EECS150 - Digital Design
Lecture 4 - Register & Flip-flops

September 9, 2013
Prof. Ronald Fearing
Electrical Engineering and Computer Sciences
University of California, Berkeley

(slides courtesy of Prof. John Wawrzynek)

http://www-inst.eecs.berkeley.edu/~cs150

Outline

- Recap

```
Adder adder4 ( ... );
Adder #(N(64)) adder64 ( ... );
```

- Introduction to rising edge trigger FF and registers
- timing for Comb. logic and registers
- introductory finite state machine example and Verilog
Only Two Types of Circuits Exist

- Combinational Logic Blocks (CL) \( \Leftarrow \) \( \text{out} = f(\text{inputs}) \)
- State Elements (registers) \( \Leftarrow \) \( y_{n+1} = f(\text{inputs}, y_n) \)

State Elements: circuits that store info

- Examples: registers, memories
- Register: Under the control of the "load" signal, the register captures the input value and stores it indefinitely.
  - The value stored by the register appears on the output (after a small delay).
  - Until the next load, changes on the data input are ignored (unlike CL, where input changes change output).
  - These get used for short term storage (ex: register file), and to help move data around the processor.
Register Details... What’s inside?

- n instances of a “Flip-Flop”
- Flip-flop name because the output flips and flops between and 0,1
- D is “data”, Q is “output”
- Also called “D-type Flip-Flop”

Flip-flop Timing Waveforms?

- Edge-triggered d-type flip-flop
  - This one is “positive edge-triggered”
  - “On the rising edge of the clock, the input d is sampled and transferred to the output. At all other times, the input d is ignored.”
- Example waveforms:
Uses for State Elements

1) As a place to store values for some indeterminate amount of time:
   - Register files (like $1$-$31$ on the MIPS)
   - Memory (caches, and main memory)

2) Help control the flow of information between combinational logic blocks.
   - State elements are used to hold up the movement of information at the inputs to combinational logic blocks and allow for orderly passage. (e.g. pipeline)

Accumulator Circuit Example

Assume $X$ is a vector of $N$ integers, presented to the input of our accumulator circuit one at a time (one per clock cycle), so that after $N$ clock cycles, $S$ holds the sum of all $N$ numbers.

\[
S = 0; \quad \text{Repeat } N \text{ times} \quad S = S + X;
\]

• We need something like this:
  • But not quite.
  • Need to use the clock signal to hold up the feedback to match up with the input signal.
**Accumulator Circuit**

- Put register, with clock signal controlling its load, in feedback path.
- On each clock cycle the register prevents the new value from reaching the input to the adder prematurely. (The new value just waits at the input of the register).

**Timing:**

![Timing Diagram]

**Flip-Flop Timing Details**

Three important times associated with flip-flops:
- setup time
- hold time
- clock-to-q delay.
Accumulator Revisited

- Note:
  - Reset signal (synchronous)
  - Timing of X signal is not known without investigating the circuit that supplies X. Here we assume it comes just after $S_{i-1}$.
  - (sometime after rising edge of clock for synchronous system)
  - Observe transient behavior of $S_i$.

Pipelining to improve performance (1/2)

Extra Registers are often added to help speed up the clock rate.

Note: delay of 1 clock cycle from input to output.
Clock period limited by propagation delay of adder/shifter.
Pipelining to improve performance (2/2)

- Insertion of register allows higher clock frequency.
- More outputs per second.

Timing...

Level-sensitive Latch Inside Flip-flop

Positive Level-sensitive latch:

When CLK is high, latch is transparent, when clk is low, latch retains previous value.

Positive Edge-triggered flip-flop built from two level-sensitive latches:
Flip-flops on Virtex5 FPGA

Other flip-flops in the chip input/output cells, and in the form of registers in the DSP slices and memory block interfaces.

Virtex5 Slice Flip-flops

4 flip-flops / slice (corresponding to the 4 6-LUTs)

Each takes input from LUT output or primary slice input.

Edge-triggered FF vs. level-sensitive latch.
Clock-enable input (can be set to 1 to disable) (shared).
Positive versus negative clock-edge.
Synchronous vs. asynchronous reset.
SRHIGH/SRLow select reset (SR) set.
REV forces opposite state.
INIT0/INIT1 used for global reset (not shown - usually just after power-on and configuration).
Virtex5 Flip-flops “Primitives”

D Flip-Flop with Synchronous Reset and Set and Clock Enable

Provided by the CAD tools. This maps to single slice flip-flop.

Verilog Instantiation Template (only if you want structural/low level)

// FDRSE: Single Data Rate D Flip-Flop with
// Synchronous Clear Set and Clock Enable (posedge clk).
FDRSE #(.INIT(1'b0) // Initial value of register
    ) FDRSE_inst (  
    .Q(Q), // Data output
    .C(C), // Clock input
    .CE(CE), // Clock enable input
    .D(D), // Data input
    .R(R), // Synchronous reset input
    .S(S) // Synchronous set input
    );
State Elements in Verilog

Always blocks are the only way to specify the "behavior" of state elements. Synthesis tools will turn state element behaviors into state element instances.

D-flip-flop with synchronous set and reset example:

```
module dff(q, d, clk, set, rst);
  input d, clk, set, rst;
  output q;
  reg q;
  always @(posedge clk)
    if (rst)
      q <= 1'b0;
    else if (set)
      q <= 1'b1;
    else
      q <= d;
  endmodule
```

D-flip-flop with synchronous set and reset example:

```
module dff(q, d, clk, set, rst);
  input d, clk, set, rst;
  output q;
  reg q;
  always @(posedge clk)
    if (rst)
      q <= 1'b0;
    else if (set)
      q <= 1'b1;
    else
      q <= d;
  endmodule
```

How would you add an CE (clock enable) input?

Finite State Machines

State Transition Diagram

Implementation Circuit Diagram

Holds a symbol to keep track of which bubble the FSM is in.

CL functions to determine output value and next state based on input and current state.

out = f(in, current state)
next state = f(in, current state)
Finite State Machines

module FSM1(clk, rst, in, out);
input clk, rst;
input in;
output out;

// Defined state encoding:
parameter IDLE = 2'b00;
parameter S0 = 2'b01;
parameter S1 = 2'b10;
reg out;
reg [1:0] state, next_state;

// always block for state register
always @(posedge clk)
if (rst) state <= IDLE;
else state <= next_state;

// always block for combinational logic portion
always @(state or in)
case (state)
  // For each state def output and next
  IDLE : begin
    out = 1'b0;
    if (in == 1'b1) next_state = S0;
    else next_state = IDLE;
  end
  S0 : begin
    out = 1'b0;
    if (in == 1'b1) next_state = S1;
    else next_state = IDLE;
  end
  S1 : begin
    out = 1'b1;
    if (in == 1'b1) next_state = S1;
    else next_state = IDLE;
  end
  default: begin
    next_state = IDLE;
    out = 1'b0;
  end
endcase
endmodule

Each state becomes a case clause.
For each state define:
  Output value(s)
  State transition

The register to hold the “state” of the FSM.

Must use reset to force to initial state.
Reset not always shown in STD

Combination logic signals for transition.

wire vs reg?

~cs150/fa13/resources/Nets.pdf
Example - Parallel to Serial Converter

```
module ParToSer(ld, X, out, clk);
    input [3:0] X;
    input ld, clk;
    output out;
    reg [3:0] Q;
    wire [3:0] NS;
    assign NS =
        (ld) ? X : {Q[0], Q[3:1]};
    always @ (posedge clk)
        Q <= NS;
    assign out = Q[0];
endmodule
```

Specifications:
- Specifies the muxing with "rotation"
- Forces Q register (flip-flops) to be rewritten every cycle
- Connect output

Parameterized Version

Parameters give us a way to generalize our designs. A module becomes a “generator” for different variations. Enables design/module reuse. Can simplify testing.

```
module ParToSer(ld, X, out, CLK);
    input [3:0] X;
    input ld, clk;
    output out;
    reg [N-1:0] Q;
    wire [N-1:0] NS;
    assign NS =
        (ld) ? X : {Q[0], Q[N-1:1]};
    always @ (posedge clk)
        Q <= NS;
    assign out = Q[0];
endmodule
```

Parameters give us a way to generalize our designs. A module becomes a “generator” for different variations. Enables design/module reuse. Can simplify testing.

```
module ParToSer(ld, X, out, CLK);
    input [N-1:0] X;
    input ld, clk;
    output out;
    reg [N-1:0] Q;
    wire [N-1:0] NS;
    assign NS =
        (ld) ? X : {Q[0], Q[N-1:1]};
    always @ (posedge clk)
        Q <= NS;
    assign out = Q[0];
endmodule
```

Replace all occurrences of "3" with "N-1".

```
parameter N = 4;
```

Declare a parameter with default value.
Note: this is not a port. Acts like a “synthesis-time” constant.

```
ParToSer #(N(8))
ps8 ( ... );
```

```
ParToSer #(N(64))
ps64 ( ... );
```

Overwrite parameter N at instantiation.

```
ParToSer #(.N(8))
ps8 ( ... );
```

```
ParToSer #(.N(64))
ps64 ( ... );
```

Overwrite parameter N at instantiation.
Main Points

• All synchronous systems = CL + state registers
• basic rising edge triggered FF timing
• simple FSM: state transition diagram ➔ Verilog