Outline
1. Steps in an organized design approach
2. Finite State Machines
2. Achieving performance and efficiency 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 Machine Types**

Finite State Machines can be classified by the following attributes:

- **Moore or Mealy type outputs**
  
  ![Diagram of Moore and Mealy FSMs](Diagram)

  - **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:

<table>
<thead>
<tr>
<th>State</th>
<th>Min.</th>
<th>Gray</th>
<th>1-hot</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>000</td>
<td>000</td>
<td>0000001</td>
</tr>
<tr>
<td>S1</td>
<td>001</td>
<td>001</td>
<td>0000010</td>
</tr>
<tr>
<td>S2</td>
<td>010</td>
<td>011</td>
<td>0000100</td>
</tr>
</tbody>
</table>

... 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](https://example.com/diagram.png)

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

![Diagram](https://example.com/diagram.png)
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 interring 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)
begnin
  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](image1.png) ![“Draw block” with motion vector (1,2) in frame 2](image2.png)

**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 window in frame 2](image3.png)
System Requirements

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):

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

• 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
  - \(4096 \times 15 \times 256 = 4.027 \times 10^9\) add-accumulates/second
  - At 260 MHz, at least 16 adders in parallel \((4027/260 = 15.5)\)
- Searches/block [(4x4) on (10x10) example]:

![Diagram showing the top-down design process and calculations](image)

**First Attempt**

- Assign one search per Accumulator

![Diagram showing the first attempt process and calculations](image)
Second Attempt:

- Stagger Startup of Accumulators

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

Cycle
1  | \( r_{0,0} \cdot S_{0,0} \) |
2  | \( r_{0,1} \cdot S_{0,1} \) | \( r_{0,0} \cdot S_{0,1} \) |
3  | \( r_{0,2} \cdot S_{0,2} \) | \( r_{0,1} \cdot S_{0,2} \) | \( r_{0,0} \cdot S_{0,2} \) |
4  | \( r_{0,3} \cdot S_{0,3} \) | \( r_{0,2} \cdot S_{0,3} \) | \( r_{0,1} \cdot S_{0,3} \) | \( r_{0,0} \cdot S_{0,3} \) |

Problem: 16-port R mem required.

But Notice!

R pattern
\( r_{0,0} \)

Final Solution:

- Pipeline R

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

Cycle
1  | \( r_{0,0} \cdot S_{0,0} \) |
2  | \( r_{0,1} \cdot S_{0,1} \) | \( r_{0,0} \cdot S_{0,1} \) |
3  | \( r_{0,2} \cdot S_{0,2} \) | \( r_{0,1} \cdot S_{0,2} \) | \( r_{0,0} \cdot S_{0,2} \) |
4  | \( r_{0,3} \cdot S_{0,3} \) | \( r_{0,2} \cdot S_{0,3} \) | \( r_{0,1} \cdot S_{0,3} \) | \( r_{0,0} \cdot S_{0,3} \) |

2 S mem ports required
Step 2: Design Datapath

Datapath Details:

- Detailed hardware required to implement above

PE = Processing Element

To comparator

Comparator:

PEout

PEready

Vectorx

Vectory

motionX

motionY

BestDist

Peout < BestDist?
Coding Datapath

PE: Note, accumulator can't overflow - saturate at FF

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

always @ (posedge clock) Rpipes <= 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).

Comparator Module.

```verilog
module Comparator (clock, CompStart, PEstOut, PEstReady, vectorX, vectorY, BestDist, newDist, motionX, motionY);
input clock;
input CompStart; // goes high when distortion calculations start
input [16:0] PEstOut; // Outputs of ED6 as one long vector
input [15:0] PEstReady; // Goes high when that PE has a new distortion input [2:0] 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] newDist, newEst;
reg [3:0] motionX, motionY;
reg newBest;

always @ (posedge clock)
if (CompStart == 1) BestDist <= 8'hFF; // initialize to highest value
else if (newEst == 1)
begin
 BestDist <= newDist;
 motionX <= vectorX;
 motionY <= vectorY;
 end
always @ (CompStart or PEstOut or PEstReady)
begin
 newDist = PEstOut [PEstReady? PEstReady: 0] ? (CompStart) newEst = 0; // no PE to ready
 else if (newDist < BestDist) newBest = 1;
 else newBest = 0;
 end
endmodule
```
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:
PEready [15:0]; // PEready[1]=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

```plaintext
module controller (clck, start, s1sel, s2sel, refdata, cmpdata, peout);
input clck, start, s1sel, s2sel, refdata, cmpdata, peout;
output [15:0] S1S2mux, NewDist, PEready, CompStart;
output [3:0] VectorX, VectorY;
output [7:0] AddressR, AddressS1, AddressS2;
reg [15:0] S1S2mux;
reg [15:0] NewDist;
reg PEready;
reg [3:0] VectorX, VectorY;
reg [7:0] AddressR, AddressS1, AddressS2;
reg [12:0] count;
reg completed;
integer i;

always @(posedge clck)
begin
  if (start == 0) count = 12'b0;
  else if (completed == 0) count = count + 1'b1;
end
always @(count)
begin
  if (count == 12'b10) begin
    if (cmpdata) count = 12'b0;
    else if (RefCount) count = N.
  end
  if (count == 12'b11) count = count + 1'b1;
end
endmodule
```

Reset Strategy
- Reset needed to initialize 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?