Abstraction

- Aspects of a component that might be used in a larger design:
  - Interface (or behavior): How do you use it? What is it supposed to do?
  - Implementation: How does it work? What’s inside the box?
- 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)
• Schematic representation

  \[ A \xrightarrow{\text{combinational logic}} B \]

• 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
Example: Alice and Bob going to a party

- Definition
  - 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:

- 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.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>
Another truth table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>

**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)
module NOT(
    input wire A,
    output reg B);

always @* begin

    case (A)

        1'b0: begin
            B = 1'b1;
        end

        1'b1: begin
            B = 1'b0;
        end

    endcase
end

endmodule

Verilog

- 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
- Note the code inside the “always @*” block
  - This is a program that computes a value of B (for a given value of A)
  - “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 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.
Combinational logic in Verilog

• 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)
  – To be safe: set a default value, then specify conditions under which it should have a different value

module NOT(
    input wire A,
    output reg B);

    always @* begin
        B = 1'b0;
        case (A)
            1'b0: begin
                B = 1'b1;
            end
        endcase
    end
endmodule
Truth-table style Verilog

- Can always enumerate all possible inputs and specify the desired output value for each input
- But this is tedious
- Verilog provides more powerful constructs so we can express circuits more concisely

```verilog
module NOT(
    input wire A,
    output reg B);

    always @* begin
        if (A == 1'b0) begin
            B = 1'b1;
        end else if (A == 1'b1) begin
            B = 1'b0;
        end
    end
endmodule

== is used to compare two values. Also !=, <, <=, >, >=
```
Output can depend on multiple input bits

- Salary raise for football coach if Michigan beats Ohio State or wins its bowl game
- Truth table:

Verilog for coach_raise

```verilog
module coach_raise(
    input wire OSU,
    input wire Bowl,
    output reg raise);

    always @* begin
        raise = 1'b0;

        if (OSU == 1'b1 || Bowl == 1'b1) begin
            raise = 1'b1;
        end
    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:

```verilog
module coach_raise0(
    input wire OSU,
    input wire Bowl,
    output reg raise0);

always @* begin
    raise0 = 1'b0;
    if (OSU == 1'b1 || Bowl == 1'b1)
        begin
            raise0 = 1'b1;
        end
end
endmodule

module coach_raise1(
    input wire OSU,
    input wire Bowl,
    output reg raise1);

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

Verilog for multiple output bits (two modules)
Combine into one function that computes both coach_raise1 and coach_raise0

module coach_raise10(
    input wire OSU,
    input wire Bowl,
    output reg raise1,
    output reg raise0);

    always @* begin
        raise1 = 1'b0;
        raise0 = 1'b0;

        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 if (OSU == 1'b1 && Bowl == 1'b1) begin
            raise1 = 1'b1;
            raise0 = 1'b1;
        end
    end
endmodule

Verilog allows array notation for multi-bit values

module coach_raise10(
    input wire OSU,
    input wire Bowl,
    output reg [1:0] raise);

    always @* begin
        raise[1:0] = 2'b00;

        if (OSU == 1'b0 && Bowl == 1'b1) begin
            raise[1:0] = 2'b01;
        end else if (OSU == 1'b1 && Bowl == 1'b0) begin
            raise[1:0] = 2'b01;
        end else if (OSU == 1'b1 && Bowl == 1'b1) begin
            raise[1:0] = 2'b11;
        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
    out[4:0] = in1[3:0] + in2[3:0];
  end
endmodule

Or could use (very long!) truth table

• Truth table for addition

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
    end
    endmodule
Abstraction

• 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 (not just specify its behavior).

Connecting components

• Combinational logic was our first level of abstraction. Now we’ll connect 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.
Connecting components in Verilog

- Verilog uses syntax similar to function calls to express how components are connected
  - However, it means something quite different: i.e. it's instantiating a component and specifying the connections to that component
  - The component evaluates its inputs and generates its outputs continuously, rather than a one-time execution

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

Building blocks

```verilog
module mult(
    input wire [3:0] in1,
    input wire [3:0] in2,
    output reg [3:0] out);

    always @* begin
        out = in1 * in2;
    end
endmodule

module add(
    input wire [3:0] in1,
    input wire [3:0] in2,
    output reg [3:0] out);

    always @* begin
        out = in1 + in2;
    end
endmodule
```
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

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 is this a bad idea?
### Tri-state driver

- Allows us to choose between outputs

  ![Tri-state driver diagram](image)

  - If `drive==1`, then `output = input`
  - If `drive==0`, then output is unconnected

<table>
<thead>
<tr>
<th>drive</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>unconnected</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>unconnected</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- 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
Sequential logic

- Combinational logic can compute lots of functions of inputs. Think of these as little programs.
- Is there any program you can not implement with combinational logic?

What does the following code do?

```verilog
module counter(
    input wire count,
    output reg [1:0] LED_RED);

    always @* begin
        if (count == 1'b0) begin
            LED_RED[1:0] = 2'h0;
        end else begin
            LED_RED[1:0] = 2'h0;
            LED_RED[1:0] = 2'h1;
            LED_RED[1:0] = 2'h2;
            LED_RED[1:0] = 2'h3;
        end
    end
endmodule
```
What do you need to implement all programs?

• 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, if write==1, then the register’s value will be set to data_in
  - Register always drives current stored value onto data_out

\[\text{clock} \rightarrow \text{data}_\text{in} \rightarrow \text{register} \rightarrow \text{data}_\text{out} \rightarrow \text{write}\]
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

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

Writing and reading memory

• Steps to write memory (e.g., memory[3] = 100)
  – When does the new value get written to memory?

• Steps to read memory (e.g., read memory[3])
Executing a sequence of steps

• An algorithm is a series of steps (we’ll call these “states”)
• Example: baking a cake
  – Reset
  – Pour ingredients into pan
  – Mix ingredients
  – Bake pan in oven
  – End
• Build a circuit that goes through this sequence of steps (reset, pour, mix, bake, end)
  – Don’t worry about executing the action at each step
  – Assume reset signal initializes circuit to state reset

A baking machine

• Circuit

• Truth table for compute_next_state
module compute_next_state(
    input wire [2:0] state,
    output reg [2:0] next_state);
always @* begin
    next_state = state_reset;
    case (state)
        state_reset: begin
            next_state = state_pour;
        end
        state_pour: begin
            next_state = state_mix;
        end
        state_mix: begin
            next_state = state_bake;
        end
        state_bake: begin
            next_state = state_end;
        end
        state_end: begin
            next_state = state_end;
        end
    endcase
end
endmodule
module state_register(
    input wire clock,
    input wire reset,
    input wire [2:0] data_in,
    output reg [2:0] data_out);

    always @(posedge clock) begin
        if (reset == 1'b1) begin
            data_out <= state_reset;
        end else begin
            data_out <= data_in;
        end
    end
endmodule

module top(
    input wire reset,
    input wire clock);

    wire [2:0] next_state;
    wire [2:0] state;

    state_register u1 (clock, reset, next_state, state);
    compute_next_state u2 (state, next_state);

endmodule

parameter state_reset = 3'h0;
parameter state_pour  = 3'h1;
parameter state_mix   = 3'h2;
parameter state_bake  = 3'h3;
parameter state_end   = 3'h4;
Input can affect the sequence of steps

• Truth table for computing next state

<table>
<thead>
<tr>
<th>state</th>
<th>smoke</th>
<th>next_state</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td></td>
<td>pour</td>
</tr>
<tr>
<td>pour</td>
<td></td>
<td>mix</td>
</tr>
<tr>
<td>mix</td>
<td></td>
<td>bake</td>
</tr>
<tr>
<td>bake</td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>

Concise representation of truth table for computing next state

<table>
<thead>
<tr>
<th>state</th>
<th>smoke</th>
<th>next_state</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td></td>
<td>pour</td>
</tr>
<tr>
<td>pour</td>
<td></td>
<td>mix</td>
</tr>
<tr>
<td>mix</td>
<td></td>
<td>bake</td>
</tr>
<tr>
<td>bake</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bake</td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>
module compute_next_state(
    input  wire [2:0]  state,
    input  wire  smoke;
    output  reg [2:0]  next_state);

always @* begin

    next_state = state_reset;

    case (state)
        state_reset: begin
            next_state = state_pour;
        end

        state_pour: begin
            next_state = state_mix;
        end

        state_mix: begin
            next_state = state_bake;
        end

        state_bake: begin
            if (smoke == 1'b0) begin
                next_state = state_bake;
            end else begin
                next_state = state_end;
            end

        state_end: begin
            next_state = state_end;
        end

        endcase

    end

endmodule

Finite-state machines

- Execute a sequence of steps
- Does it matter what number I assign to each state?
Executing the actions for each step

- Still need to execute action(s) at each step
- E.g., for cake FSM
  - Need machine to pour the ingredients
  - Need mixer to mix the ingredients
  - Need oven to bake the cake
- Datapath carries out each step
  - For the cake example, datapath would be a machine to pour the ingredients, a mixer, and an oven
- At each step, the FSM specifies what the datapath should be doing during that step, by outputting control signals that control the datapath.

<table>
<thead>
<tr>
<th>state</th>
<th>pourer</th>
<th>mixer</th>
<th>oven</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pour</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mix</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bake</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Combined truth table for next state and control signals

<table>
<thead>
<tr>
<th>state</th>
<th>smoke</th>
<th>next_state</th>
<th>pourer</th>
<th>mixer</th>
<th>oven</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td></td>
<td>pour</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pour</td>
<td></td>
<td>mix</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mix</td>
<td></td>
<td>bake</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bake</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bake</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td>end</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What variables determine the control signals?
- What variables determine the next state?
always @* begin
    pourer = 1'b0;
    mixer = 1'b0;
    oven = 1'b0;
    next_state = state_reset;
    case (state)
        state_reset: begin
            next_state = state_pour;
        end
        state_pour: begin
            pourer = 1'b1;
            next_state = state_mix;
        end
        state_mix: begin
            mixer = 1'b1;
            next_state = state_bake;
        end
        if (smoke == 1'b0) begin
            next_state = state_bake;
        end else begin
            next_state = state_end;
        end
    endcase
end

- 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?

- What do finite-state machines remind you of?
Carrying out a sequence of computations

• Instead of kitchen operations, we’ll execute a sequence of computations.
• E.g., exchange variable a with variable b
  step 1: tmp = a
  step 2: a = b
  step 3: b = tmp

• Digital circuit has two parts:
  – Control unit (implemented as a finite-state machine): 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: other components needed to implement algorithm (registers, memory, combinational logic, wires).
Example: A finite-state machine to read or write memory contents

• User specifies:
  – Whether to carry out a new command (start)
  – The type of access. type = 0 (READ), or type = 1 (WRITE)
  – What memory address to access (SW[3:0])
  – If writing, what data to write to memory (SW[7:4])

• Behavior:
  – On read command, circuit should read memory location and display to LED_RED[3:0]
  – On write command, circuit should write memory location with the specified value, then start displaying the contents of that location
  – Otherwise keep reading the last address specified

Datapath
• Inputs of the control unit

• Outputs of the control unit

• High-level design of finite-state machine

---

Control unit (truth table)

<table>
<thead>
<tr>
<th>state</th>
<th>start</th>
<th>type</th>
<th>next state</th>
<th>address write</th>
<th>mem write</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
always @* begin
    address_write = 1'b0;
    mem_write = 1'b0;
    next_state = state_reset;

    case (state)
        state_reset: begin
            next_state = state_decide;
        end

        state_decide: begin
            if (start == 1'b0) begin
                next_state = state_decide;
            end else if (type == 1'b0) begin
                next_state = state_read1;
            end else begin
                next_state = state_write1;
            end

        end case
end
Entire circuit

• How does this differ from Lab 3?

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;
end

case (state)
    state_reset: begin
        next_state = state_loop;
    end

    state_loop: begin
        // if i is 16, then we're done
        if (equal16_out == 1'b1) begin
            next_state = state_end;
        end else begin
            next_state = state_read1;
        end
    end

endcase
state_read1: begin
    // transfer i to memory address
    i_drive = 1'b1;
    address_write = 1'b1;
    next_state = state_read2;
end

state_read2: begin
    // read memory[i]
    memory_drive = 1'b1;
    element_write = 1'b1;
    next_state = state_compare;
end

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

state_write_max: begin
    // update max
    element_drive = 1'b1;
    max_write = 1'b1;
    next_state = state_increment;
end

state_increment: begin
    // increment i
    plus1_drive = 1'b1;
    i_write = 1'b1;
    next_state = state_loop;
end

state_end: begin
    next_state = state_end;
end
endcase
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
  – Control unit (implemented as a finite-state machine): the circuitry to execute a sequence of steps
  – Datapath: the circuitry to carry out each step by doing simple things with bits (store, communicate, compute)