How to Design Complex Digital Systems

Dr. Paul D. Franzon

Outline

1. Steps in an organized design approach
2. Finite State Machines
2. Achieving performance and efficiency
4. Example

References

1. Smith & Franzon, Chapter 10
Steps in High Level Design

Clearly Separate Control from DataPath

1. Determine MicroOperations to be performed on datapath units
   - e.g. adds, subtracts, multiplies, memory references, etc.

2. Design datapath units to perform these operations efficiently
   - Design to RTL level
   - Note later sections on efficiency

3. Identify control points
   - Control lines
   - Status lines

4. Determine reset/start/stop/transition actions
   - Especially global reset strategy

5. Determine control sequence
   - Generally MicroOp sequence required to perform overall task
   - Gives sequence of control events and status line responses

6. Determine control strategy
   - Mix of FSMs and/or counters

7. Verify before coding
Reset

- Reset is a global signal that the designer cannot modify.
- It is generally asserted on power up or a "hard" reset.
- It is used to start the machine in a "known" state.
- Thus it must be distributed to:
  - All FSMs
  - Selected counters
  - Selected status registers
Finite State Machines can be classified by the following attributes:

- **Moore or Mealy type outputs**

**Moore Outputs**
Outputs depend solely on state vector (generally, a Moore FSM is the simplest to design)

**Mealy Outputs**
Outputs depend on inputs and state vector (only use if it is significantly smaller or faster)
... FSM Types

- **State Vector Encoding**
  - Minimal encoding
    - Minimum number of bits
  - Minimum, sequential encoding
    - Minimum number of bits and states in sequence
      - Does not necessarily optimize ‘next state logic’ size
  - Gray encoding
    - state bit changes by only one bit between sequential states
      - Minimizes switching activity in state vector register
  - One-hot encoding
    - one bit per state
      - usually gives fastest ‘next state’ logic

**Example:** 7-state FSM, states S0 … S7:
… FSM Types

• **Resets:**
  - Reset usually occurs only on power-up and when someone hits the ‘reset’ button
  - **Asynchronous** reset:
    - FSM goes to reset state whenever reset occurs
  - **Synchronous** reset:
    - FSM goes to reset state on the next clock edge after reset occurs
  - Asynchronous reset leads to smaller flip-flops while synchronous reset is ‘safer’ (noise on the reset line is less likely to accidently cause a reset).

• **Fail-Safe Behavior:**
  - If the FSM enters an ‘illegal’ state due to noise is it guaranteed to then enter a legal state?
  - ‘Yes’ is generally desirable
… FSM Types

• **Sequential Next state or output logic**
  - Usually, these blocks are combinational logic only
  - However, can place sequential logic (e.g. a counter, or a toggle-flip-flop) in these blocks if it is advantageous
  - **AVOID DOING THIS AS MUCH AS YOU CAN UNLESS YOU ARE REALLY SURE ABOUT WHAT YOU ARE DOING**
    - *Sequential next state or output logic can get very confusing to design and debug*
  - **Registered or Unregistered Outputs**
    - Do not register the outputs unless you need to ‘deglitch’ the outputs (for example, for asynchronous handshaking - combinational logic has to be assumed to be glitchy) or are pipelining the control
    - *e.g.*

![Diagram](Next State Logic) → ![State Vector] → ![Output Logic] → ![reg_out]
Example - Drag Racing Lights

At the start of a new race (‘car’), go through the Red-Yellow-Green sequence:

Moore Machine:

Nomenclature: inputs
car?
On states: red yellow green

Mealy Machine:

Nomenclature: inputs / outputs
car? / red yellow green
module traffic_light_controller (clock, reset, car, red, yellow, green);

input clock;
input reset;
input car;
output red, yellow, green;

parameter [1:0] // synopsys enum states
S0 = 2'b00,
S1 = 2'b01,
S2 = 2'b10,
S3 = 2'b11;

reg [1:0] /* synopsys enum states */ current_state, next_state;// synopsys state_vector current_state
reg red, yellow, green;

/*------- Sequential Logic ----*/
always@(posedge clock or negedge reset)
if (!reset)   current_state <= S0;else  current_state <= next_state;

/* next state logic and output logic */
always@(current_state or car)
begin
  red = 0; yellow = 0; green = 0; /* defaults to prevent latches */
  case (current_state) // synopsys full_case parallel_case
    S0: begin
      red = 1;
      if (car) next_state = S1
      else next_state = S0;
    end
    S1: begin
      yellow = 1;
      next_state = S2;
    end
    S2 : begin
      green = 1;
      next_state = S0;
    end
default: next_state = S0;
  endcase
end
endmodule


**FSM Verilog Notes**

1. Code each FSM by itself in one module.
2. Separate Sequential and Combinational Logic
3. Is this reset Synchronous or Asynchronous?

   - Asynchronous usually results in less logic (reset is actually synchronized when it enters the chip).

4. Note use of Synthesis directives:
   - `//synopsys enum states` and `//synopsys state_vector current_state` tell Synopsys what the state vector is.
     - You can optionally use Synopsys FSM optimization procedures
   - Why can we state `//synopsys full_case parallel_case` for FSMs?

5. How to we prevent accidently inferring latches?
**FSM State Encoding Options**

Can either do `by hand` in Verilog source code or by reassigning states in Synopsys:

- **Binary or Sequential (minimal) encoding:**
  
  State 0 = 000
  State 1 = 001, etc.

- **Gray encoding gives the minimum change in the state vector between states:**
  
  State 0 = 000
  State 1 = 001
  State 2 = 011, etc

  - Reduces state transition errors caused by asynchronous inputs changing during flip-flop set-up times.
  - Minimizes power consumed in state vector flip-flops

  Synopsys: `set_fsm_encoding_style gray` //+ See manual

- **One-hot encoding assigns one flip-flop per state:**
  
  State 0 = 0001
  State 1 = 0010
  State 2 = 0100, etc

  - Fastest but largest design

  Synopsys: `set_fsm_encoding_style one_hot`

- **Custom:** Assign states by hand in Verilog of Synopsys
Registering FSM Outputs

Sometimes useful to **register** the outputs of FSMs:

- Necessary when these outputs are interfacing asynchronously with other modules or off-th-chip
  - e.g. RAS and CAS outputs for a memory interface
- Useful if delays in output combinational logic are making it hard to meet timing requirements in the module they are connected to.
  - Assumes flip-flop $t_{cp\_Q}$ is faster (might not be - look in library sheets)

**e.g.**
```
always@(posedge clock)
begin
  red  <= int_red;
  yellow <= int_yellow;
  green <= int_green;
end
...  
```

```
case (current_state)
  S0: begin int_red=1;
  ...  
```

- Note: changes now delayed one clock when compared with previous version
Tricks to Achieving Performance

• Performance Metrics
  - Throughput (operations/second)
  - Latency (seconds from start to finish of operation)
  - Performance/Area
  - Performance/Power

• Exploit potential parallelism of hardware
  - Parallel vs. Serial operation
  - Pipelining to speed up serial operations that can not be parallelized (or don’t need parallelization)

Theoretically, a bit-serial machine with one flop between each bit operator is the most area-performance efficient architecture
  - Real-world constraints usually take you away from such a parallel/pipelined solution
Improving Performance

• Watch those memory references
  • Effective memory bandwidth usually determines overall performance
    ◆ Strive to maximize utilization of memory channels
    ◆ Memories are usually large
      ➔ Increasing # of other functional units to keep memory 100% busy is usually a small area hit in comparison
    ◆ For DRAM maximize use of burst/page modes
  • Keep an eye out for algorithmic techniques to improve performance, for example
    ◆ Search algorithms that minimize memory accesses
      ➔ Might have to trade memory size or constrain organization
    ◆ shift instead of */2
    ◆ etc.
Implementation Efficiency

- Look for opportunities to share resources

- Look for opportunities to shave clock cycles
  - Watch critical path though

- Avoid large FSMs
  - Can be slow
  - Partition into smaller FSMs instead

- Watch out for those ‘large units’
  - On-chip memories, floating point, multiply, divide, etc.

- Seek Simplicity
THINK HARDWARE

Improve performance by:

- avoiding unnecessary priority structures in logic
- optimizing logic for late-arriving signals
- structuring arithmetic for performance
- avoiding area-inefficient code
- avoiding high fanout signals
- pipelining for high performance
- exploiting high performance cores from Vendors
Example

Motion Estimator

Task:

- Detect blocks of video data in successive frames that are related only via a translation
  - Digital Video is captured as blocks of 16x16 pixels
  - Want to determine if block has moved largely unchanged
    - If true can transmit motion vector rather than block
    - Permits high level of compression
- Example (4x4 block)

Reference Block in Frame 1

“Draw block” with motion vector (1,2) in frame 2
Search Algorithm

Describe for 16x16 reference block:
1. Move a window the size of the reference block over search space in the second frame
2. For each window location \((i,j)\) determine the distortion vector

\[
D(i, j) = \sum_{m=0}^{15} \sum_{n=0}^{15} | r_{m,n} - S_{m+i,N+j} |
\]

3. Maintain the best distortion and appropriate motion vector produced so far.

For Example (4x4 block):

Search Block Location \((i,j) = (-3,3)\)

\(D = 3\) (3 pixels different in this B&W example)
System Requirements:

- 16x16 Reference Block
- 31x31 Search Window
- Each stored in one two-read-ported memory
  - In reality one memory per frame
- Grey-scale coded pixels (8 bits/block)
- 4096 reference blocks in a frame
- Conduct search at 15 frames per second
  - (Encoding does not have to be real time)
- Clocks available: 130, 260 MHz
- 0.25 μm CMOS library
Step 1: System Design

Elements to thinking:

- **Bottom-up design**
  - Determine critical bottlenecks (paths & other bottlenecks)

- **Top-down design**
  - Determine use of pipelining and parallelism to meet performance constraints

Critical Bottlenecks:

- **Elemental Arithmetic Operation (add-accumulate):**

  \[ D = D + | r_{mn} - S_{m+i,N+j} | \]

  - Design, synthesize ➔ Can operate at 260 MHz with some timing margin left over

- **Memories:**
  - Single access per clock cycle
... System Design

Top Down Design

- Number of add-accumulates per clock cycle:
  - 4096 blocks per 1/15 of a second
  - \((31-15) \times (31-15) = 256\) searches/block
  - \(16 \times 16 = 256\) add-accumulates per search
  - \(\Rightarrow 4096 \times 15 \times 256 \times 256 = 4.027 \times 10^9\) add-accumulates/second
  - At 260 MHz \(\Rightarrow\) At least 16 adders in parallel \((4027/260=15.5)\)

- Searches/block \([(4x4) on (10x10) example]\):
  - 7 searches per column
  - 7 searches per row
  - \((10-4) \times (10-4)\) total searches
First Attempt

- Assign one search per Accumulator

\[
\begin{array}{c}
\text{R mem} \\
\text{S mem} \\
\text{Accum} \\
\text{Accum} \\
\text{Accum} \\
\text{Accum} \\
\end{array}
\]

Vector: \((-8,-8)\) \((-8,-7)\) \((-8,-6)\) \((-8,-5)\) ..... 

Cycle

1. \(r_{0,0} - S_{0,0}\) \(r_{0,0} - S_{0,1}\) \(r_{0,0} - S_{0,2}\) \(r_{0,0} - S_{0,3}\) ..... 

2. \(r_{0,1} - S_{0,1}\) \(r_{0,1} - S_{0,2}\) \(r_{0,1} - S_{0,3}\) \(r_{0,1} - S_{0,4}\) .....
Second Attempt:

- Stagger Startup of Accumulators

\[
\begin{array}{cccc}
\text{R mem} & \text{S mem} \\
\text{Accum} & \text{Accum} & \text{Accum} & \text{Accum} \\
\text{Vector:} & (-8,-8) & (-8,-7) & (-8,-6) & (-8,-5) \\
\text{Cycle} & 1 & |r_{0,0}-S_{0,0}| & \cdots & \cdots \\
2 & |r_{0,1}-S_{0,1}| & |r_{0,0}-S_{0,1}| & \cdots & \cdots \\
3 & |r_{0,2}-S_{0,2}| & |r_{0,1}-S_{0,2}| & |r_{0,0}-S_{0,2}| & \cdots & \cdots \\
4 & |r_{0,3}-S_{0,3}| & |r_{0,2}-S_{0,3}| & |r_{0,1}-S_{0,3}| & |r_{0,0}-S_{0,3}| & \cdots & \cdots \\
\end{array}
\]
... System Design

Final Solution:
- Pipeline R

Vector: (-8, -8) (-8, -7) (-8, -6) (-8, -5) ..... 

Cycle
1  | r_{0,0}-S_{0,0} | ..... 
2  | r_{0,1}-S_{0,1} | | r_{0,0}-S_{0,1} | ..... 
3  | r_{0,2}-S_{0,2} | | r_{0,1}-S_{0,2} | | r_{0,0}-S_{0,2} | ..... 
4  | r_{0,3}-S_{0,3} | | r_{0,2}-S_{0,3} | | r_{0,1}-S_{0,3} | | r_{0,0}-S_{0,3} | ..... 

2 S mem ports required 

15 | r_{0,15}-S_{0,15} | | r_{0,14}-S_{0,15} | | r_{0,13}-S_{0,15} | | r_{0,12}-S_{0,15} | 
16 | r_{1,1}-S_{1,1} | | r_{0,15}-S_{0,16} | | r_{0,14}-S_{0,16} | | r_{0,13}-S_{0,16} |
Step 2: Design Datapath

Datapath Details:

- Detailed hardware required to implement above

PE = Processing Element

R mem

S mem

|A-B|

+ 

> 

To comparator
… Datapath

Comparator:

PEout

PEready

Vectorx

Vectory

motionX

motionY

BestDist

Peout < BestDist?
Coding Datapath

**PE:** Note, accumulator cant overflow – saturate at FF

```verilog
module PE (clock, R, S1, S2, S1S2mux, newDist, Accumulate, Rpipe);
    input clock;
    input [7:0] R, S1, S2; // memory inputs
    input S1S2mux, newDist; // control inputs
    output [7:0] Accumulate, Rpipe;
    reg [7:0] Accumulate, AccumulateIn, Difference, Rpipe;
    reg Carry;

    always @(posedge clock) Rpipe <= R;
    always @(posedge clock) Accumulate <= AccumulateIn;

    always @(R or S1 or S2 or S1S2mux or newDist or Accumulate)
        begin // capture behavior of logic
            difference = R - S1S2mux ? S1 : S2;
            if (difference < 0) difference = 0 - difference;
            // absolute subtraction
            {Carry, AccumulateIn} = Accumulate + difference;
            if (Carry == 1) AccumulateIn = 8'hFF; // saturated
            if (newDist == 1) AccumulateIn = difference;
            // starting new Distortion calculation
        end
endmodule
```

Motion Estimator Processing Element (PE).
module Comparator (clock, CompStart, PEout, PErady, vectorX, vectorY, BestDist, motionX, motionY);

input clock;
input CompStart; // goes high when distortion calculations start
input [8\*16:0] PEout; // Outputs of PEs as one long vector
input [15:0] PErady; // Goes high when that PE has a new distortion
input [3:0] vectorX, vectorY; // Motion vector being evaluated
output [7:0] BestDist; // Best Distortion vector so far
output [3:0] motionX, motionY; // Best motion vector so far
reg [7:0] BestDist, newDist;
reg [3:0] motionX, motionY;
reg newBest;

always @(posedge clock)
  if (CompStart == 0) BestDist <= 8`hFF; //initialize to highest value
  else if (newBest == 1)
    begin
      BestDist <= newDist;
      motionX <= vectorX;
      motionY <= vectorY;
    end

always @(BestDist or PEout or PErady)
  begin
    newDist = PEout [PEready*8+7 : PErady*8];
    if (((PEready == 0) || (start == 0)) newBest = 0; // no PE is ready
    else if (newDist < BestDist) newBest = 1;
    else newBest = 0;
  end

endmodule

Comparator Module.
Step 3. Identify Control Points

PE control lines:
S1S2mux [15:0]; // IIS1-S2 mux control
NewDist [15:0]; // =1 when PE is starting a new distortion calculation

Comparator control lines:
CompStart;11 = // when PEs running
PEready [15:0]; // PEready[I]=1 when PEi has a new distortion vector
VectorX [3:0];
VectorY [3:0]; // Motion vector being evaluated

Memory control lines:
• Memories organized in row-major format
  • e.g. R(3,2) is stored at location 3*15+2-1 = 46
AddressR [7:0]; // address for Reference memory (0,0). ..(15,15)
AddressS1 [9:0]; // address for first read port of Search mem
AddressS2 [9:0]; // second read port of Search mem (0,0)-(30,30)
Step. 4 Design Controller

Best Strategy : Counter

module control (clock, start, S1S2mux, NewDist, CompStart, PEready, VectorX, VectorY, AddressR, AddressS1, AddressS2);
input clock;
input start; // = 1 when ‘going’
output [15:0] S1S2mux;
output [15:0] NewDist;
output CompStart;
output [15:0] PEready;
output [3:0] VectorX, VectorY;
output [7:0] AddressR;
output [9:0] AddressS1, AddressS2;
reg [15:0] S1S2mux;
reg [15:0] NewDist;
reg CompStart;
reg [15:0] PEready;
reg [3:0] VectorX, VectorY;
reg [7:0] AddressR;
reg [9:0] AddressS1, AddressS2;
reg [12:0] count;
reg completed;
type integer i;

always @(posedge clock)
if (start == 0) count <= 12’b0;
else if (completed == 0) count <= count + 1’b1;
always @(count)
begin
for (i=0; i<15; i = i+1)
begin
NewDist[i] = (count[7:0] == i);
PEready[i] = (NewDist[i] && ! (count < 8’d256));
S1S2mux[i] = (count[3:0] > i);
end
AddressR = count[7:0];
VectorX = count[3:0] - 4’d7;
VectorY = count[11:8]>>4 - 4’d7;
complete = (count = 4’d16 * (8’d256 + 1));
end
endmodule

Reset Strategy

- Reset needed to intialize entire chip in known state
  - Does not apply here, as long as “start” comes from a unit that does use a reset
Conclusions

What are my three “mantras”

What are common control strategies?

What are common speed-up strategies?

What is “reset” for?