Computer Architecture

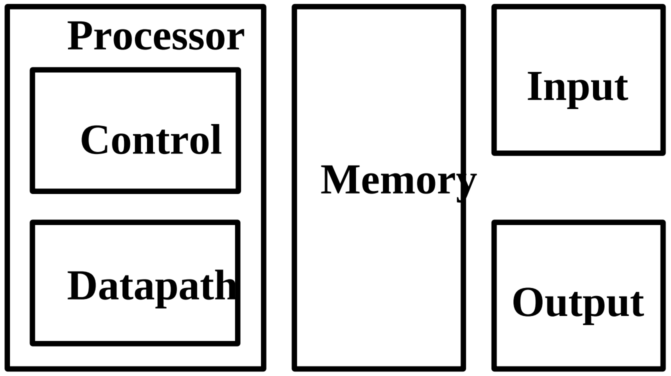
Processor Design - 1

Outline of These Slides

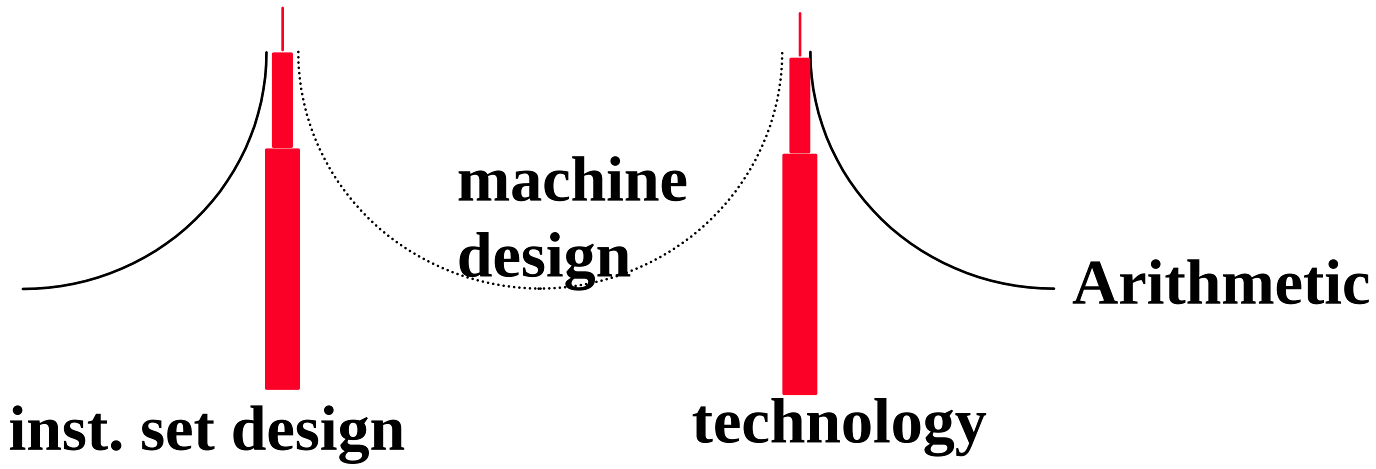
- Overview
- Design a processor: step-by-step
- Requirements of the instruction set
- Components and clocking
- Assembling an adequate Data path
- Controlling the data path

The Big Picture: Where Are We Now?

- **The five classic components of a computer**



- **Today's topic: design a single cycle processor**



The CPU

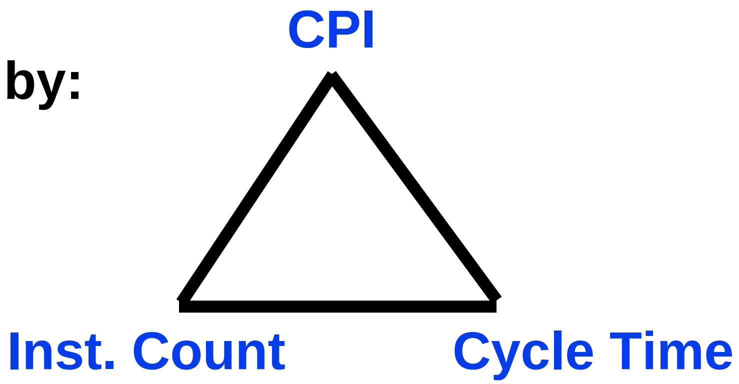
°**Processor (CPU): the active part of the computer, which does all the work (data manipulation and decision-making)**

°**Datapath: portion of the processor which contains hardware necessary to perform operations required by the processor (the brawn)**

°**Control: portion of the processor (also in hardware) which tells the datapath what needs to be done (the brain)**

Big Picture: The Performance Perspective

- **Performance of a machine is determined by:**
  - Instruction count
  - Clock cycle time
  - Clock cycles per instruction



- **Processor design (datapath and control) will determine:**
  - Clock cycle time
  - Clock cycles per instruction

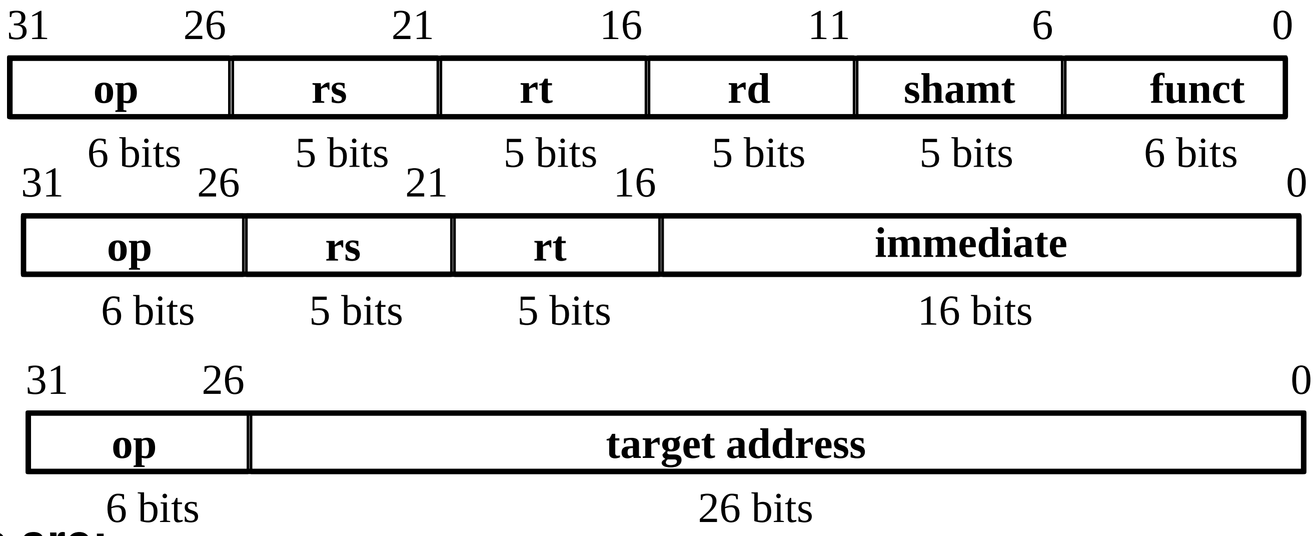
- **What we will do Today:**
  - Single cycle processor:
    - **Advantage: One clock cycle per instruction**
    - **Disadvantage: long cycle time**

How to Design a Processor: Step-by-step

- **1. Analyze instruction set $\rightarrow$ datapath <u>requirements</u>**
  - the meaning of each instruction is given by the *register transfers*
  - datapath must include storage element for ISA registers
    - possibly more
  - datapath must support each register transfer
- **2. Select set of datapath components and establish clocking 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**

The MIPS Instruction Formats

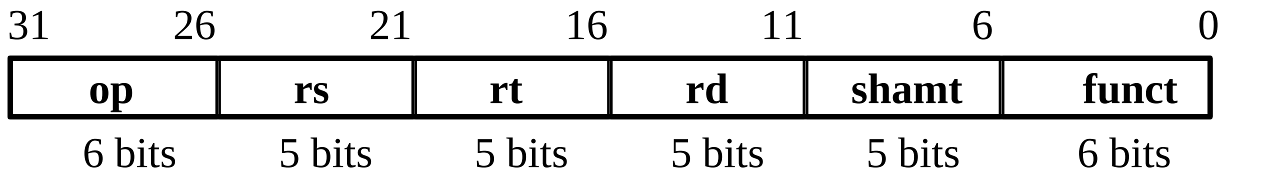
- **All MIPS instructions are 32 bits long. The three instruction formats:**
  - R-type
  - I-type
  - J-type



- **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

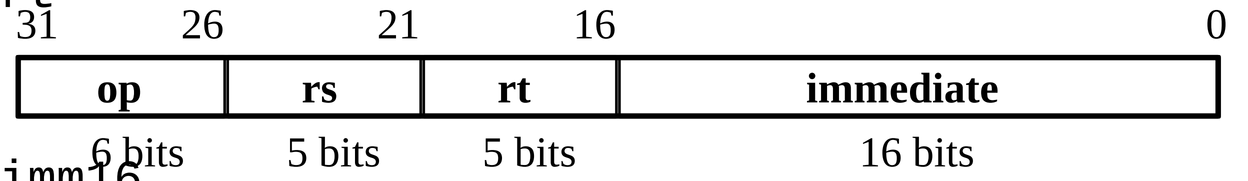
Step 1a: The MIPS-lite Subset for Today

- **ADD and SUB**



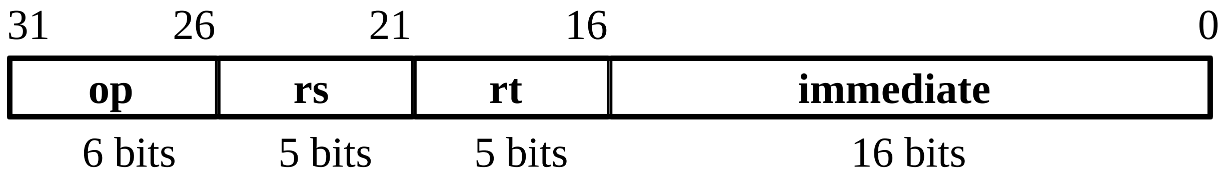
  - `addU rd, rs, rt`
  - `subU rd, rs, rt`

- **OR Immediate:**



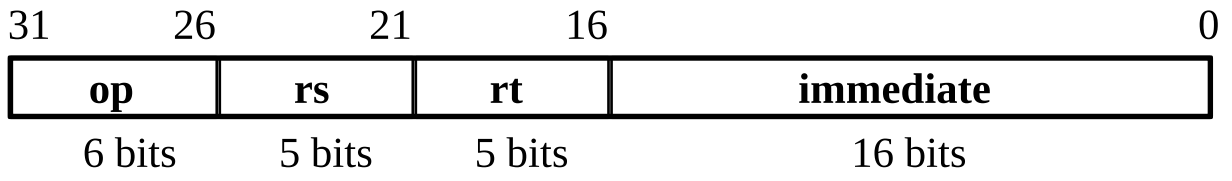
  - `ori  rt, rs, imm16`

- **LOAD / STORE Word**



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

- **BRANCH:**



  - `beq rs, rt, imm16`

Logical Register Transfers

- **Register Transfer Logic gives the meaning of the instructions**
- **All start by fetching the instruction**

op | rs | rt | rd | shamt | funct = MEM[PC]

op | rs | rt | Imm16 = MEM[PC]

inst	**Register Transfers**
ADDU	R[rd] ← R[rs] + R[rt]; PC ← PC + 4
SUBU	R[rd] ← R[rs] – R[rt]; PC ← PC + 4
ORi	R[rt] ← R[rs] | zero_ext(Imm16); PC ← PC + 4
LOAD	R[rt] ← MEM[R[rs] + sign_ext(Imm16)]; PC ← PC + 4
STORE	MEM[R[rs] + sign_ext(Imm16)] ← R[rt]; PC ← PC + 4
BEQ	if (R[rs] == R[rt]) then PC ← PC + 4 + sign_ext(Imm16)] || 00
	else PC ← PC + 4

Step 1: Requirements of the Instruction Set

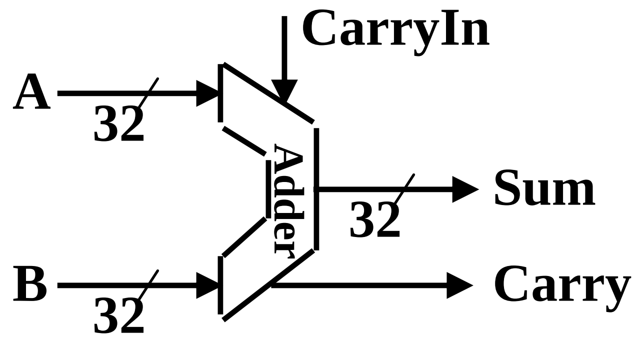
- **Memory**
  - instruction & data
- **Registers (32 x 32)**
  - read RS
  - read RT
  - Write RT or RD
- **PC**
- **Extender**
- **Add and Sub register or extended immediate**
- **Add 4 or extended immediate to PC**

Step 2: Components of the Datapath

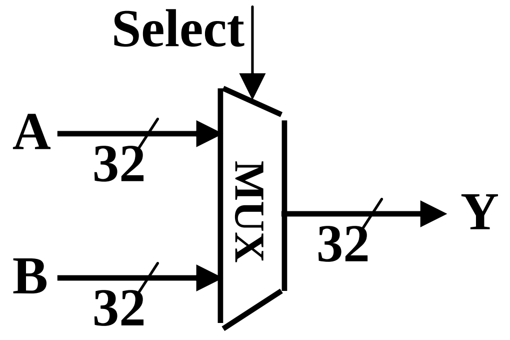
- **Combinational Elements**
- **Storage Elements**
  - Clocking methodology

Combinational Logic Elements (Basic Building Blocks)

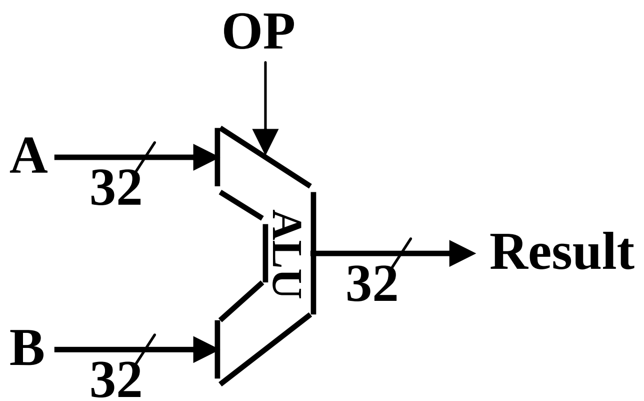
- **Adder**



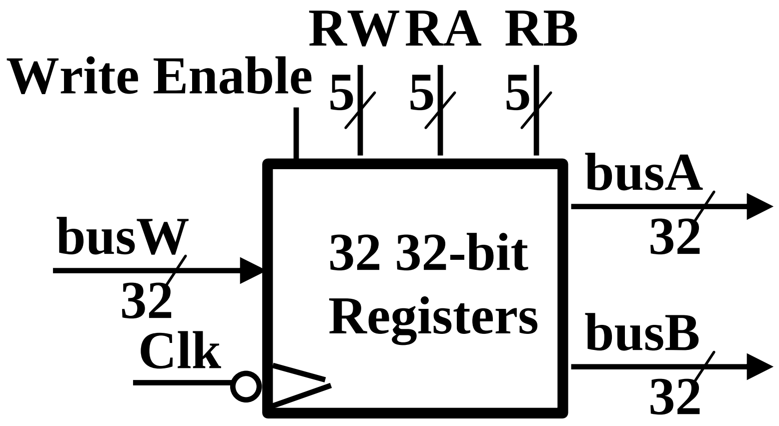
- **MUX**



- **ALU**



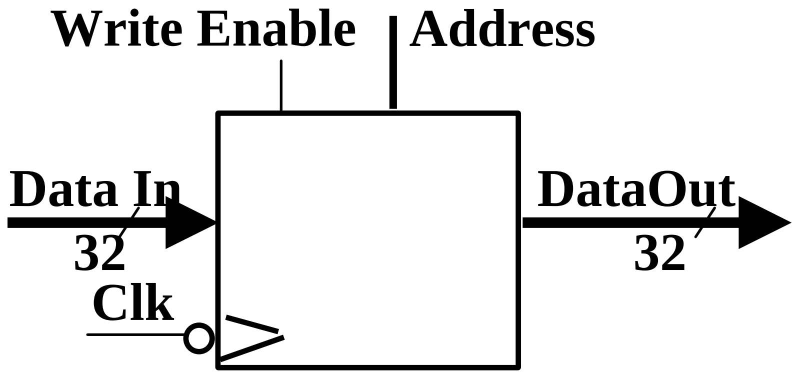
Storage Element: Register File



- **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.”

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 $\rightarrow$ Data Out valid after "access time."

Memory Hierarchy (Ch. 7)

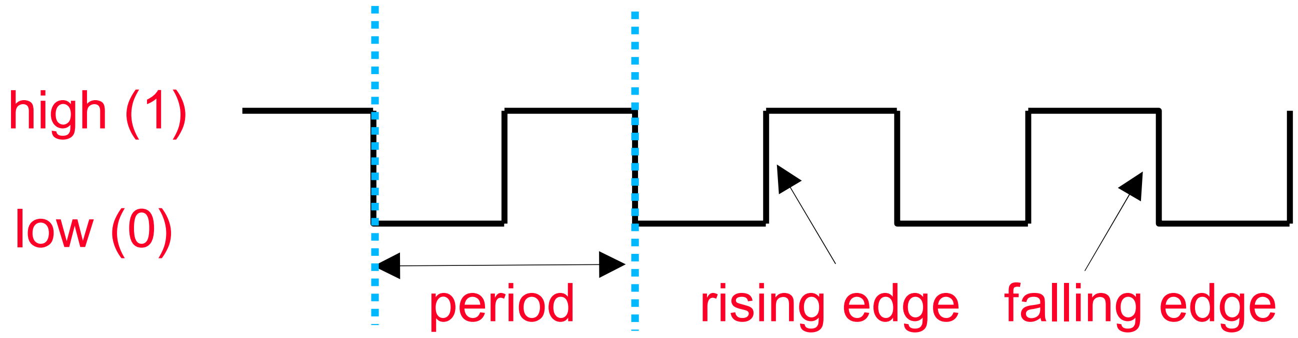
- **Want a single main memory, both large and fast**
- **Problem 1: large memories are slow while fast memories are small**
  - Example: MIPS registers (fast, but few)
- ***Solution*: mix of memories provides illusion of single large, fast memory**
  - Cache: a small, fast memory; Holds a copy of part of a larger, slower memory
  - Imem, Dmem are really separate caches memories

Digression: Sequential Logic, Clocking

- **Combinational circuits: no memory**
  - Output depends only on the inputs
- **Sequential circuits: have memory**
  - How to ensure memory element is updated neither too soon, nor too late?
  - Recall hardware multiplier
    - Product/multiplier register is the writable memory element
    - Gate propagation delay means ALU result takes time to stabilize; Delay varies with inputs
    - Must wait until result stable before write to product/multiplier register else get garbage
    - How to be certain ALU output is stable?

Adding a Clock to a Circuit

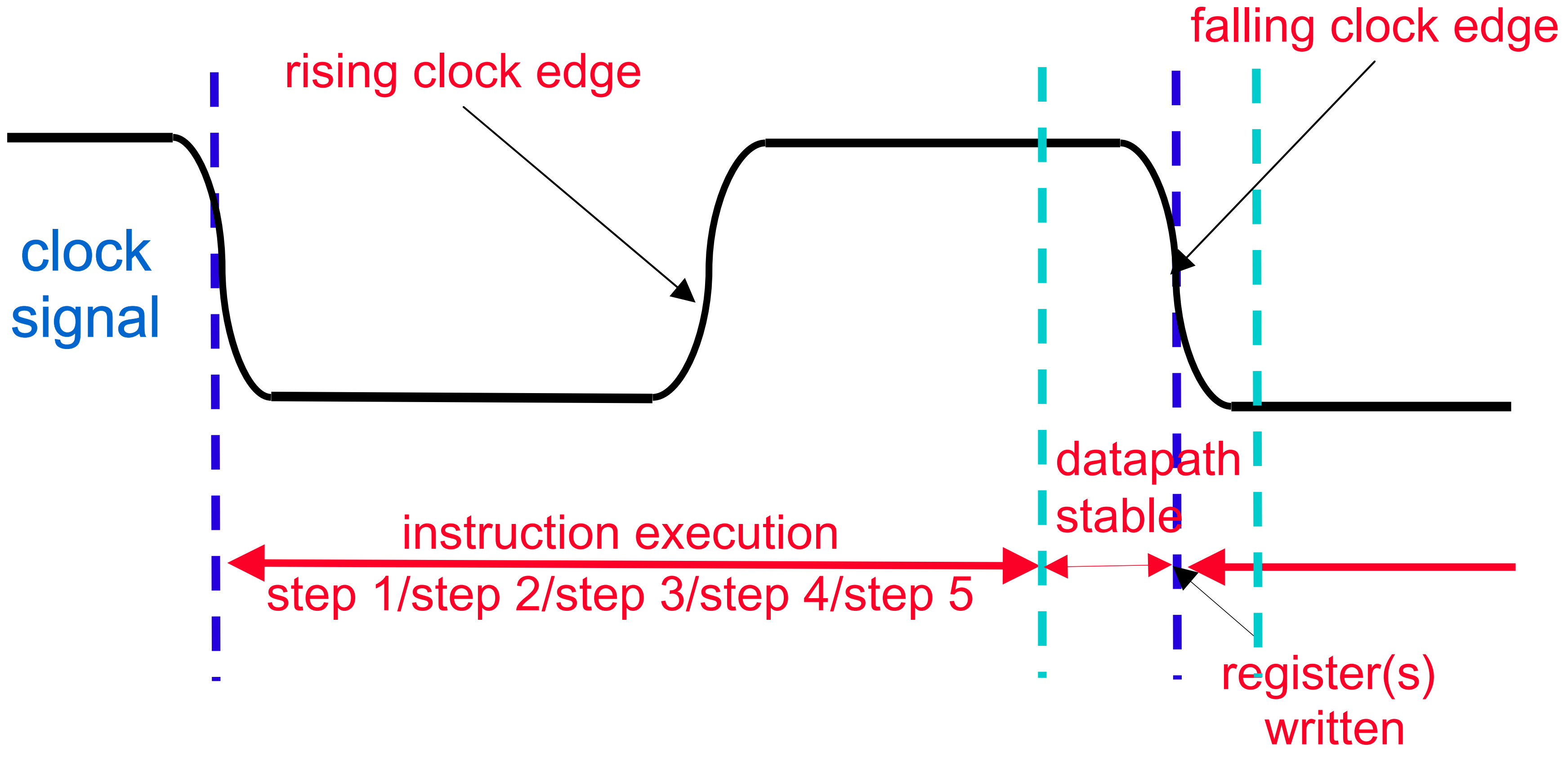
- **Clock: free running signal with fixed *cycle time* (*clock period*)**



  - **Clock determines *when* to write memory element**
    - **level-triggered - store clock high (low)**
    - **edge-triggered - store only on clock edge**
  - **We will use negative (falling) <u>edge-triggered methodology</u>**

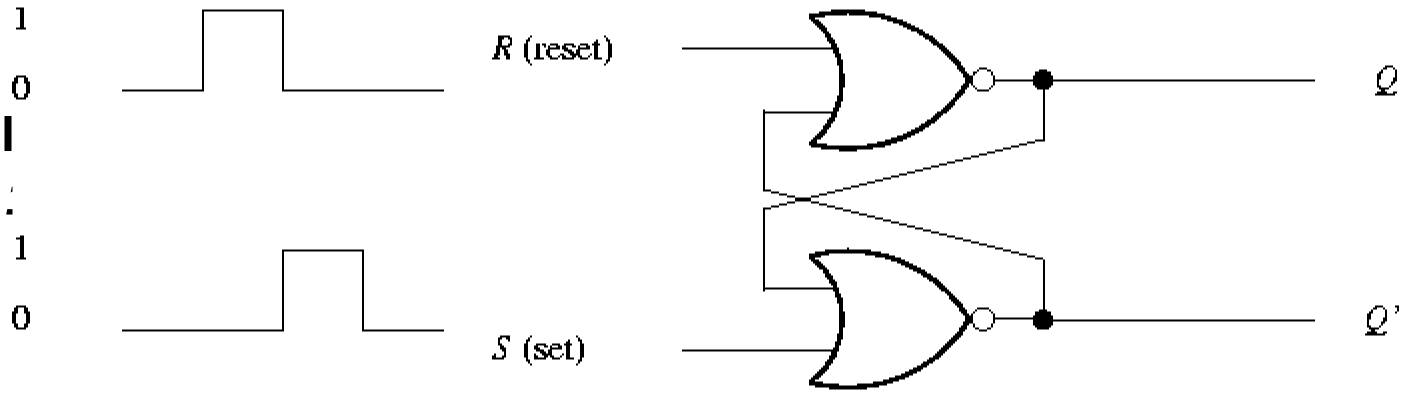
Role of Clock in MIPS Processors

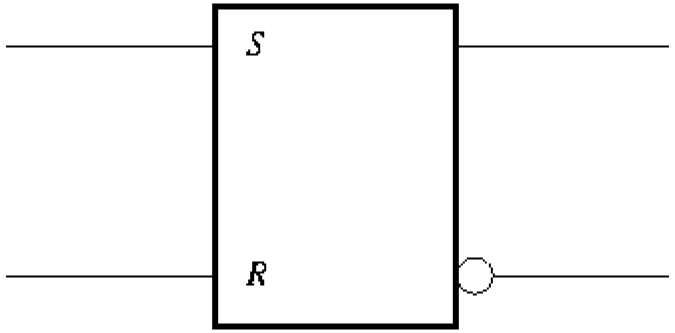
- **single-cycle machine: does everything in one clock cycle**
  - instruction execution = up to 5 steps
  - must complete 5th step *before* cycle ends



SR-Latches

- **SR-l**
- ***S*** **= :**

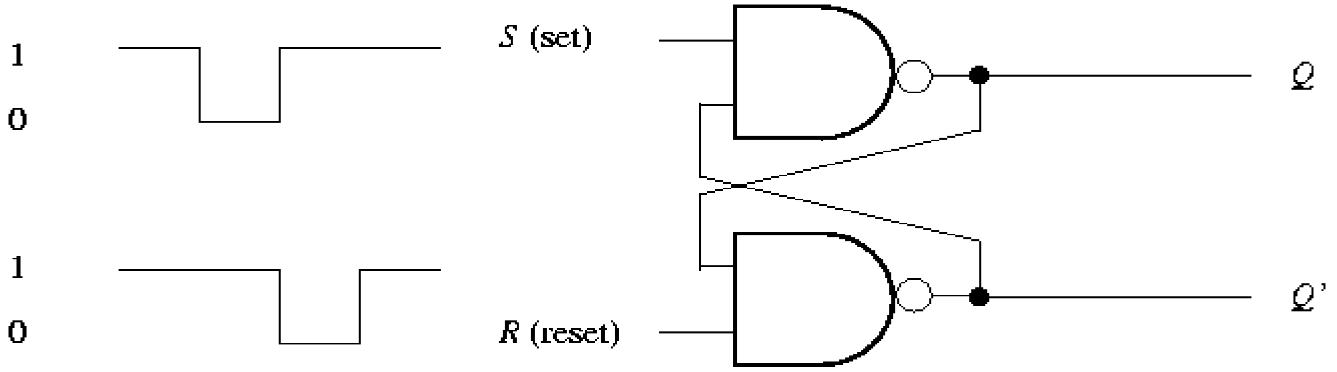


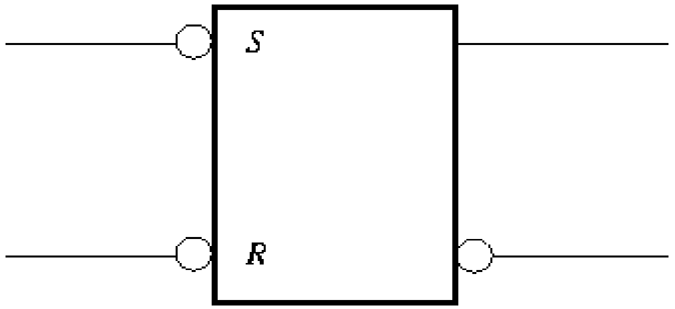


- **Symbol for SR-Latch with NOR gates**

SR-Latches

- **SR-latch with NAND Gates, also known as S´R´ -latch**
- ***S = 0 and R = 0 not allowed***

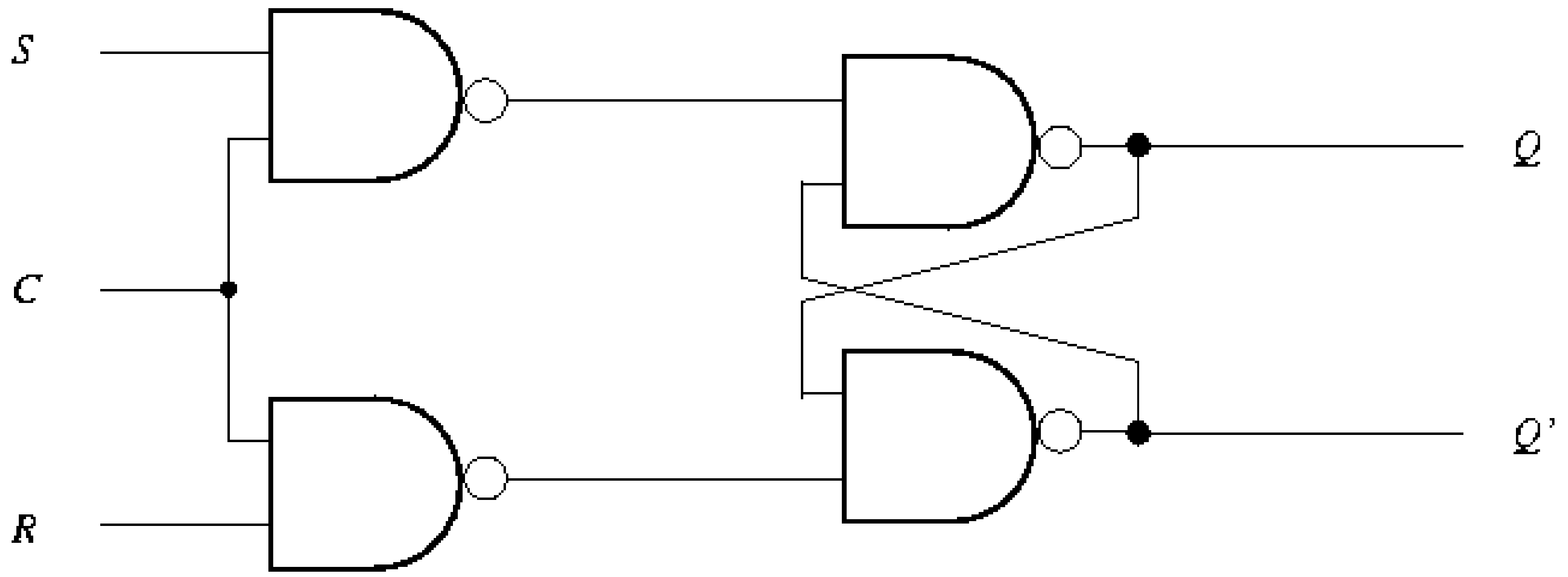




- **Symbol for SR-Latch with NAND gates**

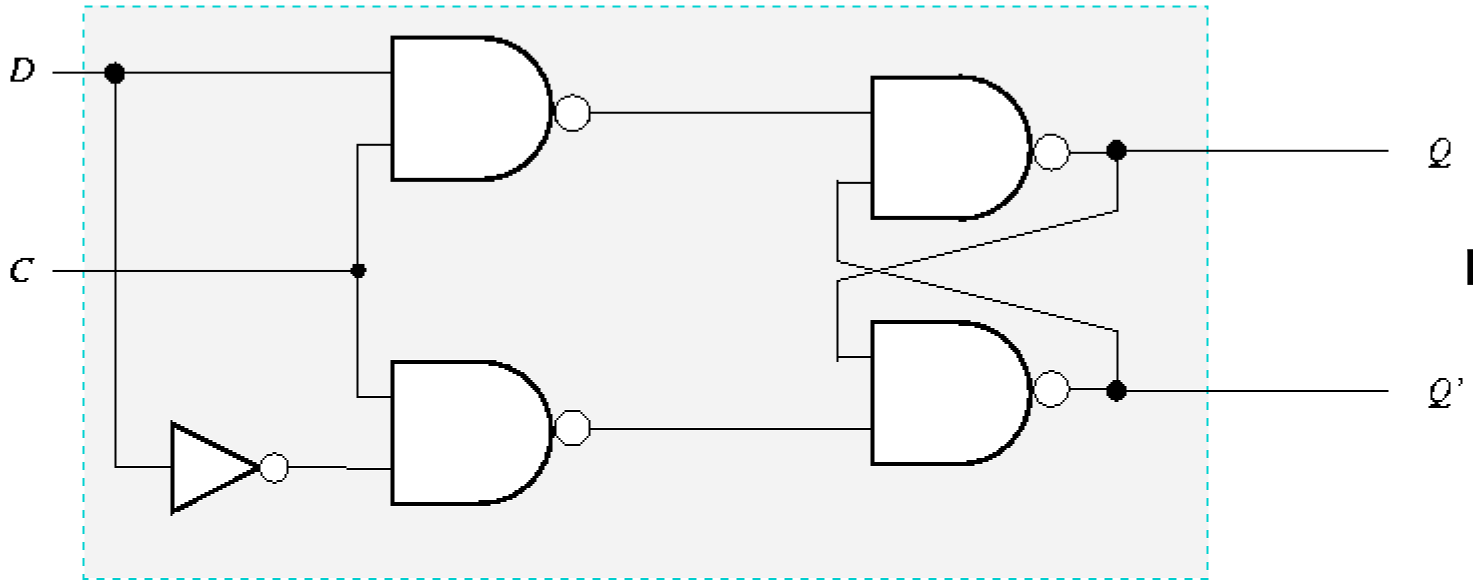
SR-Latches with Control Input

•

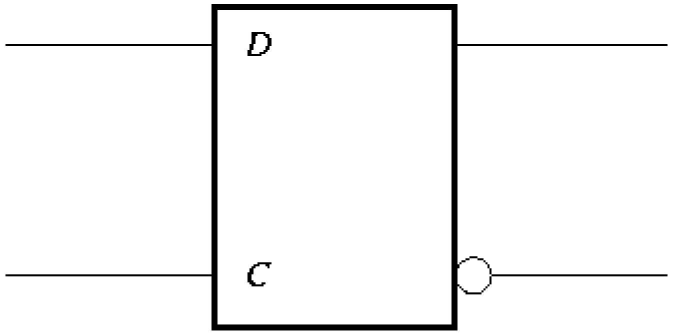


- *C* = 0, no change of state;
- *C* = 1, change is allowed;
  - If *S* = 1 and *R* = 1, *Q* and *Q* ´are Indeterminated

D-Latches



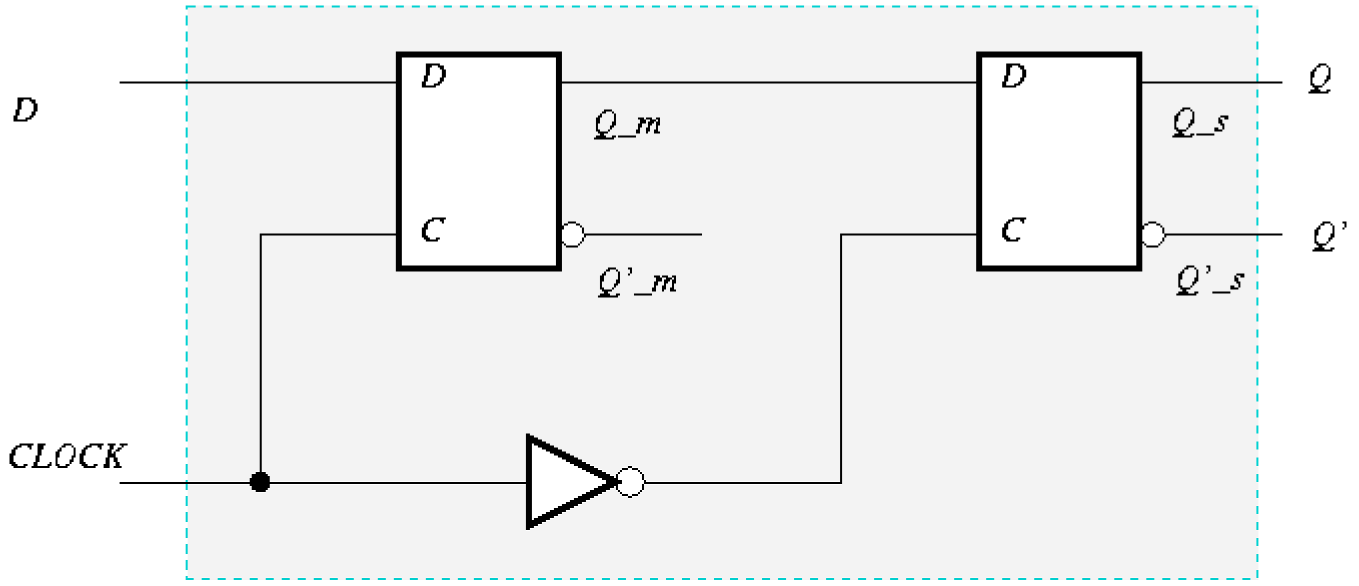
put *C*

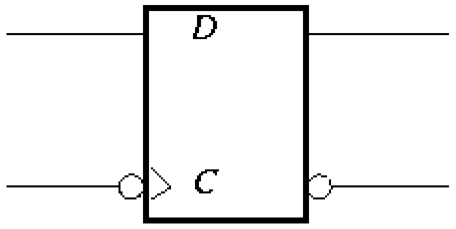


- *C* = 0, no change of state;
  - $Q\ (t + \delta t\ ) = Q\ (t\ )$
- *C* = 1, change is allowed;
  - $Q\ (t + \delta t\ ) = D\ (t\ )$
  - **No Indetermined Output**

Master-Slave Flip-Flop

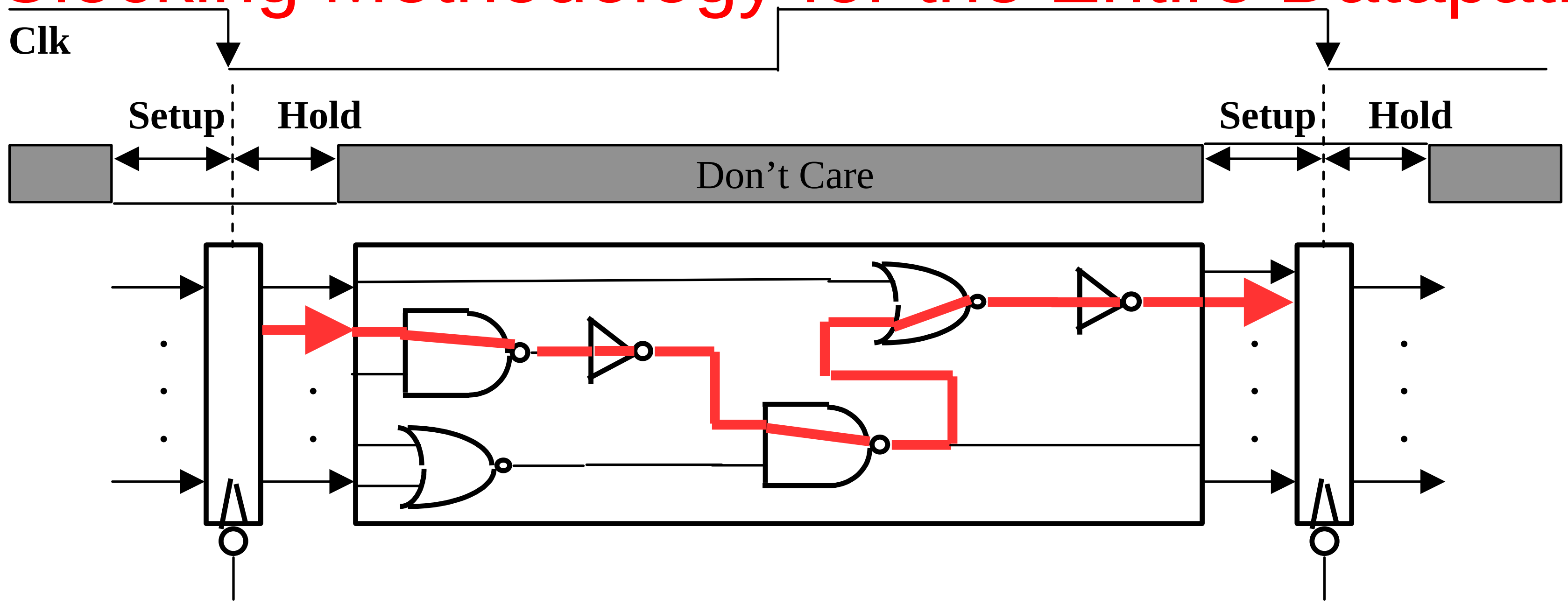
- **Negative-edge triggered D-Flip Flop**





- **Symbol for D-Flip Flop.**
- **Arrowhead (>) indicates an edge-triggered sequential circuit.**
- **Bubble means that triggering is effective during the High→Low *C* transition**

Clocking Methodology for the Entire Datapath



- **Design/synthesis based on *pulsed-sequential circuits***
  - *All combinational inputs remain at constant levels and only clock signal appears as a pulse with a fixed period T_{cc}*
- **All storage elements are clocked by the same clock edge**
- **Cycle time T_{cc} = CLK-to-q + longest delay path + Setup time + clock skew**
- **(CLK-to-q + shortest delay path - clock skew) > hold time**

Step 3: Assemble Data Path Meeting Requirements

- Register Transfer <u>Requirements</u>
  $\Rightarrow$ Datapath "<u>Assembly</u>"
- Instruction Fetch
- Read Operands and Execute Operation

Stages of the Datapath (1/6)

Problem: a single, atomic block which "executes an instruction" (performs all necessary operations beginning with fetching the instruction) would be too bulky and inefficient

Solution: break up the process of "executing an instruction" into stages, and then connect the stages to create the whole datapath

- **Smaller stages are easier to design**
- **Easy to optimize (change) one stage without touching the others**

Stages of the Datapath (2/6)

There is a *wide* variety of MIPS instructions: so what general steps do they have in common?

Stage 1: instruction fetch

- No matter what the instruction, the 32-bit instruction word must first be fetched from memory (the cache-memory hierarchy)
- Also, this is where we increment PC
(that is, PC = PC + 4, to point to the next instruction: byte addressing so + 4)

Stages of the Datapath (3/6)

Stage 2: **Instruction Decode**

- upon fetching the instruction, we next gather data from the fields (*decode* all necessary instruction data)
- first, read the Opcode to determine instruction type and field lengths
- second, read in data from all necessary registers

  -for add, read two registers

  -for addi, read one register

  -for jal, no reads necessary

Stages of the Datapath (4/6)

° *Stage 3: ALU (Arithmetic-Logic Unit)*

- ***the real work of most instructions is done here: arithmetic (+, -, *, /), shifting, logic (&, |), comparisons (slt)***
- ***what about loads and stores?***
  - ***lw $t0, 40($t1)***
  - ***the address we are accessing in memory = the value in $t1 + the value 40***
  - ***so we do this addition in this stage***

Stages of the Datapath (5/6)

° **<u>Stage 4</u>: Memory Access**

- actually only the load and store instructions do anything during this stage; the others remain idle
- since these instructions have a unique step, we need this extra stage to account for them
- as a result of the cache system, this stage is expected to be just as fast (on average) as the others

Stages of the Datapath (6/6)

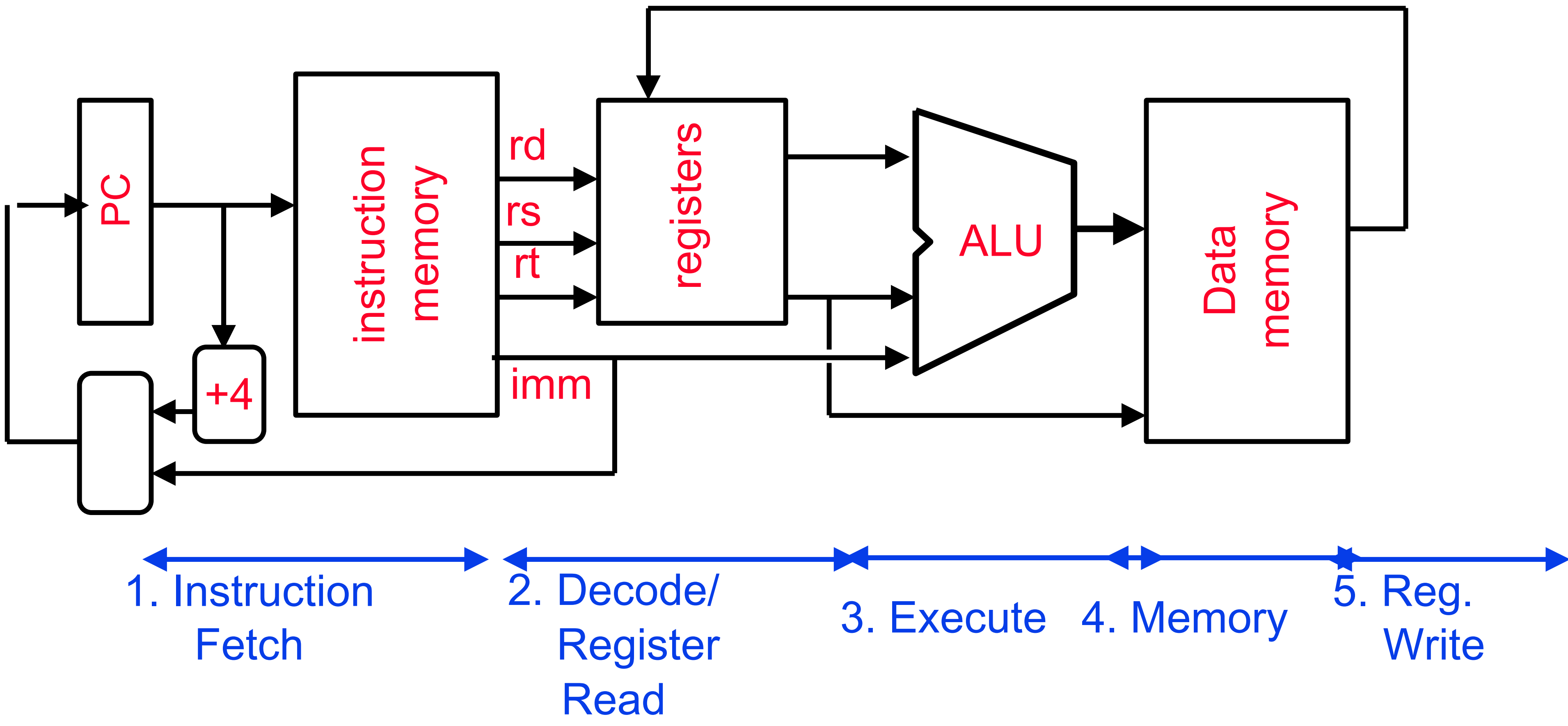
°**Stage 5: Register Write**

- most instructions write the result of some computation into a register
- examples: arithmetic, logical, shifts, loads, slt
- what about stores, branches, jumps?

  -don’t write anything into a register at the end

  -these remain idle during this fifth stage

Generic Steps: Datapath

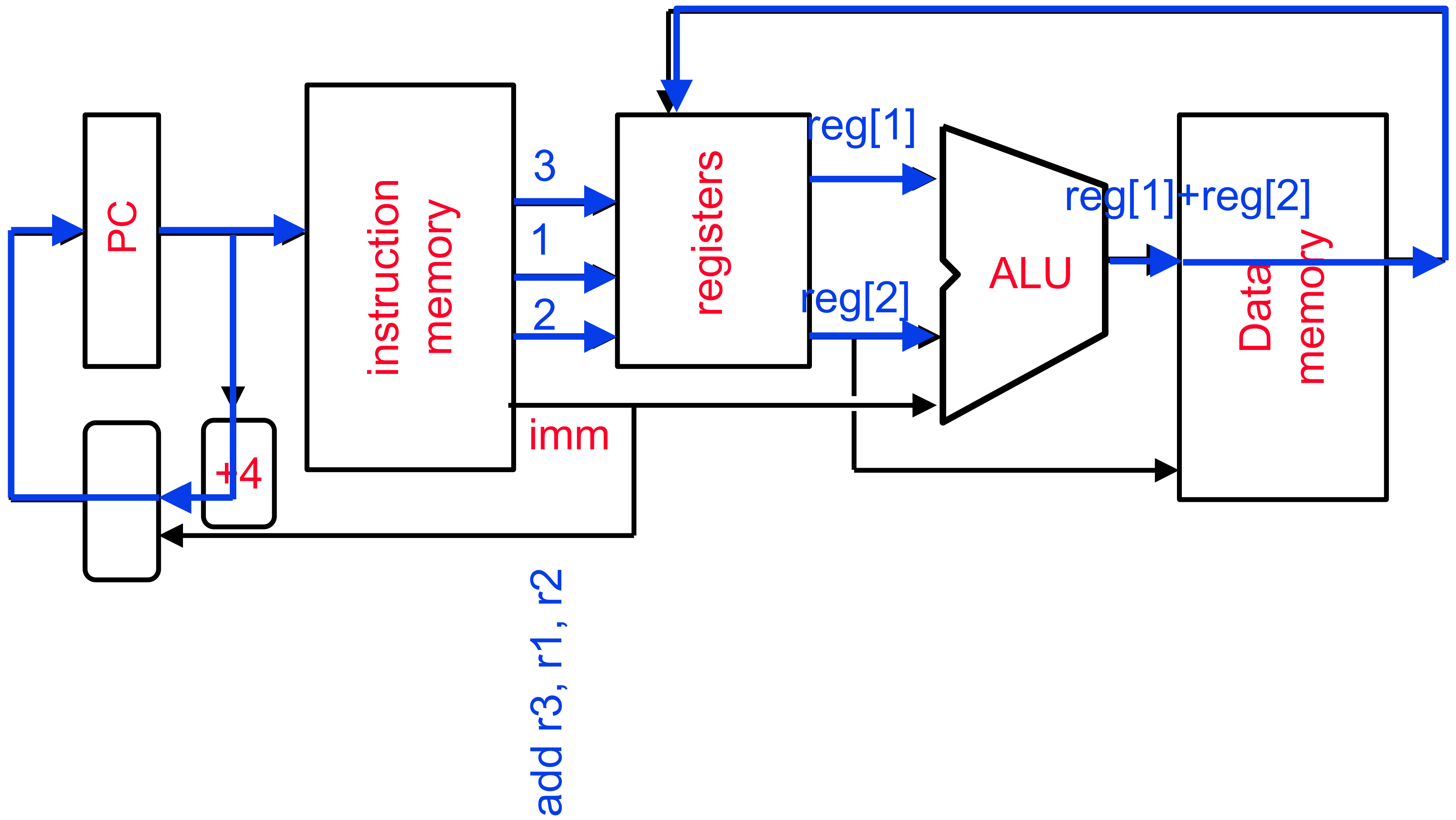


Datapath Walkthroughs (1/3)

```
add   $r3, $r1, $r2 # r3 = r1+r2
```

- Stage 1: fetch this instruction, incr. PC ;
- Stage 2: decode to find it's an `add`, then read registers `$r1` and `$r2` ;
- Stage 3: add the two values retrieved in Stage 2 ;
- Stage 4: idle (nothing to write to memory) ;
- Stage 5: write result of Stage 3 into register `$r3` ;

Example: add Instruction

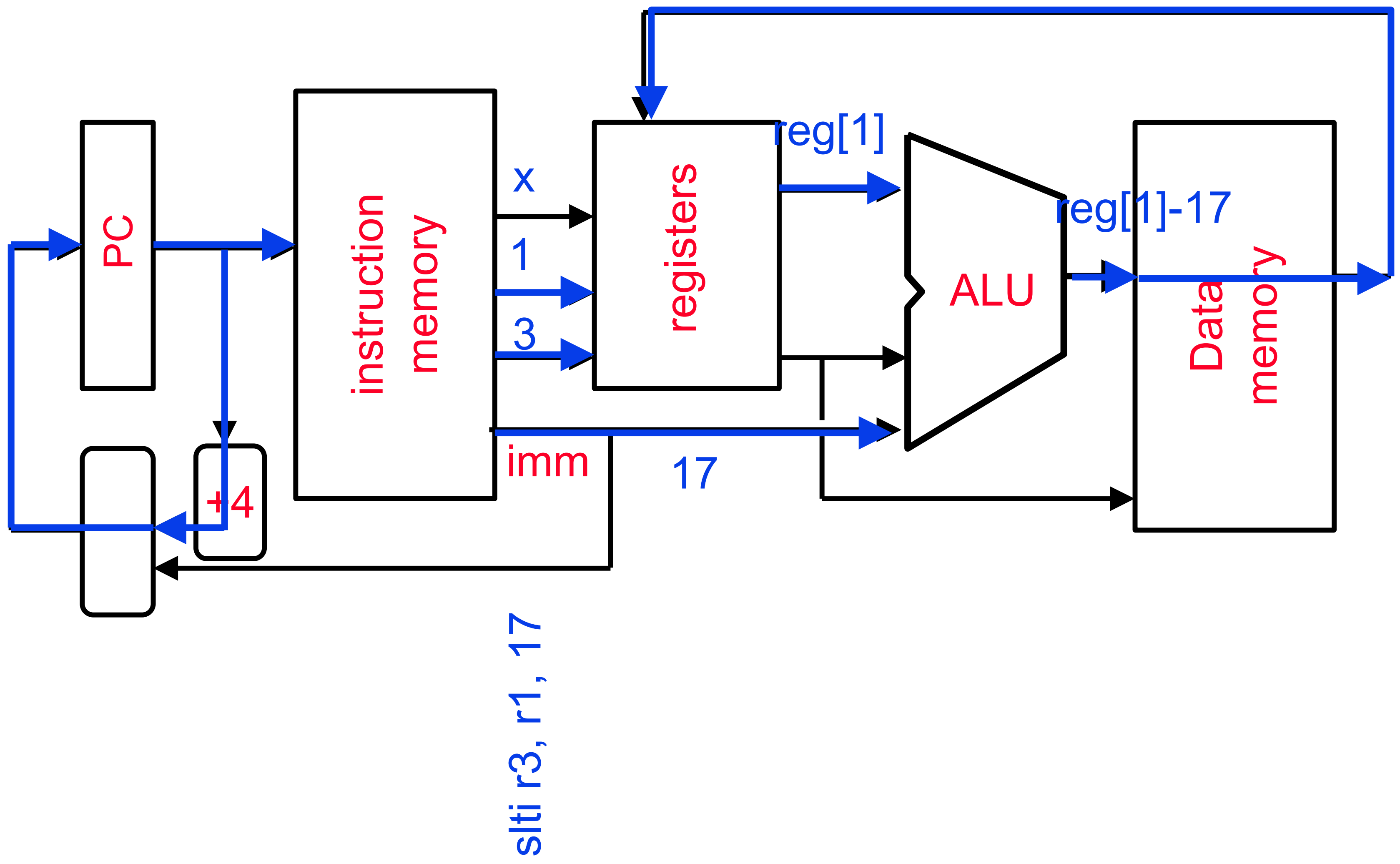


Datapath Walkthroughs (2/3)

`slti   $r3, $r1, 17`

- Stage 1: fetch this instruction, inc. PC
- Stage 2: decode to find it's an `slti`, then read register `$r1`
- Stage 3: compare value retrieved in Stage 2 with the integer 17
- Stage 4: go idle
- Stage 5: write the result of Stage 3 in register `$r3`

Example: slti Instruction

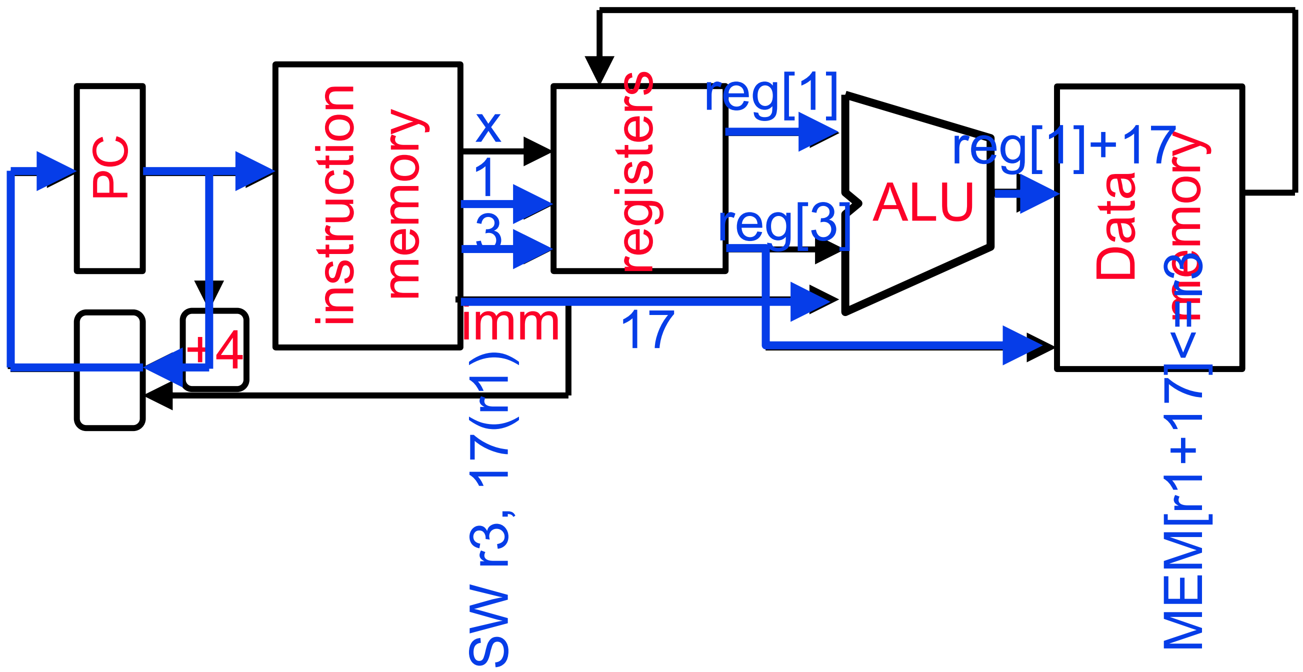


Datapath Walkthroughs (3/3)

```
sw    $r3, 17($r1)
```

- Stage 1: fetch this instruction, inc. PC
- Stage 2: decode to find it's a sw, then read registers $r1 and $r3
- Stage 3: add 17 to value in register $41 (retrieved in Stage 2)
- Stage 4: write value in register $r3 (retrieved in Stage 2) into memory address computed in Stage 3
- Stage 5: go idle (nothing to write into a register)

Example: sw Instruction



Why Five Stages? (1/2)

Could we have a different number of stages?

- Yes, and other architectures do

So why does MIPS have five if instructions tend to go idle for at least one stage?

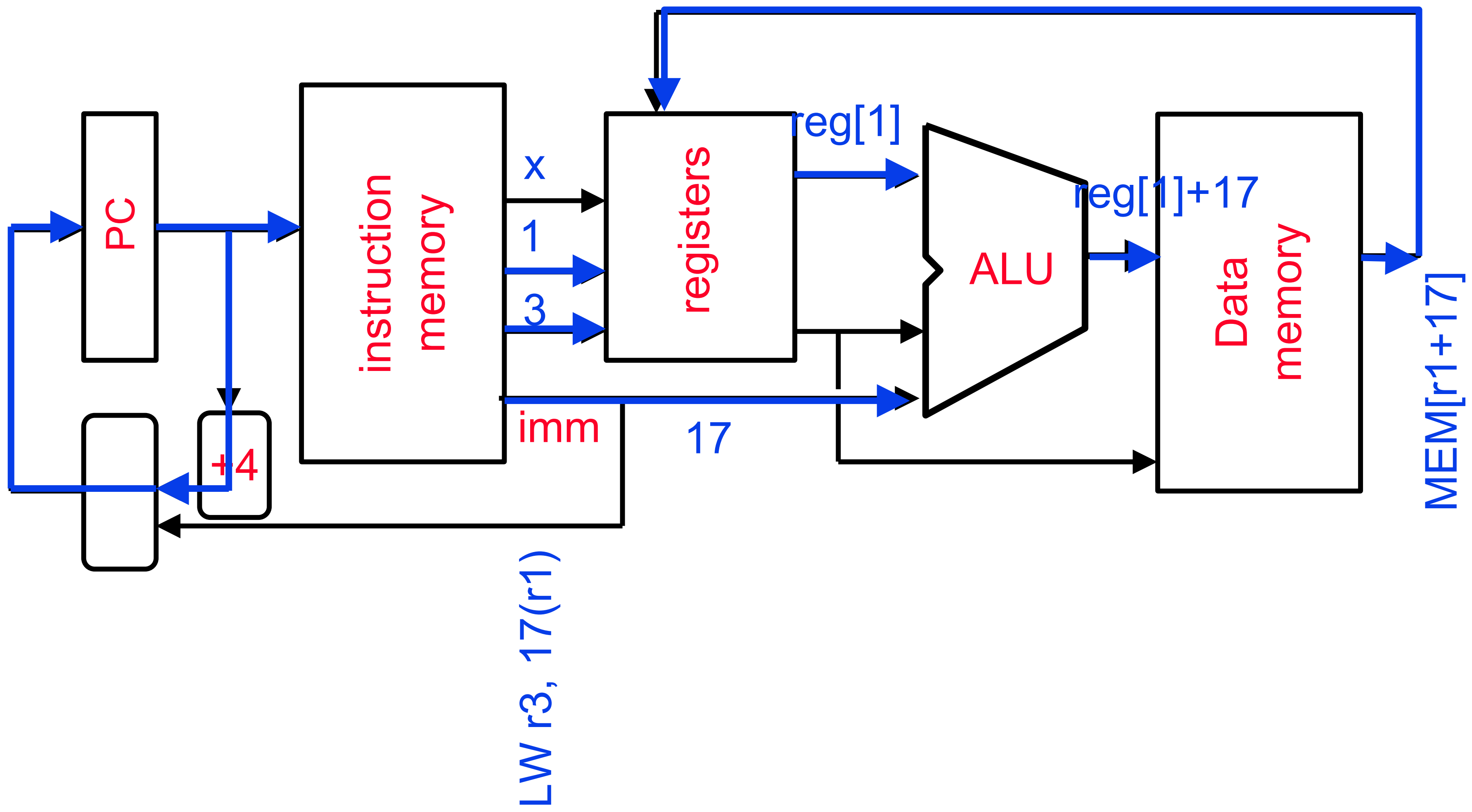
- There is one instruction that uses all five stages: the load

Why Five Stages? (2/2)

`lw   $r3, 17($r1)`

- Stage 1: fetch this instruction, inc. PC
- Stage 2: decode to find it's a `lw`, then read register `$r1`
- Stage 3: add 17 to value in register `$r1` (retrieved in Stage 2)
- Stage 4: read value from memory address compute in Stage 3
- Stage 5: write value found in Stage 4 into register `$r3`

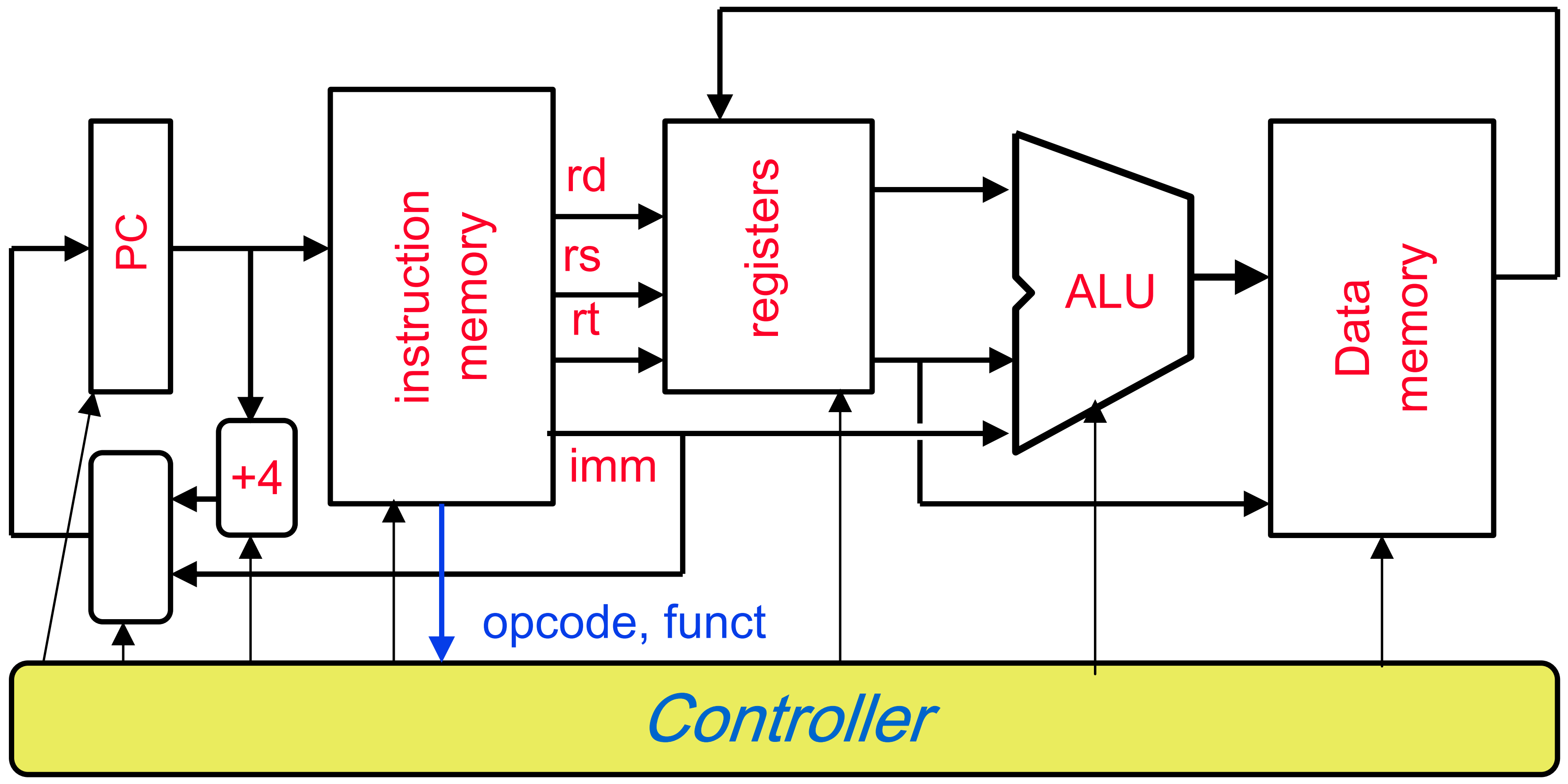
Example: `lw` Instruction



Datapath Summary

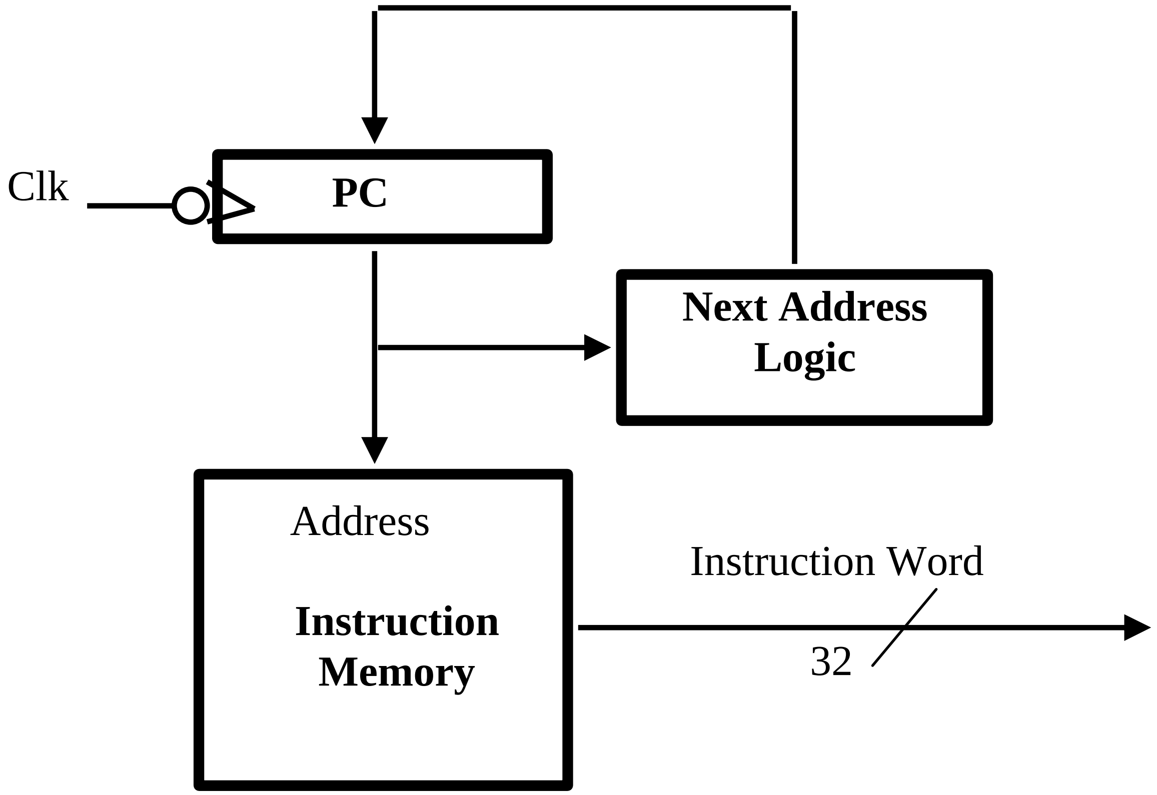
°The datapath based on data transfers required to perform instructions

°A *controller* causes the right transfers to happen



Overview of the Instruction Fetch Unit

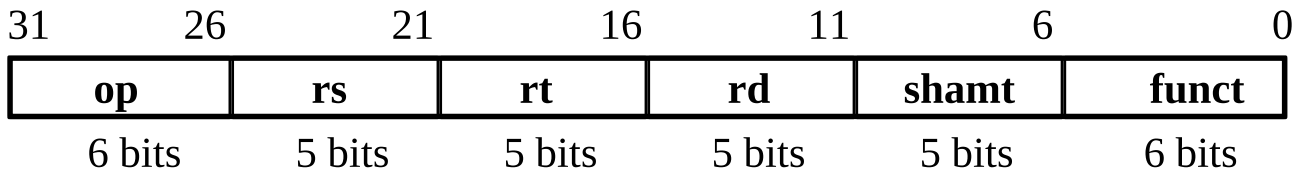
- **The common operations**
  - Fetch the Instruction: mem[PC]
  - Update the program counter:
    - Sequential Code: PC $\leftarrow$ PC + 4
    - Branch and Jump: PC $\leftarrow$ "something else"

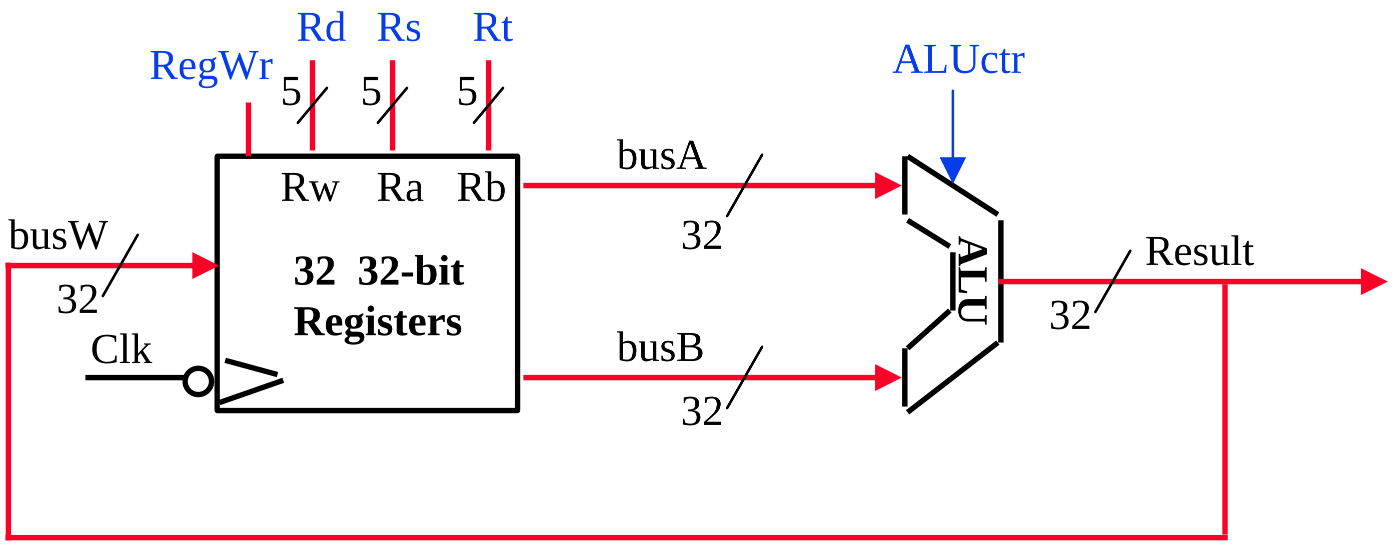


Add & Subtract

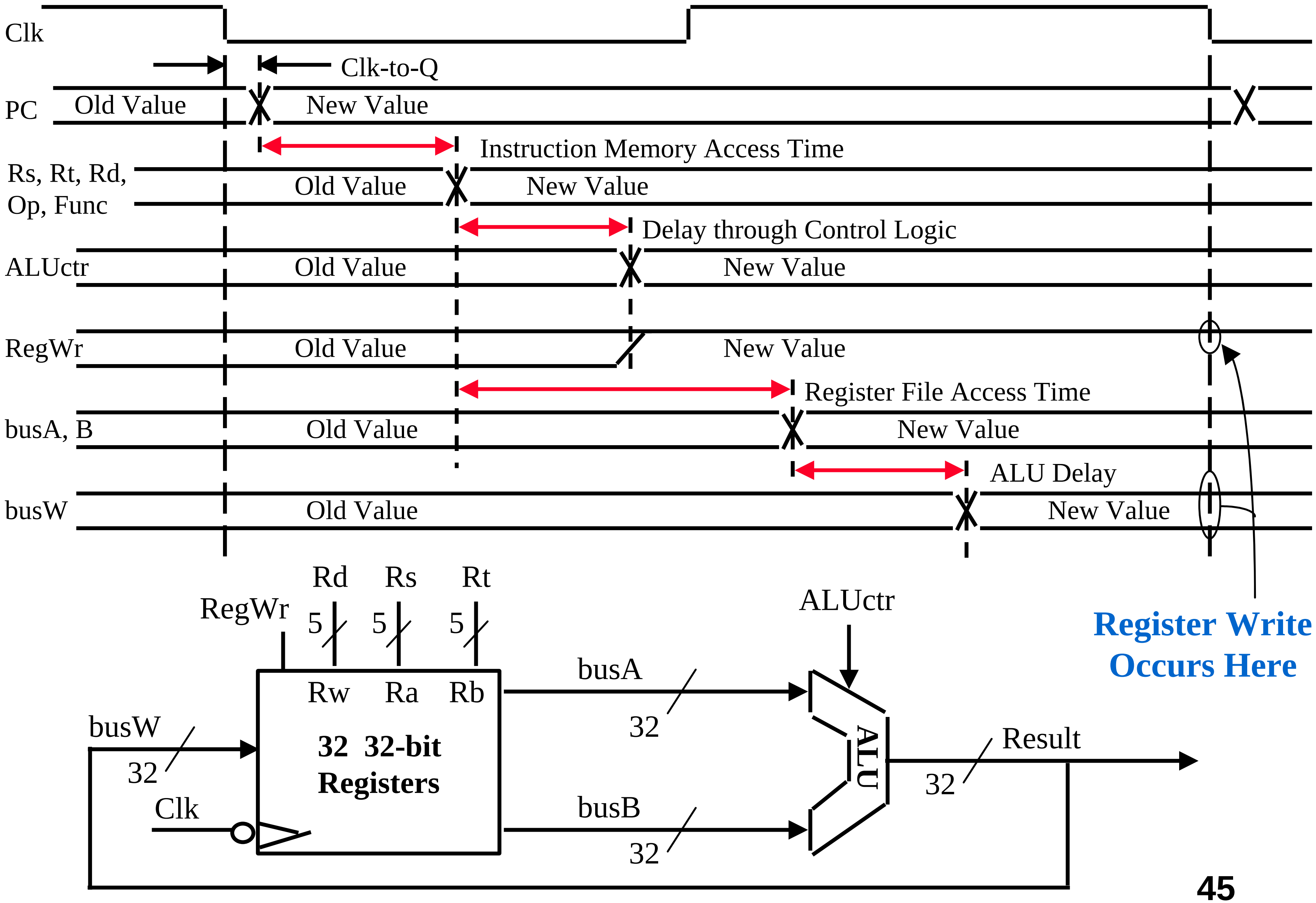
R[rd] ← R[rs] op R[rt]; Example: **`addu rd, rs, rt`**

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



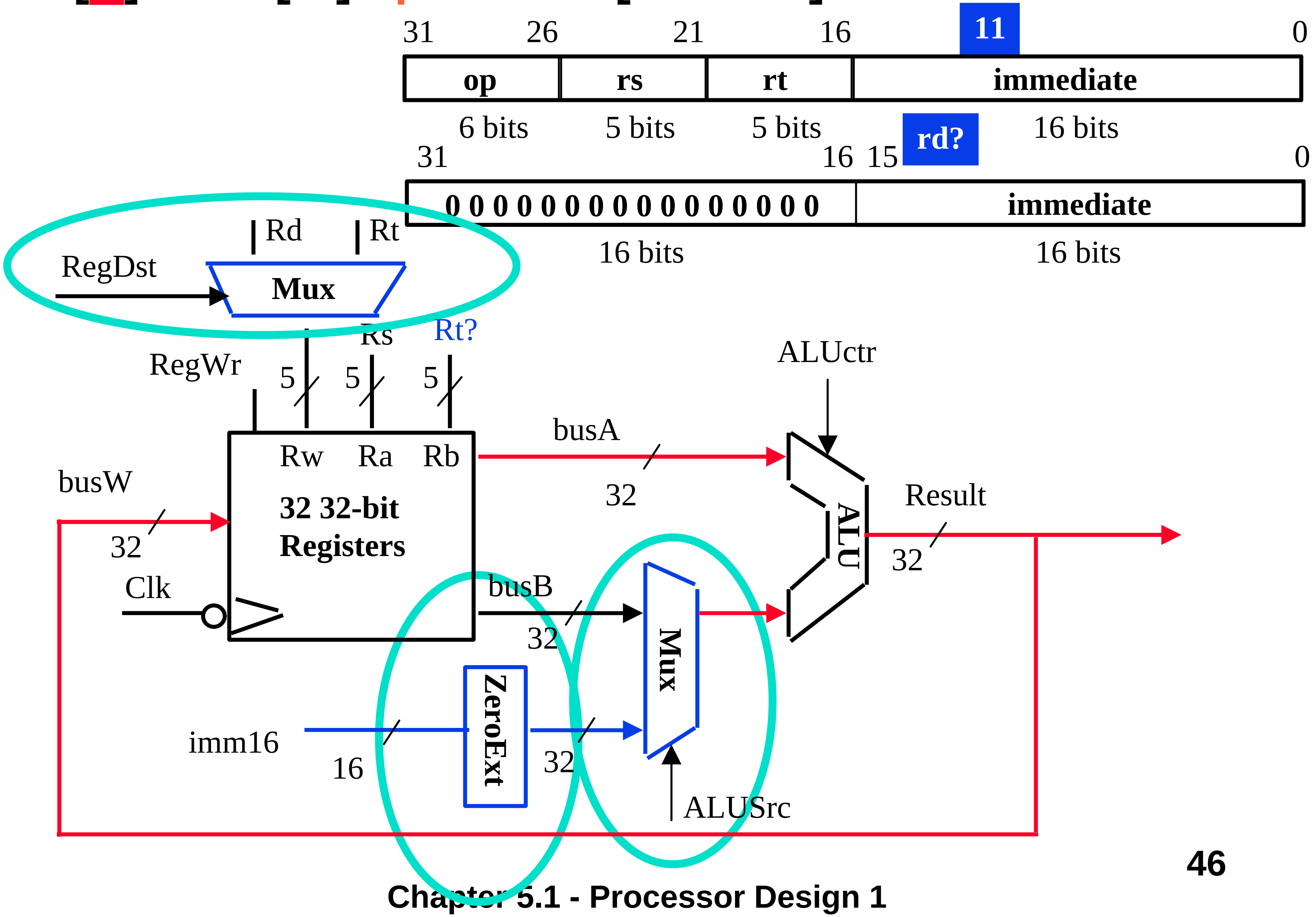


Register-Register Timing: One complete cycle



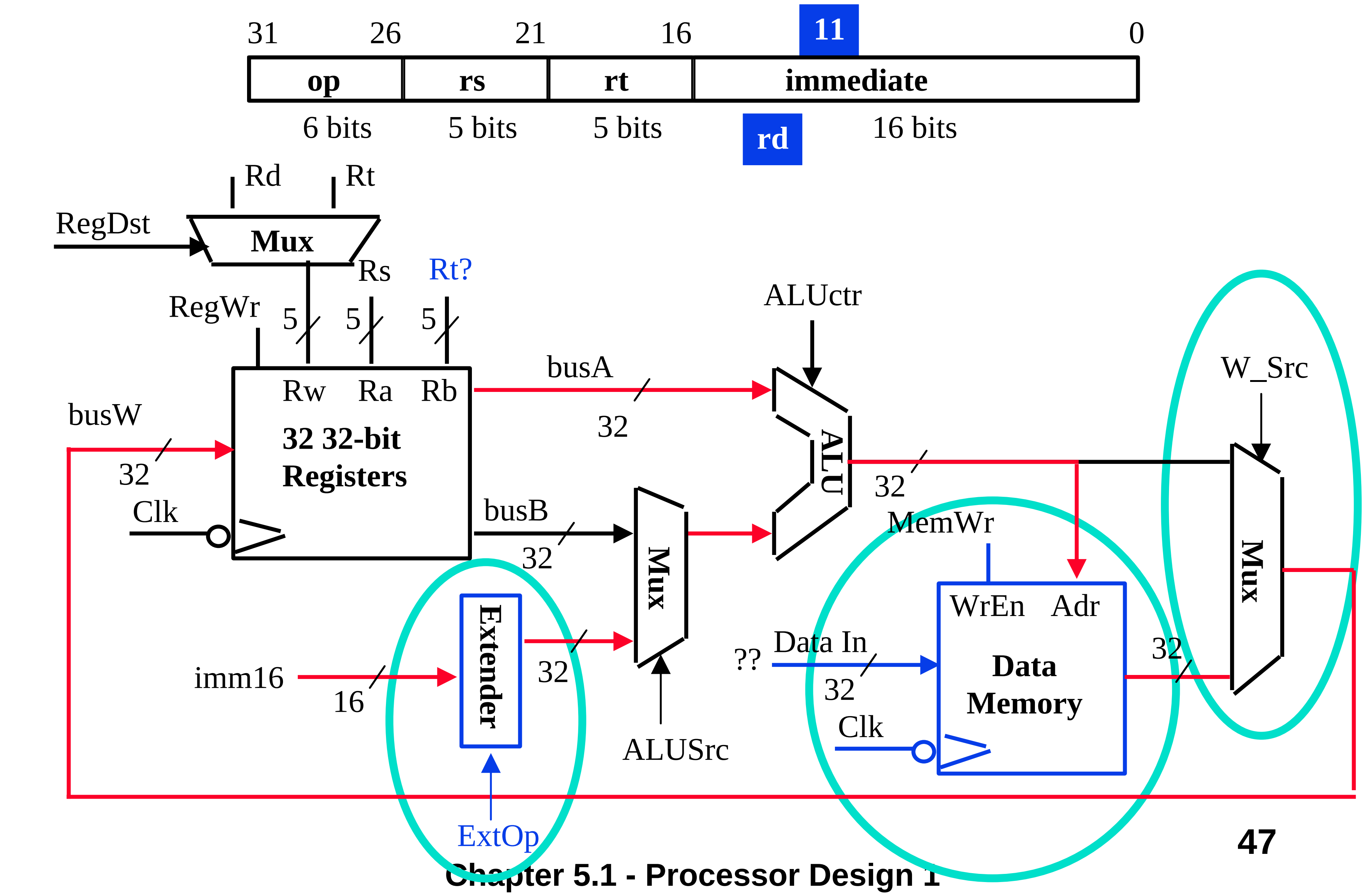
Logical Operations With Immediate

- R[rt] ← R[rs] op ZeroExt[imm16]



Load Operations

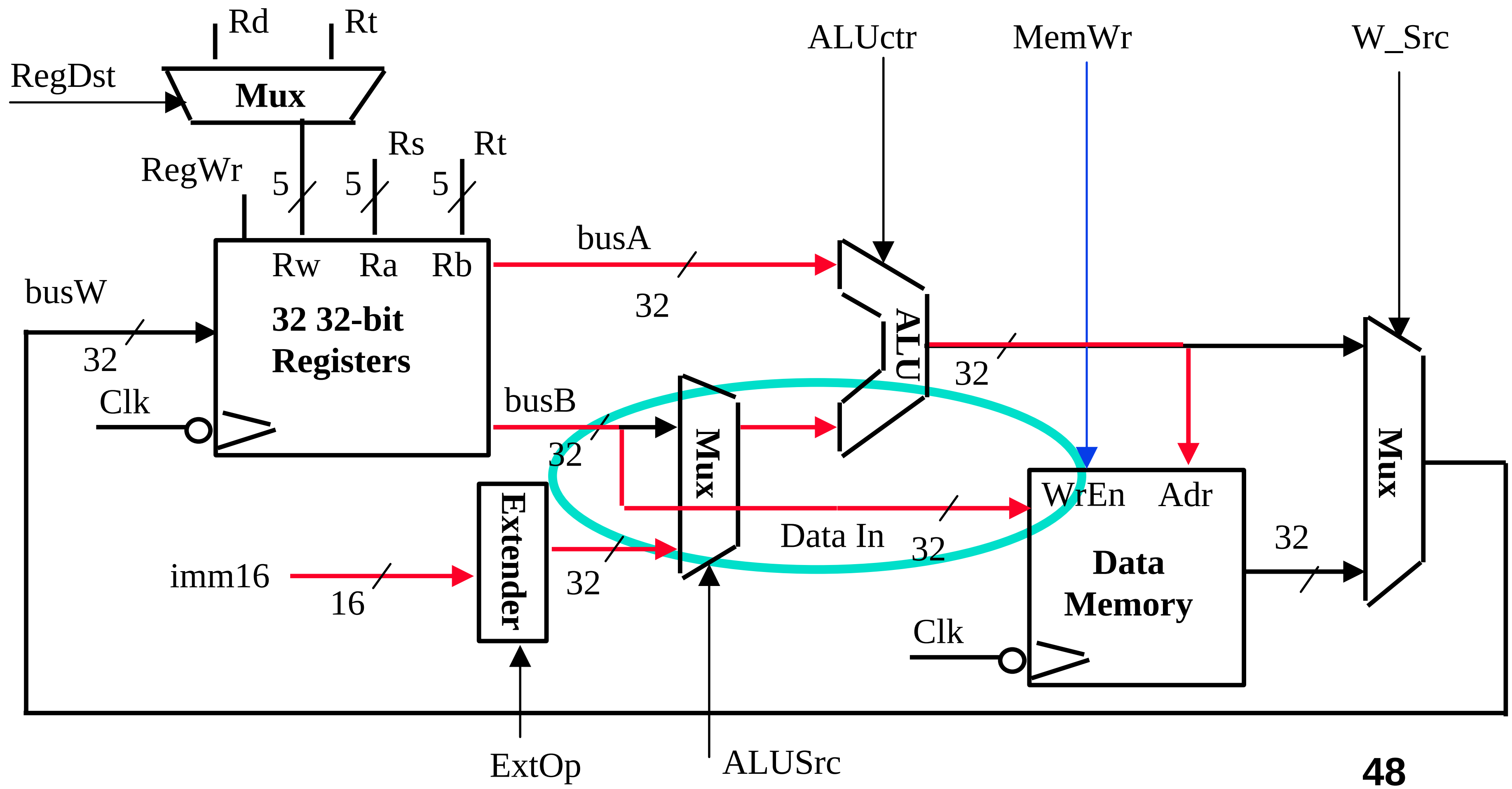
- R[rt] ← Mem[R[rs] + SignExt[imm16]]; Example: `lw rt, rs, imm16`



Store Operations

- **Mem[R[rs] + SignExt[imm16] ← R[rt]]; Example:** `sw rt, rs, imm16`

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



The Branch Instruction

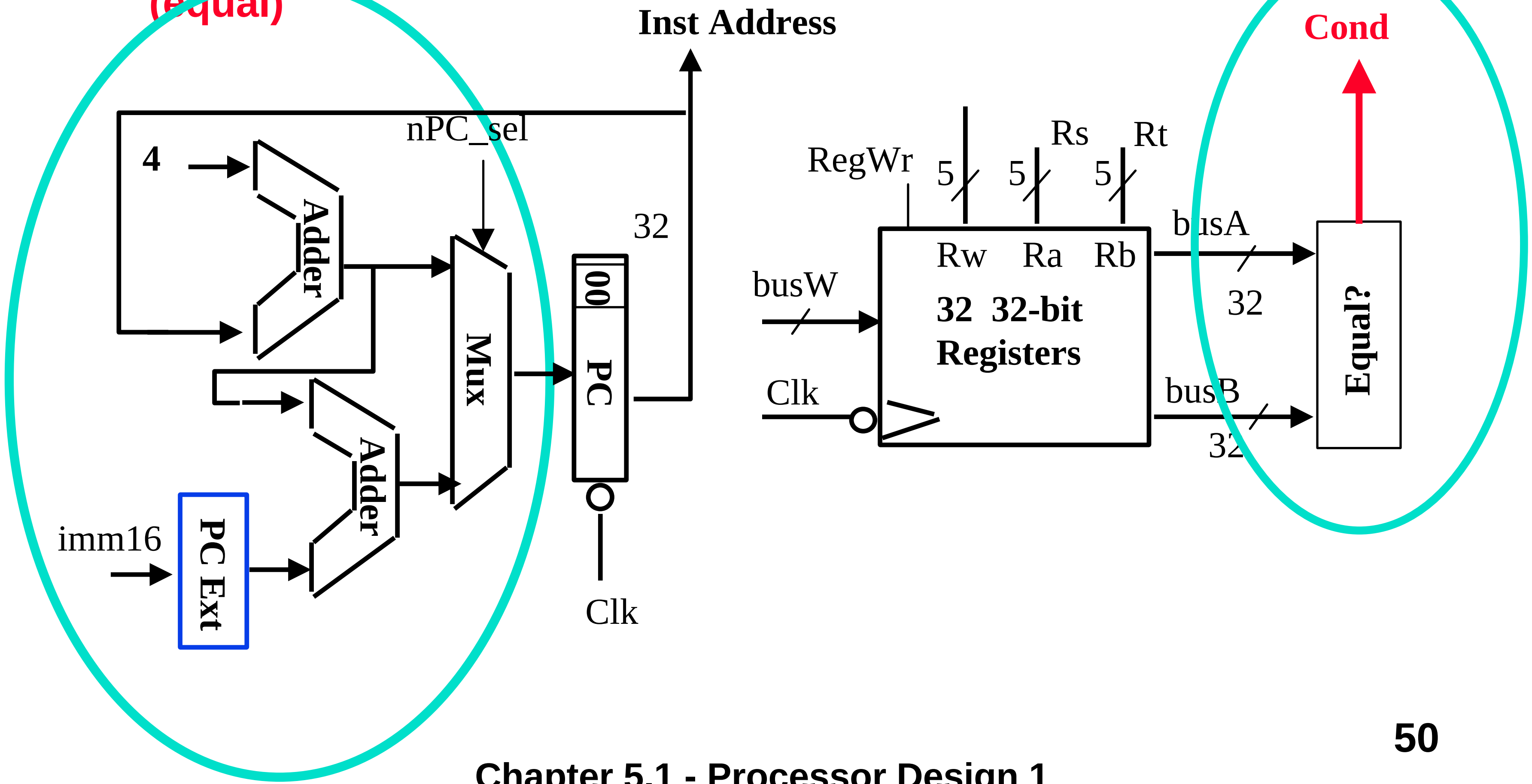
31	26	21	16	0
op	**rs**	**rt**	**immediate**	
6 bits	5 bits	5 bits	16 bits	

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

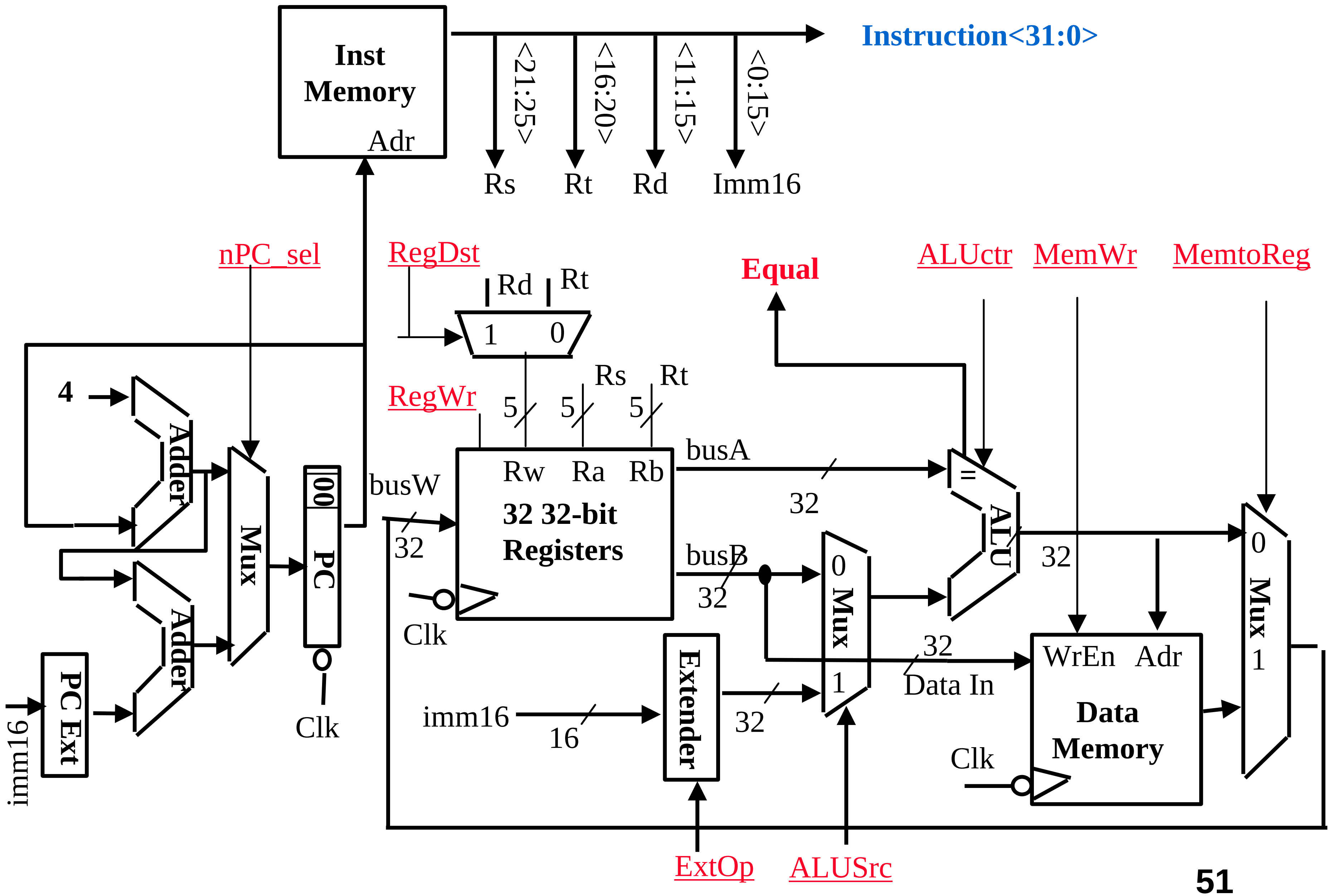
Datapath for Branch Operations

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

- **beq rs, rt, imm16** **Datapath generates condition (equal)**



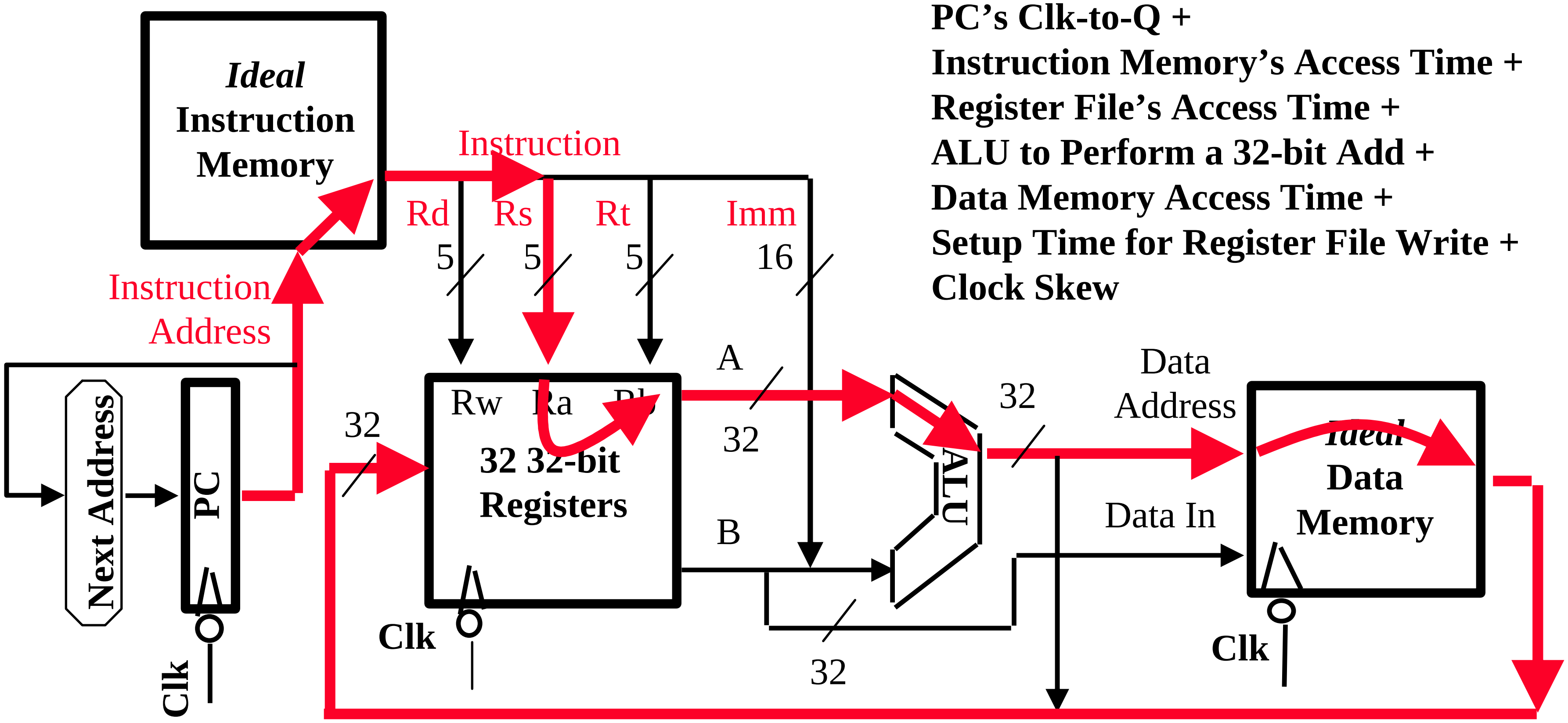
Summary: A Single Cycle Datapath



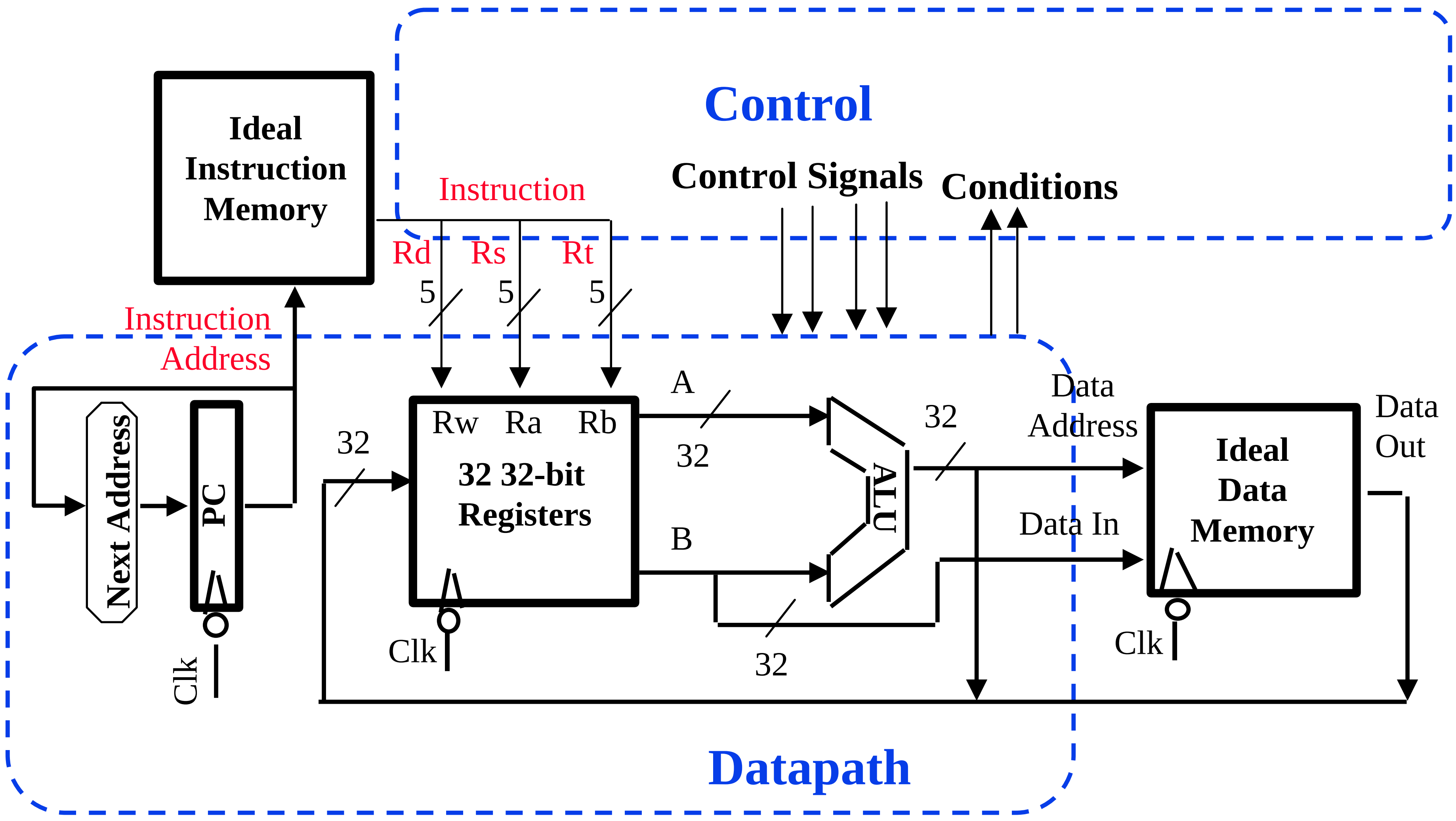
An Abstract View of the Critical Path

- **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."

Critical Path (Load Operation) =
PC's Clk-to-Q +
Instruction Memory's Access Time +
Register File's Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew



An Abstract View of the Implementation



Steps 4 & 5: Implement the control

In The Next Section

Summary: MIPS-lite Implementations

- **single-cycle: uses single l-o-n-g clock cycle for each instruction executed**
- **Easy to understand, but not practical**
  - slower than implementation that allows instructions to take different numbers of clock cycles
    - fast instructions: (`beq`) fewer clock cycles
    - slow instructions (`mult?`): more cycles
  - multicycle, pipelined implementations later
- **Next time, finish the single-cycle implementation**

Summary

- **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
- **MIPS makes it easier**
  - Instructions same size
  - Source registers always in same place
  - Immediates same size, location
  - Operations always on registers/immediates
- **Single cycle datapath: CPI = 1, T_{CC} $\rightarrow$ long**
- **Next time: implementing control**