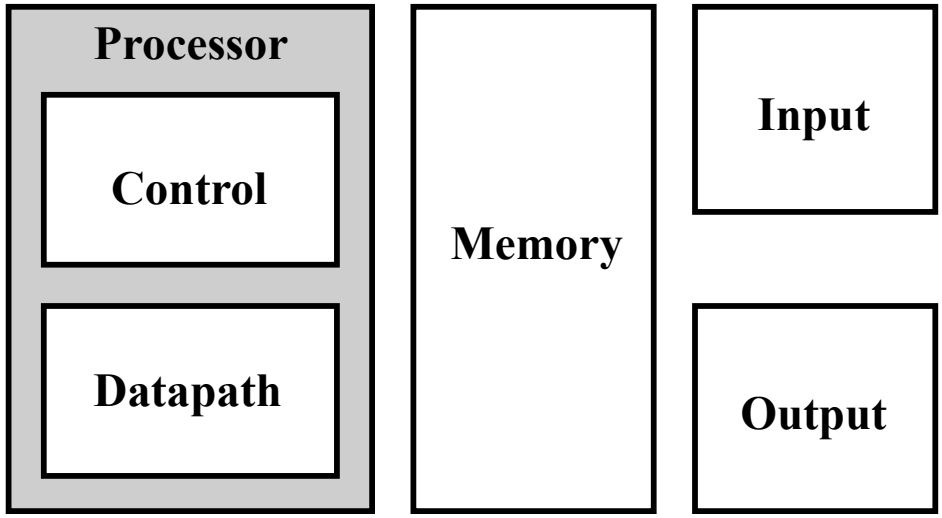
Designing a Single Cycle Datapath

or

The Do-It-Yourself CPU Kit

The Big Picture: Where are We Now?

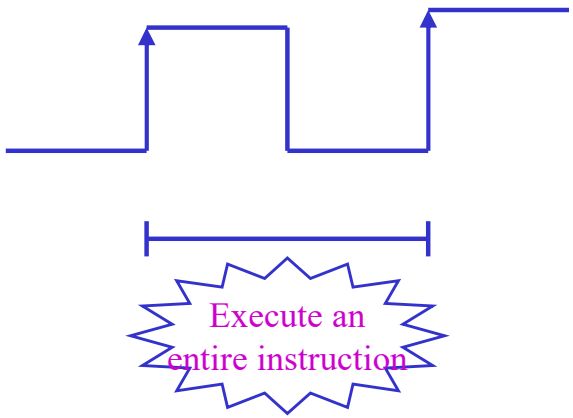
- The Five Classic Components of a Computer



- Today's Topic: the ALU and Datapath Design, next topic Control Design

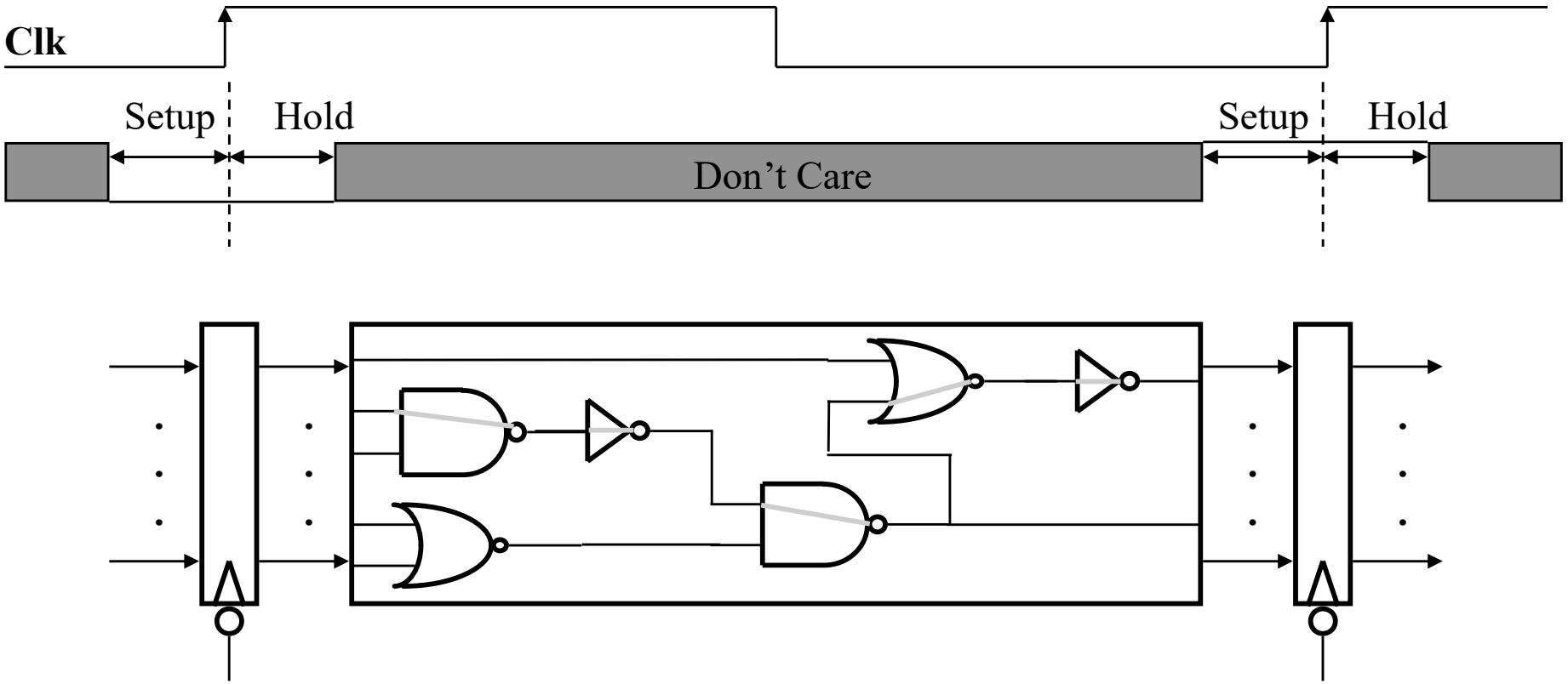
The Big Picture: The Performance Perspective

- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction
- Starting today:
  - Single cycle processor:
    - Advantage: One clock cycle per instruction
    - Disadvantage: long cycle time



- ET = Insts * CPI * Cycle Time

Clocking Methodology



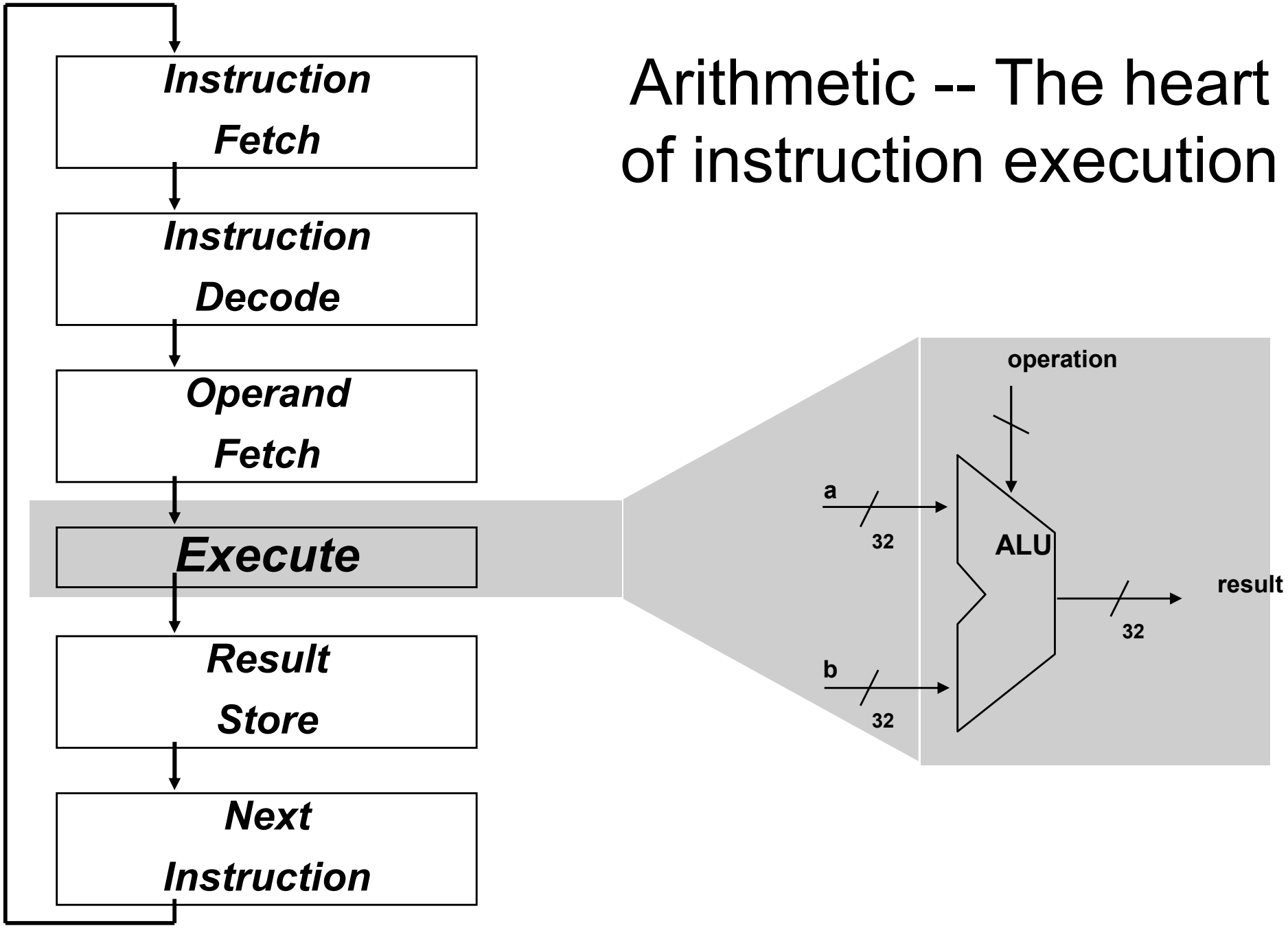
- All storage elements are clocked by the same clock edge

- This picture is valid whether we're talking about a single-cycle processor, multi-cycle processor, pipelined processor, …

The Processor: Datapath & Control

- We're going to look at an implementation of MIPS simplified to contain only:
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq`
- Generic Implementation:
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do

Arithmetic -- The heart of instruction execution



Computer Arithmetic

- We’re going to go over this pretty fast, but it shouldn’t be very much you haven’t seen before.

Okay, let’s stop and recall a little math…

Recall: 2's complement

- We would like a ***number system*** that provides
  - obvious representation of 0,1,2...
  - uses an adder for both unsigned and signed addition
  - single value of 0
  - equal coverage of positive and negative numbers
  - easy detection of sign
  - easy negation

Two's Complement Representation

- 2's complement representation of negative numbers
  - Take the bitwise inverse and add 1
- Biggest 4-bit Binary Number: 7 Smallest 4-bit Binary Number: -8

Decimal	Two's Complement Binary
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Two's Complement Arithmetic

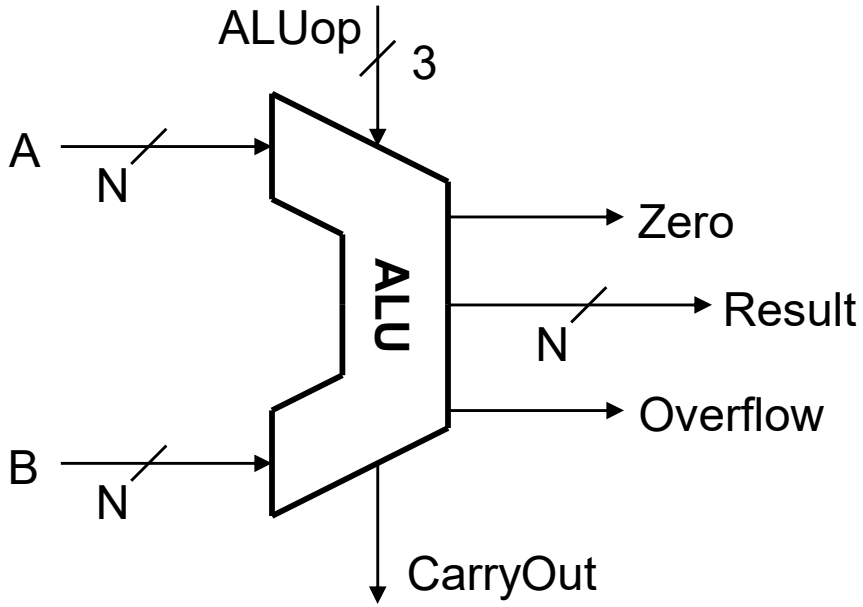
Decimal	2's Complement Binary	Decimal	2's Complement Binary
0	0000	-1	1111
1	0001	-2	1110
2	0010	-3	1101
3	0011	-4	1100
4	0100	-5	1011
5	0101	-6	1010
6	0110	-7	1001
7	0111	-8	1000

- Examples: $7 + (-6) = 1$ $\qquad$ $3 + (-5) = -2$

$$\begin{array}{rcccc} & 0 & 1 & 1 & 1 \\ + & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

$$\begin{array}{rcccc} & 0 & 0 & 1 & 1 \\ + & 1 & 0 & 1 & 1 \\ \hline \end{array}$$

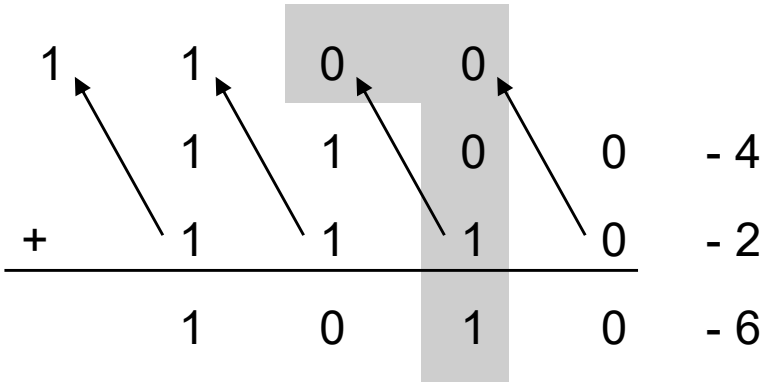
Designing an Arithmetic Logic Unit

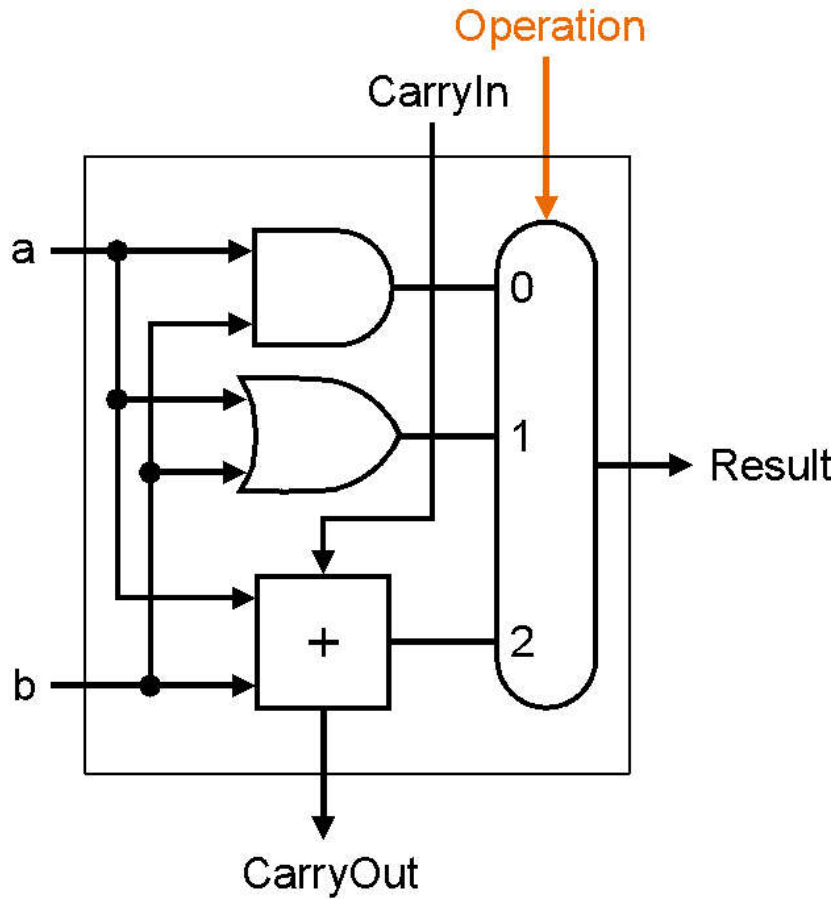


- **ALU Control Lines (ALUop) Function**
  - **000 And**
  - **001 Or**
  - **010 Add**
  - **110 Subtract**
  - **111 Set-on-less-than**

A One Bit ALU

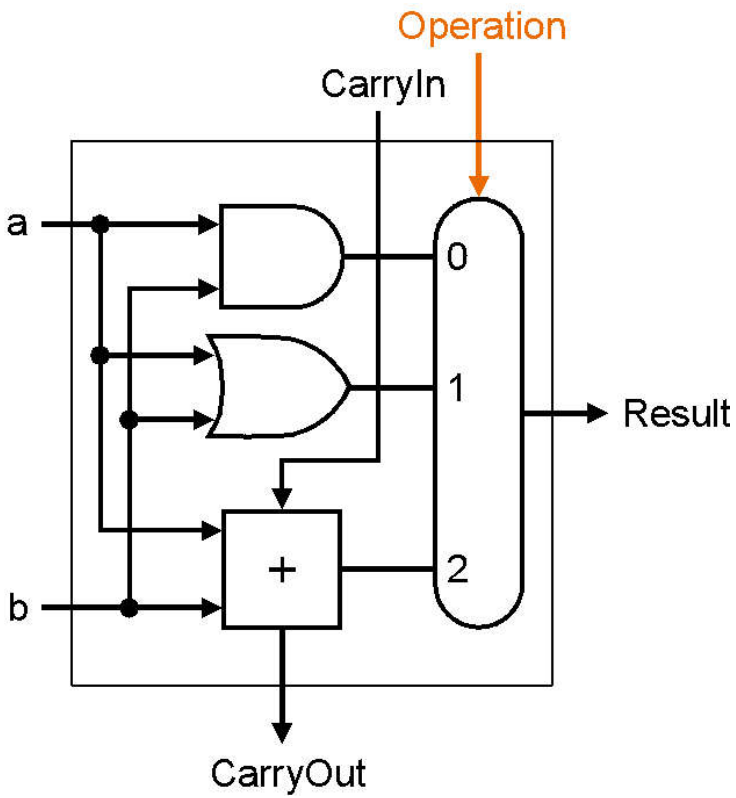
- This 1-bit ALU will perform AND, OR, and ADD for a single bit position.



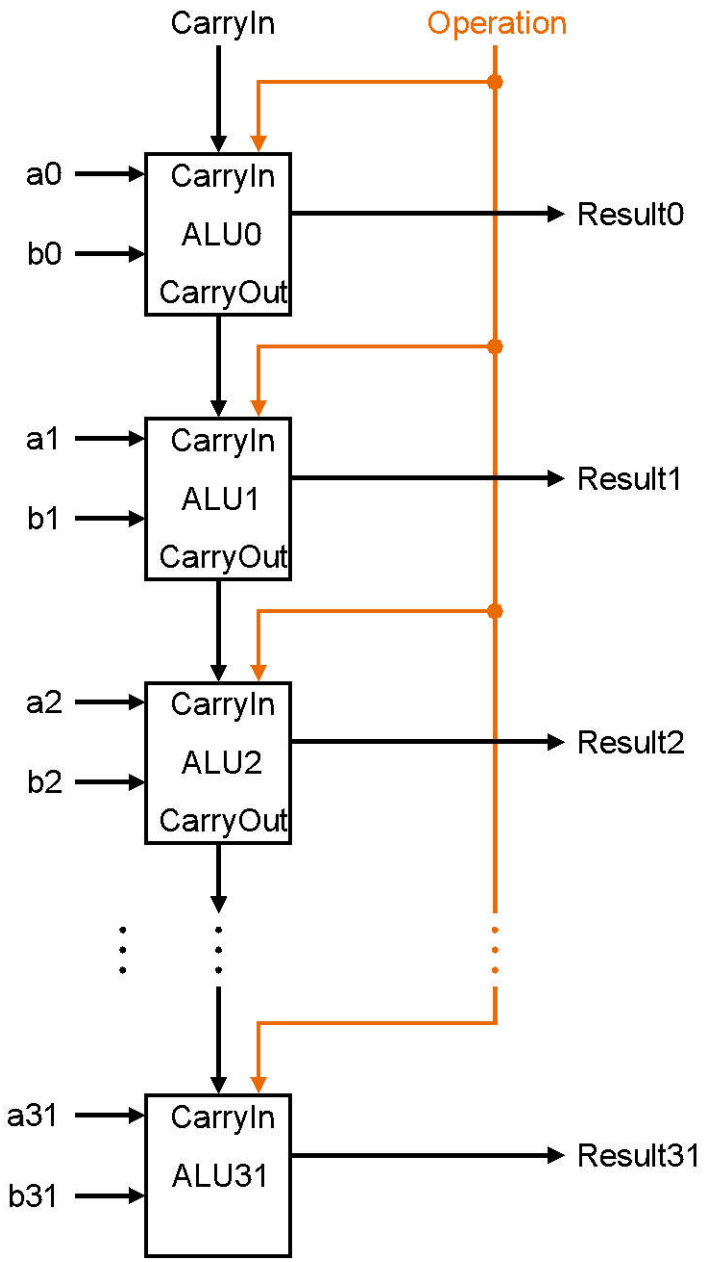


A 32-bit ALU

1-bit ALU

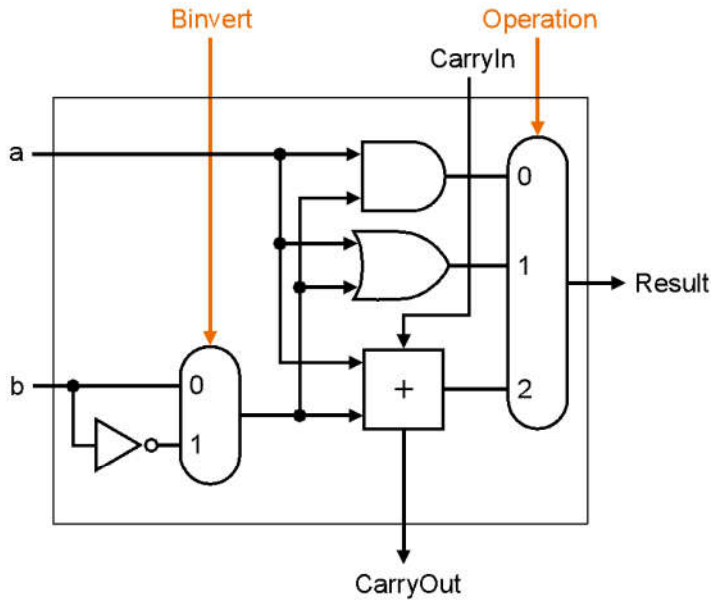


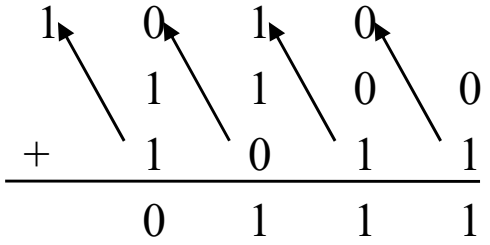
32-bit ALU



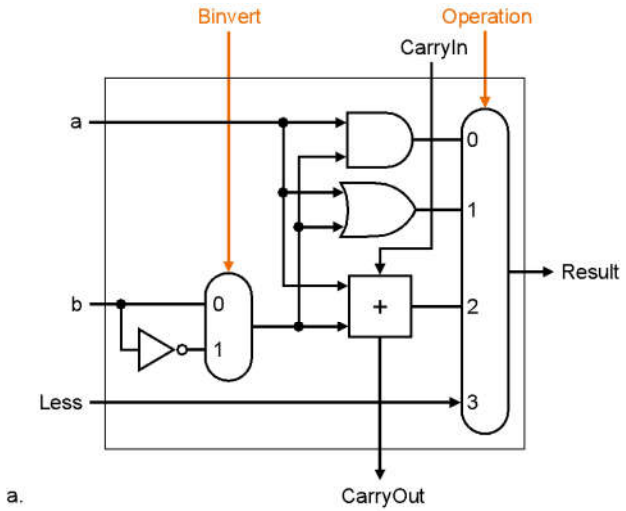
How About Subtraction?

- Keep in mind the following:
  - (A - B) is the same as: A + (-B)
  - 2's Complement negate: Take the inverse of every bit and add 1
- Bit-wise inverse of B is !B:
  - A - B = A + (-B) = A + (!B + 1) = A + !B + 1
  - How does this help us?

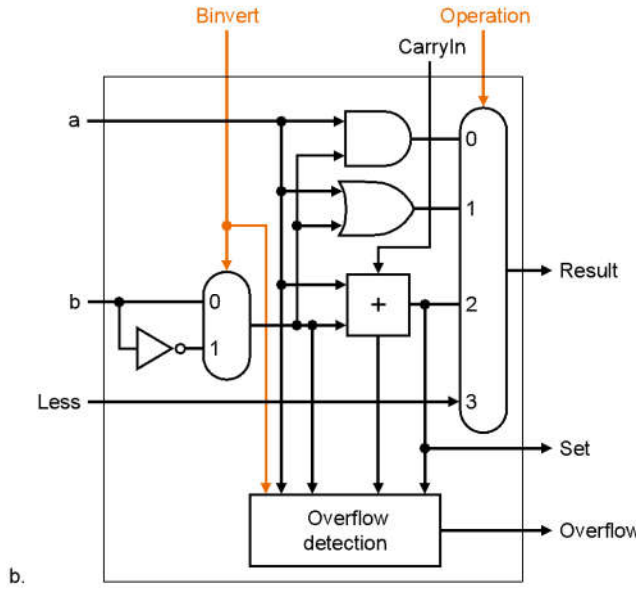




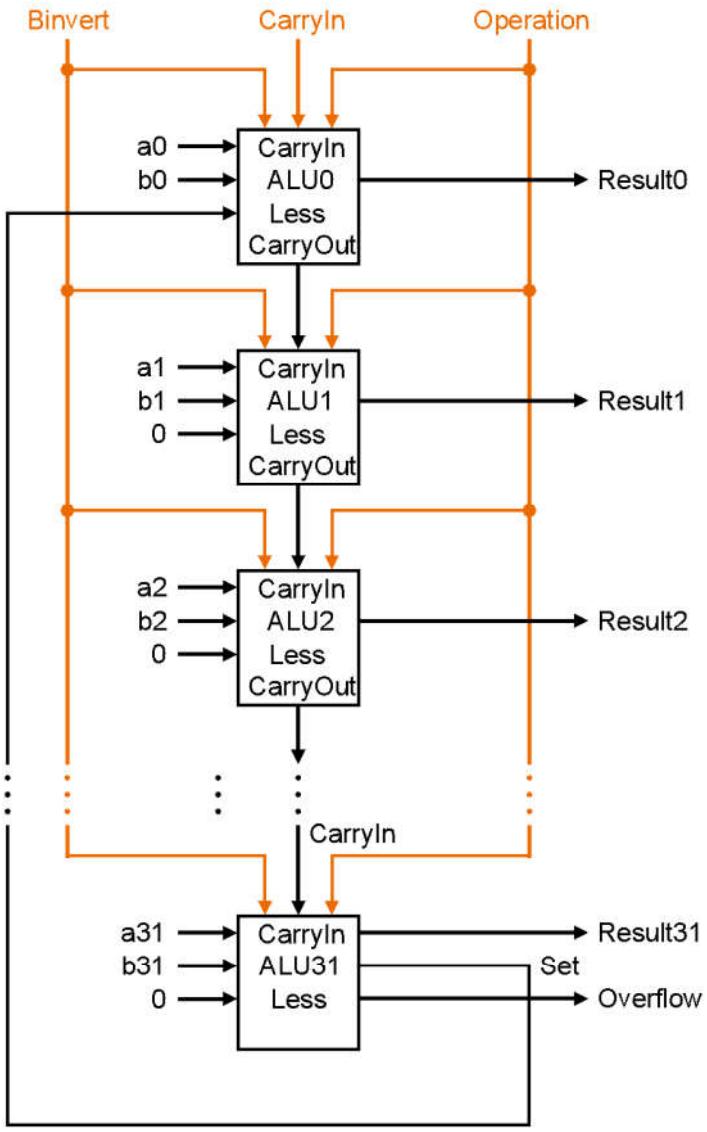
Full ALU



a.



b.

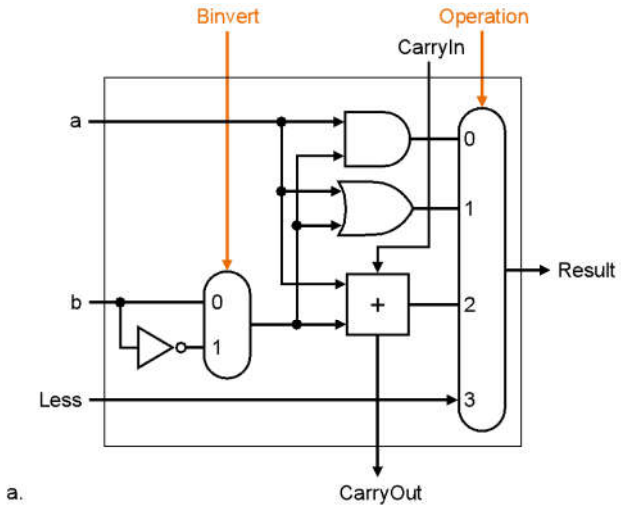


sign bit (adder output from bit 31)

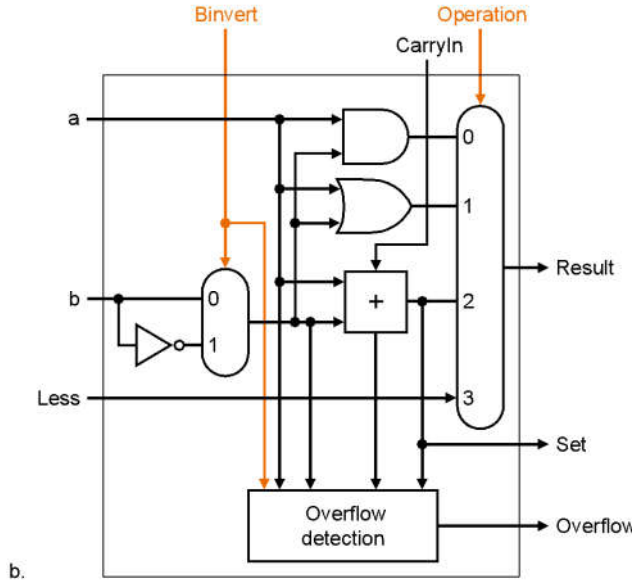
what signals accomplish ADD?

	Binvert	CIn	Oper
A	1	0	2
B	0	1	2
C	1	1	2
D	0	0	2
E	NONE OF THE ABOVE		

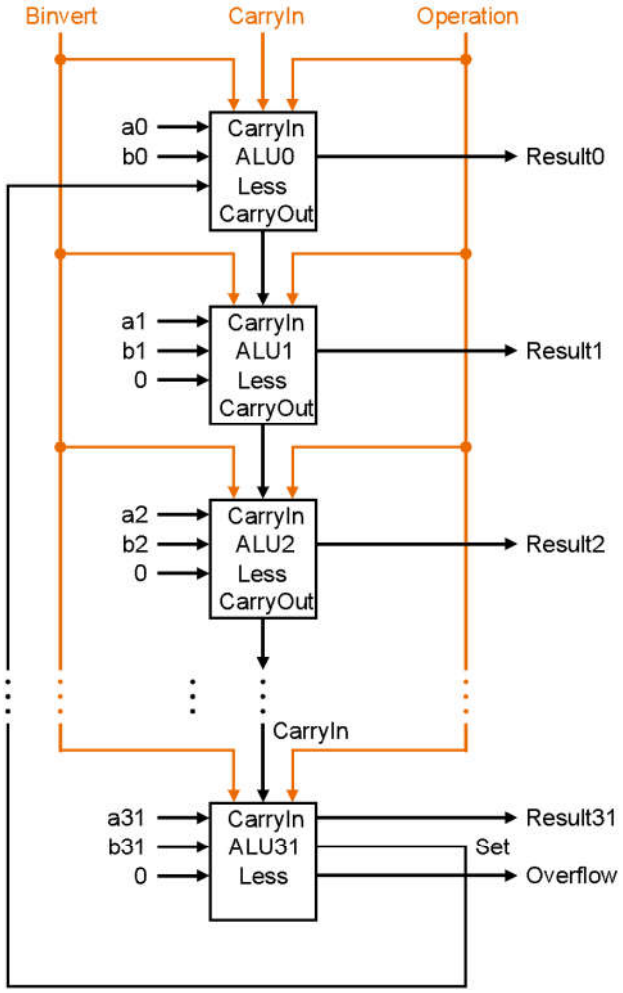
Full ALU



a.



b.



sign bit (adder output from bit 31)

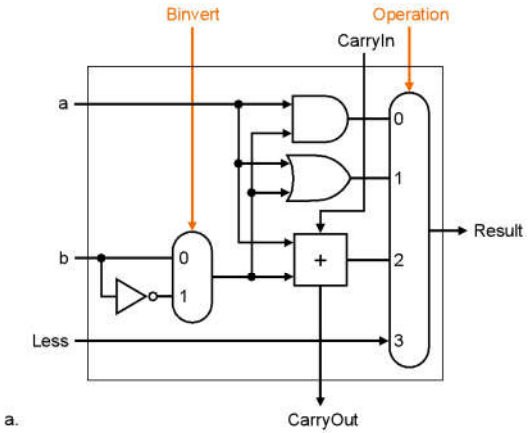
Recall:
slt $t0, $t1, $t2
Means:
if($t1<$t2)
$t0 = 1
else
$t0 = 0

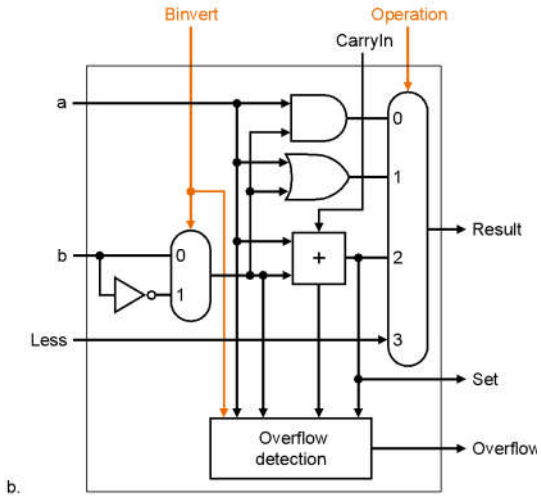
Which signals accomplish SLT
(assume A is $t1, B is $t2)

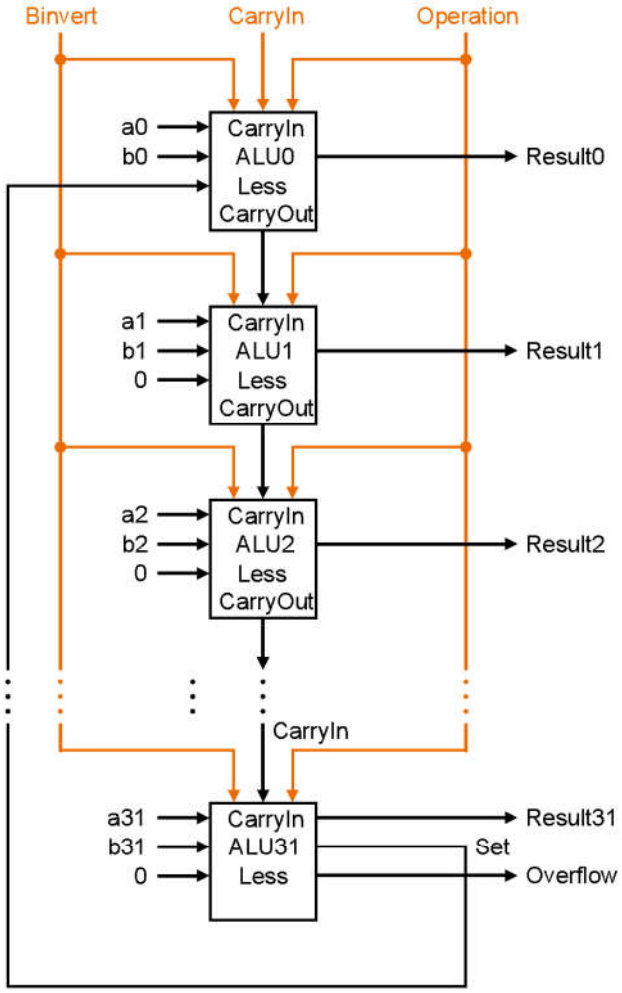
	Binvert	CIn	Oper
A	1	0	2
B	0	1	2
C	1	1	3
D	0	0	3
E	NONE OF THE ABOVE		

(not a poll question – let's work it out together).

Full ALU







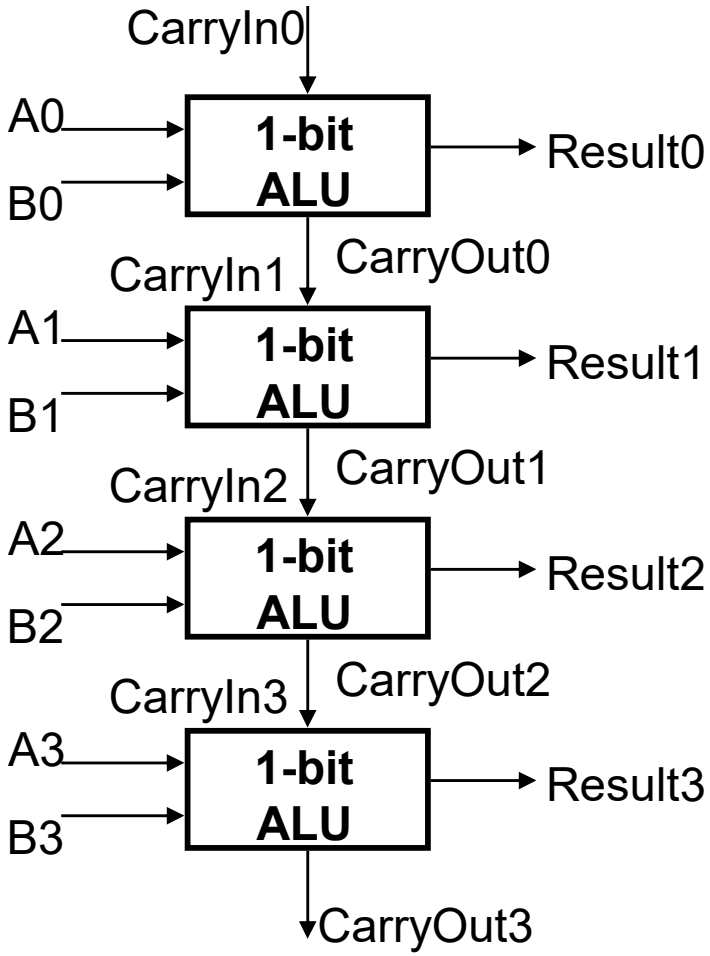
sign bit (adder output from bit 31)

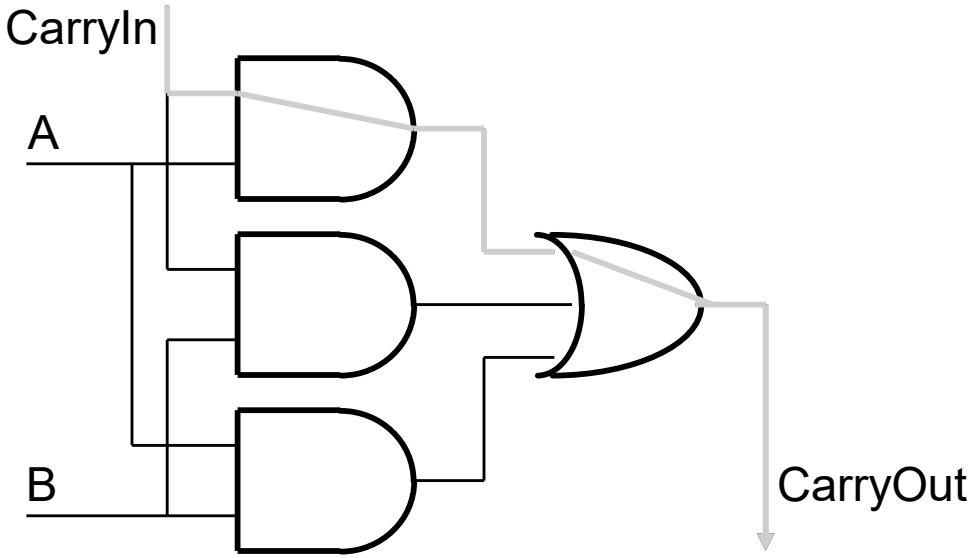
what signals accomplish?

	Binvert	CIn	Oper
add?			
sub?			
and?			
or?			
beq?			
slt?			

The Disadvantage of Ripple Carry

- The adder we just built is called a “Ripple Carry Adder”
  - The carry bit may have to propagate from LSB to MSB
  - Worst case delay for an N-bit RC adder: 2N-gate delay

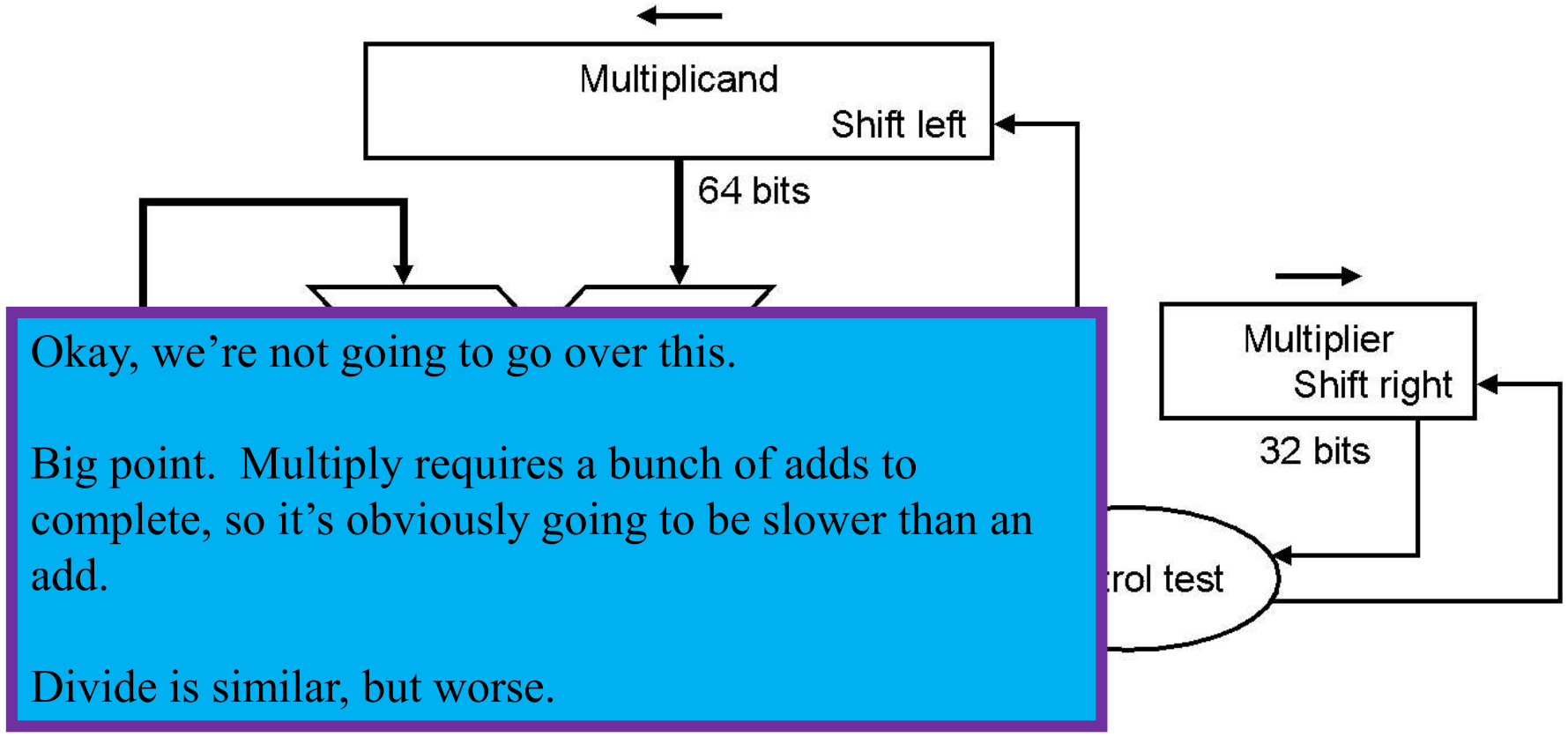




The point -> ripple carry adders are slow. Faster addition schemes are possible that *accelerate* the movement of the carry from one end to the other.

MULTIPLY HARDWARE

- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg



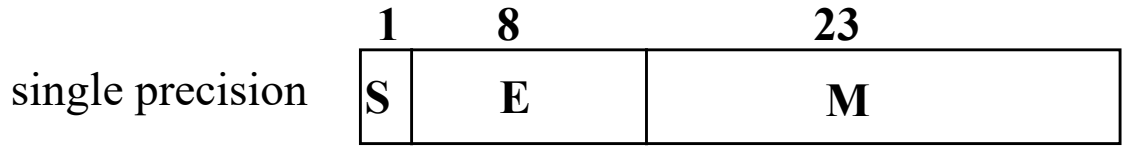
Okay, we're not going to go over this.

Big point. Multiply requires a bunch of adds to complete, so it's obviously going to be slower than an add.

Divide is similar, but worse.

Floating-Point Numbers

Representation of floating point numbers in IEEE 754 standard:



sign exponent: mantissa:

de, normalized
and w/ hidden
it: 1.M

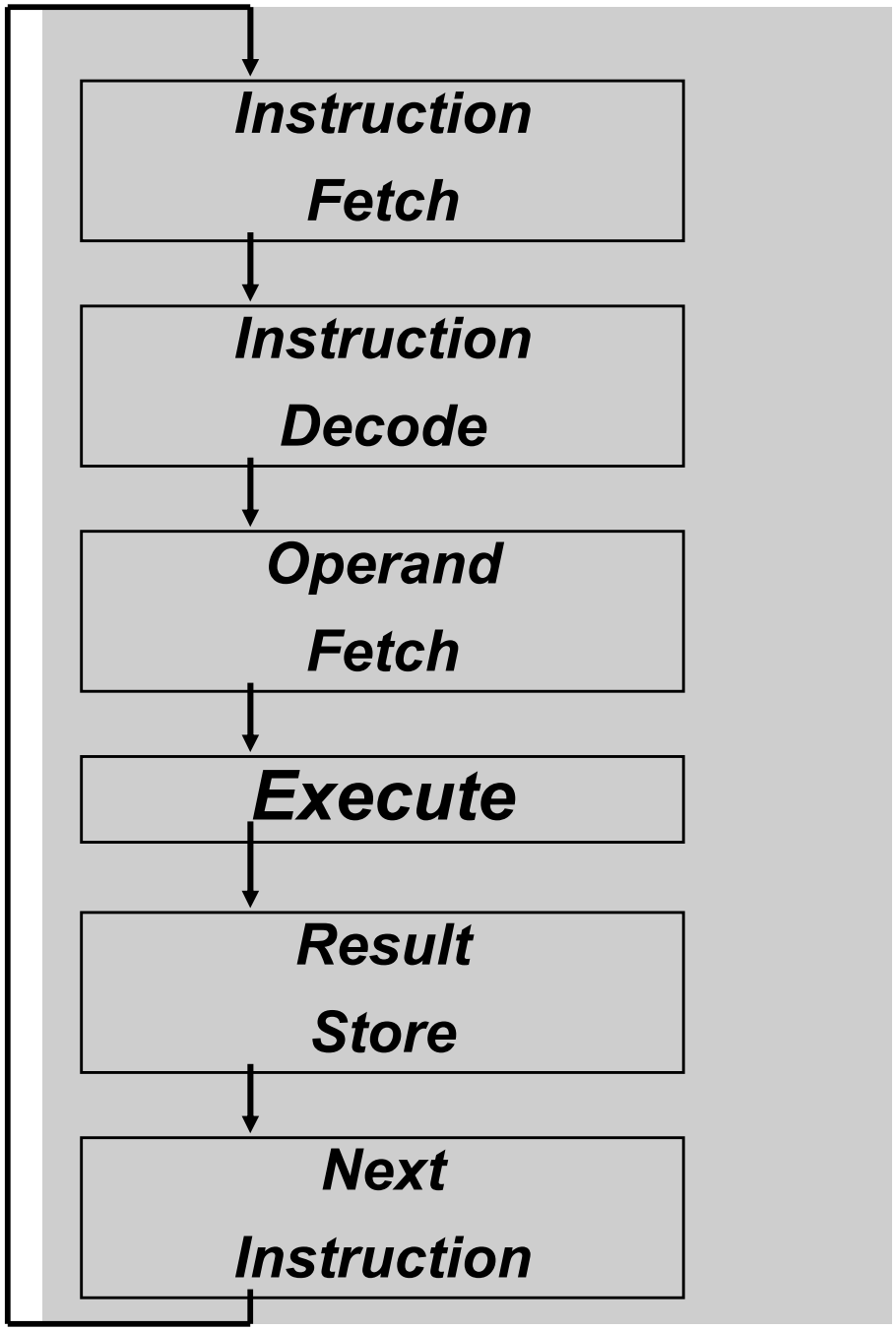
Okay, how much do we remember about floating point numbers? Do we really need to go over this?

Maybe just some high level points?

Floating Point

- Typically will have their own ALU (and register file!)
- Because they require normalization, then math, then renormalization, then rounding (a lot of serialized steps)…
- FP computations are always slower than integer.
  - Eg, Skylake
    - Int add 1, int mul 4, FP add 3, FP mul 5

- Okay, now that we have an ALU…



Let's look beyond just Execute now.

Register Transfer Language (RTL)

- is a mechanism for describing the movement and manipulation of data between storage elements:

R[3] <- R[5] + R[7]

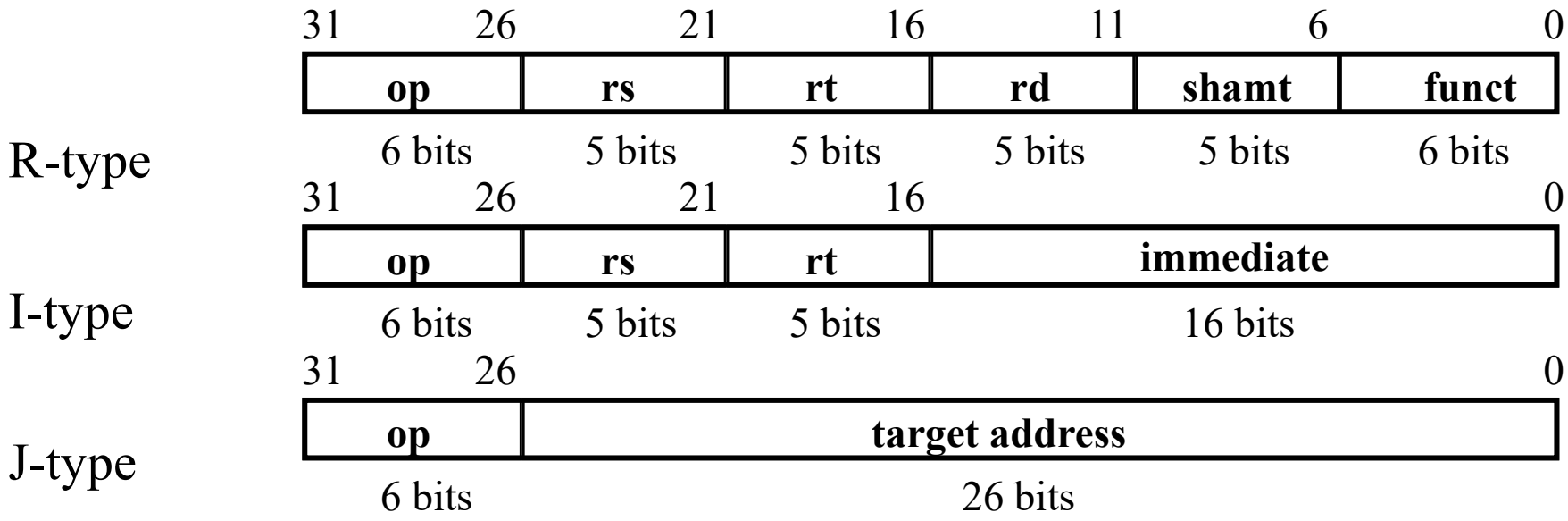
PC <- PC + 4 + R[5]

R[rd] <- R[rs] + R[rt]

R[rt] <- Mem[R[rs] + immed]

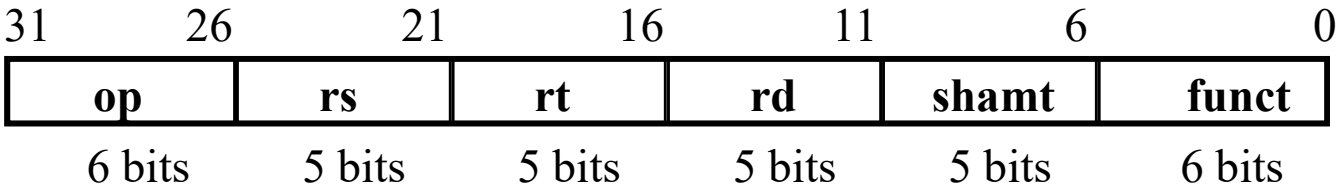
Review: The MIPS Instruction Formats

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



The MIPS Subset

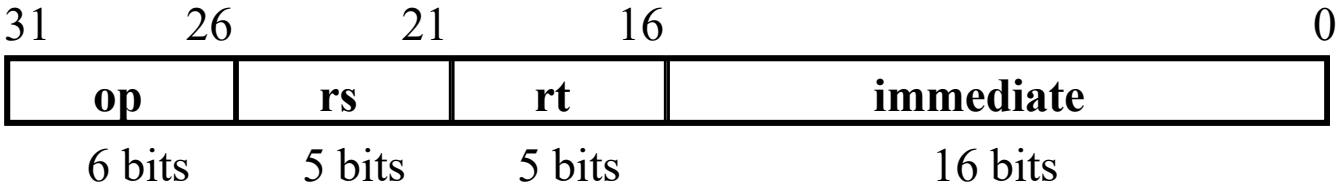
- R-type
  - *add rd, rs, rt*
  - sub, and, or, slt



PC = PC+4

R[rd] = R[rs] OP R[rt]

- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16

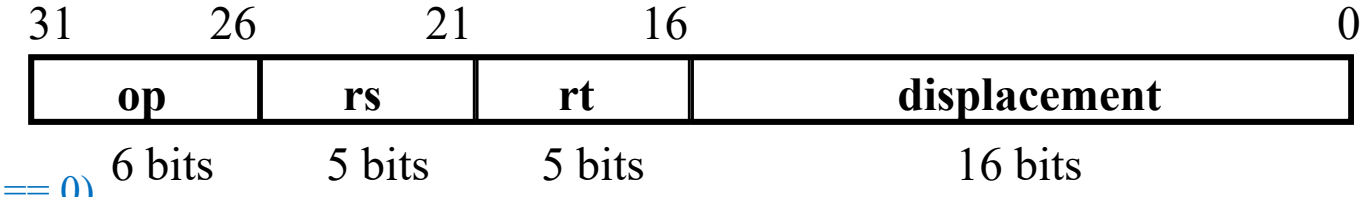


PC = PC+4

R[rt] = Mem[R[rs] + SE(imm)] OR

Mem[R[rs] + SE(imm)] = R[rt]

- BRANCH:
  - beq rs, rt, imm16



ZERO = (R[rs] – R[rt] == 0)

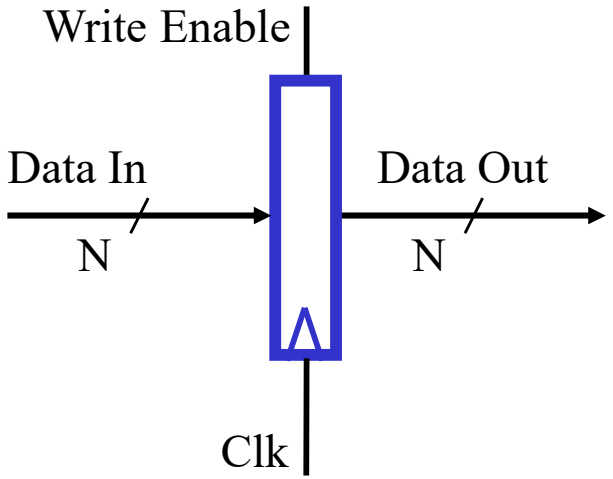
PC = if (ZERO) PC + 4+ (SE(Imm)<<2)

Else PC = PC+4

Storage elements

- RTL describes data movement between storage elements, but we don’t actually have our data elements yet.
- So…

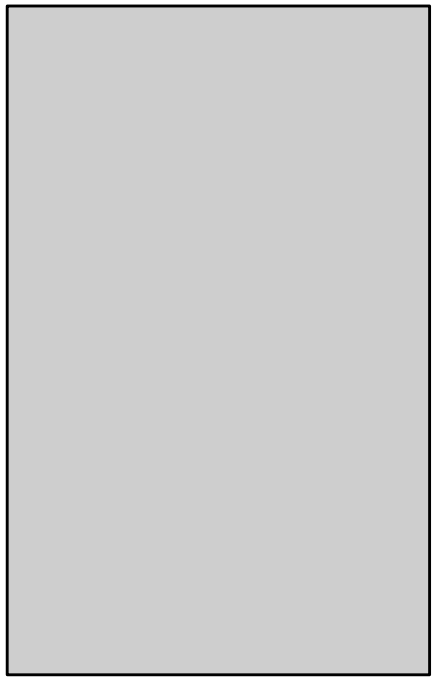
Storage Element: Register



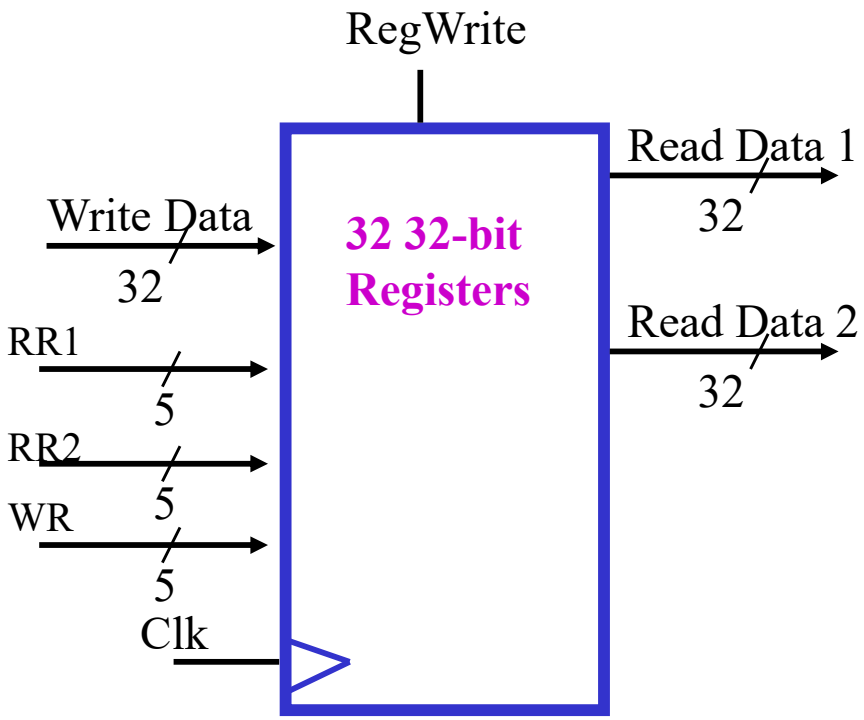
- Register
  - Similar to the D Flip Flop except
    - N-bit input and output
    - Write Enable input
  - Write Enable:
    - 0: Data Out will not change
    - 1: Data Out will become Data In (on the clock edge)

- But we need a whole bunch of registers
  - Register File

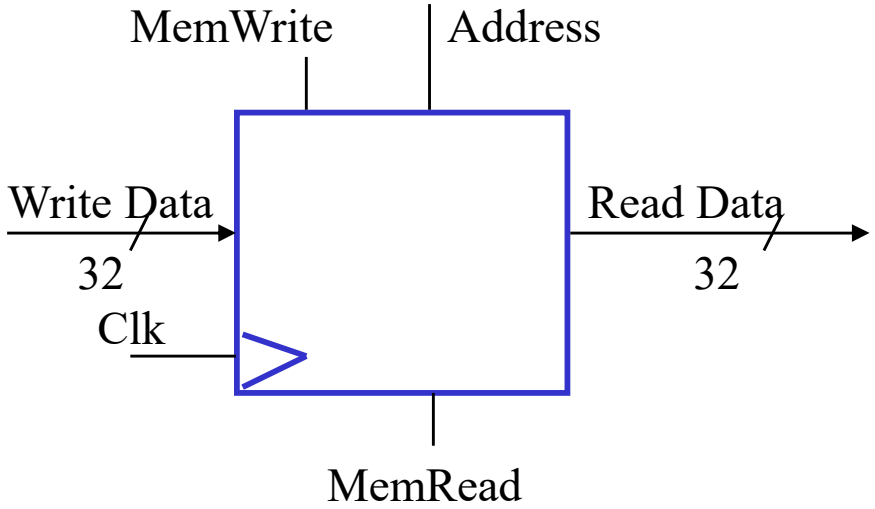
Reg File



Register File



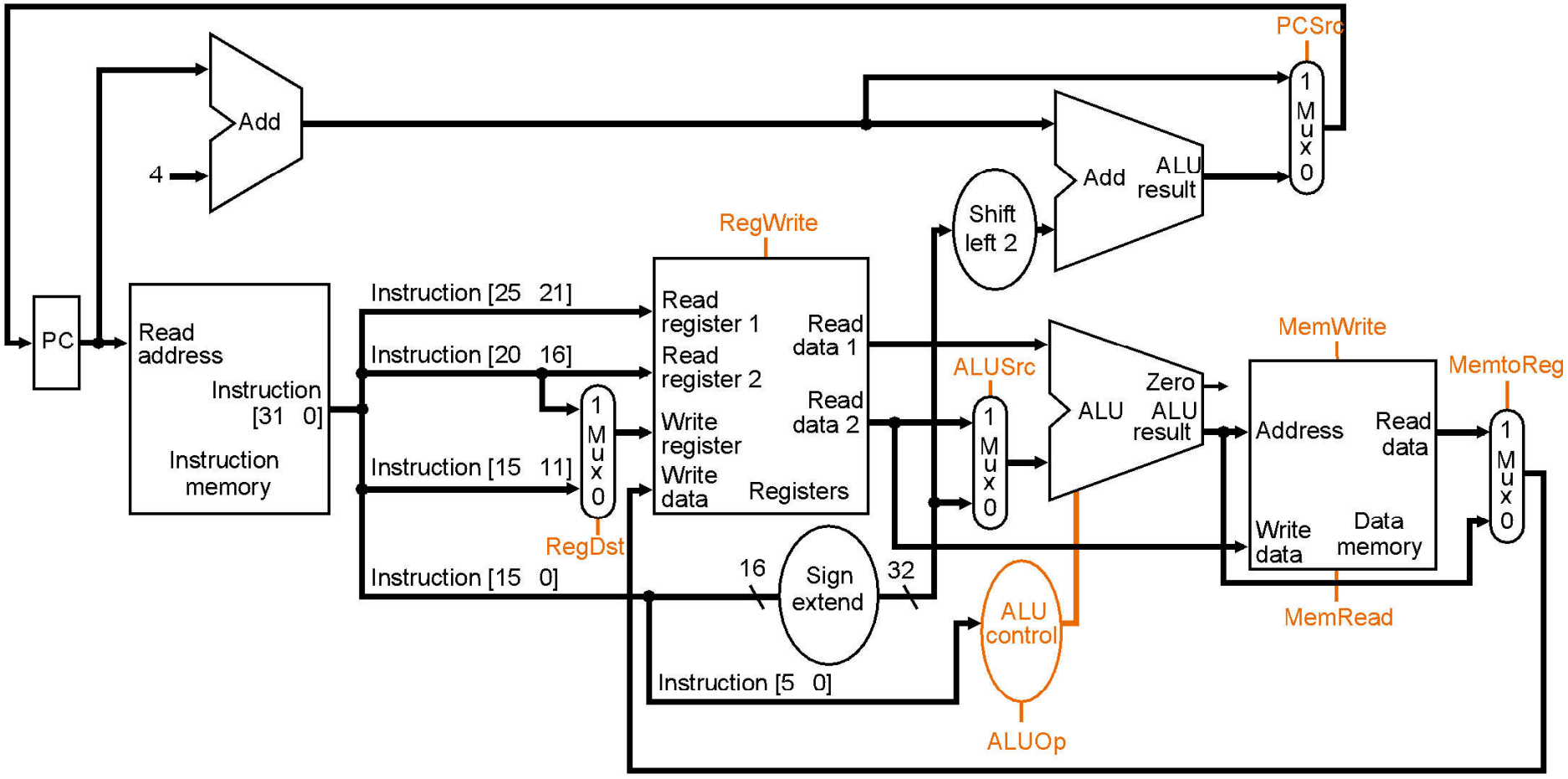
Memory



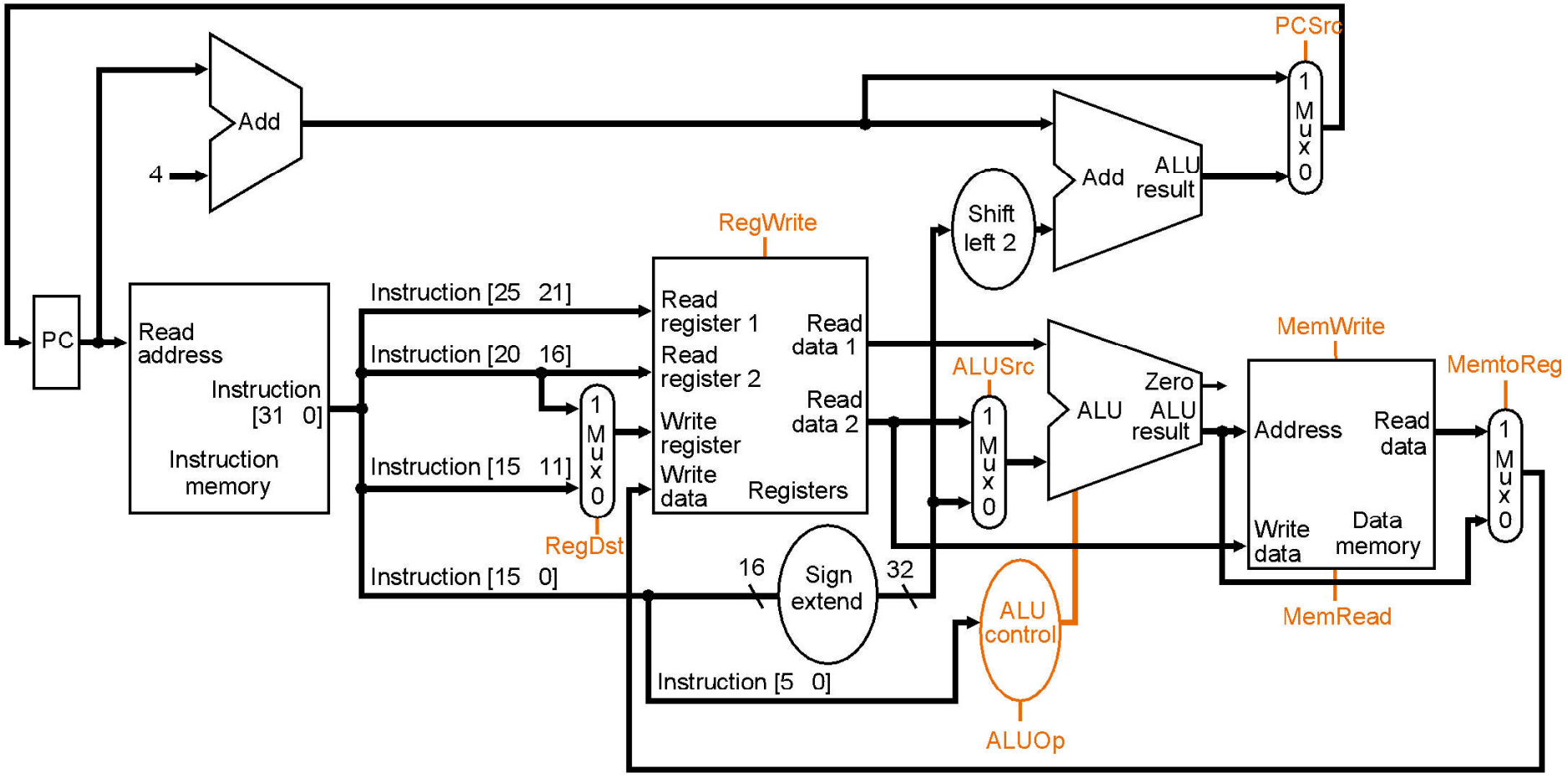
Can we layout a high-level design to do all this now?

Putting it All Together: A Single Cycle Datapath

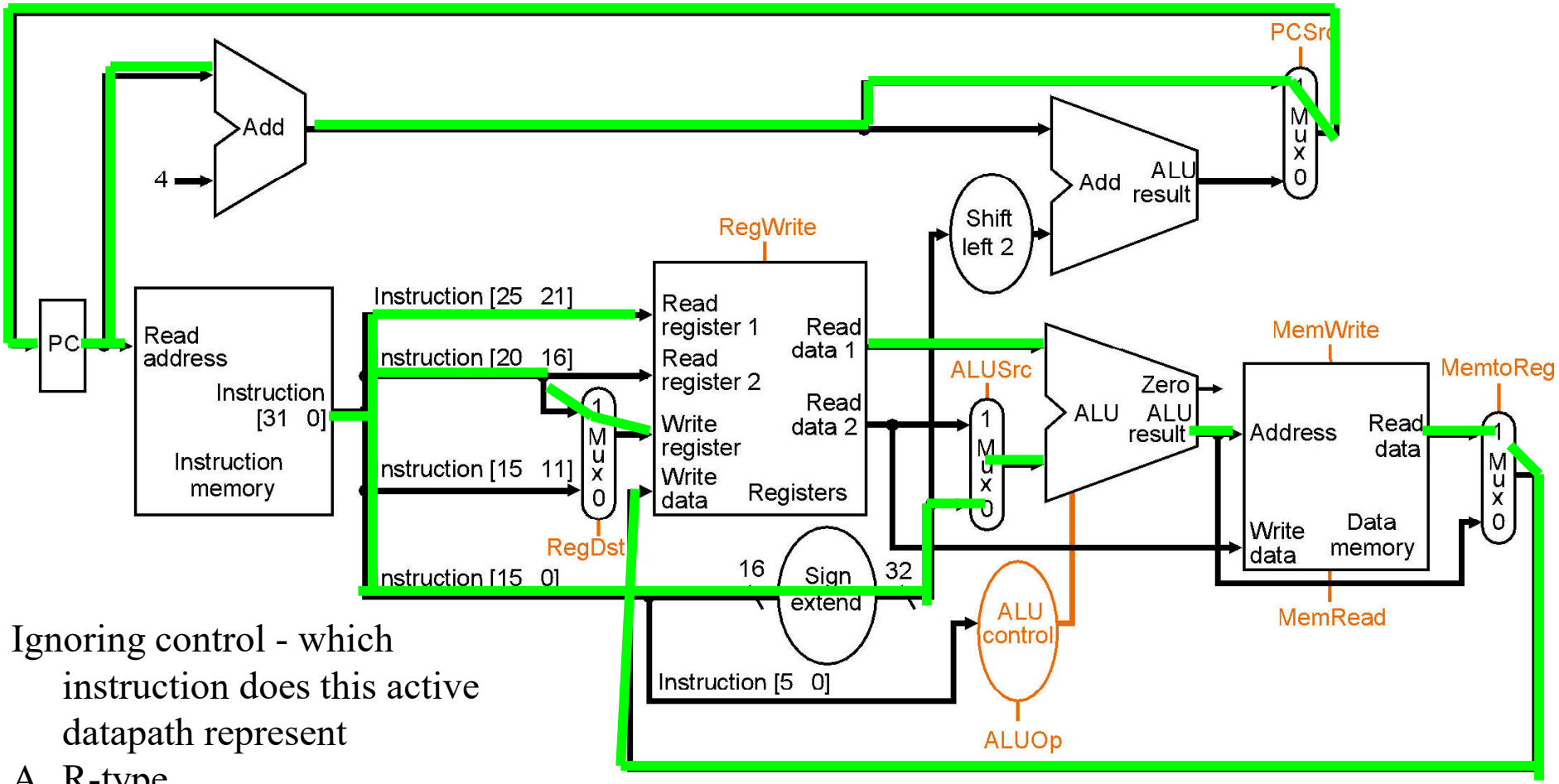
- We have everything except control signals (later)



How do we make this execute an R-type instruction?



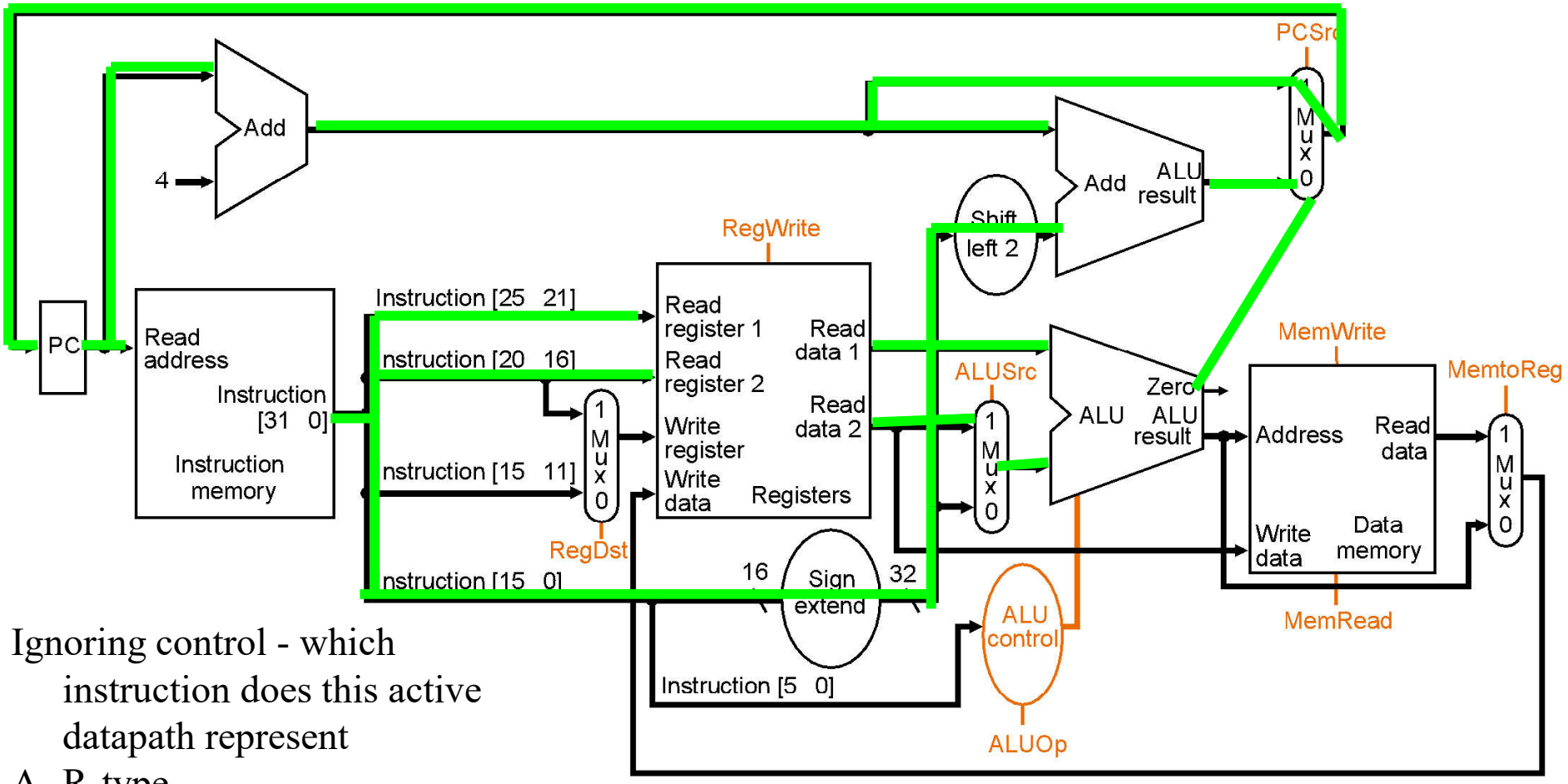
Active Single-Cycle Datapath



Ignoring control - which instruction does this active datapath represent

A. R-type
B. lw
C. sw
D. Beq
E. None of the above

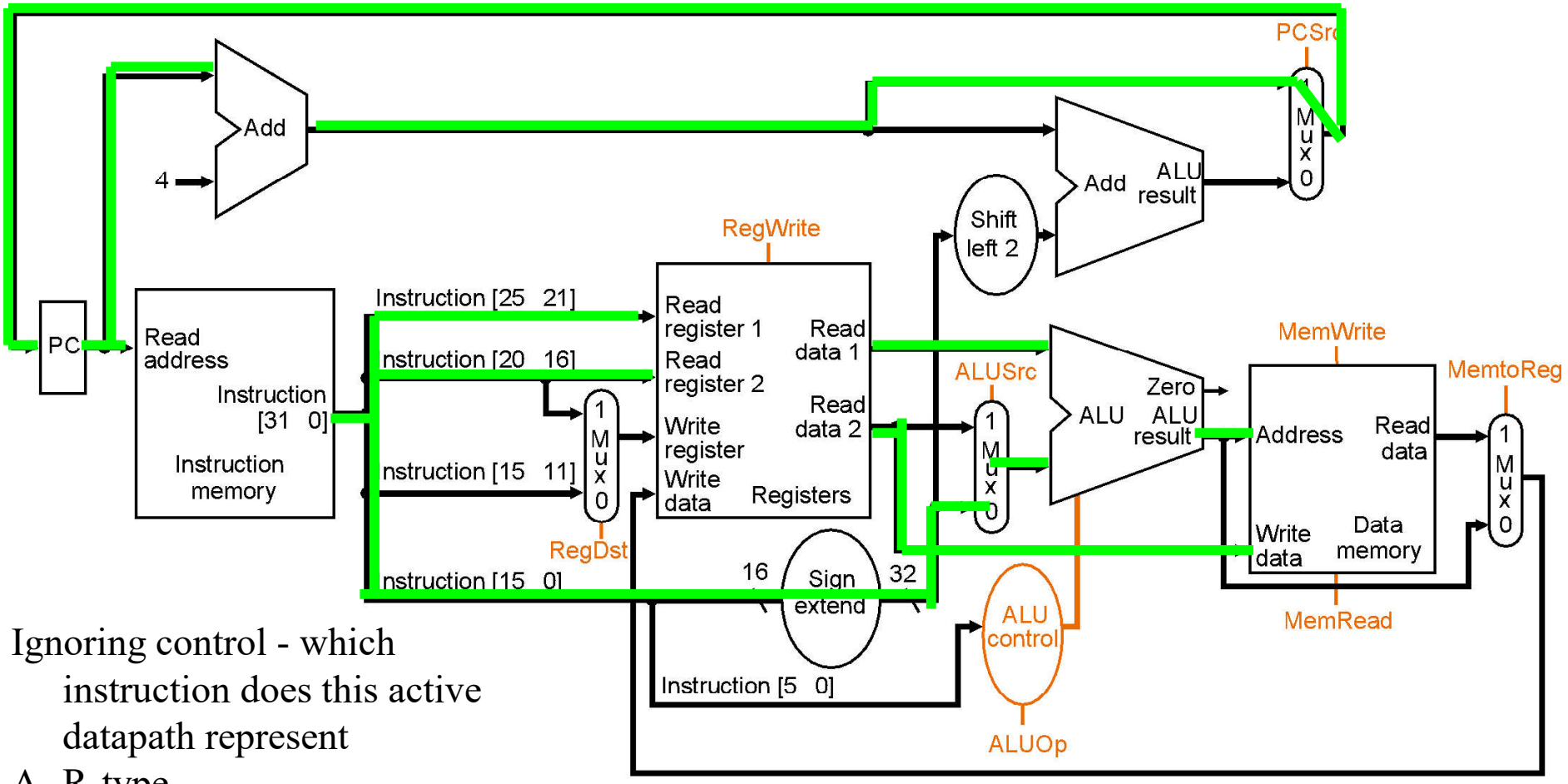
Active Single-Cycle Datapath



Ignoring control - which instruction does this active datapath represent

A. R-type
B. lw
C. sw
D. Beq
E. None of the above

Active Single-Cycle Datapath



Ignoring control - which instruction does this active datapath represent

A. R-type
B. lw
C. sw
D. Beq
E. None of the above

Key Points

- CPU is just a collection of state and combinational logic
- We just designed a the datapath for a very rich processor, at least in terms of functionality
- ET = IC * CPI * Cycle Time
  - where does the single-cycle machine fit in?