ECE 351
Verilog and FPGA Design

Lecture 11: **Review Roy's HW #2 solution**
Decision statements
For loops and loop unrolling
Review for midterm exam
Other looping statements

Roy Kravitz
Electrical and Computer Engineering Department
Maseeh College of Engineering and Computer Science

Portland State UNIVERSITY

Review Homework #2 Solution

Roy's solution: ..\examples\ece351sp20_hw2_solutions

Portland State UNIVERSITY

Decision Statements

Source material drawn from:

- Roy's ECE 351 and ECE 571 lecture material
- *RTL Modeling with SystemVerilog* by Stuart Sutherland
- *Logic Design and Verification Using SystemVerilog* by Donald Thomas

SystemVerilog `case`

Compares 0, 1, x, z by position

```
case (case_expression)
     case_item1 : case_item_statement1;
     case_item2 : case_item_statement2;
     case_item3 : case_item_statement3;
     case_item4 : case_item_statement4;
     default : case_item_statement5;
endcase
```

```
if (case_expression === case_item1) case_item_statement1;
     else if (case_expression === case_item2)
               case_item_statement2;
     else if (case_expression === case_item3)
               case_item_statement3;
     else if (case_expression === case_item4)
               case_item_statement4;
     else   case_item_statement5;
```

- The typical use of a case statement is to:
  - specify a variable as the case expression, and then list explicit values to be matched as the list of case selection items

SystemVerilog case (cont'd)


```
case (case_expression) // case statement header
  case_item_1 : begin
    case_statement_1a;
    case_statement_1b;
  end
  case_item_2 : case_statement_2;
  default     : case_statement_default;
endcase
```

```
case (case_expression)
  case_item1  :  case_item_statement1;
  case_item2  :  begin
                 case_item_statement2a;
                 case_item_statement2b;
                 case_item_statement2c;
                 end
  case_item3  :  case_item_statement3;
  default     :  case_item_statement5;
endcase
```

- Case statement header
  - case, casez, casex keyword
- Case expression
  - constants (e.g. 1'b1), an expression that evaluates to a constant, or a vector
- Case item
  - the expression that is compared against the case expression.
  - C-style break is implied following each case item statement
- Case item statement
  - one or more statements that is executed if the case item matches the current case expression.
  - If more than one statement is required, they must be enclosed with begin…end
- Case default
  - optional, but can include statements to be executed if none of the defined case items match the current case expression

SystemVerilog case (cont'd)

The SystemVerilog standard specifically defines that case statements must evaluate the case selection items in the order in which they are listed

- This infers that:
  - there is a priority to the case items
  - treated the same as in a series of if...else...if decisions
  - If there are overlapping case items the first one that matches is selected
- Software tools such as synthesis compilers will typically try to optimize out the additional logic required for priority encoding the selection decisions if the tool can determine that all of the selection items are mutually exclusive
- Simulation and synthesis might interpret case statements differently.
  - Improperly coded case statements can cause unintended synthesis optimizations or unintended latches which could easily lead to non-functional chips

Unique and priority case

- SystemVerilog provides the `unique`, `priority`, and `inside` modifiers for case decisions.
- These modifiers are placed before the `case` keyword:

```
unique case
(<case_expression>)
   ... // case items
endcase
```

```
priority case
(<case_expression>)
   ... // case items
endcase
```

- Designed to address coding traps:
  - Give information to synthesis to aid optimization
  - Assertions (simulation error reporting mechanisms)
  - Imply that there is a case item for all the possible legal values that case expression might assume

`unique case`

- `unique` is an assertion which implies:
  - All possible values of case expression are in the case items
  - At simulation run time, a match must be found in case items
  - At run time, only one match will be found in case items
- `unique` guides synthesis:
  - It indicates that no priority logic is necessary
  - This produces parallel decoding which may be smaller/faster
- `unique` usage
  - Use when each case item is unique and only one match should occur
  - Using a "default" case item removes the testing for non-existent matches, but the uniqueness test remains
- The `unique` modifier allows designers to explicitly specify that
  - the order of the case selection items is not significant
  - the selections are permitted to be evaluated in parallel
  - Software tools can optimize out the inferred priority of the selection order

priority case

- A `priority case` statement specifies that:
  - At least one case select expression must match the case expression when it is evaluated
  - If more than one case select expression matches the case expression when it is evaluated, the first matching branch must be taken
- Priority is an assertion which implies:
  - All possible values for case expression are in case items
  - Using a "default" case item
    - causes the priority requirement to be dropped since all cases are available to be matched
    - indicates that more than one match in case item is OK
- Priority guides synthesis
  - Indicates that all other testable conditions are don't cares and may be used to simplify logic which can produce logic which is possibly smaller and/or faster

priority case statements (cont'd)


```
module decoder (output logic [3:0] y,
                input logic [1:0] a);
always_comb
   priority case (a)
      0 : y = 1;
      1 : y = 2;
      2 : y = 4;
   endcase
endmodule
```

What if a = 3?

- Simulation: warning
- Synthesis: OK

priority case statements (cont'd)


```
module decoder (output logic [3:0] y,
                input logic [1:0] a);
always_comb
   priority case (a)
      0 : y = 1;
      1 : y = 2;
      2 : y = 4;
      default: y = 3;
   endcase
endmodule
```

What if a = 3?

- Simulation: OK, no warning.
- Synthesis: OK

case...inside


```
casez (case_expression) inside
 case_item1 : case_statement1;
 case_item2 : case_statement2;
 case_item3 : case_statement3;
 case_item4 : case_statement4;
 default : case_item_statement5;
endcase
```

One way masking:

case expression: ZXZ1X

case item : 0?1??

expression	z	x	z	1	x	
item	0	?	1	?	?	
Match (1) /No Match (0) Decision	0	1	0	1	1	0 & 1 & 0 & 1 & 1 = 0 (no match)

- case...inside
  - Only matches wildcards in the case items
  - any X or Z bits in the case expression are not masked
- allows mask bits to be used in the case items

casez

```
casez (case_expression)
 case_item1 : case_statement1;
 case_item2 : case_statement2;
 case_item3 : case_statement3;
 case_item4 : case_statement4;
 default : case_item_statement5;
endcase
```

- z: high-impedance
- x: unknown
- ? : don't care

- casez allows "z" (or "?") to be treated as don't care values in:
  - the case expression
  - the case item when doing case comparison
  - ex: a *case item* 2'b1? in **casez** can match *case expression* of 2'b10, 2'b11, 2'b1x, 2'b1z

Can do the same operation with case inside

casex

- casex
  - allows both "z" and "x" to be treated as don't care values (?) in either the case expression and/or the case item when doing case comparison
  - any x or z bits in both the case expression and case items are masked out from the comparison.
- Everything for casez also applies for casex, plus "x" is now also a wildcard
- The propagated x's can easily cause problems when combined with casex statements
- To avoid these problems, the recommendation is:
  - Do NOT use casex at all for synthesizable code.
  - May be useful for testbenches

If - else conditional statements

- Used to make decisions (to execute or not to execute…that is the question) upon certain conditions
- Keywords are `if` and `else`
- Only permitted in initial and always blocks
- Three types of conditional statements
  - Type 1 - if, no else)
    - if (expression) <true statement>
  - Type 2 – if with one else statement
    - if (expression) <true statement>; else <false statement>;
  - Type 3 – Nested if-else-if statement
    - if (expression 1) <true statement 1>;
      else if (expression 2) <true statement 2>;
      else if (expression 3) <true statement 3>;
      else <default statement>;

If - else Conditional Statements (cont'd)

- Operation
  - <expression> is evaluated
  - If it is true (1 or a non-zero value) the true-statement is executed
  - If it is false (zero) or ambiguous (x) the false-statement is executed
- <expression> can contain any operators
- True-statement and false-statement can each be a single statement or a block of multiple statements
  - Block of statements must be grouped with begin…end
  - A single statement does not need to be grouped but it doesn't hurt to surround it with begin…end

Unique and priority if...else decisions

- The SystemVerilog `unique` and `priority` decision modifiers also work with if...else decisions.
- These modifiers can also reduce ambiguities with this type of decision, and can trap potential design errors early in the modeling phase of a design.
- The if...else statement is often nested to create a series of decisions.

unique if...else


```
logic [2:0] sel;
always_comb begin
unique if (sel == 3'b001) mux_out = a;
else if (sel == 3'b010) mux_out = b;
else if (sel == 3'b100) mux_out = c;
end
```

- In simulation, a series of if...else...if decisions will be evaluated in the order in which the decisions are listed
- To maintain the same ordering in hardware implementation priority encoded logic would be required
- The unique modifier indicates that the designer's intent is that the order of the decisions is not important. Software tools:
  - can optimize out the inferred priority of the decision order
  - will check decision sequence to ensure that all decision conditions in a series of if...else...if decisions are mutually exclusive

priority if...else

```
always_comb begin
priority if (irq0) irq = 4'b0001;
else if (irq1) irq = 4'b0010;
else if (irq2) irq = 4'b0100;
else if (irq3) irq = 4'b1000;
end
```

- The `priority` modifier indicates that the designer's intent is that the order of the decisions is important
- Because the model explicitly states that the decision sequence above should be evaluated in order all software tools should maintain the inferred priority encoding
- The `priority` modifier ensures consistent behavior from software tools
  - Simulators, synthesis compilers, equivalence checkers, and formal verification tools can all interpret the decision sequence in the same way
- a `priority if...else` must specify all conditions thus preventing unintentional latched logic

SystemVerilog `for` loops

Source material drawn from:

- Roy's ECE 351 and ECE 571 lecture material
- *RTL Modeling with SystemVerilog* by Stuart Sutherland
- *Logic Design and Verification Using SystemVerilog* by Donald Thomas

SystemVerilog looping statements

- Allow execution of a programming statement or begin..end group to be executed multiple times
- Looping statements:
  - for
  - repeat
  - while
  - do..while
  - foreach
  - forever

  } restrictions for synthesis

Portland State UNIVERSITY

for loops

- Syntax:

  for (*initial_assignment*; *condition; step_assignment*)
  procedural_statement

- Repeats the execution of the procedural statement a certain number of times
  - *initial_assignment* is the starting value of the loop index
  - *condition* specifies when loop execution must stop; statement(s) in the loop are executed as long as the condition is true
  - *step_assignment* specifies the assignment to modify (typically to increment or decrement the step count
- Ex:

```
integer k;
for (k = 0; k < MAX_RANGE; k = k + 1) begin
  if (hold_data[k] == 0)
    // do something
end
```


Portland State UNIVERSITY

for loops (cont'd)

SystemVerilog permits declaration of the loop variable in the **for** loop

```
module chip (...); // SystemVerilog style loops
  ...
  always_ff @(posedge clock) begin
    for (bit [4:0] i = 0; i <= 15; i++)
      ...
  end

  always_ff @(posedge clock) begin
    for (int i = 1; i <= 1024; i += 1)
      ...
  end
endmodule
```

Scope of *i* is local to the for loops

When declared in this way, the loop variable is an automatic variable and

- Cannot be referenced hierarchically
- Has no existence (or value) outside the loop

for loops (cont'd)

SystemVerilog: Multiple assignments are supported

```
for (int i=1, j=0; i*j < 128; i++, j+=3)
  ...

for (int i=1, byte j=0; i*j < 128; i++, j+=3)
  ...
```

Synthesizing **for** loops

- Synthesis compilers "unroll" the loop
  - Statement or `begin..end` group is replicated the number of times that the loop iterates
  - For synthesis the number of loop iterations must be a fixed number of times (called a static loop)
- Code synthesizable for loops w/ 0 delay -> result is combinational logic

Loop unrolling

- Static loop (also called a data-independent loop) -> number of iterations can be determined w/o having to know the value of any nets or variables

```
parameter N = 4;
logic [N-1:0] a, b, y;

always_comb begin
  for (int i=0; i<=N-1; i++)
    y[i] = a[i] ^ b[(N-1)-i]; // XOR a and reverse order of b
end
```

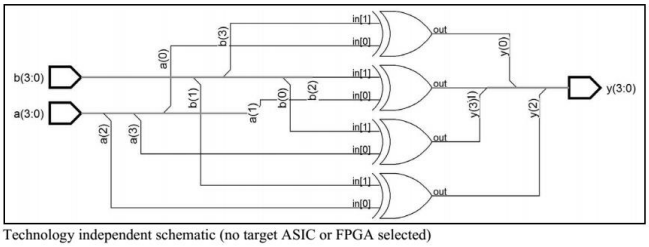
```
always_comb begin
  y[0] = a[0] ^ b[3-0];
  y[1] = a[1] ^ b[3-1];
  y[2] = a[2] ^ b[3-2];
  y[3] = a[3] ^ b[3-3];
end
```

Loop unrolling (cont'd)

```
module bus_xor
#(parameter N = 4)            // bus size
(input  logic [N-1:0] a, b,   // scalable input size
 output logic [N-1:0] y       // scalable output size
);

  always_comb begin
    for (int i=0; i<N; i++) begin
      y[i] = a[i] ^ b[(N-1)-i]; // XOR a and reverse order of b
    end
  end

endmodule: bus_xor
```



Technology independent schematic (no target ASIC or FPGA selected)

Source: Sutherland Fig. 6.7

Loop unrolling (cont'd)

- Data-dependent loops cannot be synthesized because synthesis compiler cannot determine the number of times to replicate the logic inside the for loop

```
always_comb begin
  // find lowest bit that is set in a 32-bit vector
  low_bit = '0;
  end_count = 32;
  for (int i=0; i<end_count; i++) begin
    if (data[i]) begin
      low_bit = i;
      end_count = i;  // cause loop to terminate early
    end
  end
end
```

Loop unrolling (cont'd)

```
module find_lowest_bit
#(parameter N = 4)              // bus size
(input  logic [N-1:0]           data,
 output logic [$clog2(N):0]     low_bit
);

  logic done;  // local flag

  always_comb begin
    // find lowest bit that is set in a vector
    low_bit = '0;
    done = '0;
    for (int i=0; i<=N-1; i++) begin
      if (!done) begin
        if (data[i]) begin
          low_bit = i;
          done = '1;
        end
      end
    end
  end
endmodule: find_lowest_bit
```

Source: Sutherland Fig. 6.8

Review for midterm exam

Review for Midterm exam

- Exam will be open book, open notes, open internet
  - You will need internet and D2L access to complete the exam
  - Unless the problem specifically says so we are not going to deduct points for syntax errors (missing ; etc.) but we do expect your code to be indented and formatted so that we can read it
  - I do not recommend trying to simulate your answers – it will take too long and you have a strict deadline…however using a context-sensitive text editor like Notepad++ or the ModelSim editor for the programming questions is allowed and encouraged.
- The exam will be:
  - ~60% - T/F, multiple choice, short answer
  - ~40% - SystemVerilog programming
- All students in the class who have not made special arrangements with me are expected to take the exam on Thursday from 2:00 PM – 4:30 PM (extra 30 minutes, but still in time to make a 4:40 PM class)

Review for midterm exam (cont'd)

- Fair game for exam:
  - SystemVerilog language rules (literals, vectors, part and part selects, net and var types, etc.)
  - Modules, ports, hierarchy, port by name, port by position, instantiating modules, parameters, etc.)
  - User-defined types (typedef, enum, struct, union, etc)
  - RTL expression operators (bitwise and reduction, logical, arithmetic, shifts, etc.)
  - Continuous assigns and procedural blocks (always and its variants, initial, blocking and non-blocking)
  - Decision statements (case, if..else)
  - For loops and loop unrolling
- Will not use the Quiz mechanism in D2L
- You can expect to write SystemVerilog code

Test process

- There will be a Zoom meeting during the exam time and all who are taking the exam are expected to be on it (mic and video muted).
  - You can ask questions either privately to me or to everyone using the chat and I will answer the same way (either privately or to everyone)
  - You can request via chat that we move to a private Zoom breakout room if you'd rather talk
  - I will project a list of questions/answers via the document camera as they come in…so check every once in a while
  - I will periodically share a countdown timer so you know how much time you have left
- I will prepare a .zip file containing the exam. The .zip file will contain a:
  - .pdf of the exam
  - a .txt (or .sv) file for each of the questions (i.e. there will be 3 text files if there are 3 questions on the exam)
  - The .zip file will be made available to you for download ~10 minutes before the exam is scheduled to start

Test process (cont'd)

- My preference is that you provide your answers to a problem in the associated .txt file (ex: prob1.txt) using a text editor (like Notepad++) but you may also complete the full exam on the .pdf and scan/upload the completed exam
  - Please use one method or the other (.txt files for all answers or handwritten answers on the .pdf) – do not mix methods.
  - If you choose to take the exam on the "hardcopy" we will deduct points if we cannot read your answer
    - We will likely grade these "hardcopy" exams after the others so there may be a small delay in receiving your grade
- When you have completed the exam please create a single .zip or .rar file with all of your answers and upload it to your Exam #1 dropbox
  - IMPORTANT: You will only be able to submit one file and make one submission so make doubly sure that you submit what you want us to grade – there will be no do-overs
- The dropbox will close at 4:35 PM so allow time to bundle your answers and upload them to D2L before then

More Looping Statements

Source material drawn from:
- Roy's ECE 351 and ECE 571 lecture material
- *RTL Modeling with SystemVerilog* by Stuart Sutherland
- *Logic Design and Verification Using SystemVerilog* by Donald Thomas

repeat loops

- Syntax:
  repeat (*loop_count*)
  procedural_statement
- Executes the procedural statement *Loop_count* times
  - Procedural statement could be a begin…end block of statements
- Ex:

```
repeat (count)
  sum = sum + 10;

repeat (shift_by) begin
  wdog_reg = wdog_reg << 1;
end

accum = repeat(load_count) @posedge(clk_rtc)  //event ctrl
  accum + 1;
```

Synthesizing repeat loops

- Synthesizable if the number of times the loop will iterate is fixed and not dependent on value of something that can change
- A static zero-delay repeat loop will synthesize to combinational logic
  - If output of combinational logic is registered in flip-flops the total propagation delay of the combinational logic mus be less than a clock cyle

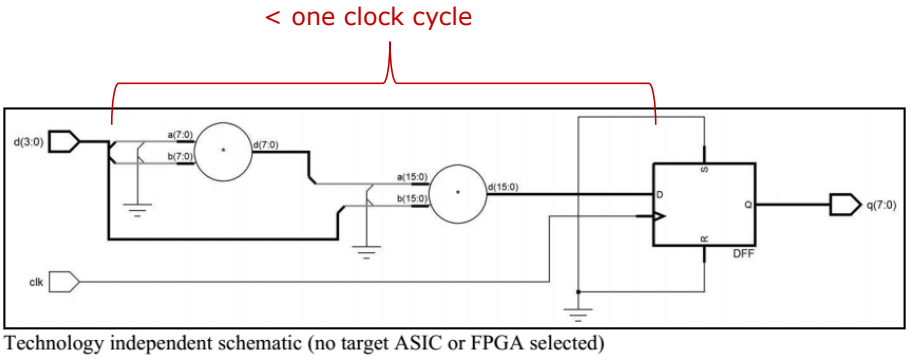
Synthesizing repeat loops (cont'd)


```
module exponential
#(parameter E = 3,     // power exponent
  parameter N = 4,     // input bus size
  parameter M = N*2    // output bus size
)
(input  logic         clk,
 input  logic [N-1:0] d,
 output logic [M-1:0] q
);

  always_ff @(posedge clk) begin: power_loop
    logic [M-1:0] q_temp; // temp variable for inside the loop
    if (E == 0)
      q <= 1;  // do to power of 0 is a decimal 1
    else begin
      q_temp = d;
      repeat (E-1) begin
        q_temp = q_temp * d;
      end
      q <= q_temp;
    end
  end: power_loop
endmodule: exponential
```


Source: Sutherland Ex. 6.9

Synthesizing `repeat` loops (cont'd)



Technology independent schematic (no target ASIC or FPGA selected)

Source: Sutherland Fig. 6.9

`while` loops

- Syntax:
  while (*condition*)
  procedural_statement
- Executes the procedural statement or `begin..end` block of statements *until the specified condition becomes false*
  - If the condition is false to begin with the procedural statement is never executed
  - If the condition is an `x` or a `z` it is treated as false
- Ex:

```
while (shift_by > 0) begin
  acc = acc << 1;
  shift_by = shift_by - 1;
end
```

`do…while` loops

As with C, test is at end of loop so loop always executes at least once

```
always_comb begin
  do begin
    done = 0;
    OutOfBound = 0;
    out = mem[addr];
    if (addr < 128 || addr > 255) begin
      OutOfBound = 1;
      out = mem[128];
    end
    else if (addr == 128) done = 1;
    addr -= 1;
  end
  while (addr >= 128 && addr <= 255);
end
```


Forever Loops

- Syntax:
  forever
  procedural_statement
- Continuously execute the procedural statement
  - Could be a begin…end block of statements
- The only way out of the loop is with a disable statement
- Some form of timing controls must be used otherwise the forever loop will loop forever in zero delay
- Ex:

```
initial begin
  clk1hz = 0;

  #5 forever
    #10 clk1hz = ~clk1hz;
end
```

Statement labels

"Classic" Verilog: permits named blocks to improve readability and to aid hierarchical reference

```
begin: rx_valid_state
  Rxready <= '1;
  breakVar = 1;
  for (int j=0; j<NumRx; j+=1) begin: loop1
    for (int i=0; i<NumRx; i+=1) begin: loop2
      if (Rxvalid[i] && RoundRobin[i] && breakVar)
        begin: match
          ATMcell <= RxATMcell[i];
          Rxready[i] <= 0;
          SquatState <= wait_rx_not_valid;
          breakVar = 0;
        end: match
    end: loop2
  end: loop1
end: rx_valid_state
```


Statement labels (cont'd)

SystemVerilog: allows <u>any</u> procedural statement to be preceded by a label.

```
<label> : <statement>
```

```
always_comb begin : decode_block
  decoder : case (opcode)
    2'b00:
      outer_loop: for (int i=0; i<=15; i++)
        inner_loop: for (int j=0; j<=15; j++)
          //...
    ... // decode other opcode values
  endcase
end : decode_block
```

Uses

- Document code (dubious)
- Tool-specific (identify code for debug, coverage analysis tools)
- **`disable`** can be used to abort execution of named statement

Statement labels (cont'd)

Can also be used to name blocks

```
begin: block1    // named block
  ...
end: block1

block2: begin    // labeled block
          ...
        end
```

Either form works

New loop control statements

"Classic" Verilog provides **disable** which can be used to jump:

- To end of loop (possibly continuing with execution of next pass)
- To break out of the loop completely

```
// find first bit set within a range of bits
always @* begin
  begin: loop
    integer i;
    first_bit = 0;
    for (i=0; i<=63; i=i+1) begin: pass
      if (i < start_range)
        disable pass;  // continue loop
      if (i > end_range)
        disable loop;  // break out of loop
      if ( data[i] ) begin
        first_bit = i;
        disable loop;  // break out of loop
      end
    end // end of one pass of loop
  end // end of the loop
  ... // process data based on first bit set
end
```

Named block

New loop control statements (cont'd)

Can also be used to return early from a task

```
task add_up_to_max (input  [ 5:0] max,
                    output [63:0] result);
  integer i;
  begin
    result = 1;
    if (max == 0)
      disable add_up_to_max; // exit task
    for (i=1; i<=63; i=i+1) begin
      result = result + result;
      if (i == max)
        disable add_up_to_max; // exit task
    end
  end
endtask
```


New loop control statements (cont'd)

SystemVerilog introduces C language **break**, **continue**, **return**

- Not necessary to have named **begin…end** blocks
- No **goto**
- Are synthesizable

```
// find first bit set within a range of bits
always_comb begin
  first_bit = 0;
  for (int i=0; i<=63; i=i+1) begin
    if (i < start_range) continue;
    if (i > end_range)   break;  // exit loop
    if ( data[i] ) begin
      first_bit = i;
      break;  // exit loop
    end
  end  // end of the loop
  ...  // process data based on first bit set
end
```

New loop control statements (cont'd)

SystemVerilog **return** statement can be used in tasks or functions

```
task add_up_to_max (input  [ 5:0] max,
                    output [63:0] result);
  result = 1;
  if (max == 0) return; // exit task
  for (int i=1; i<=63; i=i+1) begin
    result = result + result;
    if (i == max) return; // exit task
  end
endtask

function automatic int log2 (input int n);
  if (n <=1) return 1;  // exit function early
  log2 = 0;
  while (n > 1) begin
    n = n/2;
    log2++;
  end
  return log2;
endfunction
```

Void functions have no return

Next Time

- Midterm e-Exam will be 2:00 PM – 4:30 PM on Thu, 07-May
- Next lecture will be Tue, 12-May Topics:
  - Looping statements (wrap-up)
  - Tasks and functions
  - Modeling combinational logic
- You should:
  - Read Sutherland Ch 7
- Homework, projects and quizzes
  - (tentative) Homework #3 will be assigned Tue, 12-May