Bridge of Life Education

SOC Design

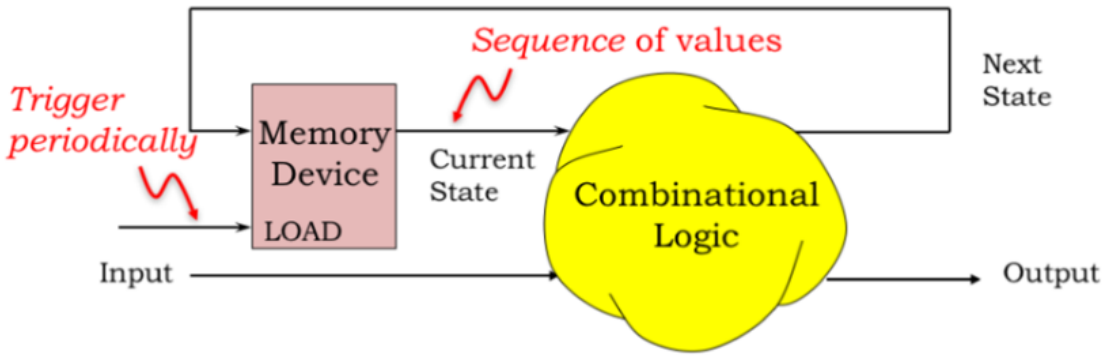
Structure Design

Topics

- Structure Design in Verilog
- IP Coding Style for Basic Combinational/Sequential Logic
  - Template
  - Class
  - Recursive
- A HLS NoC Design
- Composing Module with Forward path only
- Composing Module with Feedback path
- Evaluating Order Discipline

Can we use C/C++ HLS for pure structural design ?

Describe Structure Design in Verilog and C/C++




```
// Combinational Logic
always @(Input, CurrentState) begin
   assign NextState = F(Input, CurrentState);
   assign Output = Y(Input, CurrentState);
end
```

Combinational logic

```
// Register/Flip-flop instantiation
always @( posedge clk ) begin
                CurrentState <= NextState;
end
```

State variables

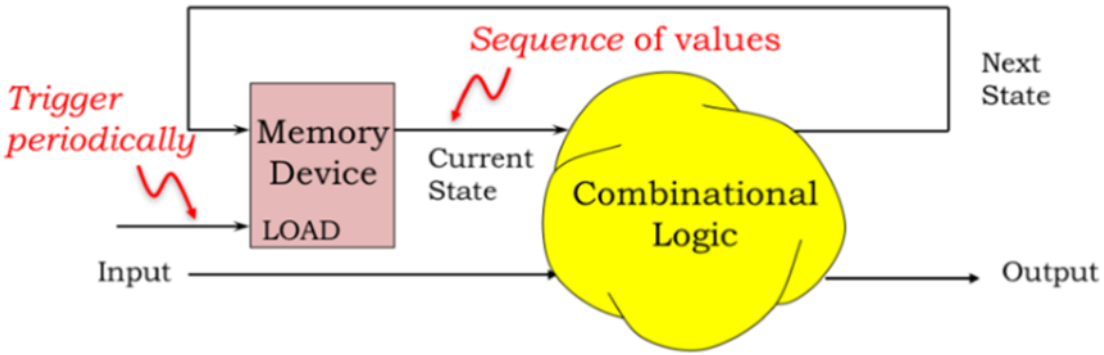
```
struct InS { bool In[N]; }
Struct OutS{  bool Out1[M]; }
```

```
struct State {  bool S[L]; }
```

```
State F(InS& InV, State& CurState) {
  State NextS= 0;
  // function
  return  NextS;
}

OutS Y(InS& InV, State& CurState) {
   OutS Temp_Out = 0;
   // Function Y
   return Temp_Out;
}
```

```
void Top(InS& Input, OutS& Output)
  static State CurState =0;

   Output = Y(Input, CurState);
   CurState = F(Input, CurState);
}
```

A Template to Implement Sequential Logic



Enumerate type to define list of states
void function_name (arguments) {
  states by defining **static variables**
  next-state variable definition
  output-local variable definition

  switch-case to implement all transitions

  Update states
  assign output-variables to output arguments
}

Example Code: Combinational-Lock, ParallelSerialParallel
https://drive.google.com/drive/folders/1aW0c6s7zn1Y-FTvQo3MQJUIW8N65l8nm?usp=share_link

An Example : Counter

states by defining static variables

next-state variable definition

output-local variable definition

switch-case to implement all transitions

Update states

assign output-variables to output arguments

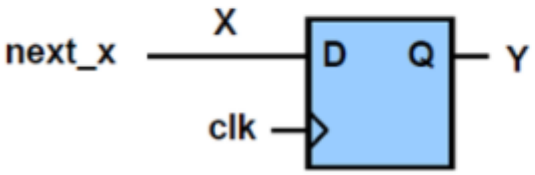
```
typedef enum{wait_for_one, wait_for_zero} counter_state_type;
void counter(
        bool        count_button,
        ap_uint<8> &seven_segments_data,
        ap_uint<4> &seven_segments_enable) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=count_button
#pragma HLS INTERFACE ap_none port=seven_segments_data
#pragma HLS INTERFACE ap_none port=seven_segments_enable

    static ap_uint<5>         number = 0;
    static counter_state_type state  = wait_for_one;

    ap_uint<5>         next_number;
    counter_state_type next_state;
    bool out_local;

    switch(state) {

    case wait_for_one:
        if (count_button == 1) {
            if (number+1 == 10) {
                next_number = 0;
            } else {
                next_number = number+1;
            }
            next_state  = wait_for_zero;
        } else {
            next_number = number;
            next_state  = wait_for_one;
        }
        break;
    case wait_for_zero:
        if (count_button == 1) {
            next_number = number;
            next_state  = wait_for_zero;
        } else {
            next_number = number;
            next_state  = wait_for_one;
        }
        break;

    default:
        break;
    }
    number = next_number;
    state  = next_state;
    seven_segments_data = get_seven_segment_code(number);
    seven_segments_enable = 0b1110;
}
```

```
ap_uint<8> get_seven_segment_code(ap_uint<5> number) {
#pragma HLS INLINE

    ap_uint<8> code = seven_segment_code[0];
    switch(number) {

    case 0:
        code = seven_segment_code[0];
        break;
    case 1:
        code = seven_segment_code[1];
        break;
    case 2:
        code = seven_segment_code[2];
        break;
    case 3:
        code = seven_segment_code[3];
        break;
    case 4:
        code = seven_segment_code[4];
        break;
    case 5:
        code = seven_segment_code[5];
        break;
    case 6:
        code = seven_segment_code[6];
        break;
    case 7:
        code = seven_segment_code[7];
        break;
    case 8:
        code = seven_segment_code[8];
        break;
    case 9:
        code = seven_segment_code[9];
        break;
    default:
        break;
    }

    return code;
}
```

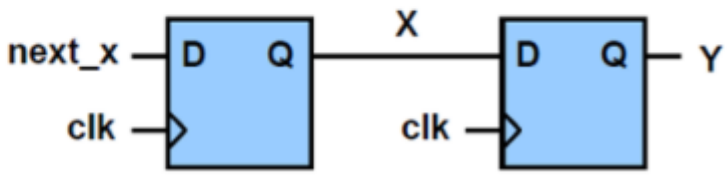

https://drive.google.com/drive/folders/1eKBjmZRUdKzConHTvstnC6owNiMx6x5l?usp=sharing

Blocking v.s. Non-blocking

```
always @( posedge clk )
begin
   x = next_x;
   y = x;
end
```




```
always @( posedge clk )
begin
   x <= next_x;
   y <= x;
end
```

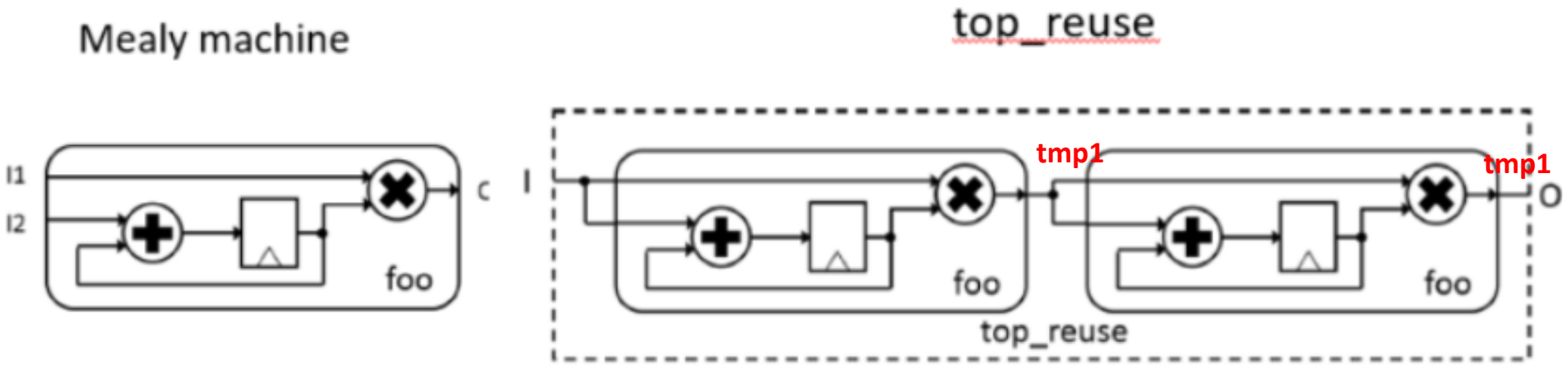


C/C++: How to represent non-blocking behavior?

Remark

- The synthesized module will execute one invocation of this function to completion once per cycle
- a function invocation captures the events of a clock cycle, starting with the combinational propagations based on current state and input values, ending with the synchronous next-state update.
- sequential states using static variables that retain their values across invocations – declared *static*
- Pragma
  - Function: **ap_ctrl_none**
  - Output : **ap_none** : remove the default control signals that become extraneous
  - **#pragma HLS LATENCY max=0** : Force HLS to ignore the target clock period restriction.

Composing Modules – Forward Path




```
void foo(int I1, int I2, int *O) {
   static int L; // latch

   *O = I1*L;  // read current-L
    L  = I2+L;  // assign next-L
}
```

```
void fxn_reuse_try(int I, int *O) {
   int tmp; // output of left module

   foo(I,I,&tmp);      // left in figure
   foo(tmp,tmp,O); // right in figure
}
```

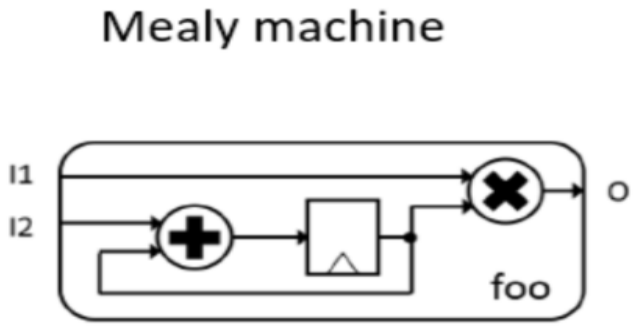
```
void fxn_reuse_try(int I, int *O) {
   int tmp; // output of left module

   foo<1> (I, I  ,  &tmp);  // left
   foo<2> (tmp,tmp,O); // right
}
```

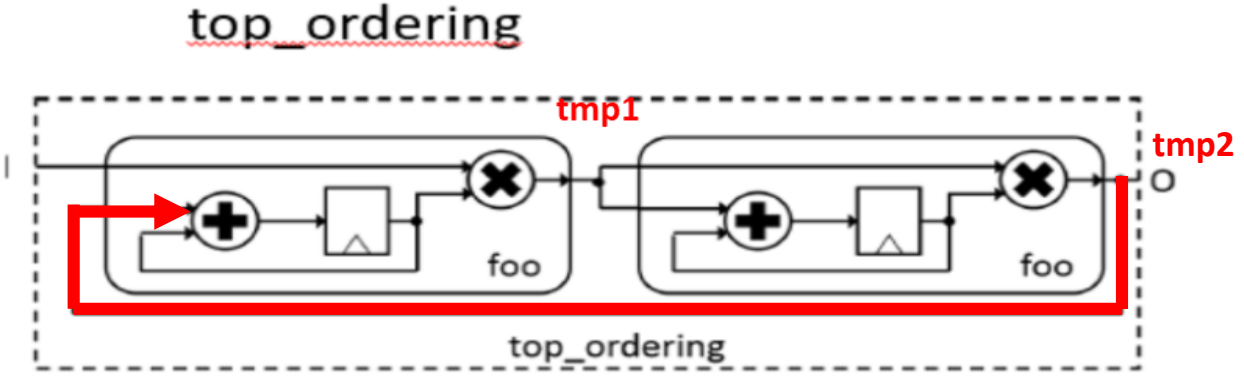
- **Function call in C != module instantiation**
- Two executions of the same function instance, not two copies of the function
- Repeated calls update the same static variable L
- Use **templatized function** with different instance name

Function call in C != module instantiation

- In C: Two executions of the same function instance, not two copies of the function
- In HLS:
  - multiple calls to a purely combinational function (no side-effect through static or global variables) from a combinational context will result in replicated instances.
  - Multiple calls to a purely combinational function from a sequential context will result in a single instance reused over different clock cycles. e.g. function call in a for-loop
- Use C++ template function to uniquify

Composing Modules with Feedback in C






```
void foo(int I1, int I2, int *O) {
   static int L; // latch

   *O=I1*L; // read current-L
   L=I2+L; // assign next-L
}
```

~~void fxn_ordering_try(int I, int *O) {~~
~~int tmp1; // output of left module~~
~~int tmp2; // output of right module~~

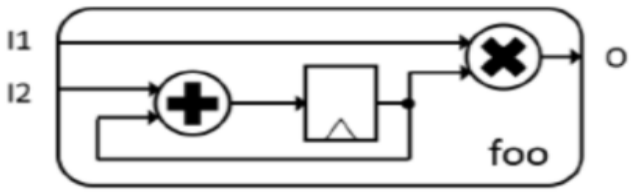
~~foo<1>(I,tmp2,&tmp1); // left in figure~~
~~foo<2>(tmp1,tmp1,&tmp2); // right in figure~~
~~*O=tmp2;~~
~~}~~

Function Evaluation is not Reactive.

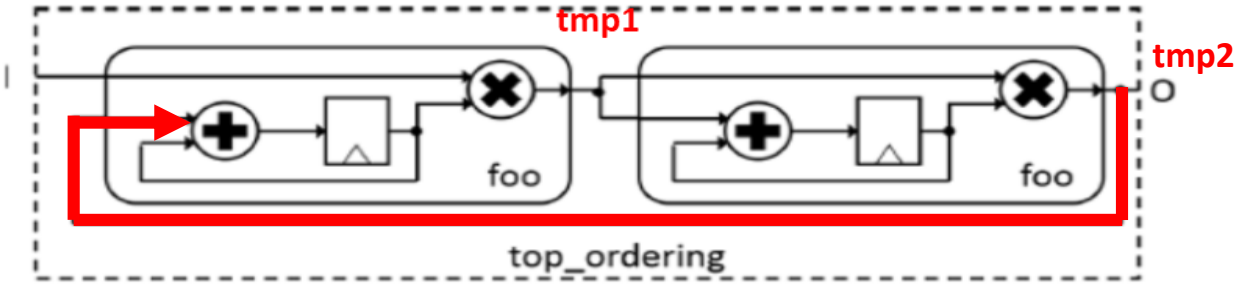
- In C: Evaluate only when they are called and in the order they are called, not re-evaluate
- In Verilog:
  - combinational value in a logic circuit re-evaluate spontaneously in reaction to changes in its dependent values
  - All synchronous state updates take place at exactly the same moment as the final act in a clock period
- Use C++ object **class**

Composing Modules with Feedback in C

Mealy machine



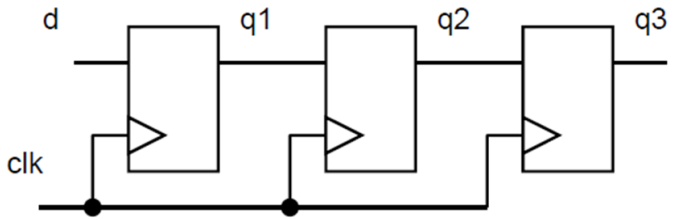
top_ordering




```
class foo_class{
public:
  int L;   // latch
  foo_class() {  L = 1; }
  void query( int I1, int *O) { *O = I1*L;   } // output logic
  void update(int I2)  { L=I2+L;       } // next-state update
};
```

```
void top_ordering(int I, int *O){
  static foo_class foo1,foo2;
  int tmp1, tmp2;
            //combinational-query behaviors
            foo1.query(I,&tmp1);
            foo2.query(tmp1,&tmp2);
            *O=tmp2;
            //state-update behaviors
            foo1.update(tmp2);
            foo2.update(tmp1);
}
```

Evaluation Order Discipline




```
module pipeb2 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q3 = q2;
    q2 = q1;
    q1 = d;
  end
endmodule
```

Simple Forward Pipeline

- **From downstream to upstream**

Pipeline with Feedback

- Order for query method: Obey data dependence.
  - Output must be purely combinational
- Order for update method:
  - The update method must be the last invoked on the object after all query methods have been called.
  - An update method can only be called after all query methods depending on any of the affected member variables have been called.
  - General discipline: update all member variables in an object.

Software v.s. Hardware Difference

- In software, a statement is evaluated once in a sequential manner v.s. in event-driven hardware scheduling, out-of-order, and re-evaluate if RHS variables/signals change.
- In software C program, statements are evaluated in a blocking manner, vs. in hardware, it is non-blocking, i.e., runs concurrently.
- In the software C program, the function call is not equal to module instantiation in hardware. Multiple function calls only instantiate the function once. The problem is especially troublesome if static variables are used. To have multiple distinct functions (module), it uses a class object.
- When composing multiple modules, use template class to replace function.
- Provide two methods, “query” for output generation, “update” for next state. Follow the order rules for the invocation of query and update methods.