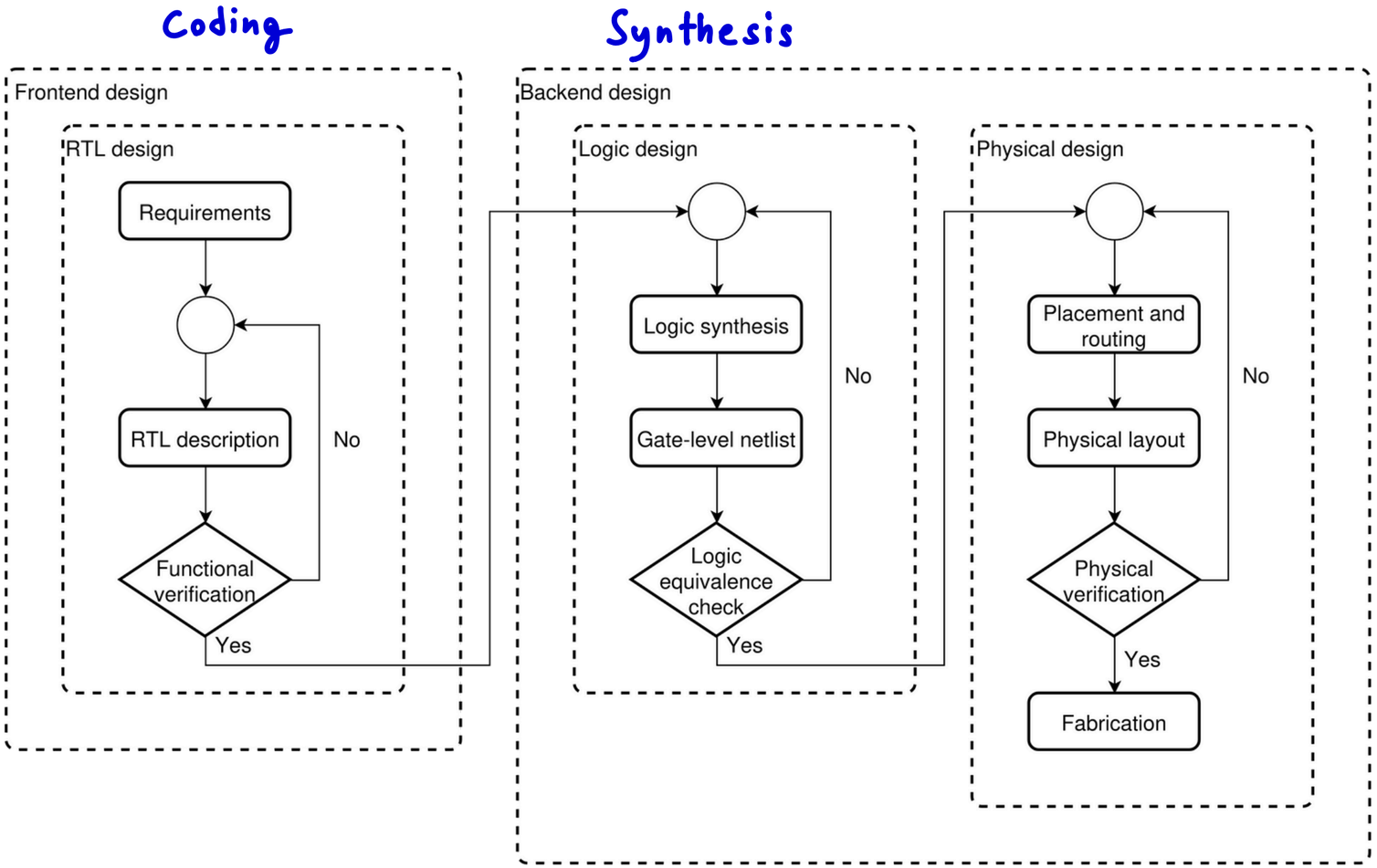
Lecture 4
Hardware Discription Language with Verilog

Design Flow



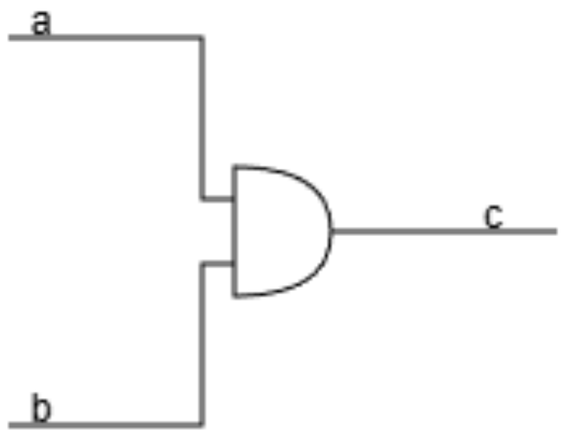
This lecture covers the RTL design part.

Verilog mindset

C, Python, and more	Verilog
Write instructions for the CPU to execute step by step.	Describe physical components and how they are wired together.

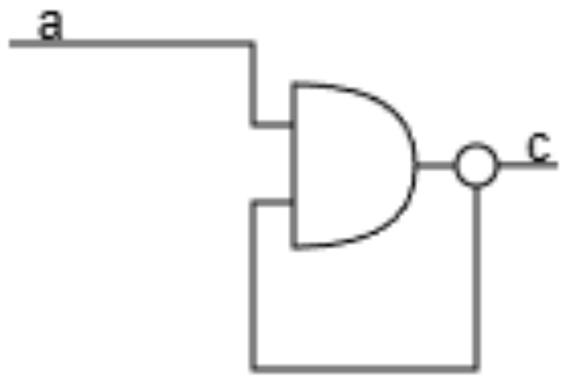
Combinational Circuits

- Output depends only on the current input.
- No memory.
- If input changes, output updates immediately.
- Known examples of this type of circuit are the **adder** and the **multiplexer**.



Sequential Circuits

- Output depends on both the current input and the stored past value.
- Has memory (at least conceptually).
- Known examples of this type of circuit are the **latch** and the **flip-flop**.



Variable

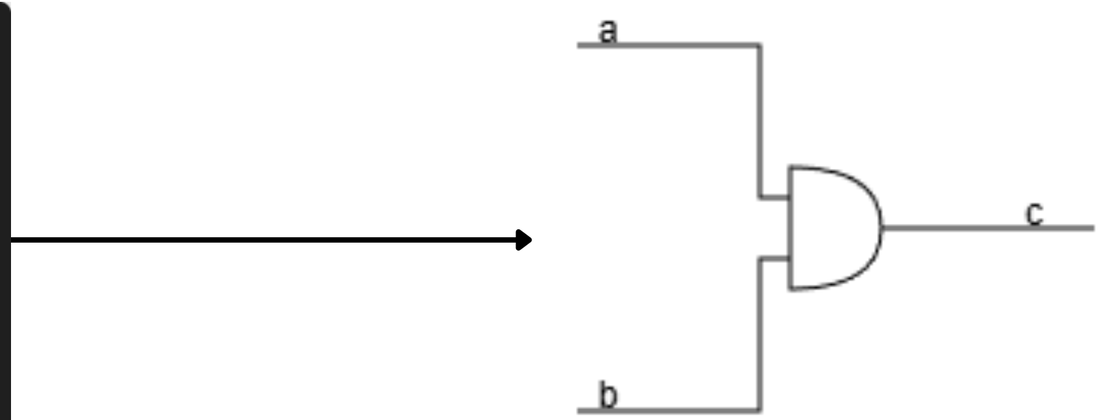
Wire

Syntax: wire *name_of_variable*;

- A wire represents the **connection** between the hardware components, such as logic gates.
- It's typically visualized by a physical wire on a circuit.

Example Code

```
module wire_example;
    wire a;
    wire b;
    wire c;

    assign c = a & b;
endmodule
```



Variable

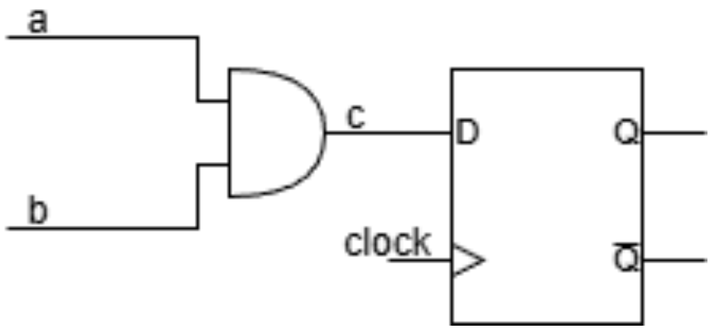
Register (reg)

Syntax: reg *name_of_variable*;

- Register or reg is an **abstract** data storage unit capable of holding a value.
- The physical unit that would be inferred is **decided by Synthesizer** (Compiler equivalent of Verilog).
- Typically, the inferred data storage unit is a flip-flop.

Example

```
module reg_example;
    wire a;
    wire b;
    reg c;

    always @(posedge clk) begin
        c <= a & b;
    end
endmodule
```



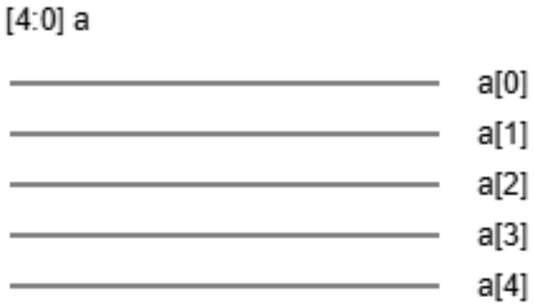
Variable

Grouped Wire or Register

(wire | reg) [MSB index:LSB index] *name_of_variable*;

- When dealing with a multi-bit system, instead of declaring a wire or register for each bit, we can create variables with certain bit widths using the given syntax.
- We can visualize a wire variable with a specified bit width as a group of wires, while for a reg variable, it is closer to an array of 1-bit values.

```
module grouped_wire;
    wire [4:0] a;
endmodule
```




```
module grouped_reg;
    reg [4:0] a;
endmodule
```

Assigning Value and Value Representation

- With a 1-bit variable, you can assign the value with the following syntax.
  - Syntax: *variable_name* (= | <=) (1 | 0);
- For a multi-bit variable, you can assign the value by representing the value in different number bases.
  - Syntax: *variable_name* (= | <=) (number of bits)'(s)(base)(value);

Example

```
module value_representation;
    reg [3:0] a;
    reg [3:0] b;
    reg [3:0] c;

    initial begin
        a = 4'sb1001;
        b = 4'd9;
        c = 4'hA;
    end
endmodule
```

Supported operations

Arithmetic	+, -, *, /, %, **
Relational	<, >, <=, >=
Equality	==, !=
Logical	&&, ||, !
Bitwise	&, |, ^, ~
Shift	<<, >>, Arithmetic shift (<<<, >>>)

*The power sign only works if the exponent is a constant.

Signedness

Syntax: (wire | reg) (signed) [MSB index:LSB index] *variable_name*;

- If the variable is declared without a “signed”, the synthesizer will treat the variable as unsigned by default.
- When using an operation, one thing that must be considered is the signedness of the variable.
- Some of the operations may have different processes when dealing with signed and unsigned values.
- For example, the arithmetic right shift operator (>>>):
  - Given a = 4'b1101, the unsigned decimal value is 13, while the signed value is -3.
  - a >>> 1; Without considering the signedness, this gives the value 0110, which is equal to 6 both unsigned and signed.
  - a >>> 1; Considering the signedness, this gives us a value of 1110, which is equal to -2 for signed and 14 for unsigned. In conclusion, shifting will try to preserve the signed bit.

Example of Declaring Signed Variable

```
module add_example (
    input wire clk,
    input wire [3:0] x,
    input wire signed [3:0] y,
    output reg [3:0] sum
);

always @(posedge clk) begin
    sum <= x + y;
end
endmodule
```

Module Declaration

```
module example_and_module
```

Module name:
Syntax: module *name_of_variable*;

```
(
    input wire clk,
    input wire a,
    input wire b,
    output reg c
);
```

Module's port:

- Input syntax: input wire *input_variable_name*;
- Output syntax: output (wire | register) *output_variable_name*;
- Input variables must be wires as it represents the connection from the outside module.

```
reg [4:0] dummy_register;
wire dummy_wire;
```

Internal variable

```
always @(posedge clk) begin
    c <= a & b;
end
```

Circuit Description

```
endmodule
```

End of Module

Module Instantiaion

```
example_and_module first_instance
```

Module instance:

- Syntax: *module_name instance_name*

```
(
    .clk(clk),
    .a(in_x),
    .b(in_y),
    .c(c)
);
```

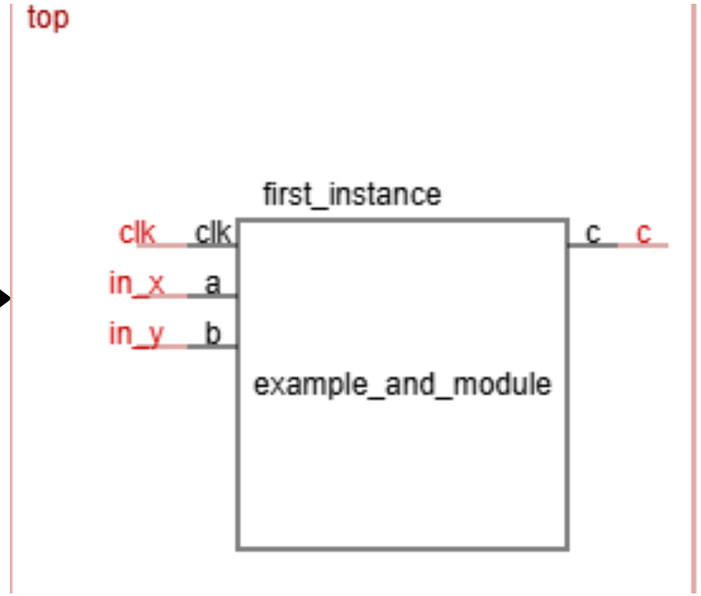
Module connections:

- Syntax: *.port_name*(*variable_name*)
- Input port: The variable can be either a wire or a register.
- Output port: The variable must be a wire, as it represents the connection between the top module and the output variable of the module.

Visualize Module Instantiaion

```
module top;
    wire clk;
    wire in_x;
    wire in_y;
    wire c;

    example_and_module first_instance (
        .clk(clk),
        .a(in_x),
        .b(in_y),
        .c(c)
    );
endmodule
```

```
module example_and_module (
    input wire clk,
    input wire a,
    input wire b,
    output reg c
);

reg [4:0] dummy_register;
wire dummy_wire;

always @(posedge clk) begin
    c <= a & b;
end
endmodule
```



Circuit Modeling

There are 3 main ways to model your circuit

- Structural modeling or gate-level modeling
- Dataflow modeling
- Behavioral modeling

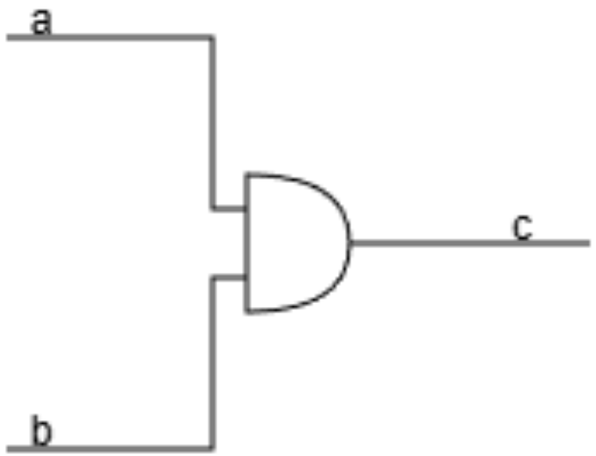
Circuit Modeling

Structural Modeling or Gate-Level Modeling

- This modeling method models the circuit by explicitly instantiating the gate and manually wiring each one of them.
- This is the closest modeling method to actual hardware.
- This method is susceptible to optimization by Synthesizer.
- Syntax: (and | or | xor | nand | nor | xnor | not) *instance_name*(output wire, input wire, input wire);

Example Code

```
module example_gate_level (
    input wire a,
    input wire b,
    output wire c
);
    and (c, a, b)
endmodule
```



Circuit Modeling

Dataflow Modeling

- Instead of instantiating gates individually, you can use a Boolean expression to describe how your circuit works.
- Typically, this method is only used when modeling combinational circuits and wouldn't be used to model a sequential circuit.
- The variables that are being assigned (on the left-hand side of "=") must be a wire.
- Dataflow also allows you describe a circuit as a conditional statement.

Example Code for 4-to-1 Multiplexer Using Expression

```
module mux4_1 (
    input wire i0, i1, i2, i3,
    input wire [1:0] s,
    output wire out
);

    assign out = (~s[0] & ~s[1] & i0) |
                 (~s[0] & s[1] & i1) |
                 (s[0] & ~s[1] & i2) |
                 (s[0] & s[1] & i3);
endmodule
```

Example Code for 4-to-1 Multiplexer Using Conditional Statement

```
module mux4_1 (
    input wire i0, i1, i2, i3,
    input wire [1:0] s,
    output wire out
);

    assign out = s[1] ? (s[0] ? i3 : i2) : (s[0] ? i1 : i0);
endmodule
```

Circuit Modeling

Behavioral Modeling

- Behavioral modeling allows you to describe the behavior of your circuit using always blocks and conditional statements, such as if-else or case statements.
- This method is much more common in practice as it is the most intuitive way to describe your circuit.
- Behavioral modeling allows you describe both combinational circuits and sequential circuits.

Example Code for 4-to-1 Multiplexer Using Behavioral Modeling

```
module mux4_1 (
    input wire i0, i1, i2, i3,
    input wire [1:0] s,
    output reg out
);

    always @(*) begin
        case (s)
            2'b00: out = i0;
            2'b01: out = i1;
            2'b10: out = i2;
            2'b11: out = i3;
            default: out = 1'b0;
        endcase
    end
endmodule
```

Circuit Modeling

Always Block

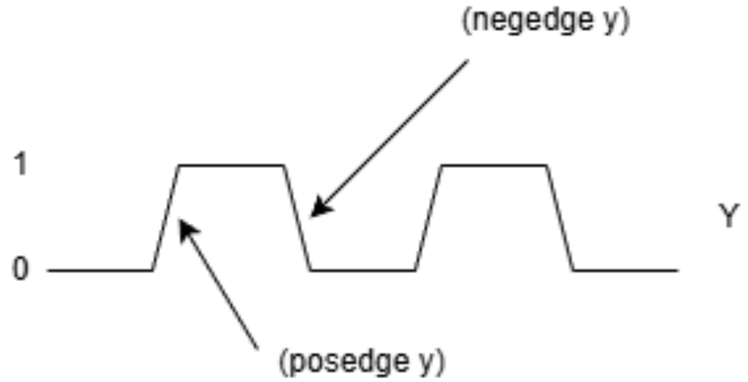
Always block syntax

```
always @ (event)
 [statement]

or

always @(event) begin
 [multiple-line statement]
end
```

- If the (event) occurs, the block will execute the statement(s).
- Within a block, only registers can be assigned (on the left-hand side of “=” or “<=”).

Circuit Modeling

Always Block (Continue)



- The (event) refers to a specific event from the specified variable.
  - Given **y** is a 1-bit wire or register, the state overtime shown in the image above.
  - The (posedge y) refers to the event where a wire or register **y** transitions from 0 to 1 or a rising edge.
  - The (negedge y) refers to the event where a wire or register **y** transitions from 1 to 0 or a falling edge.
  - The (y) refers to any transition event, including both positive edge and negative edge.
  - The (event) can have multiple events within the block. For example, **(y or x)**, which means it will detect both event (y) or event (x).
  - The (*) means every change of every wire or register within a block will be considered as an event.

Circuit Modeling

Blocking Assignment (=)

- When using blocking assignment “=” within an always block, the assignment will attempt to execute in **order**.
- For example, let the initial values of a, b, and c be 2, along with the following assignment statements:
  - b = a;
  - c = b;
  - Suddenly, if a changes from 2 to 5 (a = 5), then the value of c and b will be 5 (c = a).

Non-Blocking Assignment (<=)

- When using non-blocking assignment “<=” within an always block, the assignment will attempt to execute in parallel by **evaluating the value of every statement and assigning the value at the same time.**
- This assignment can be used only inside the always block or the initial block.
- For example, let the initial values of a, b, and c be 2, along with the following assignment statements:
  - b <= a;
  - c <= b;
  - Suddenly, if a changes from 2 to 5 (a = 5), then the value of c will be 2, while the value of b will be 5 (c = b before b got assigned).

Circuit Modeling

Example of Modeling a Combinational Circuit with Behavioral Modeling

Following the concept of a combinational circuit:

- Output depends only on the current input (no clock, no memory).
- If input changes, output updates immediately.

This means you can describe a circuit using this method by:

- Let (event) be (*), since a combinational circuit is sensitive to every input change.
- Ensure every branch of the conditional statement(s) assigns the output (no holding on to the previous value).

Example Code for 4-to-1 Multiplexer Using Behavioral Modeling

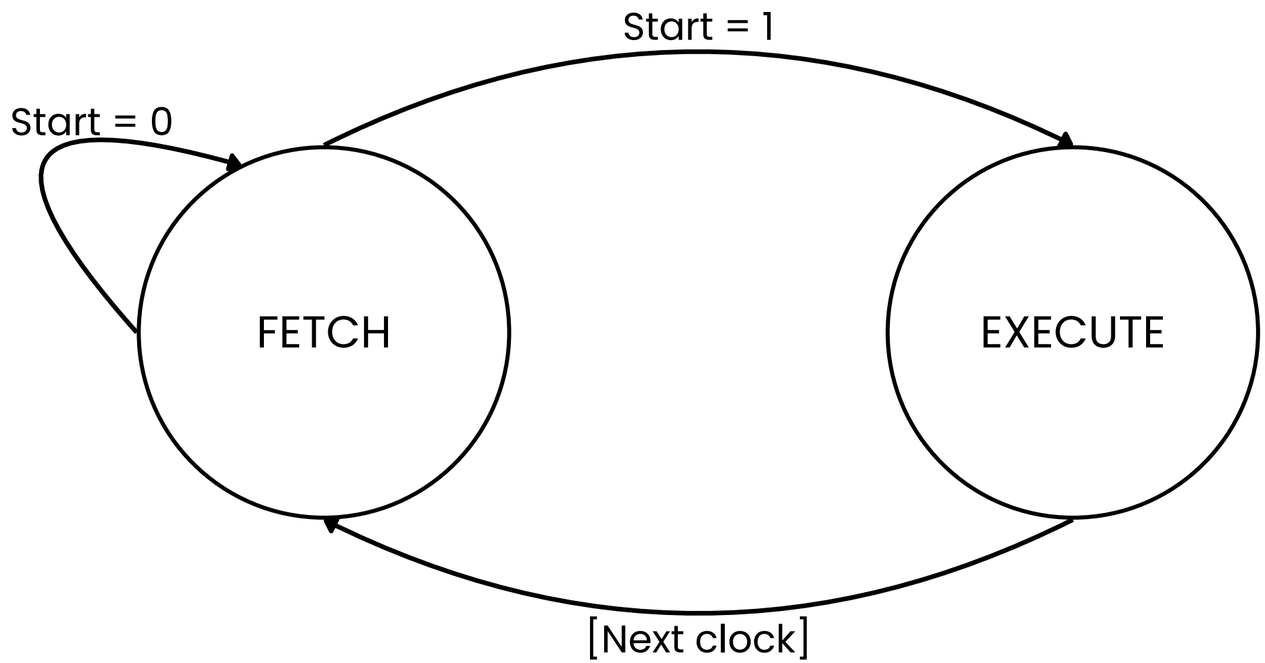
```
module mux4_1 (
    input wire i0, i1, i2, i3,
    input wire [1:0] s,
    output reg out
);

    always @(*) begin
        case (s)
            2'b00: out = i0;
            2'b01: out = i1;
            2'b10: out = i2;
            2'b11: out = i3;
            default: out = 1'b0;
        endcase
    end
endmodule
```

Note: Although the output is a register, Synthesizer would still treat it as a wire, since this will be interpreted as a combinational circuit.

Circuit Modeling

Example of Modeling a Sequential Circuit with Behavioral Modeling

Let’s try to modeling circuit for the following finite state machine (FSM).



- State changes only occur when it is the positive edge of the clock (synchronous).
- Start may be triggered (start = 1) at any time and does not depend on the clock.
- Another input will be a 2-bit value that will indicate which operation will be executed.
- The module must show whether it is ready to receive input or not (no if at EXECUTE).

Finite State Machine

Define Module and Output Logic

```
module cpu_fsm (
    input wire clk,
    input wire rst,
    input wire start,
    input wire [1:0] op,
    output reg [1:0] sel,
    output reg busy
);

localparam  FETCH = 1'b0,
            EXECUTE = 1'b1;

reg state, next_state;
```

- Here, we receive the following inputs:
  - clk: Clock.
  - rst: Reset signal.
  - start: Triggers the execution.
  - op: Operation that will be executed.
- Outputs:
  - busy: The status signal that indicates whether or not the module is ready to receive another input.
  - sel: The operation that is being executed.

What type of circuit would this synthesize to?

```
always @(*) begin
    sel = 2'b00;
    busy = 1'b0;

    case (state)
        FETCH: begin
            busy = 1'b0;
        end
        EXECUTE: begin
            busy = 1'b1;
            sel = op;
        end
    endcase
end
endmodule
```

- Here, we define output logic by:
  - At the start:
    - Set both busy and sel to 0.
  - If the current state is FETCH:
    - Set busy to 0 (ready).
  - If the current state is EXECUTE.
    - Set busy to 1 (not ready).
    - Set sel to op.

Finite State Machine

Handling Input and State Changes

```
always @(posedge clk) begin
    if (rst) begin
        state <= FETCH;
    end else begin
        state <= next_state;
    end
end
```

- As stated previously, the state only changes at the positive edge of the clock, so here, we define **(event)** to **(posedge clk)**.
- When (posedge clk) occurs, if there is no signal sent to the module to reset the state, then the state would be set to **next_state**.
- This is a **synchronous sequential circuit**, as the **state is the memory** that holds a value, and the **changes depend on the clock**.

```
always @(*) begin
    case (state)
        FETCH:
            if (start) begin
                next_state = EXECUTE;
            end
        EXECUTE: next_state = FETCH;
    endcase
end
```

- Since the start signal can be triggered at any time, we create the logic that would **store the supposed next state value** in the next_state register until the next positive edge of the clock.
- During the FETCH state, if start = 1, then we assign next_state = EXECUTE. Otherwise, the **next_state would hold the value** that it held previously.
- For EXECUTE, the next_state will be set to FETCH.
- This is an **asynchronous sequential circuit**, as **the next_state is the memory**, but **it doesn't depend on the clock**.

Full Code

```
module cpu_fsm (
    input wire clk,
    input wire rst,
    input wire start,
    input wire [1:0] op,
    output reg [1:0] sel,
    output reg busy
);

localparam  FETCH = 1'b0,
            EXECUTE = 1'b1;

reg state, next_state;

always @(posedge clk) begin
    if (rst) begin
        state <= FETCH;
    end else begin
        state <= next_state;
    end
end

always @(*) begin
    case (state)
        FETCH:
            if (start) begin
                next_state = EXECUTE;
            end
        EXECUTE: next_state = FETCH;
    endcase
end
```

```
always @(*) begin
    sel = 2'b00;
    busy = 1'b0;

    case (state)
        FETCH: begin
            busy = 1'b0;
        end
        EXECUTE: begin
            busy = 1'b1;
            sel = op;
        end
    endcase
end
endmodule
```

Unit Testing with Testbench

Initialization

```
`timescale 1ns/1ps
```

Define the timescale.

```
module cpu_fsm_tb;
    reg clk;
    reg rst;
    reg start;
    reg [1:0] op;
    wire [1:0] sel;
    wire busy;
```

Define the simulation variables.

- The input variables should be registers, as the testbench must control the value to simulate the real input.
- The output variables should be wires, as the output should only be observed and not altered.

```
cpu_fsm dut (
    .clk(clk),
    .rst(rst),
    .start(start),
    .op(op),
    .sel(sel),
    .busy(busy)
);
```

Instantiate the device under test and connect the simulation variables.

```
always #5 clk = ~clk;
```

Create the clock signal.

Unit Testing with Testbench

Define the Test Cases

```
    initial begin
        clk = 0;
        rst = 1;
        start = 0;
        op = 2'b00;
        #20;
        rst = 0;
        #10;
        op = 2'b10;
        start = 1;
        #10;
        start = 0;
        #60;
        op = 2'b01;
        start = 1;
        #10;
        start = 0;
        #60;
        $finish;
    end
endmodule
```

Set the simulation input value at time 0 ns.

Delay. Syntax: #number_of_nanoseconds

Set the simulation input at time 20 ns.

.
.
.

- Here you can simulate the input value and see the behavior of the module.
- Simulation will be described within an initial block, and the process itself will alternate between delaying to the specific time and setting the simulation value at that time.

Edge Case

```
module add_example (
    input wire clk,
    input wire [3:0] x,
    input wire [3:0] y,
    output reg [3:0] sum
);

always @(posedge clk) begin
    sum <= x + y;
end
endmodule
```

What would the sum become if x and y are 7 and 9, respectively?

SAP-1

What it is

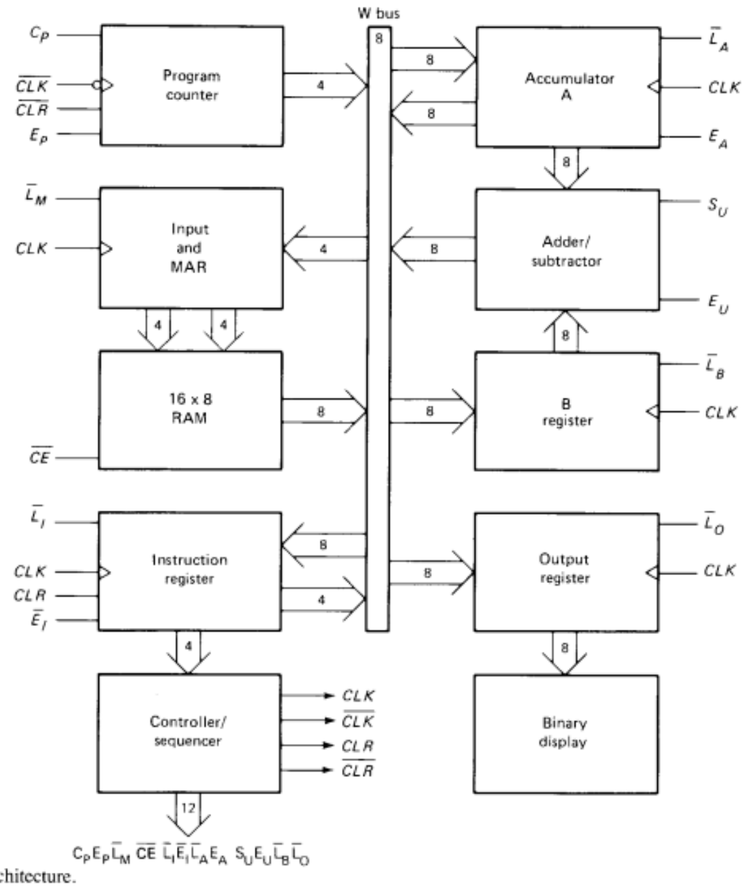


Fig. 10-1 SAP-1 architecture.

SAP-1, or simple-as-possible-1, is a simplified computer architecture designed for educational purposes.

SAP-1

Module breakdown

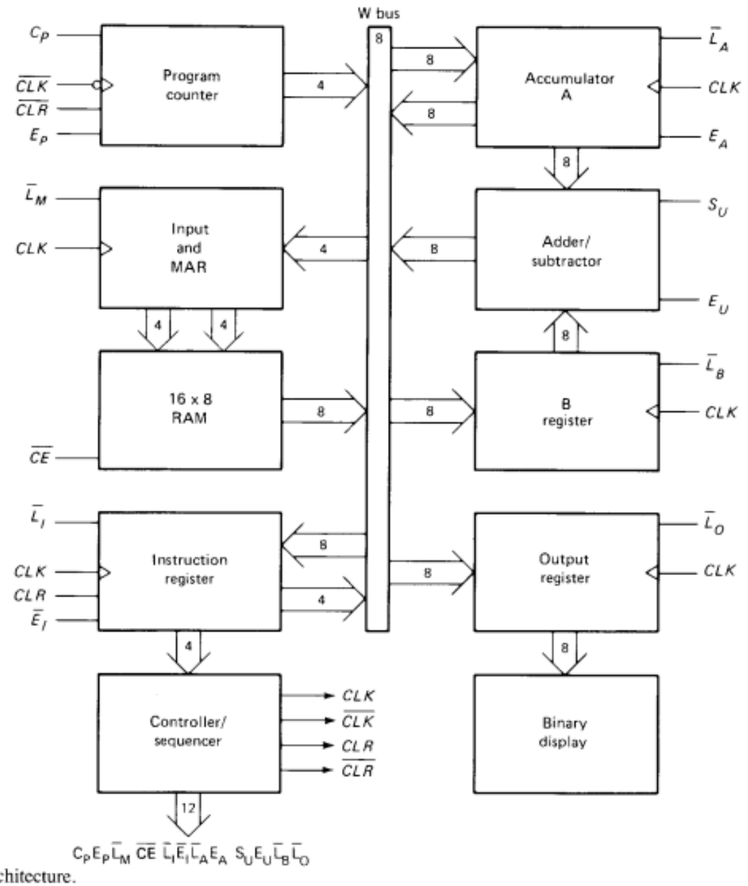


Fig. 10-1 SAP-1 architecture.

- Sequential
  - Program counter
  - Input register
  - MAR
  - Instruction register
  - Accumulator
  - B register
  - Output register
  - Control unit (FSM)

- Combinational
  - ALU
  - RAM
  - Binary display
  - Bus (multiplexer)