Advanced Synthesis Techniques
Reminder From Last Year

- Use UltraFast Design Methodology for Vivado
  - [www.xilinx.com/ultrafast](http://www.xilinx.com/ultrafast)

- Recommendations for Rapid Closure
  - **HDL**: use HDL Language Templates & DRC
  - **Constraints**: Timing Constraint Wizard, DRC
  - **Iterate in Synthesis** (converge within 300ps)
    - **Real problems seen post synthesis** (long path...)
    - Faster iterations & higher impact
    - Improve area, timing, power
  - Only then, iterate in next steps
    - opt, place, *phys_opt*, route, *phys_opt*

Tools→Report→Report DRC

Worst path **post Synthesis**: 4.3ns
13 levels of logic!

Worst path **post Route**: 4.1ns
4 levels of logic
Advanced Synthesis Techniques Overview

- Advantages of C Synthesis over RTL Synthesis
- Advanced Synthesis Techniques for Design Closure
- Case Study: design closure at Synthesis level
HLS & IP Integrator (IPI) vs. RTL Synthesis

**Verification advantage**: e.g. video processing
- RTL: 1 frame per ~5 hours
- C++: 1 frame per second

**Actual example:**
- Traditional Flow
  - 240 people*mo
  - 10 people
  - 2 years
- HLS Based Flow
  - 16 people*mo
  - 2 people
  - 8 month

**15x faster**
- Faster for derivative designs
  - C++ reuse
  - Scales with parameters
  - Device independent
HLS Automates Micro-architecture Exploration

while (i++)
for j =0 .. N
\[ y(i) = y(i) + b(j) \times x(i-j) \]

while (i++)
for j =0 .. N
\[ y(i) = y(i-j) + b(j) \times x(i) \]

Project specification

Architecture choices

Fully parallel: N DSP + cascade

Fully Folded: 1 DSP (default)

Fully parallel: N DSP no cascade

while (i++)
for j =0 .. N
\[ y(i) = y(i) + b(j) \times x(i-j) \]

while (i++)
for j =0 .. N
\[ y(i) = y(i-j) + b(j) \times x(i) \]

Micro-architecture choices

Algorithmic delay

Pipeline register

Xilinx

Vivado users group
while (1)
  M3(i) = M1(i-2) * C2
  A1(i) = M3(i) + x(i)
  M2(i) = M1(i-1) * C1
  A2(i) = A1(i) + M2(i)
  M1(i) = A2(i) * C0
  i++

### Schedule 1: 14 cycles
Sequential process (CPU model)
**Minimal HW Resources:** 1 MULT, 1 ADD

### Schedule 2: 10 cycles
Parallelism within each iteration
**Better performance (~29%)**
2 MULT, 1 ADD

### Schedule 3: 9 cycles
Loop pipelining
**Best performance (~36%)**
2 MULT, 1 ADD
Vivado Synthesis Flow

 VHDL, Verilog
 VHDL-2008, SystemVerilog
 more compact: advanced types...
 verification friendly: UVM, SVA...

Syntax check
Build file hierarchy

Design hierarchy
Unroll loops
Build Logic:
• Arithmetic
• RAM
• FSM
• Boolean logic

Module generators
RTL Optimizations
Boolean optimization
Technology mapping

XDC

P&R or DCP
• Architecture-Aware Coding
• Priority Encoders
• Loops
• Clocks & Resets
• Directives & Strategies
• Case Study
HDL code needs to match DSP hardware (e.g. DSP48E2)
- Signage, width of nets, optimal pipelining...

Signed arithmetic with pipelining

Verify that DSP are inferred efficiently

Use templates & Coding style examples:
- Complex multiplier
- Squarer (UG901)
- Multiply-accumulate
- Dynamic pre-adder
- FIR (UG579)
- Large accumulator
- Rounding (2015.3)
- XOR (2016.1)
- ...
DSP Block Inference Improvements

**Squarer: 1 DSP**

\[
(a - b)^2 = (a + b)^2 - 4ab
\]

\[
\left(\sum_{k=0}^{n} (a_k - b_k)^2\right)
\]

**Complex multiplier: 3 DSP**

\[
(a+bi)*(c+di) = ((c-d)*a + S) + ((c+d)*b + S)i
\]

with \( S = (a-b)*d \)

Wider arithmetic requires more pipelining

*e.g. MULT 44x35 requires 4 MULT 27x18 & ADD*

Pipe-lined MULT 44x35 in HDL

Mapped to 4 DSP Blocks (27x18 MULT)

Verify proper inference for full DSP block performance!
Architecture-Aware RAM & ROM

- HDL code needs to match BRAM Architecture
  - Registered address (sync read), optional output register
  - 32K configurations
    - Width=1 x Depth=2^{15} (32K) = 32Kx1
    - Width=2 x Depth=2^{14} (16K) = 16Kx2
    - ...
    - Width=32 x Depth=2^{10} (1K) = 1Kx32
  - 36K configuration
    - Width=36 x Depth=2^{10} (1K) = 1Kx36

- Wider & Deeper Memories
  - Automatically inferred by Synthesis

Verify that BRAM are inferred efficiently!
RAM Decomposition: Example

- **32Kx32 RAM**

  - `define W 32
  - `define D 15

  - High Performance & Power (default w/ timing constraints)
    - 1 level, 32 BRAM active

  - Low Power & Performance
    - UltraScale cascade-MUX
    - 32 levels, 1 BRAM active
    - (* cascade_height = 32 *) ...

  - Performance/Power Trade-off
    - Hybrid LUT & UltraScale Cascade
    - 4 levels, 4 BRAM active
    - (* cascade_height = 4 *) ...

Verify that BRAM are decomposed efficiently!
RAM & ROM Recommendations

Use pipeline Reg for performance

No logic in-between

No Fanout

In same hierarchy!

Run `phys_opt` to move Reg in & out based on timing

Add extra pipeline for best performance!

Verify that BRAM are pipelined efficiently!
Beware of Priority Logic

if (c0) q = a0;
else if (c1) q = a1;
else if (c2) q = a2;
else if (c3) q = a3;
else if (c4) q = a4;
else if (c5) q = a5; ...

Priority encoded logic ➞ long paths

if (c0) q = a0;
if (c1) q = a1;
if (c2) q = a2;
if (c3) q = a3;
if (c4) q = a4;
if (c5) q = a5; ...

Removing else’s won’t help!!

Priority logic will hurt Timing Closure!
Priority Logic with “for” loops

```c
flag = 0;
for (i=0 ; i<31 ; i=i+1)
    if (c[i])
        flag = 1;
```

Same as if...if...if...

```c
flag = 0;
for (i=0 ; i<31 ; i=i+1)
    if (c[i]) begin
        flag = 1;
        break;  //SystemVerilog
    end
```

Same as if...else if...else if...

`break` won’t help!!

“break” does not reduce logic!

Best code in this case: flag = |c

Think Simple!
Priority Logic with “case” Statement

**case** (c)
- v0: q=a0;
- v1: q=a1;
- v2: q=a3;
- v3: q=a4;
- v4: q=a5;...

**CASE won’t help either!**
(note: values are variables)

In Verilog:
CASE (c) //synthesis parallel_case
(watch for simulation mismatch!)

In SystemVerilog:
**unique case** (c)  // works with “if” too

If conditions are mutually exclusive, make it clear!

**Note:** please use complete conditions
- .v  **full_case** (simulation may not match) or **default** & assign **don’t_care**
- .sv  **priority** (for case & if)
if (c0) q = a0;
else if (c1) q = a1;
else if (c2) q = a2;
else if (c3) q = a3;
else if (c4) q = a4;
else if (c5) q = a5; …

c0 = (S == 0);
c1 = (S == 1);
c2 = (S == 2);
c3 = (S == 3);
c4 = (S == 4);

Automated in most cases…
Even with registered conditions!

unique if (c0) …
in SystemVerilog

1-hot conditions
(here: binary encoded)

case (S)
0: q = a0
1: q = a1
2: q = a2
…
or:
q = A[S]

If conditions are mutually exclusive, do not use a priority logic
Use “unique if” in SystemVerilog

BAD

GOOD

GOOD

If conditions are mutually exclusive, do not use a priority logic
Use “unique if” in SystemVerilog
When you can’t avoid \( O(n) \), you still can!

**BAD:** \( N \) deep

**GOOD:** \( N/2 + 1 \) deep...
or \( N/4 + 2 \)...
or \( \log(N) \) recursively

**Improve timing even when conditions are not mutually exclusive!**
Beware of Loop Unrolling – Avoid “if”

`c = 0;`  
`for (i=0 ; i<8 ; i=i+1)`  
`if (a[i])`  
`    c = c+1;`  

Get rid of “if”

`c = 0;`  
`for (i=0 ; i<8 ; i=i+1)`  
`    c = c+a[i];`  

BAD: area & depth $O(N)$

GOOD: area & depth $\log_3(N)$

“if” in loops can seriously hurt timing!
Beware of Loop Unrolling – Arithmetic’s

\[
Q = 0 \\
\text{for } i = 0 \text{ to } 3 \\
\text{ for } j = 0 \text{ to } 3 \\
Q = Q + A + i + j
\]

\[
Q = \ldots \\
= 16 \times A + 48 \\
= A \ll 4 + 48
\]

\[
Q = (A + 3) \ll 4
\]

BAD: up to 36 \(N\) bit adder
GOOD: 1 \(N-3\) bit adder
BEFTER: 1 \(N-4\) bit adder

Loops (in general) can hurt timing!
Here: symbolic arithmetic optimization may not happen
Avoid Gated Clock Transformation

- Very common in ASIC design (low power)
- Consolidate the clocks to minimize clock skew

BAD: 2 clocks, 1 gated
GOOD: 1 clock

Avoid gated clocks – they will hurt timing closure (will cause clock skew)
Avoid [Async] Resets

- What we recommended
  - Reduce the number of “control sets” \{clk, rst, ce\}
  - Avoid Reset / avoid Async Reset

```
always@(posedge clk orposedge rst)
begin
  if(rst)
    begin
      //din_dly1 <= 16'b0;
      //din_dly2 <= 16'b0;
      dout <= 16'b0;
    end
  else
    begin
      din_dly1 <= din;
      din_dly2 <= din_dly1;
      dout <= din_dly2;
    end
end
```

Does this really remove reset?

**BAD:** Attempt to remove Reset created Enable and Reset is still Async...

Verify that removing Reset did not add Enables
- Guides Synthesis to directly connect net to DFF control pin
  - When Synthesis is choosing to not use the reset or enable
  - When there are multiple possibilities of enable or set/reset

Guide Synthesis to connect a signal to Enable, Reset
Vivado RTL Synthesis has now 8 Strategies

- Each Strategy is a combination of options & directives
- Directives have a specific purpose
  - For quick pipe-cleaning iterations
    - Flow_RuntimeOptimized
  - For best area
    - Flow_AreaMultThresholdDSP
    - Flow_AreaOptimized_medium
    - Flow_AreaOptimized_high
  - For performance
    - Vivado_Synthesis_Default
    - Flow_PerfOptimized_high
    - Flow_PerfThresholdCarry
  - For congested designs
    - Flow_AlternateRoutability

- Taking the best of all Strategies can give you 10% better QoR
Case Study

- **Problem**
  - Area explosion & bad timing in a design

- **Locating the cause of the issue**
  - Find offending module & synthesize it Out Of Context
  - Look for suspicious operators on Elaborated view (how??)
  - Cross-probe to source files

- **Resolution**
  - Fix the source code and/or use synthesis options
Case Study: Locating the Cause of the Issue

- Look for suspicious operators
  - Ctrl-F in Elaborated Schematic
  - Select suspicious operators (here: MULT, MOD...)
  - Press F4 to view schematic

- Press F7 to cross-probe
Case Study: More Useful / Fun Tips

Double Click to expand paths

Go back & forth

Cross-probe from HDL to schematic!!! (RTL or gate)

Select text & right-click

Press F4
Case Study: Analysis of QoR Issue

- Should this code generate arithmetic’s?

```vhdl
for i in 0 to 23 loop
    my_buffer((cnt*24+i) MOD 264) <= inp(i);
end loop;
```

- `cnt (values: 0..10) * 24 + i (values: 0..23) ➔ 264 constants`
- No MULT, ADD, or MOD necessary!

- How to fix it?

Please propose a code change to improve QoR...
Case Study: Resolution

#1 timing closure technique: careful analysis of Synthesis results!
Conclusion

- Use HLS & IPI when you can
  - 15X more productive for design & verification
  - HLS often achieves better quality of results

- Iterate in Synthesis for design closure!
  - Adopt SystemVerilog or VHDL-2008 for higher productivity
  - Use templates for big blocks
  - Investigate QoR issues
    - Locate possible Synthesis QoR issues
    - Recode or use tools options as needed
  - Final gunshot approach try different Strategies
  - Do not move to P&R until timing is closed (within 300 ps)