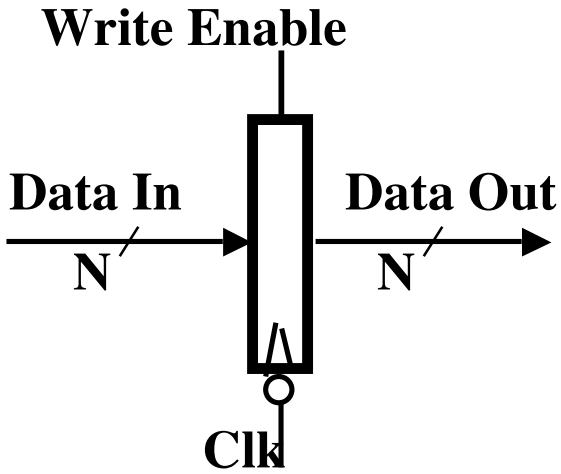
- Our memory is always byte-addressed
  - We can `lb` from 0x0, 0x1, 0x2, 0x3, …
- `lw` only reads word-aligned requests
  - We only call `lw` with 0x0, 0x4, 0x8, 0xC, …
  - I.e., the last two bits are always 0
- memArray is a word wide and $2^8$ deep
  - `reg [31:0] memArray [0:256-1];`
  - Size = 4 Bytes/row * 256 rows = 1024 B
  - If we're simulating `lw/sw`, we R/W words
  - What bits select the first 256 words? [9:2]!
  - 1st word = 0x0 = 0b000 = memArray[0];
    2nd word = 0x4 = 0b100 = memArray[1], etc.

# How to Design a Processor: step-by-step

- 1. Analyze instruction set architecture (ISA) => datapath requirements
  - meaning of each instruction is given by the *register transfers*
  - datapath must include storage element for ISA registers
  - datapath must support each register transfer
- 2. Select set of datapath components and establish clocking methodology
- 3. Assemble datapath meeting requirements
- 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
- 5. Assemble the control logic (hard part!)

# Storage Element: Register (Building Block)

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



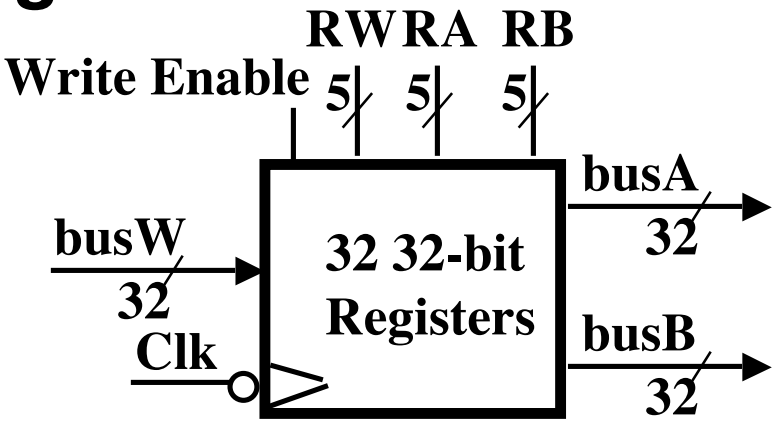

# Verilog 32-bit Register for MIPS Interpreter

```
// Behavioral model of 32-bit Register:
// positive edge-triggered,
// synchronous active-high reset.
module reg32 (CLK,Q,D,wEnb);
    input  CLK, wEnb;
    input  [31:0] D;
    output [31:0] Q;
    reg    [31:0] Q;

    always @ (posedge CLK)
      if (wEnb)
        Q = D;
endmodule // reg32
```

# 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 (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) 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."

# Verilog Register File for MIPS Interpreter (1/3)

```
// Behavioral model of register file:
// 32-bit wide, 32 words deep,
// two asynchronous read-ports,
// one synchronous write-port.
// Dump register file contents to
// console on pos edge of dump signal.
```

# Verilog Register File for MIPS Interpreter (2/3)

```
module regFile (CLK, wEnb, DMP,
 writeReg, writeD, readReg1, readD1,
 readReg2, readD2);
   input CLK, wEnb, DMP;
   input [4:0] writeReg, readReg1,
                 readReg2;
   input [31:0] writeD;
   output [31:0] readD1, readD2;
   reg [31:0] readD1, readD2;
   reg [31:0] array [0:31];
   reg dirty1, dirty2;
   integer    i;
```

- 3 5-bit fields to select registers: 1 write register, 2 read register

# Verilog Register File for MIPS Interpreter (3/3)

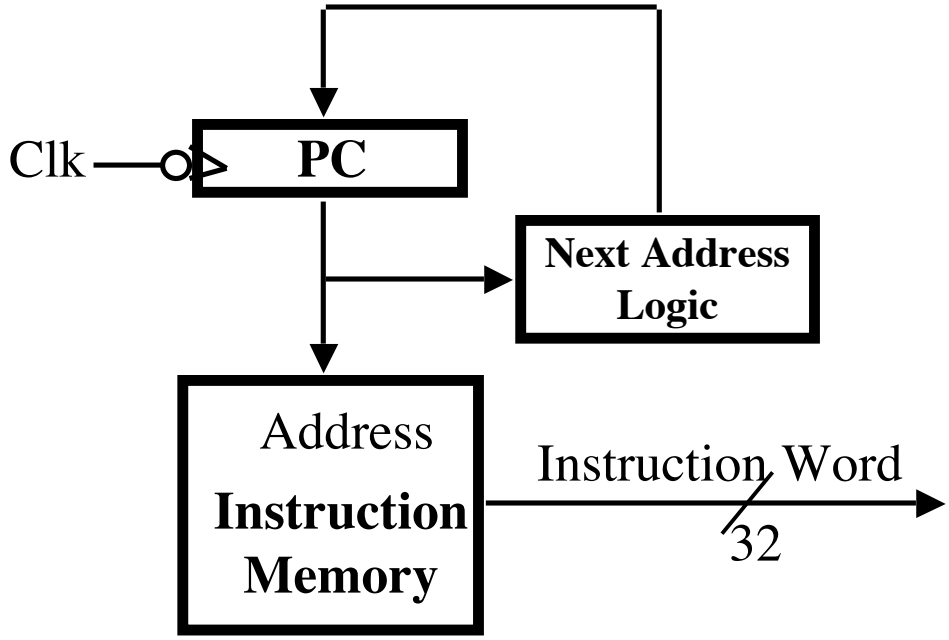
```
always @ (posedge CLK)
      if (wEnb)
        if (writeReg!=5'h0) // why?
          begin
            array[writeReg] = writeD;
            dirty1=1'b1;
            dirty2=1'b1;
          end
always @ (readReg1 or dirty1)
      begin
          readD1 = array[readReg1];
          dirty1=0;
      end
```

# Step 3: Assemble DataPath meeting requirements

- Register Transfer Requirements
  ⇒ Datapath Assembly
- Instruction Fetch
- Read Operands and Execute Operation

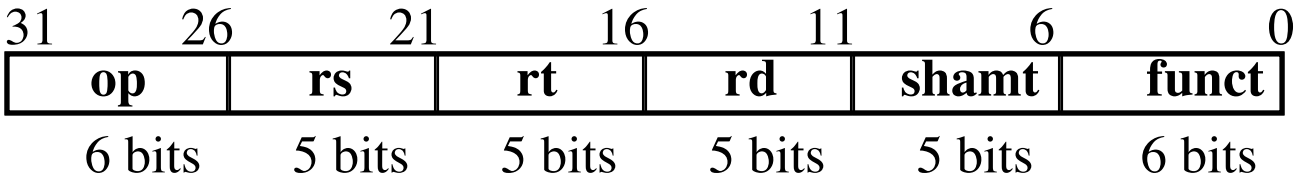
# 3a: Overview of the Instruction Fetch Unit

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

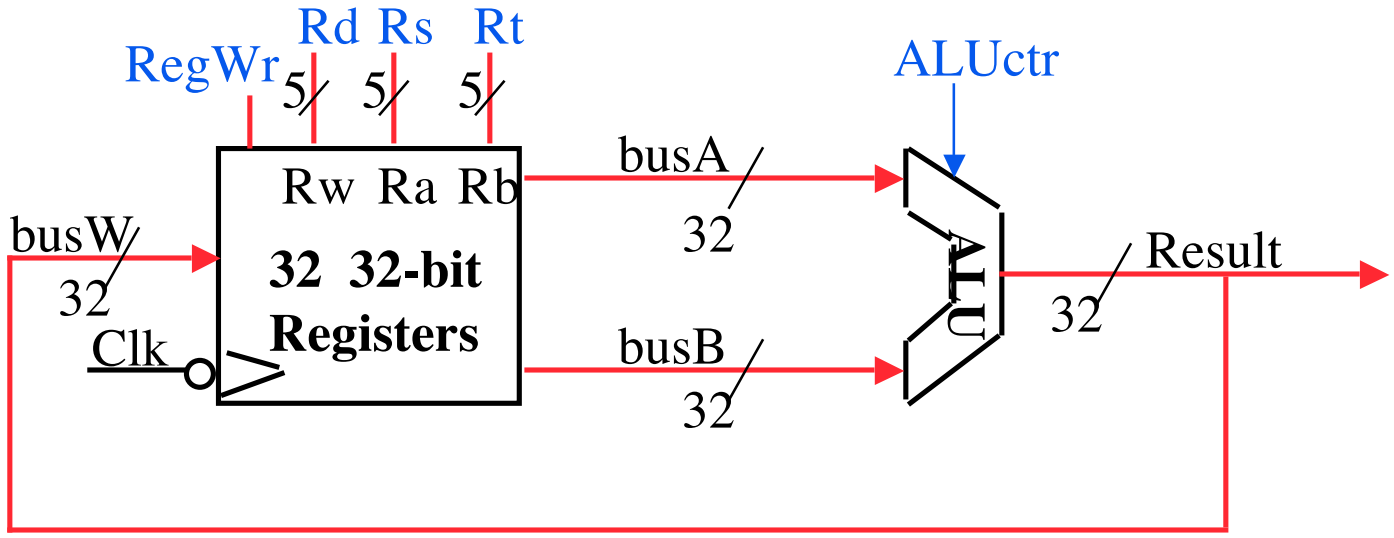



# 3b: Add & Subtract

- R[rd] = R[rs] op R[rt]  Ex.: `addU rd,rs,rt`
  - Ra, Rb, and Rw come from instruction's Rs, Rt, and Rd fields

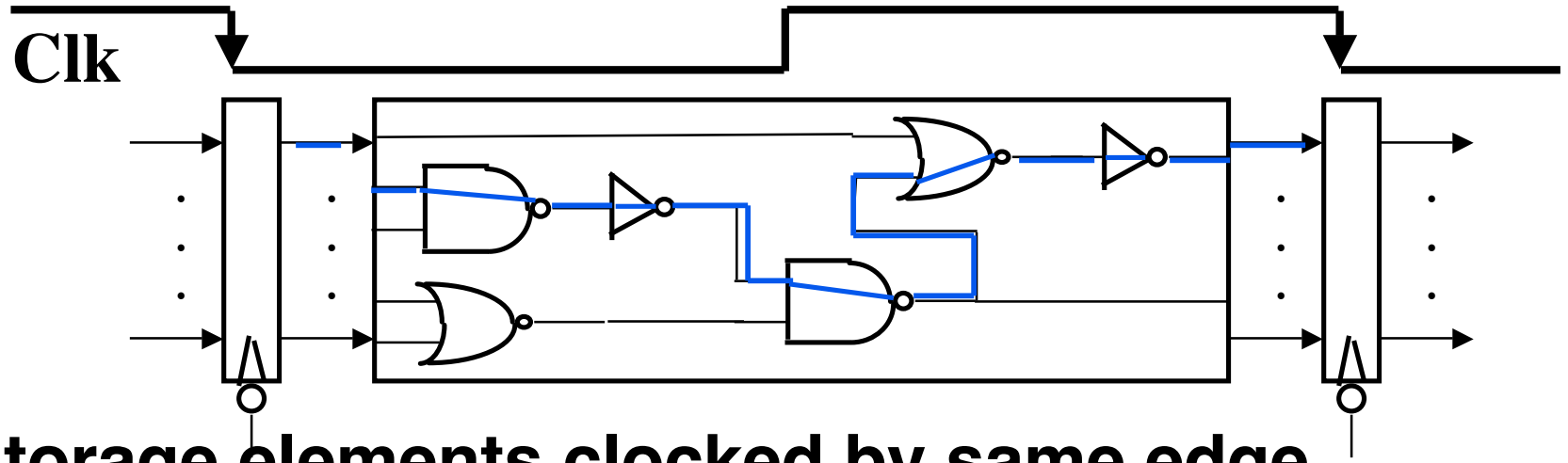



  - ALUctr and RegWr: control logic after decoding the instruction



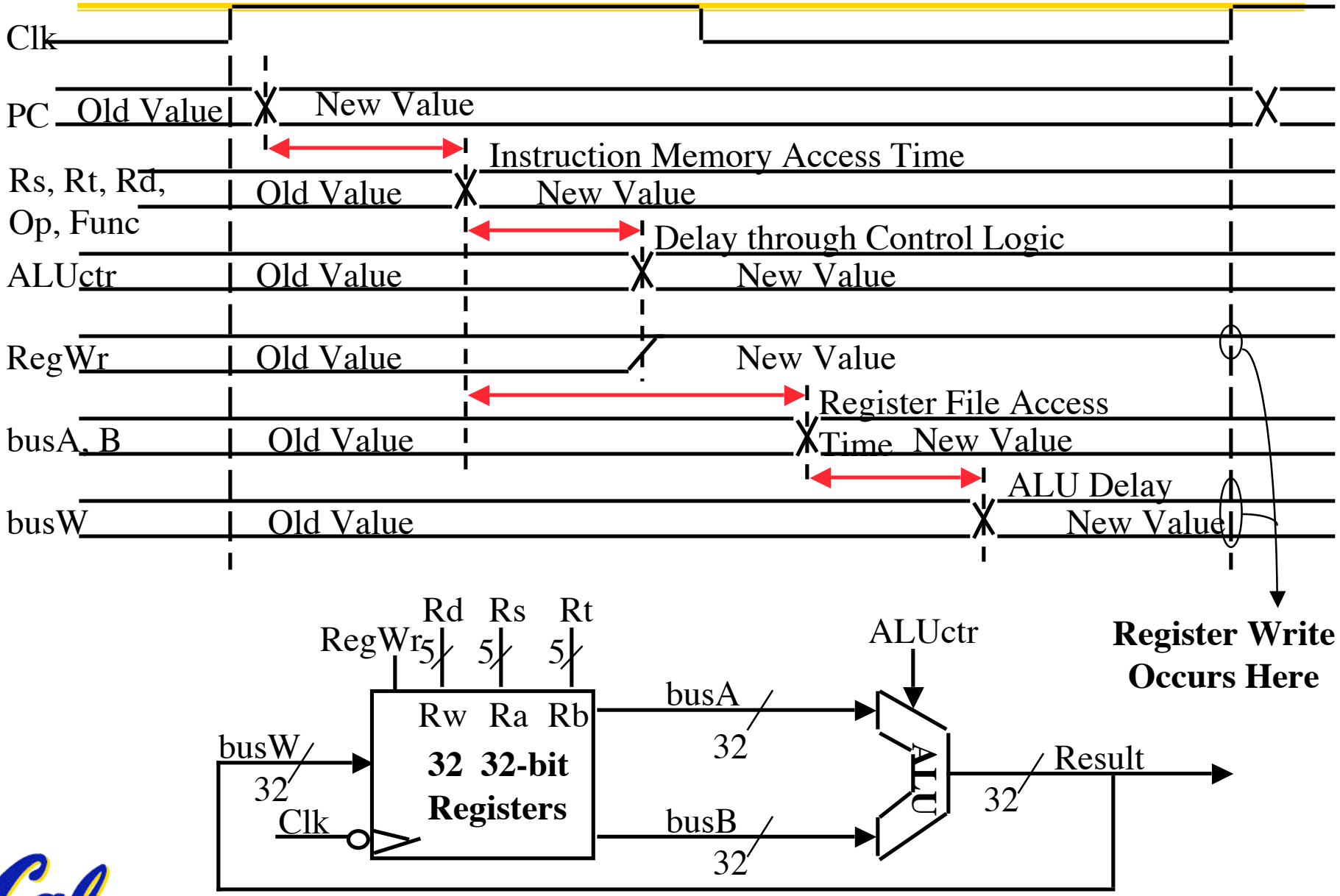

- Already defined register file, ALU

# Clocking Methodology



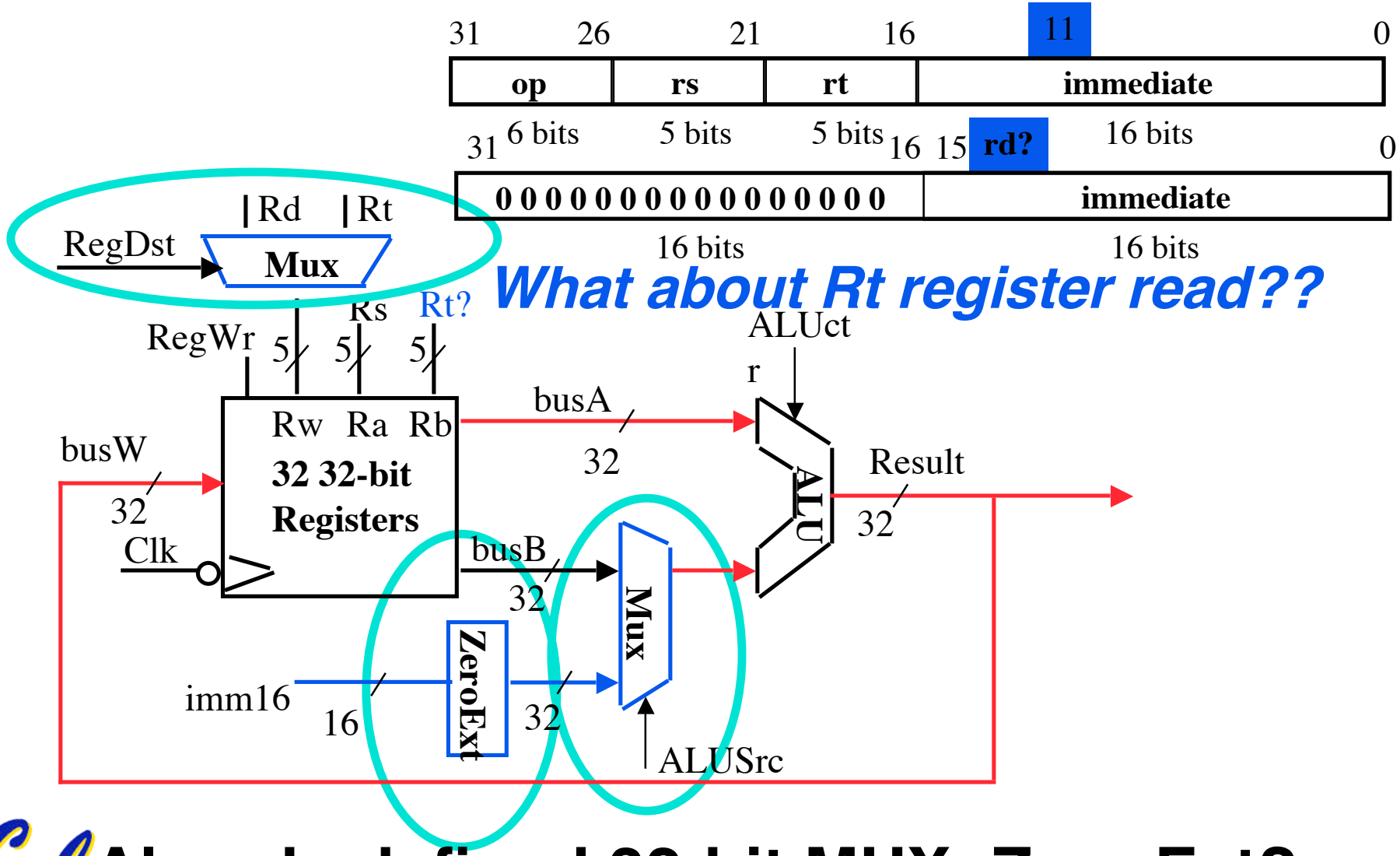

- Storage elements clocked by same edge
- Being physical devices, flip-flops (FF) and combinational logic have some delays
  - Gates: delay from input change to output change
  - Signals at FF D input must be stable before active clock edge to allow signal to travel within the FF, and we have the usual clock-to-Q delay
- "Critical path" (longest path through logic) determines length of clock period

# Register-Register Timing: One complete cycle



# 3c: Logical Operations with Immediate

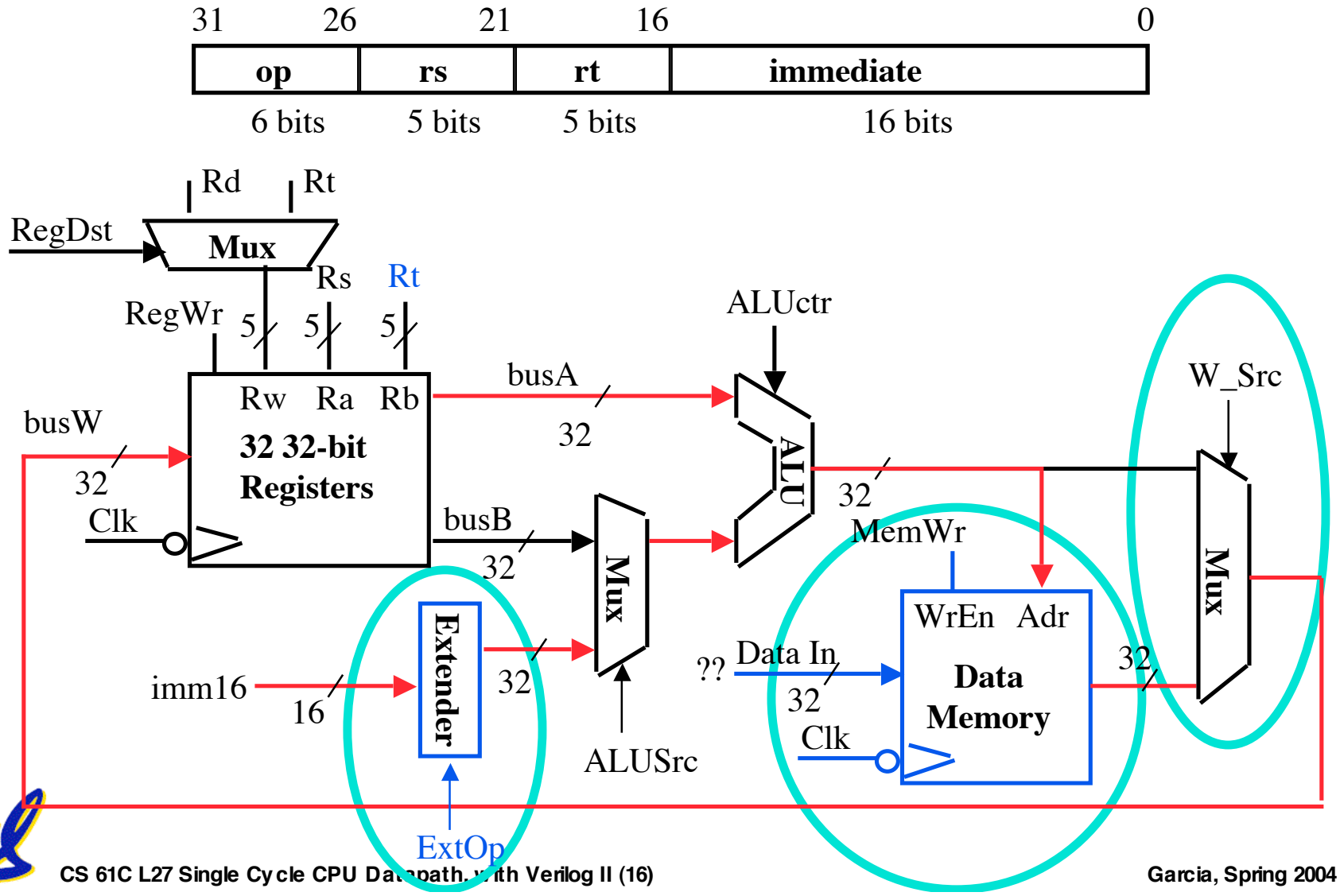
- R[rt] = R[rs] op ZeroExt[imm16] ]




- Already defined 32-bit MUX; Zero Ext?

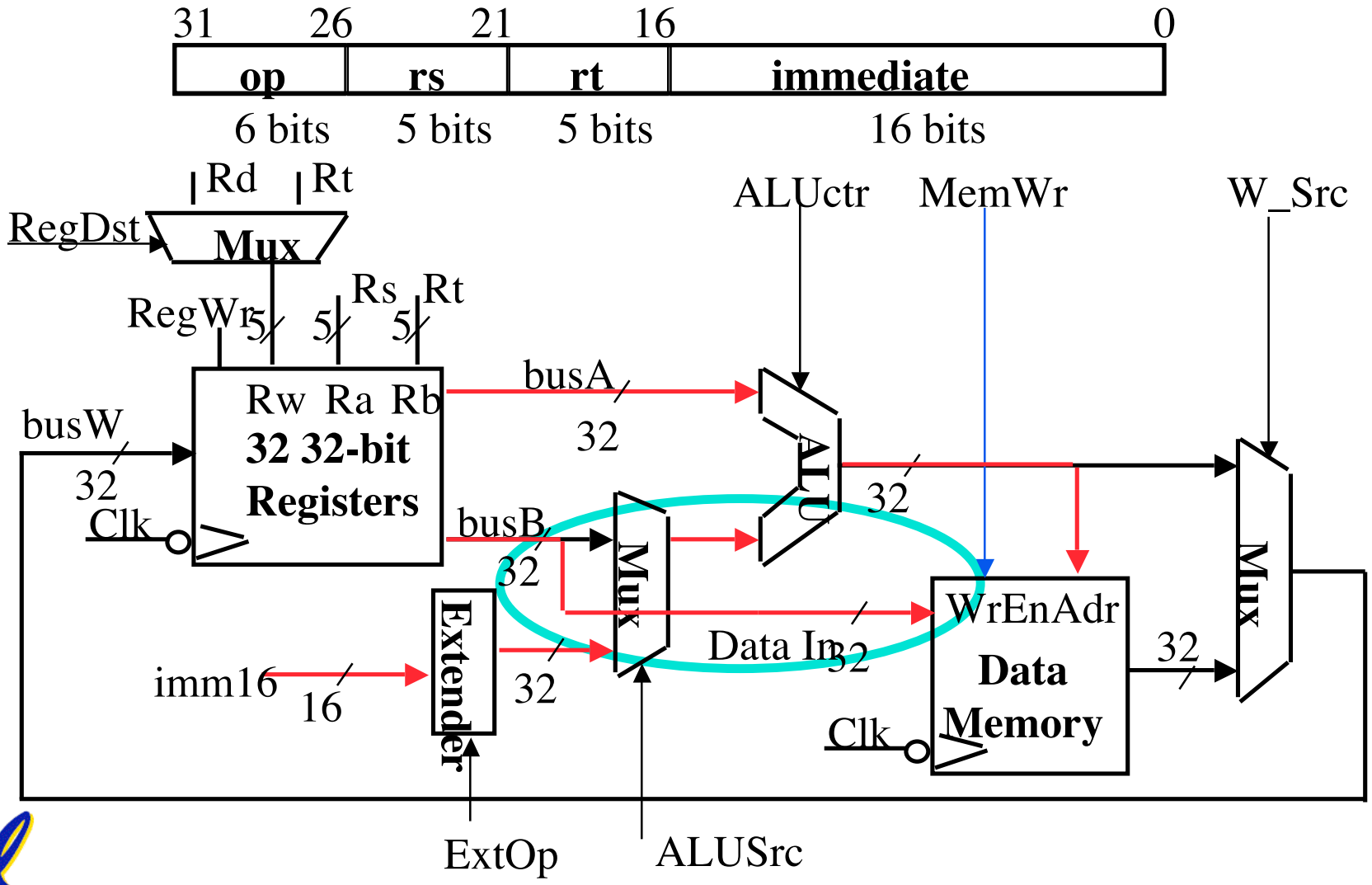
# 3d: Load Operations

- R[rt] = Mem[R[rs] + SignExt[imm16]]
  Example: `lw rt,rs,imm16`

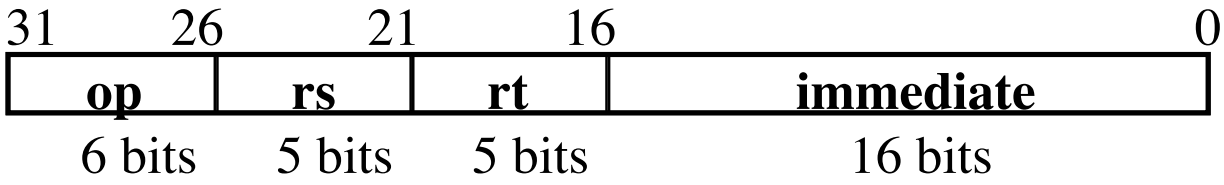


# 3e: Store Operations

- **Mem[ R[rs] + SignExt[imm16] ] = R[rt]**
  **Ex.: `sw rt, rs, imm16`**



# 3f: The Branch Instruction



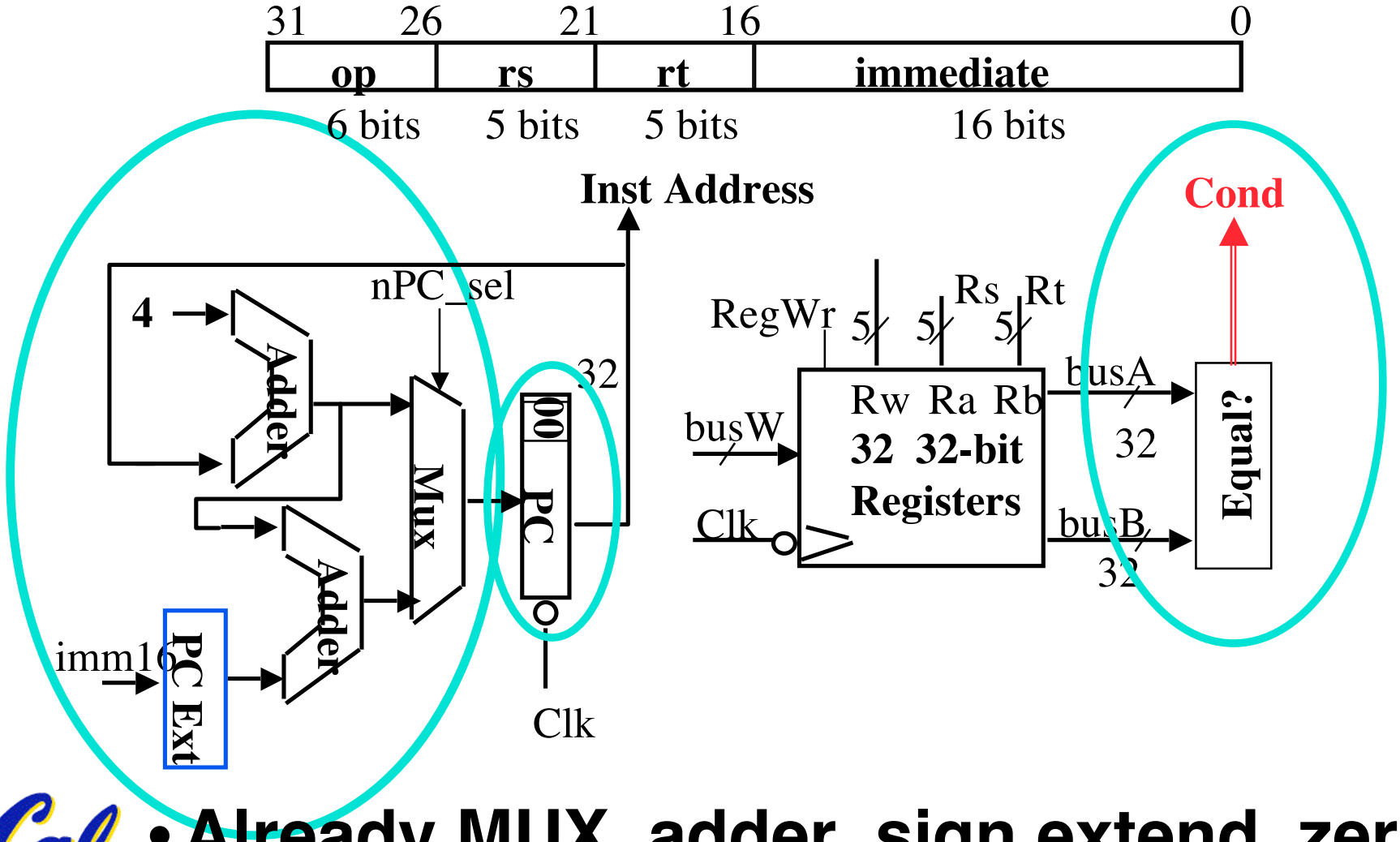

- `beq rs, rt, imm16`
  - mem[PC] Fetch the instruction from memory
  - Equal = R[rs] == R[rt]  Calculate branch condition
  - if (Equal) Calculate the next instruction's address
    - PC = PC + 4 + ( SignExt(imm16) x 4 )

    else
    - PC = PC + 4

# Datapath for Branch Operations

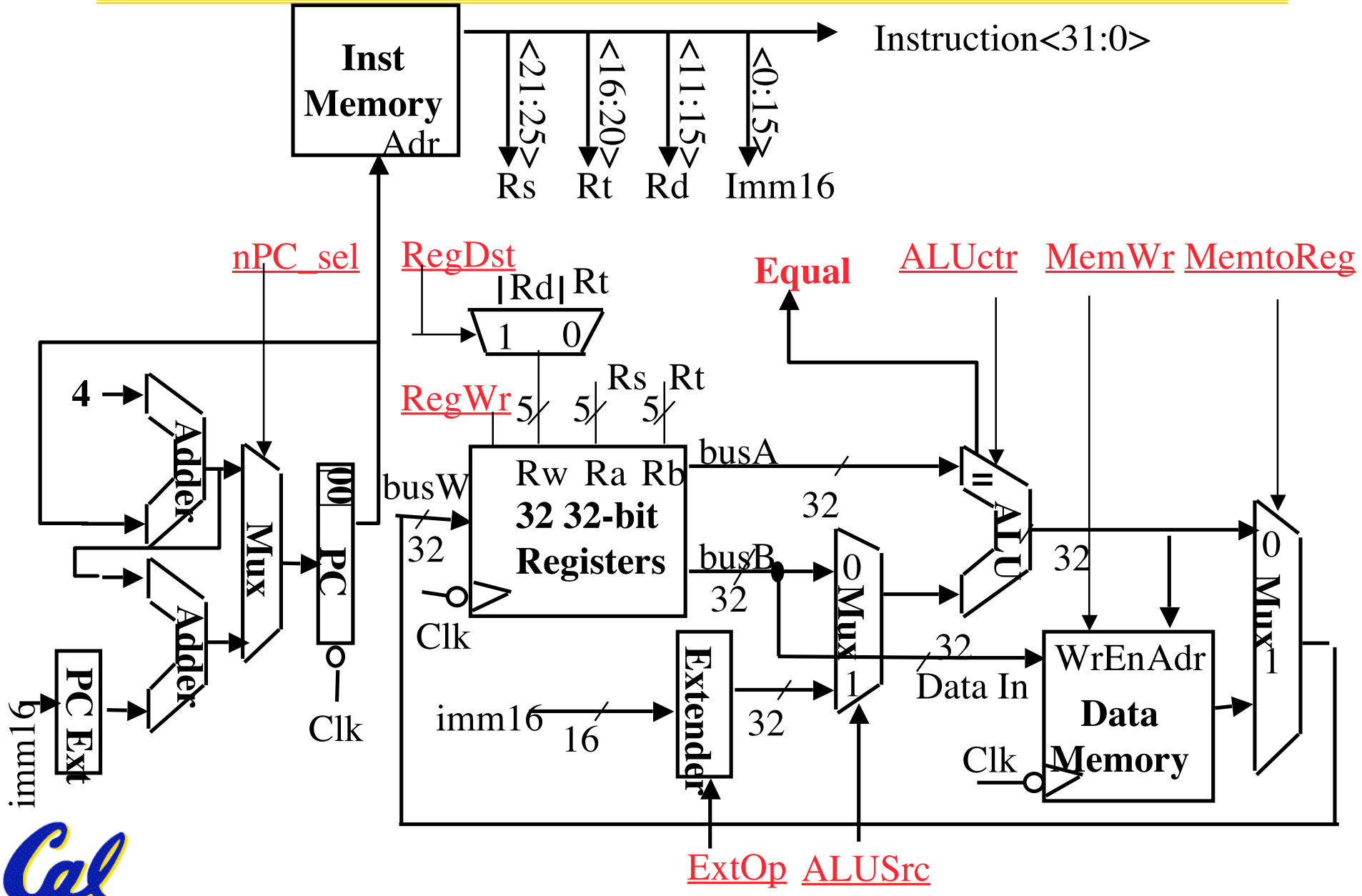
- beq rs, rt, imm16

Datapath generates condition (equal)

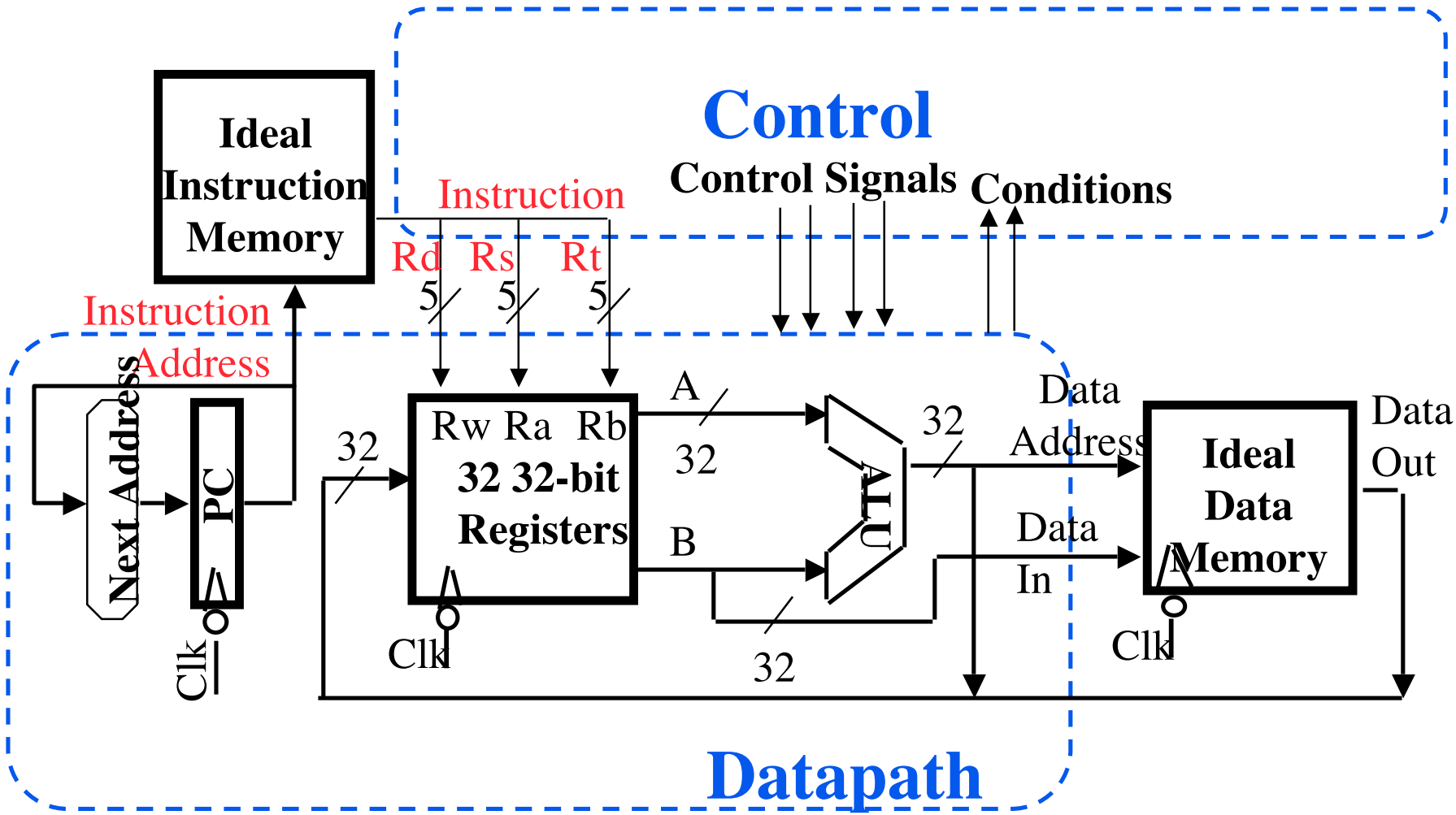



- Already MUX, adder, sign extend, zero

# Putting it All Together:A Single Cycle Datapath



# An Abstract View of the Implementation




# Peer Instruction

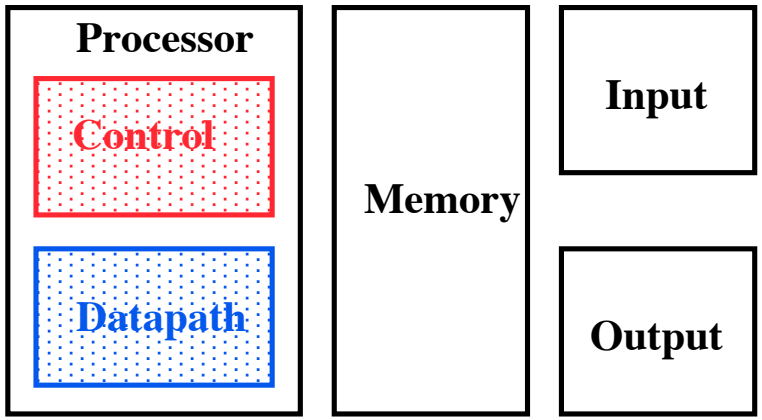
**Suppose we're writing a MIPS interpreter in Verilog. Which sequence below is best organization for the interpreter?**

A. `repeat` loop that fetches instructions
B. `while` loop that fetches instructions
C. Decodes instructions using `case` statement
D. Decodes instr. using chained `if` statements
E. Executes each instruction
F. Increments PC by 4

| | |
|---|---|
| 1: | ACEF |
| 2: | ADEF |
| 3: | AECF |
| 4: | AEDF |
| 5: | BCEF |
| 6: | BDEF |
| 7: | BECF |
| 8: | BEDF |
| 9: | EF |
| 0: | FAE |

# Summary: Single cycle datapath

- 5 steps to design a processor
  - 1. Analyze instruction set => datapath requirements
  - 2. Select set of datapath components & establish clock methodology
  - 3. Assemble datapath meeting the requirements
  - 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  - 5. Assemble the control logic
- Control is the hard part
- Next time!




# Dwarfing the importance of this lecture…

…is the importance that tomorrow you get out and

# VOTE!