Why Custom Hardware?

- Size – Area constraints may force us to use a specific item rather than a general solution
  - Smart Card
- Cost – mass production of specific item may be cheaper than utilizing a general solution
  - Coke Machine???
  - Most of these apps are generally done by low-cost microprocessors.
- Power – A general solution might require more power than we have available on our system
  - Cell Phone
- Or …

Performance

Some Challenges

- How do we take something running in software and make it fast?
  - Algorithmic changes
    - By FAR, the most significant
  - Architecture choices
  - Implementation choices
  - Synthesis Optimizations
  - Place and Route Optimizations

Algorithmic Changes

- Algorithms are generally developed to function on a general-purpose computer
  - Sequential algorithms
  - Floating point precision
  - Memory access is either un-optimized or optimized for cache accesses – but still sequentially
    - A sequential algorithm sees no difference between interacting with 1 or 1000 arrays of data
- In order to work effectively in hardware:
  - Parallel algorithms
    - Stream of concurrent processes
    - Many identical components doing different solutions
  - Integer precision
  - Optimized for finite memory sizes and ports
  - Removal of processing "walls" where all processing is forced to wait for the completion of an event before proceeding

LFSR Example

- Fibonacci vs. Galois Linear Feedback Shift Register (LFSR)

a_{r-1} a_{r-2} ... a_1 a_0

q_1 q_2 ... q_{r-1} q_r

$\oplus$

a_{r-1} ... a_1 a_0

q_r q_{r-1} ... q_2 q_1

Turbo Decoder Example

Turbo Encoder

Information bits
Transmitted information bits
Interleaver
Encoder 1
Encoder 2
Transmitted parity bits
Puncture

CHANNEL

Turbo Decoder

Received information bits
max-logMAP Decoder 1 *(1)*
De-puncture
Interleaver
De-interleaver
Received parity bits
max-logMAP Decoder 2 *(1)*

Architecture Choices

- A Brief survey of Verilog blocking vs. nonblocking assignments
- Pipelining
  - Register Retiming
- State Machine Implementations
  - Structure
  - Encoding
- FPGA Details

Blocking vs. Nonblocking Assignments

- Verilog has 2 types of assignments to a register datatype
  - Blocking (a = b)
  - Nonblocking (a <= b)
- Blocking Assignments
  - Performed immediately (simulation blocks)
- Nonblocking Assignments
  - Performed after all active events in the event queue

Simplified Verilog Simulation Reference Model

```
In all the examples that follow, T refers to the current simulation time,
   and all events are held in the event queue, ordered by simulation time.

while (there are events) {
  if (there are active events) {
    E = any active event;
    if (E is an update event) {
      update the modified object;
      add evaluation events for sensitive processes to event queue;
    }
    else { // this is an evaluation event, so ...
      evaluate the process;
      add update events to the event queue;
    }
  }
  else if (there are nonblocking update events) {
    activate all nonblocking update events;
  }
  else {
    advance T to the next event time;
    activate all inactive events for time T;
  }
}
```


References

- Cliff Cummings - http://www.sunburst-design.com/papers/
  - Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!
  - Verilog Nonblocking Assignments With Delays, Myths & Mysteries

Difference between blocking and nonblocking

```
always @ (posedge clk)
begin
  b = a;
  c = b;
end
```

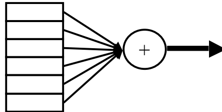
```
always @ (posedge clk)
begin
  b <= a;
  c <= b;
end
```

a b c

a b c

Pipelining via Register Retiming

- Pipelining
  - explicitly by the designer
  - automatically by the synthesis tool
- Consider a population counter



Population Count Example

```
module test (a, pop, clk, rst);
input [32:0] a;
input clk, rst;
output [6:0] pop;
reg [6:0] pop;

always @ (posedge clk or posedge rst) begin
  if (rst) begin
    pop   <= 0;
  end
  else begin
    pop <= a[0]  + a[1]  + a[2]  + a[3]  + a[4]  + a[5]  + a[6]  + a[7] +
           a[8]  + a[9]  + a[10] + a[11] + a[12] + a[13] + a[14] + a[15] +
           a[16] + a[17] + a[18] + a[19] + a[20] + a[21] + a[22] + a[23] +
           a[24] + a[25] + a[26] + a[27] + a[28] + a[29] + a[31] + a[31];

  end
end
endmodule
```


Population Count Pipelined (1)

```
module test (a, pop, clk, rst);

input [32:0] a;
input clk, rst;
output [6:0] pop;

reg [2:0]
p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11;
reg [4:0] q1,q2,q3,q4;
reg [5:0] r1,r2;
reg [6:0] pop;

always @ (posedge clk or posedge
rst) begin
  if (rst) begin
    pop   <= 0;
  end
```

```
  else begin
     p1  <= a[0]  + a[1]  + a[2];
     p2  <= a[3]  + a[4]  + a[5];
     p3  <= a[6]  + a[7]  + a[8];
     p4  <= a[9]  + a[10] + a[11];
     p5  <= a[12] + a[13] + a[14];
     p6  <= a[15] + a[16] + a[17];
     p7  <= a[18] + a[19] + a[20];
     p8  <= a[21] + a[22] + a[23];
     p9  <= a[24] + a[25] + a[26];
     p10 <= a[27] + a[28] + a[29];
     p11 <= a[31] + a[31];
     q1  <= p1  + p2  + p3;
     q2  <= p4  + p5  + p6;
     q3  <= p7  + p8  + p9;
     q4  <= p10 + p11;
     r1  <= q1 + q2;
     r2  <= q3 + q4;
     pop <= r1 + r2;
   end
end
endmodule
```


Population Count Pipelined (2)

```
module test (a, pop, clk, rst);

input [32:0] a;
input clk, rst;
output [6:0] pop;

reg [2:0] p1,p2,p3,p4,p5,p6,p7,p8;
reg [4:0] q1,q2;
reg [6:0] pop;

always @ (posedge clk or posedge rst)
begin
  if (rst) begin
    pop   <= 0;
  end
```

```
  else begin
     p1  <= a[0]  + a[1]  + a[2]  + a[3];
     p2  <= a[4]  + a[5]  + a[6]  + a[7];
     p3  <= a[8]  + a[9]  + a[10] +
    a[11];
     p4  <= a[12] + a[13] + a[14] + a[15];
     p5  <= a[16] + a[17] + a[18] + a[19];
     p6  <= a[20] + a[21] + a[22] + a[23];
     p7  <= a[24] + a[25] + a[26] + a[27];
     p8  <= a[28] + a[29] + a[30] + a[31];
     q1  <= p1 + p2 + p3 + p4;
     q2  <= p5 + p6 + p7 + p8;
     pop <= q1 + q2
   end
end
endmodule
```

Population Count Retimed

```
module test (a, pop, clk, rst);
input [32:0] a;
input clk, rst;
output [6:0] pop;
reg [6:0] pop;
reg [6:0] p1, p2, p3, p4;

always @ (posedge clk or posedge rst) begin
  if (rst) begin
    pop   <= 0;
    p1    <= 0;
    p2    <= 0;
    p3    <= 0;
  end
  else begin
    p1  <= a[0]  + a[1]  + a[2]  + a[3]  + a[4]  + a[5]  + a[6]  + a[7] +
           a[8]  + a[9]  + a[10] + a[11] + a[12] + a[13] + a[14] + a[15] +
           a[16] + a[17] + a[18] + a[19] + a[20] + a[21] + a[22] + a[23] +
           a[24] + a[25] + a[26] + a[27] + a[28] + a[29] + a[31] + a[31];
    p2  <= p1;
    p3  <= p2;
    p4  <= p3;
    pop <= p4; // pop <= p3; pop <= p2; pop <= p1;
  end
end
endmodule
```


Population Count Example Times

- Solo pop count => 148 MHz
- Pipelined version 1 => 296 MHz
- Pipelined version 2 => 279 MHz
- Retimed pop <= p4; => 308 MHz
- Retimed pop <= p3; => 316 MHz
- Retimed pop <= p2; => 328 MHz
- Retimed pop <= p1; => 293 MHz

State Machines

- Use one sequential always block for next state assignment:
  - Typically 3 lines of code

```
always @ (posedge clk or negedge rst_n)
  if (!rst_n) state <= IDLE;
  else        state <= next;
```

- Put the state machine outputs in a separate combinational always block
- This 2-block style is efficient because output assignments are only required to be listed once for each state in the case statement
- Avoid Verilog race conditions
  - Code all sequential always blocks using non-blocking assignments
  - Code all combinational blocks using blocking assignments

State Machine Encoding

- In Verilog, use parameters as state assignments rather than ‘defines
  - ‘defines are global
  - This avoids redefinition warnings when multiple state machines exist in a compilation
- Use one-hot encoding for high-performance state machines
  - This technique uses more registers, but simplifies next-state logic
  - That said, the number of excess registers is typically trivial when compared to the controlled datapath
  - In Verilog, using reverse **case** statement ensures inference of efficient comparison logic that only does 1-bit comparisons against the onehot bits of the **state** and **next** vectors.
- http://www.sunburst-design.com/papers/CummingsSNUG2003SJ_SystemVerilogFSM_rev1_1.pdf

Verilog one-hot encoding

```
module fsm_cc1_3oh (output reg rd, ds,input
   go, ws, clk, rst_n);
parameter
  IDLE = 0,
  READ = 1,
  DLY = 2,
  DONE = 3;
reg [3:0] state, next;
always @(state or go or ws) begin
next = 4'b0;
case (1'b1)
  state[IDLE] : if (go)  next[READ] = 1'b1;
                else     next[IDLE] = 1'b1;
  state[READ] :          next[ DLY] = 1'b1;
  state[ DLY] : if (!ws) next[DONE] = 1'b1;
                else     next[READ] = 1'b1;
  state[DONE] :          next[IDLE] = 1'b1;
endcase
end
```

```
always @(posedge clk or negedge
   rst_n)
  if (!rst_n) begin
    rd <= 1'b0;
    ds <= 1'b0;
  end
  else begin
    rd <= 1'b0;
    ds <= 1'b0;
  case (1'b1)
    next[READ] : rd <= 1'b1;
    . . .
```


FPGA Details

- 4-input LUT's
  - Attempt to architect logic into a pipeline of 4-bit inputs
- SRL/FIFO
  - Building shift registers or FIFO's from LUT's can be much cheaper than "simpler" solutions using registers
    - An 8-bit shift register can be done in one LUT, but requires 8 flip-flops + routing resources.
- Carry-chains
  - Ripple carry adders are typically much faster in an FPGA than an "faster" adder such as a carry lookahead as the carry chains are built into the fabric of the FPGA
- Block Rams
- Multipliers
- Power PC
- ????

Implementation Choices

- Duplicating Flip-Flops
- Coding styles for speed
- Selection instead of Evaluation
  - throw silicon at it
- Instantiate Target Technology primitives

Duplicating Flip-Flops

- High-fanout nets can be slow and hard to route
- Duplicating flip-flops can fix both problems
  - Reduced fanout shortens net delays
  - Each flip-flop can fanout to a different physical region of the chip to help with routing
- Design tradeoffs
  - Gain routability and performance
  - Design area increases

fn1 D Q

fn1 D Q

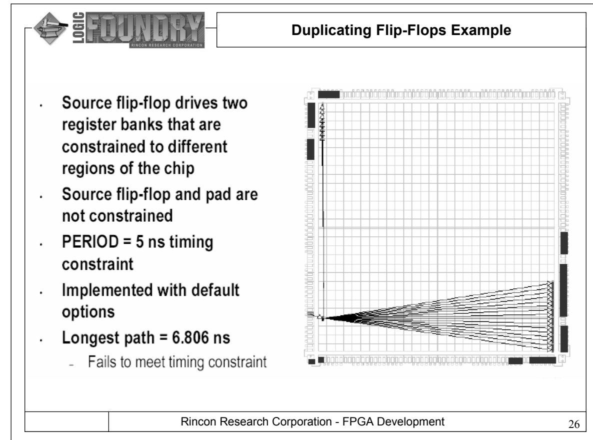
fn1 D Q

Tips on Duplicating Flip-Flops (Xilinx)

- Name duplicated flip-flops _a, _b: *NOT* _1, _
  - Numbered flip-flops are mapped by default into the same slice
  - You generally want duplicated flip-flops to be separated
    - Especially if the loads are spread across the chip
- Explicitly create duplicate flip-flops in your HDL code
  - Most synthesis tools have automatic fanout-control features
    - However, they do not always pick the best division of loads
    - Also, duplicated flip-flops will be named _1, _2
  - Many synthesis tools will optimize-out duplicated flip-flops
- Do not duplicate flip-flops that are sourced by asynchronous signals
  - Synchronize the signal first
  - Feed synchronized signal to multiple flip-flops

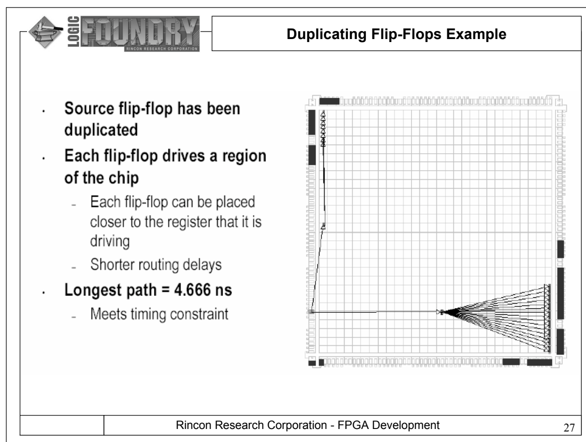
Duplicating Flip-Flops Example

- **Source flip-flop drives two register banks that are constrained to different regions of the chip**
- **Source flip-flop and pad are not constrained**
- **PERIOD = 5 ns timing constraint**
- **Implemented with default options**
- **Longest path = 6.806 ns**
  - Fails to meet timing constraint



Duplicating Flip-Flops Example

- **Source flip-flop has been duplicated**
- **Each flip-flop drives a region of the chip**
  - Each flip-flop can be placed closer to the register that it is driving
  - Shorter routing delays
- **Longest path = 4.666 ns**
  - Meets timing constraint



Duplicating Logic

- Similar to the above example, but rather than simply duplicating flip-flops, sometimes it is necessary to duplicate entire sections of logic

LOGIC FOUNDRY RINCON RESEARCH CORPORATION

Coding Styles for speed

- Avoid Nested IF and CASE Statements
  - Nested IF or CASE statements infer cascaded logic
  - More levels of logic _lower performance
- Evaluate late-arriving signals last
  - When nested IFs are necessary, put critical input signals on the first (outer) IF statement
    - The critical signal ends up in the last logic stage

LOGIC FOUNDRY RINCON RESEARCH CORPORATION

Select rather than Evaluate

- Sometimes it is faster to select from all possible outcomes than it is to compute the correct outcome
  - Rather than wait for this 2-bit signal to arrive

```
acc <= acc + 2bit_signal ;
```

  - Compute all possibilities and use the control as an output selector

```
o1 <= acc + 00;
o2 <= acc + 01;
o3 <= acc + 10;
o4 <= acc + 11;
case (2bit_signal) begin
   00: acc <= o1;
   01: acc <= o2;
   ...
```


LOGIC FOUNDRY RINCON RESEARCH CORPORATION

Instantiate when speed needed

- Xilinx says …
  - Use instantiation only when it is necessary to access device feather or increase performance or decrease area
  - Limit the location of instantiated components to a few source files to make it easier to locate these components when porting the code
- So, we use instantiation frequently
  - It is almost always necessary to increase performance
  - Reusable code is a bit of a myth

LOGIC FOUNDRY RINCON RESEARCH CORPORATION

Synthesis Optimizations

- Pipelining
  - Certain tools allow automatic pipelining of Xilinx multipliers and ROM's
- Retiming
  - We have discussed this
- Overconstraining
  - Always overconstrain the synthesis tool as they are never correct. Period.
    - Do not pass these constraints on to the layout tools – use the correct constraints in layout

Place and Route Optimizations

- Floorplanning
  - ENIAC example
- RPM's (relationally placed macros)
- Custom Placement
- Multi-Pass Place and Route

RPM's

B L O C K R A M

M U L T I P L I E R

Port A of Multiplier (input operand)

Port B of Multiplier (input operand)

Port P of Multiplier (output operand)

Location of RLOC_ORIGIN

Unusable Slices

Figure 2: **18 X 18 Multiplier Floorplan**

Summary

- Algorithmic changes
  - By FAR, the most significant
- Architecture choices
- Implementation choices
- Synthesis Optimizations
- Place and Route Optimizations