CMPE 415
Suggested Coding and Design Practices
Prof. Ryan Robucci

Some examples and discussion are cited from Clifford E. Cummings' papers
http://www.sunburst-design.com/papers
Data Pipeline and RTL

- General single-clock domain register-transfer-logic structure to synthesize (right):
- In general a synthesizer approaches synthesis one clock domain at a time:
  - According to their clock inputs, sequential gates connected to a given clock and the combinatorial gates between them form the gates within a clock domain
  - Logic optimizers work to simplify logic within a domain and must satisfy timing constraints like setup and hold with additional allowance for clock jitter and clock skew between gates
  - Implementation also requires routing clock signals through special clock networks that minimize skew
- You should first learn style that supports a single-clock domain synthesis and causes PRE-SYNTHESIS functional simulation results to match results from post-synthesis timing and ultimately hardware
  - Can expand understanding to multiple clock domains with cross-domain paths, multi-cycle paths, asynchronous inputs and external connections later
A) Use non-blocking statements to assign outputs of sequential gates
B) Use blocking statements to assign outputs of combinatorial gates
C) Avoid unnecessary latches when coding for FPGAs, but when necessary use non-blocking statements
   • Latches fall outside the scope of basic synchronous timing analysis assumptions and complicate timing and functional testing.
D) Do not make assignments to the same variable from more than one always block
   • It confuses synthesizers and can lead to non-deterministic simulation behavior if multiple assignments are made at the same point in time
E) Consider what each output bit is assigned for any possible evaluation under any input and sequential state condition
   • When mixing sequential and combinatorial code, it is easy to get into the mindset of coding for sequential logic outputs (which commonly defaults to retaining previous values in many cases) and overlook what happens with the intermediate combinatorial logic in all cases
   • When coding sequential code missed cases can lead to unintended enables, storing and use of old data
F) Do not use any delay operator for the purpose of affecting synthesis

Related guidelines:
**Rule: No Continuous Feedback**

For synthesizable code, continuous assignments and logic should not have feedback, this would represent combinatorial circuit feedback (which has hysteresis / memory)

*Simulation-only / modeling code* can combinatorial feedback only if delays are added

**Primitives**

```plaintext
wire y, a, b, c;
and (y, a, b, c)
```

**Continuous assignments**

```plaintext
wire y, a, b, c
assign y = a & b & c;

wire y, d0, d1, sel
assign y = sel ? d1 : d0;
```

**Feedback**

```plaintext
assign y = y & a & b & c;
assign y = a & b;
assign b = y & a;
assign y = (en & x) | (~en & y);
assign y = en ? x : y;
nor n1(q, r, q_n);
nor n1(q_n, s, q);
assign q = ~(r | q_n);
assign q_n = ~(s | q);
```

*SR Latch*
A basic register

```
reg q;
always @(posedge clk)
begin
    q <= d;
end
```

The sensitivity list includes only a timing control signal, the clock. Output is only updated on clock edges even if d is changing, this requires memory.
### Synchronous reset

```verilog
reg q;
always @(posedge clk)
  if(reset)
    q <= 0;
  else
    q <= d;
```

The sensitivity list only includes the clock, indicating that the set and clear only propagate to the output upon the rising clock edge.

### Synchronous reset (active low)

```verilog
reg q;
always @(posedge clk)
  if(reset_n)
    q <= 0;
  else
    q <= d;
```
Asynchronous reset

```verilog
class reg q;
always @(posedge clk, posedge clear)
  if (clear)
    q <= 0;
  else
    q <= d;
```

- The sensitivity list includes the clear signal, indicating that the clear should propagate to the output immediately. The edge specifier in the sensitivity list is required by some synthesizers, though not required for simulation.

Asynchronous reset (active low)

```verilog
class reg q;
always @(posedge clk, negedge clear_n)
  if (~clear_n)
    q <= 0;
  else
    q <= d;
```

- As a general rule, review the synthesis tool's documentation regarding recommended coding templates. Some synthesizers are more flexible than others. Some may, for instance, require that the reset condition be handled first with an if else construct as shown.
Register Synthesis: Resets

- For FPGAS, often asynchronous resets are more efficient than synchronous resets since the inbuilt technology registers often have async. resets already on them, while synchronous resets would involve additional logic like an AND gate in the input data path.

- Synthesizers may require conforming to code templates and require that async. Reset be handle by if statements immediately following trigger statement.

```verilog
always @ (posedge clk, negedge clr_n)
  if (clr_n==0)...
  else ...
```

```verilog
always @ (posedge clk)
  if (reset_n==0)...
  else...
```

Note use of negedge.
Registered-Output Logic

Combinatorial Only includes all inputs:

```vhdl
reg y;
always @(a, b)  //all comb. dependencies listed
  y = a & b;
```

Could also have used y<= a & b; but we follow the practice of using blocking assignments for all combinatorial logic

Sequential (registered-output combinatorial logic):

```vhdl
reg q;
always @(posedge clk)
  q <= a & b;
```
module fbosc1 (y1, y2, clk, rst);
output y1, y2;
input clk, rst;
reg y1, y2;

always @(posedge clk or posedge rst)
  if (rst) y1 = 0; // reset
  else y1 = y2;

always @(posedge clk or posedge rst)
  if (rst) y2 = 1; // preset
  else y2 = y1;
endmodule

This code attempts to model a swap of y1 and y2.
Timing of execution of parallel always blocks is not guaranteed
in simulation – though synthesis will probably work since
  synthesis approaches each always blocks somewhat independently at first.
Simulation of parallel blocks

Which one first? Does it even matter?

always @(posedge clk, posedge rst)
if (rst) y2 = 1; // preset
else y2 = y1;

always @(posedge clk, posedge rst)
if (rst) y1 = 0; // reset
else y1 = y2;
module fbosc2 (y1, y2, clk, rst);
    output y1, y2;
    input clk, rst;
    reg y1, y2;

    always @(posedge clk or posedge rst)
        if (rst) y1 <= 0; // reset
        else y1 <= y2;

    always @(posedge clk or posedge rst)
        if (rst) y2 <= 1; // preset
        else y2 <= y1;
endmodule
module amb_parallel_swap();

    reg clk, rst;
    reg y1, y2;
    reg z1, z2;

initial clk = 0;
always #50 clk = ~clk;

initial begin
    rst = 1;
    #10;
    rst = 0;
end

initial begin
    #1000 $finish;
end

always @(posedge clk , posedge rst) 
    if (rst) y1 = 0; // reset
    else y1 = y2;

always @(posedge clk , posedge rst) 
    if (rst) y2 = 1; // preset
    else y2 = y1;

always @(posedge clk , posedge rst) 
    if (rst) z1 <= 0; // reset
    else z1 <= z2;

always @(posedge clk , posedge rst) 
    if (rst) z2 <= 1; // preset
    else z2 <= z1;
endmodule
**Intentional Pipeline for Timing**

What if timing requirement is not satisfied?

Critical Path Timing requirement:

\[ T_{CLK_TO_Q} + PD + T_{setup} < T_{clk} \]

```verilog
module calc(q, a, b, c, d, clk);
    output q;
    input a, b, c, d;
    input clk;
    reg [31:0] q;
    reg [31:0] tmp1;
    reg [31:0] tmp2;
    always @(posedge clk) begin
        tmp1 = a*b;
        tmp2 = c*d;
        q = tmp1 * tmp2;
    end
endmodule
```
• Can reduce the clock speed
  – But this slows the entire system

• Can introduce pipelining
  – Overall propagation of computation is longer (two clock cycles incurring multiple setup and hold times)
  – Maintains fast system clock

• When timing is satisfied, introducing pipelining in the critical path of a system can allow an increase the clock rate and therefore overall system throughput
module calc(q, a, b, c, d, clk);
output q;
input a, b, c, d;
input clk;
reg [31:0] q;
reg [31:0] tmp1;
reg [31:0] tmp2;
always @(posedge clk) begin
  tmp1 <= a*b;
  tmp2 <= c*d;
  q <= tmp1*tmp2;
end
endmodule
module pipeb2 (q3, d, clk);
output [7:0] q3;
input [7:0] d; input clk;
reg [7:0] q3, q2, q1;
always @(posedge clk) begin
    q3 = q2;
    q2 = q1;
    q1 = d;
end
endmodule

module pipeb1 (q3, d, clk);
output [7:0] q3;
input [7:0] d; input clk;
reg [7:0] q3, q2, q1;
always @(posedge clk) begin
    q1 = d;
    q2 = q1;
    q3 = q2;
end
endmodule
Bad Parallel Block Pipeline Implementations

```verilog
module pipeb3 (q3, d, clk);
  output [7:0] q3;
  input [7:0] d;
  input clk;
  reg [7:0] q3, q2, q1;
  always @(posedge clk) q1=d;
  always @(posedge clk) q2=q1;
  always @(posedge clk) q3=q2;
endmodule

module pipeb4 (q3, d, clk);
  output [7:0] q3;
  input [7:0] d;
  input clk;
  reg [7:0] q3, q2, q1;
  always @(posedge clk) q2=q1;
  always @(posedge clk) q3=q2;
  always @(posedge clk) q1=d;
endmodule
```

These may synthesize correctly, but simulation may not match
Good Pipeline Implementations

Use non-blocking statements for registers

module pipen1 (q3, d, clk);
    output [7:0] q3;
    input [7:0] d;
    input clk;
    reg [7:0] q3, q2, q1;
    always @(posedge clk) begin
        q1 <= d;
        q2 <= q1;
        q3 <= q2;
    end
endmodule

module pipen2 (q3, d, clk);
    output [7:0] q3;
    input [7:0] d;
    input clk;
    reg [7:0] q3, q2, q1;
    always @(posedge clk) begin
        q3 <= q2;
        q2 <= q1;
        q1 <= d;
    end
endmodule

module pipen3 (q3, d, clk);
    output [7:0] q3;
    input [7:0] d;
    input clk;
    reg [7:0] q3, q2, q1;
    always @(posedge clk) q1<=d;
    always @(posedge clk) q2<=q1;
    always @(posedge clk) q3<=q2;
endmodule

module pipen4 (q3, d, clk);
    output [7:0] q3;
    input [7:0] d;
    input clk;
    reg [7:0] q3, q2, q1;
    always @(posedge clk) q2<=q1;
    always @(posedge clk) q3<=q2;
    always @(posedge clk) q1<=d;
endmodule

Order doesn't matter
Cascading Combinatorial Logic
Ex: AND-OR

Guideline: When modeling combinatorial logic with an always block, use blocking assignments.

- Works, but requires multiple passes in simulation
- Efficient sim

module ao4 (y, a, b, c, d);
output y;
input a, b, c, d;
reg y, tmp1, tmp2;
always @(a or b or c or d) begin
    tmp1 <= a & b;
    tmp2 <= c & d;
    y <= tmp1 | tmp2;
end
endmodule

module ao2 (y, a, b, c, d);
output y;
input a, b, c, d;
reg y, tmp1, tmp2;
always @(a or b or c or d) begin
    tmp1 = a & b;
    tmp2 = c & d;
    y = tmp1 | tmp2;
end
endmodule

module ao5 (y, a, b, c, d);
output y;
input a, b, c, d;
reg y, tmp1, tmp2;
always @(a,b,c,d,tmp1,tmp2) begin
    tmp1 <= a & b;
    tmp2 <= c & d;
    y <= tmp1 | tmp2;
end
endmodule
Mixing Comb. and Sequential Example: xor-DFF

module nbex1 (q, a, b, clk, rst_n);
output q;
input clk, rst_n;
input a, b;
reg q, y;
always @(a or b)
  y = a ^ b;
always @(posedge clk or negedge rst_n)
  if (!rst_n) q <= 1'b0;
  else q <= y;
endmodule

module nbex2 (q, a, b, clk, rst_n);
output q;
input clk, rst_n;
input a, b;
reg q;
always @(posedge clk or negedge rst_n)
  if (!rst_n) q <= 1'b0;
  else q <= a ^ b;
endmodule
module ba_nba2 (q, a, b, clk, rst_n);
output q;
input a, b, rst_n;
input clk;
reg q;
always @ (posedge clk or negedge rst_n) begin: ff
  reg tmp;
  if (!rst_n)
    q <= 1'b0;
  else begin
    tmp = a & b;
    q <= tmp;
  end
end
endmodule

Mixing Comb and Sequential Example: xor-DFF
Use of local declarations for temporary and trimmed signals

*WARNING:Xst:646 - Signal <ff/tmp> is assigned but never used. This unconnected signal will be trimmed during the optimization process.
DFF: Poor and possibly unsupported style

Mix of blocking and non-blocking To same variable.

module ba_nba6 (q, a, b, clk, rst_n);
output q;
input a, b, rst_n;
input clk;
reg q, tmp;
always @(posedge clk or negedge rst_n)
if (!rst_n)
  q = 1'b0; // blocking assignment to "q"
else begin
  tmp = a & b;
  q <= tmp; // nonblocking assignment to "q"
end
endmodule
Assignments to the same variable from multiple always blocks

These blocks are make mutually exclusive assignments
May make sense. May sim, but synth. usually complains of multiple drivers.
Swapping Example

Non-blocking

```verilog
always @(posedge clk , posedge rst) begin
    if (rst) begin
        z1 <= 0; // reset
    end else begin
        z1 <= z2;
        z2 <= z1;
    end
end
```

Blocking

```verilog
always @(posedge clk , posedge rst) begin: swap
    reg temp;
    if (rst) begin
        y1 <= 0; // reset
    end else begin
        temp = y1;
        y1 <= y2;
        y2 <= temp;
    end
end
```

Order doesn't matter

temp won’t exist in the synthesized design
Guideline: Use non-blocking for EVERY register

It is better to develop the habit of coding all sequential always blocks, even simple single-block modules, using nonblocking assignments as shown in Example 14.

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

module dffx (q, d, clk, rst);
  output q;
  input d, clk, rst;
  reg q;
  always @(posedge clk)
    if (rst) q <= 1'b0;
    else q <= d;
endmodule
```
General warning for Sloppy combinations

Be very careful to consider every output for every path in the decision tree.
module arith (output reg q, input a, input b, input clk);
    always @(posedge clk) begin: blk
        reg asq, bsq;
        if (a>b) begin
            asq=a*a;
            bsq=b*b;
            q <= asq + bsq;
        end else begin
            q <= asq - bsq;
        end
    endmodule
module arith (output reg q, input a, input b, input clk);
    always @(posedge clk) begin: blk
        reg asq, bsq;
        if (a>b) begin
            q <= asq + bsq;
        end else begin
            q <= asq - bsq;
        end
        asq = a*a;
        bsq = b*b;
    end
endmodule
Combinatorial

module arith (output reg q, input a, input b);
    always @(a,b) begin: blk
        reg asq,bsq;
        q = asq + bsq;
        asq = a*a;
        bsq = b*b;
    end
endmodule

Registered Logic – an attempt to only add a registered output

module arith (output reg q, input a, input b, input clk);
    always @(posedge clk)
    begin: blk
        reg asq,bsq, q_pre;
        q_pre = asq + bsq;
        asq = a*a;
        bsq = b*b;
        q <= q_pre;
    end
endmodule
module andReg(q, a, b, clk, rst_n);
  output q;
  input a, b, rst_n;
  input clk;
  reg q, tmp;
  always @(posedge clk , negedge rst_n)
    if (~rst_n)
      q <= 1'b0;
    else begin
      tmp <= a & b;
      q <= tmp;
    end
endmodule

Non-blocking assignment breaks our rules
Multiple Clock Domains

As a beginner, avoid the creation of additional clock domains caused by using various signals as a clock

<table>
<thead>
<tr>
<th>Avoid use of a logic signal as a clock</th>
<th>Synchronous logic</th>
</tr>
</thead>
</table>
| ```
assign gt = a>b;
reg q;
always @(posedge gt)
begin
  a <= b;
end
``` | ```
reg q;
always @(posedge clk)
begin
  if a>b:
    a <= b;
end
``` |

A synthesizer may wish to partition its task by organizing logic into clock domains, optimizing the logic within them, and then performing timing analysis (e.g. critical path propagation delay, setup time and hold time checks) within the domain and attempt to handle signals that cross clock domains. Furthermore, FPGAs use a special hardware routing network for clocks that is distinct from general logic signals. Creation of a additional clock domains requires care and should be avoided at this time.
No Double-Edge Clocking in this Course

always @(posedge clk) 
always @(negedge clk) 
always @(posedge clk)

clk

b
c
d
q
Gated Clocks

- Gated clocks avoid unnecessary switching in a system and reduce power.

- Gating introduces clock skew between parts, complicating timing by introducing **additional clock domains**.

- It is best to use special hardware to create additional clocks which have not yet been taught.

- For now, consider it safer to implement an enable.

- For those that want to read ahead:  
  http://www.xilinx.com/support/answers/38099.html
Gated Clock Issues

- `clk_gated` is delayed from `clk`
  - Increased opportunity for a race condition from register A to register B
  - Increase delay from clk edge to updated Q on B
    registered reduces allowed propagation time in logic on path2

- Timing analysis and tools must be able to account for this but timing on combinatorial paths are not as predictable
Gated Clock

```verilog
module gated_clock (clk, reset_, clk_gate, data, Q);
    input clk, reset_, clk_gate, data;
    output Q;
    reg Q;
    wire clk_gated = clk && clk_gate;
    always @(posedge clk_gated, negedge reset_)
        if (reset_==0) Q<=0; else Q<= data;
end
endmodule
```
Gated Clock Functionality using Enable

module not_gated_clock (clk, reset_, data_gate, data,Q);
  input  clk, reset_,data_gate,data;
  output Q;
  reg Q;
  always @ (posedge clk or negedge reset_)
    if (reset_==0) Q<=0; else if (data_gate) Q<= data;
    //else assignment to previous value is inferred
end
endmodule