Synthesis and Loops

CMPE 415 Programmable Logic Devices

Prof. Ryan Robucci
References

• (Schaumont 2010) Schaumont, Patrick
A Practical Introduction to
Hardware/Software Codesign, Springer
ISBN 978-1-4614-3737-6
Single Assignment Code

- Consider the following code:
  ```
  a = b + 1;
  a = a * 3;
  ```

- This is the same as
  ```
  a = (b + 1) * 3;
  ```

- This only assigning a, and there is a single assignment to it.

- It can be implemented in hardware with
  ```
  [register b] → (+1) → (x3) → [register a]
  ```
A technique uses variable creation and renaming to create single assignments in section of code:

\[ a1 = b + 1; \]
\[ a2 = a1 \times 3; \]
We’ll first motivate MERGE with another example:

```c
a=b;
for(i=1;i<6;i++){
    a=a+i;
}
```

(Schaumont 2010)

**Attempt:**

```c
a1=b;
for(i=1;i<6;i++){
    a2= a? + i; //what to use?
    // for first iteration need a1,
    // thereafter need a2
}
```

(Schaumont 2010)
Solution is to use introduce the concept of a MERGE*. A MERGE maps to a mux in hardware, typically creating a loop or feedback.

Still must ensure that no combinatorial loops are formed, adding registers and multiple clock cycles as needed (below a2 may be implemented a register to ensure this).

```verbatim
a1=b;
for(i=1;i<6;i++){
    a3=MERGE(i==1?a1:a2);
    //now a1 and a2 will be registers
    a2=a3+1;
}
```

*MERGE is a concept, not an ACTUAL VERILOG CONSTRUCT

(Schaumont 2010)
Unrolling and Simplification

• Unrolling can be used to create single assignment code:

\[
\begin{align*}
a1 &= b; \\
a2 &= a1+1; \\
a3 &= a3+2; \\
a4 &= a4+3; \\
a5 &= a5+4; \\
a6 &= a6+5;
\end{align*}
\]

This can be implemented in one clock cycle with HW and 5 adders.

• Simplification can trim unnecessary complexity from run time

\[
a = b+15;
\]

a single adder in one clock cycle
Single Assignment Code Example

**Original Code**

```c
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```

**Single Assignment Code**

```c
int gcd(int a1, int b1) {
    while (MERGE(_?a1:a2) !=
            MERGE(_?b1:b2)) {
        if (a3 > b3) {
            a2 = a3 - b3;
        } else {
            b2 = b3 - a3;
        }
    }
    return a2;
}
```

(Schaumont 2010)
Single Assignment Code allows examination of data dependencies and hardware resources such as what can be done in a single clock cycle (combinatorial) and where a register is required. These concepts are also important when writing behavioral HDL code in Verilog or VHDL.
Today we'll discuss a few constructs which involve a concern of **Control Loops** and **Data (dependency) Loops** in procedural HDL.

We'll first concern ourselves with combinatorial behavior and then allow additional flexibility with sequential.

Should not attempt to synthesize any code that implements unresolvable dependency loops as combinatorial HW (executes in one clock cycle).
Think of each statement as a node on a graph with the edges denoting dependencies. Nodes can be producers and consumers of values. A graph with loops cannot be directly resolved as a combinatorial circuit. The inputs not generated from within the code are also nodes (they represent an assignment from elsewhere).

\[
\begin{align*}
y_1 &= a + b; \\
y_2 &= y_1 + c; \\
y_3 &= y_1 + y_2 \times y_2;
\end{align*}
\]
Branches can be thought of as multiplexers that depend on the evaluation of conditional expression. A new flag variable based on the condition evaluation may be introduced to make this clear.

```c
y1 = a+b;
if (y1 < 0)
    y2 = 0;
else
    y2 = y1;

y1 = a+b;
flag = (y1 < 0);
y2 = flag?0:y1;
```
Single Assignment Code (3)

- To achieve the status of single assignment code, every variable may only be assigned once.
- We may need to convert code to an equivalent single-assignment code to understand its underlying structure. To do this introduce additional variables when variables are assignment more than once.

```
y = a+b;
s = y+c;
y = y+s*s;
```

```
y1 = a+b;
s = y1+c;
y2 = y1+s*s;
```

```
y = a+b;
if (y < 0)
  y = 0;
```

```
y1 = a+b;
flag = (y1 < 0);
y2 = flag?0:y1;
```
always @ (a,b) begin
y = 0;
y = y&a;
y = y&b;
end

always @ (a,y) begin
y = ~(y&a);
end

always @ (a,b,yA,yB) begin
yA = ~(yB&a);
yB = ~(yA&b);
end

always @ (posedge clk) begin
y = ~(y&a);
end

No feedback after substitutions

Feedback

Feedback

This clearly does not attempt to describe combinatorial hardware it is edge-triggered describing sequential hardware. A register y is inserted.
A test for data dependency loops

- A conceptual test to understand if procedural code is implementing strictly combinatorial logic is to set every combinatorial result assigned to x at the entry to the code.
- If the behavior would change in any way under any case, then the code was not strictly implementing combinatorial logic

```verilog
always @(a,y) begin
    y=1'bx;
    y = ~(y&a);
end

always @(a,b) begin
    y=1'bx;
    y = 0;
    y = y&a;
    y = y&b;
end

always @(a,b,yA,yB) begin
    yA=1'bx; yB=1'bx;
    yA = ~(yB&a);
    yB = ~(yA&b);
end

always @(a,b,c,s,y) begin
    p=1'bx; y=1'bx;
    if (s) begin
        p = (a&b);
        y = ~(yA|c);
    end else begin
        y = a|b;
    end
end
```
A test for data dependency loops in mixed comb. and seq. blocks

• The test works for internal combinatorial values computed in edge-triggered blocks.

• In fact, a regular practice of setting combinatorial values to x at code entry may help identify issues in pre-synthesis simulation.

```verilog
class always @ (posedge clk) begin
    _y = 1’bx;
    y <= _y;
    _y = ~(a&b);
end
```
```verilog
always @ (posedge clk)
begin
    _y = 1’bx;
    _y = ~(a&b);
    y <= _y;
end
```
Combinatorial Synthesis: Loops

- Static Loops: Number of iterations defined at compile time. Can directly perform finite unrolling. Often synthesizers cannot convert non-static loops to combinational circuit.

- In the example below, the condition that is checked before every iteration is dependent on assignments within the body of the loop.

- Furthermore, the multiple data movements are problematic.

```plaintext
While Loop

temp=datain;
count =0;
for (index=0; |temp; index=index+1)
begin
  if temp[0]==1 (count=count +1)
  temp>>=1;
end
```
Combinatorial Synthesis: Loops

- Should rewrite to have static loop count and no implied data movement:

```plaintext
While Loop

count = 0;
for (index = 0; index < 8; index = index + 1) begin
    if temp[index] == 1 (count = count + 1);
end
```
Registered logic (mix comb and seq.) should be separated to understand the dependencies. New variables may be introduced to denote the difference in signals before and after a register.

```verilog
always @(posedge clk) begin
  if (counter == CNT_MAX)
    counter <= 0;
  else
    counter <= counter +1;
end
```

Feedback is perhaps unclear here. See rewrite below.

```verilog
always @(posedge clk) begin
  counter <= counter_comb;
end
```

Feedback across clock cycles is OK.

```verilog
always @ (*) begin
  flag = counter == CNT_MAX;
  counter_comb = flag?0:counter+1;
end
```

No feedback in comb. part.
The keyword `disable` may be used to implement a “break” from a loop. Consider this not yet covered and avoid for now.
Sequential Synthesis: Loops

• Note you may be able describe a **sequential** circuit with non-static loops, but this is commonly NOT SUPPORTED by synthesizers.

```
  count = 0;
  for (index=0; |temp; index=index+1) begin
    @(posedge clk);
    if temp[0]==1 (count=count +1)
    temp>>1;
  end
```

"while" loop with iteration synced to a clock
**Synthesis: Multicycle Operation**

Typical to employ multi-cycle operations to reduce hardware through resource sharing (reuse of hardware in difference clock cycles) and reduce the critical path lengths.

```verilog
always @ (posedge clk)
begin
    temp = a * b;
    @(posedge clk)
    y = temp * c;
end
```

- We’ll want to understand how to implement multi-cycle operations using state machines
Key Points

- Combinatorial data dependency loops cannot be synthesized.
- Static for loops can be synthesized by being unrolled.
- Dynamic for loops may not be understood by the synthesizer.
- Dynamic for loops with timing control may be synthesized as a “multicycle operation” or a state machine.
- We'll want to formalize multi-cycle operations as state machines.
A structural “for loop”: For...Generate

- Uses a special indexing variable defined using genvar. Within a generate block indicated using keyword generate

- Use for repetitive structural instantiations

```verilog
genvar index;
generate
  for (index=0; index < 8; index=index+1)
    begin: gen_code_label
      BUFR BUFR_inst (  
        .O(clk_o(index)), // Clock buffer output  
        .CE(ce), // Clock enable input  
        .CLR(clear), // Clock buffer reset input  
        .I(clk_i(index)) // Clock buffer input  
      );
    end
endgenerate
```
An adder is a good example since it involves a repetitive structure with interconnection between instantiations (*use for param)

genvar index;
genenerate
for (index=0; index < 8; index=index+1)
    begin: gen_code_label
        adder adder_inst (  
            .cin(c[index]),
            .a(a[index]),
            .b(b[index]),
            .cout(c[index+1]),
            .y(u[index])
        );
    end
endgenerate
Concluding Points

- Combinatorial dependency loops cannot be synthesized.
- Static for loops can be synthesized by being unrolled.
- Dynamic for Loops may not be synthesizable.
- Dynamic for Loops with timing control may be synthesized as a “multicycle operation” or a state machine.
- Repetitive/Patterned Structural Instantiations may be done with for...generate loops.
- We'll want to formalize multi-cycle operations as state machines.