Design Approach
(Simple Design)
Steps in the Design of Small Modules

1. Determine/Obtain Specification
2. Structure design to register transfer level
3. Capture design as Verilog
4. Verify Design
5. Synthesize design
6. Verify Results
7. Place and Route
8. Final Verification
The Specification

► Details the **behavior** and **interface** of each module in the design.

► Very important step – Many chips fail in the marketplace because the feature set was not well thought through.

► Covers multiple levels of the design, all the way from the chip level functionality down to model level timing.
The Specification (continue..)

Specification should include following . . .

1. A description of the top-level behavior of the module
2. A description of all its inputs and outputs, their timing, and constraints
3. Performance requirements and constraints
The Specification (continue..)

Ideally, the top-level behavior of the chip and each module should be captured in a high-level language, such as C++, Java, Matlab or ‘Specification and Description Language’ (SDL).

- Helps to simulate and verify the behavioral specification, fewer bugs will be introduced at this level.
- The structure of the high-level language description of the module does not have to relate to the internal hardware about to be designed.
Structuring the Design

When this step is given short shrift, the resulting design is often buggy and inefficient in performance and/or area.
1. **Determine the control strategy**

   - *Clearly separate Control from Data-path*
   
     - The data-path and controller are joined by clearly identified *control signals* and *status signals*.
     
     - Typically the controller consists of some combination of Finite State Machines and counters.
     
     - The data path consists of a number of registers and combinational logic.
     
     - The controller determines the sequence of events that take place on the data-path.
1. Determine the control strategy (continue..)

- Key elements must be considered when determining the control strategy
  - **Reset Strategy:**
    - Every chip has global reset
    - Becomes active on power up and when the reset button is pushed
    - Typically active low
    - Sets the contents of various registers to a predetermined value to put the chip into a known state.
  - **What is to be performed in each clock period:**
  - **Special attention to transitions:**
    - Transition between different modes of operation
2. Determine the register transfer level structure of the design

- **Module inputs and outputs**
  - Identify, name, and determine the function of each input and output signal.

- **Registers and register outputs**
  - Identify each register and give a signal name to its output(s).
  - The purpose of each register must be clear in the designer’s head.
  - Usually it is dangerous to mix together registers that are clocked every clock cycle with registers that are clocked less or more frequently.
  - Because it complicates synthesis as well as it complicates the design.
Structuring $\rightarrow$ Sub-steps (continue..)

2. Determine the register transfer level structure of the design (continue..)

- **Combinational logic blocks and their functions**
  - Identify the blocks of logic, including arithmetic units, multiplexers, comparators, deselectors, coders, decoders, etc. Not a lot of detail is needed at this stage.

*Always design before coding*
3. Capture the design as Verilog

   If the previous steps are done correctly, then this step is straightforward.

   - Edge-triggered flipflops and their associated input logic are built using procedural blocks containing `always @(posedge clock)` statements.
   - Level sensitive latches (and associated input logic), if used, are built using procedural blocks headed with `always @(clock)` statements.
   - Simple combinational logic is built using continuous assignment statements.
   - More complex combinational logic is built using procedural blocks that do not contain ‘clock’ in the timing sensitivity list.
Capture the design as Verilog (continue..)

always @(posedge clock)
  case (sel)
    0: E <= D + B;
    1: E <= B;
  endcase
Capture the design as Verilog (continue..)

```verilog
assign H = C | F;

always @(A or B or C)
  case (A)
    0: D = B;
    1: D = C;
    2: D = 0;
    3: D = 1;
  endcase
```

Design Example - 1

A simple down counter…

1. Specification

- For this module, the specification will be conveyed by a simple document identifying the module and specifying the behavior of the inputs and outputs.

```
in[3:0] latch dec clock zero
```
Overall Function:

► A count down with the following functions.
  - When ‘latch’ is high, the value ‘in’ is loaded into the timer.
  - When ‘dec’ is high, the internal value is decremented by one every clock period.
  - The count down stops at zero, whereupon ‘zero’ goes high.
  - The internal count down value stays at 0 then until the counter is reloaded.
Design Example – 1 (continue...) (continue...)

Inputs:
► clock: Clock - frequency varies.
► in: Initial number loaded into counter
► latch: when latch is high, the counter is loaded from ‘in’
► Dec: Decrement the counter when ‘dec’ is high

Outputs:
► Zero: Goes high when count reaches zero. Stays high until latch goes high.
Step 1. Identify inputs and outputs

in[3:0] __________
latch ________
dec _________
clock_________

_________ zero
Step 2. Identify Registers

- in[3:0]
- latch
- dec
- clock
- zero
Step 3. Identify inputs and outputs

- in[3:0]
- latch
- dec
- clock
- -1
- ==0?
- zero
Capture the design as verilog

module counter (clock, in, latch, dec, zero);
input clock;       // clock
input [3:0] in;   // input initial count
input latch;       // latch input
input dec;       // decrement
output zero;       // zero flag
reg [3:0] value;
wire zero;

// count flipflops with input multiplexer and subtractor
always @(posedge clk)
begin
  if(latch)
    value <= in;
  else if (dec && !zero)
    value <= value - 1'b1;
end

Assign zero = ~|value; // combinational logic for zero flag
endmodule
Complex designs
Steps in High-level design

- Time spent generating a “paper” design before coding is more than rewarded with a smaller, faster design, and a much quicker debugging cycle.
- An important guiding principle in high-level design is to clearly separate the datapath from the controller.

*Always clearly separate datapath from controller*
Step-1: Structuring the datapath

- From the structural and performance specification, determine the functional units needed and their connectivity.
  - The quality of the design at this stage will largely determine the overall performance and area of the function.
  - It is important for the designer to determine the degree of parallelism and pipelining required.
Step-2: Identify Control Points

- The control points consist of control lines and status lines.

- Control lines originate in the controller and determine the detailed operations to be performed on the datapath during any one clock cycle.

- Status lines originate in the datapath and indicate the status outputs of important units, for example zero and overflow flags.
Step-3: Determine transition points in control flow

- It is important to identify critical points in the control sequence.
- For example, start, stop, and transitions between different modes of operation.
- These points generally require special attention to get them right.
Step-4: Determine Control Strategy and Control Flow

► FSMs. One or more state machines might be necessary to control events.

► When more than one FSM is needed, either from a functional point of view or to limit the number of states for speed reasons, a top-down control approach is better than an interconnected one.

► In a top-down approach, higher-level controllers control lower-level ones.
  
  ▪ The state of the lower-level machines depends only on the state of the higher-level machine.
Generation of Control sequences

Finite State Machine (FSM)

Next State Logic

Output Logic

Current state

Next state

inputs

Encoded control lines

Clock
Generation of Control sequences (continue…)

Hierarchical control preferred
Step-4: Determine Control Strategy and Control Flow (cont…)

- Microcode. Sometimes instead of a large FSM, a micro-coded controller is used as the design can be made smaller and faster.

- It consists of a micro-coded ROM (which is sometimes RAM), a micro program counter (uPC), and next uPC address logic.

- The next micro-code address is either the previous address plus one (sequential addressing through the micro-code) or determined from the ‘jump’ field of the micro-coded ROM.
Micro Program Counter (uPC)

Dr. Behnam Arad
EEE/CSC-273
Step-4: Determine Control Strategy and Control Flow (cont.)

- **Counters.**
  - Often it is necessary to count events making counters useful in the control strategy.

- **Combinational Decoders.**
  - Often decode logic is needed that takes the output of one of the above and decodes it into the separate control points.
  - By using these decoders, the sequential control portion (FSM, microcode, or counter) can often be made smaller than if these decoders were not present.
Elements Used in Building Controllers – Counter.
Elements Used in Building Controllers – Decode Logic
Reset is a global signal distributed across the entire chip that initializes the chip to a known state. It is generally toggled on power-up or when a reset button is depressed. Without reset, the chip would power up in a random state.
Step-6: Verify before Coding

- Generally, it is a useful practice to hand-simulate the design before coding.
- Draw a timing diagram capturing critical events to help you find potential bugs early.
Design Partitioning

► Only modules at the bottom of the hierarchy should contain logic
  ▪ The higher level modules should consist of only instantiations of lower-level modules and wires.

► Critical paths should be contained entirely within one module.
  ▪ Critical Path: the slowest path of combinational logic that exists between registers.

► Whenever possible, register the outputs of modules.
  ▪ Whenever possible, the output of a module should be a flipflop output.
Design Partitioning (continue...) 

- Potentially sharable resources should be within one module
  - In general, keep closely related portions of the design within one module (or even better, within one procedural block)

- Separate modules that will have different synthesis strategies

- Keep modules as small as possible, consistent with the above guidelines
  - Synthesis has a better chance of optimizing small designs than large ones.