ECE 4514
Digital Design II
Spring 2008

Behavioral Modeling:
Non-blocking assignments

A *Language* Lecture

Patrick Schaumont

Today

- First part on writing *behavioral models* - models which make use of more advanced constructs to design always and initial blocks
- Today: Introduce *non-blocking procedural assignments*
- The first 20 minutes of this lecture are really important

Pay Attention Now
.. it will earn itself back later

The assignment statements so far

	Continuous Assignment	Procedural Assignment
Operation	=	=
Where	using stand-alone *assign* statement	inside of *always* and *initial*

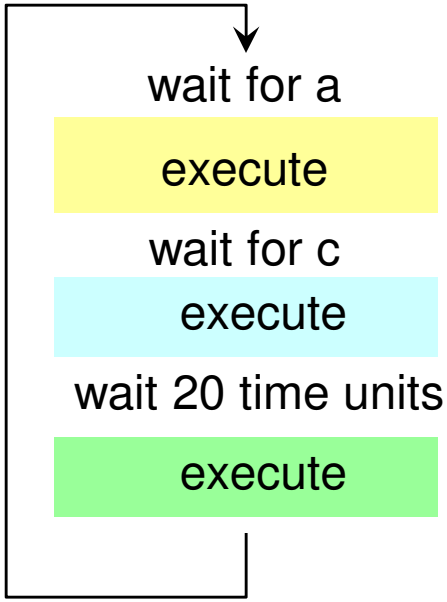
Example	`wire q;`
	`reg a, b;`
	`assign q = a & b;`
		`wire q;`
		`reg a, b;`
		`always @(b)`
		`  a = b + 5;`

Valid LHS	net (wire)	reg
Valid RHS	expression of net or reg	expression of net or reg
Evaluated	when any part of RHS changes	**procedural execution**

What is 'procedural execution' ?

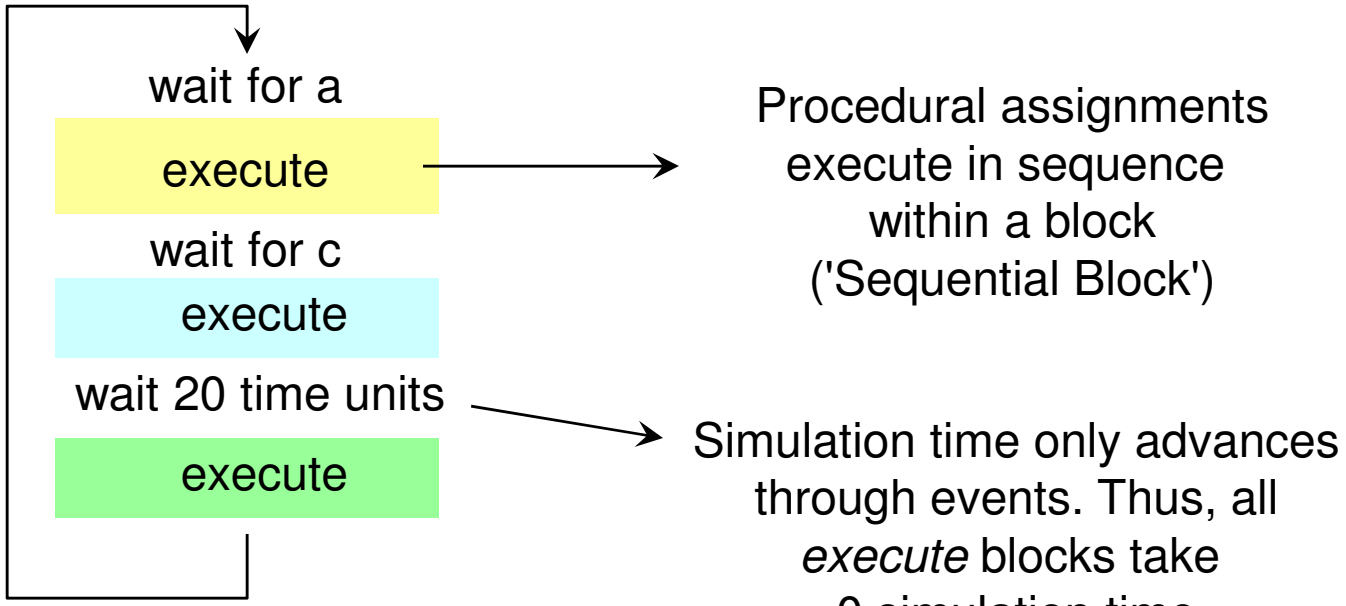
- An always block is either executing, or it is waiting for an event from its sensitivity list
- When it is executing, it proceeds in a sequential fashion through the statements of the code, until it runs into an *event control statement* (@, wait, #)

```
always begin
  @(upedge a);
  b = a + 1;
  wait (c == 3);
  #20 b = a + 1;
end
```



What is 'procedural execution' ?

- An always is either executing, or it is waiting for an event from its *sensitivity list*
- When it is executing, it proceeds in a sequential fashion through the statements of the code, until it runs into an *event control statement* (@, wait, #)



The assignment statements so far

	Continuous Assignment	Procedural Assignment
Operation	=	=
Wh		*ays*
Exa		`a = b + 5;`
Valid LHS	net (wire)	reg
Valid RHS	expression of net or reg	expression of net or reg
Evaluated	when any part of RHS changes	procedural execution

The full name of this assignment is:

Procedural *Blocking* Assignment

Procedural Blocking Assignment

- Execution in a sequential block cannot complete without the assignment being complete

```
reg [7:0] a, b;
always @(posedge clk) begin
   a = b + 2;
   b = a * 3;
end
```

Blocking assignment:

assignment on *a* MUST complete before assignment on *b* can start

Procedural Blocking Assignment

- Execution in a sequential block cannot complete without the assignment being complete

```
reg [7:0] a, b;
always @(posedge clk) begin
   #10 a = b + 2;
   b = a * 3;
end
```

1. **Wait 10;**
2. Read value of b and add 2 to it;
3. Assign the result to a;
4. Read the value of a and multiply by three;
5. Assign the result to b;

Procedural Blocking Assignment

- Execution in a sequential block cannot complete without the assignment being complete

```
reg [7:0] a, b;
always @(posedge clk) begin
   a = #10 b + 2;
   b = a * 3;
end
```

1. Read value of b and add 2 to it;
2. **Wait 10;**
3. Assign the result to a;
4. Read the value of a and multiply by three;
5. Assign the result to b;

Procedural Blocking Assignment

- Execution in a sequential block cannot complete without the assignment being complete

```
reg [7:0] a, b;
always @(posedge clk) begin
   #5 a = #10 b + 2;
   #5 b = a * 3;
end
```

1. **Wait 5;**
2. Read value of b and add 2 to it;
3. **Wait 10;**
4. Assign the result to a;
5. **Wait 5;**
6. Read the value of a and multiply by three;
7. Assign the result to b;

Procedural Blocking Assignment

- Execution in a sequential block cannot complete without the assignment being complete

Blocking Assignment means
a will always be assigned before b

```
reg [7:0] a, b;
always @(posedge clk) begin
   #5 a = #10 b + 2;
   #5 b = a * 3;
end
```

1. **Wait 5;**
2. Read value of b and add 2 to it;
3. **Wait 10;**
4. **Assign the result to a;**
5. **Wait 5;**
6. Read the value of a and multiply by three;
7. **Assign the result to b;**

Procedural Non-Blocking Assignment

	Continuous Assignment	Proc. Blocking Assignment	Proc. Non-Blocking Assignment
Operation	=	=	<=
Where	using stand-alone *assign* statement	inside of *always* and *initial*	inside of *always* and *initial*

Example	`wire q;`
	`reg a, b;`
	`assign q = a & b;`
		`wire q;`
		`reg a, b;`
		`always @(b)`
		`  a = b + 5;`
			`wire q;`
			`reg a, b;`
			`always @(b)`
			`  a <= 5;`

Valid LHS	net (wire)	reg	reg
Valid RHS	expression of net or reg	expression of net or reg	expression of net or reg
Evaluated	when any part of RHS changes	procedural execution	at the end of current time step

What does 'at the end of current time step' mean?

- The simulator processes all the events with a given timestamp according to the nature of the event

Given the set of evaluate-events at time T

First, do the following (in any order)

Evaluate the RHS of *all* assignments
Assign the LHS of all procedural blocking assignments
Assign the LHS of all continuous assignments
Evaluate inputs and outputs of all primitives (gates)
Print $display statements

Then, do the following (in any order)

Change the LHS of all nonblocking assignments

Then, do the following (in any order)

Print $monitor statements

What does 'at the end of current time step' mean?

- The simulator processes all the events with a given timestamp according to the nature of the event

Given the set of evaluate-events at time T

First, do the following (in any order)

Evaluate the RHS of *all* assignments
Assign the LHS of all procedural blocking assignments
Assign the LHS of all continuous assignments
Evaluate inputs and outputs of all primitives (gates)
Print $display statements

So, it looks like non-blocking assignments can 'split' a given point in time in two

Then, do the following (in any order)

Change the LHS of all nonblocking assignments

Then, do the following (in any order)

Print $monitor statements

Effect of a non-blocking statement

Blocking

```
reg [7:0] a, b;

initial b = 0;
initial a = 4;

always @(a or b)
  begin
    a = b + 2;
    b = a * 3;
 end
```

Non-Blocking

```
reg [7:0] a, b;

initial b = 0;
initial a = 4;

always @(a or b)
  begin
    a <= b + 2;
    b <= a * 3;
 end
```

Effect of a non-blocking statement

Blocking

```
reg [7:0] a, b;

initial b = 0;
initial a = 4;

always @(a or b)
  begin
    a = b + 2;
    b = a * 3;
 end
```

a = 0 + 2 = 2
b = 2 * 3 = 6

Non-Blocking

```
reg [7:0] a, b;

initial b = 0;
initial a = 4;

always @(a or b)
  begin
    a <= b + 2;
    b <= a * 3;
 end
```

a = 0 + 2 = 2
b = 4 * 3 = 12

Effect of a non-blocking statement

Blocking

```
reg [7:0] a, b;

initial b = 0;
ini
always @(a or b)
  begin
    a = b + 2;
    b = a * 3;
  end
```

a = 0 + 2 = 2
b = 2 * 3 = 6

Non-Blocking

```
reg [7:0] a, b;

initial b = 0;
ini                    ;
always @(a or b)
  begin
    a <= b + 2;
    b <= a * 3;
  end
```

a = 0 + 2 = 2
b = 4 * 3 = 12

Non-Blocking Assignment means
a and b are updated at the same time

Another example

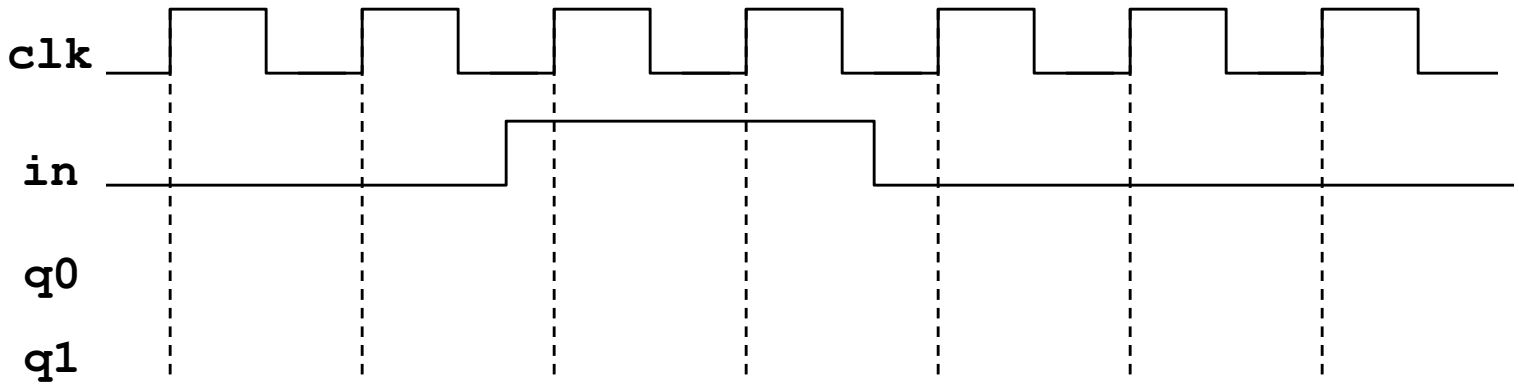
Using blocking assignments:

```
module fsm_mod(q1, q0, in, clk);
    output q1, q0;
    input clk, in;
    reg q1, q0;

    always @(posedge clk) begin
        q1 = in;
        q0 = in | q1;
    end
endmodule
```

Another example

Using blocking assignments:

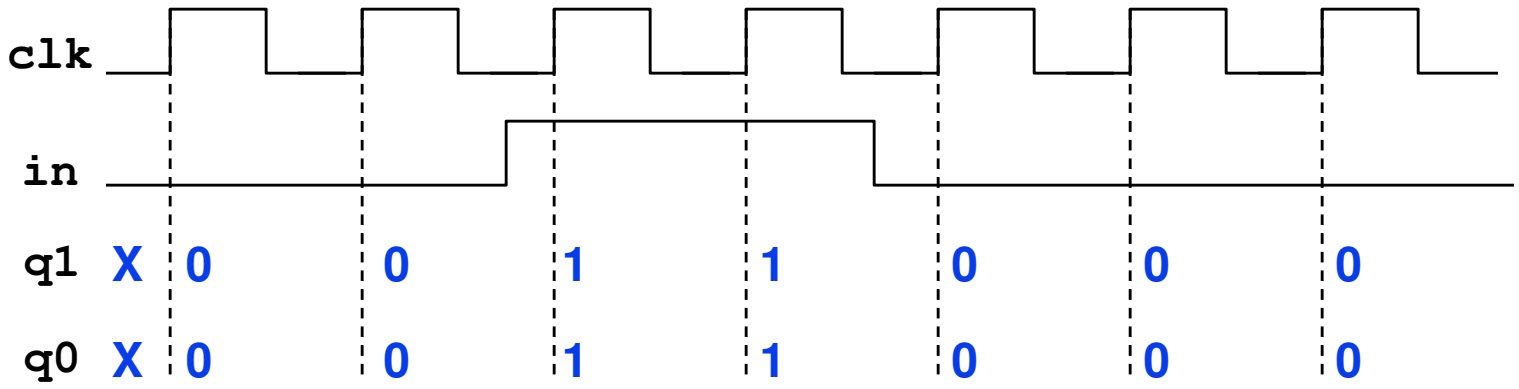
```
module fsm_mod(q1, q0, in, clk);
    output q1, q0;
    input clk, in;
    reg q1, q0;

    always @(posedge clk) begin
        q1 = in;
        q0 = in | q1;
    end
endmodule
```



Another example

Using blocking assignments:

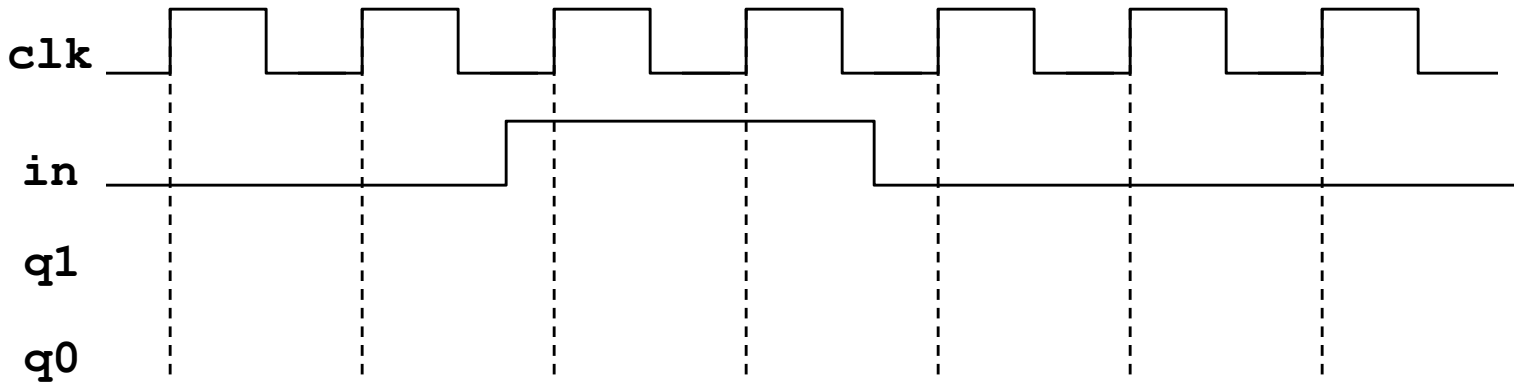
```
module fsm_mod(q1, q0, in, clk);
    output q1, q0;
    input clk, in;
    reg q1, q0;

    always @(posedge clk) begin
        q1 = in;
        q0 = in | q1;
    end
endmodule
```



Another example

Using non-blocking assignments:

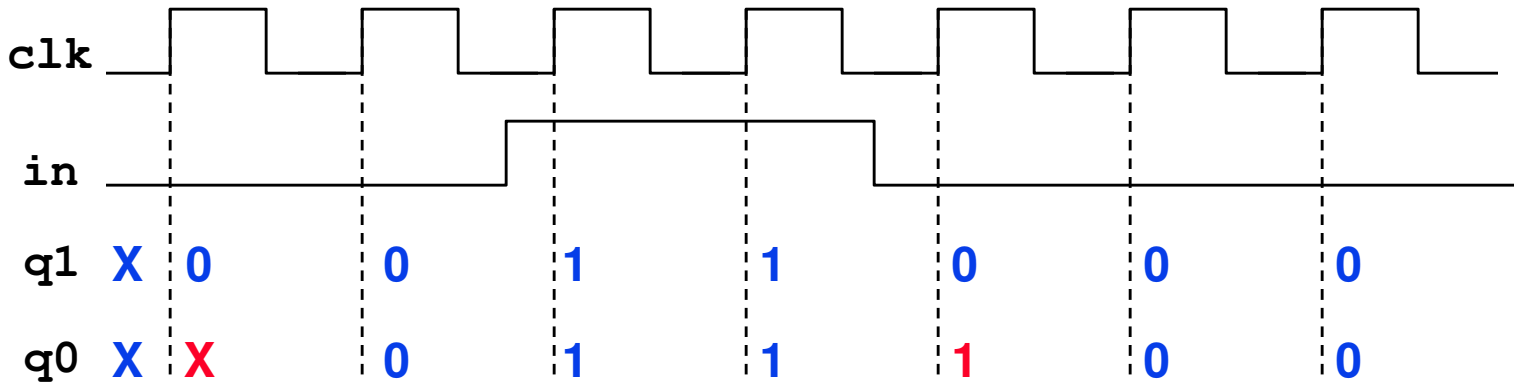
```
module fsm_mod(q1, q0, in, clk);
    output q1, q0;
    input clk, in;
    reg q1, q0;

    always @(posedge clk) begin
        q1 <= in;
        q0 <= in | q1;
    end
endmodule
```



Another example

Using non-blocking assignments:

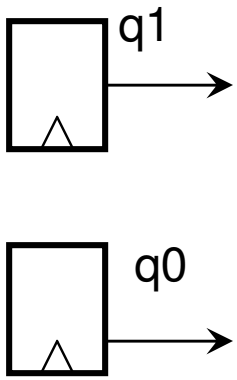
```
module fsm_mod(q1, q0, in, clk);
    output q1, q0;
    input clk, in;
    reg q1, q0;

    always @(posedge clk) begin
        q1 <= in;
        q0 <= in | q1;
    end
endmodule
```

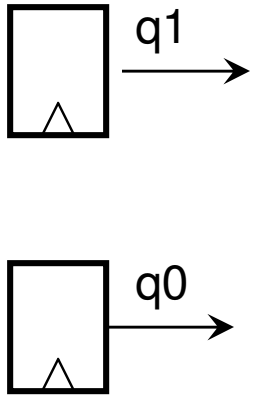


Think about the *meaning* of this behavior

```
module fsm_mod(q1, q0, in, clk);
    output q1, q0;
    input clk, in;
    reg q1, q0;

    always @(posedge clk) begin
        q1 = in;
        q0 = in | q1;
    end
endmodule
```

```
module fsm_mod(q1, q0, in, clk);
    output q1, q0;
    input clk, in;
    reg q1, q0;

    always @(posedge clk) begin
        q1 <= in;
        q0 <= in | q1;
    end
endmodule
```

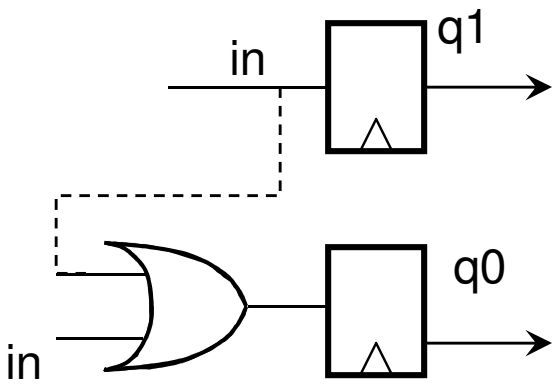


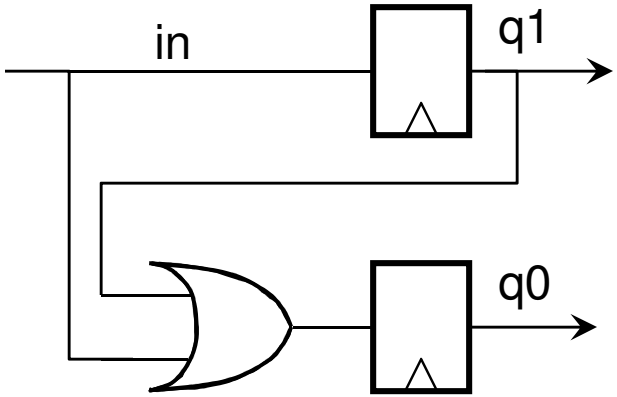


Think about the *meaning* of this behavior

```
module fsm_mod(q1, q0, in, clk);
    output q1, q0;
    input clk, in;
    reg q1, q0;

    always @(posedge clk) begin
        q1 = in;
        q0 = in | q1;
    end
endmodule
```

```
module fsm_mod(q1, q0, in, clk);
    output q1, q0;
    input clk, in;
    reg q1, q0;

    always @(posedge clk) begin
        q1 <= in;
        q0 <= in | q1;
    end
endmodule
```





Non-blocking assignments for registers

- Non-blocking assignments are used to model registers in a consistent fashion in the context of an always block
  - A single reg variable used to express register
  - Writing to the variable (using <=) means connecting to the register input
  - Reading from the variable means connecting to the register output
- More compact then gate-level model of a register
- More compact then the 'multiplexed datapath' method (with wire/reg combo)

Another example: A chain of 3 registers

```
module chain(q, i, clk);
  output q;
  input i, clk;

  reg q, q1, q2;

  always @(posedge clk) begin
     q <= q1;
     q1 <= q2;
     q2 <= i;
  end
endmodule
```

Another example: A chain of 3 registers

```
module chain(q, i, clk);
  output q;
  input i, clk;

  reg q, q1, q2;

  always @(posedge clk) begin
    q <= q1;
    q1 <= q2;
    q2 <= i;
  end
endmodule
```

all of these have the same behavior

```
q1 <= q2;
q  <= q1;
q2 <= i;
```

```
q2 <= i;
q1 <= q2;
q  <= q1;
```

Another example: A chain of 3 registers

- Mux-datapath method:

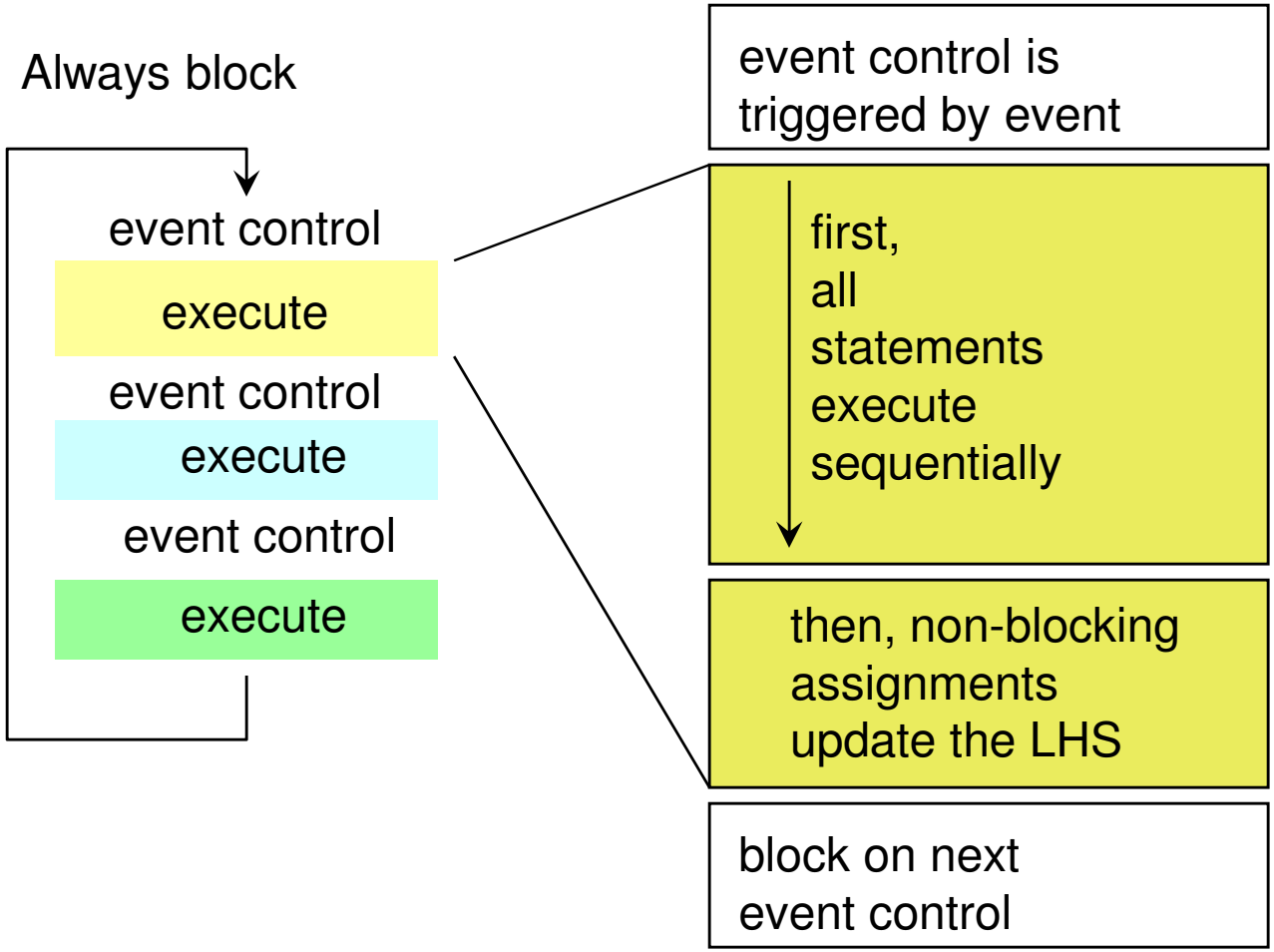
```
module chain(q, i, clk);
  output q;
  input i, clk;

  reg q, q1, q2;
  wire q_next, q1_next, q2_next;

  always @(posedge clk) begin
     q  = q_next;
     q1 = q1_next;
     q2 = q2_next;
  end

  assign q_next = q1;
  assign q1_next = q2;
  assign q2_next = i;

endmodule
```

Another example: A chain of 3 registers

- Structural method:

```
module flipflop(q, i, clk);
  output q;
  input i, clk;
  // implementation of flipflop ..
endmodule

module chain(q, i, clk);
  output q;
  input i, clk;

  wire q1, q2;

  flipflop F1(q, q1, clk);
  flipflop F2(q1, q2, clk);
  flipflop F3(q2, i, clk);

endmodule
```

Non-blocking assignments in a nutshell:



Delays and non-blocking assignments

```
initial
  begin
    a = 1;
    #5 b = a + 1;
  end
```

```
initial
  begin
    a <= 1;
    #5 b <= a + 1;
  end
```

Delays and non-blocking assignments

```
initial
  begin
    a = 1;
    #5 b = a + 1;
  end
```

t = 0: assign a with 1
t = 5: evaluate a + 1 and assign b

```
initial
  begin
    a <= 1;
    #5 b <= a + 1;
  end
```

t = 0: assign a with 1 at the end of timestep 0
t = 5: evaluate a + 1 and assign b at the end of timestep 5

Can avoid indeterminate results !

```
initial
  a = 0;
initial
  #5 a = a + 1;
initial
  #5 b = a + 1;
```

```
initial
  a = 0;
initial
  #5 a <= a + 1;
initial
  #5 b <= a + 1;
```

Can avoid indeterminate results !

```
initial
  a = 0;
initial
  #5 a = a + 1;
initial
  #5 b = a + 1;
```

t = 5: a is assigned with 6

t = 5: b is indeterminate, because a changes at t=5 using a blocking assignment

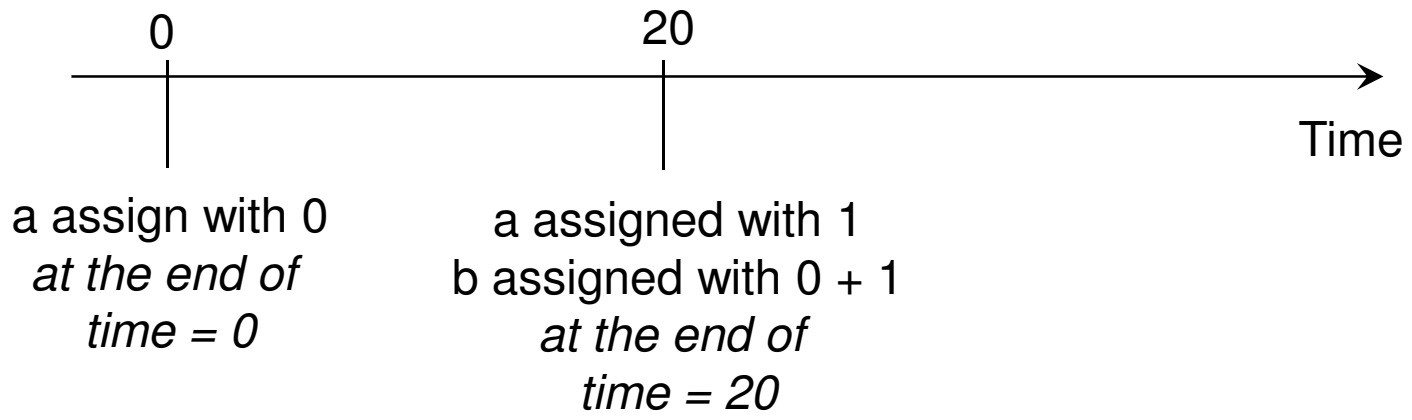
```
initial
  a = 0;
initial
  #5 a <= a + 1;
initial
  #5 b <= a + 1;
```

t = 5: a is assigned with 6 *at the end of t=5*

t = 5: b assigned with 6 *at the end of t = 5*

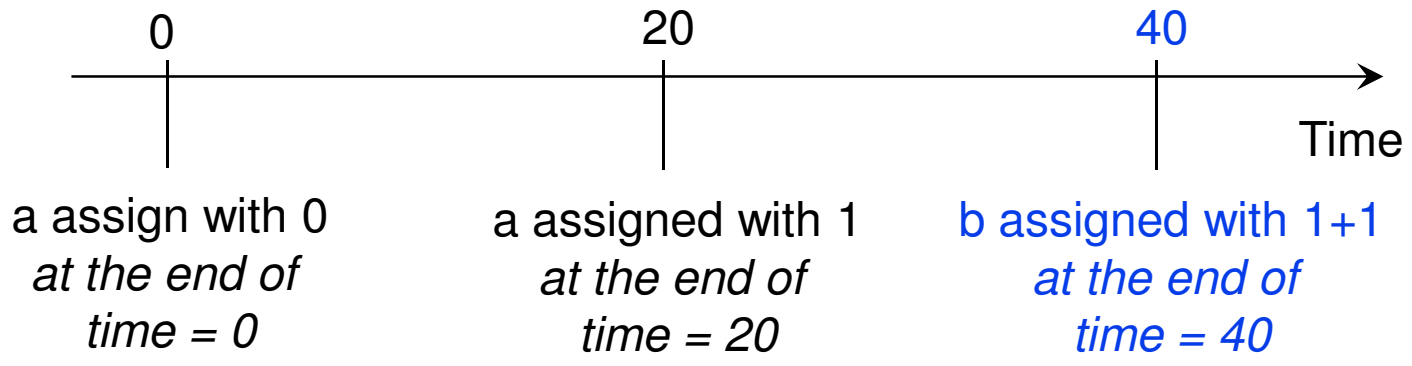
We can use intra-assignment delay, too

```
initial
  begin
  a <= 0;
  #20 a <= 1;
      b <= a + 1;
  end
```



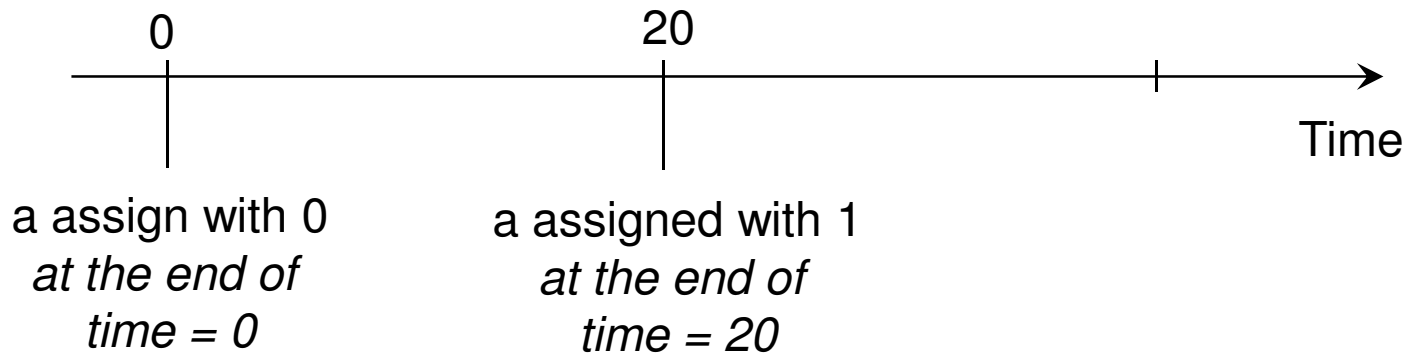
We can use intra-assignment delay, too

```
initial
  begin
  a <= 0;
  #20 a <= 1;
  #20 b <= a + 1;
  end
```



We can use intra-assignment delay, too

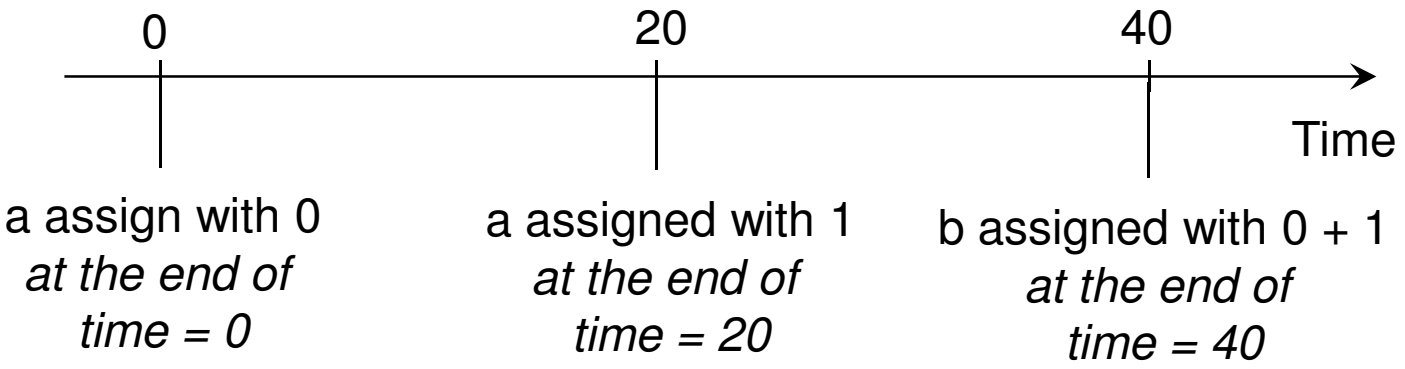
```
initial
  begin
  a <= 0;
  #20 a <= 1;
      b <= #20 a + 1;
  end
```



We can use intra-assignment delay, too

```
initial
  begin
  a <= 0;
  #20 a <= 1;
      b <= #20 a + 1;
  end
```

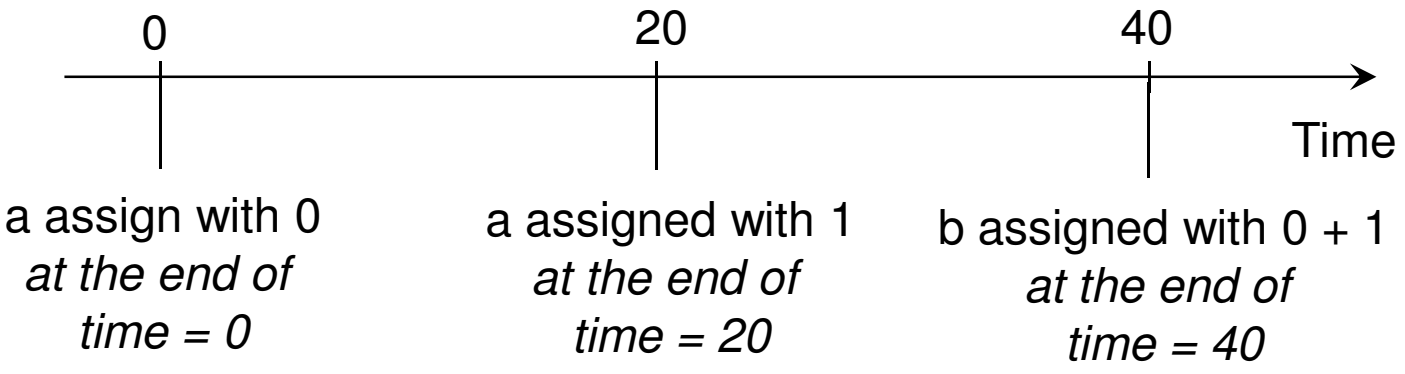
1) This is one 'sequential block'
2) All these statements will execute at time t = 20.
3) b will only be assigned at *the end of* time = 40



We can use intra-assignment delay, too

```
initial
  begin
  a <= 0;
  #20 a <= 1;
      b <= #20 a + 1;
      c <= #20 b + 1;
  end
```

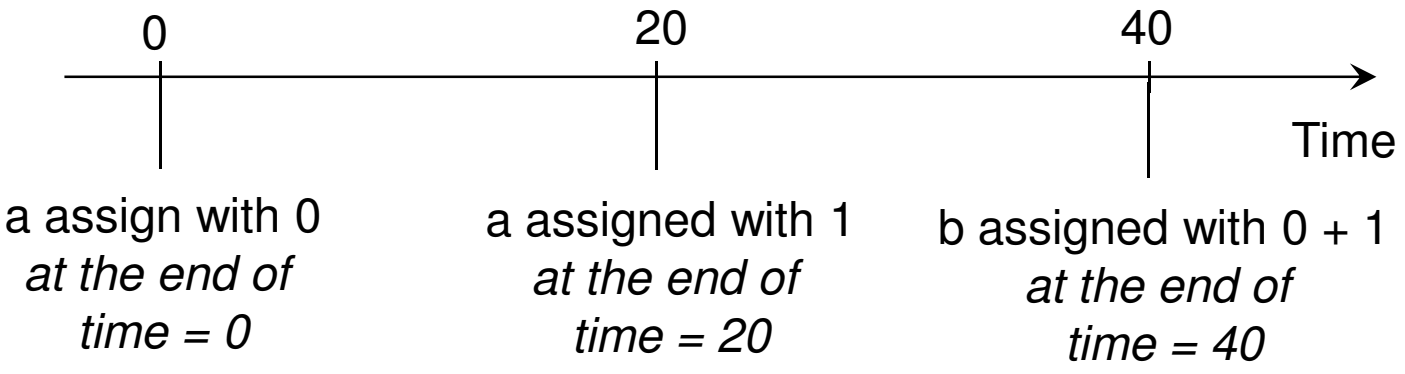
When is c assigned?



We can use intra-assignment delay, too

```
initial
  begin
  a <= 0;
  #20 a <= 1;
      b <= #20 a + 1;
      c <= #20 b + 1;
  end
```

When is c assigned?



c assigned with (b-value @20)
at the end of time = 40

Summary of non-blocking assignments

- Commonly used to model registers in always blocks
- Simulate the effect of multiple concurrent expressions
- We will use non-blocking assignments to design hardware modules, FSM, datapaths etc in a single always block.
- In practice DO NOT MIX non-blocking and blocking assignments in the same always block
- Further Reading:
  - Covered in part in Palnitkar Chapter 7
  - Covered in IEEE 1364 standard Chapter 9.2 (also, check out the examples)