

Abstraction

Aspects of a component that might be used in a larger design:

- How do you use it? What is it supposed to do?
behavior (or interface)
- How does it work? What's inside the box?
implementation

Abstraction makes it possible for humans to design and build complex systems

- package together low-level pieces
- use the package as a bigger building block in a more complex design
- to use in larger design, only need to know a component's behavior / interface (not its implementation)
- similar to modular programming

We use abstraction all the time. E.g. taking notes

- lowest-level of abstraction: picking up pen, moving it across paper
- middle-level of abstraction: call certain shapes "letters". Now writing means to string letters together.
- higher-level of abstraction: call combinations of letters "words". Now writing means to string words together.

Always possible to think at lower levels of abstraction, or at higher levels of abstraction

Levels of abstraction in a computer system

Digital logic systems are really complex

- Intel Core i7 processor has 731 million transistors

Levels of abstraction

- applications and operating system (high-level language, e.g. C++)
- applications and operating system (machine-level language)
----- hardware/software boundary
- computer processor
- datapath and control
- combinational and sequential logic
- logic gates
- transistors
- solid-state physics

We'll start at a fairly low-level of abstraction (combinational and sequential logic) and build up

- first goal: given a specific algorithm, be able to build a circuit that implements that algorithm
- second goal: build a single circuit that can run **any** algorithm (i.e. a general-purpose computer)

What can you do with bits?

Group bits together into an array of bits

- how many different values can be represented with 4 bits?

- word = array of bits of default size (e.g., 4 bits)

Communicate bits

- a wire is used to communicate one bit between components
- we say that the wire is communicating a “signal”

Store bits and recall them later

- store bits in “registers”

Combinational logic

Need more than communication and storage to do anything interesting with bits

Compute output bit(s) based on input bit(s)

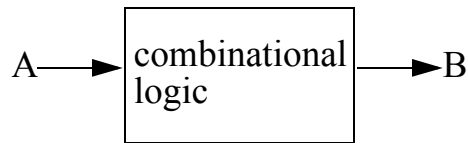
E.g. Alice and Bob going to a party

- two wires: A, B
- A is 1 if Alice is going to the party; 0 otherwise
- B is 1 if Bob should go to the party; 0 otherwise
- build a circuit that computes whether Bob should go to the party or not, depending on what Alice is doing

Bob goes if (and only if) Alice is going:

Bob goes if Alice is NOT going:

Schematic representation



B's value is computed based on A's value

- in mathematical terms, B is a function of A, or $B=f(A)$
- if A's value changes, B's value will change a little later (propagation delay)

Combinational logic: outputs depend only on the current combination of inputs.

- we say that a logic block “drives” the output

What can you say about a block of logic?

- express how it behaves (behavior)
- describe how it's implemented (implementation)
- which of these do we need to use a block of logic as a building block in a larger circuit?

Hardware description languages

How can you describe the desired behavior of an arbitrary combinational logic block?

E.g.

A	B
0	0
1	2
2	4
3	6

E.g.

A	B
2	1
3	1
4	0
5	1
6	0
7	1
8	0
9	0
10	0
11	1
12	0
13	1
14	0
15	0

Verilog

Verilog is a Hardware Description Language (HDL)

- used to describe the desired hardware behavior
- an automated tool knows how to translate this description into an implementation (synthesis)

Similar syntax to C

To express numbers, specify the number of bits and the base

- 1'b0: 1 bit, base binary, value 0
- 1'b1: 1 bit, base binary, value 1

Instead of {}, use begin/end

```
module NOT(  
    input wire A,  
    output reg B);  
  
    always @* begin  
        if (A == 1'b0) begin  
            B = 1'b1;  
        end else begin  
            B = 1'b0;  
        end  
    end  
end  
endmodule
```

Note the code inside the “always @*” block

- this is a program that computes a value of B (for a given value of A)

To translate to a truth table:

- write the possible values of A
- for each possible value, run through the code inside the “always @*” block and use the ending value of B

B should get some value assigned to it by the end of the “always @*” block (for all possible input values)

- best to end the if-else if, etc. with an “else” clause

“always @*” means that this block of logic should always be computing its value. Combinational logic: always generate some output value based on some inputs.

== is used to compare two values. Also !=, <, <=, >, >=

= is used to assign a value to a variable

Wire/reg rule: if a variable is assigned a value in the module (through a statement such as B = 1'b1), it should be declared reg. Otherwise it should be declared wire.

Output can depend on multiple input bits

Salary raise for Rich Rodriguez if Michigan beats Ohio State or wins its bowl game

Truth table

```
module RR_raise(  
    input wire OSU,  
    input wire Bowl,  
    output reg raise);  
  
    always @* begin
```

```
        end  
endmodule
```

|| (OR) returns true if either (or both) expressions are true

&& (AND) returns true if both expressions are true

Multiple output bits

Small salary raise if U-M beats OSU, or if U-M wins its bowl game. Big salary raise if they win both games.

Truth table

Could have two separate functions for RR_raise1 and RR_raise0. Function for RR_raise0 is same as RR_raise above.

```
module RR_raise1(
  input wire OSU,
  input wire Bowl,
  output reg raise1);

  always @* begin
    if (OSU == 1'b1 && Bowl == 1'b1) begin
      raise1 = 1'b1;
    end else begin
      raise1 = 1'b0;
    end
  end
endmodule
```

Or could combine into one function that computes both RR_raise1 and RR_raise0.

```
module RR_raise10(
  input wire OSU,
  input wire Bowl,
  output reg raise1,
  output reg raise0);

  always @* begin
    if (OSU == 1'b0 && Bowl == 1'b0) begin
      raise1 = 1'b0;
      raise0 = 1'b0;
    end else if (OSU == 1'b0 && Bowl == 1'b1) begin
      raise1 = 1'b0;
      raise0 = 1'b1;
    end else if (OSU == 1'b1 && Bowl == 1'b0) begin
      raise1 = 1'b0;
      raise0 = 1'b1;
    end else begin
      raise1 = 1'b1;
      raise0 = 1'b1;
    end
  end
endmodule
```

Verilog allows array notation for multi-bit values

```
module RR_raise10_array(  
    input wire OSU,  
    input wire Bowl,  
    output reg [1:0] raise);  
  
    always @* begin  
        if (OSU == 1'b0 && Bowl == 1'b0) begin  
  
            end else if (OSU == 1'b0 && Bowl == 1'b1)begin  
  
            end else if (OSU == 1'b1 && Bowl == 1'b0)begin  
  
            end else begin  
  
            end  
  
    end  
  
end  
  
endmodule
```

Verilog operators

Verilog provides operators so we can describe these truth tables more concisely

- comparison operators: ==, !=, <, <=, >, >=
- logical operators: &&, ||, !
- arithmetic operations: +, -

Verilog's arithmetic and comparison operators interpret multi-bit values as binary unsigned integers

```
module add(  
    input wire [3:0] in1,  
    input wire [3:0] in2,  
    output reg [4:0] out);  
  
    always @* begin  
  
    end  
  
endmodule
```

Truth table

Could write the same combinational logic function in truth-table-style Verilog

```
module add(  
    input wire [3:0] in1,  
    input wire [3:0] in2,  
    output reg [4:0] out);  
  
    always @* begin  
        if (in1 == 4'b0000 && in2 == 4'b0000) begin  
            out = 5'b00000;  
        end else if (in1 == 4'b0000 && in2 == 4'b0001) begin  
            out = 5'b00001;  
        end  
        ...  
        end else if (in1 == 4'b1111 && in2 == 4'b1110) begin  
            out = 5'b11101;  
        end else begin  
            out = 5'b11110;  
        end  
    end  
end  
  
endmodule
```

We've shown how to specify the functionality of a combinational logic block. This is called "design entry" (Verilog is one way to enter a design).

- Later, we'll show how to implement a combinational logic block (truth table) with transistors.

Connecting components

Combinational logic was our first level of abstraction. Now we'll connected combinational logic blocks together to make a bigger system.

Output of one logic block can be fed into the input of the next logic block.

E.g., compute $A*B + C*D$

All logic blocks are working all the time

- top multiplier is always computing $A*B$
- bottom multiplier is always computing $C*D$
- adder is always adding its inputs

Similar to connecting different stages of a water treatment plant. Each stage processes the water or adds some substance; stages are connected via pipes.

What do you get when you combine several blocks of combinational logic?

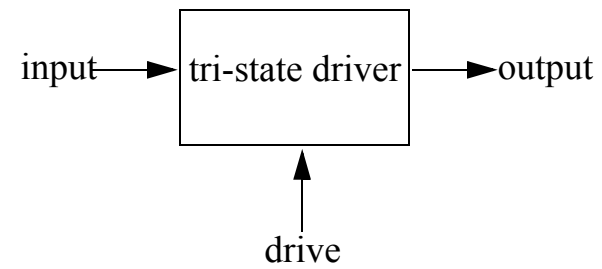
How (or whether) to connect multiple outputs?

```
module top(  
    input wire [3:0] A,  
    input wire [3:0] B,  
    input wire [3:0] C,  
    input wire [3:0] D,  
    output wire [3:0] result);  
  
    mult u1 (A, B, result);  
    mult u2 (C, D, result);  
endmodule
```

Why isn't this a good idea?

Tri-state driver

Allows us to choose between outputs



- if $drive == 1$, then $output = input$
- if $drive == 0$, then output is unconnected

Sequential logic

Combinational logic can compute lots of functions of inputs.
Think of these as little programs.

Is there any program you can't implement with combinational logic?

Tri-state drivers can be used to select between different signals

Bus: a set of wires (usually one word in "width") that can be driven by several different logic blocks

Need:

- some sense of time
- a way to remember where you are

Time

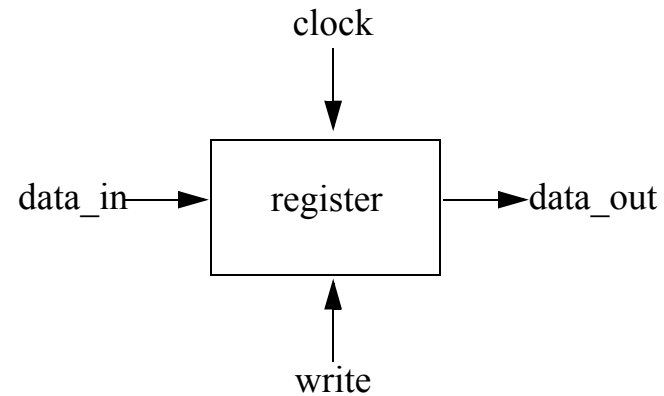
- clock is a periodic signal, alternates between 0 and 1

- events will occur when clock goes from 0->1 (positive edge of clock)
- clock has a frequency. E.g. computer might have a clock rate of 3.4 GHz (3.4 billion clock cycles per second).
What's the clock period for this frequency?

Registers

Registers provide a way to remember a value (usually a multi-bit value of default size, i.e. a word)

- can store a value in a register, then recall that value later
- will mark time by the positive edges of a clock signal



- at each positive edge of the clock, the register's value will be set to data_in if write==1
- register always drives current stored value onto data_out

```

module register(
    input wire clock,
    input wire write,
    input wire [3:0] data_in,
    output reg [3:0] data_out);

    always @(posedge clock) begin
        if (write == 1'b1) begin
            data_out <= data_in;
        end
    end
end

endmodule

```

Is a register combinational logic?

Memory

Memory is a selectable array of registers

- usually each array element is one word in width
- write to or read from the selected array element
- must have some way to specify which array element you want to select, i.e. the “address” (like an array index)

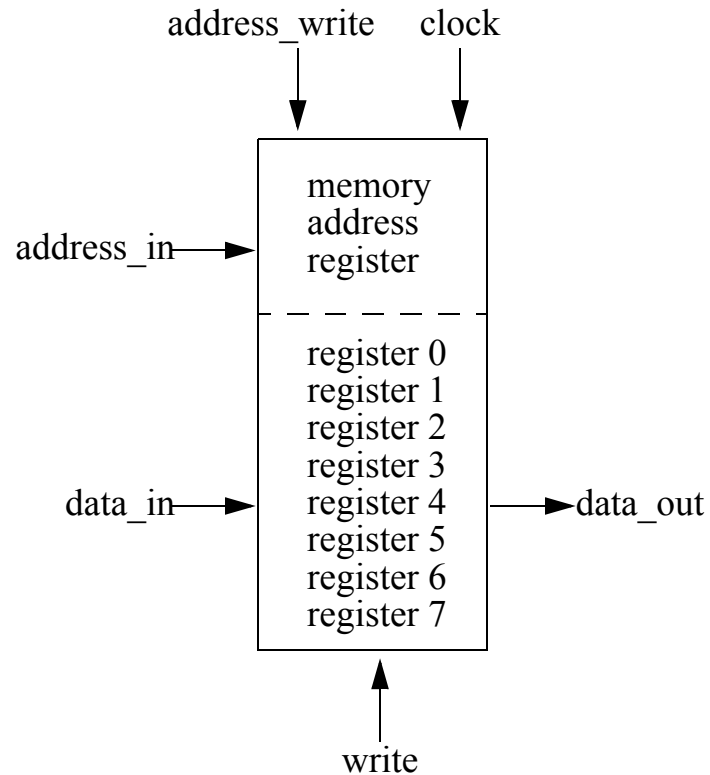
How would you write an element of an array in a high-level programming language (e.g., C++)?

We call the index of the memory array the “address”. It specifies which array element we want to read/write.

- memory (the type we use) has a dedicated register to store the address (memory address register, or MAR)
- memory operations are based on the address currently stored in the memory address register

Writing and reading memory

Steps to write memory (e.g., `memory[3] = 100`)



Memory is always driving the value of the selected element to `data_out`

- when does the new value get written to memory?

Steps to read memory (e.g., `read memory[3]`)

Build a circuit that counts up from 0

How to modify circuit so it stops counting, say at 2?

Finite-state machines

What do we mean by the “state” of something?

How would you describe the “state” of the counting circuit at a given point in time?

Consider a circuit that repeats some part of the sequence:

- 0, 1, **1**, 2, 2, 2, ...
- how would you describe its state at a given time?

What to do in a state?

At each state, I can determine what to do. What do I need to determine?

state	output	next_state

```
always @* begin
    if (state == 2'h0) begin

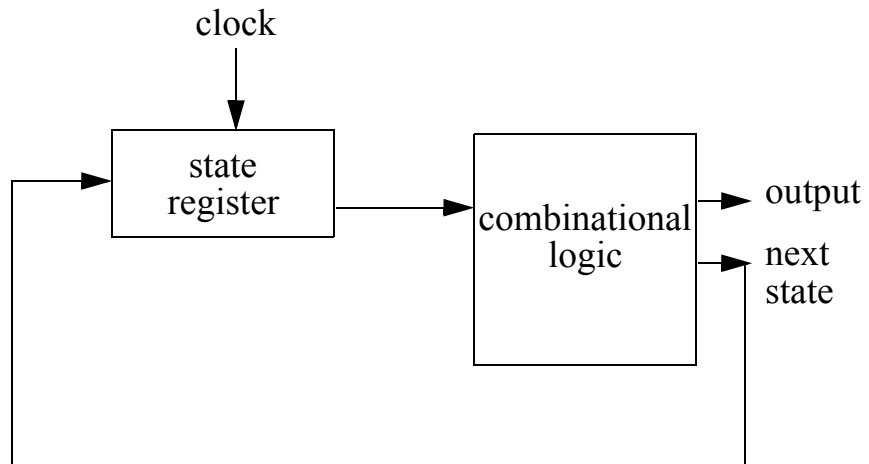
        end else if (state == 2'h1) begin

        end else if (state == 2'h2) begin

        end else if (state == 2'h3) begin

        end
end
end
```

Circuit with input



Verilog code to store the next state into the state register

```
always @(posedge clock) begin
    state <= next_state;
end
```

Count as before

- 0, 1, 1, 2, 2, 2, ...
- but if clear==1 and you're printing 2, then start back at 0

state	clear	output	next_state

```

always @* begin
  if (state == 2'h0) begin
    output = 2'h0;
    next_state = 2'h1;

  end else if (state == 2'h1) begin
    output = 2'h1;
    next_state = 2'h2;

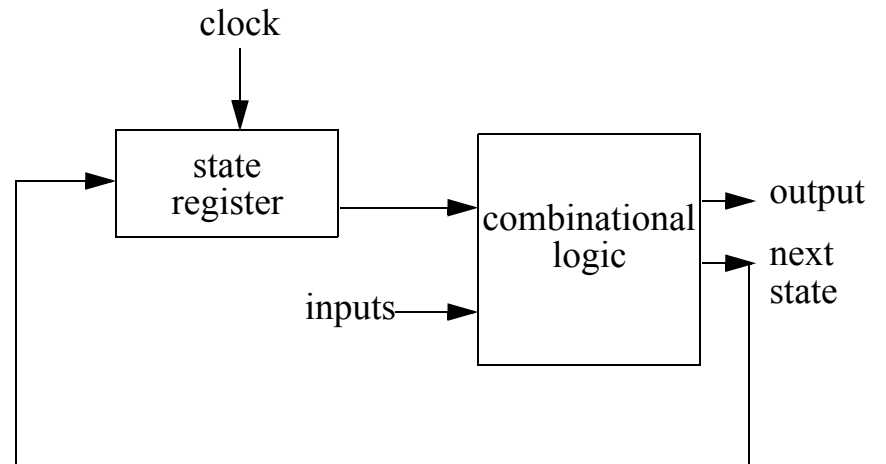
  end else if (state == 2'h2) begin
    output = 2'h1;
    next_state = 2'h3;

  end else if (state == 2'h3) begin
    output = 2'h2;

  end

end
end

```



Discussion of finite-state machines

What do finite-state machines remind you of?

What variables determine the current output?

What variables determine the next state?

Does it matter what number I assign to each state?

We'll refer to states symbolically (instead of with a number) to make the FSM easier to read

- parameter state_end = 2'h3;

Digital circuits

Control unit (implemented as a finite-state machine)

- FSM executes a series of steps to carry out the algorithm
- at each step, the FSM generates control signals that control the rest of the circuit

Datapath

- FSM not enough to implement the algorithm
- need other components: registers, memory, combinational logic
- datapath shows the components of the digital circuit (usually doesn't show the control unit) and how they're connected

Example

A finite-state machine to read or write memory contents

- 4-bit memory address
- 4-bit memory data
- if input READ is 1, circuit should read memory location DPDT_SW[3:0] and display to LED_RED[3:0]
- if input WRITE is 1, circuit should write memory location DPDT_SW[3:0] with the value DPDT_SW[7:4], then start displaying the contents of that location
- otherwise keep reading the last address specified

Datapath

What are the inputs of the control unit?

What are the outputs of the control unit?

High-level design of finite-state machine

Control unit (truth table)

state	READ	WRITE	next_state	mem_write	address_write

Control unit (Verilog)

```
always @* begin

    if (state == state_reset) begin
        address_write =
        mem_write =
        next_state =

    end else if (state == state_decide) begin
        address_write =
        mem_write =
```

```
end else if (state == state_read1) begin
    address_write =
    mem_write =
    next_state =

end else if (state == state_write1) begin
    address_write =
    mem_write =
    next_state =

end else if (state == state_write2) begin
    address_write =
    mem_write =
    next_state =

end

end
```

Verilog to update the state register

```
always @(posedge clock) begin
  if (reset == 1'b1) begin
    state <= state_reset;
  end else begin
    state <= next_state;
  end
end
end
```

This code will appear in all your finite state machines

Easiest and safest to assign some default values to the outputs, then only write the changes in the FSM's if statements

- what should the default (safe, i.e. no action) values be for the outputs?

Implementing algorithms in hardware: datapath

1. Write the algorithm as a program in C++ (or another software language).
2. Rewrite according to FSM limitations
 - no loops; only gotos
 - if statements can only execute gotos
 - comparison and arithmetic operations take input only from registers
3. List the datapath elements from the C++ program
 - scalar variables =>
 - array variables =>
 - arithmetic operations and comparison =>
4. Connect data signals for datapath elements

- we'll use a one-bus design style. Output of most datapath elements are connected to the inputs of most datapath elements via a shared bus. This style is general but not optimal.
 - registers and memory get input from bus
 - combinational logic get input from registers. Why can't combinational logic get its inputs from the bus?
-
- we'll connect the outputs as needed, as we write the steps of the finite-state machine

Implementing algorithms in hardware: **control unit**

5. List the control signals for the datapath elements. These will be outputs from the control unit.
 - registers
 - memory
 - tri-state driver
6. Write control unit FSM
 - most steps involve a transfer of some data from one datapath element to another
 - if an FSM step needs to transfer data between datapath elements, make sure there's a connection for the data to flow between these datapath elements
 - connect an input from combinational logic or register to FSM when the FSM needs that input to decide next state
 - add datapath elements as needed

Example: find maximum element in an array

Write algorithm as C++ program

Re-write algorithm with limitations

Datapath for max

Control unit for max (Verilog)

```
always @* begin
    element_write = 1'b0;
    element_drive = 1'b0;
    max_write = 1'b0;
    i_write = 1'b0;
    i_drive = 1'b0;
    plus1_drive = 1'b0;
    memory_write = 1'b0;
    memory_drive = 1'b0;
    address_write = 1'b0;
    next_state = state_reset;

    if (state == state_reset) begin
        next_state = state_loop;
    end else if (state == state_loop) begin
        // are we done?
        if (equal16_out == 1'b1) begin
            next_state = state_end;
        end else begin
            next_state = state_read1;
        end
    end else if (state == state_read1) begin
        // transfer i to memory address
        i_drive = 1'b1;
        address_write = 1'b1;
        next_state = state_read2;
```

```
    end else if (state == state_read2) begin
        // read memory[i]
        memory_drive = 1'b1;
        element_write = 1'b1;
        next_state = state_compare;
    end else if (state == state_compare) begin
        // is memory[i] more than the current max?
        if (greater_out == 1'b1) begin
            next_state = state_write_max;
        end else begin
            next_state = state_increment;
        end
    end else if (state == state_write_max) begin
        // update max
        element_drive = 1'b1;
        max_write = 1'b1;
        next_state = state_increment;
    end else if (state == state_increment) begin
        // increment i
        plus1_drive = 1'b1;
        i_write = 1'b1;
        next_state = state_loop;
    end else if (state == state_end) begin
        next_state = state_end;
    end
end
```

Datapath for max (Verilog): top.v (partial)

```
register u3 (clock, reset, element_write, bus,
  element_out);
register u4 (clock, reset, max_write, bus,
  max_out);
register u5 (clock, reset, i_write, bus, i_out);

greater u6 (element_out, max_out, greater_out);
equal16 u7 (i_out, equal16_out);
plus1 u8 (i_out, plus1_out);

ram u9 (bus, ~address_write, clock, bus,
  memory_write, memory_out);

tristate u10 (element_out, bus, element_drive);
tristate u11 (plus1_out, bus, plus1_drive);
tristate u12 (i_out, bus, i_drive);
tristate u13 (memory_out, bus, memory_drive);

// display max on HEX2, HEX3
hexdigit u16 (max_out[3:0], HEX2);
hexdigit u17 (max_out[7:4], HEX3);

control u18 (clock, reset, greater_out,
  equal16_out, element_write, element_drive,
  max_write, i_write, i_drive, plus1_drive,
  memory_write, memory_drive, address_write);
```

Datapath for max (Verilog): greater.v

```
module greater(
  input wire [7:0] in1,
  input wire [7:0] in2,
  output reg out);

  always @* begin
    if (in1 > in2) begin
      out = 1'b1;
    end else begin
      out = 1'b0;
    end
  end

endmodule
```

Summary

Now you know how to implement simple programs in hardware

- datapath: the circuitry to do simple things with bits (store, communicate, compute)
- control unit (implemented as a finite-state machine): the circuitry to execute a sequence of steps