CS61C : Machine Structures

Lecture 5.1.2

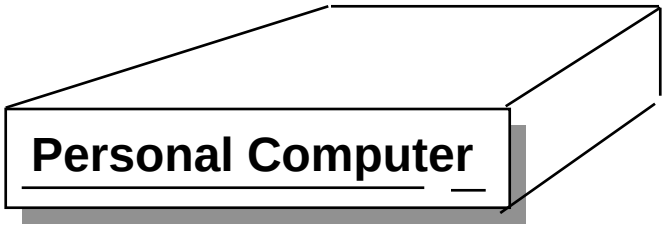
CPU Design II

2004-07-20

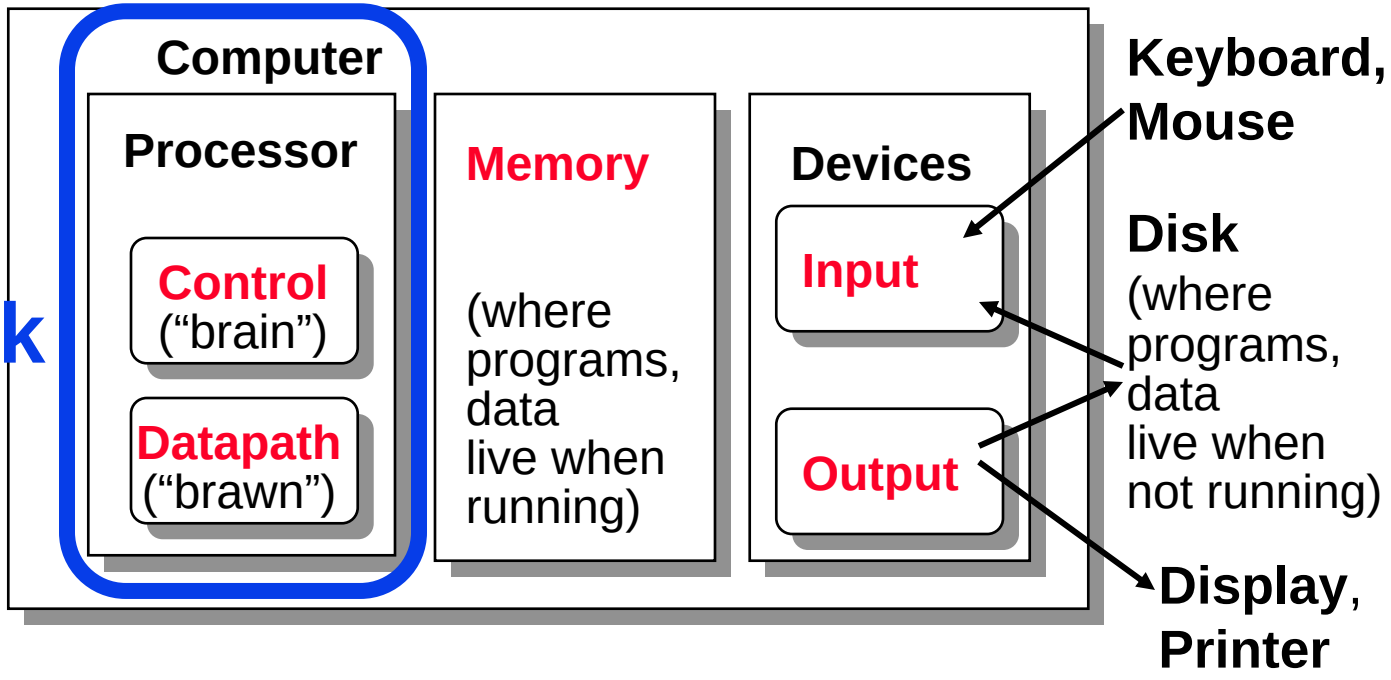
Kurt Meinz

`inst.eecs.berkeley.edu/~cs61c`

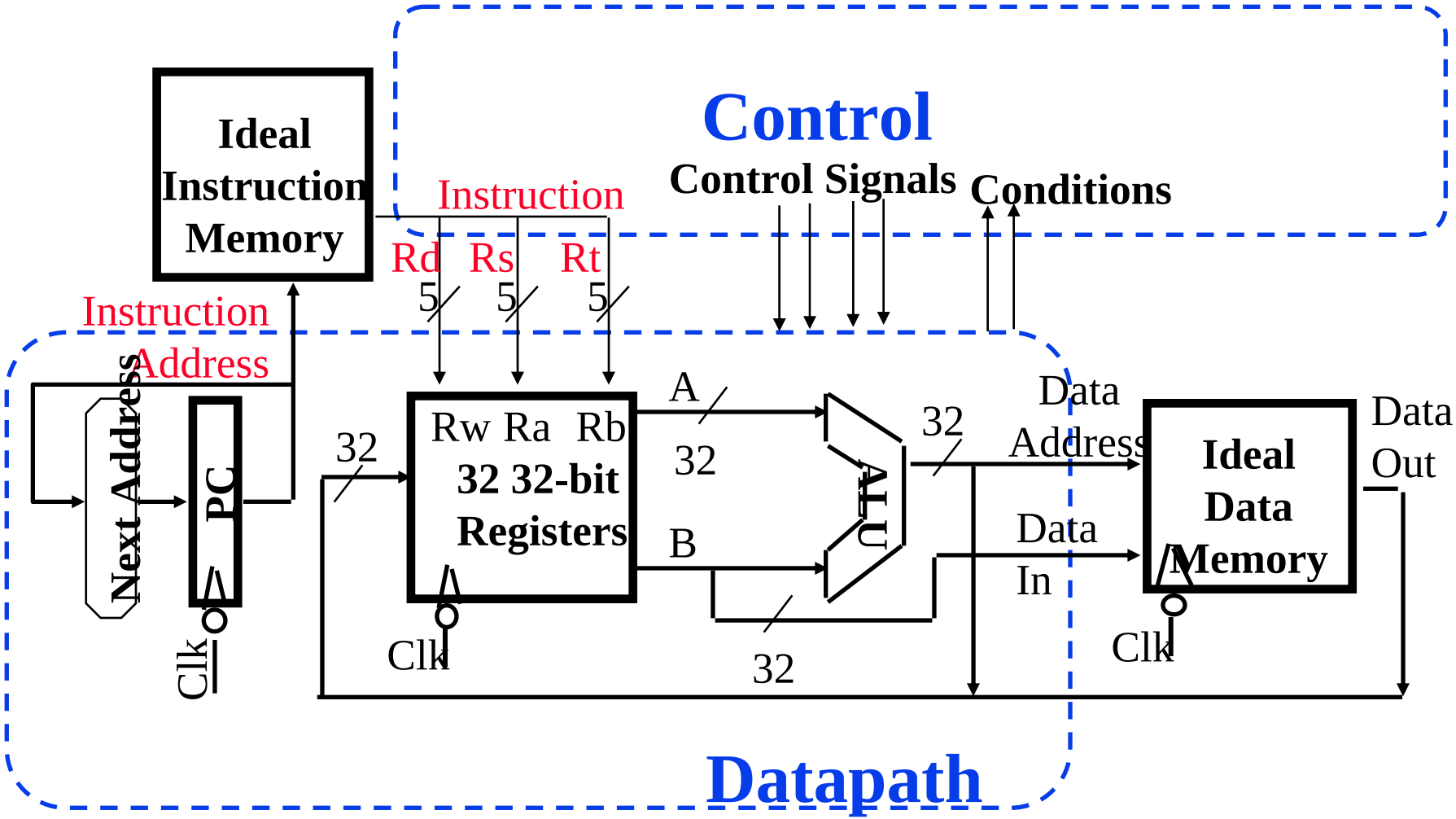
Anatomy: 5 components of any Computer



This week



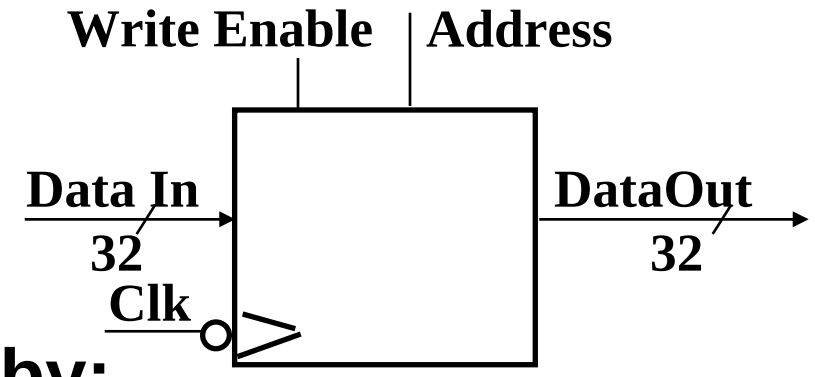
Step 1: Abstract Implementation



Step 2b: Components of the Datapath

- **Combinational Elements**
- **Storage Elements**
  - Clocking methodology

Storage Element: Idealized Memory



- Memory (idealized)
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is selected by:
  - Address selects the word to put on Data Out
  - 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."

Verilog Memory for MIPS Interpreter (1/3)

```
//Behavioral modelof Random Access Memory:
// 32-bit wide, 256 words deep,
// asynchronous read-port if RD=1,
// synchronous write-port if WR=1,
// initialize from hex file ("data.dat")
// on positive edge of reset signal,
// dump to binary file ("dump.dat")
// on positive edge of dump signal.
module mem
 (CLK,RST,DMP,WR,RD,address,writeD,readD);
    input CLK, RST, DMP, WR, RD;
    input [31:0] address, writeD;
    output [31:0] readD;
    reg [31:0] readD;
    parameter  memSize=256;
    reg [31:0] memArray [0:memSize-1];
    integer    chann,i;
```

Verilog Memory for MIPS Interpreter (2/3)

```
integer    chann,i;
always @ (posedge RST)
  $readmemh("data.dat", memArray);

// write if WR & positive clock edge (synchronous)
always @ (posedge CLK)
  if (WR) memArray[address[9:2]] =
                              writeD;

// read if RD, independent of clock (asynchronous)
always @ (address or RD)*
  if (RD)
      readD = memArray[address[9:2]];

endmodule
```

☹See how sneaky sensitivity lists can be!

Use an assign!

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

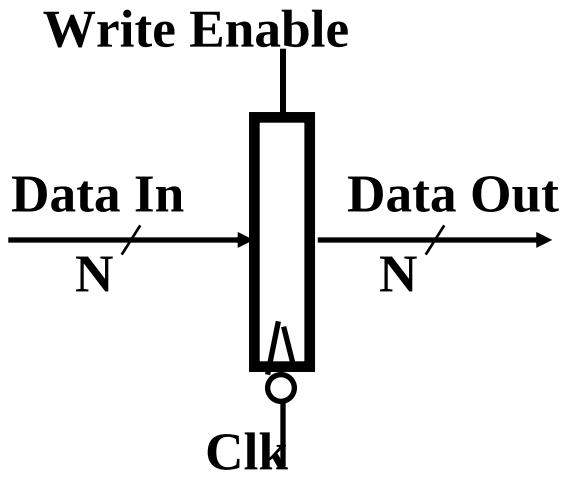
- Our memory is always byte-addressed
  - We can `lb` from 0x0, 0x1, 0x2, 0x3, …
- `lw` only reads word-aligned requests
  - We only call `lw` with 0x0, 0x4, 0x8, 0xC, …
  - I.e., the last two bits are always 0
- memArray is a word wide and 2^8 deep
  - `reg [31:0] memArray [0:256-1];`
  - Size = 4 Bytes/row * 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.

Verilog Memory for MIPS Interpreter (3/3)

```
  end;
  always @ (posedge DMP)
    begin
      chann = $fopen("dump.dat");
      if (chann==0)
        begin
          $display("$fopen of dump.dat
failed.");
          $finish;
        end       // Temp variables chan, i
      for (i=0; i<memSize; i=i+1)
        begin
          $fdisplay(chann, "%b",
                            memArray[i]);
        end
    end // always @ (posedge DMP)
endmodule // mem
```

Storage Element: Register (Building Block)

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



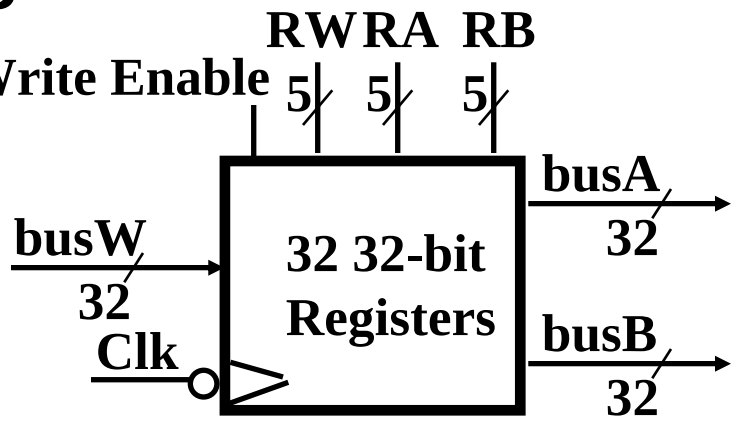
Verilog 32-bit Register

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

    reg [31:0] Q;
    always @ (posedge CLK)
      if (RST) Q = 0; else 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 (1/4)

```
// 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 (2/4)

```
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 (3/4)

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

Verilog Register File (4/4)

Problem 1: dirty1 is awful!!

```
assign readD2 = array[readReg2];
```

Problem 2:

Synchronous reads?

- **must happen on posedge**
- **must get new value if written**

```
always @ (posedge clock)
 if (readReg2 == writeReg)
   readD2 = writeD; //"forwarding"!
 else
   readD2 = array[readReg2];
```

☹Can you see the bug?

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!)

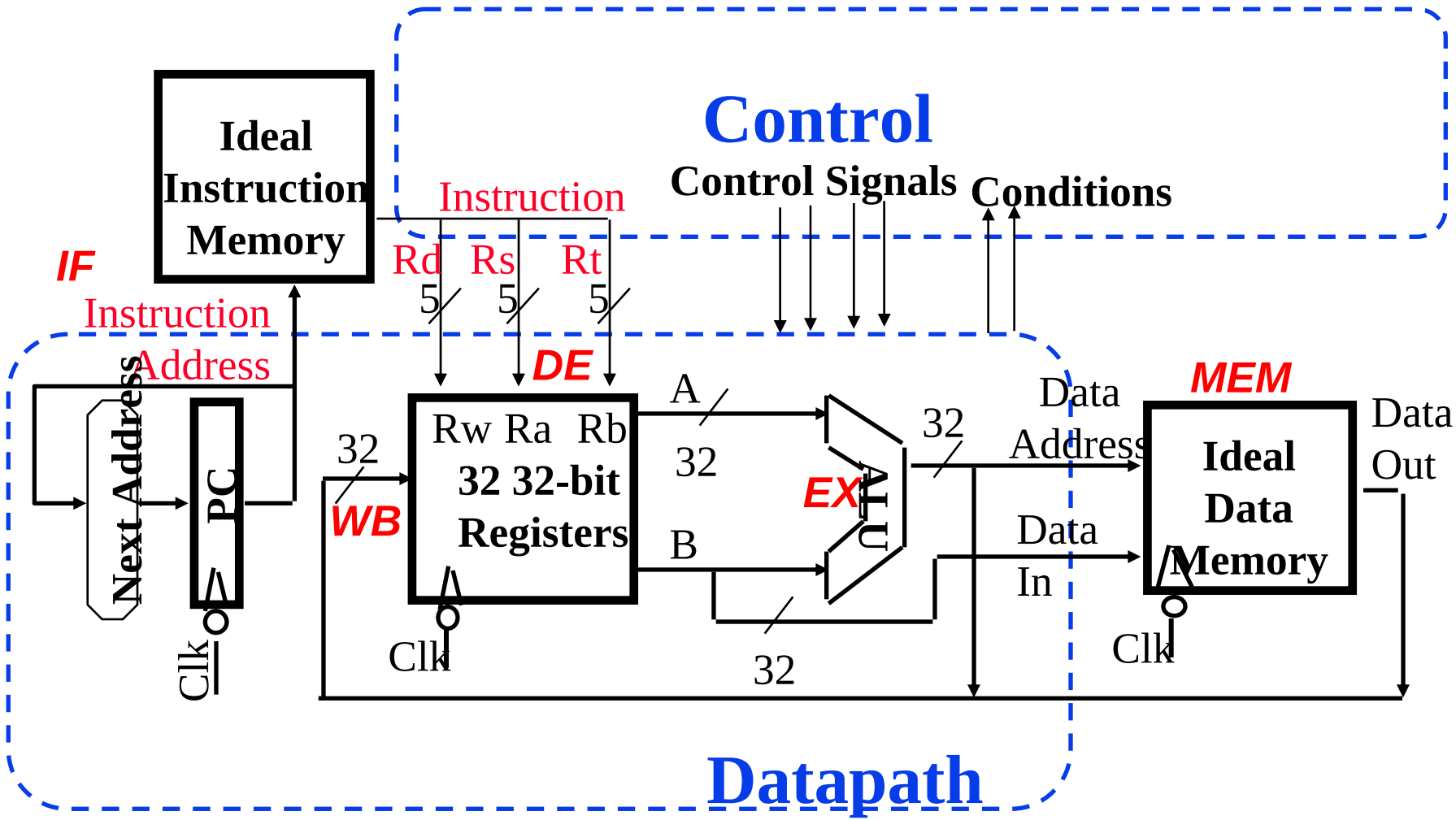
Step 3: Assemble DataPath meeting requirements

- Register Transfer Requirements

  ⇒ Datapath Assembly

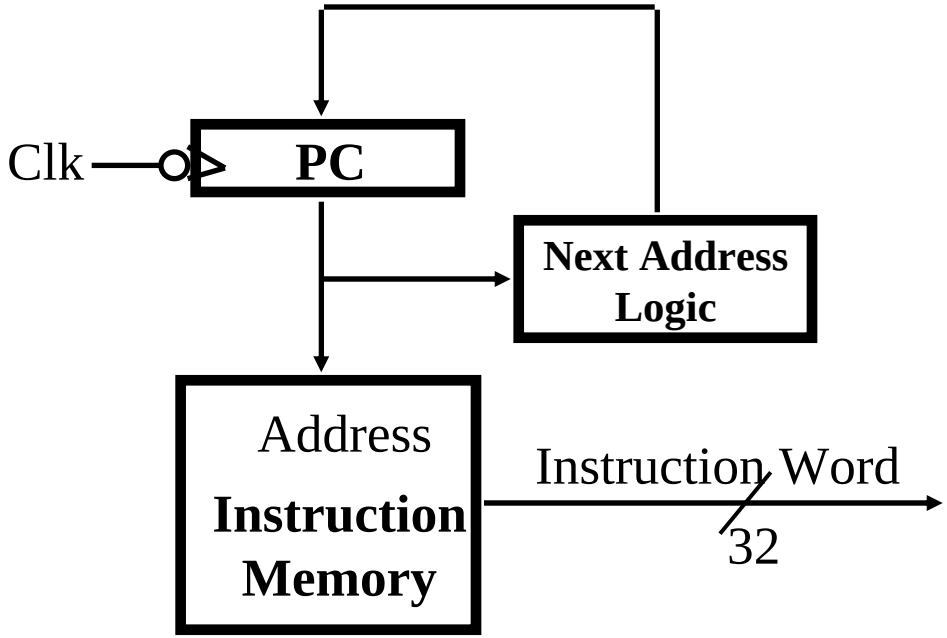
- Dataflow: *(Functional Union of all ISA ops)*
  - Instruction Fetch — IF
  - Read Operands — DE
  - ALU Operation (if necessary) — EX
  - Memory Operation (if necessary) — MEM
  - Write back to Registers (if necessary) — WB

Step 3: Abstract Implementation



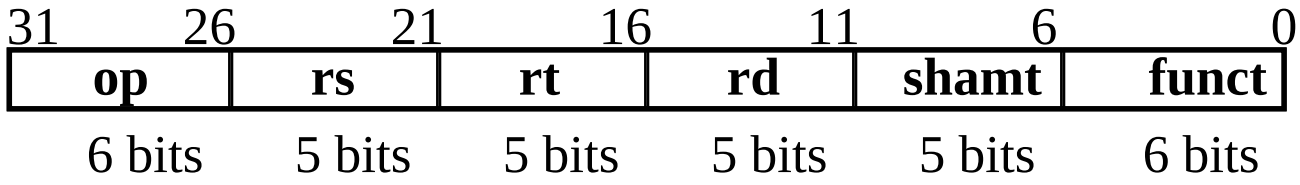
3a: IF: Instruction Fetch

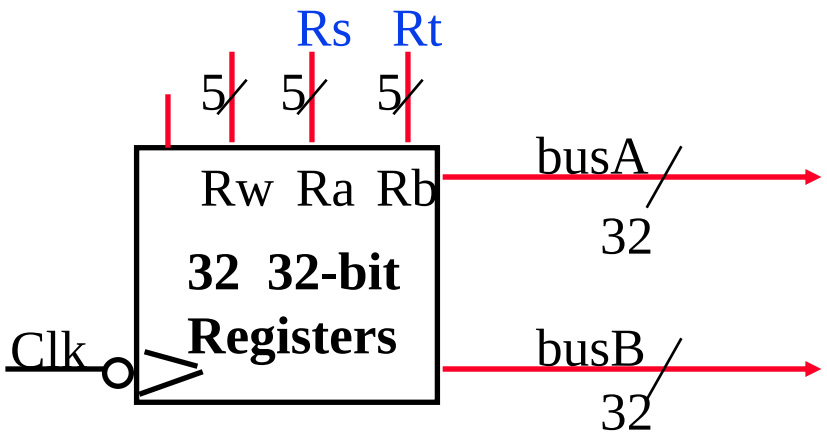
- **The common RTL operations**
  - **Fetch the Instruction: mem[PC]**
  - **Update the program counter:**
    - **Sequential Code: PC = PC + 4**
    - **Branch and Jump: PC = "something else"**



3a: DE: Decode (Read Operands)

- 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



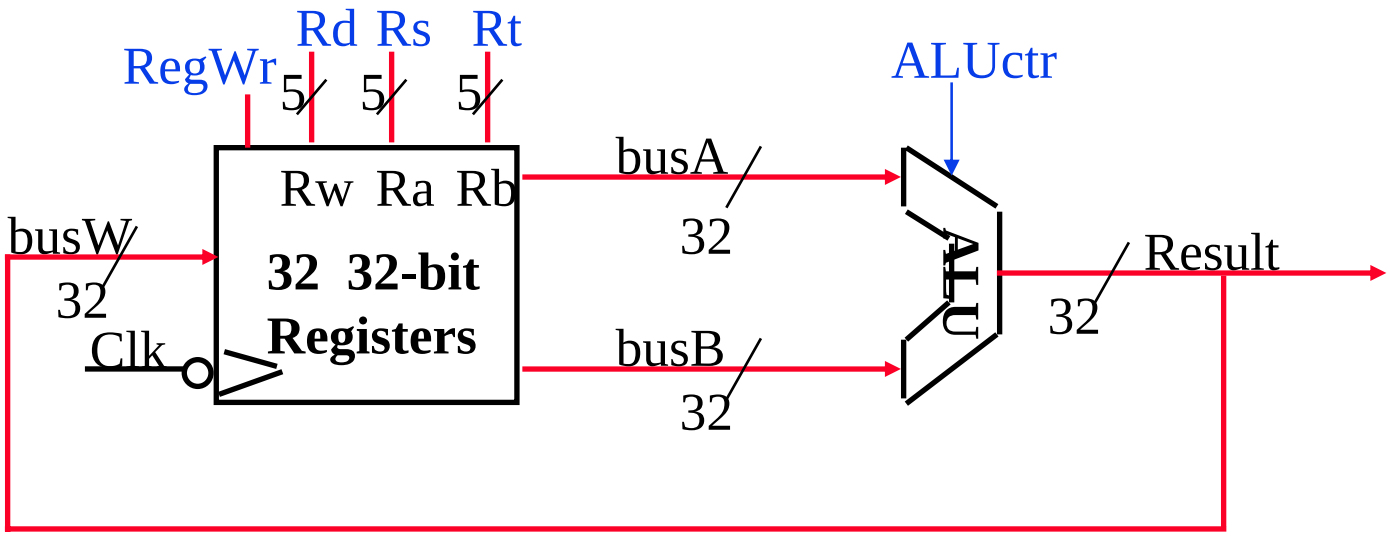


IF and DE are held in common for all ops.

Now, we split up behavior by op type …

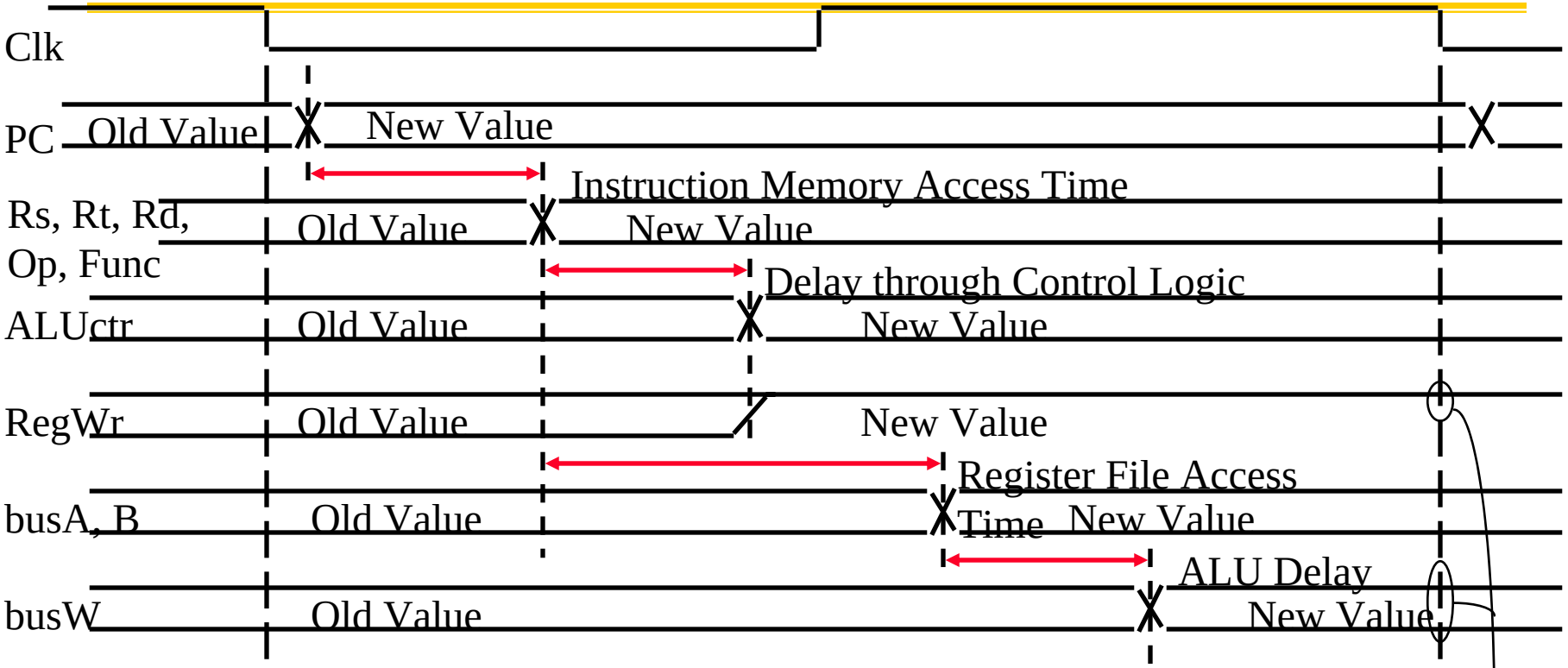
3b: Add & Subtract

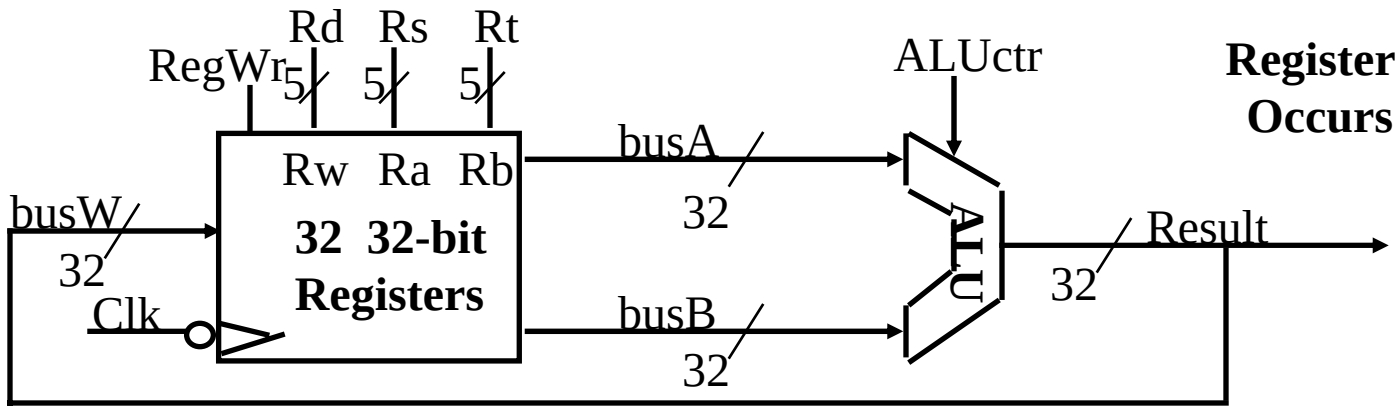
- R[rd] = R[rs] op R[rt] Ex.: addU rd, rs, rt
  - ALUctr and RegWr: control logic after decoding the instruction



- Already defined register file, ALU

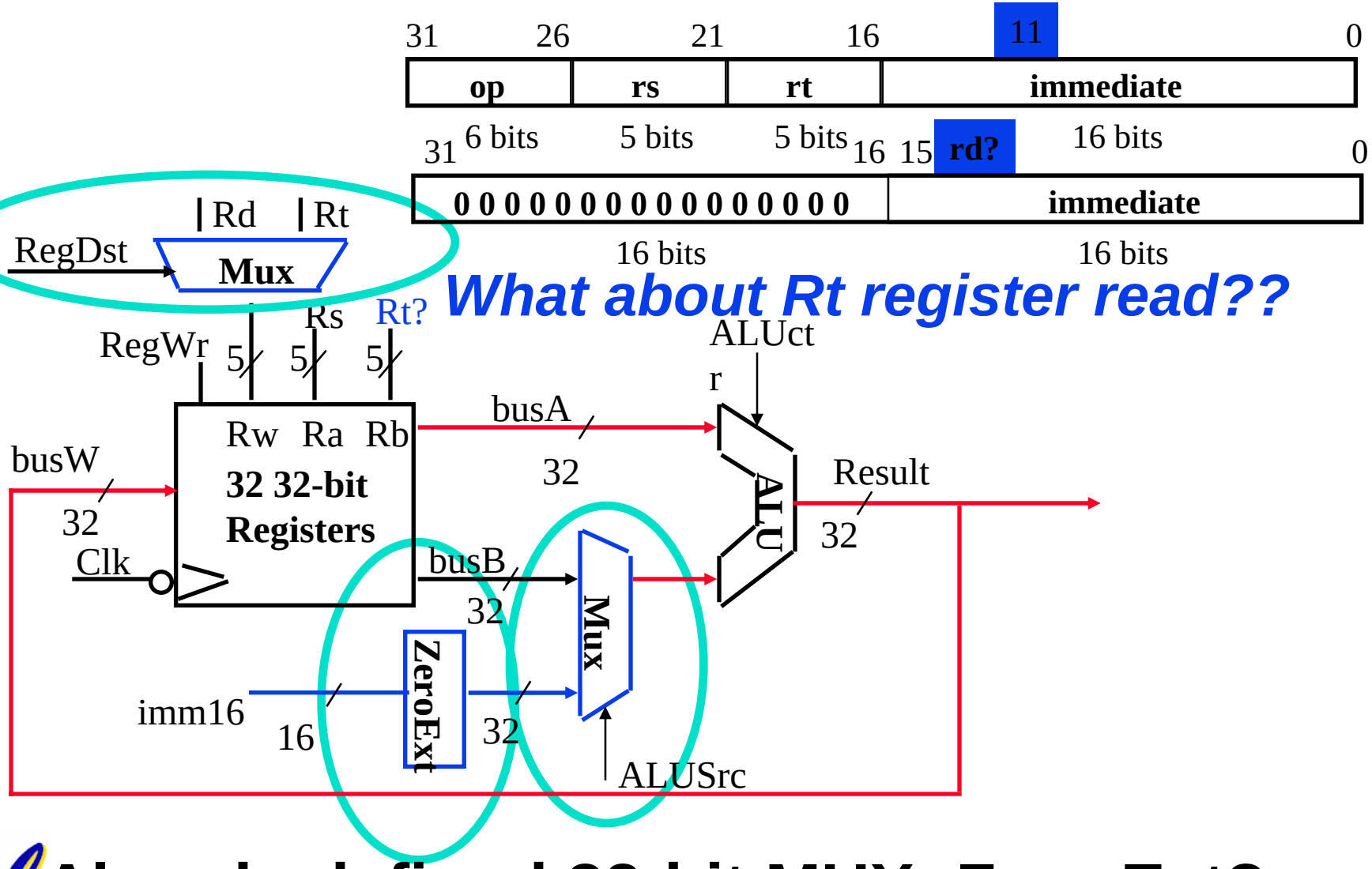
Register-Register Timing: One complete cycle





3c: Logical Operations with Immediate

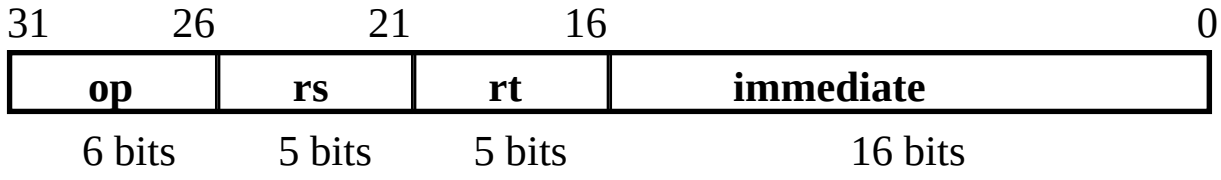
- R[rt] = R[rs] op ZeroExt[imm16]

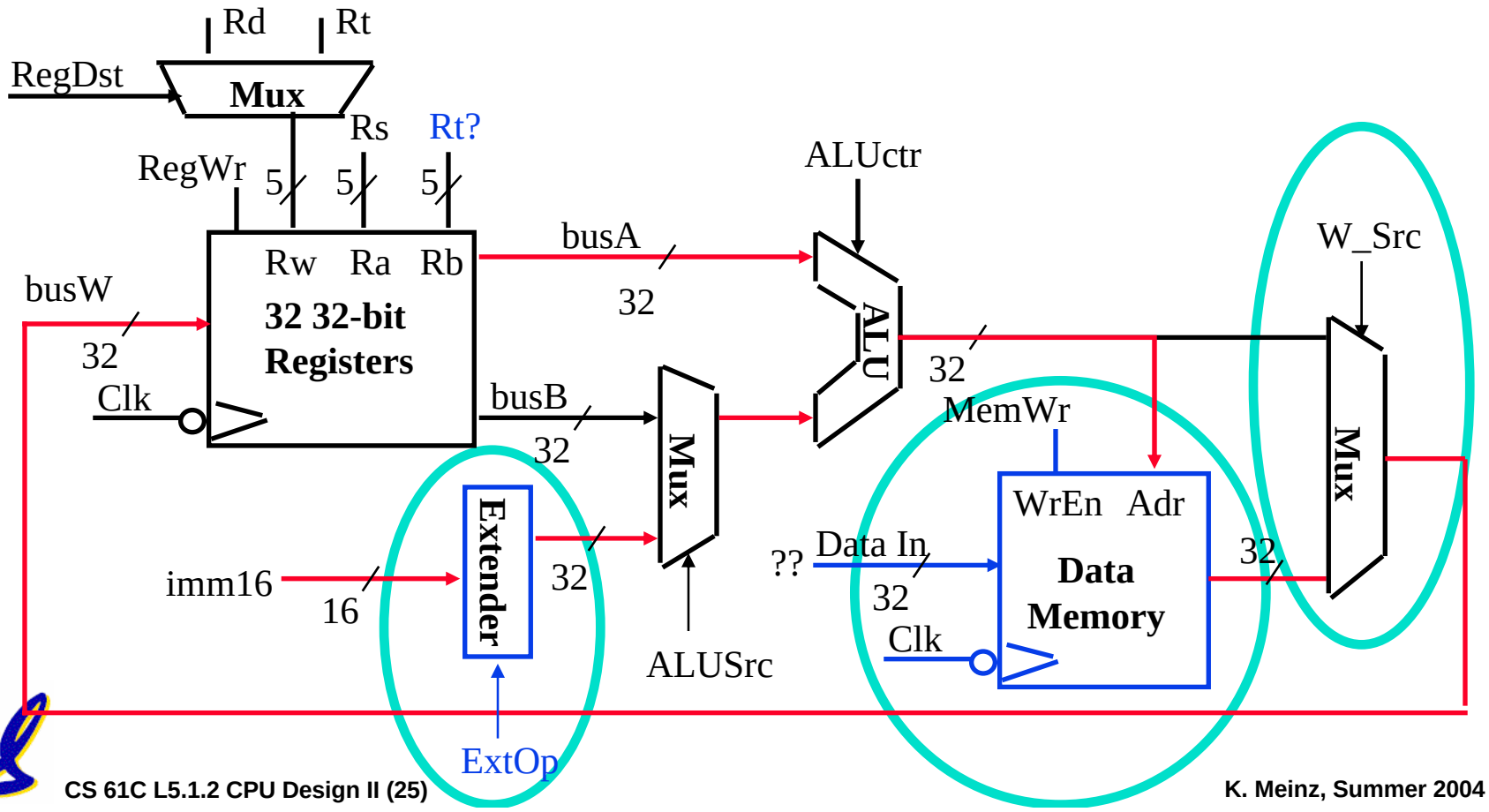


- Already defined 32-bit MUX; Zero Ext?

3d: Load Operations

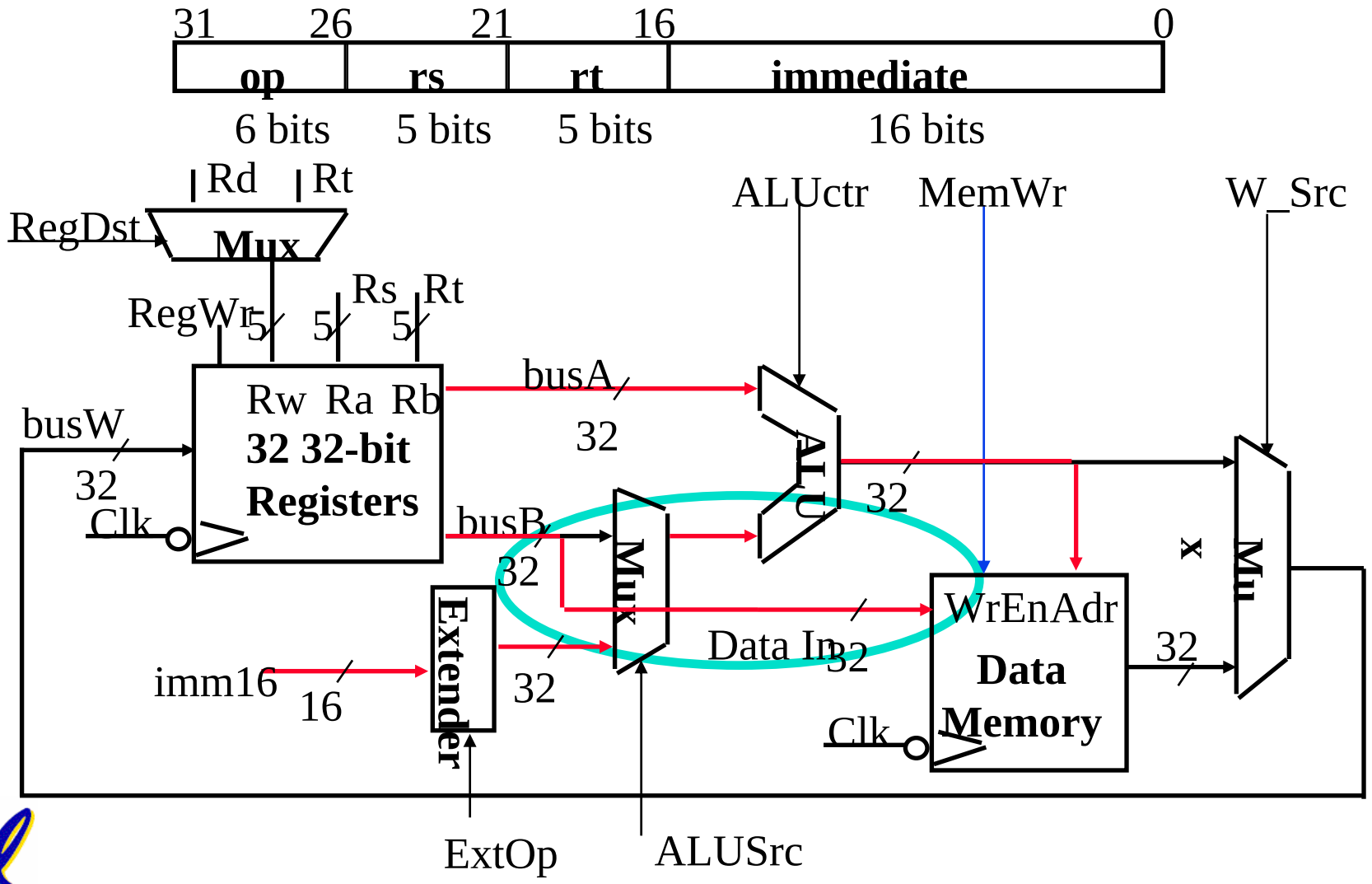
- R[rt] = Mem[R[rs] + SignExt[imm16]]
  Example: lw rt, rs, imm16



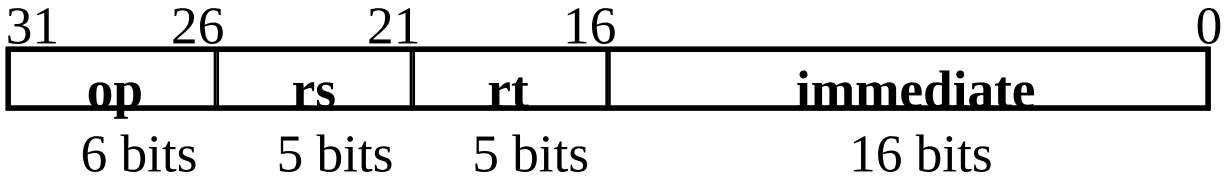


3e: Store Operations

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



3f: The Branch Instruction



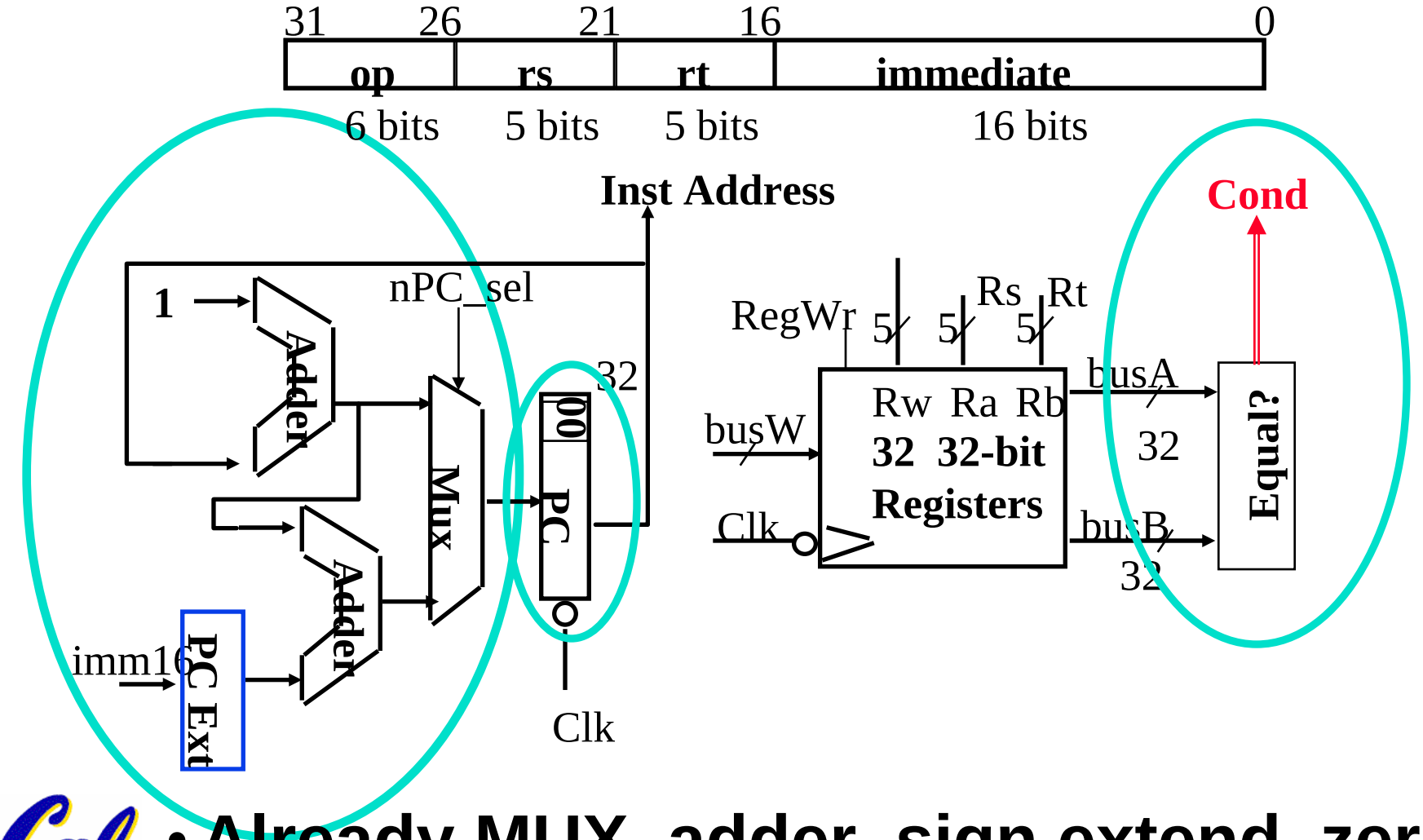
- **beq rs, rt, imm16**
  - **mem[PC]Fetch the instruction from memory**
  - **Equal = R[rs] == R[rt] Calculate the branch condition**
  - **if (Equal) Calculate the next instruction's address**
    - **PC = PC + 4 + (SignExt(imm16) x 4)**

    else
    - **PC = PC + 4**

Datapath for Branch Operations

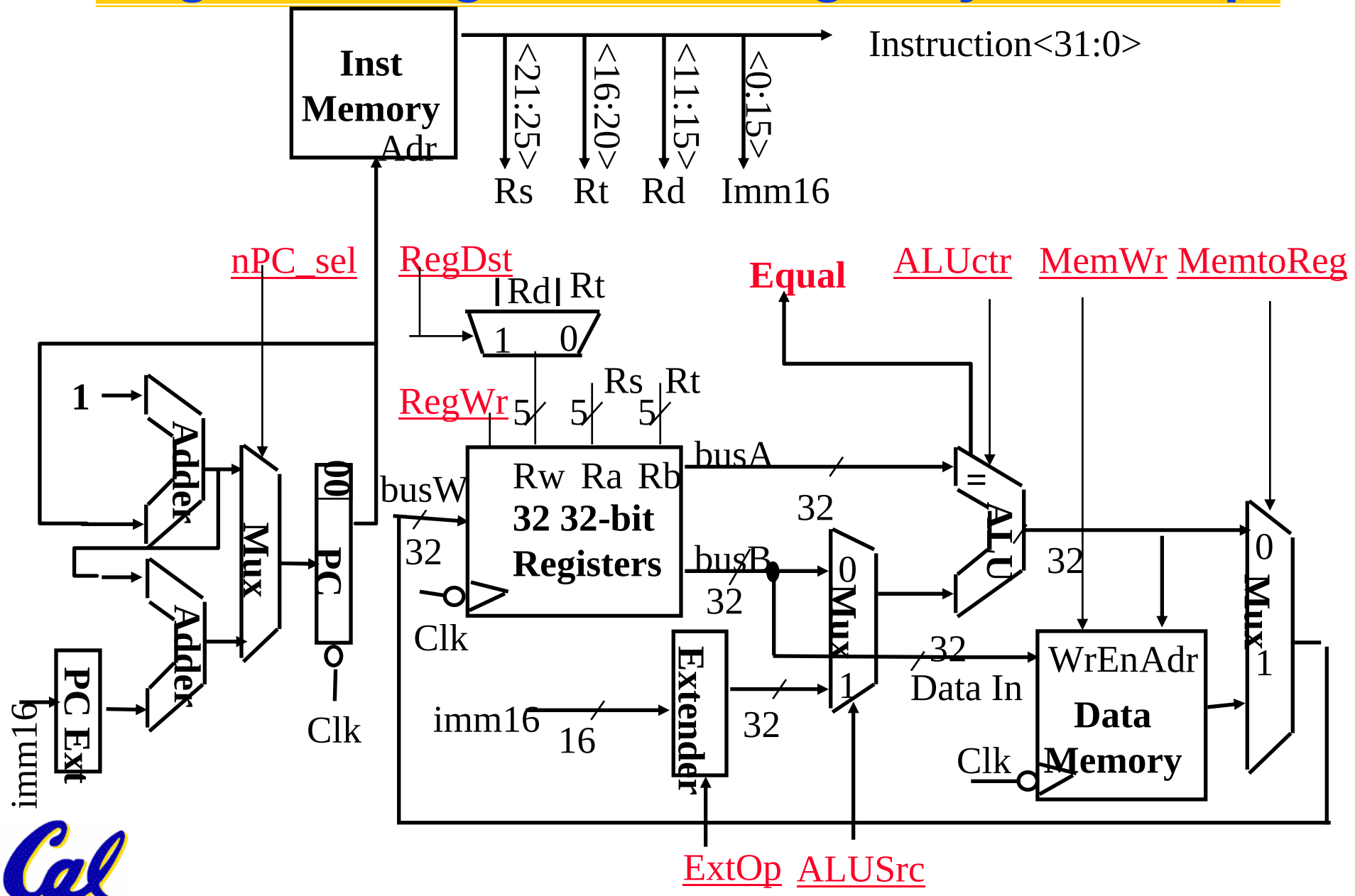
- **beq rs, rt, imm16**

Datapath generates condition (equal)

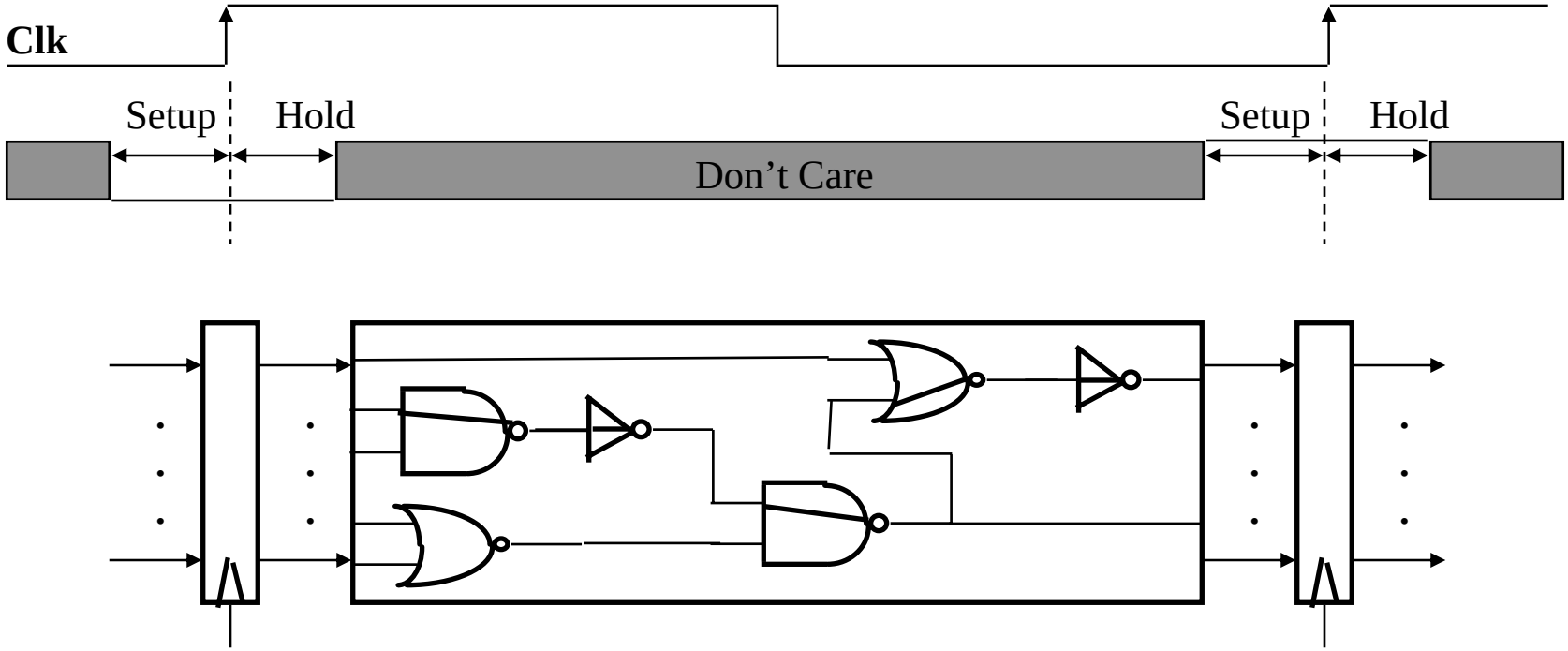


- **Already MUX, adder, sign extend, zero**

Putting it All Together:A Single Cycle Datapath



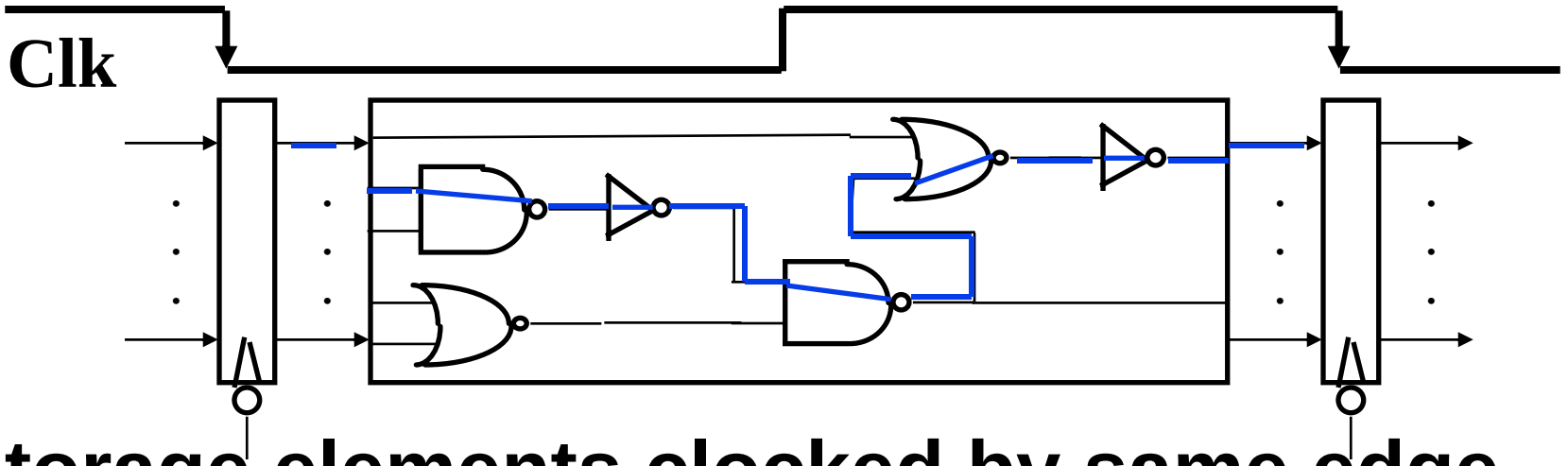
Recall: Clocking Methodology



- All storage elements are clocked by the same clock edge

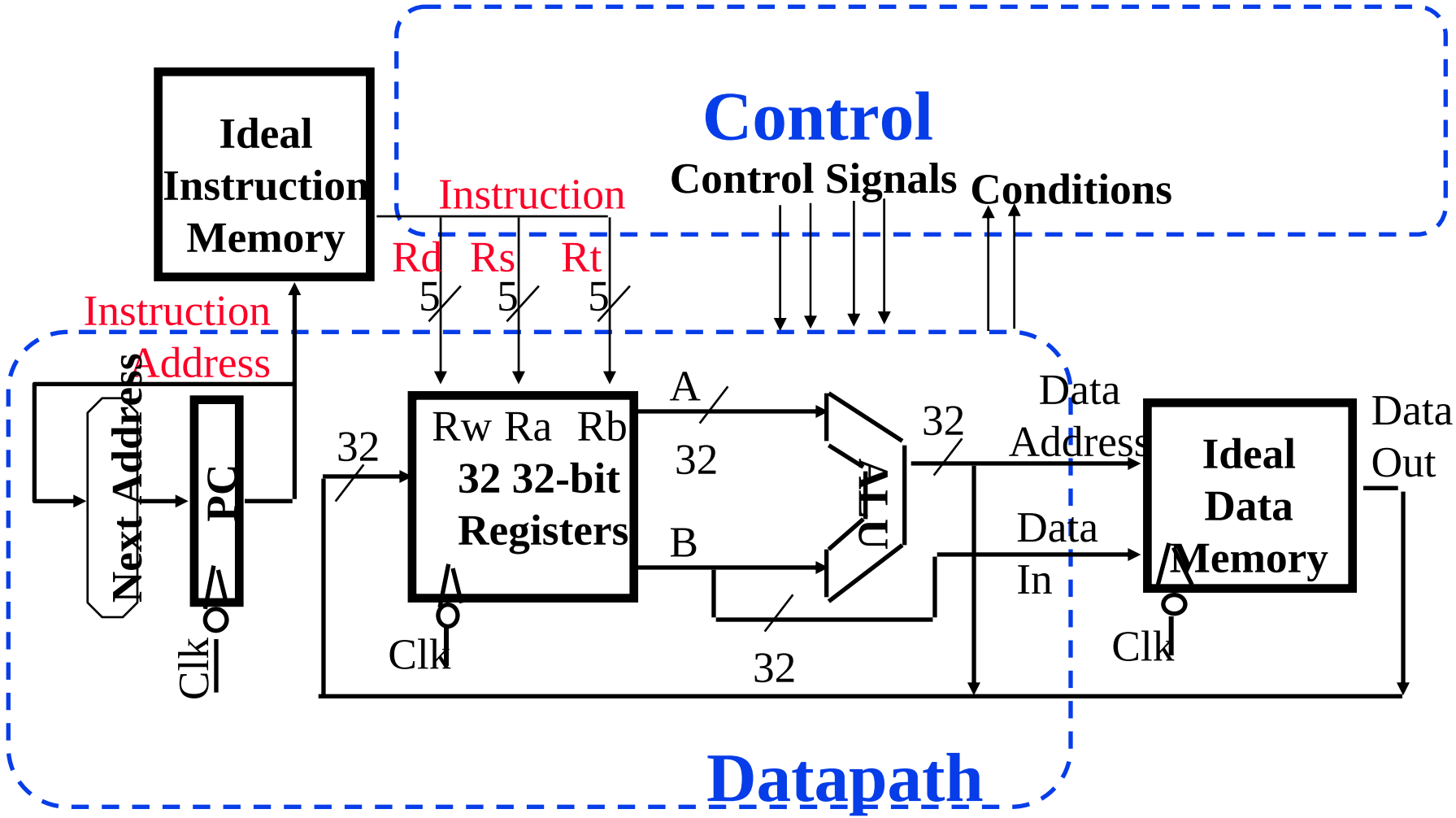
- Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew

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

An Abstract View of the Implementation



An Abstract View of the Critical Path

- This affects how much you can overclock your PC!

Critical Path (Load Operation) =
Delay clock through PC (FFs) +
Instruction Memory's Access Time +
Register File's Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Stable Time for Register File Write