CPU CLOCK & SINGLE-CYCLE DATAPATH PART I

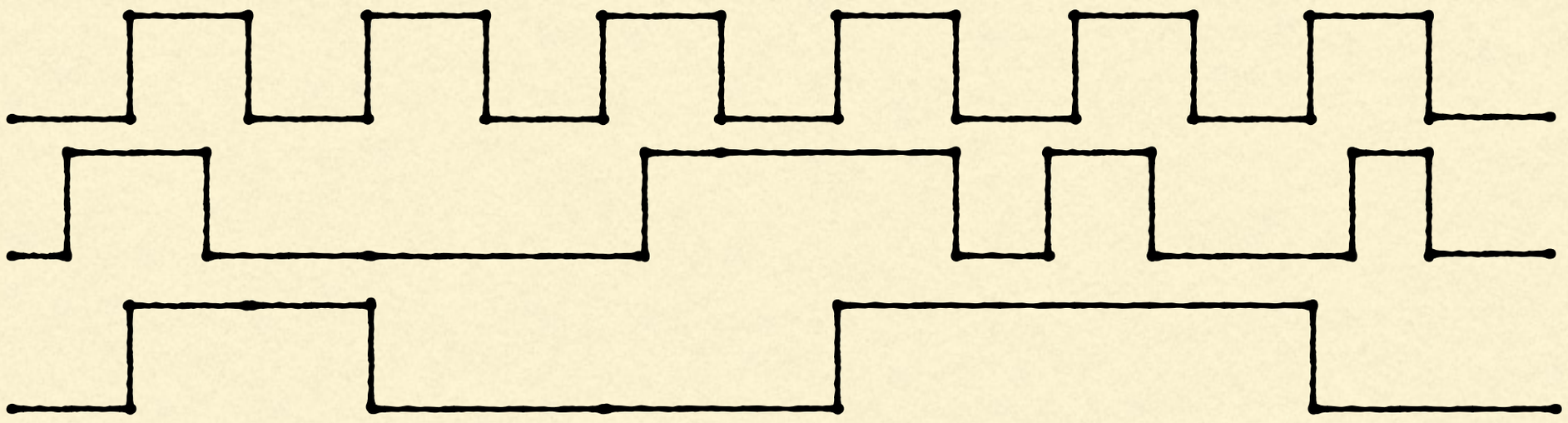
Ayman Hajja, PhD

INTRODUCING EDGE-TRIGGERED D-TYPE FLIP FLOP

- Edge-triggered d-type flip-flop

On the rising edge of the clock, the input d is sampled and transferred to the output. At all other times, the input d is ignored

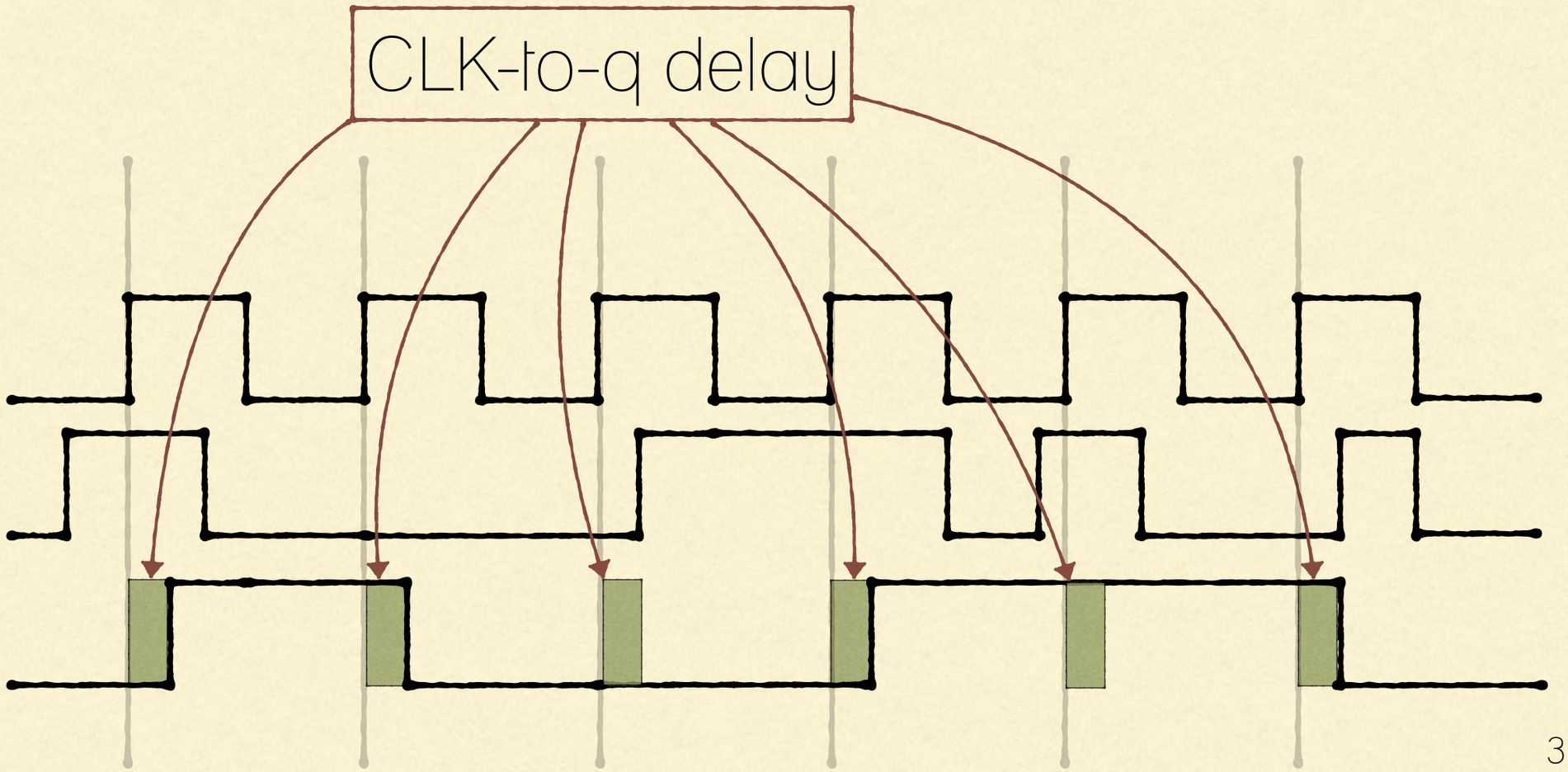
- Example waveform:
- CLK
- d (input)
- q (output)



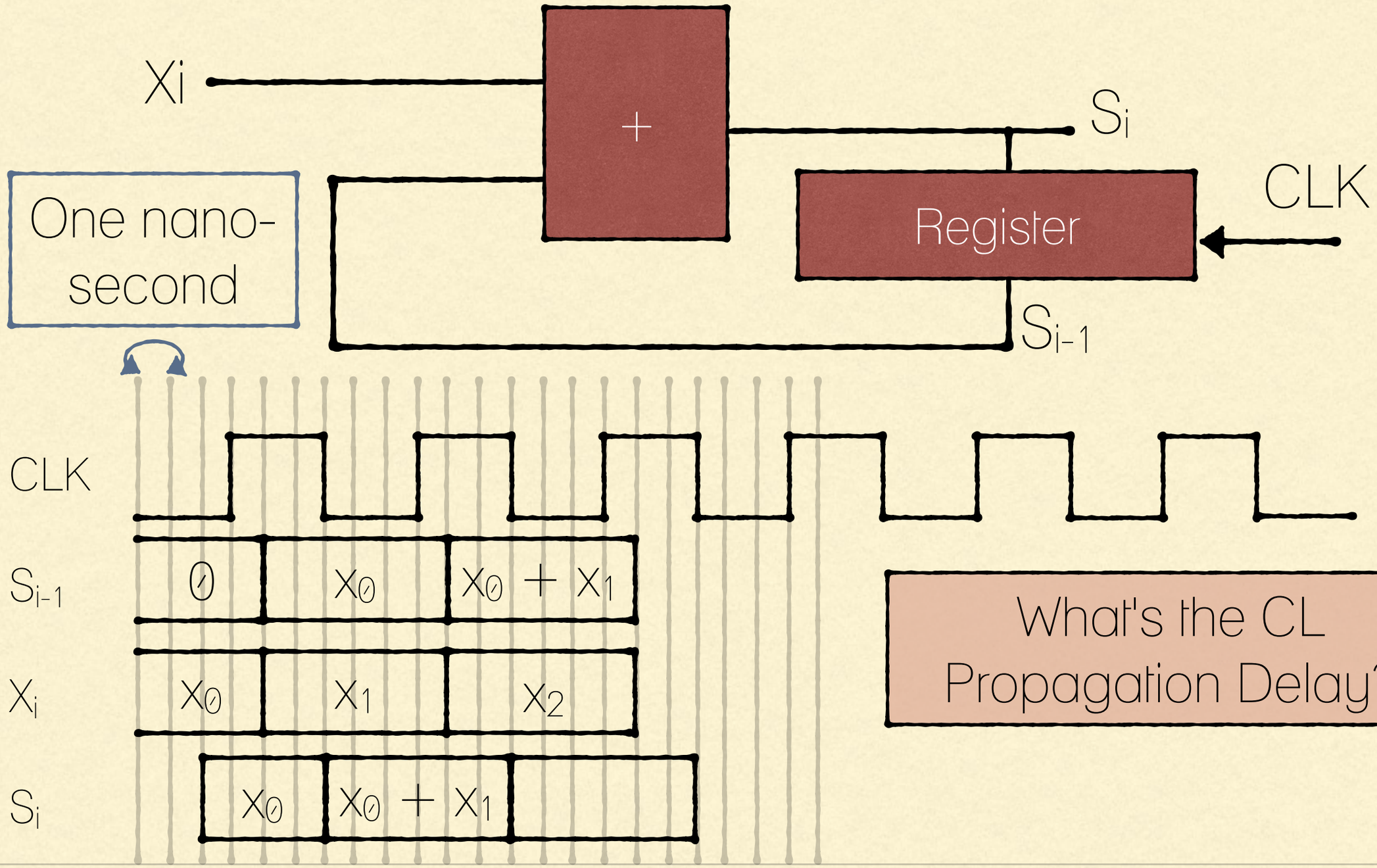
INTRODUCING EDGE-TRIGGERED D-TYPE FLIP FLOP

In reality, there's a delay in the flip-flop for the output to be reflected when the input changes

- CLK
- d (input)
- q (output)

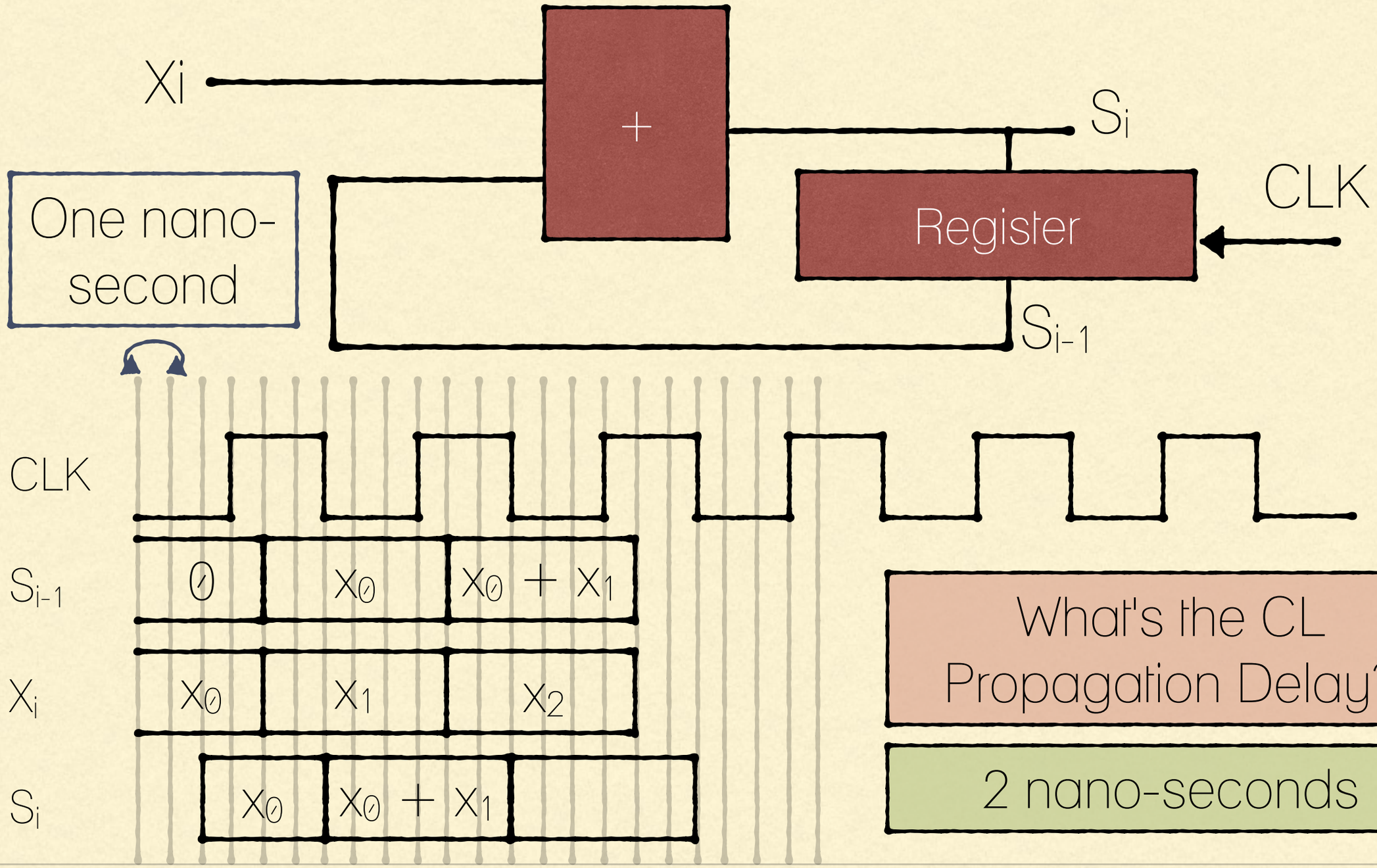


ACCUMULATOR EXAMPLE



What's the CL Propagation Delay?

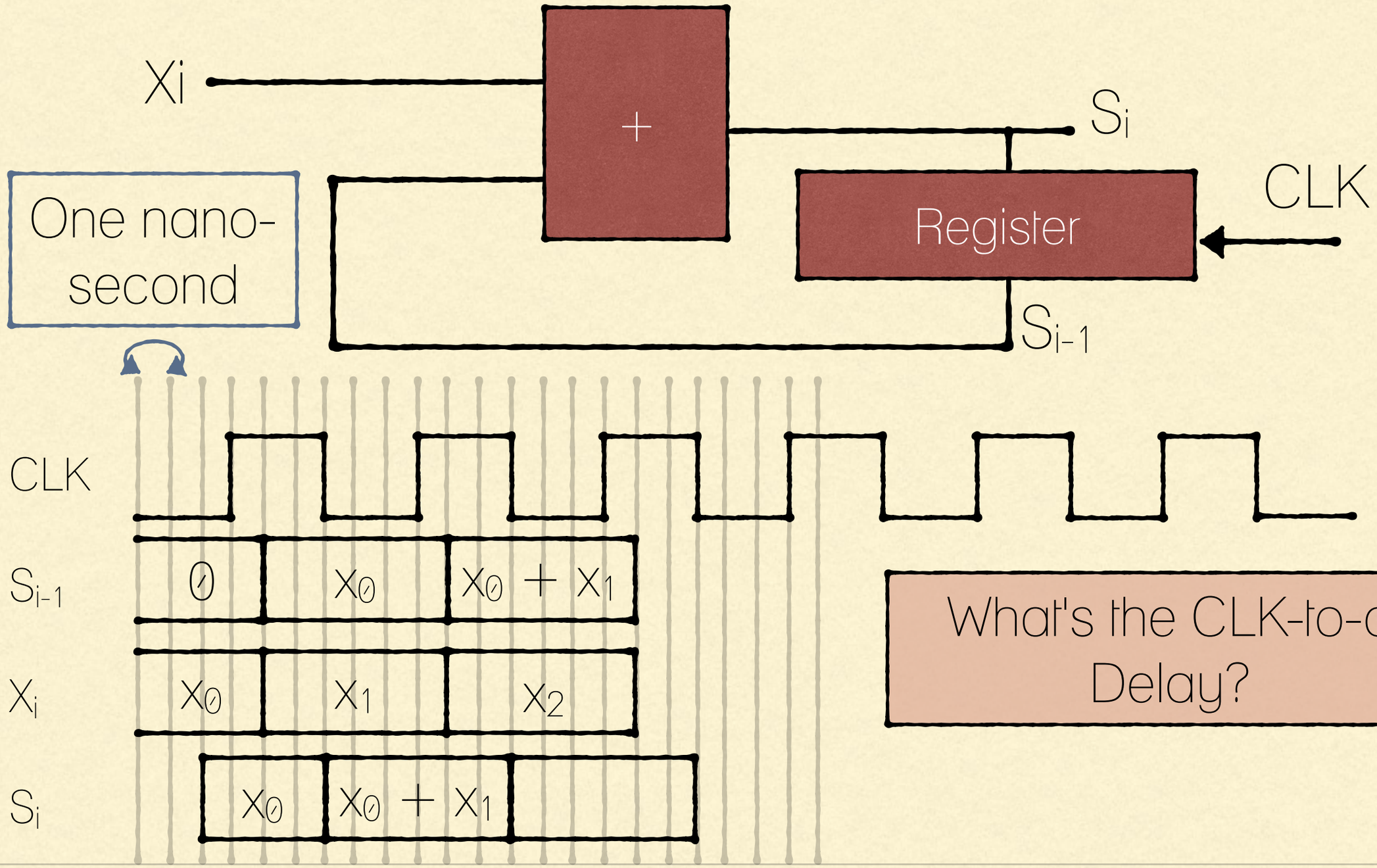
ACCUMULATOR EXAMPLE



What's the CL Propagation Delay?

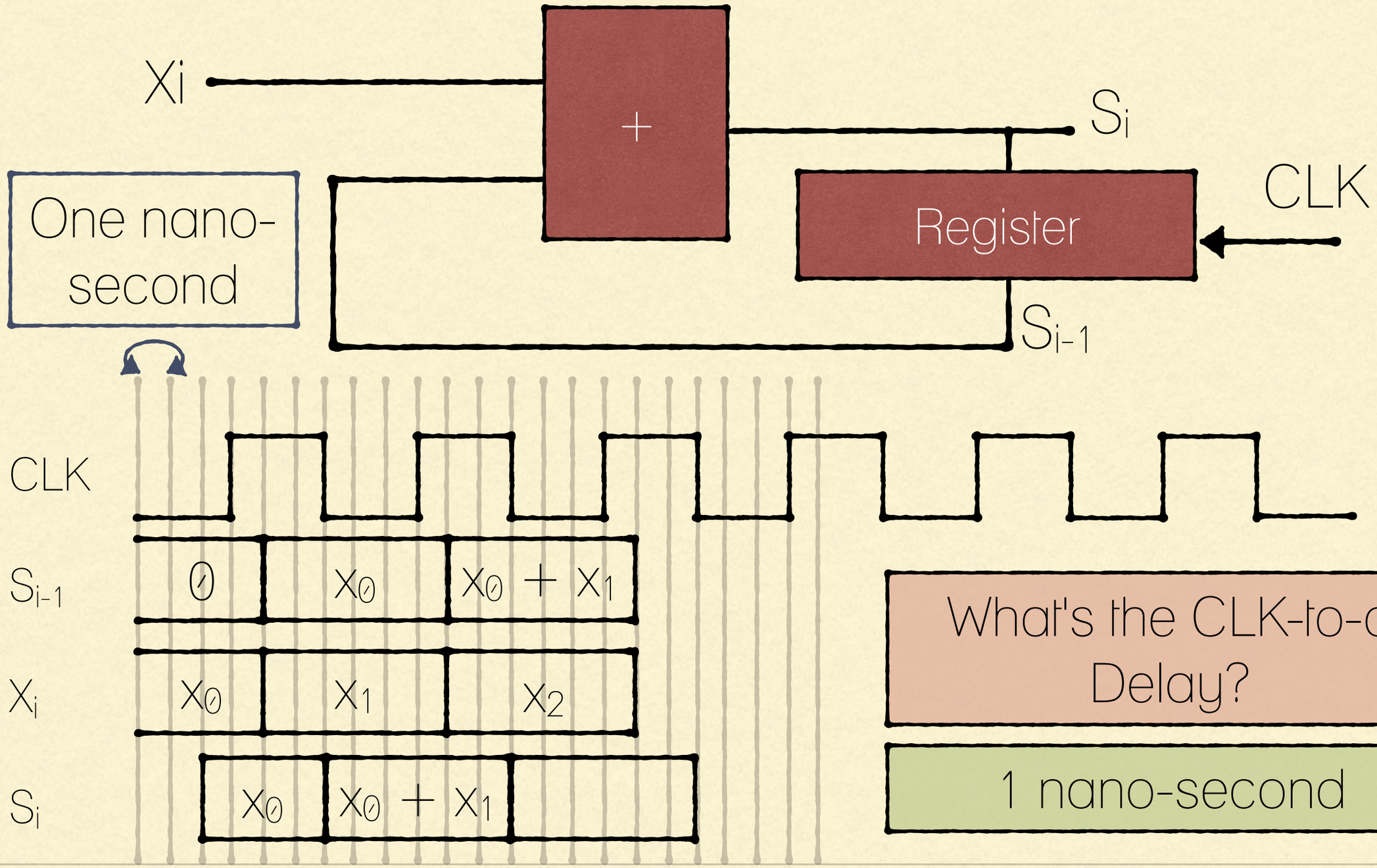
2 nano-seconds

ACCUMULATOR EXAMPLE

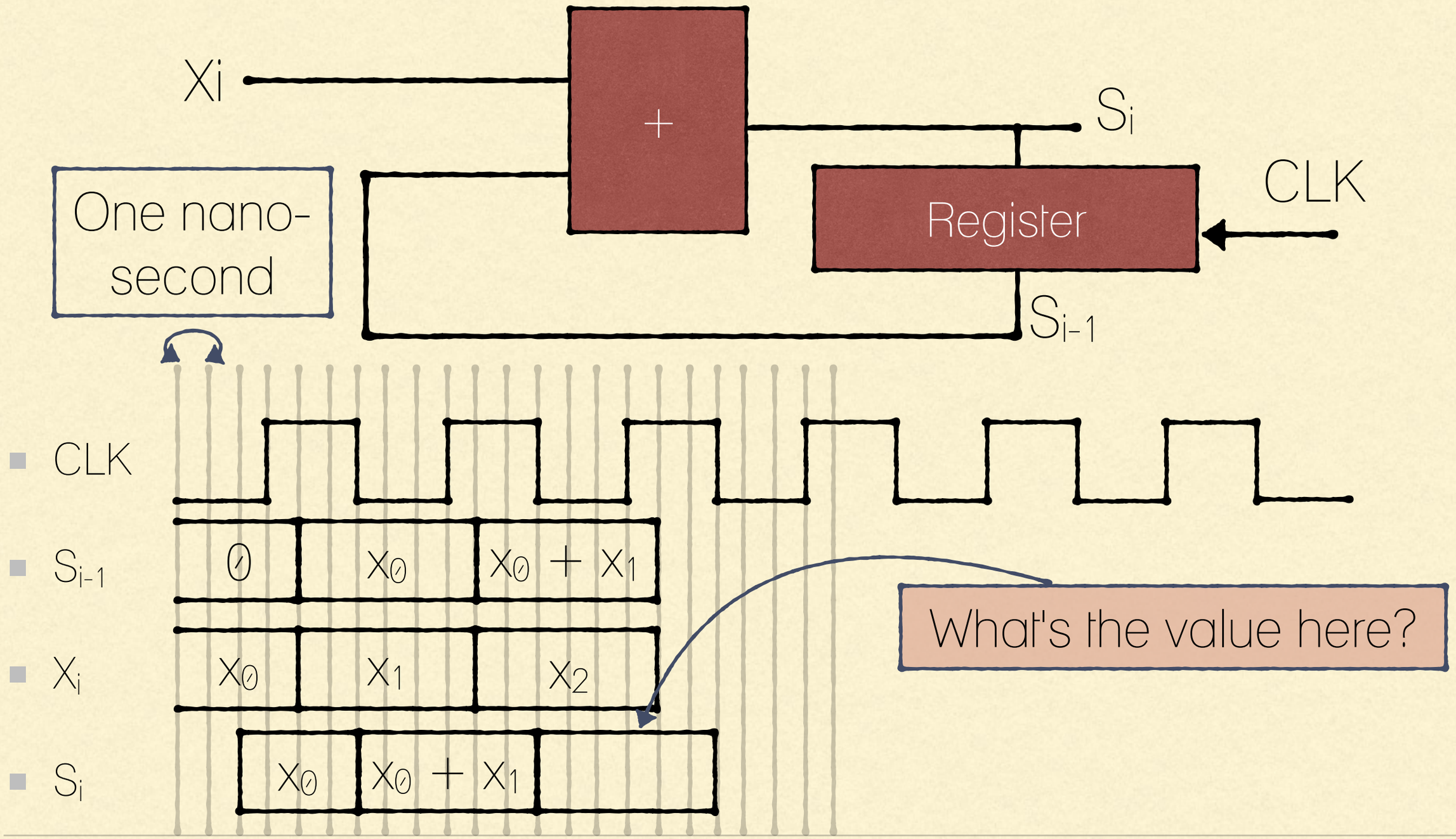


What's the CLK-to-q Delay?

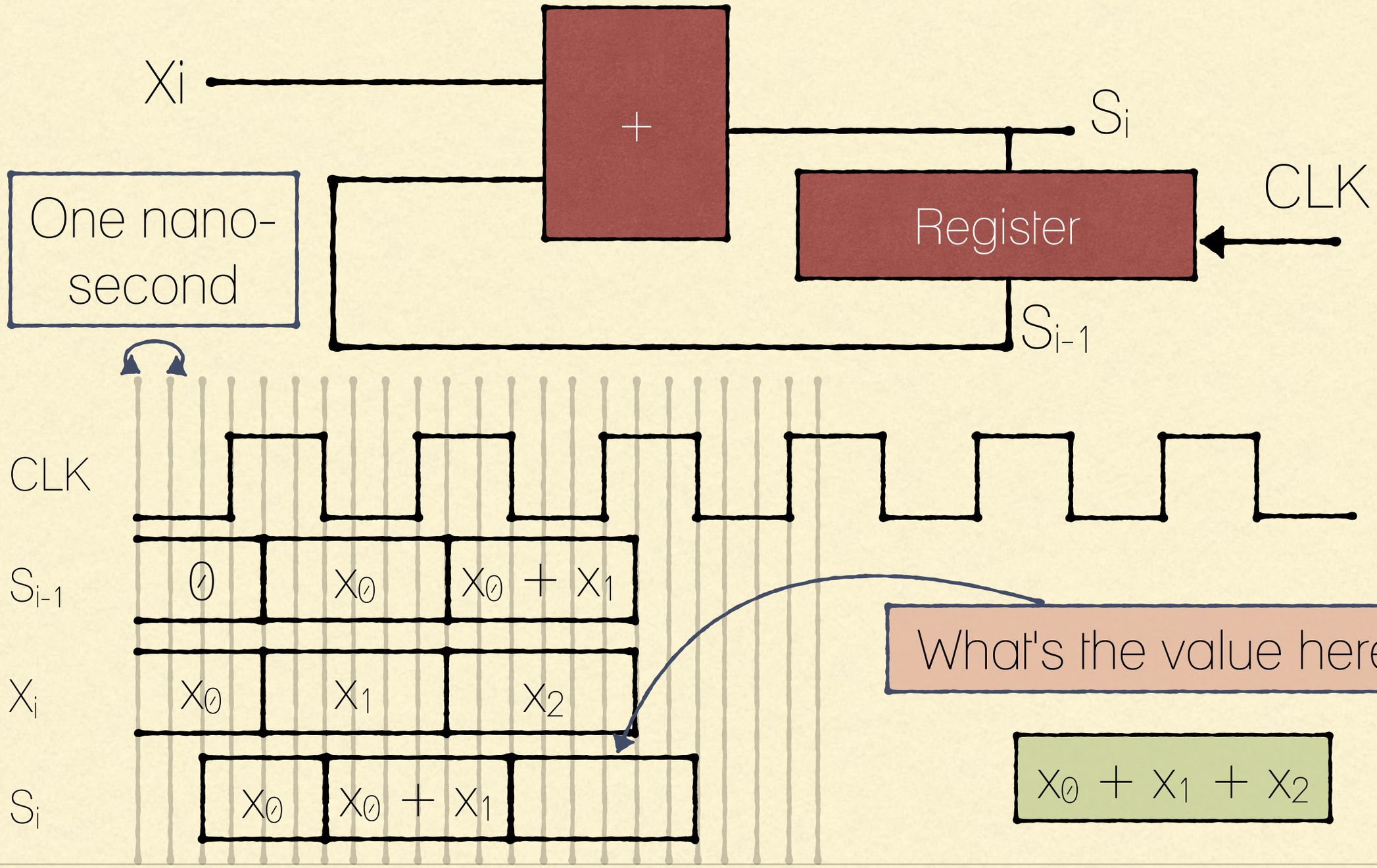
ACCUMULATOR EXAMPLE



ACCUMULATOR EXAMPLE



ACCUMULATOR EXAMPLE



HARDWARE TIMING TERMS

- Setup Time; when the input must be stable before the edge of the CLK

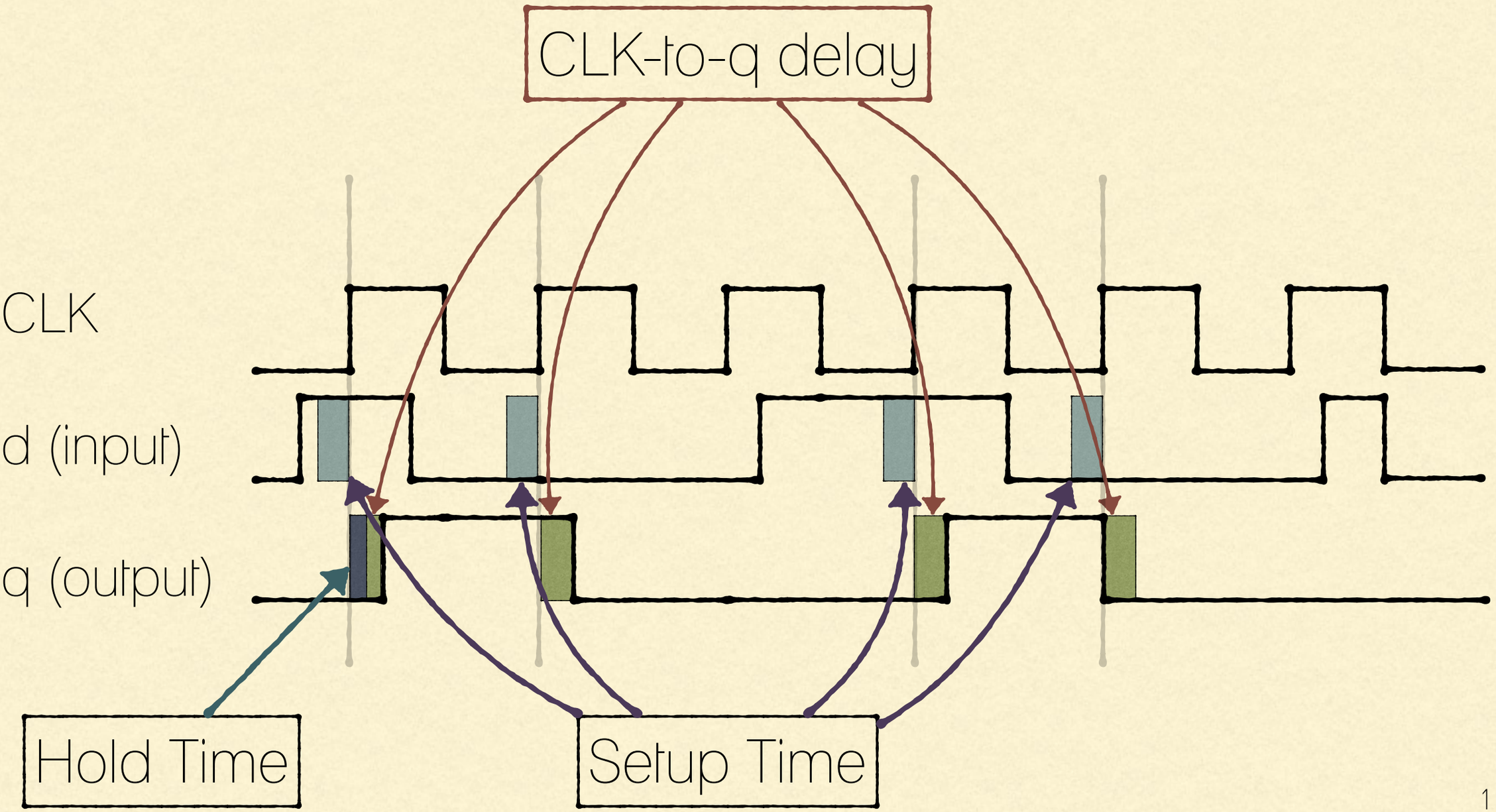
- Hold Time; when the input must be stable after the edge of the CLK

- CLK-to-Q Delay; how long it takes the output to change, measured from the edge of the CLK

    - Hold Time is part of the CLK-to-Q Delay

INTRODUCING EDGE-TRIGGERED D-TYPE FLIP FLOP

- CLK
- d (input)
- q (output)



CAMERA ANALOGY TIMING TERMS

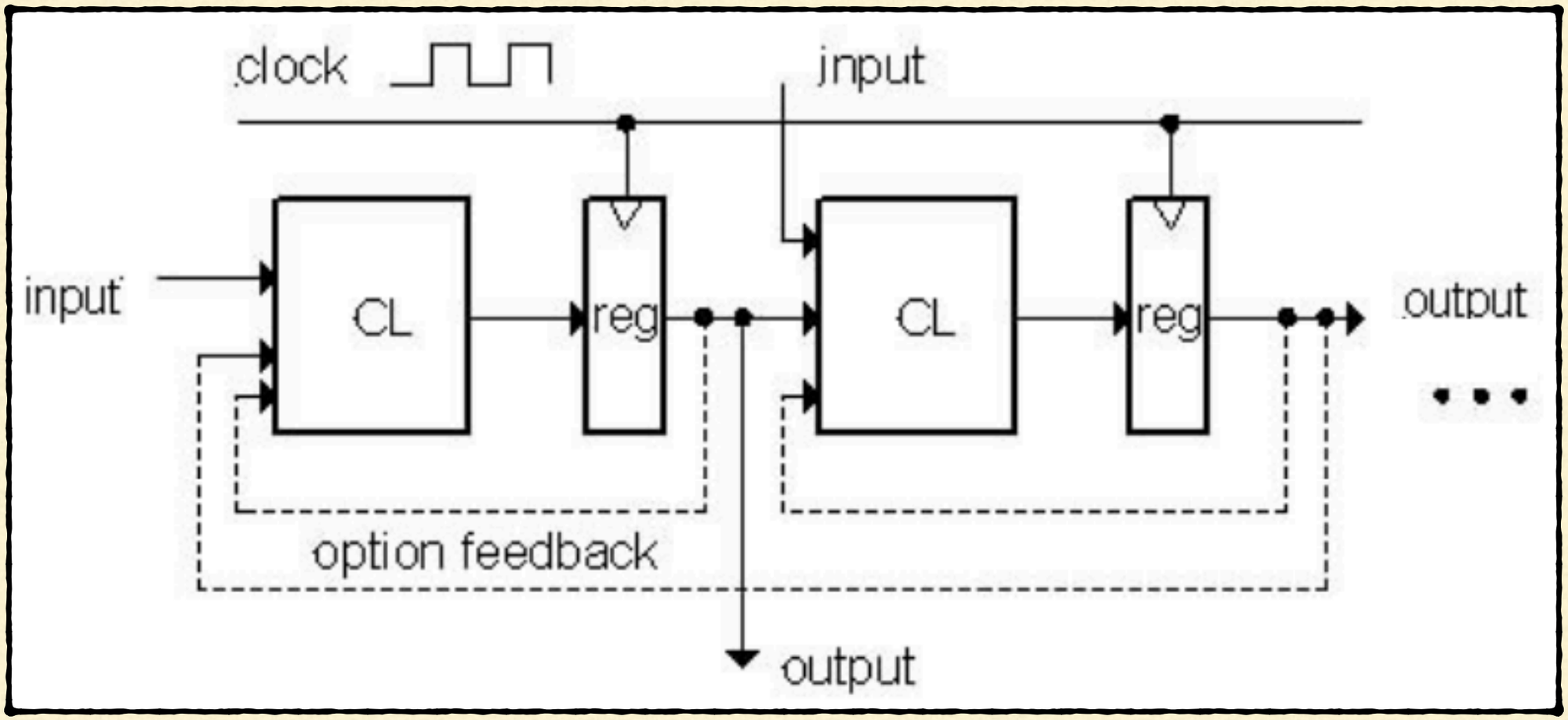
- Want to take a portrait — timing right before and after taking picture

- Setup time — don't move since about to take picture (open camera shutter)

- Hold time — need to hold still after shutter opens until camera shutter closes

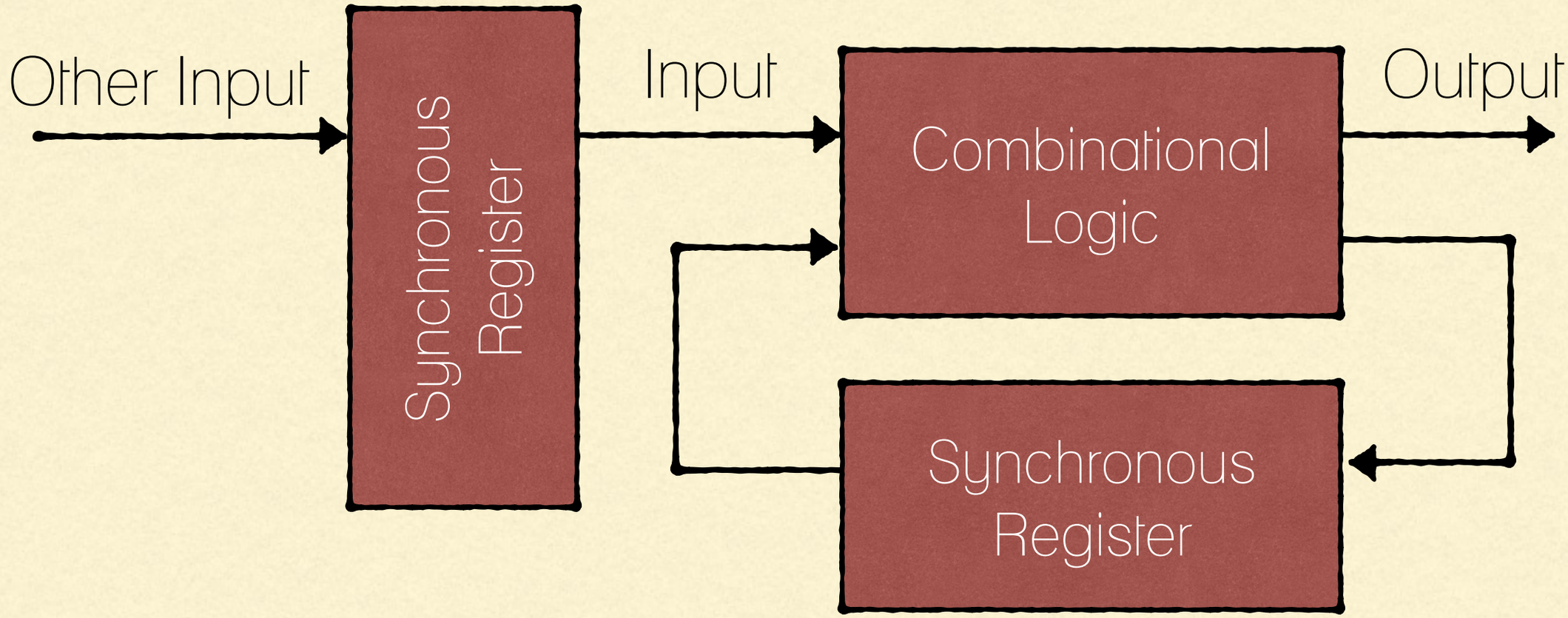
- Time click to data — time from open shutter until can see image on output (view-screen)

MODEL FOR SYNCHRONOUS SYSTEM



- Collection of Combinational Logic blocks separated by registers
- Clock signal(s) connects only to clock input of registers
- Clock (CLK): steady square wave that synchronizes the system

MAXIMUM CLOCK FREQUENCY

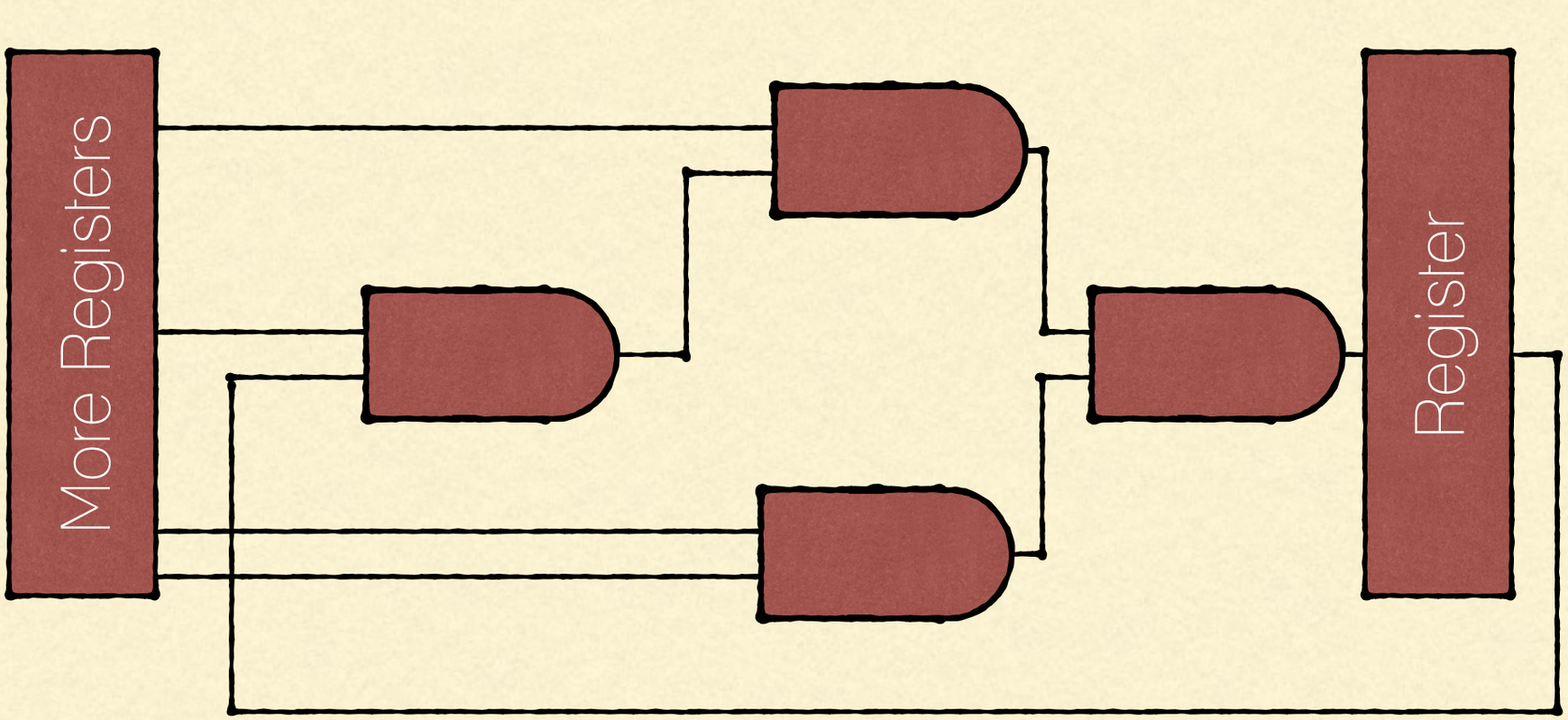


- Total Delay (seconds) = Setup Time + CLK-to-Q Delay + CL Delay
- Frequency =(Hz) 1/Period

MAXIMUM CLOCK FREQUENCY

- Period: 1 second. Frequency: 1 Hertz
- Period: 1 millisecond (10^{-3} sec). Frequency: 1 Kilohertz
- Period: 1 microsecond (10^{-6} sec). Frequency: 1 Megahertz
- Period: 1 nanoseconds (10^{-9} sec). Frequency: 1 Gigahertz

QUESTION



CLK-to-q delay: 2ns
Setup: 1ns
Hold: 1ns
'AND' delay: 1ns

What's the maximum clock frequency?

A: 5 GHz

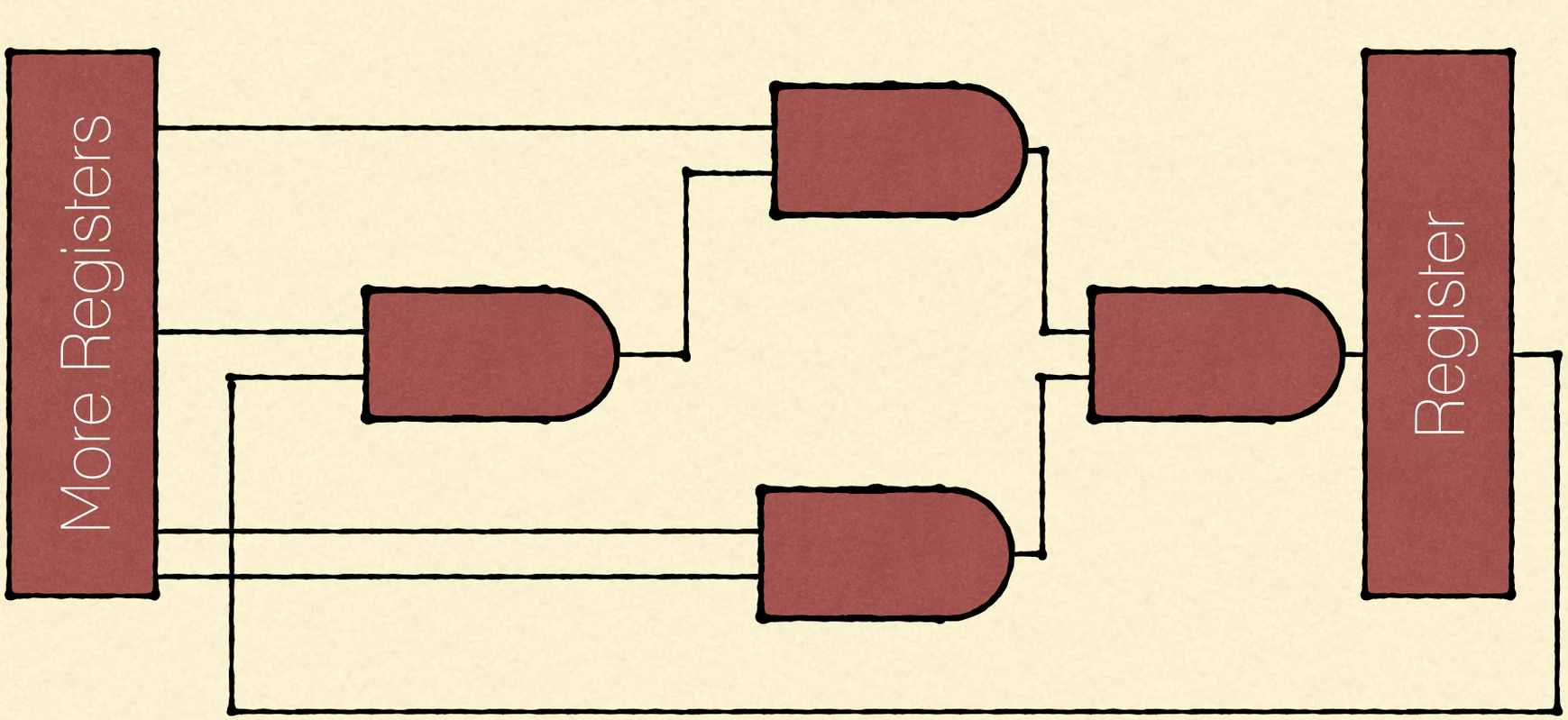
B: 200 MHz

C: 500 MHz

D: 1/7 GHz

E: 1/6 GHz

QUESTION



CLK-to-q delay: 2ns
Setup: 1ns
Hold: 1ns
'AND' delay: 1ns

What's the maximum clock frequency?

A: 5 GHz

B: 200 MHz

C: 500 MHz

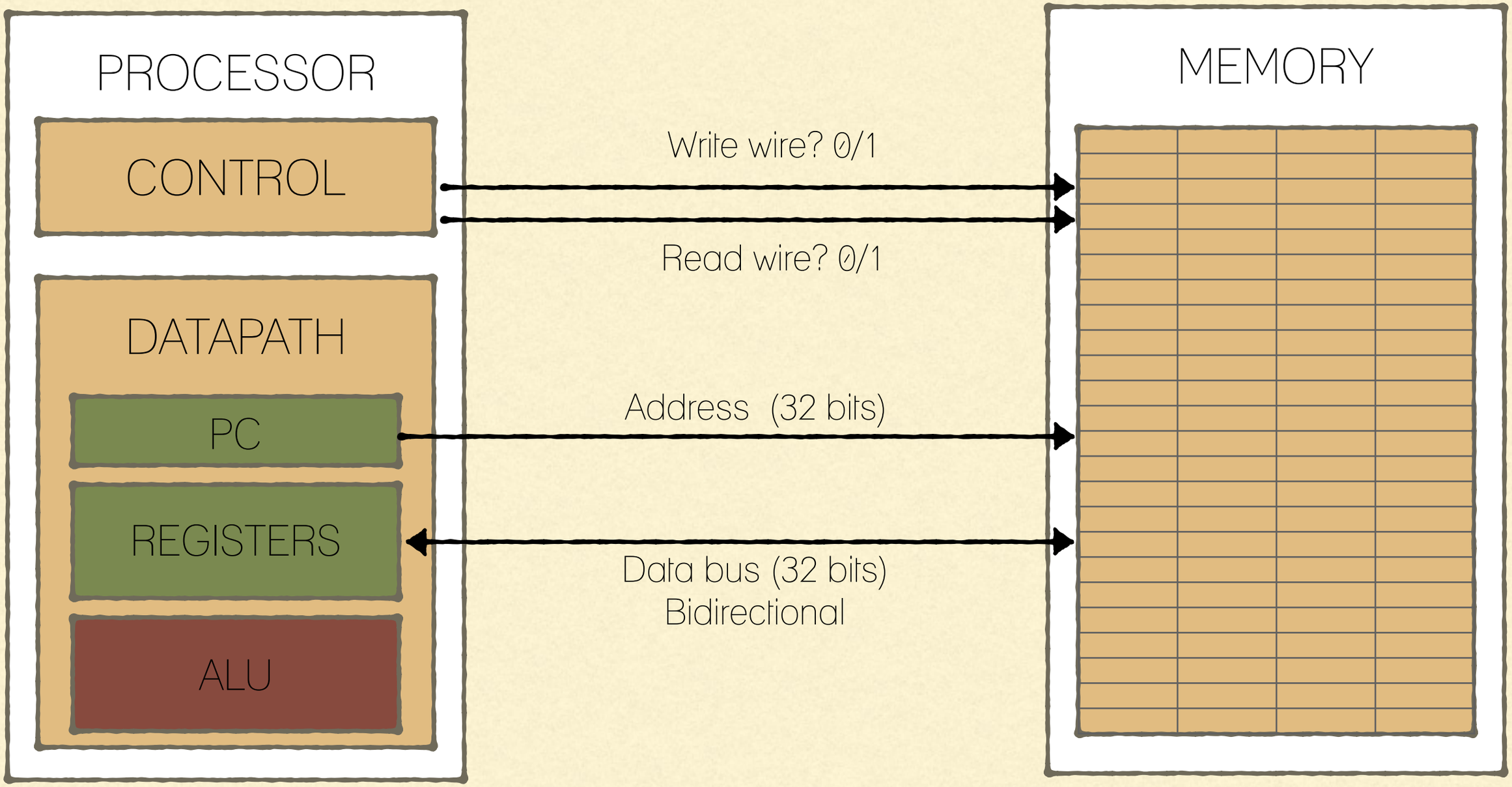
D: 1/7 GHz

E: 1/6 GHz

THE CPU

- Processor (CPU): the active part of the computer that does all the work (data manipulation and decision-making)
  - Datapath: portion of the processor that contains hardware necessary to perform operations required by the processor
  - Control: portion of the processor (also in hardware) that tells the datapath what needs to be done (the brain)

MEMORY ADDRESSES ARE IN BYTES



FIVE STAGES OF INSTRUCTION EXECUTION

- Stage 1: Instruction Fetch
- Stage 2: Instruction Decode (and Register Read)
- Stage 3: ALU (Arithmetic-Logic Unit)
- Stage 4: Memory Access
- Stage 5: Register Write

STAGES OF EXECUTION

- Stage 1: Instruction Fetch
  - Regardless of what the instruction is, the 32-bit instruction word must be first fetched from memory
  - This is where we increment the PC (PC = PC + 4, to point to the next instruction)

STAGES OF EXECUTION

- Stage 2: Instruction Decode (and Register Read)
    - After fetching the instruction, we gather data from the fields (decode all necessary instruction data)

STAGES OF EXECUTION

- Stage 2: Instruction Decode (and Register Read)
    - After fetching the instruction, we gather data from the fields (decode all necessary instruction data)
    - In this stage, we;
        1. Read the opcode to determine instruction type and field lengths
        2. Second, read in data from all necessary registers:
            - Provide an example of an instruction that reads 2 registers?
            - Provide an example of an instruction that reads 1 register?
            - Provide an example of an instruction that reads `0` registers?

STAGES OF EXECUTION

- Stage 2: Instruction Decode (and Register Read)
    - After fetching the instruction, we gather data from the fields (decode all necessary instruction data)
    - In this stage, we;
        1. Read the opcode to determine instruction type and field lengths
        2. Second, read in data from all necessary registers:
            - For add, read two registers
            - For addi, read one register
            - For j, no reads necessary

STAGES OF EXECUTION

- Stage 3: ALU (Arithmetic Logic Unit)
  - The real work of most instructions is done here: arithmetic (+, -, *, and /), shifting, logic (&, |), comparisons (slt)
- What about loads (e.g. load word) and stores (e.g. store word)?

STAGES OF EXECUTION

- Stage 3: ALU (Arithmetic Logic Unit)
  - The real work of most instructions is done here: arithmetic (+, -, *, and /), shifting, logic (&, |), comparisons (slt)
- What about loads (e.g. load word) and stores (e.g. store word)?
  - Yes. They both use the Arithmetic Logic Unit

STAGES OF EXECUTION

- Stage 4: Memory Access
  - Only the load and store instructions do anything during this stage; the others remain idle during this stage or skip it all together
  - Since these instructions have a unique step, we need this extra stage to account for them

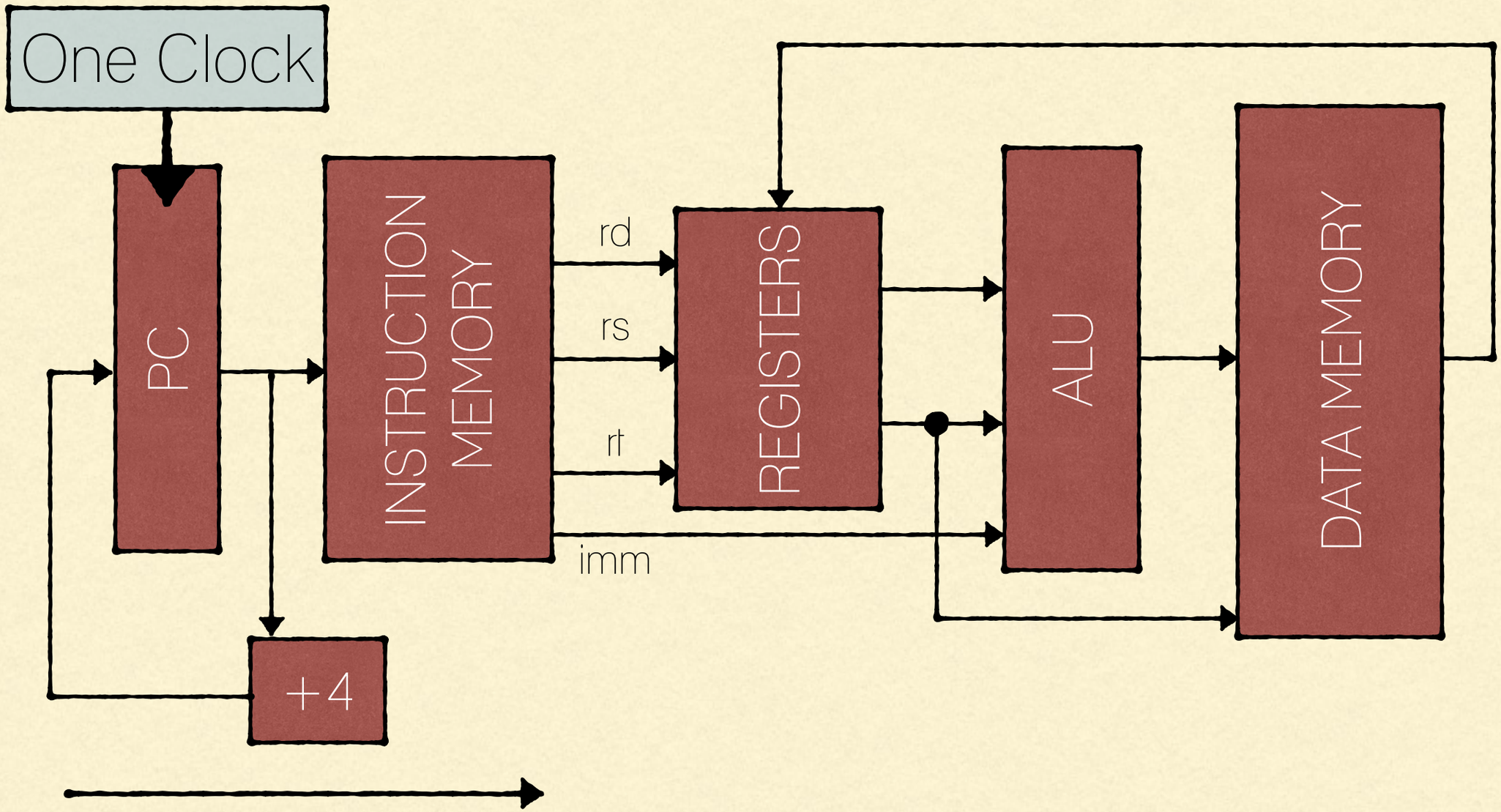
STAGES OF EXECUTION

- Stage 5: Register Write
  - Most instructions write the result of some computation into a register (arithmetic, logical, shifts, loads, slt)
  - How about stores, branches, and jumps?

STAGES OF EXECUTION

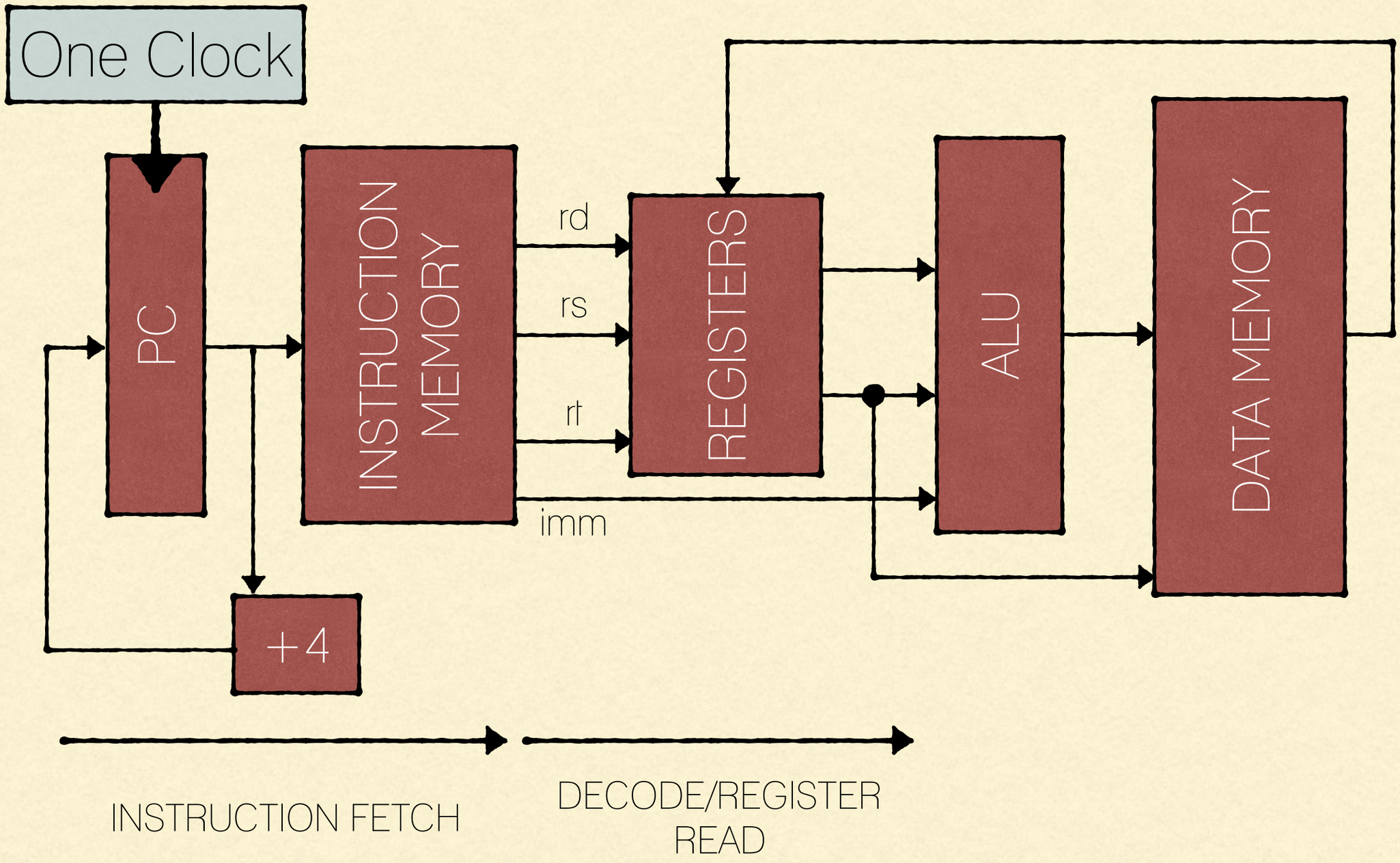
- Stage 5: Register Write
  - Most instructions write the result of some computation into a register (arithmetic, logical, shifts, loads, slt)
  - How about stores, branches, and jumps?
    - Nothing happens in this stage

STAGES OF EXECUTION ON DATAPATH: SINGLE-CYCLE

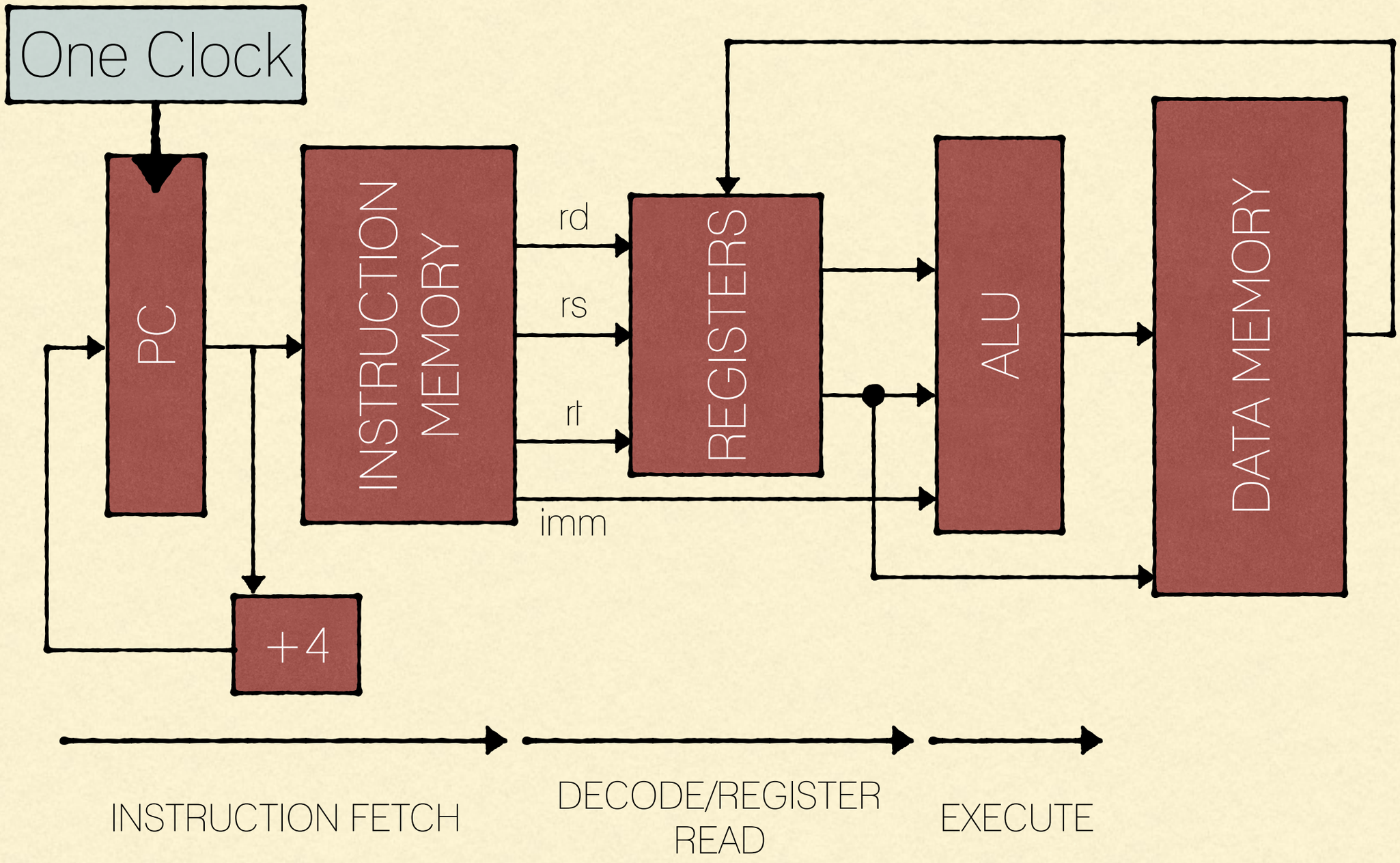


INSTRUCTION FETCH

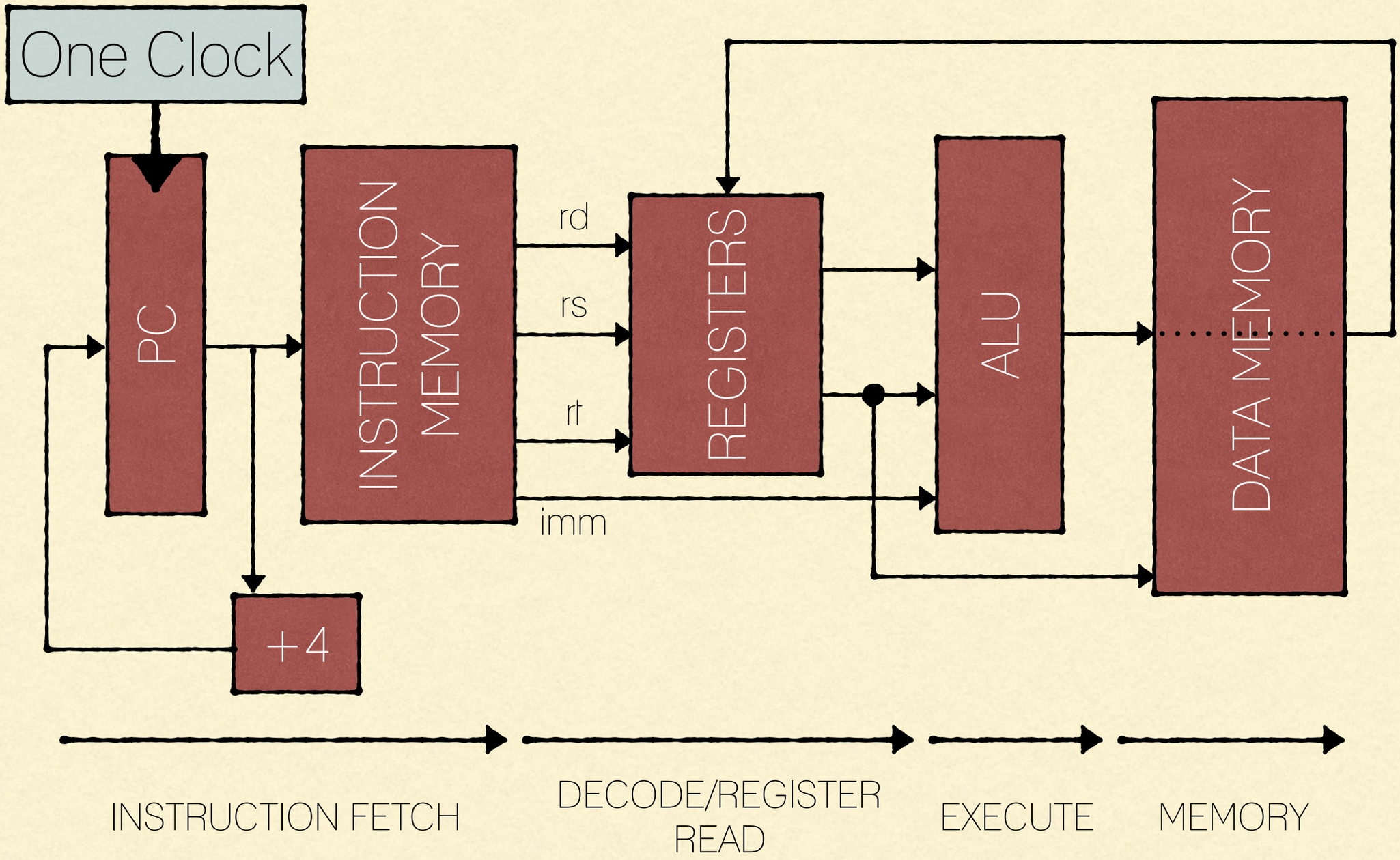
STAGES OF EXECUTION ON DATAPATH: SINGLE-CYCLE



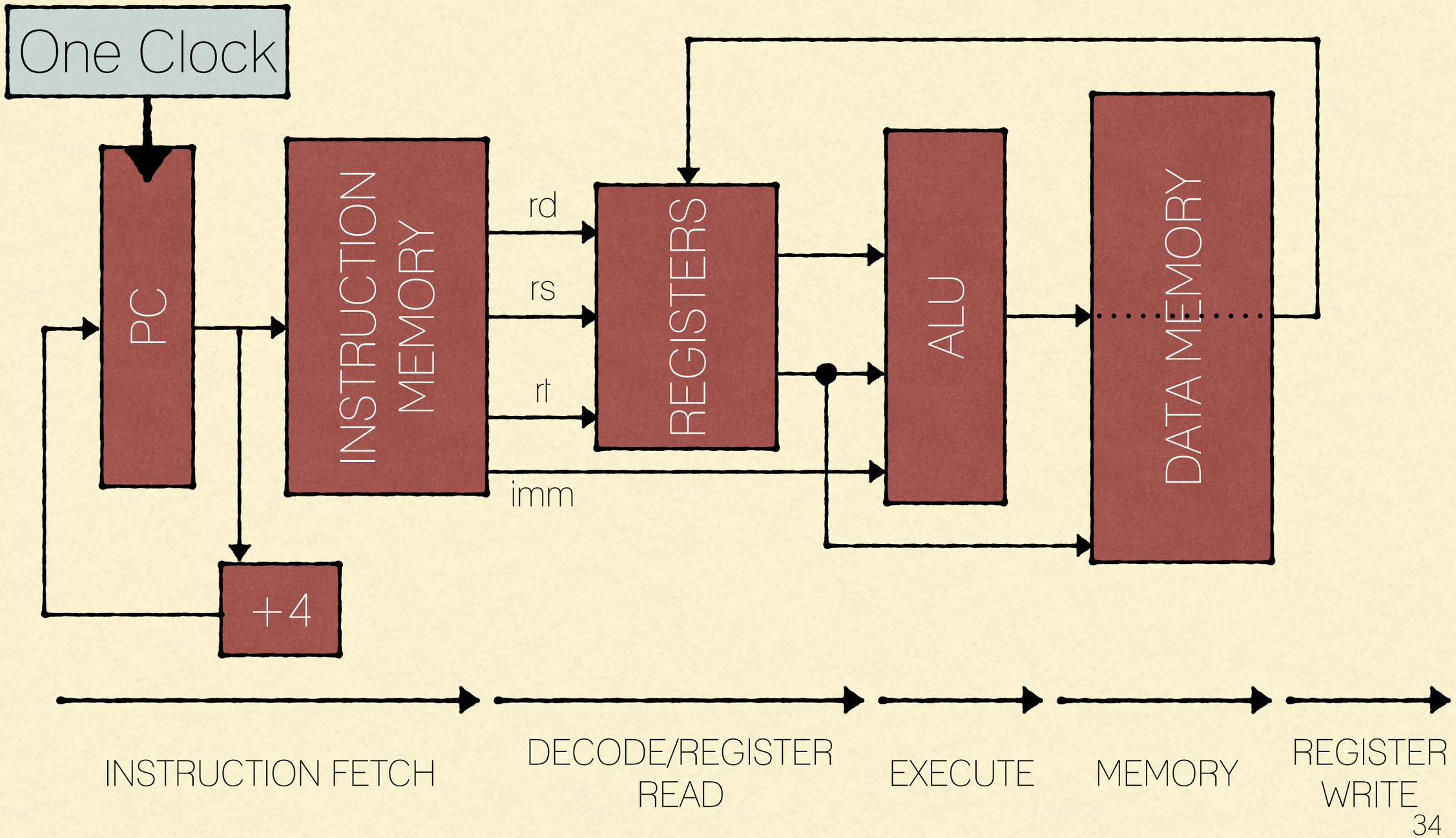
STAGES OF EXECUTION ON DATAPATH: SINGLE-CYCLE



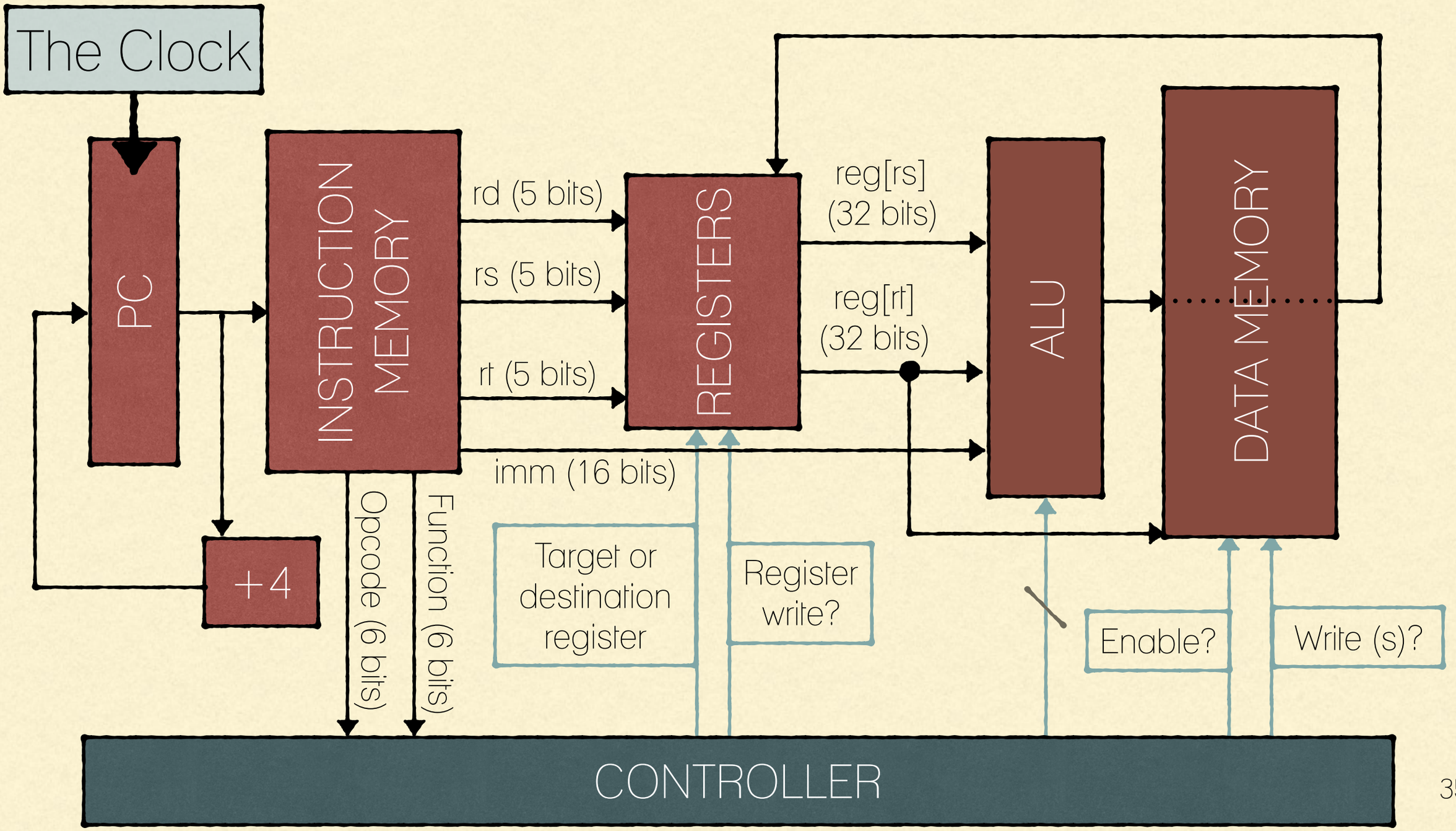
STAGES OF EXECUTION ON DATAPATH: SINGLE-CYCLE

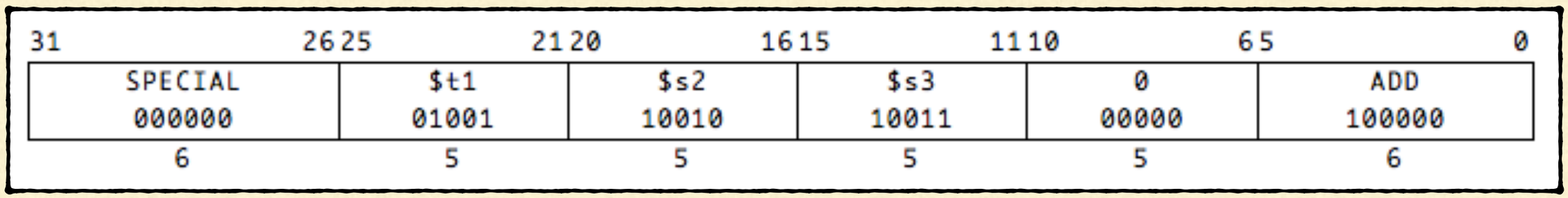


STAGES OF EXECUTION ON DATAPATH: SINGLE-CYCLE



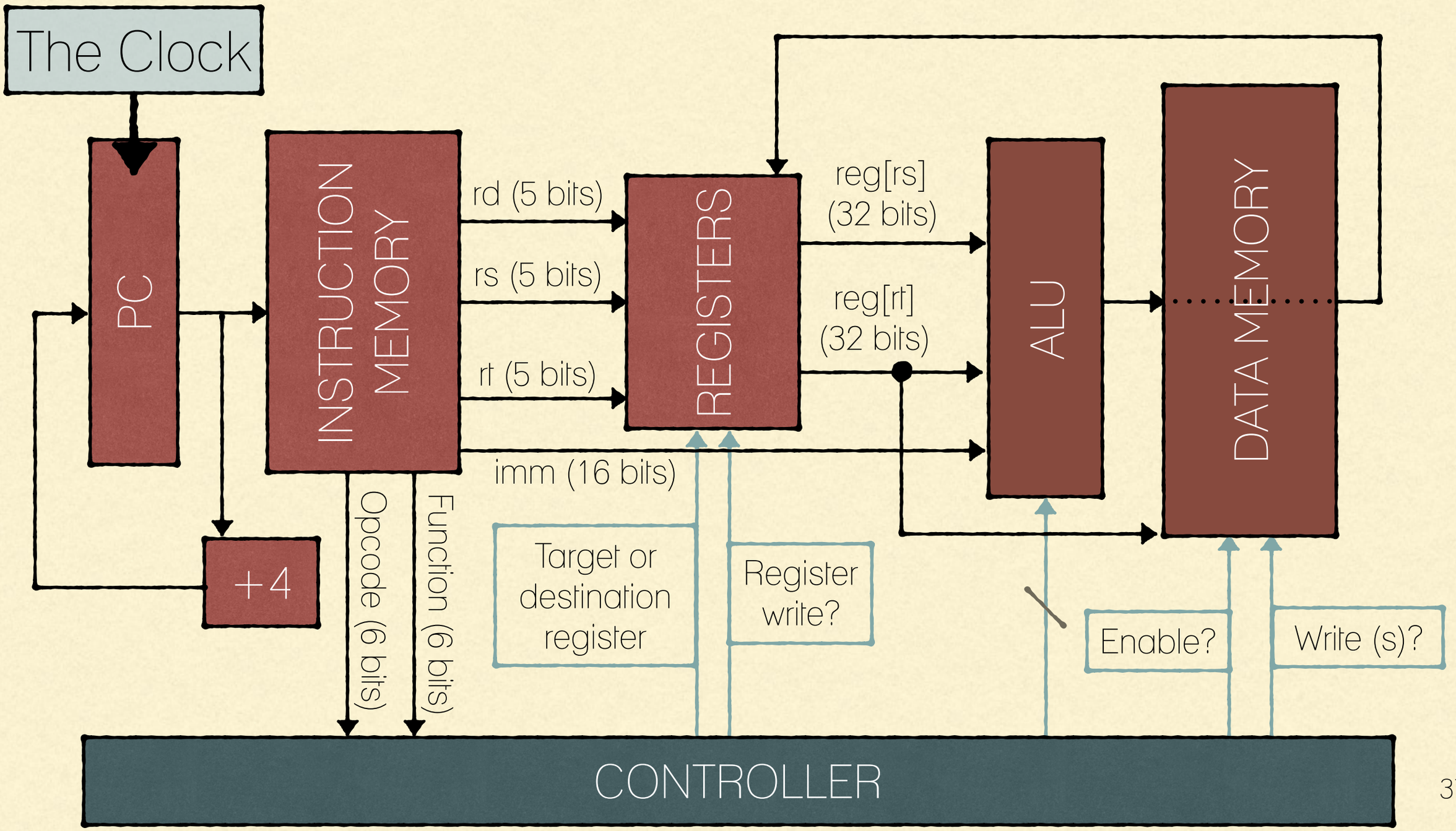
STAGES OF EXECUTION ON DATAPATH: SINGLE-CYCLE

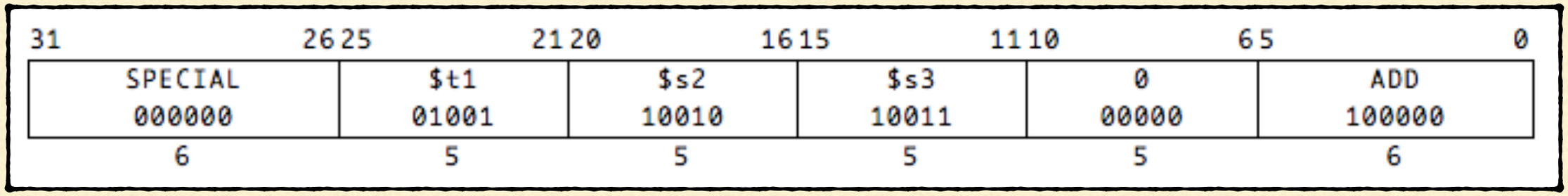


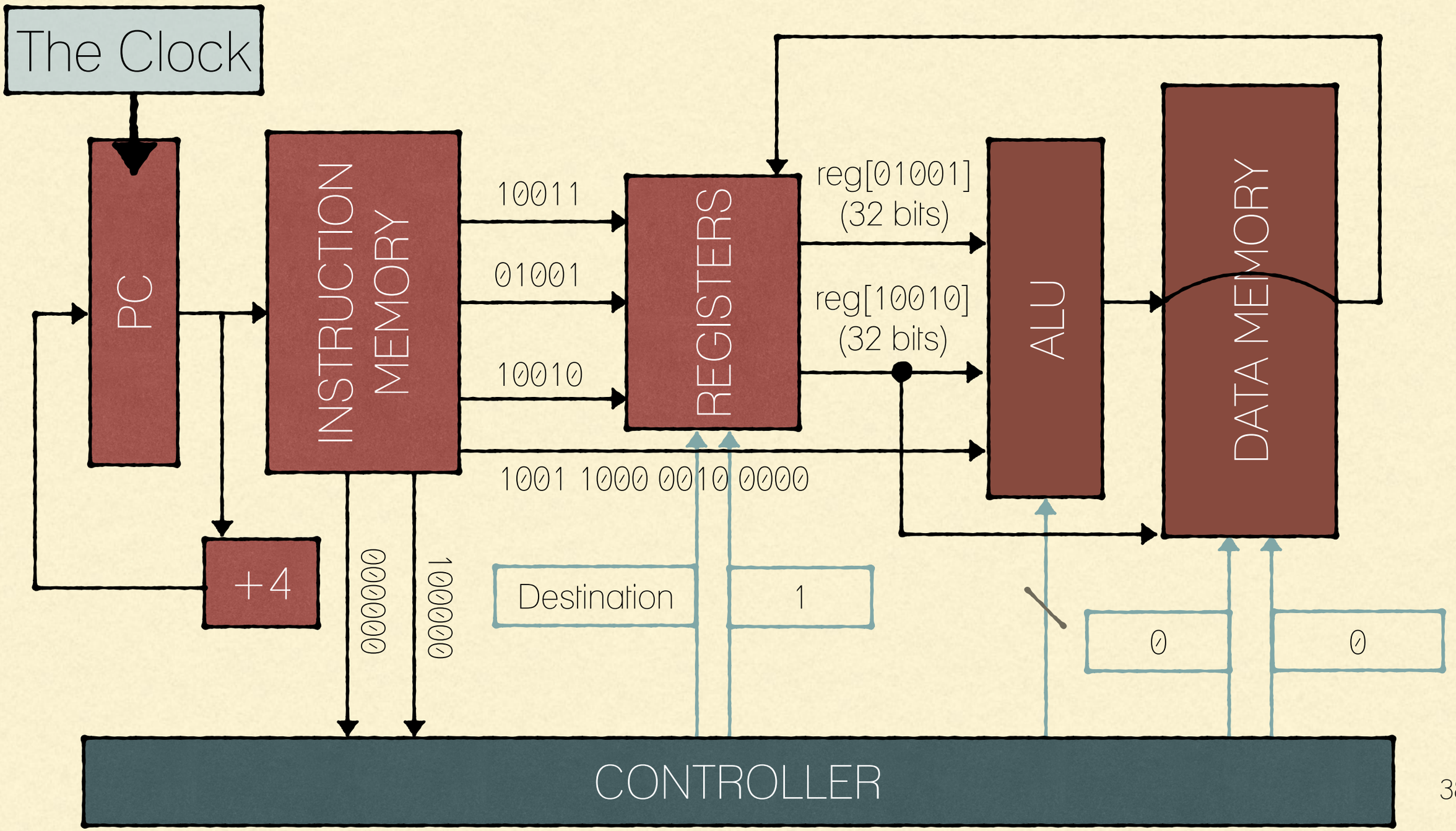


- Add $s3, $t1, $s2
  - Stage 1: fetch this instruction, increment PC
  - Stage 2: decode to determine that it's add, then read registers $s1 and $s2
  - Stage 3: add the two values retrieved in Stage 2
  - Stage 4: idle (nothing to write to memory)
  - Stage 5: write result of Stage 3 into register $s3

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	$t1 01001	$s2 10010	$s3 10011	0 00000	ADD 100000
6	5	5	5	5	6



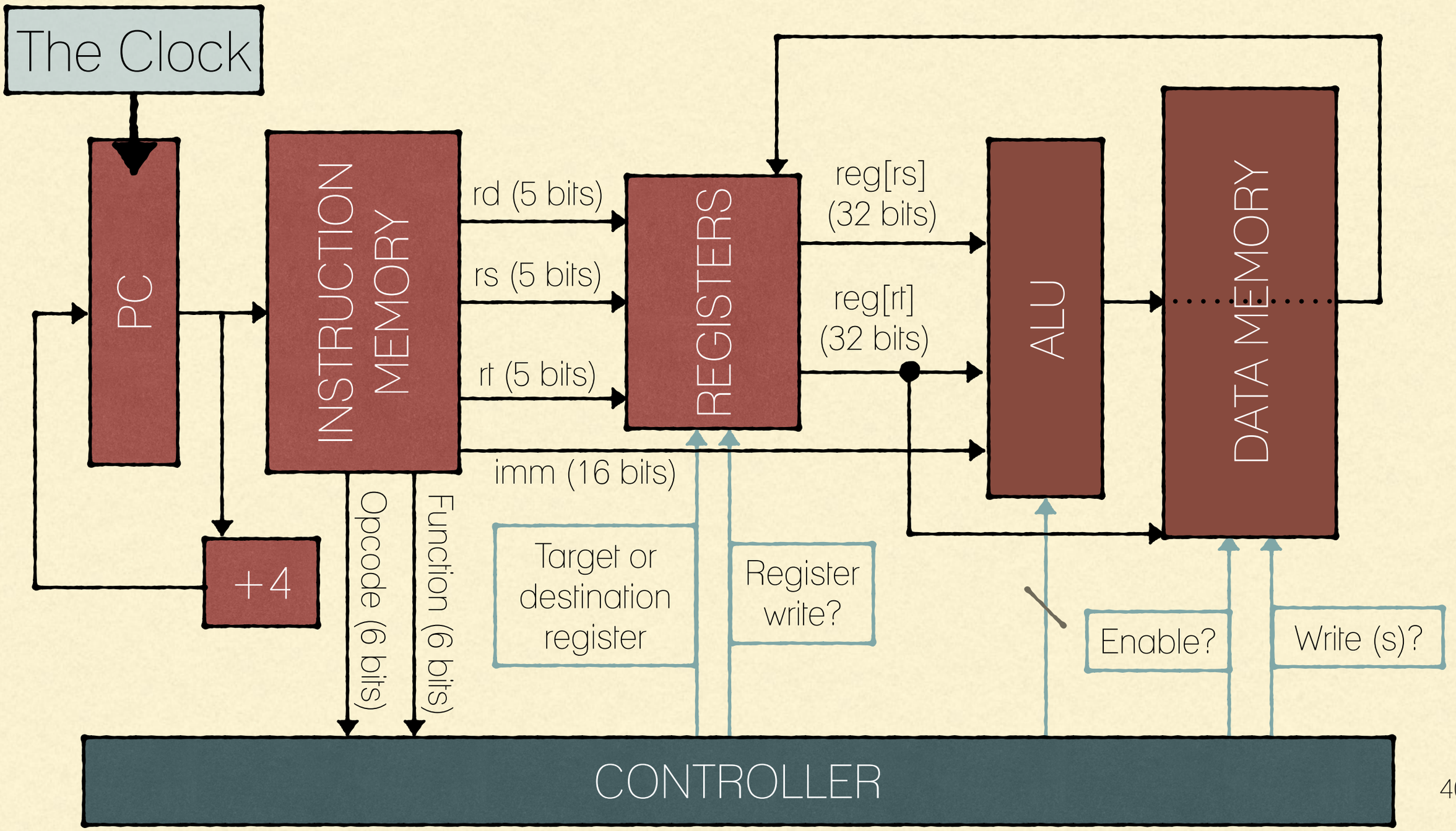

31 26 25 21 20 16 15 11 10 6 5 0
SPECIAL
000000
$t1
01001
$s2
10010
$s3
10011
0
00000
ADD
100000
6 5 5 5 5 6

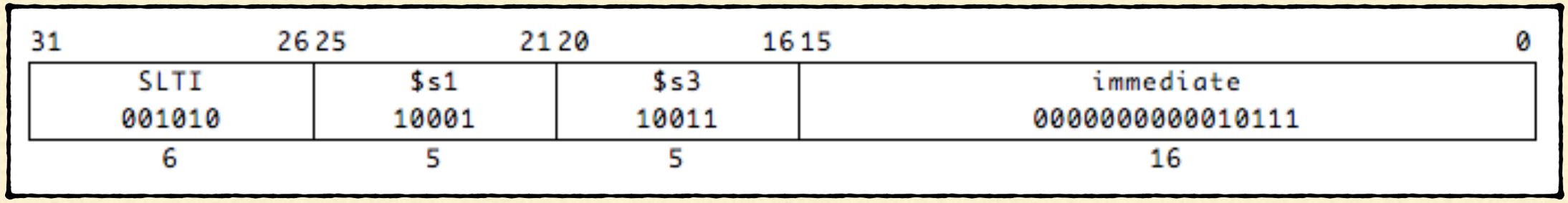

The Clock
PC
INSTRUCTION MEMORY
10011
01001
10010
REGISTERS
reg[01001] (32 bits)
reg[10010] (32 bits)
ALU
DATA MEMORY
1001 1000 0010 0000
+4
000000
100000
Destination
1
0
0
CONTROLLER

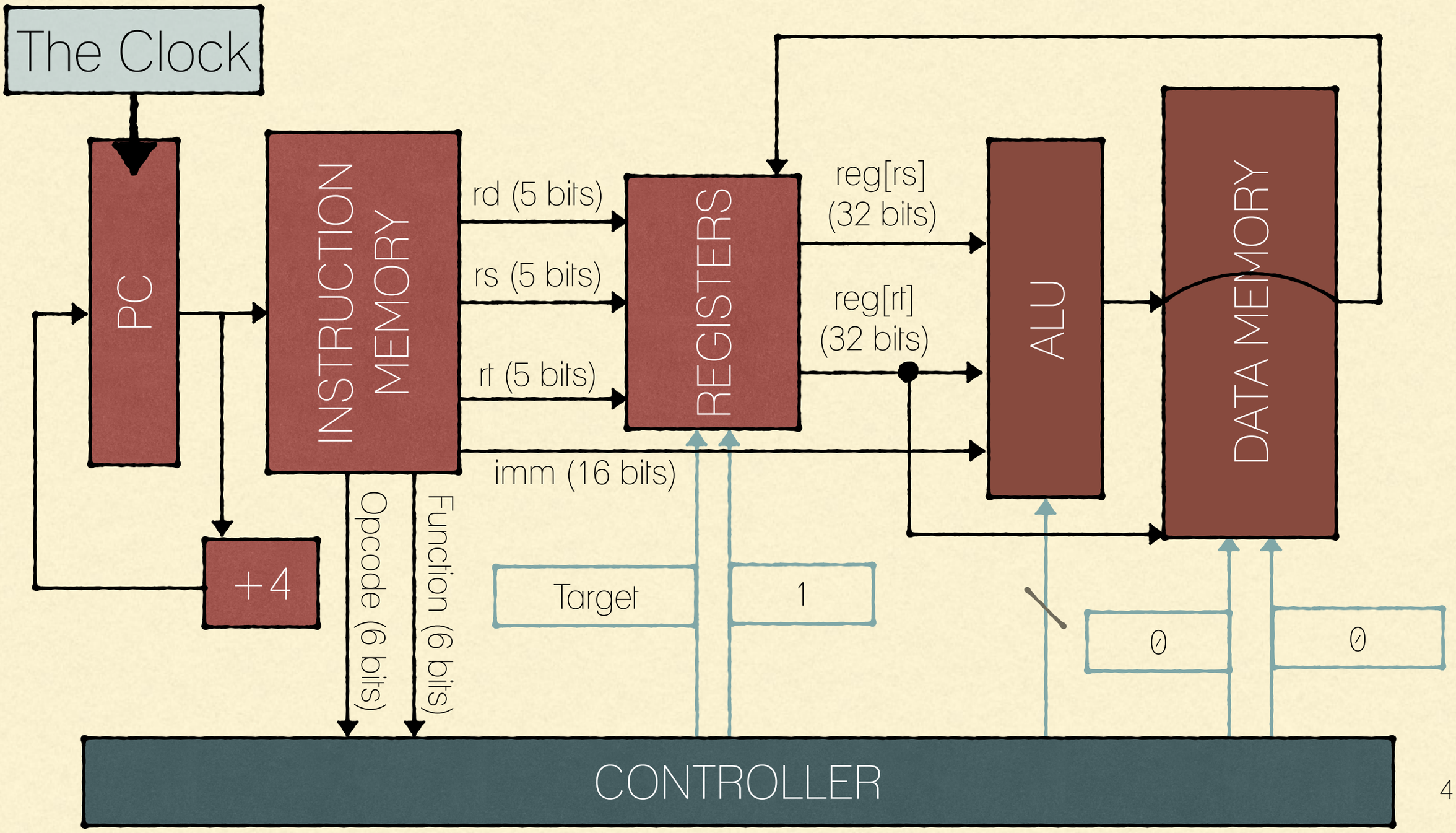
DATAPATH WALKTHROUGHS

- slti $s3, $s1, 0x17
  - Stage 1: fetch this instruction, increment PC
  - Stage 2: decode to determine that it's slti, then read register $s1
  - Stage 3: compare the value retrieved in Stage 2 with the integer 0x17
  - Stage 4: idle (nothing to write to memory)
  - Stage 5: write result of Stage 3 into register $s3

31 26	25 21	20 16	15 0
SLTI 001010	$s1 10001	$s3 10011	immediate 0000000000010111
6	5	5	16



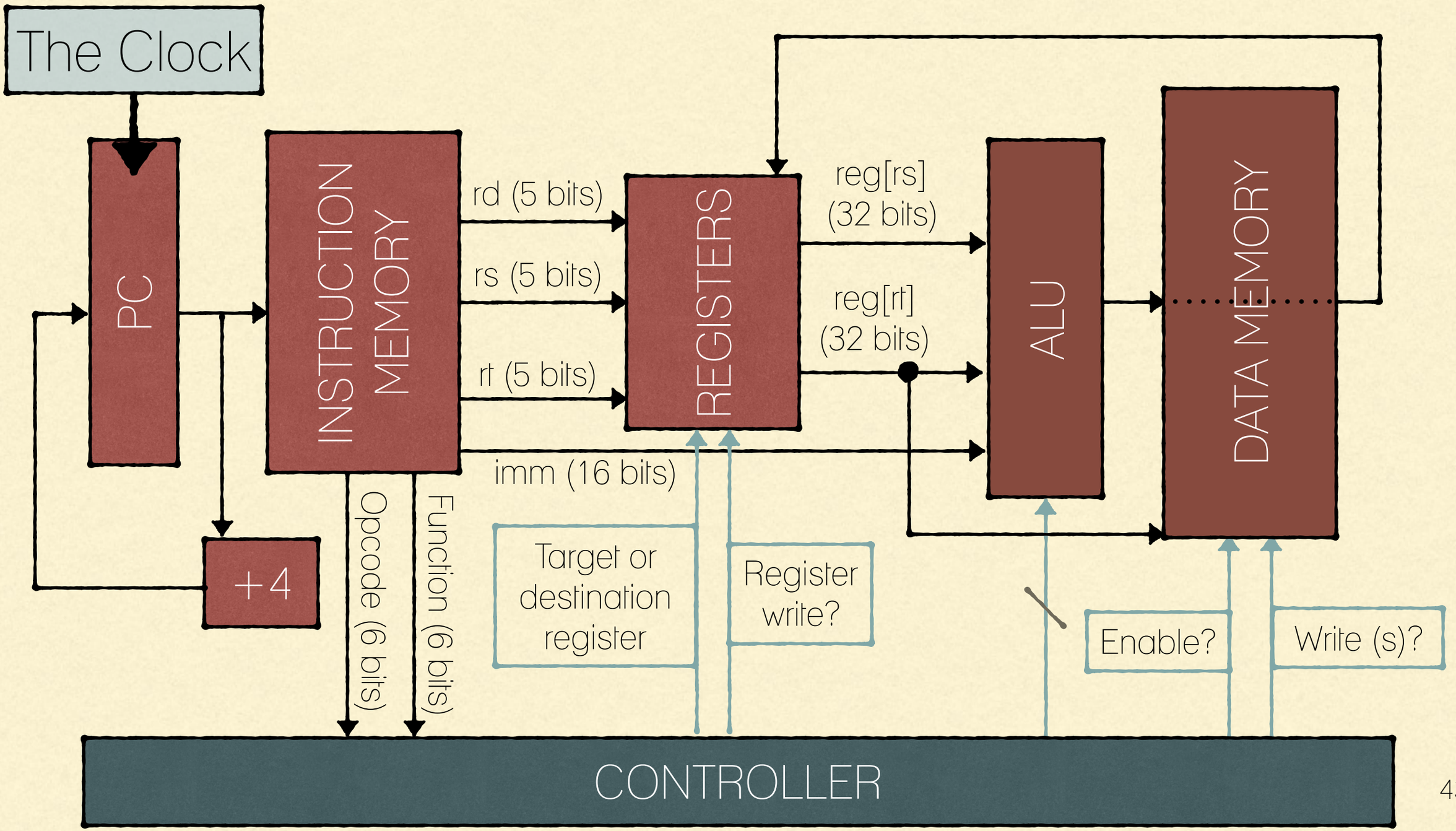

31 26 25 21 20 16 15 0
SLTI
001010
$s1
10001
$s3
10011
immediate
0000000000010111
6
5
5
16

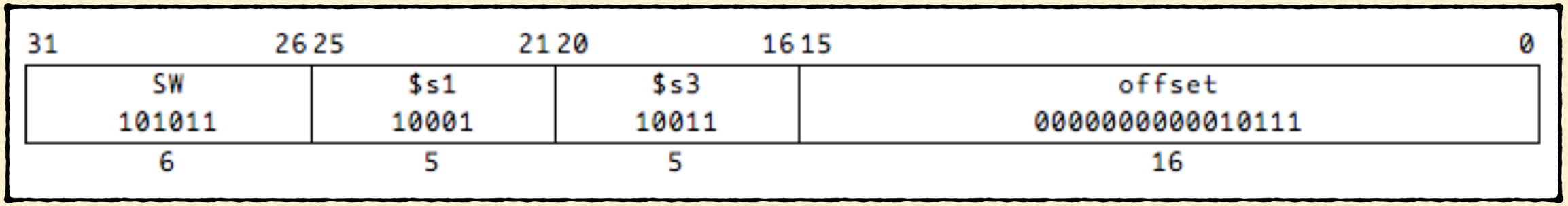

The Clock
PC
+4
INSTRUCTION MEMORY
Opcode (6 bits)
Function (6 bits)
rd (5 bits)
rs (5 bits)
rt (5 bits)
imm (16 bits)
REGISTERS
Target
1
reg[rs] (32 bits)
reg[rt] (32 bits)
ALU
DATA MEMORY
0
0
CONTROLLER

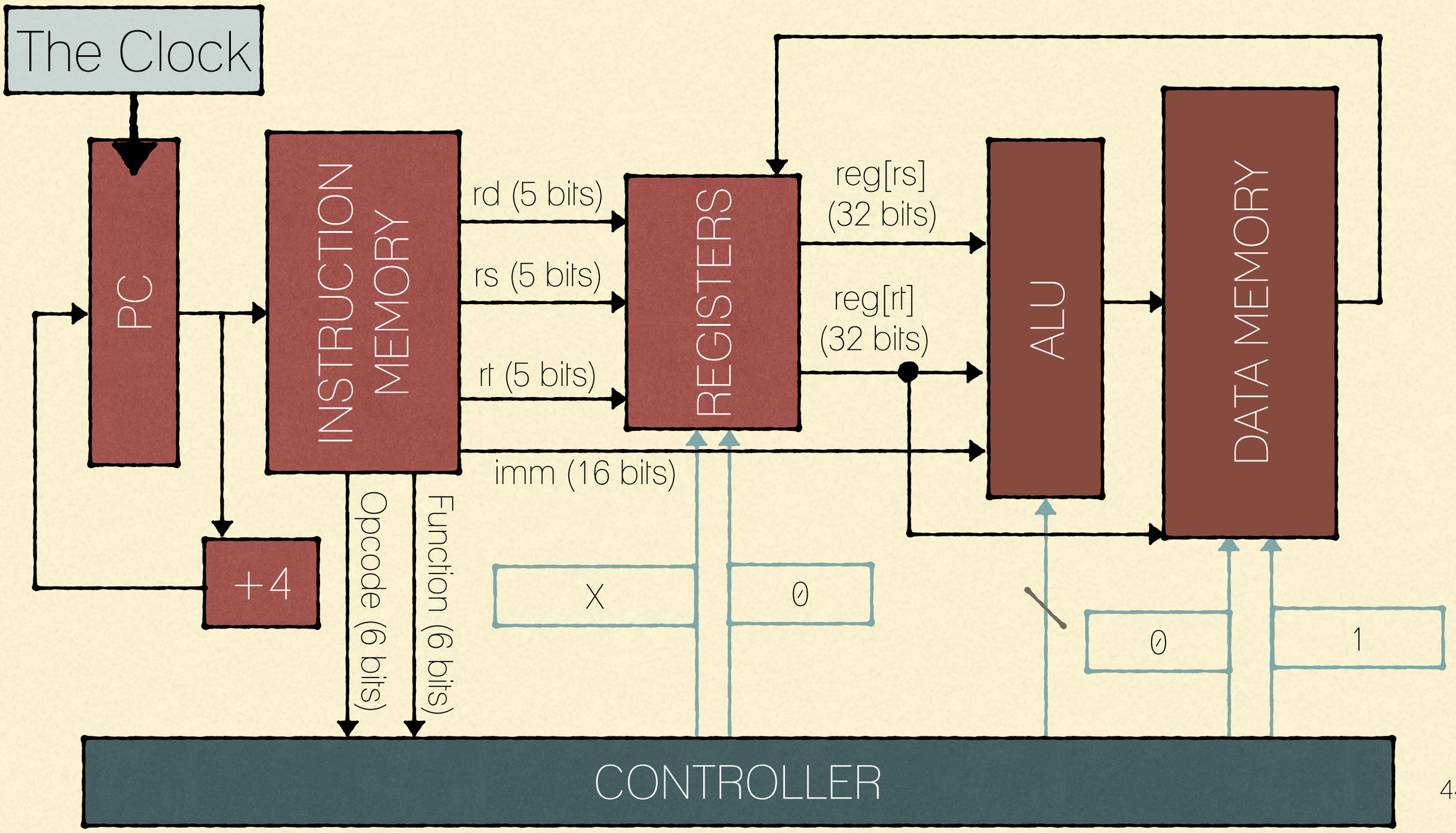
DATAPATH WALKTHROUGHS

- sw $s3, 0x17($s1)
  - Stage 1: fetch this instruction, increment PC
  - Stage 2: decode to determine that it's sw, then read registers $s1 and $s3
  - Stage 3: add 0x17 to value in register $s1 (retrieved in Stage 2)
  - Stage 4: write value in register $s3 (retrieved in Stage 3) into memory address computed in Stage 3
  - Stage 5: idle (nothing to write into a register)

31 26	25 21	20 16	15 0
SW	$s1	$s3	offset
101011	10001	10011	0000000000010111
6	5	5	16



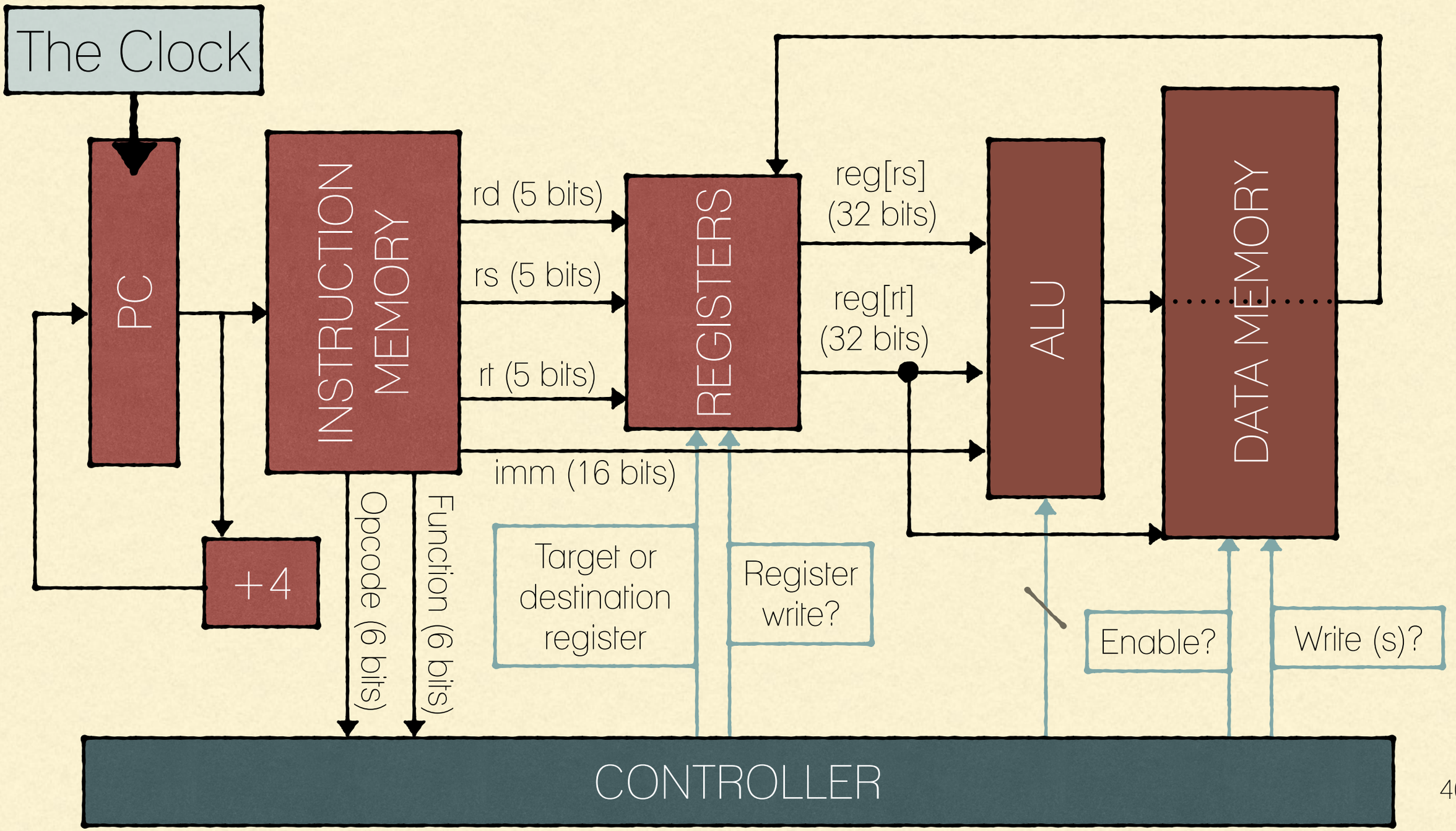

31
26 25
21 20
16 15
0
SW
101011
6
$s1
10001
5
$s3
10011
5
offset
0000000000010111
16


The Clock
PC
INSTRUCTION MEMORY
rd (5 bits)
rs (5 bits)
rt (5 bits)
imm (16 bits)
REGISTERS
reg[rs] (32 bits)
reg[rt] (32 bits)
ALU
DATA MEMORY
+4
Opcode (6 bits)
Function (6 bits)
X
0
0
1
CONTROLLER

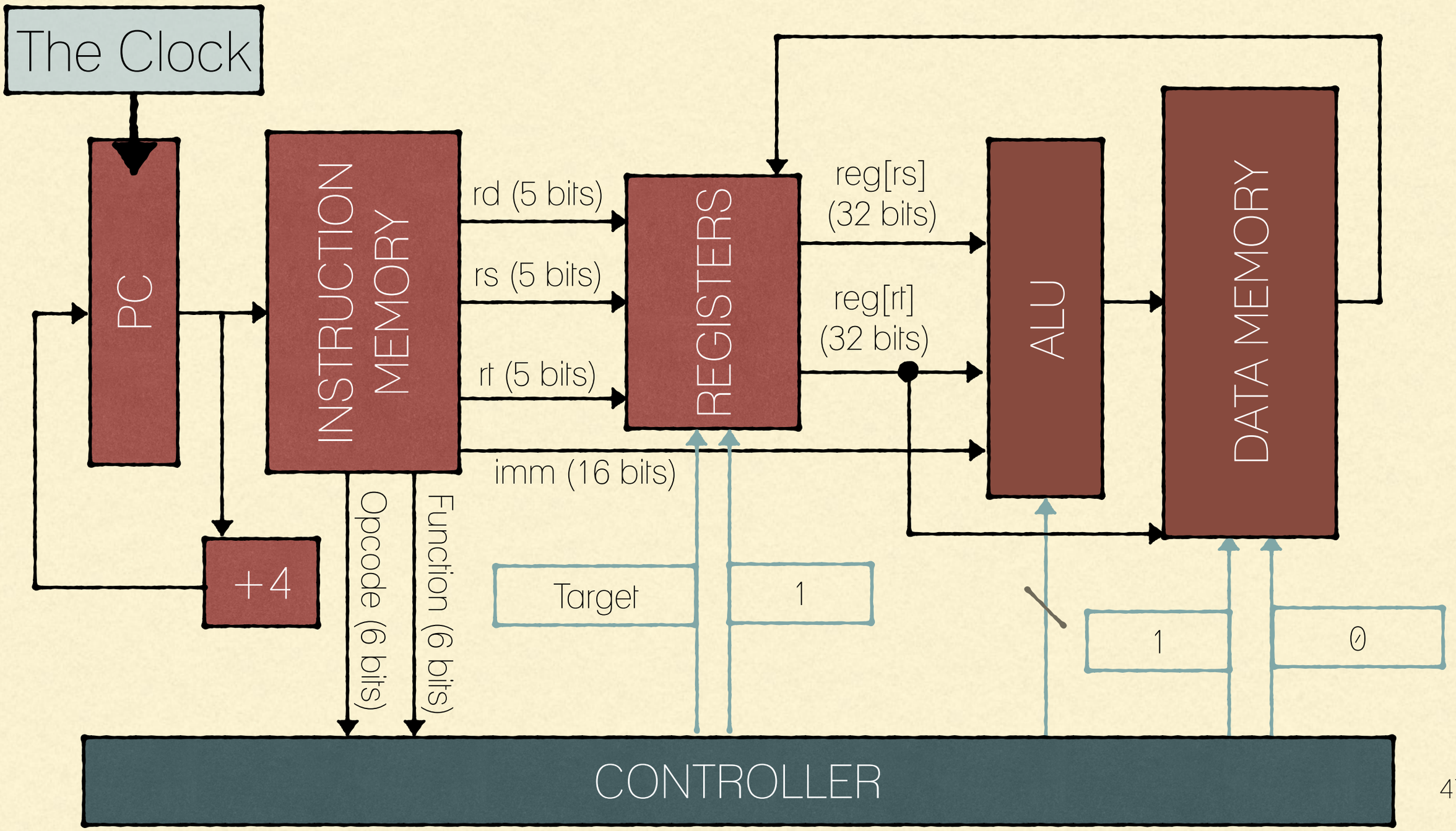
DATAPATH WALKTHROUGHS

- lw $s3, 0x17($s1)
  - Stage 1: fetch this instruction, increment PC
  - Stage 2: decode to determine that it's lw, then read register $s1
  - Stage 3: add 0x17 to value in register $s1 (retrieved in Stage 2)
  - Stage 4: read value from memory address computed in Stage 3
  - Stage 5: write value read in Stage 4 into register $s3

31 26	25 21	20 16	15 0
LW	$s1	$s3	offset
100011	10001	10011	0000000000010111
6	5	5	16



31 26	25 21	20 16	15 0
LW	$s1	$s3	offset
100011	10001	10011	0000000000010111
6	5	5	16



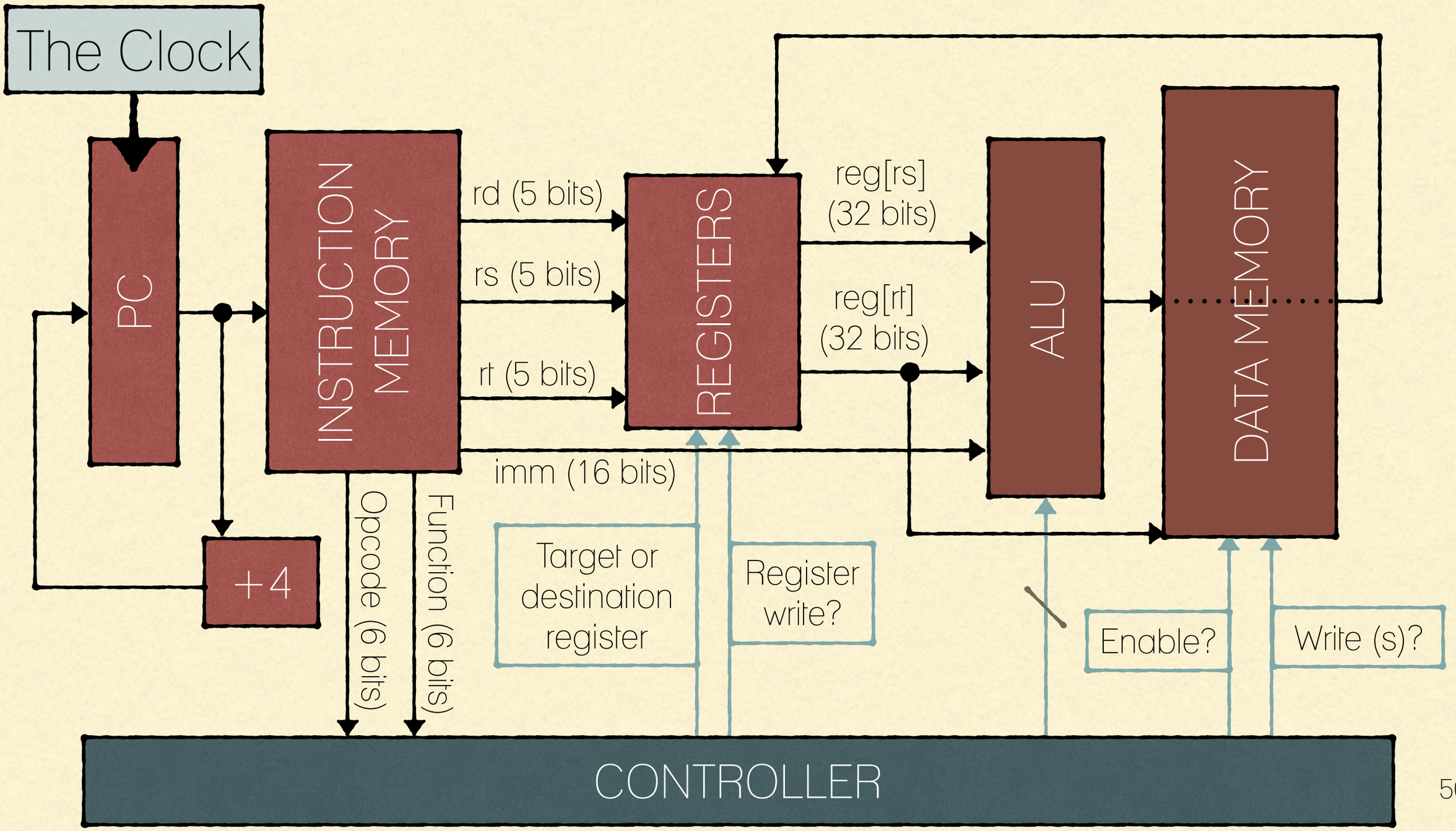
QUESTION

- Which one of these MIPS instructions use the least number of stages?
  - lw
  - beq
  - j
  - jal
  - add

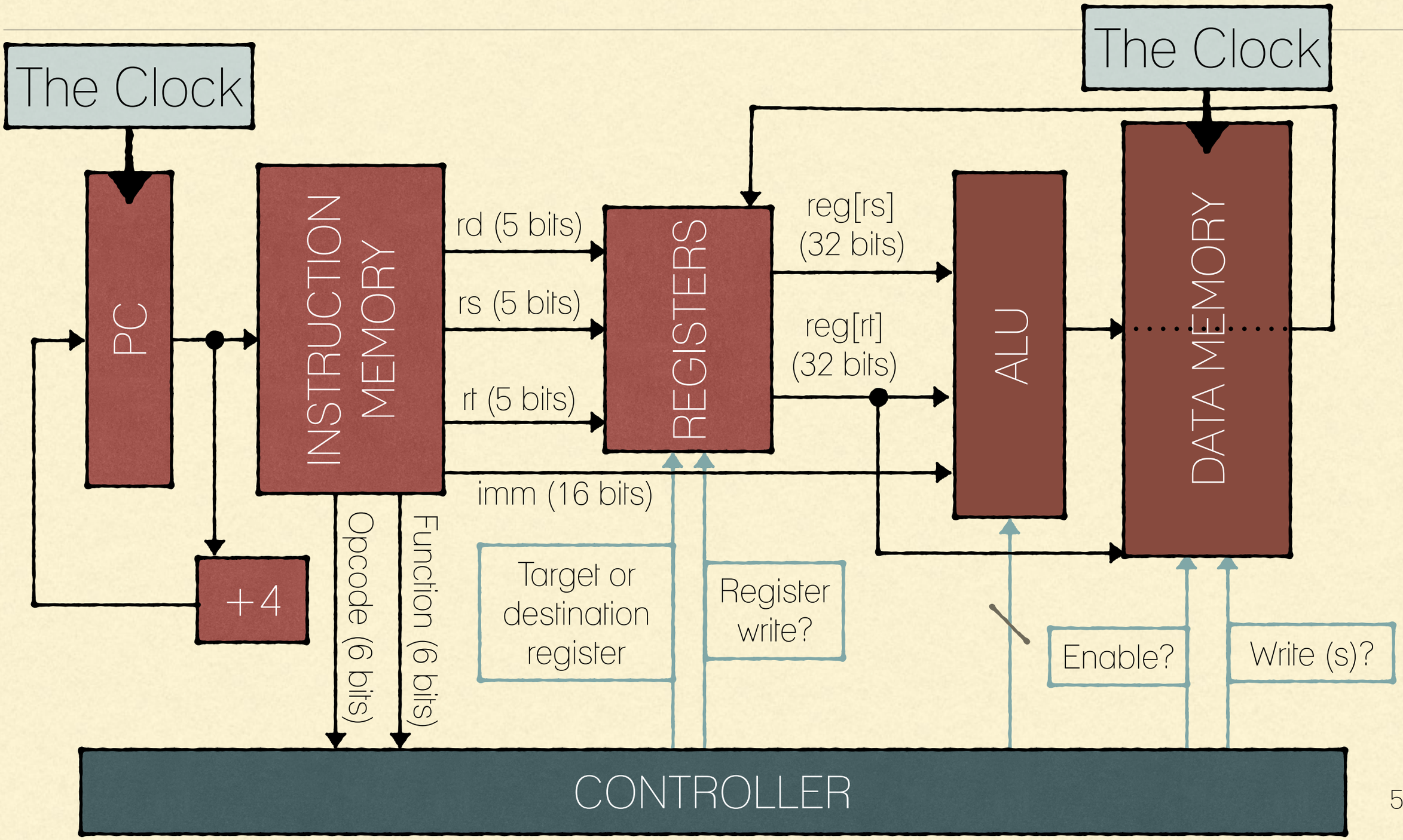
QUESTION

- Which one of these MIPS instructions use the least number of stages?
  - lw
  - beq
    - j (jump)
  - jal
  - add

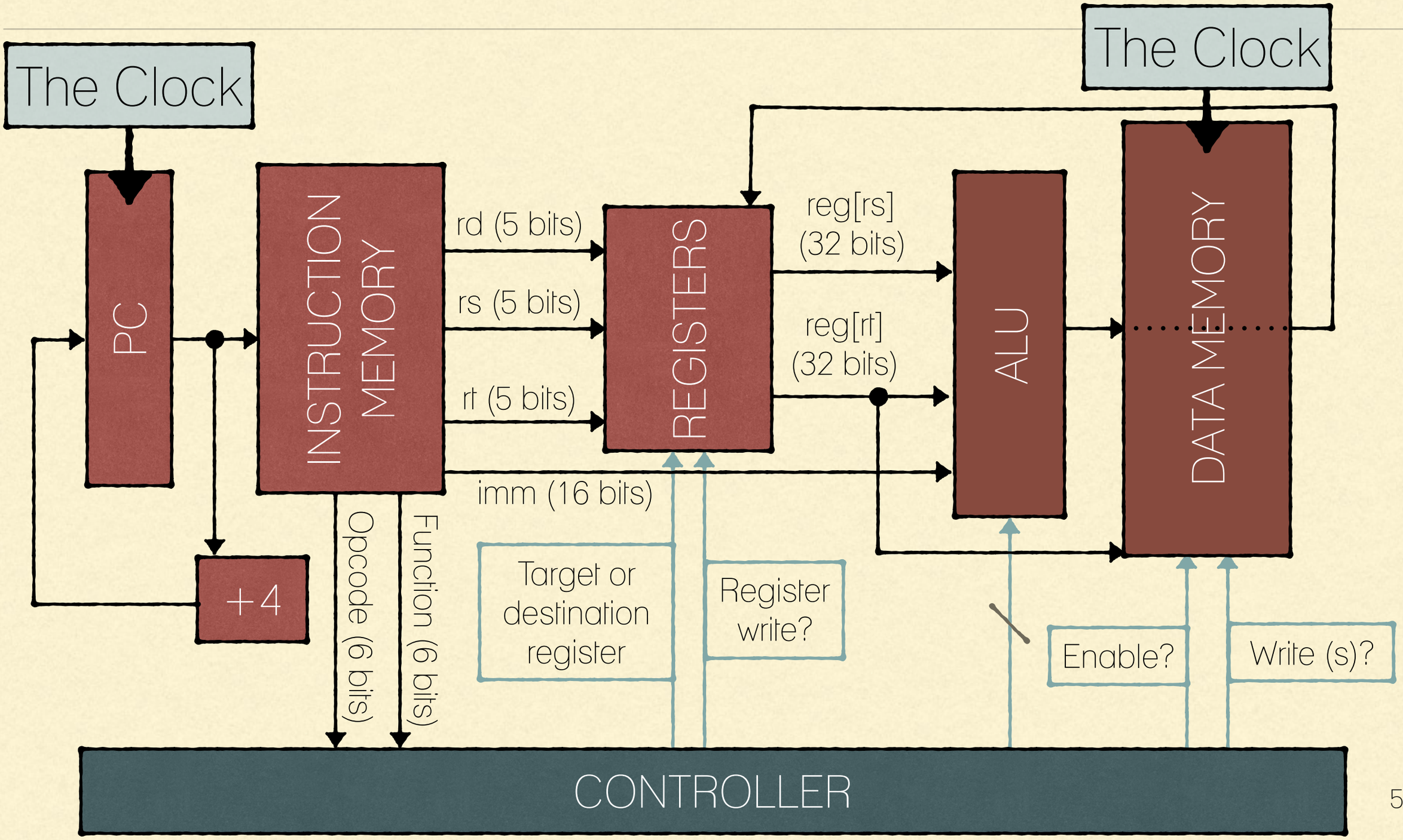
POTENTIAL PROBLEM 1: ACCIDENTAL WRITE TO MAIN MEMORY



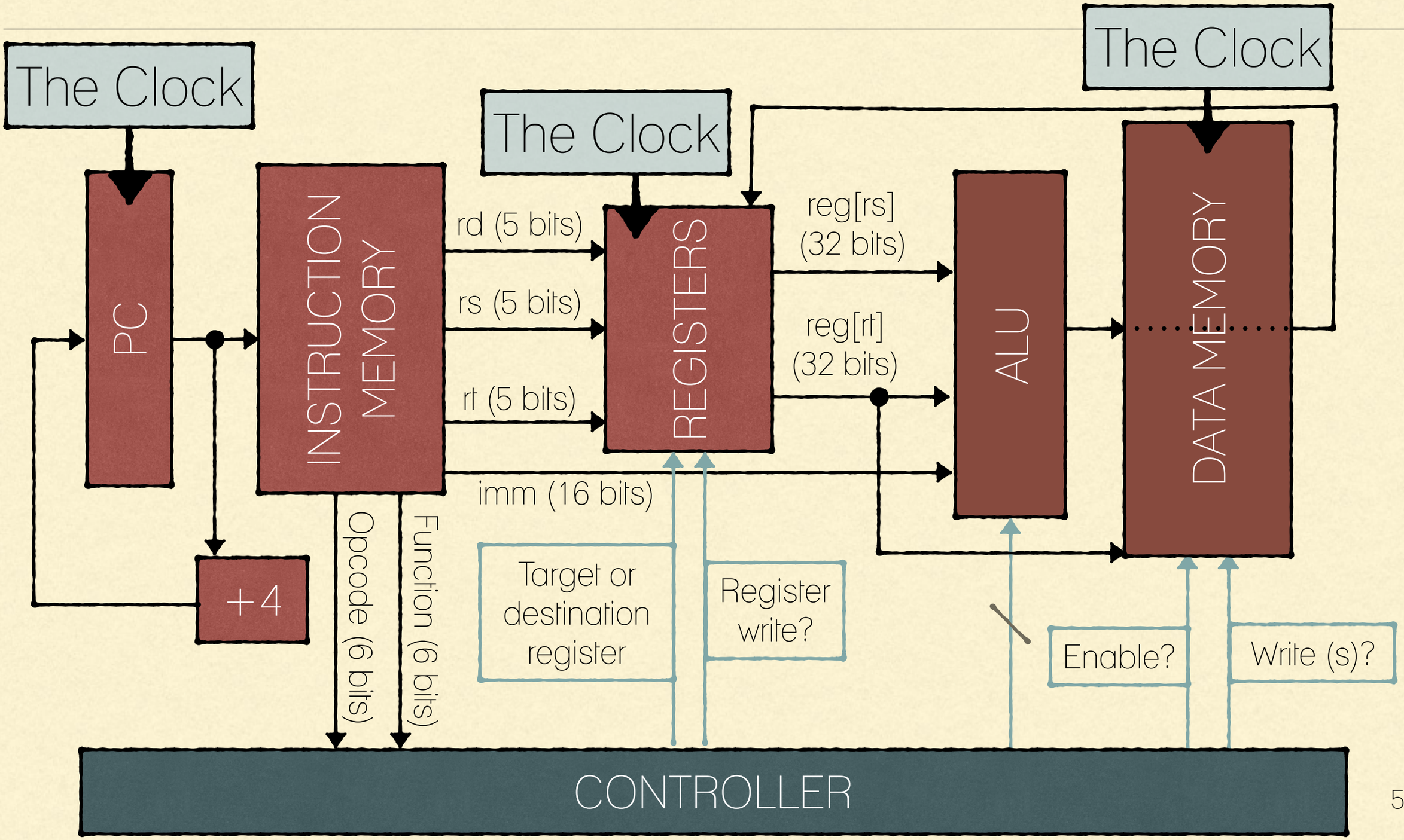
CHANGE 1: MEMORY BYTES WILL BE SYNCHRONOUS (CONNECTED TO THE SAME CLOCK)



POTENTIAL PROBLEM 2: ACCIDENTAL WRITE TO REGISTERS



CHANGE 2: REGISTERS IN THE REGISTER FILE ARE ALSO CLOCKED



THE MIPS-LITE SUBSET

- ADD & SUB
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD WORD and STORE WORD
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16

REGISTER TRANSFER LEVEL (RTL)

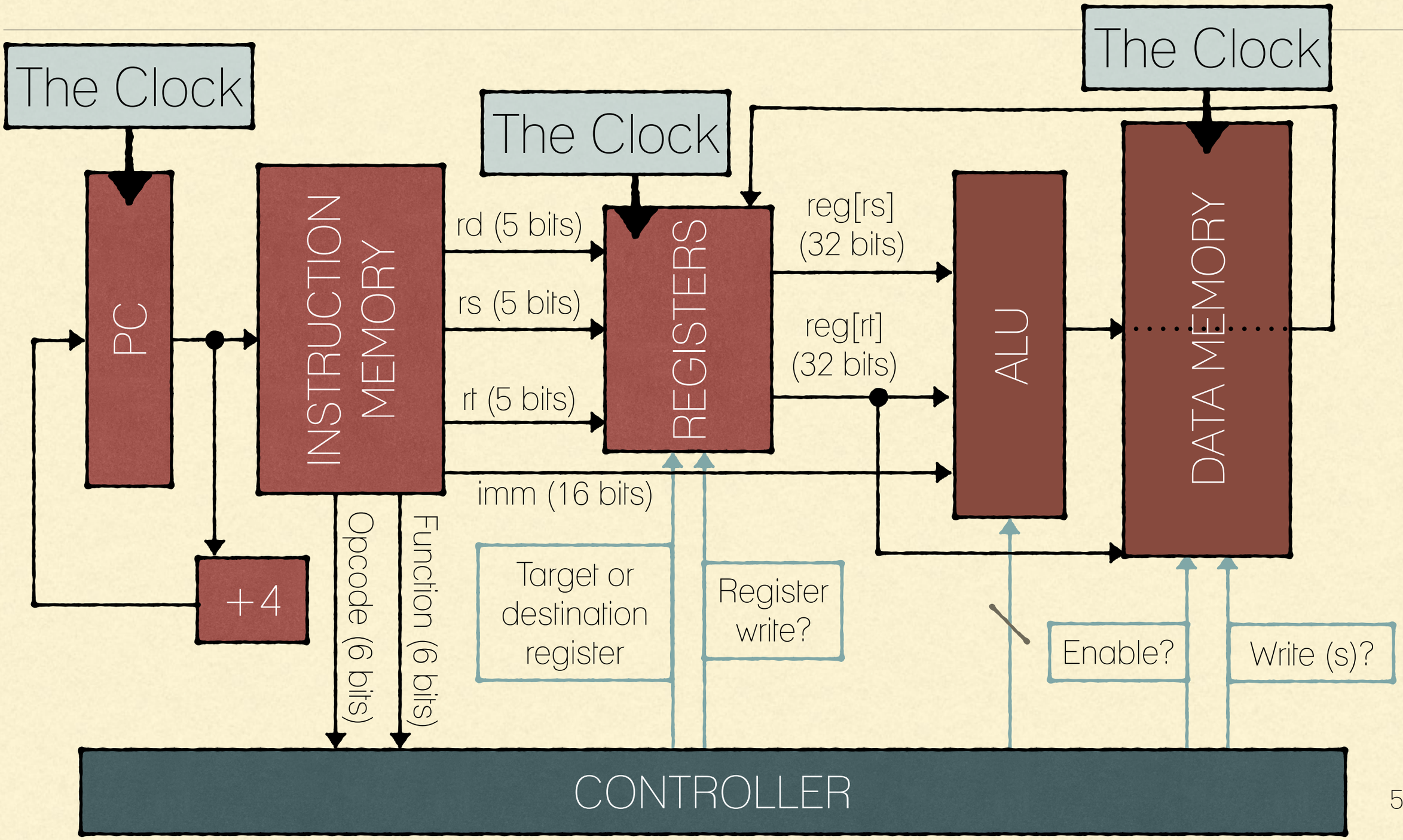
- Also called Register Transfer Language
- RTL gives the meaning of the instruction

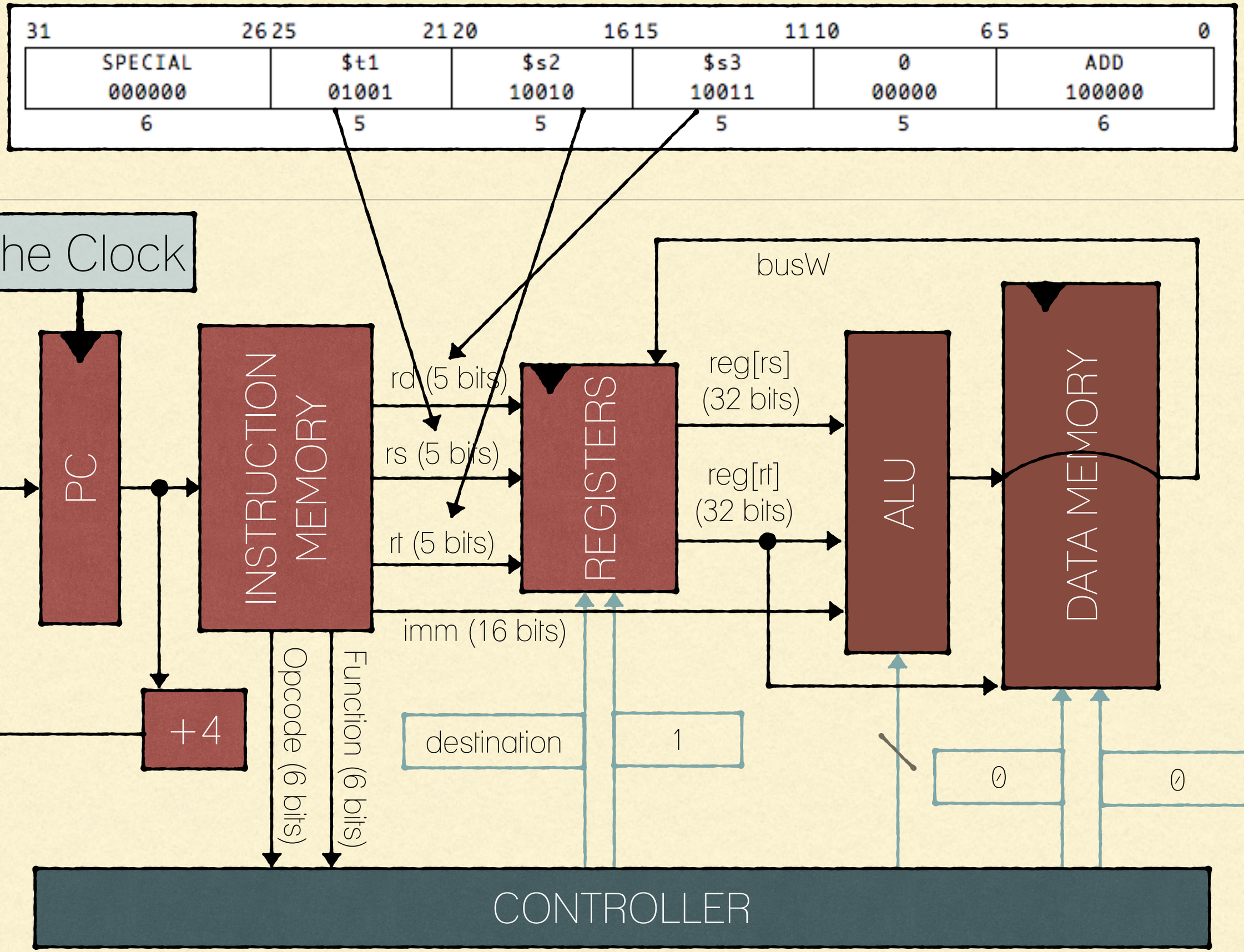
Instruction	Register Transfers
ADD	R[rd] ← R[rs] + R[rt]; PC ← PC + 4
SUB	R[rd] ← R[rs] - R[rt]; PC ← PC + 4
ORI	R[rt] ← R[rs] | zero_ext(Imm16); PC ← PC + 4
LOAD	R[rt] ← MEM[R[rs] + sign_ext(Imm16)]; PC ← PC + 4
STORE	MEM[R[rs] + sign_ext(Imm16)] ← R[rt]; PC ← PC + 4
BEQ	if (R[rs] == R[rt]) PC ← PC + 4 + {sign_ext(Imm16), 2'b00} else: PC ← PC + 4

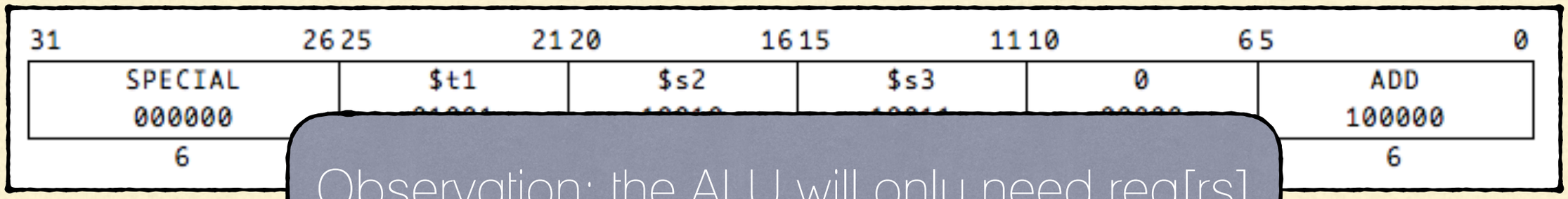
ADD & SUBTRACT

- R[rd] ← R[rs] + R[rt]; PC ← PC + 4

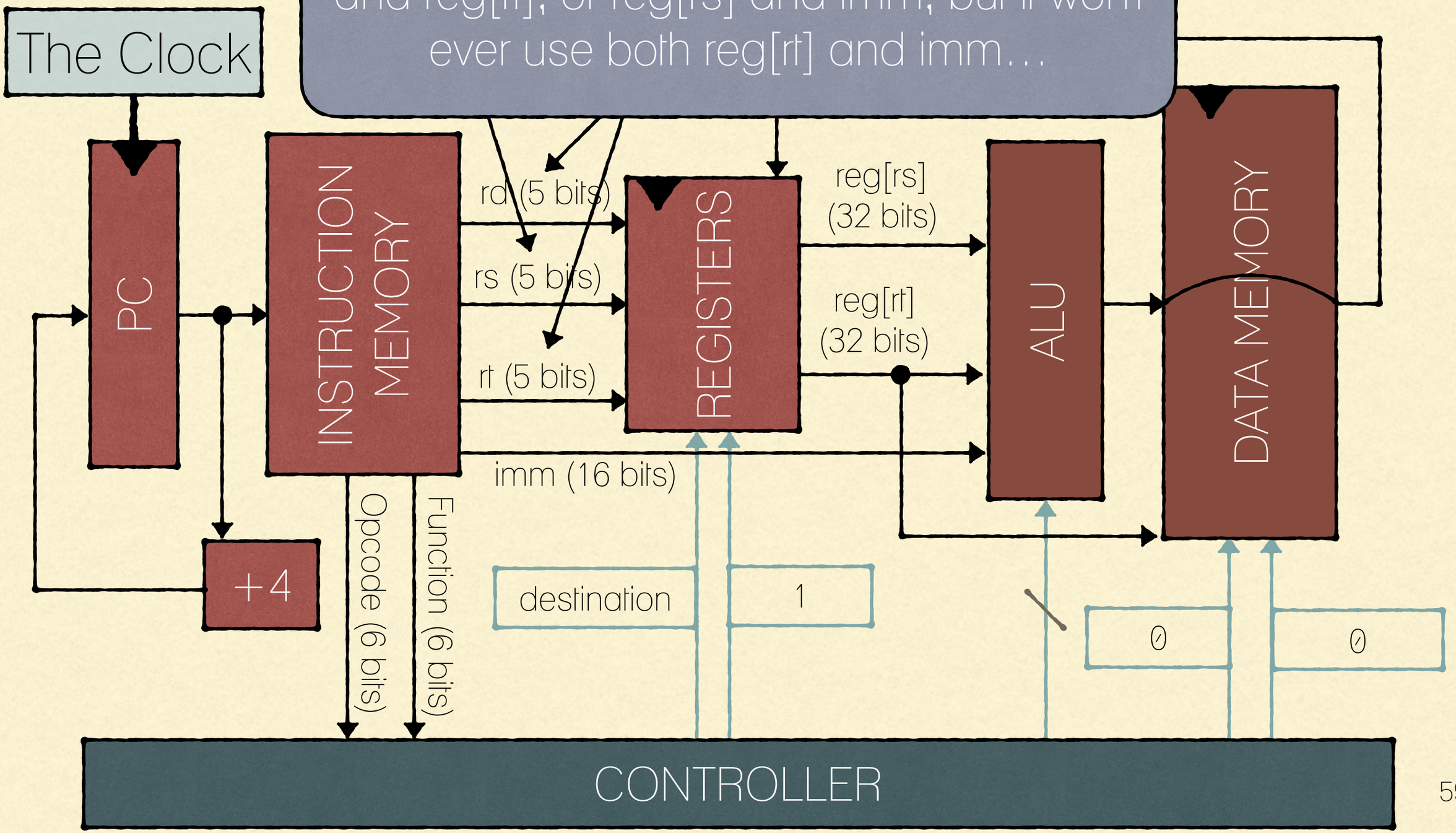
31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	$t1 01001	$s2 10010	$s3 10011	0 00000	ADD 100000
6	5	5	5	5	6

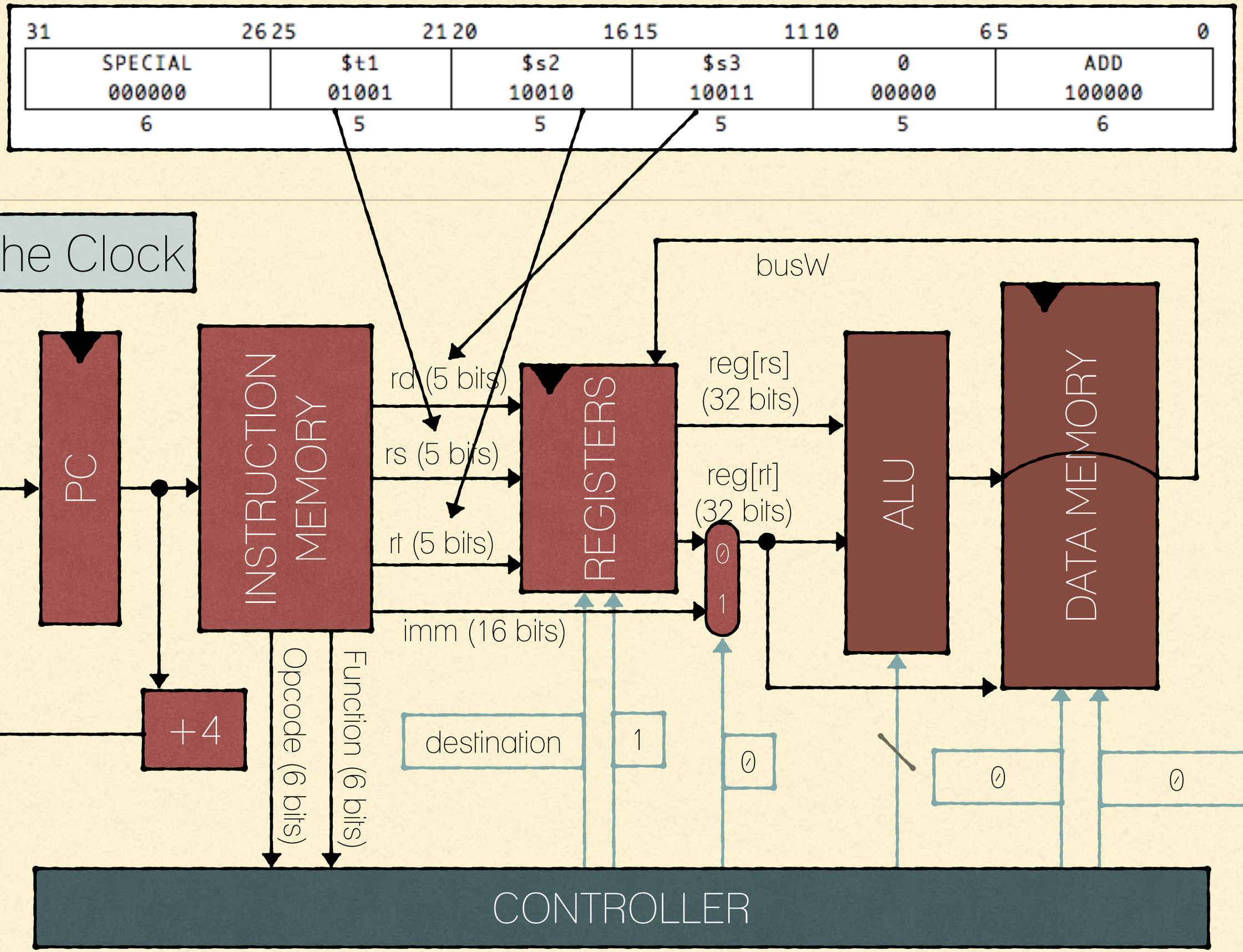



31 26 25 21 20 16 15 11 10 6 5 0
SPECIAL 000000
$t1 01001
$s2 10010
$s3 10011
0 00000
ADD 100000
6 5 5 5 5 6
The Clock
PC
INSTRUCTION MEMORY
rd (5 bits)
rs (5 bits)
rt (5 bits)
REGISTERS
busW
reg[rs] (32 bits)
reg[rt] (32 bits)
ALU
DATA MEMORY
imm (16 bits)
Opcode (6 bits)
Function (6 bits)
+4
destination
1
0
0
CONTROLLER



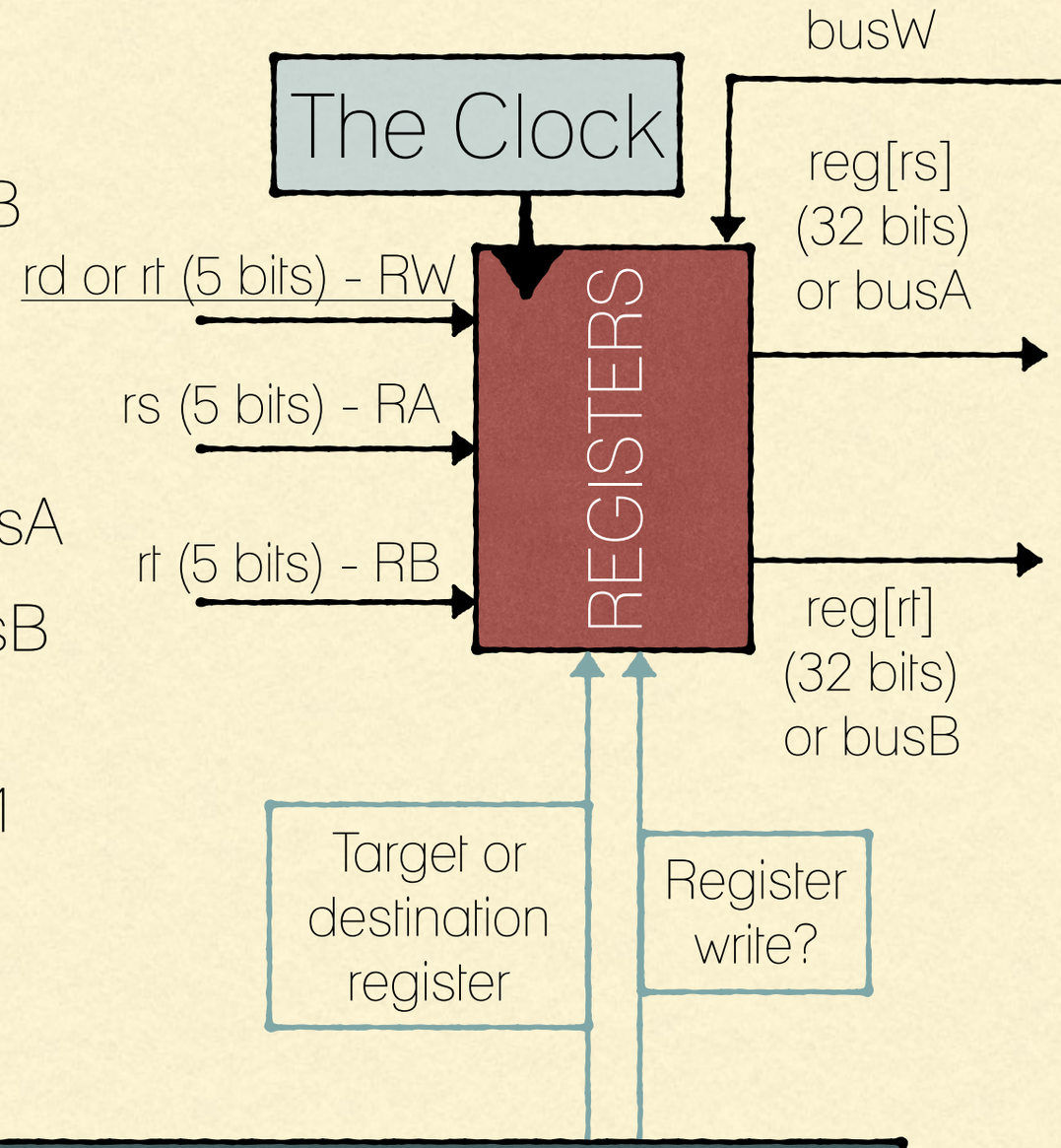
Observation: the ALU will only need reg[rs] and reg[rt], or reg[rs] and imm, but it won't ever use both reg[rt] and imm…

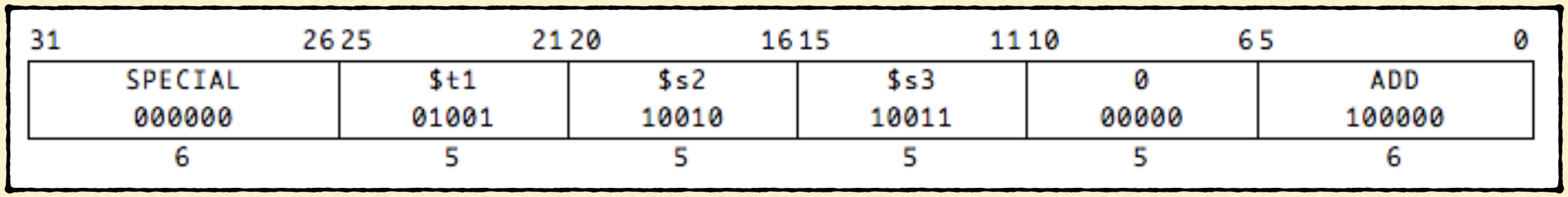


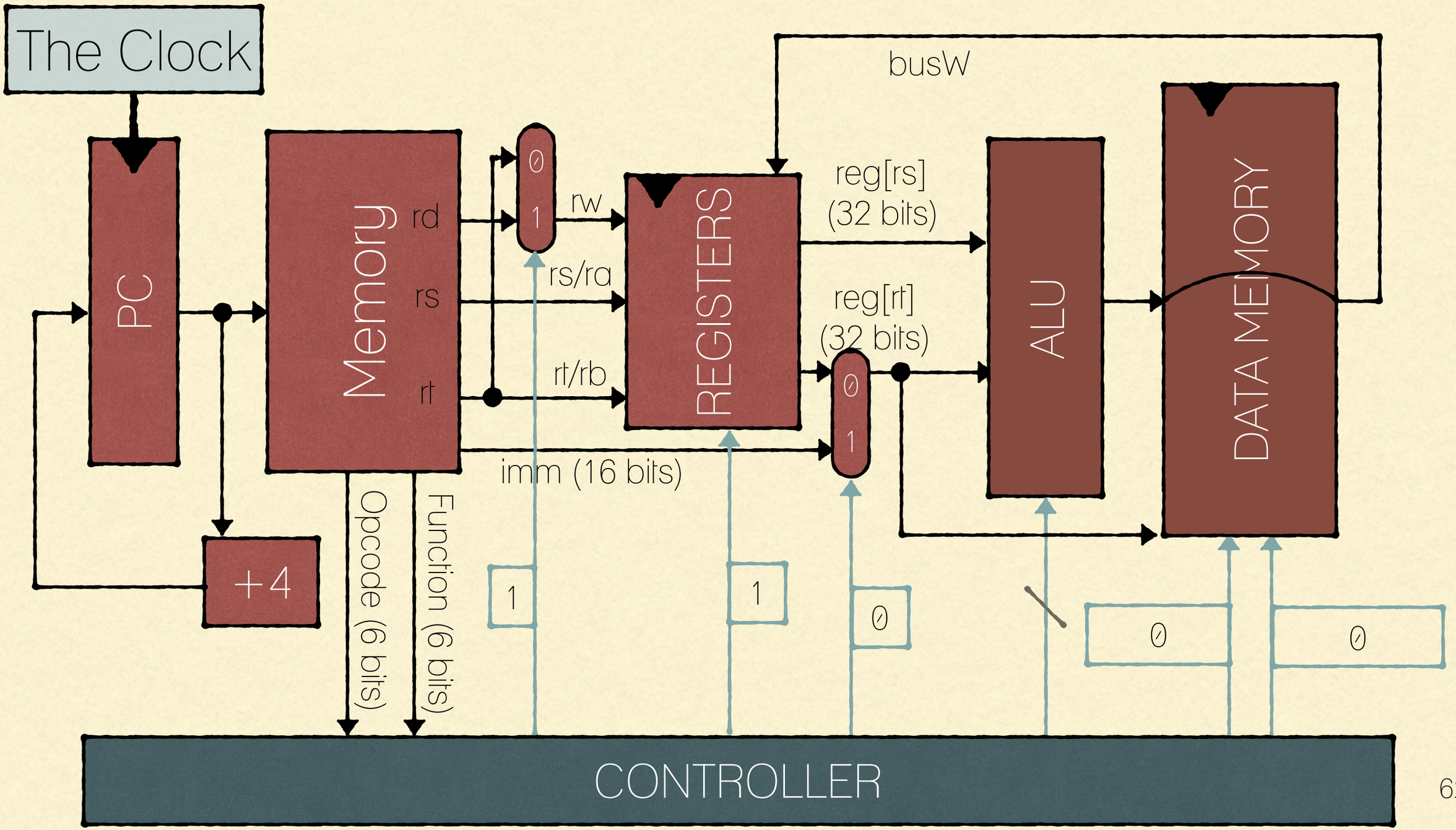

31 26 25 21 20 16 15 11 10 6 5 0
SPECIAL 000000
$t1 01001
$s2 10010
$s3 10011
0 00000
ADD 100000
6 5 5 5 5 6
The Clock
PC
INSTRUCTION MEMORY
rd (5 bits)
rs (5 bits)
rt (5 bits)
imm (16 bits)
REGISTERS
busW
reg[rs] (32 bits)
reg[rt] (32 bits)
0
1
ALU
DATA MEMORY
+4
Opcode (6 bits)
Function (6 bits)
destination
1
0
0
0
CONTROLLER

FEW MORE CHANGES TO THE REGISTER FILE

- Register File consists of 32 registers:
  - Two 32-bit output busses: busA & busB
  - One 32-bit data input bus: busW
- Register is selected by:
  - RA (rs) selects the register to put on busA
  - RB (rt) selects the register to put on busB
  - RW (rd or rt) selects the register to be written via busW when Write Enable is 1
- Clock input is a factor only during write operation



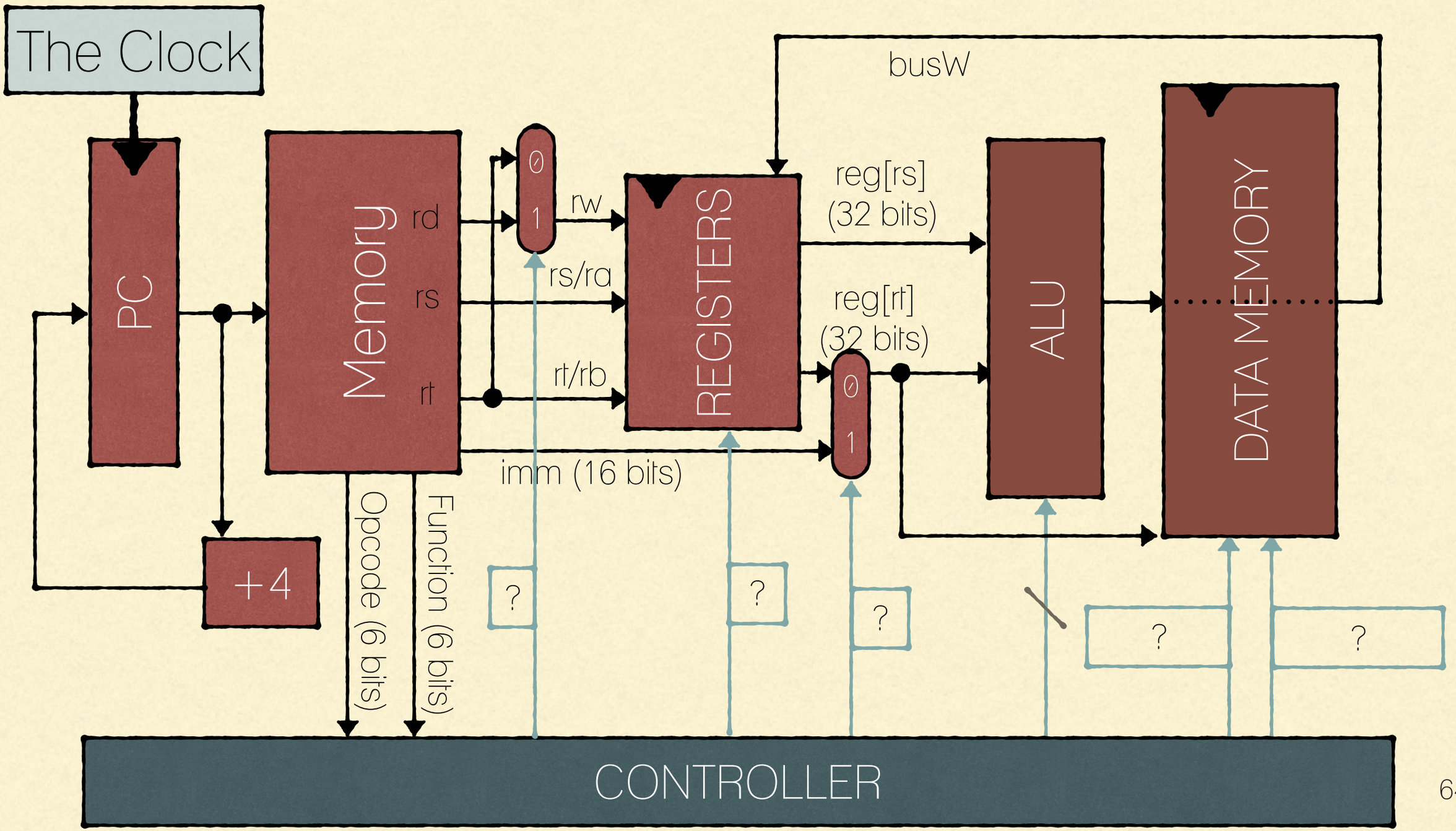

31 26 25 21 20 16 15 11 10 6 5 0
SPECIAL
000000
$t1
01001
$s2
10010
$s3
10011
0
00000
ADD
100000
6 5 5 5 5 6


The Clock
PC
Memory
rd
rs
rt
0
1
rw
rs/ra
rt/rb
REGISTERS
busW
reg[rs]
(32 bits)
reg[rt]
(32 bits)
0
1
ALU
DATA MEMORY
imm (16 bits)
+4
Opcode (6 bits)
Function (6 bits)
1
1
0
0
0
CONTROLLER

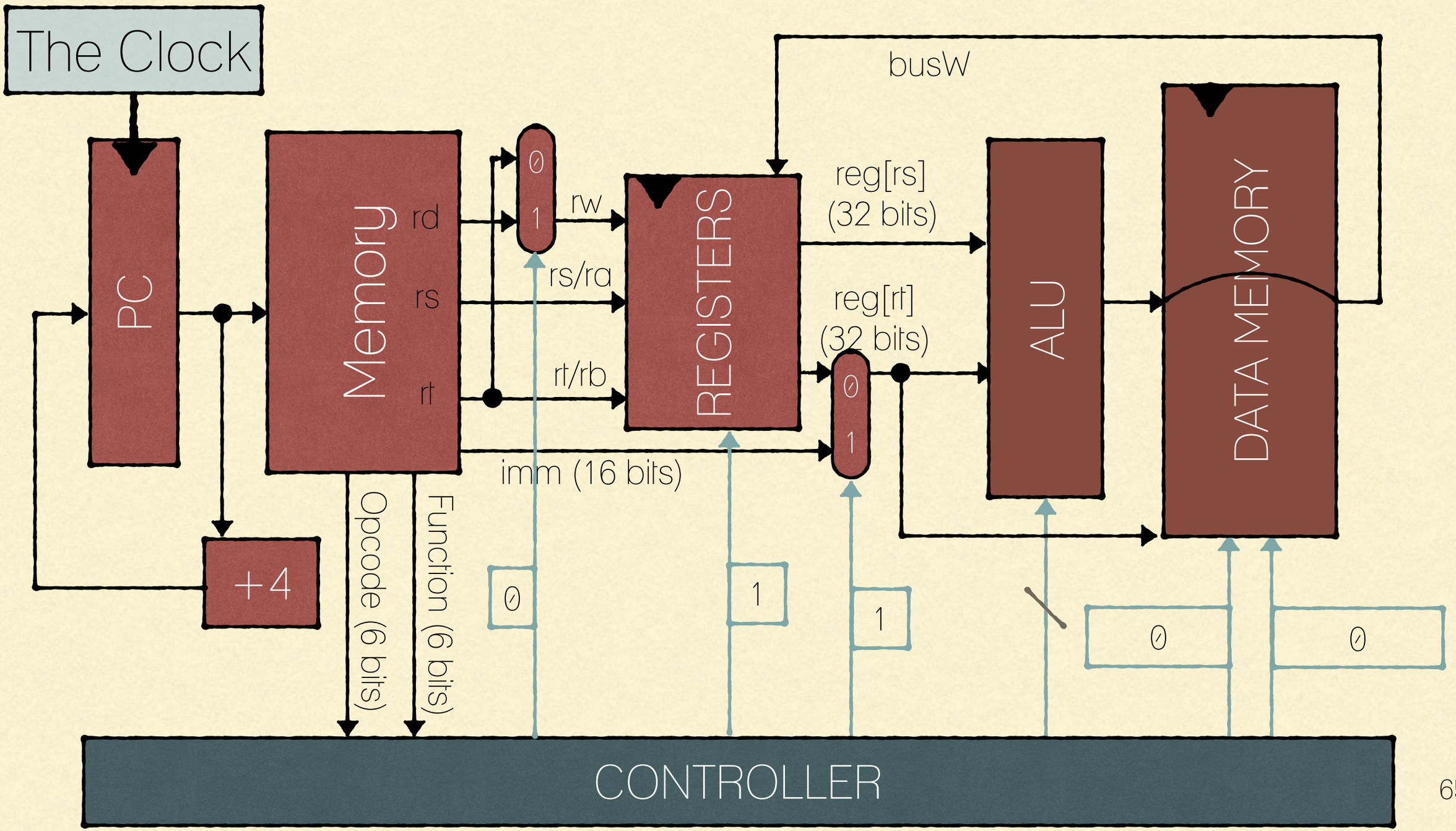
LOGICAL OR WITH IMMEDIATE

- R[rt] ← R[rs] | zero_ext(Imm16); PC ← PC + 4

R[rt] ← R[rs] | zero_ext(Imm16); PC ← PC + 4

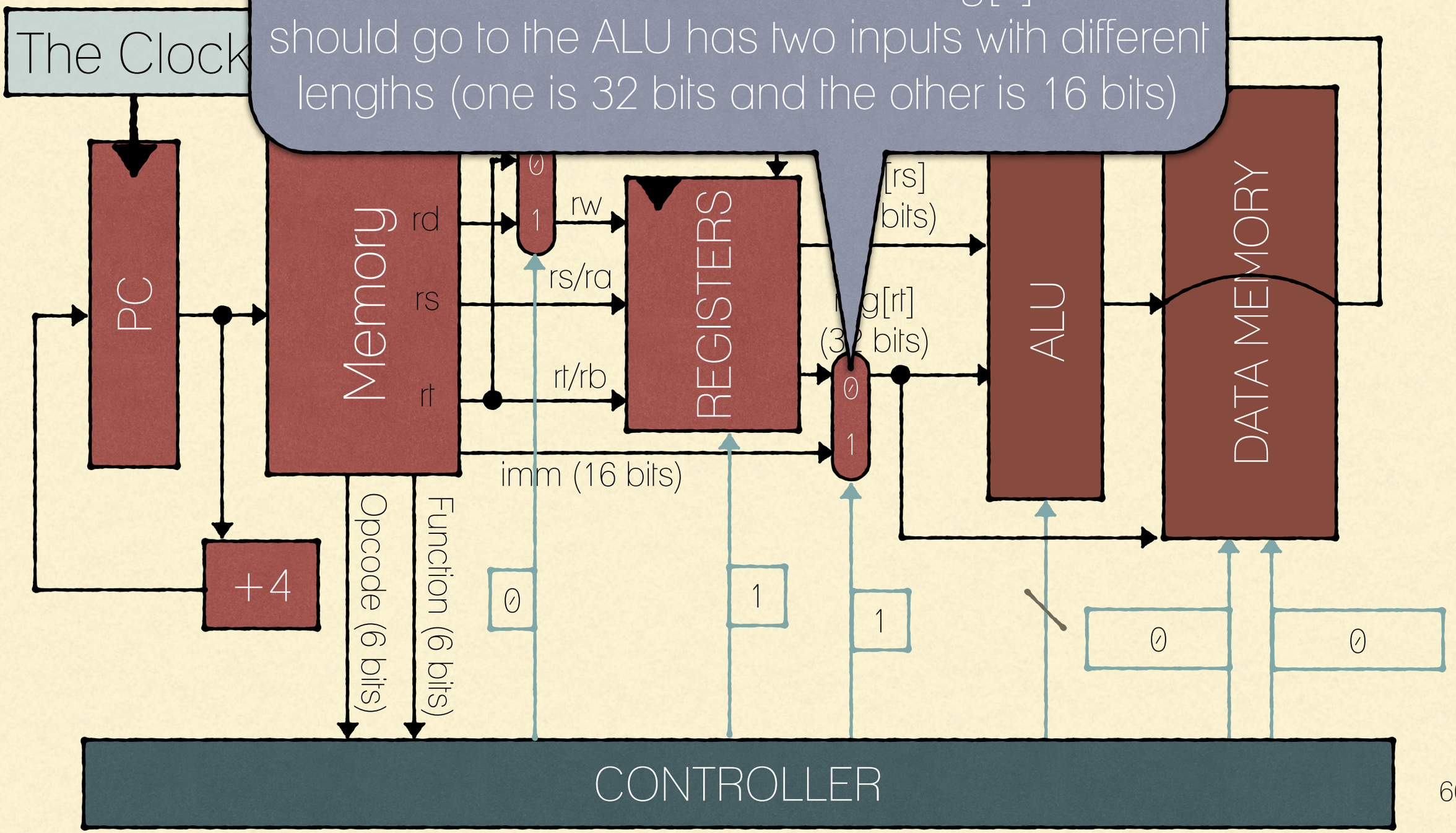


R[rt] ← R[rs] | zero_ext(Imm16); PC ← PC + 4

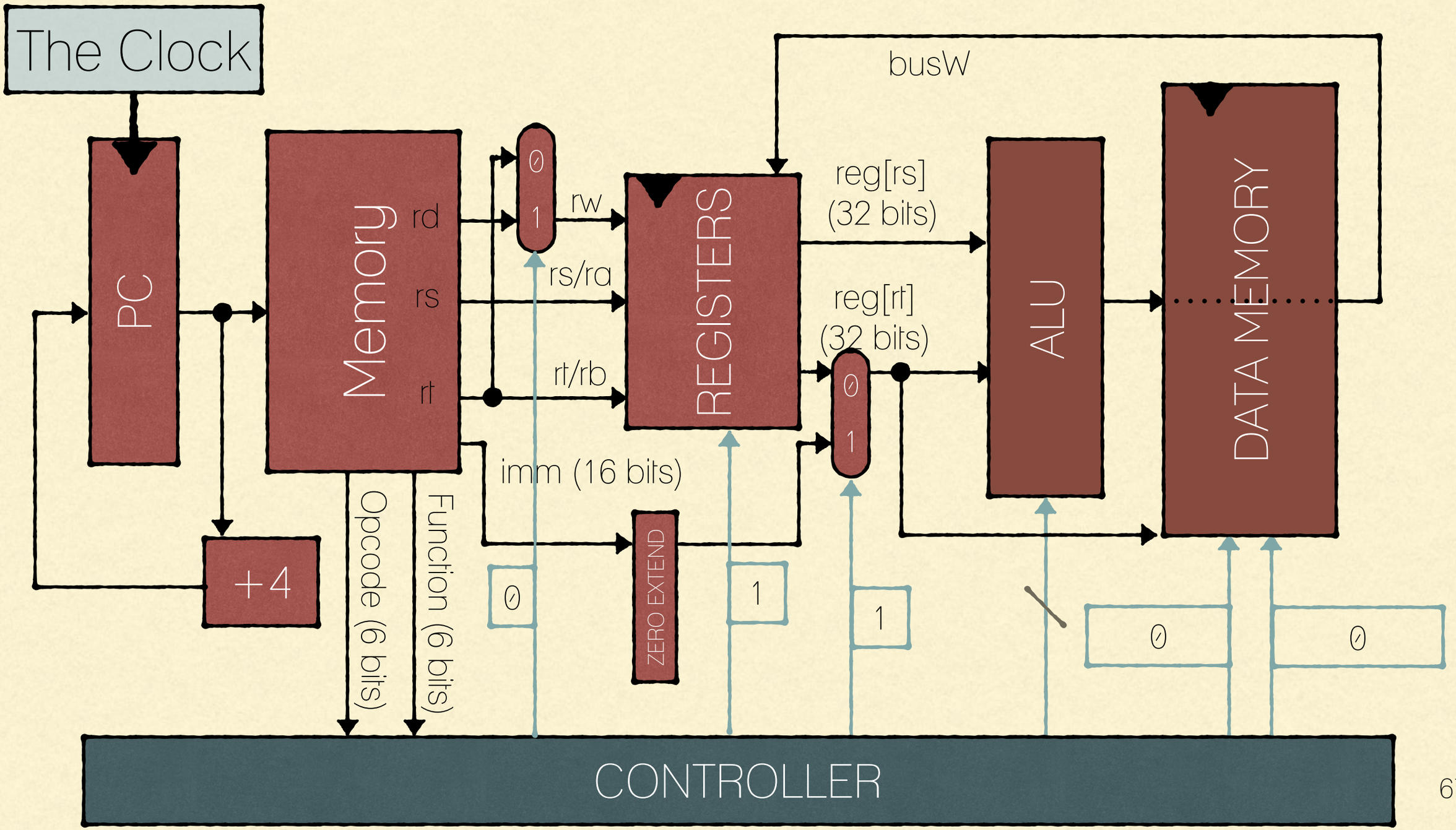


R[rt] ← PC + 4

Here, we have a minor problem!
The 'mux' that will select whether reg[rt] or imm should go to the ALU has two inputs with different lengths (one is 32 bits and the other is 16 bits)



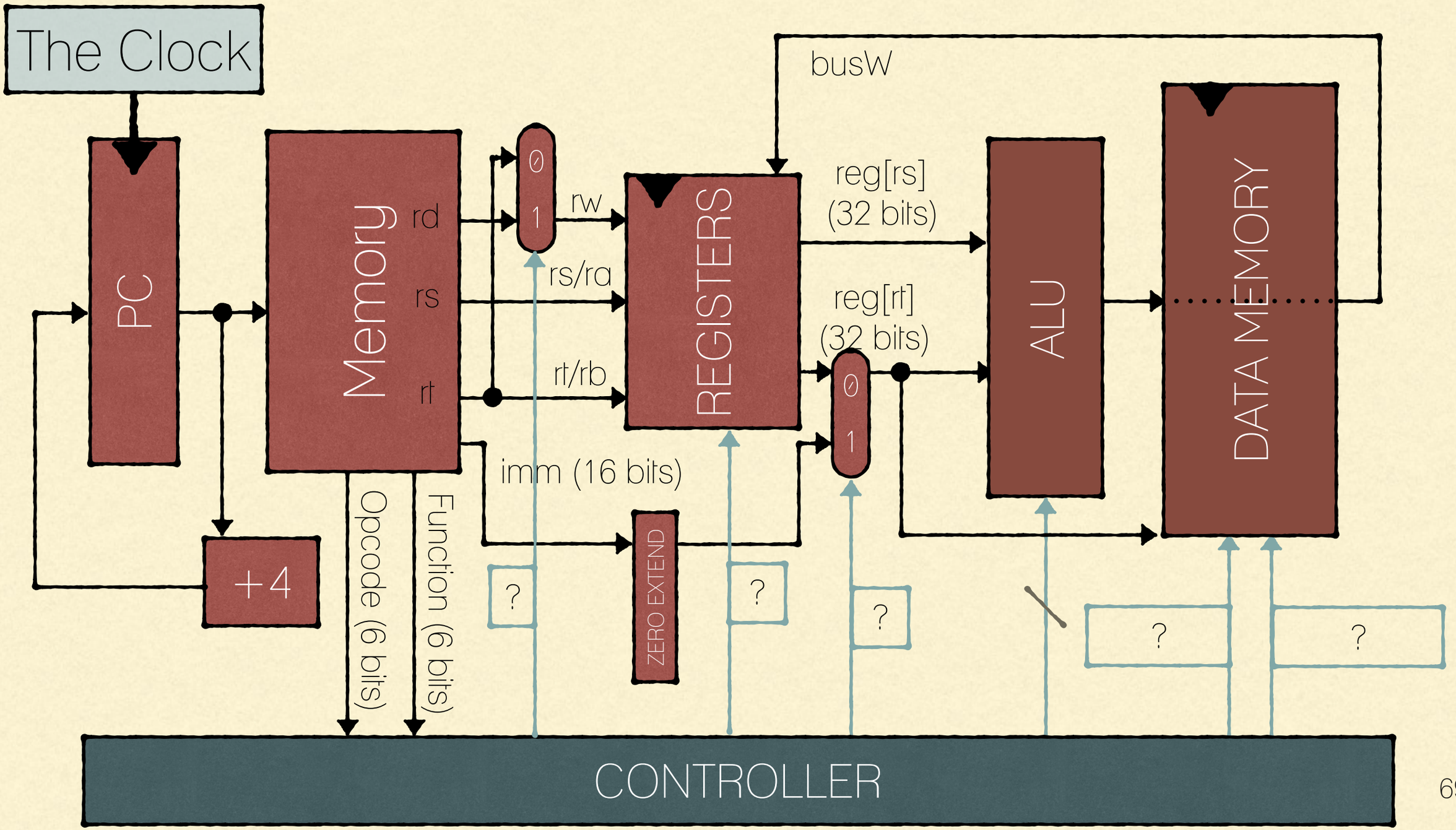
R[rt] ← R[rs] | zero_ext(Imm16); PC ← PC + 4



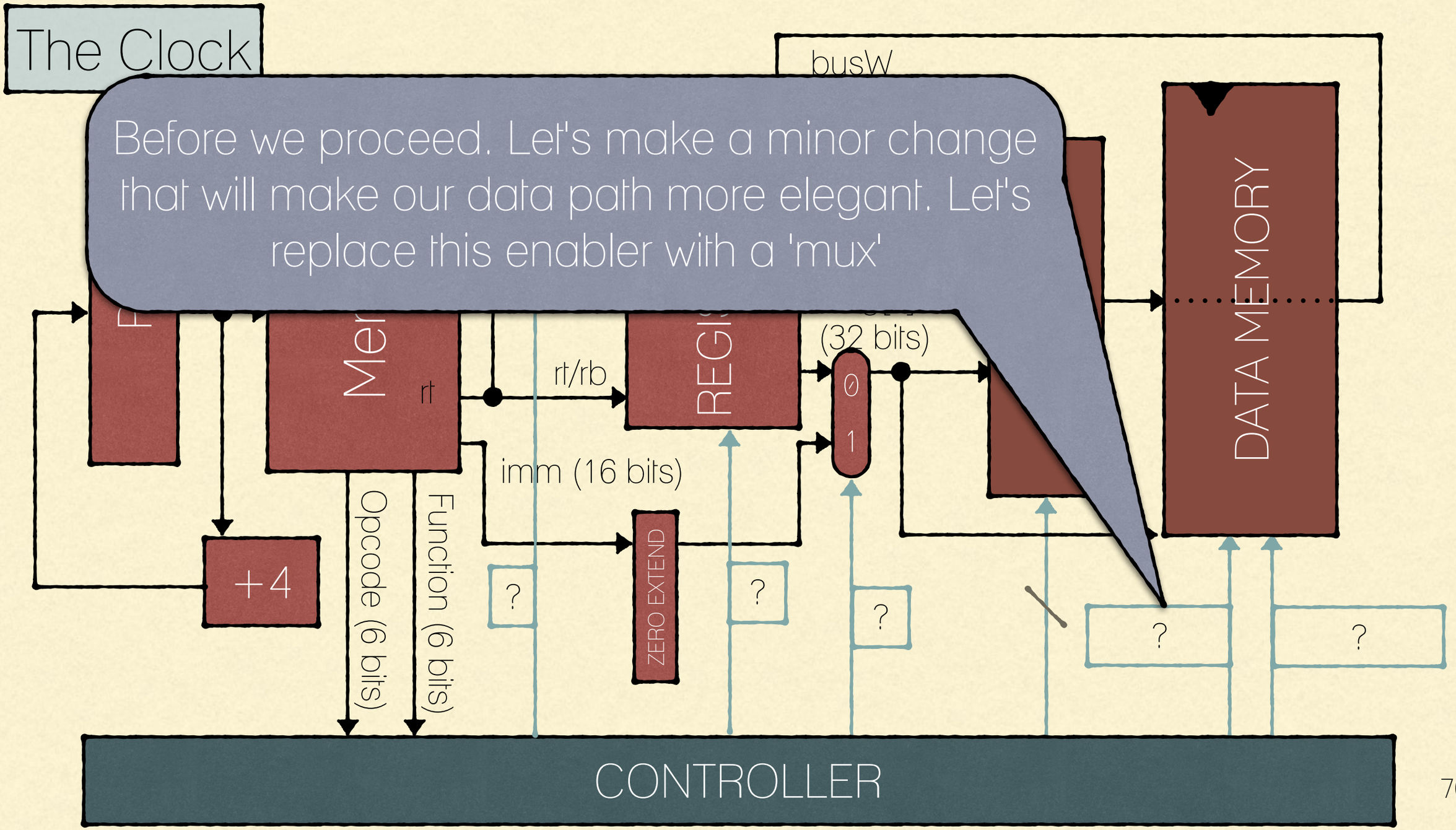
LOAD OPERATIONS

- R[rt] ← MEM[R[rs] + sign_ext(Imm16)]; PC ← PC + 4

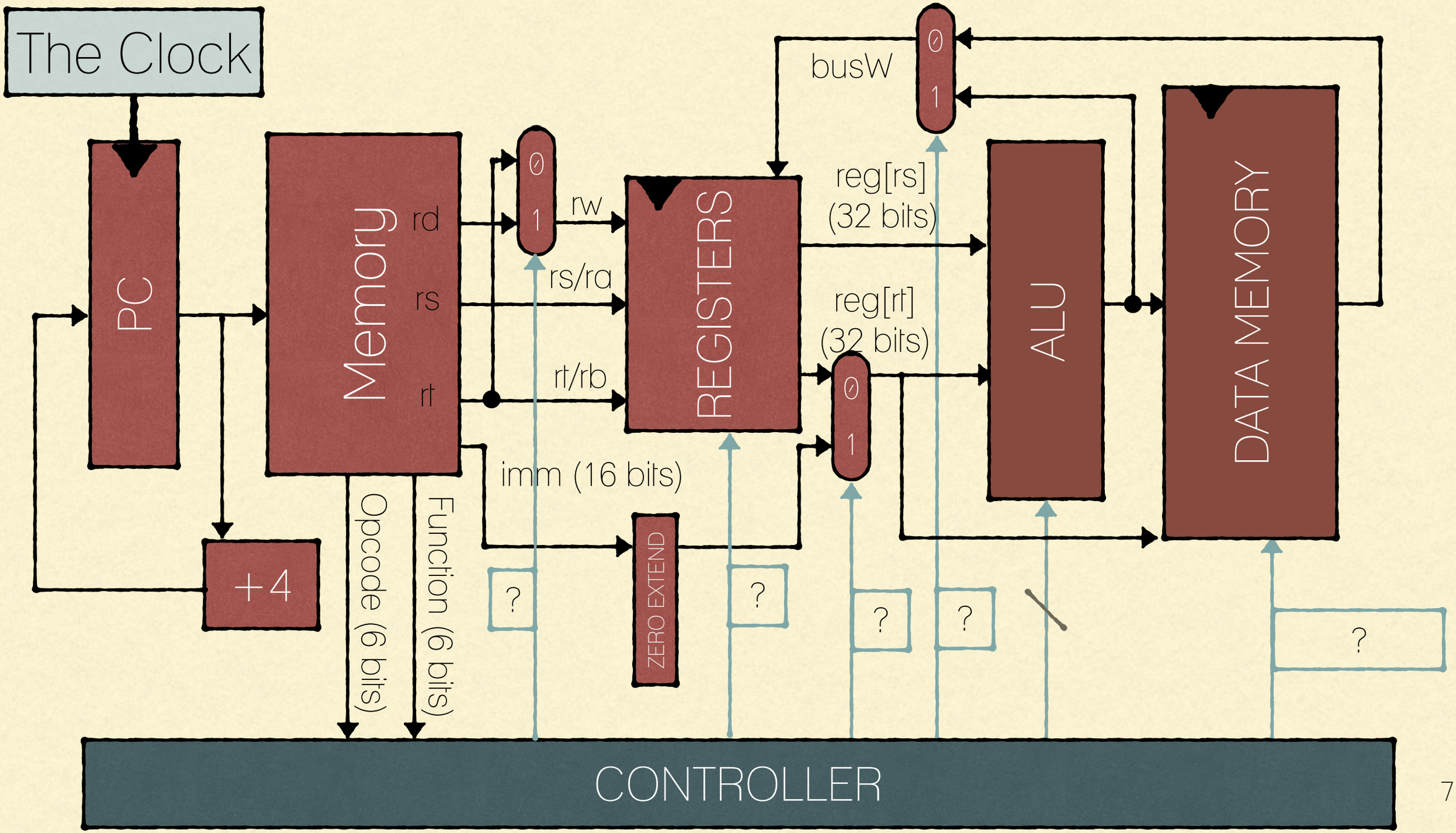
R[rt] ← MEM[R[rs] + sign_ext(Imm16)]; PC ← PC + 4



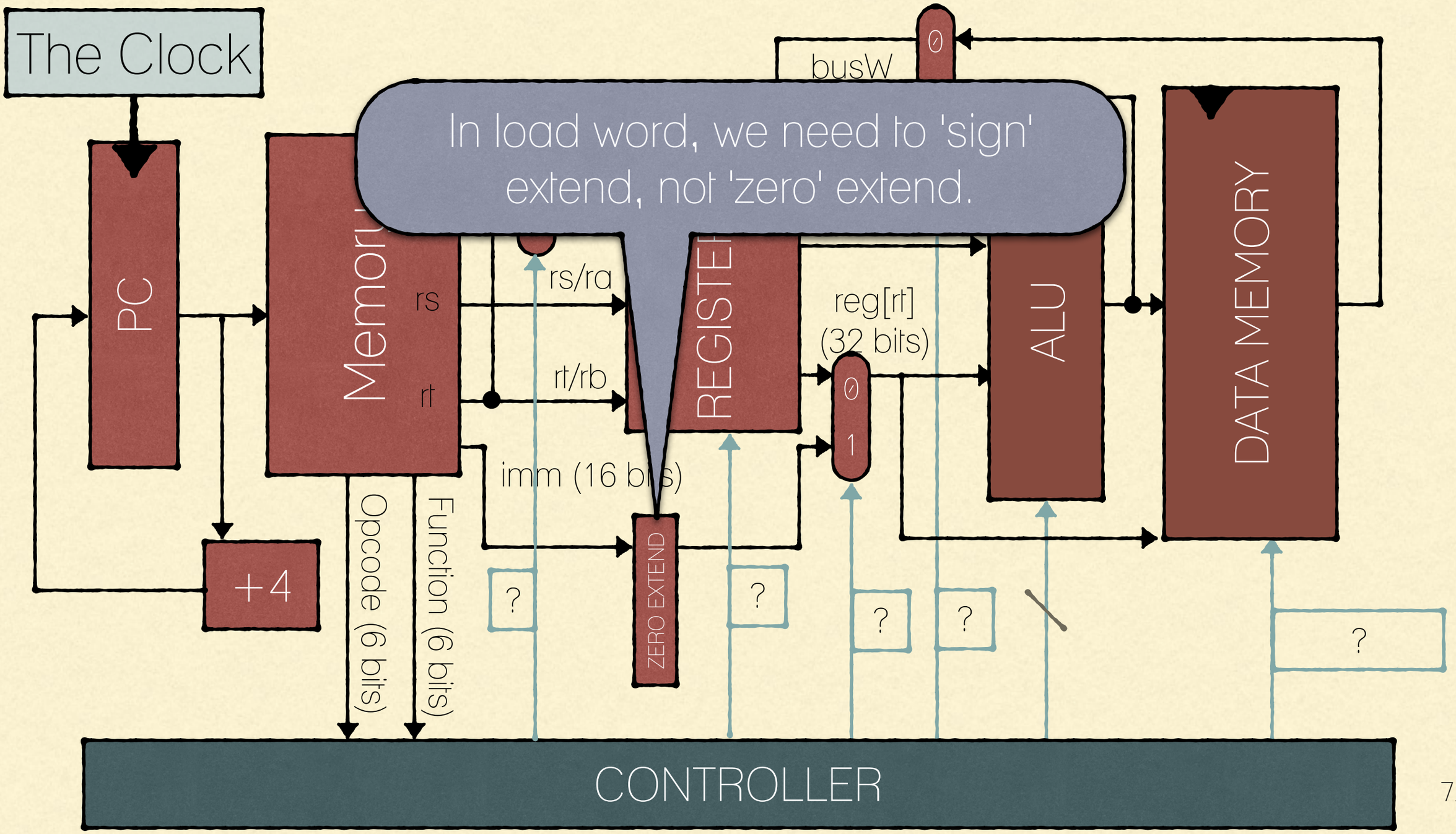
R[rt] ← MEM[R[rs] + sign_ext(Imm16)]; PC ← PC + 4



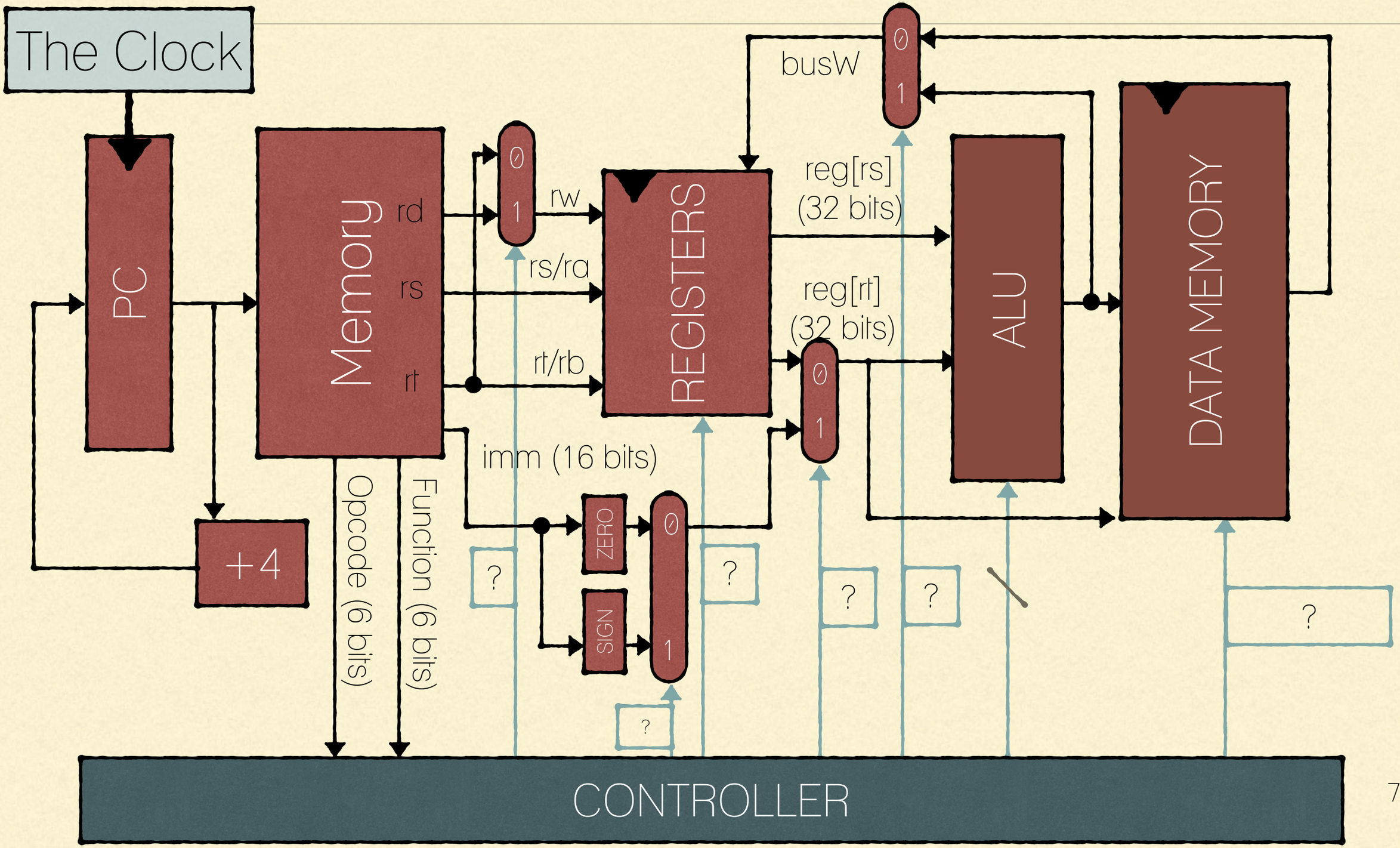
R[rt] ← MEM[R[rs] + sign_ext(Imm16)]; PC ← PC + 4



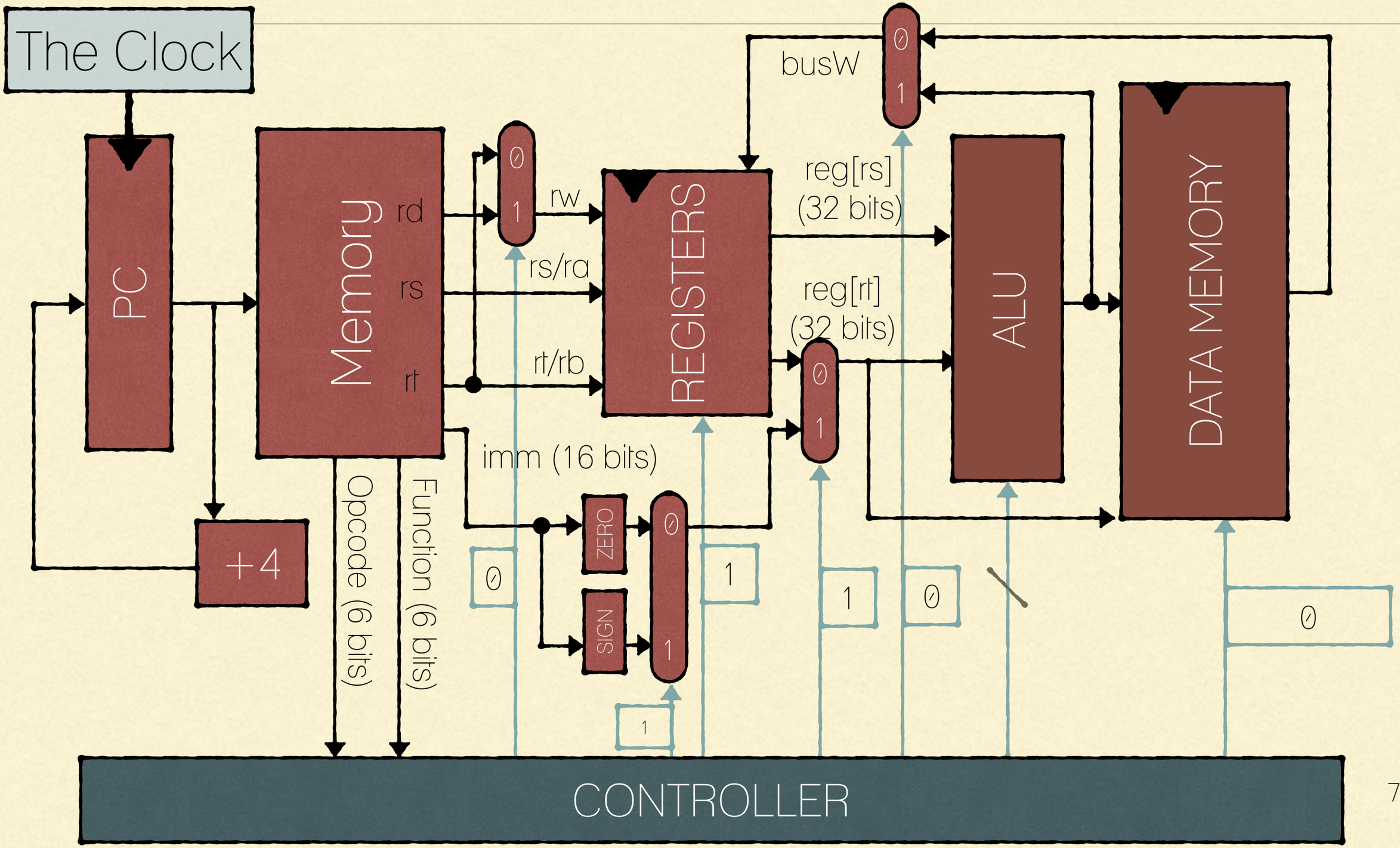
R[rt] ← MEM[R[rs] + sign_ext(Imm16)]; PC ← PC + 4



R[rt] ← MEM[R[rs] + sign_ext(Imm16)]; PC ← PC + 4



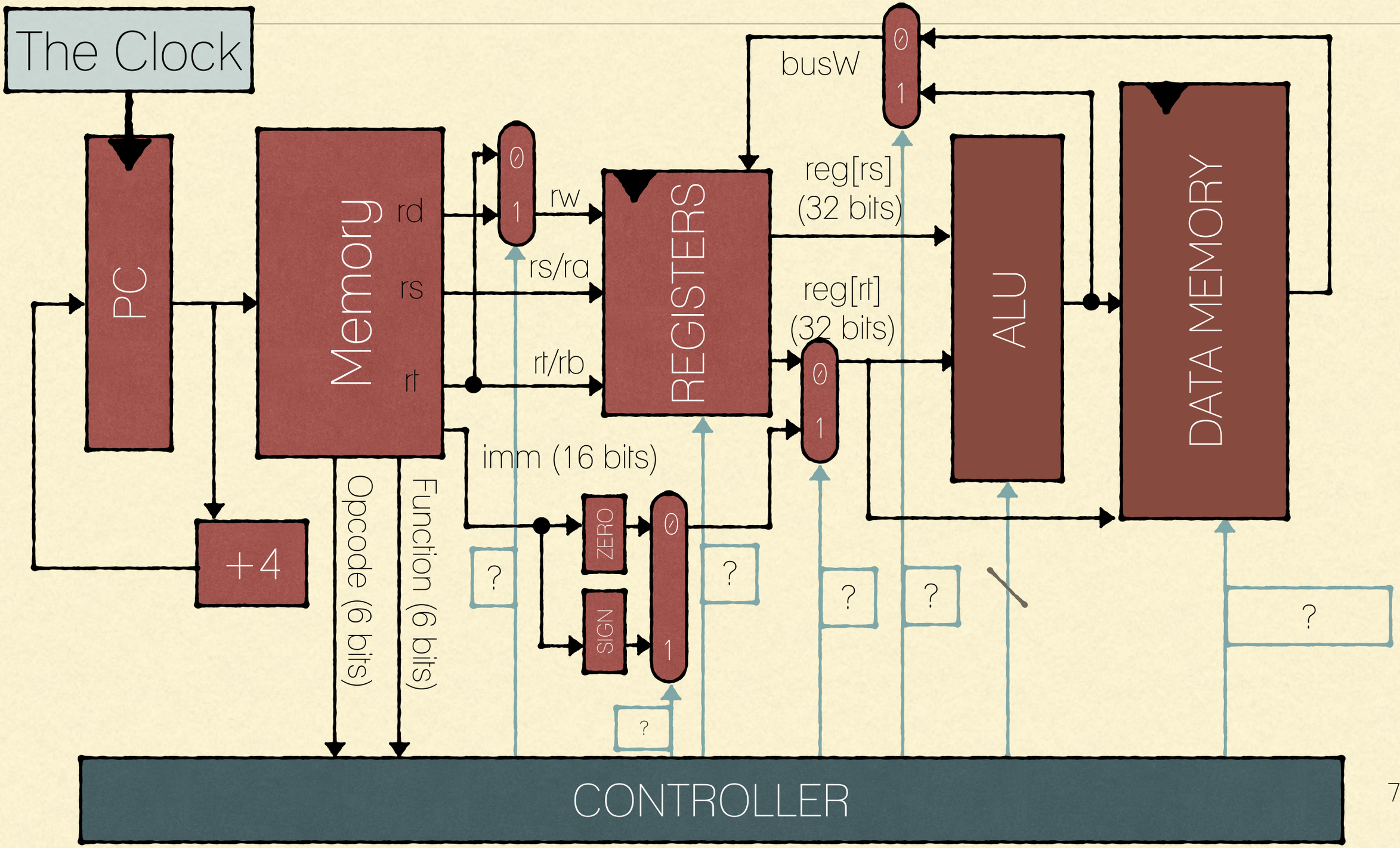
R[rt] ← MEM[R[rs] + sign_ext(Imm16)]; PC ← PC + 4



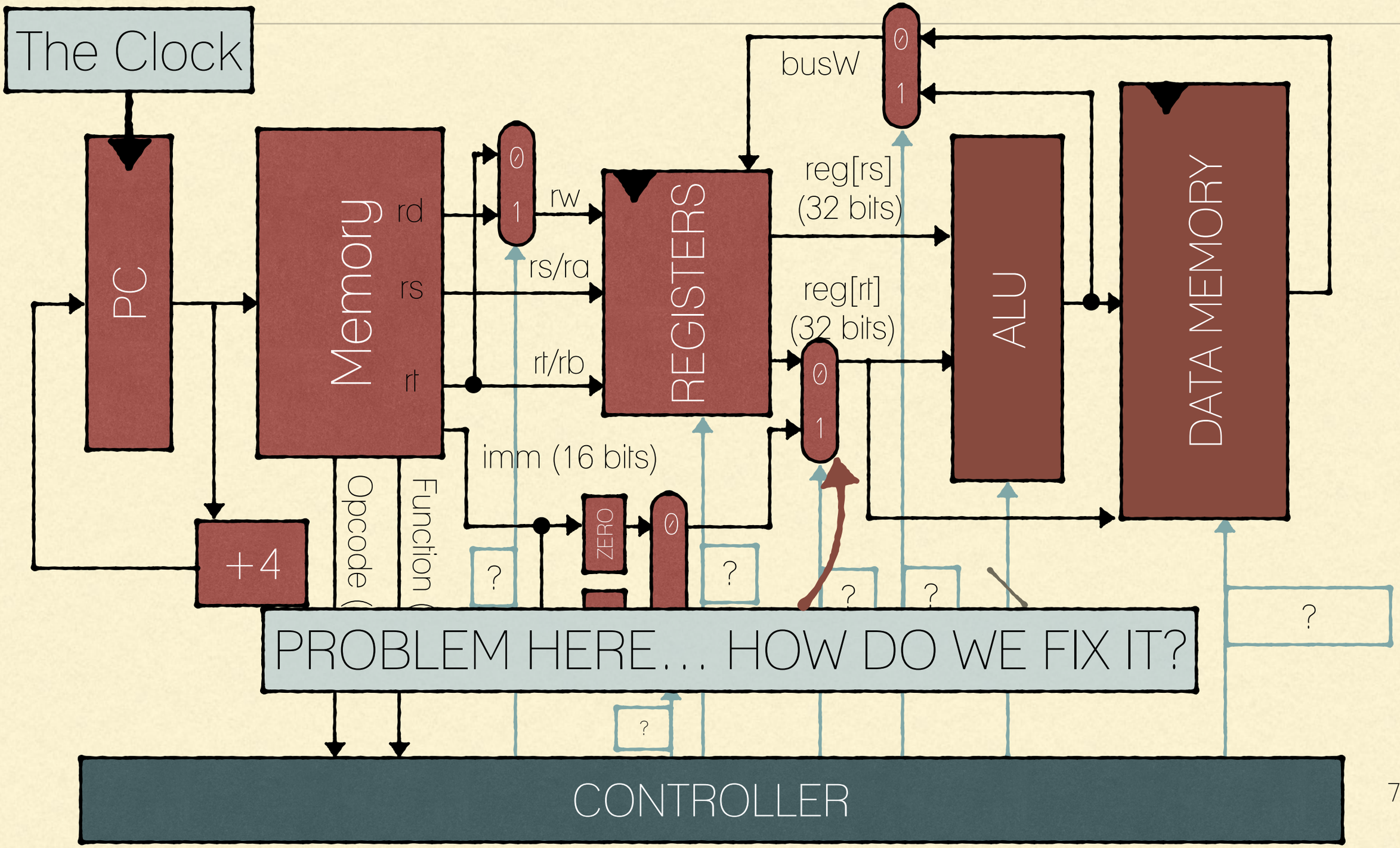
STORE OPERATIONS

- MEM[R[rs] + sign_ext(Imm16)] ← R[rt]; PC ←PC + 4

MEM[R[rs] + sign_ext(Imm16)] ← R[rt]; PC ← PC + 4



MEM[R[rs] + sign_ext(Imm16)] ← R[rt]; PC ← PC + 4



MEM[R[rs] + sign_ext(Imm16)] ← R[rt]; PC ← PC + 4