CS222: Computer Architecture

Instructor:
Dr Ahmed Shalaby http://bu.edu.eg/staff/ahmedshalaby14#

الاحترام - الادب - الاخلاق
الطالب - المعيد - الدكتور

Combinational Modeling

Combinational Modeling

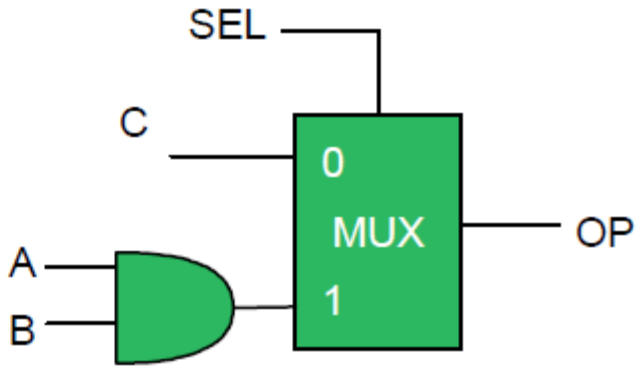
- To learn how to describe combinational logic.

- Topics:
  - RTL always statements.
  - If statements.
  - Incomplete assignment and latches.
  - Conditional Operators.
  - Tristate buffer.
  - Case statements.
  - For, while, repeat and forever loops.

RTL Always Statements

Combinational always

```
reg OP;
...
always @(SEL or A or B or C)
  if (SEL)
    OP = A and B;
  else
    OP = C;
```

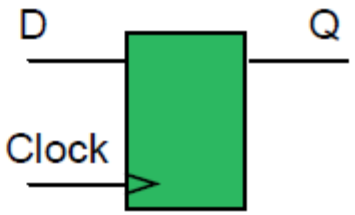


Event control

Clocked always

```
reg Q;
...
always @(posedge Clock)
  Q <= D;
```

"Non-blocking" assignment



Begin-End

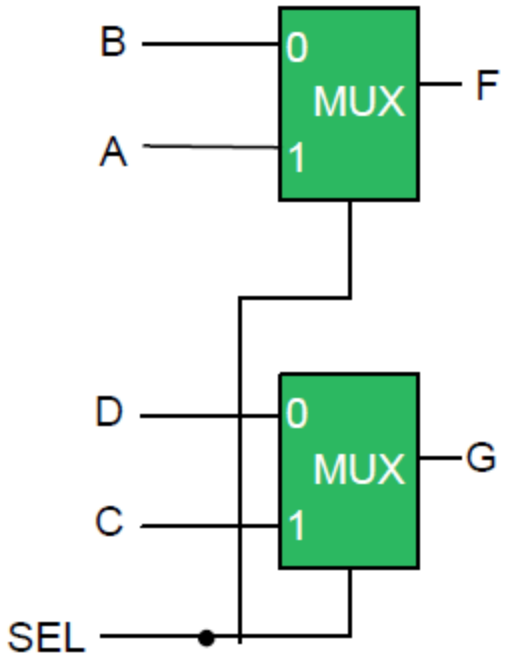
```
always @(SEL or A or B or C or D)
begin

  if (SEL)
  begin
    F = A;
    G = C;
  end
  else
  begin
    F = B;
    G = D;
  end

end
```

begin-end optional

begin-end required



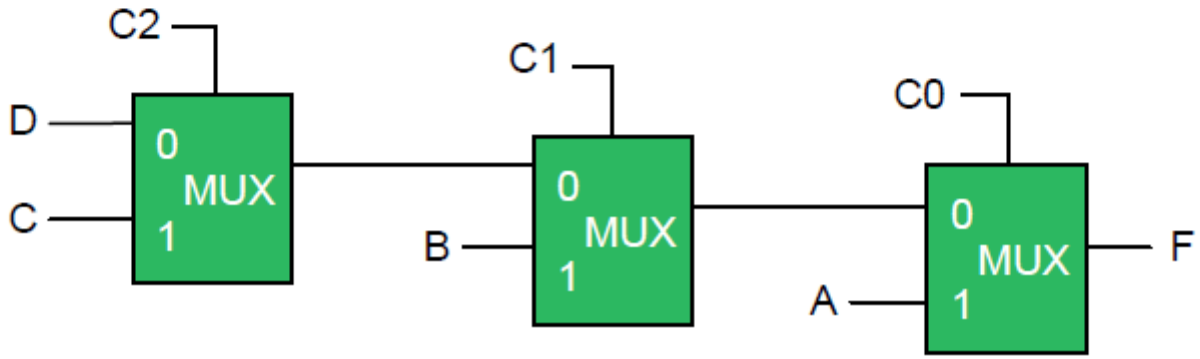
Else If

```
always @(C0 or C1 or C2 or A or B or C or D)
  if (C0)
    F = A;
  else if (C1)
    F = B;
  else if (C2)
    F = C;
  else
    F = D;
```

Complete “sensitivity list”

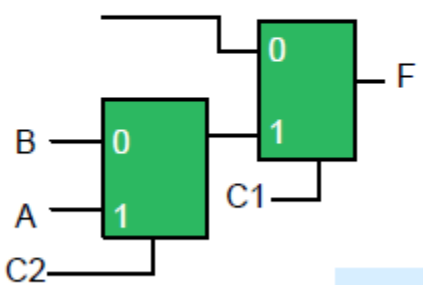
Nested if statements

Else if =>priority



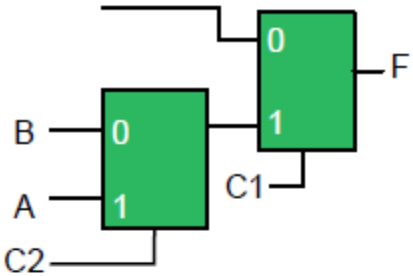
Nested If and Begin-End

```
if (C1)
  if (C2)
    F = A;
  else
    F = B;
```

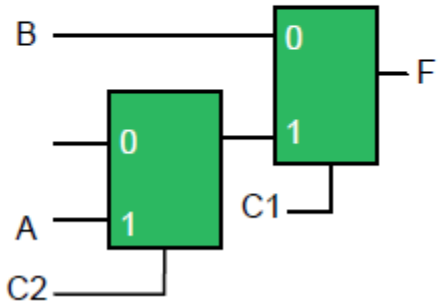


Same

```
if (C1)
  if (C2)
    F = A;
else
  F = B;
```



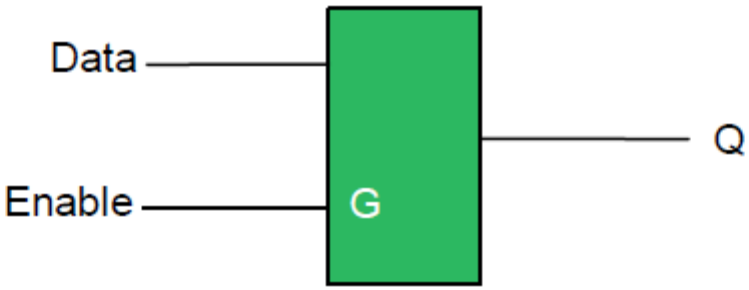

```
if (C1)
begin
  if (C2)
    F = A;
end
else
  F = B;
```



Verilog cannot read your mind!

Incomplete Assignment

```
always @(Enable or Data)
  if (Enable)
    Q = Data;
```



Beware unwanted latches!

Conditional Operator

A ? B : C	Conditional

```
value = A ? B : C;
```

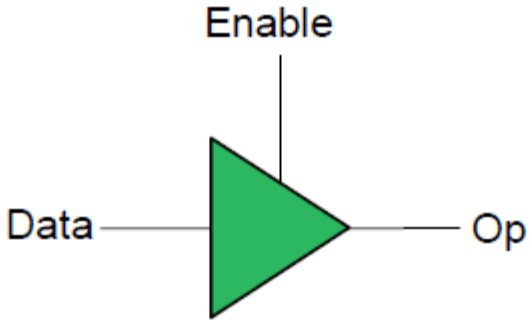
- **Equivalent to:**

```
if (A)
  value = B;
else
  value = C;
```

- **Mux using continuous assignment**

```
assign F = SEL ? A : B;
```

Tristates



Using *assign*

```
assign Op = Enable ? Data : 1'bZ;
```

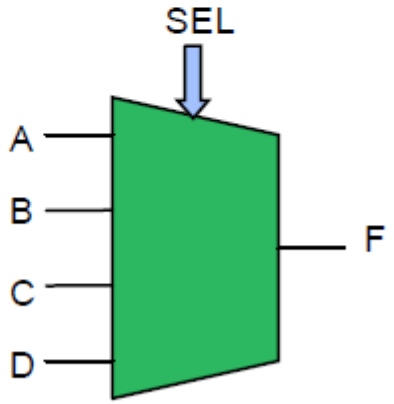
Using *always*

```
always @(Enable or Data)
  if (Enable)
    Op = Data;
  else
    Op = 1'bz;
```

Case Statement

```
always @(SEL or A or B or C or D)
  case (SEL)
    2'b00:   F = A;
    2'b01:   F = B;
    2'b10:   F = C;
    2'b11:   F = D;
    default: F = 1'bX;
  endcase
```

Simulation only



Case Statement (Cont.)

```
case (Code)
  3'b000:
    begin
      P = 1;
      Q = 1;
    end
  3'b001, 3'b010, 3'b100:
    begin
      Q = 1;
      R = 1;
    end
  3'b110, 3'b101, 3'b011:
    R = 1;
endcase
```

begin-end needed here

Alternatives

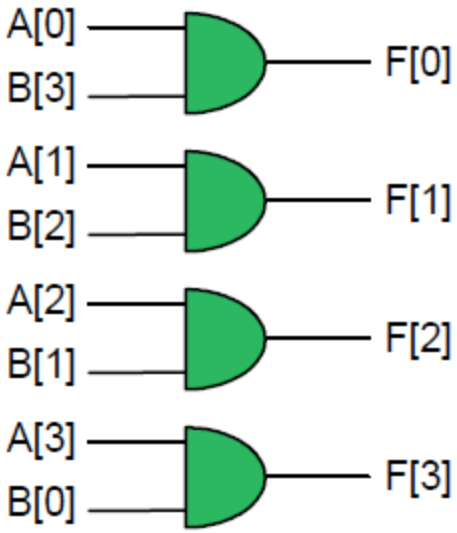
Some cases missing - OK

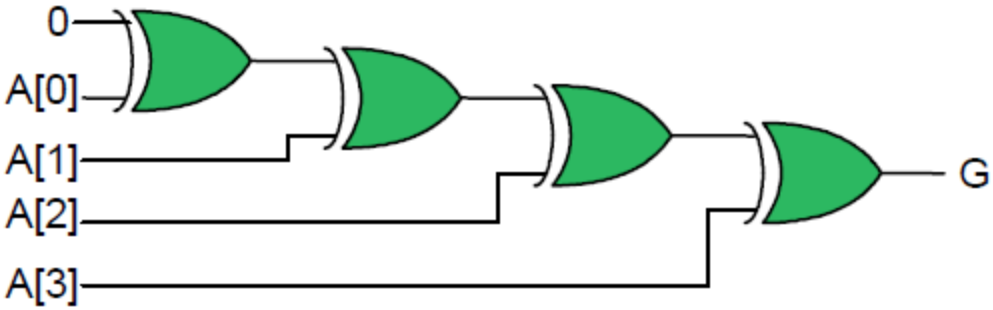
Latches may be inferred

For Loops

```
always @(A or B)
begin
  G = 0;
  for (I = 0; I < 4; I = I + 1)
  begin
    F[I] = A[I] & B[3-I];
    G = G ^ A[I];
  end
end
```

for => repeated H/W





Other Loop Statements

- **The *for* loop can be used for synthesis**

```
for (initialise; condition; assignment)
   ...
```

- **Three other loops used for test fixtures and algorithms**

```
while (condition)
  ...
```

Loops while condition is true

```
repeat (expression)
  ...
```

Loops *expression* times

```
forever
  ...
```

Infinite loop

Sequential Modeling

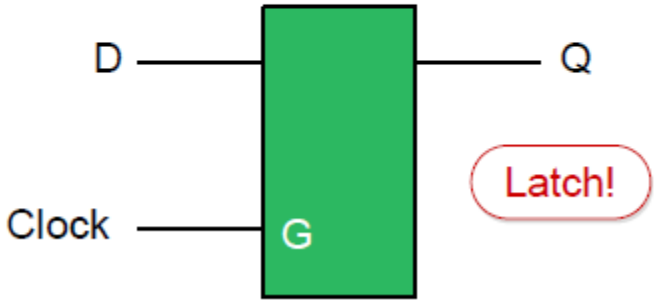
Sequential Modeling

- To learn how to describe Sequential logic.

- Topics:
  - Latch – Flip Flop.
  - Blocking – Non-Blocking assignment.
  - Shift Register.
  - Counter.
  - Finite State Machine (FSM).
  - Memory.

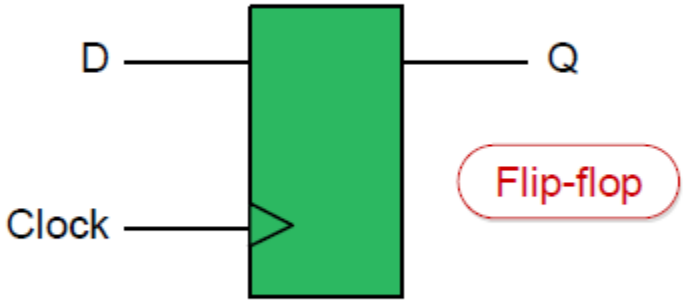
Edge Triggered Flip-Flop

```
always @(Clock)
  if (Clock)
    Q = D;
```




```
always @(posedge Clock)
  Q = D;
```

or negedge



Avoiding Simulation Races

- **Race to read and write b!**

```
always @(posedge clock)
  b = a;

always @(posedge clock)
  c = b;
```

- **To fix the simulation race...**

```
always @(posedge clock)
begin
  tmpa = a;
  #1 b = tmpa;
end

always @(posedge clock)
begin
  tmpb = b;
  #1 c = tmpb;
end
```

Not recommended!

Non-Blocking Assignments

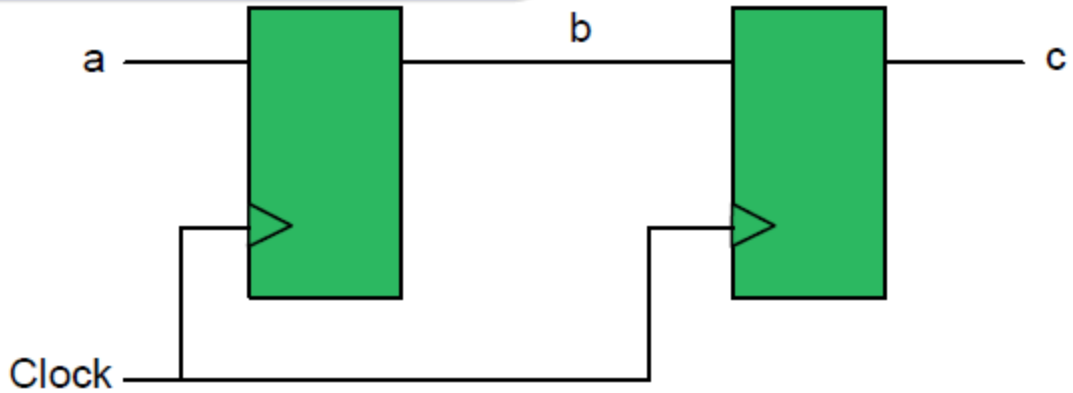
? **Preferred solution – use *non-blocking* assignments**

```
always @(posedge Clock)
  b <= a;

always @(posedge Clock)
  c <= b;
```

Recommended

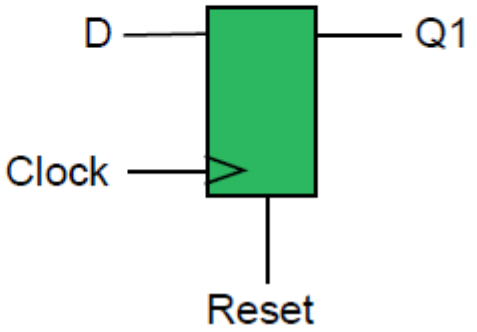
"Non-blocking" or "RTL" assignment



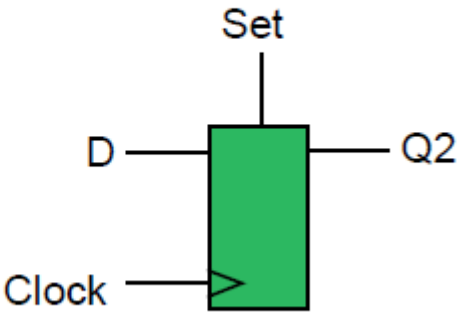
How would you write an asynchronous reset?

Asynchronous Set or Reset

```
always @(posedge Clock or posedge Reset)
  if (Reset)
    Q1 <= 0;
  else
    Q1 <= D;
```




```
always @(posedge Clock or posedge Set)
  if (Set)
    Q2 <= 1;
  else
    Q2 <= D;
```

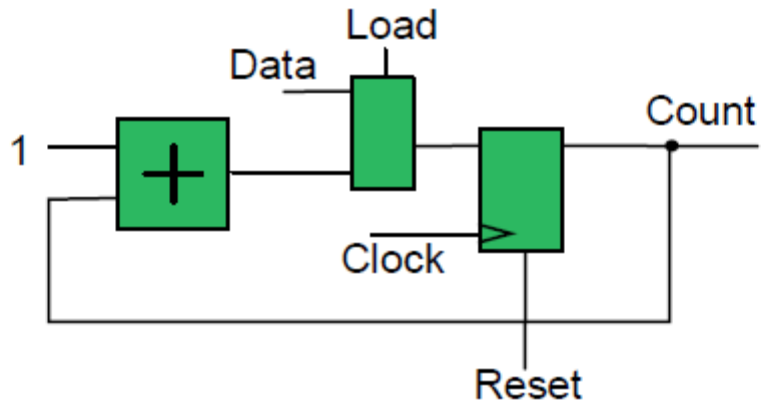


Synchronous vs. Asynchronous Actions

```
always @(posedge Clock or posedge Reset)
  if (Reset)
    Q <= 0;
  else if (Load)
    Q <= Data;
  else
    Q <= Q + 1;
```

Asynchronous reset

Synchronous load

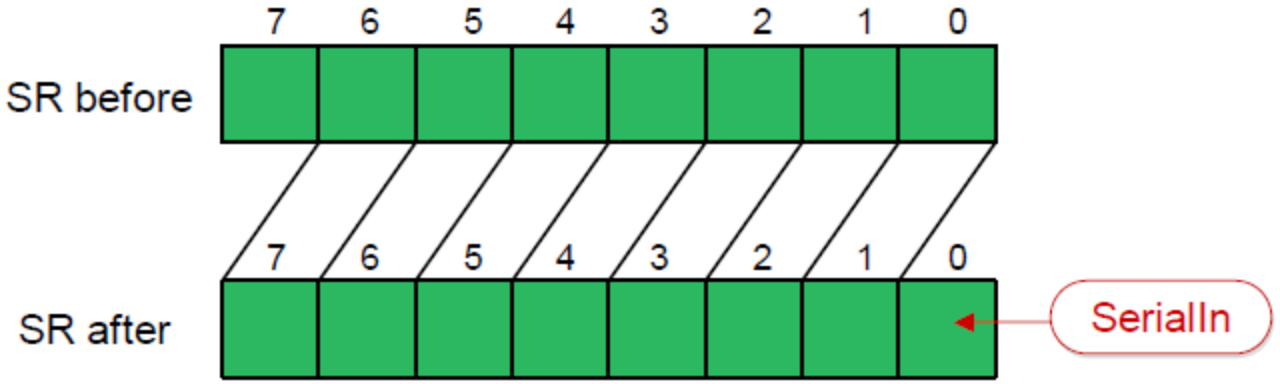


Shift Registers

Shift register using concatenation

```
reg [7:0] SR;

always @(posedge Clock or posedge Reset)
  if (Reset)
    SR <= 8'b0;
  else
    SR <= {SR[6:0],SerialIn};
```

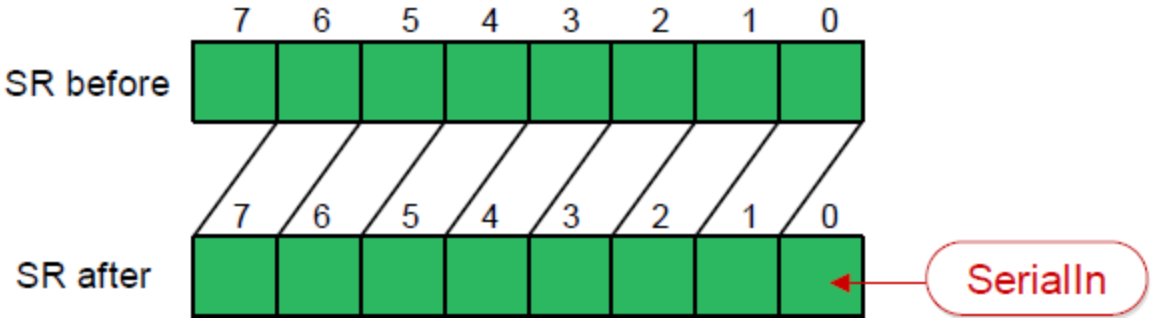


Shift Operators

<<	Left Shift
>>	Right Shift

Shift register using shift operator

```
always @(posedge Clock or posedge
Reset)
  if (Reset)
    SR <= 8'b0;
  else
    begin
      SR <= SR << 1;
      SR[0] <= SerialIn;
    end
```

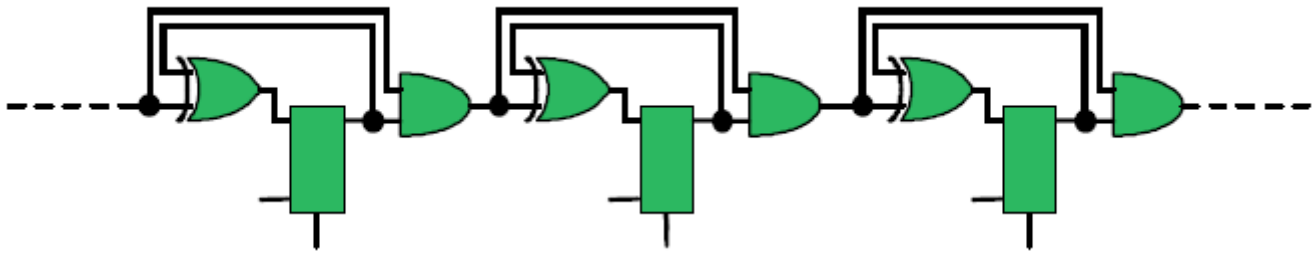


Counters

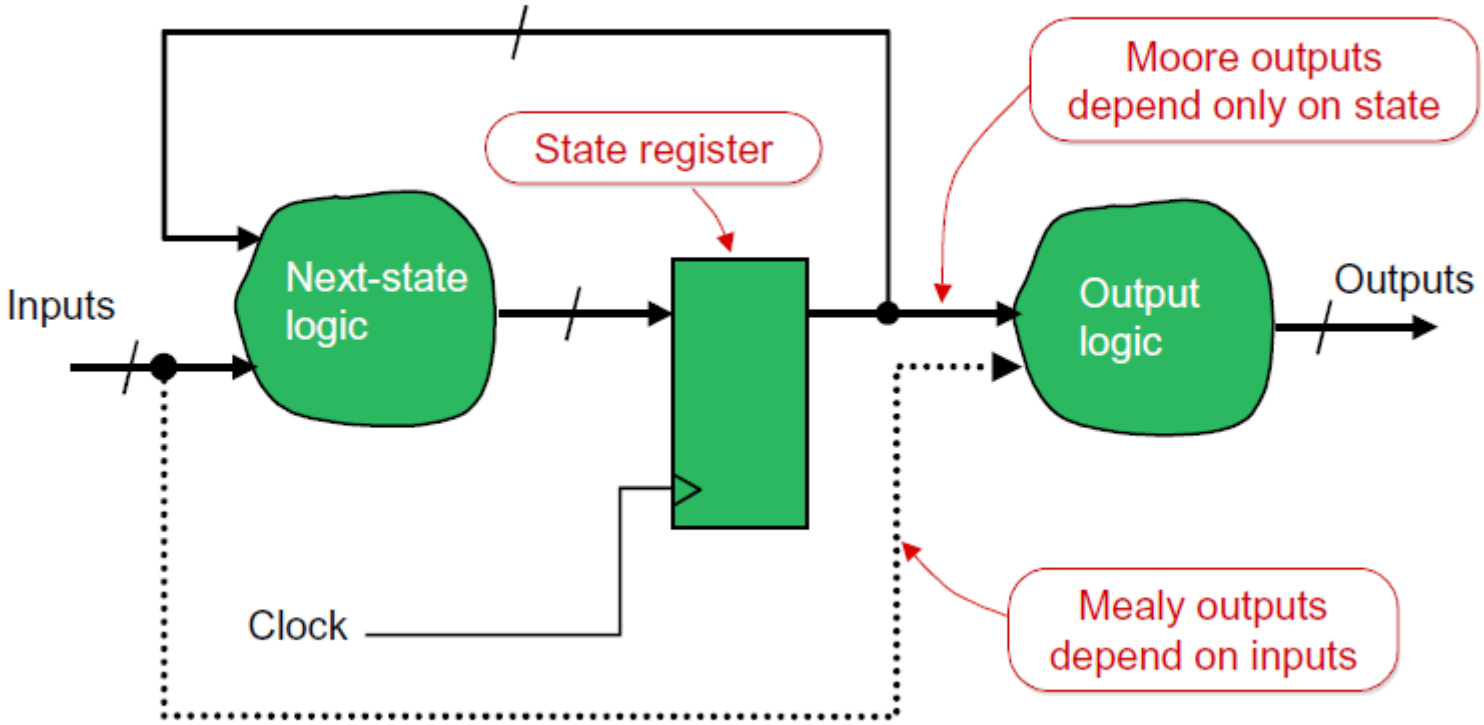
```
reg [3:0] count;

always @(posedge clk)
  count <= count + 1;
```

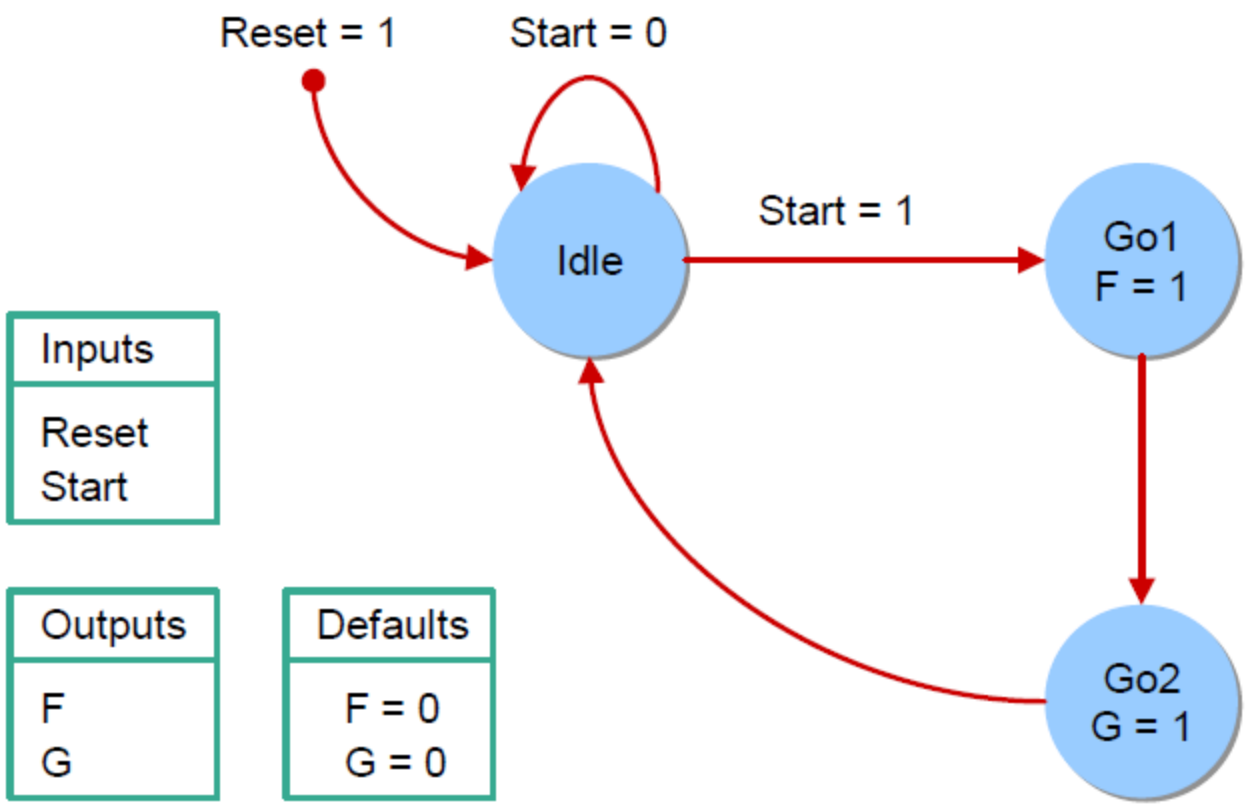
4'b1111 + 1 = 4'b0000



Finite State Machines



State Transition Diagrams



Explicit State Machine Description

```
parameter Idle = 2'b00,
          Go1  = 2'b01,
          Go2  = 2'b10;
reg [1:0] State;
...
always @(posedge Clock or posedge Reset)
  if (Reset)
  begin
    State <= Idle;
    F <= 0;
    G <= 0;
  end
  else
    case (State)
      Idle : if (Start)
             begin
               State <= Go1;
               F <= 1;
             end
      Go1  : begin
               State <= Go2;
               F <= 0;
               G <= 1;
             end
      Go2  : begin
               State <= Idle;
               G <= '0';
             end
    endcase
```

FSM should have a reset

How many flip-flops?

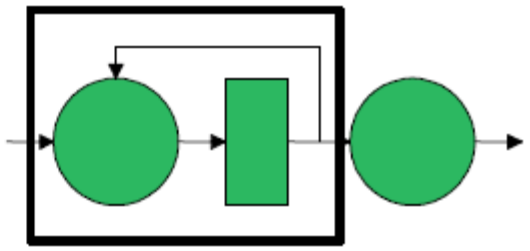
Separate Output Decoding

```
parameter Idle = 2'b00,
          Go1  = 2'b01,
          Go2  = 2'b10;
reg [1:0] State;

...

always @(posedge Clock or posedge Reset)
  if (Reset)
    State <= Idle;
  else
    case (State)
      Idle : if (Start)
               State <= Go1;
      Go1  : State <= Go2;
      Go2  : State <= Idle;
    endcase
```

```
always @(State)
begin
  F = 0;
  G = 0;
  if (State == Go1)
    F = 1;
  else if (State == Go2)
    G = 1;
end
```



Unreachable States

```
parameter Idle = 2'b00,
          Go1  = 2'b01,
          Go2  = 2'b10;
reg [1:0] State;
```

```
case (State)
  Idle : ...
  Go1  : ...
  Go2  : ...
endcase
```

State Name	Binary Encoding
Idle	2'b00
Go1	2'b01
Go2	2'b10
-	2'b11

- 3 states
- 2 flip-flops (binary or gray code)
- 4 hardware states
- 1 unreachable state

No control over 4th state

Logic may be minimized

Controlling Unreachable States

```
parameter Idle  = 2'b00,
          Go1   = 2'b01,
          Go2   = 2'b10,
          Dummy = 2'b11;
reg [1:0] State;
```

Either

```
parameter Idle = 2'b00,
          Go1  = 2'b01,
          Go2  = 2'b10;
reg [1:0] State;
```

```
case (State)
  Idle : ...
  Go1  : ...
  Go2  : ...
  default : State = Idle
endcase
```

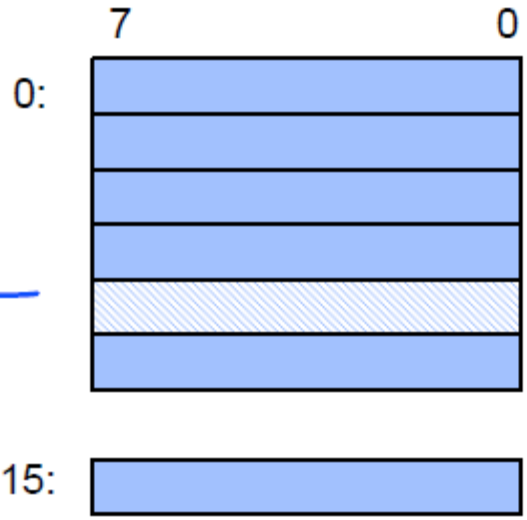
Explicitly define behavior of hardware in unreachable state

Memories

Width

Depth

```
reg [7:0] MEM [0:15];
```




```
reg [7:0] WORD;
reg [3:0] ADDR;


initial
begin
  ADDR = 4;
  WORD = MEM[ADDR];
  MEM[ADDR] = {WORD[3:0], WORD[7:4]};  // Nibble swap
end
```

Memories vs. Regs

- **An eight bit reg**

```
reg [7:0] R;
```

```
R = 8'b00001111;
```

OK

```
R[3:0] = 4'b1111;
```

- **An eight word by 1 bit memory array**

```
reg R[7:0];
```

```
R = 8'b00001111;
```

```
R[3:0] = 4'b1111;
```

Illegal references to memory

```
R[7] = 1'b1;
```

OK

RAMs

```
module smallram (clock, wr, rd, addr, data);
  input clock, wr, rd;
  input [3:0] addr;
  inout [7:0] data;

  reg [7:0] mem [0:15];

  always @(posedge clock)
    if (wr)
      mem[addr] = data;

  assign data = rd ? mem[addr]: 8'bZ;

endmodule
```

4 address bits = 16 addresses

16x8 memory

Dynamic word select

data is an inout...

What would be synthesized?

Loading Memories

? **$readmemb and $readmemh load memories from text files**

```
module rom (addr, data);
  input  [7:0] addr;
  output [7:0] data;
  reg [7:0] arom [0:'H1FF];
  assign data = arom[addr];
  initial
    $readmemb("rom.txt", arom);
endmodule
```

$readmemh - hex data

$readmemb - binary data

rom.txt

```
0000_0101          // Load at address 0
0110_1110          // Load at address 1
10011111 01100111 // Load at addresses 2 & 3
@100               // Skip addresses 4 to FF
1111_1100          // Load at address 100 (Hex)
```

Comments

Address (Hex)

Underscores are allowed

Mid Term-Project

Here are delivery and evaluation rules:

1. التسليم في مجموعات والمجموعة تتكون من ثلاث طلاب.
2. المجموعة من نفس الفصل (السكشن), غير مسموح بتكوين مجموعات خارج الفصل.
3. التسليم في الأسبوع الذى يلى امتحان منتصف الفصل الدراسى.
4. عند التسليم : سيقوم المعيد ب :
   - تسليم FPGA
   - التأكد من جهاز الحاسب و منحك وقت 40 دقيقة ل تسليم المشروع يعمل على FPGA
   - استلام التقرير.
5. **الجدول التالى يوضح درجات التقييم**

Project runs on FPGA + ModelSim Simulator + Report	Full mark + bonus based on the quality of code and report.
Project runs on ModelSim Only + Report	0.8 x Full Mark
Errors in ModelSim Simulator + Report	0.6 x Full Mark
Cheating	Zeros to all members in the Group.