How to Design Complex Digital Systems

Dr. Paul D. Franzon

Outline
1. Steps in an organized design approach
2. Maximizing Performance or Efficiency
3. Example
4. C to Verilog
5. Minimizing Power Consumption

References
1. Smith & Franzon, Chapter 10
Objectives and Motivation

Objectives

- Describe a structured approach to partitioning and designing complex systems
- Practice it on an example
- Understand how C can be “translated” into Verilog
- Introduce influence of design on power consumption

Motivation

- Start teaching you the “art” of complex design through general principles and examples
Course “Mantras”

1. One clock, one edge, flip-flops only
2. Design BEFORE coding
3. Behavior implies function
4. Clearly separate control and datapath
Steps in High Level Design

1. Determine the algorithm
   - Code it in a high level language.
   - Consider the hardware algorithm – the native parallelism of hardware can lead you to a different solution than the software one

2. Explore the design
   - What types of micro-operations must be performed?
     - Eg. Multiply Accumulates; Memory references
   - What is the critical path and how fast can it reasonably go?
     - Synthesis exercises can be useful
   - What resources are needed to achieve the performance targets?
     - Eg. How much parallelism?
... Steps

3. Design the data path.
   - Down to Register Transfer level
   - Think HARDWARE while designing and try to be efficient
   - DRAW A TIMING DIAGRAM BY HAND
   - THEN convert to code

4. Identify the control and status lines.
   - Control lines come from the controller
   - Status lines come from the datapath

5. Identify the control approach needed (counters, FSMs, or pipelined control)

6. Implement the controller
   - Needs a good “timing” diagram:
     - Reset
     - Sequence of micro-operations to be performed
     - Critical transitions understood
Control Strategies

Counter(s):
- Takes machine through a linear sequence of states with few decisions along the way

FSM(s)
- Permits branches in control decision chain
Pipeline Control

- In a sense, an "unrolled" FSM – each stage does one step (or one of several parallel steps) in an FSM; state information communicated between stages
**Reset**

- Reset is a global signal that **the designer can not modify**
- It is generally asserted on power up or a “hard” reset
- It is used to get the machine into a “known” state
- Thus it must be distributed to
  - All FSMs
  - Selected counters
  - Selected status registers

Might trigger a “high fanout” warning in synthesis
- Is this OK?

*Yes. Reset is asserted for many clock cycles – can be slow*
Achieving Efficiency

High level tradeoffs:
- Parallelism
- Pipelining
- Optimizing the critical resource
  - E.g. Memory bandwidth
- Keep resources busy
  - If a resource is idle can it be shared?
  - Goal: Everything is used every clock cycle

Algorithmic Optimizations
- E.g. Algorithms that avoid DRAM accesses
  - e.g. Compress table onto SRAM
- Exploiting common algorithms in Computer Science
  - e.g. Boyer-Moore for string matching
  - e.g. Hash tables for matches
  - e.g. Shift instead of *2 /2
Mid-level efficiency

Think hardware (area and delay):

- Avoid large FSMs
- Count the large units (* + memories, etc.)
- Avoid high-fanout signals
- Avoid priority logic
- Structure arithmetic for speed
  - E.g. CLA instead of ripple carry

Exploit existing Intellectual Property
**Design Ware**

Synopsys, and others, provide libraries of carefully optimized design blocks for you to use -- called `Design Ware`:

- Libraries include: Arithmetic, Advanced Math, DSP, Control, Sequential, and Fault Tolerant
- For +,-,*, >=, <=, >, and <, design ware is automatically used
- More complex cells must be inferred via a procedure call. e.g. cosine:
  ```verbatim
define trig (angle, cos_out);
parameter wordlength1 = 8, wordlength2 = 8;
input [wordlength-1:0] angle;
output [wordlength-1:0] cos_out;
// passes the widths to the cos function
parameter angle_width = wordlength1, cos_width = wordlength2;
`include `/afs/bp/dist/synopsys_syn/dw/sim_ver/DW02_cos_function.inc"
wire [wordlength2-1:0] cos_out;
// infer DW02_cos
assign cos_out = cos(angle);
endmodule
```
**Sub-module Summary**

ALWAYS

- Design before coding
- Design the Datapath first
  - Then controller just has to toggle the control lines in the right sequence

Styles of controllers

- Counters
- Finite State Machines
- Pipelined Finite State Machines

- THINK Hardware – use efficient structures
- Reset goes to all state registers

→ Quiz and next sub-module
**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):

Reference Block in Frame 1

Search window in frame 2

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

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

Original Location of Reference Block in Frame 1
System Requirements

System Requirements:

- 16x16 Reference Block
- 31x31 Search Window
- Each stored in one two-read-port 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
- Clocks available: 130, 260 MHz
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
Digital ASIC Design

… 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
  - \(\Rightarrow 4096 \times 15 \times 256 \times 256 = 4.027E9\) add-accumulates/second
  - At 260 MHz \(\Rightarrow\) At least 16 adders in parallel \((4027 / 260 = 15.5)\)

• Searches/block [(4x4) on (10x10) example]:

- Search (-3,-3)
- Search (-2,-3)
- Search (-1,-3)
- Search (0,-3)
- Search (1,-3)
- Search (2,-3)
- Search (3,-3)

7 searches per column
7 searches per row

\((10-4) \times (10-4)\) total searches
Digital ASIC Design

...System Design

First Attempt

- Assign one search per Accumulator

Problem: Requires 16 port S memory!
Second Attempt:

- Stagger Startup of Accumulators

Problem:
16-port R mem required.

But Notice!

R pattern
**Final Solution:**

- Pipeline R

![Diagram](image)

**Vector:**

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Vector Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>r_{0,0}-S_{0,0}</td>
</tr>
<tr>
<td>2</td>
<td>r_{0,1}-S_{0,1}</td>
</tr>
<tr>
<td>3</td>
<td>r_{0,2}-S_{0,2}</td>
</tr>
<tr>
<td>4</td>
<td>r_{0,3}-S_{0,3}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ports Required</th>
<th>2 S mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>r_{0,15}-S_{0,15}</td>
</tr>
<tr>
<td>16</td>
<td>r_{1,1}-S_{1,1}</td>
</tr>
</tbody>
</table>
Step 2: Design Datapath

Datapath Details:

- Detailed hardware required to implement above

PE = Processing Element

To comparator
Coding Datapath

**PE:** Note, accumulator can’t 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;
    reg [7:0] SelS;

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

    always@(S1S2mux or S1 or S2)
        SelS = S1S2mux?S1:S2;

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

Datapath

$PE_{out}$

$PE_{ready}$

Vector$x$

Vector$y$

$motionX$

$motionY$

Best$Dist$

$Pe_{out} < BestDist?\)
Comparator Module

module Comparator(clock, CompStart, PEout, PEready, vectorX, vectorY, BestDist, motionX, motionY);
input clock;
input CompStart; // goes high when distortion calculations start
input [8*16-1:0] PEout; // Outputs of PEs as one long vector, I think it should be [8*16-1:0] corresponding to 16 input PEout[7:0] pins, instead of [8*16:
input [15:0] PEready; // 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)
begin
    if(CompStart == 0) BestDist <= 0'hFF; // initialize to highest value
    else if(newBest == 1)
        begin
            BestDist <= newDist;
            motionX <= vectorX;
            motionY <= vectorY;
        end
end

always@ (PEready or PEout)//(posedge clock)
begin
    case{PEready}
        16'b0000000000000001 : newDist = PEout[7:0];
        16'b0000000000000010 : newDist = PEout[15:8];
        16'b0000000000000001 : newDist = PEout[23:16];
        16'b0000000000000000 : newDist = PEout[31:24];
        16'b0000000000000000 : newDist = PEout[39:32];
        16'b0000000000000000 : newDist = PEout[47:40];
        16'b0000000000000000 : newDist = PEout[55:48];
        16'b0000000000000000 : newDist = PEout[63:56];
        16'b0000000000000000 : newDist = PEout[71:64];
        16'b0000000000000000 : newDist = PEout[79:72];
        16'b0000000000000000 : newDist = PEout[87:80];
        16'b0000000000000000 : newDist = PEout[95:88];
        16'b0000000000000000 : newDist = PEout[103:96];
        16'b0000000000000000 : newDist = PEout[111:104];
        16'b0000000000000000 : newDist = PEout[119:112];
        16'b0000000000000000 : newDist = PEout[127:120];
        default : newDist = PEout[7:0];
    endcase
end

always@ (PEready or CompStart or newDist or BestDist)
begin
    if((PEready == 0) || (CompStart == 0) || newBest == 0) // no PE is ready
        else if(newDist < BestDist)
            newBest = 1;
        else
            newBest = 0;
end
endmodule
Step 3. Identify Control Points

PE control lines:
S1S2mux [15:0]; // S1-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)
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;

integer i;

always@(posedge clock) begin
if(start == 0) count <= 13'h0;
else if(completed == 0)
    count <= count + 1'b1;
end

always@(count)
begin
    for(i = 0; i < 16; i= i+1)
        begin
            PEready[i] <= (NewDist[i] && !(count < 9'd256));
        end
end

always@(count)
begin
    AddressR <= count[7:0];
    AddressS1 = (count[11:8] + count[7:4]) * 6'd32 + count[3:0];
    S1S2mux[0] = 1'b1;
    for(i = 0; i < 16; i=i+1) NewDist[i] = (count[7:0] == i);
    if(NewDist[0] != 0) AddressS2 <= AddressS2 + 5'd17;
else if(NewDist == 0) AddressS2 <= AddressS1 - 5'd16;
    else AddressS2 <= AddressS2 + 1'b1;

    VectorX <= count[3:0]; // - 4'd7;
    VectorY <= count[11:8]; // - 4'd8;
    CompStart <= start;
end

always@(count)
begin
    completed = (count == 5'd16 * (9'd256 + 1));
end
endmodule
Comments on Design

Note: Counter strategy
- No branches in FSD

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

Hierarch:
Digital ASIC Design

Top without Mem

module top_without_mem(clock, start, R, S1, S2, AddressR, AddressS1, AddressS2, BestDist, motionX, motionY); //, newDist, PEready);

    input clock;
    input start;
    input [7:0] R, S1, S2;

    output [9:0] AddressS1, AddressS2;
    output [7:0] AddressR;
    output [7:0] BestDist;
    output [3:0] motionX, motionY;

    wire [9:0] AddressS1, AddressS2;
    wire [7:0] AddressR;
    wire [7:0] R, S1, S2;
    wire [15:0] S1S2mux;
    wire [3:0] vectorX, vectorY;
    wire [15:0] PEready;
    wire CompStart;
    wire [15:0] NewDist;
    wire [8*16-1:0] PEout;

    wire [7:0] Rpipe0, Rpipe1, Rpipe2, Rpipe3, Rpipe4, Rpipe5, Rpipe6, Rpipe7, Rpipe8, Rpipe9, Rpipe10, Rpipe11, Rpipe12, Rpipe13, Rpipe14, Rpipe15;
    wire [7:0] BestDist;
    wire [3:0] motionX, motionY;
    wire start;
… Top without Mem

```
PE u0(.clock(clock), .R(R), .S1(S1), .S2(S2), .S1S2mux(S1S2mux[0]), .newDist(NewDist[0]), .Accumulate(PEout[7:0]), .Rpipe(Rpipe0));
PE u1(.clock(clock), .R((Rpipe0)), .S1(S1), .S2(S2), .S1S2mux(S1S2mux[1]), .newDist(NewDist[1]), .Accumulate(PEout[15:8]), .Rpipe(Rpipe1));
PE u2(.clock(clock), .R((Rpipe1)), .S1(S1), .S2(S2), .S1S2mux(S1S2mux[2]), .newDist(NewDist[2]), .Accumulate(PEout[23:16]), .Rpipe(Rpipe2));
PE u3(.clock(clock), .R((Rpipe2)), .S1(S1), .S2(S2), .S1S2mux(S1S2mux[3]), .newDist(NewDist[3]), .Accumulate(PEout[31:24]), .Rpipe(Rpipe3));
PE u4(.clock(clock), .R((Rpipe3)), .S1(S1), .S2(S2), .S1S2mux(S1S2mux[4]), .newDist(NewDist[4]), .Accumulate(PEout[39:32]), .Rpipe(Rpipe4));
PE u5(.clock(clock), .R((Rpipe4)), .S1(S1), .S2(S2), .S1S2mux(S1S2mux[5]), .newDist(NewDist[5]), .Accumulate(PEout[47:40]), .Rpipe(Rpipe5));
PE u6(.clock(clock), .R((Rpipe5)), .S1(S1), .S2(S2), .S1S2mux(S1S2mux[6]), .newDist(NewDist[6]), .Accumulate(PEout[55:48]), .Rpipe(Rpipe6));
PE u7(.clock(clock), .R((Rpipe6)), .S1(S1), .S2(S2), .S1S2mux(S1S2mux[7]), .newDist(NewDist[7]), .Accumulate(PEout[63:56]), .Rpipe(Rpipe7));
PE u8(.clock(clock), .R((Rpipe7)), .S1(S1), .S2(S2), .S1S2mux(S1S2mux[8]), .newDist(NewDist[8]), .Accumulate(PEout[71:64]), .Rpipe(Rpipe8));
PE u9(.clock(clock), .R((Rpipe8)), .S1(S1), .S2(S2), .S1S2mux(S1S2mux[9]), .newDist(NewDist[9]), .Accumulate(PEout[79:72]), .Rpipe(Rpipe9));
PE u10(.clock(clock), .R((Rpipe9)), .S1(S1), .S2(S2), .S1S2mux(S1S2mux[10]), .newDist(NewDist[10]), .Accumulate(PEout[87:80]),
    .Rpipe(Rpipe10));
PE u11(.clock(clock), .R((Rpipe10)), .S1(S1), .S2(S2), .S1S2mux(S1S2mux[11]), .newDist(NewDist[11]), .Accumulate(PEout[95:88]),
    .Rpipe(Rpipe11));
PE u12(.clock(clock), .R((Rpipe11)), .S1(S1), .S2(S2), .S1S2mux(S1S2mux[12]), .newDist(NewDist[12]), .Accumulate(PEout[103:96]),
    .Rpipe(Rpipe12));
PE u13(.clock(clock), .R((Rpipe12)), .S1(S1), .S2(S2), .S1S2mux(S1S2mux[13]), .newDist(NewDist[13]), .Accumulate(PEout[111:104]),
    .Rpipe(Rpipe13));
PE u14(.clock(clock), .R((Rpipe13)), .S1(S1), .S2(S2), .S1S2mux(S1S2mux[14]), .newDist(NewDist[14]), .Accumulate(PEout[119:112]),
    .Rpipe(Rpipe14));
PE u15(.clock(clock), .R((Rpipe14)), .S1(S1), .S2(S2), .S1S2mux(S1S2mux[15]), .newDist(NewDist[15]), .Accumulate(PEout[127:120]),
    .Rpipe(Rpipe15));

Comparator u21(.clock(clock), .CompStart(CompStart), .PEout(PEout), .PEready(PEready), .vectorX(vectorX), .vectorY(vectorY),
    .BestDist(BestDist), .motionX(motionX), .motionY(motionY)); // , .newDist(newDist));
control u22(.clock(clock), .start(start), .S1S2mux(S1S2mux), .NewDist(NewDist), .CompStart(CompStart), .PEready(PEready),
    .VectorX(vectorX), .VectorY(vectorY), .AddressR(AddressR), .AddressS1(AddressS1), .AddressS2(AddressS2));

dendmodule
```
Digital ASIC Design

Top

module top(clock, start, BestDist, motionX, motionY);//, newDist, PEready);

input clock;
input start;
output [7:0] BestDist;
output [3:0] motionX, motionY;

wire [9:0] AddressS1, AddressS2;
wire [7:0] AddressR;
wire [7:0] R, S1, S2;
wire [7:0] BestDist;
wire [3:0] motionX, motionY;
wire start;

top_without_mem u1(.clock(clock), .start(start), .R(R), .S1(S1), .S2(S2),
.AddressR(AddressR), .AddressS1(AddressS1), .AddressS2(AddressS2), .BestDist(BestDist),
.motionX(motionX), .motionY(motionY));

SRAM u23(.clock(clock), .WE(1'b0), .WriteAddress(10'h0), .ReadAddress1({1'b0,1'b0,
AddressR}), .WriteBus(32'h0), .ReadBus1(R));   // For R mem
SRAM u24(.clock(clock), .WE(1'b0), .WriteAddress(10'h0), .ReadAddress1(AddressS1),
.WriteBus(32'h0), .ReadBus1(S1));   // For S1 mem
SRAM u25(.clock(clock), .WE(1'b0), .WriteAddress(10'h0), .ReadAddress1(AddressS2),
.WriteBus(32'h0), .ReadBus1(S2));   // For S2 mem

endmodule
Summary – Motion Estimator Design

Key elements of design:

- Design datapath first:
  - Determined schedule of operations that achieved the maximum performance WITHIN the available memory bandwidth
  - Then sketched out the logic before capturing as RTL
- Design controller second:
  - Determined control points and sequence
  - Counter was best fit for controlling sequence
  - Count value was decoded for determining actual settings of control lines on any specific value of count

→ Quiz and next sub-module
Generally the “flow” constructs in C correlate to controller designs in Verilog, e.g.

In “C”: If \( A \leq 5 \) \{B=A+C;\} else \{B=A-C;\}

In Hardware:
C to Verilog (cont’d)

For loop:
- In “C”: B=0; for (i=0;i<=7;i++) B=B+A;
- In Hardware:

![Diagram of a for loop implementation in hardware]

0  B
\[ \text{Mux} = \text{hold} \]
\[ \text{Counter} \]
\[ \text{Mux} = \text{A} + \text{B} \]
\[ \text{Count} = 7 \]
\[ \text{Start Count} \]
\[ \text{Mux} = 0 \]
Minimizing Power Consumption

Will go over in a later set of notes, but here is the logic design impact…

- In general, at the logic level, the energy required to complete a complex task is roughly proportional to:

\[ \sum_{\text{nodes}} \text{01 and 10 logic transitions} \]

- E.g.

```
010
  "1 unit of energy"
```

```
01010
  "2 units of energy"
```

```
010
  "2 units of energy"
```

Note:

- Complex logical units (e.g. Multiplier) have a lot more internal nodes than simpler logical units
  - And thus consume more energy per operation
How to Minimize Power Consumption

- Simpler, smaller design will often also more energy efficient
- There is often a speed-power tradeoff
  E.g. Which design is more energy efficient?

  More energy efficient. (Fewer logic gates toggle per compare performed.)

- Try to eliminate useless toggling
  E.g. Which design is LESS energy efficient if B mostly DESELECTS mult output?

  Energy in MULT wasted if B does not select new mult output and inputs change
... *How to minimize power consumption*

- Memory accesses are particularly energy hungry, especially with larger memories
- Complex data motion is particularly power hungry
  - Especially Long range on-chip interconnect and Off-chip interconnect
**Submodule Summary**

**C to Verilog:**
- Useful only if you are “stuck”
- Will not give the most compact logic

<table>
<thead>
<tr>
<th>C</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>If-else</td>
<td>Mux + FSM</td>
</tr>
<tr>
<td>for</td>
<td>Logic + counter</td>
</tr>
</tbody>
</table>

**Minimizing Power Consumption**
- Minimize total number of node “toggles” (010) required to complete computational task

End of Module