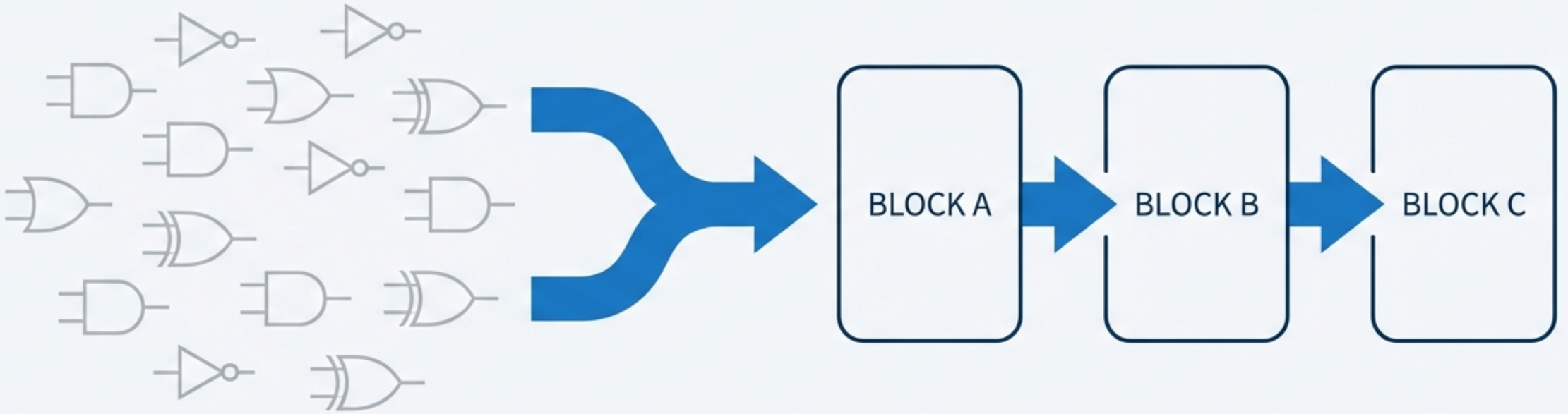
Describing Hardware: From Building with Gates to Defining the Flow of Data

An Introduction to Verilog Dataflow Modeling

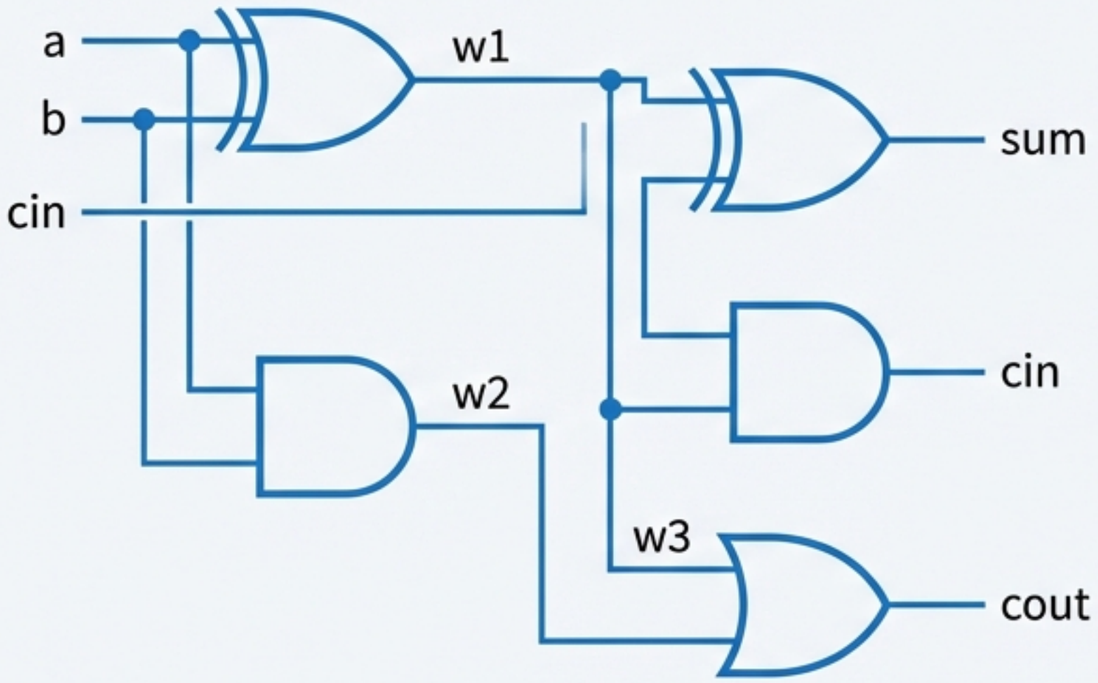


Building with individual parts.

Defining the connections.

Gate-Level Modeling is precise, but it doesn't scale for complex designs.

Gate-level modeling is the lowest level of abstraction. You can use this for small circuits, but when there is a complex circuit… it will be very difficult for you.




```
module Full_Adder_Gate (sum, cout, a, b, cin);
  output sum, cout;
  input a, b, cin;
  wire w1, w2, w3;

  xor g1(w1, a, b);
  and g2(w2, a, b);
  xor g3(sum, w1, cin);
  and g4(w3, w1, cin);
  or  g5(cout, w3, w2);
endmodule
```

Manually declaring every internal wire.

Instantiating each and every gate individually.

This approach is verbose and forces the designer to focus on gate-by-gate implementation rather than the circuit's overall behavior.

Dataflow Modeling provides a powerful way to describe how data moves.

Instead of defining the circuit by its gates, we define it by the continuous flow of data through Boolean expressions. It describes the circuit at a higher level of abstraction.


```
assign y = a & b;
```

The `assign` Statement

assign

Creates a “continuous assignment.” The left-hand side is *always* driven by the value of the right-hand side.

Think of it as a physical wire, not a one-time variable assignment. Any change on the right side immediately reflects on the left.

Let's revisit a simple circuit: The Half Adder.

Describing the Structure

```
// Gate-Level Half Adder
module half_adder_gate(sum, carry, a, b);
  output sum, carry;
  input a, b;

  xor g1(sum, a, b);
  and g2(carry, a, b);
endmodule
```

Describing the Behavior

```
// Dataflow Half Adder
module half_adder_dataflow(sum, carry, a, b);
  output sum, carry;
  input a, b;

  assign sum = a ^ b;
  assign carry = a & b;
endmodule
```

We replace gate instantiations with direct Boolean expressions.
The focus shifts from *what it's made of* to *what it does*.

The `assign` statement uses familiar operators to build expressions.

Key Rules

1. **Always Active**: Continuous assignments are always active throughout the simulation.
2. **Target Must Be a Net**: The left-hand side of an `assign` statement must be a `net` data type (e.g., `wire`). It cannot be a `reg`.
3. **Source Can Be Anything**: The right-hand side can be registers (`reg`), nets (`wire`), or constants.

Common Operators

Operator	Function	Example
&	Bitwise AND	`assign y = a & b;`
|	Bitwise OR	`assign y = a | b;`
^	Bitwise XOR	`assign y = a ^ b;`
~	Bitwise NOT	`assign y = ~a;`
{ }	Concatenation	`assign {cout, sum} = …`

A powerful feature for multi-bit outputs, like in an adder.

You can combine declaration and assignment in a single line.

This is called an “Implicit Continuous Assignment.” It’s a shorthand that reduces verbosity.

Regular Continuous Assignment (Two Lines)

```
wire out;                    // 1. Declare the net
assign out = in1 & in2; // 2. Assign to the net
```

Implicit Continuous Assignment (One Line)

```
wire out = in1 & in2; // Declare and assign simultaneously
```

Important Constraint: A net can only be declared once. Therefore, there can be only one implicit assignment per net.

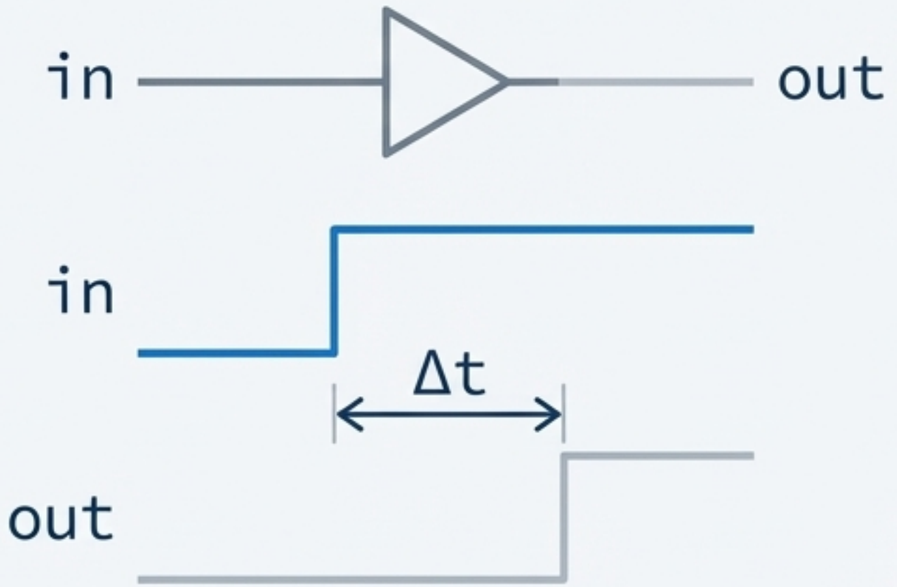
Real-world gates aren't instant. We can model this with delays.

Delay values control the time between a change in a right-hand side operand and when the new value is assigned to the left-hand side.

Modeling propagation delays is crucial for accurate simulation and timing analysis of a digital circuit.

Three Ways to Specify Delays:

1. **Regular Assignment Delay:** The most common method.
2. **Implicit Assignment Delay**: A shorthand for delays on declaration.
3. **Net Declaration Delay**: Applying a delay to the `wire` itself.



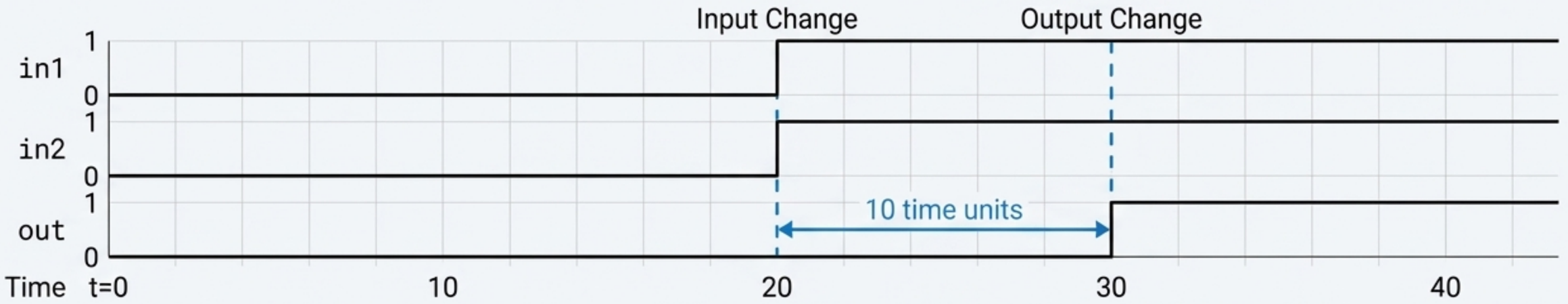
Regular Assignment Delay specifies a time lag after the `assign` keyword.

Syntax

```
assign #<delay_value> <net_LHS> = <expression_RHS>;
```

Example Code

```
// out will be updated 10 time units after
// in1 or in2 changes.
assign #10 out = in1 & in2;
```



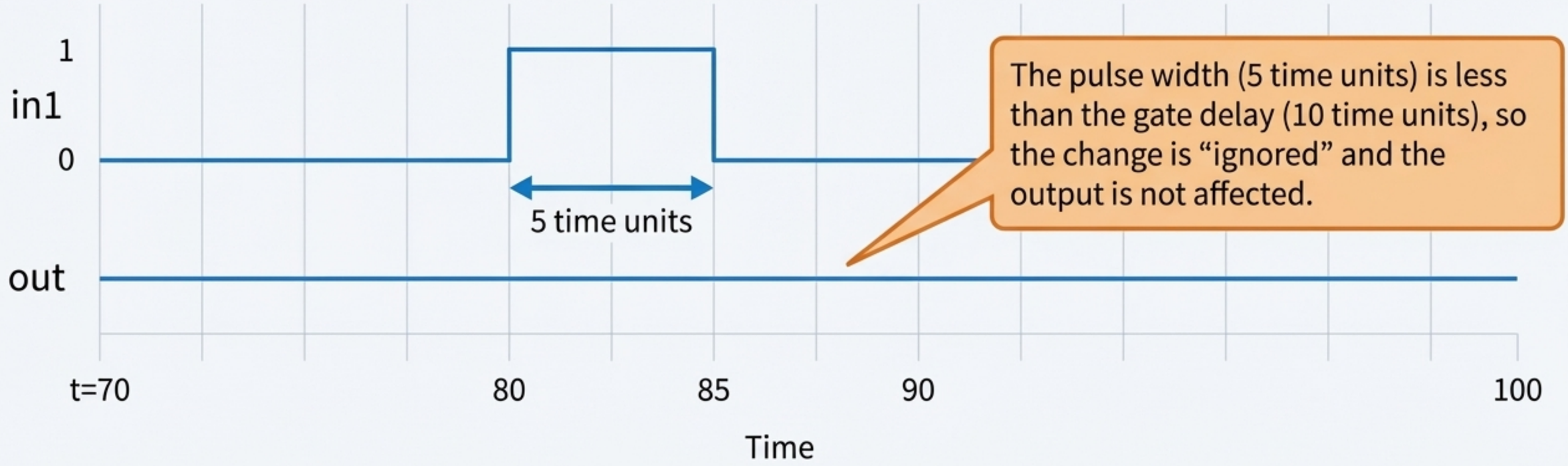
The expression `in1 & in2` is calculated as soon as the inputs change, but the result is not assigned to `out` until after the 10 time unit delay has passed.

Verilog delays have a crucial property: Inertial Delay.

An input pulse that is shorter than the specified delay of the assignment does not propagate to the output. This mimics the behavior of real physical gates which cannot react to changes that are too fast.

```
assign #10 out = in1 & in2; // comment deperty
```

assuming `in2` is high



Delays can also be specified during declaration for cleaner code.

Implicit Assignment Delay

Combines net declaration, delay specification, and continuous assignment into one statement.

```
// Declares wire 'out', specifies a 10-unit delay,
// and assigns the expression to it.
wire #10 out = in1 & in2;
```

Net Declaration Delay

Applies a delay to the net itself. Any assignment to this net will inherit this delay.

```
// Declares wire 'out' with an intrinsic 10-unit delay.
wire #10 out;

// This assignment will have a 10-unit delay.
assign out = in1 & in2;
```

These methods provide alternative styles but achieve similar results to regular assignment delays for modeling propagation time.

The advantage of Dataflow becomes clear as circuit complexity grows.

Let’s compare the implementation of a 1-bit Full Adder.

Gate-Level: 7 lines of logic

```
// Full Adder using gates
// (excluding module/port declarations)

wire s1, c1, c2;

xor g1(s1, a, b);
and g2(c1, a, b);
xor g3(sum, s1, cin);
and g4(c2, s1, cin);
or  g5(cout, c2, c1);
```

3 internal wires,
5 gate instantiations.

Dataflow: 2 lines of logic

```
// Full Adder using dataflow
// (excluding module/port declarations)

assign sum = a ^ b ^ cin;
assign cout = (a & b) | (b & cin) | (a & cin);
```

We see that the dataflow modeling approach is better...
it involves less lines of code and less willpower.

The ultimate payoff: Describing a 4-bit adder is remarkably simple.

How would you build a 4-bit full adder using Gate-Level modeling?

Verilog's arithmetic operators work seamlessly with multi-bit vectors in assign statements.

```
module full_adder_4bit (
    output [3:0] sum,
    output       cout,
    input  [3:0] a, b,
    input        cin
);

  assign {cout, sum} = a + b + cin;

endmodule
```

Verilog understands arithmetic operators.

Concatenation operator elegantly combines the carry-out and the sum vector.

This single line of code replaces what would require dozens of lines and multiple gate instantiations in a structural model. This is the power of a higher level of abstraction.

The same principles apply to other combinational circuits, like decoders.

A 2-to-4 decoder with an active-low enable signal (E).

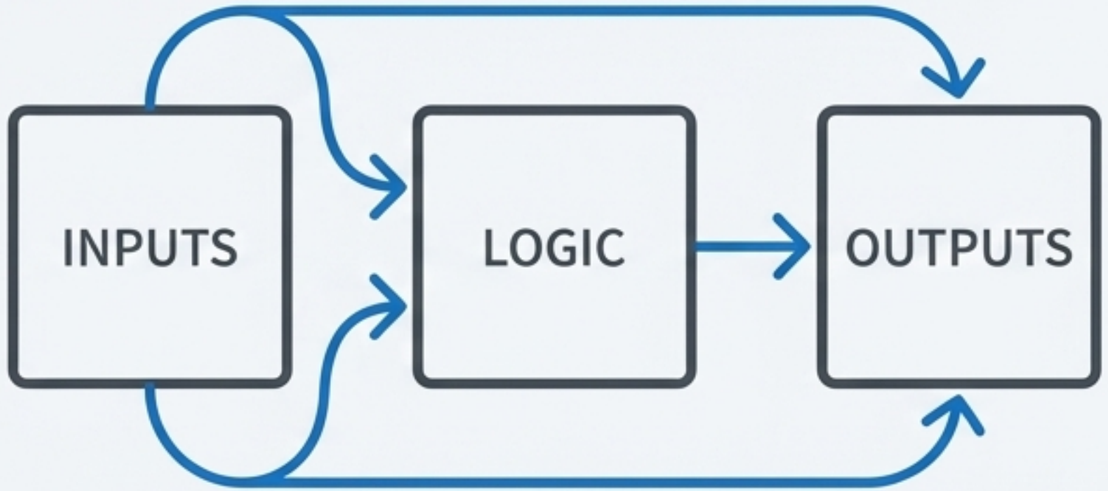
Inputs: A , B (2-bit address), E (enable, active low)
Outputs: D (4-bit output vector)

```
// A 2-to-4 decoder implementation using Dataflow Modeling.
// Two common styles are shown below.

// Method 1: Four separate assignments
// This is often clearer for distinct, independent logic.
assign D[0] = ~(~E | A | B);
assign D[1] = ~(~E | A | ~B);
assign D[2] = ~(~E | ~A | B);
assign D[3] = ~(~E | ~A | ~B);

// Method 2: Single assign with comma separation
// A more compact style for related assignments.
assign D[0] = ~(~E | A | B),
       D[1] = ~(~E | A | ~B),
       D[2] = ~(~E | ~A | B),
       D[3] = ~(~E | ~A | ~B);
```

Both methods are valid and concise. The choice is a matter of coding style. The key is that we are describing the logical function directly, not building it from gates.

Choose Dataflow Modeling when you want to focus on behavior, not structure.

- **Higher Abstraction:** Lets you describe the function of a circuit based on how data flows between registers.
- **Conciseness & Readability:** Results in significantly shorter, cleaner, and more understandable code, especially for complex circuits.
- **Scalability:** Simple expressions can be scaled to multi-bit vectors with ease, making it ideal for larger designs.



While Gate-Level modeling gives you fine-grained control over the exact structure, Dataflow Modeling offers a more powerful and efficient path for designing and implementing complex digital logic. It becomes indispensable as your designs grow.