Hardware Modeling [VU] (191.011)
– WS25 –
Behavioral Modeling

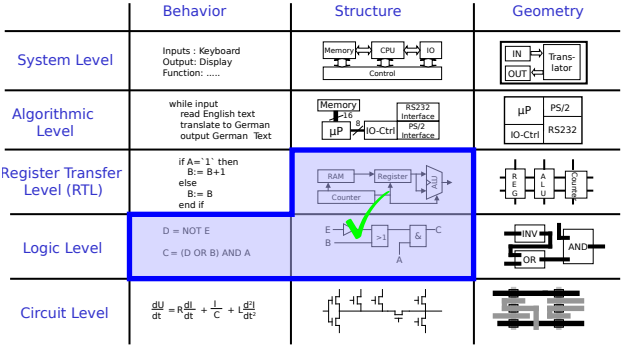
Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2025/26

Modified: 2025-12-16, 16:04 (f8a58e9)

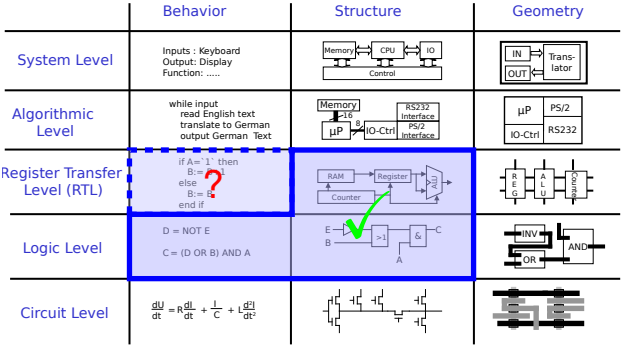
Introduction

- Concurrent assignments and structural modeling
  - Can model all **combinational** hardware
  - Hardly scales...



Introduction

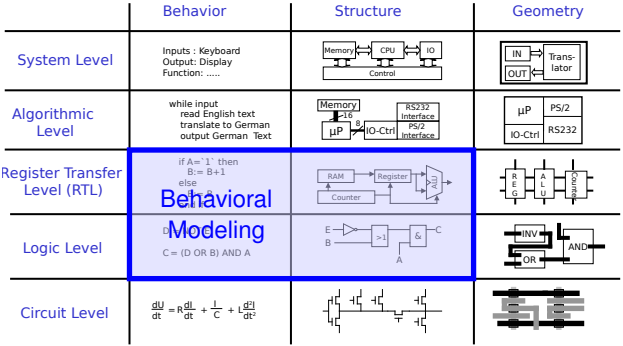
- Concurrent assignments and structural modeling
  - Can model all **combinational** hardware
  - Hardly scales...how?



Introduction

- Concurrent assignments and structural modeling
  - Can model all **combinational** hardware
  - Hardly scales...how?

⇒ *Behavioral Modeling*



Behavioral Modeling

- Revolves around processes
  - Must be synthesizable

Behavioral Modeling

- Revolves around processes
  - Must be synthesizable
  - “single-use entity and architecture”

Behavioral Modeling

- Revolves around processes
  - Must be synthesizable
  - “single-use entity and architecture”
  - Control flow statements and *variables*
  - *Sequential* description

Behavioral Modeling

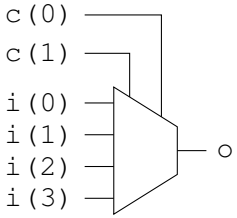
- Revolves around processes
  - Must be synthesizable
  - “single-use entity and architecture”
  - Control flow statements and *variables*
  - *Sequential* description
- **Complements** struct. modeling and concurrent assignments

Behavioral Modeling

- Revolves around processes
  - Must be synthesizable
  - “single-use entity and architecture”
  - Control flow statements and *variables*
  - *Sequential* description
- **Complements** struct. modeling and concurrent assignments
- Ubiquitous in synchronous designs

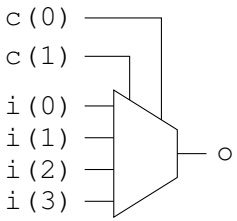
Example: Multiplexer


```
4  entity mux_41 is
5    port (
6      c : in  std_ulogic_vector(1 downto 0);
7      i : in  std_ulogic_vector(3 downto 0);
8      o : out std_ulogic
9    );
10 end entity;
```



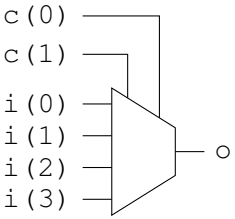
Example: Multiplexer

```
 4 entity mux_41 is
 5   port (
 6     c : in  std_ulogic_vector(1 downto 0);
 7     i : in  std_ulogic_vector(3 downto 0);
 8     o : out std_ulogic
 9   );
10 end entity;

12 architecture csa of mux_41 is
13 begin
14   o <= i(0) when not c(1) and not c(0) else
15        i(1) when not c(1) and     c(0) else
16        i(2) when     c(1) and not c(0) else
17        i(3) when     c(1) and     c(0);
18 end architecture;
```



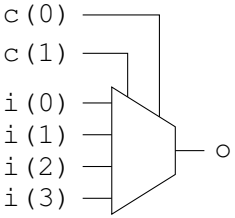
Example: Multiplexer


```
 4 entity mux_41 is
 5   port (
 6     c : in  std_ulogic_vector(1 downto 0);
 7     i : in  std_ulogic_vector(3 downto 0);
 8     o : out std_ulogic
 9   );
10 end entity;


12 architecture beh of mux_41 is
13 begin
14   process begin
15     case c is
16       when '0' & '0' => o <= i(0);
17       when '0' & '1' => o <= i(1);
18       when '1' & '0' => o <= i(2);
19       when '1' & '1' => o <= i(3);
20       when others    => null;
21     end case;
22     wait;
23   end process;
24 end architecture;
```



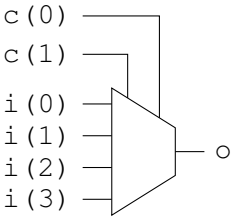
Example: Multiplexer


```
4  entity mux_41 is
5    port (
6      c : in  std_ulogic_vector(1 downto 0);
7      i : in  std_ulogic_vector(3 downto 0);
8      o : out std_ulogic
9    );
10 end entity;

12 architecture beh of mux_41 is
13 begin
14   process begin
15     case c is
16       when '0' & '0' => o <= i(0);
17       when '0' & '1' => o <= i(1);
18       when '1' & '0' => o <= i(2);
19       when '1' & '1' => o <= i(3);
20       when others    => null;
21     end case;
22     wait;
23   end process;
24 end architecture;
```



Example: Multiplexer

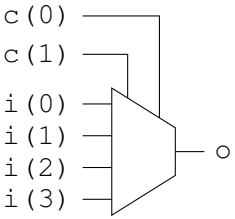

```
 4 entity mux_41 is
 5   port (
 6     c : in  std_ulogic_vector(1 downto 0);
 7     i : in  std_ulogic_vector(3 downto 0);
 8     o : out std_ulogic
 9   );
10 end entity;


12 architecture beh of mux_41 is
13 begin
14   process begin
15     case c is
16       when '0' & '0' => o <= i(0);
17       when '0' & '1' => o <= i(1);
18       when '1' & '0' => o <= i(2);
19       when '1' & '1' => o <= i(3);
20       when others    => null;
21     end case;
22     wait;
23   end process;
24 end architecture;
```



Example: Multiplexer


```
 4 entity mux_41 is
 5   port (
 6     c : in  std_ulogic_vector(1 downto 0);
 7     i : in  std_ulogic_vector(3 downto 0);
 8     o : out std_ulogic
 9   );
10 end entity;


12 architecture beh of mux_41 is
13 begin
14   process begin
15     case c is
16       when '0' & '0' => o <= i(0);
17       when '0' & '1' => o <= i(1);
18       when '1' & '0' => o <= i(2);
19       when '1' & '1' => o <= i(3);
20       when others    => null;
21     end case;
22     wait;
23   end process;
24 end architecture;
```

⇒ “Termination” of circuit?!



wait on Statement

- “Termination” of circuit not sensible
  - Stays active as long as powered

wait on Statement

- “Termination” of circuit not sensible
  - Stays active as long as powered
  - Instead: Model circuit as “sequential routine” for input changes

wait on Statement

- “Termination” of circuit not sensible
  - Stays active as long as powered
  - Instead: Model circuit as “sequential routine” for input changes

$\Rightarrow$ `wait on sensitivity_list`

wait on Statement

- "Termination" of circuit not sensible
  - Stays active as long as powered
  - Instead: Model circuit as "sequential routine" for input changes

$\Rightarrow$ `wait on sensitivity_list`

- Last element in **synthesizable** process (like `wait`)

wait on Statement

- "Termination" of circuit not sensible
  - Stays active as long as powered
  - Instead: Model circuit as "sequential routine" for input changes

$\Rightarrow$ `wait on sensitivity_list`

- Last element in **synthesizable** process (like `wait`)
- Process suspended when reaching `wait on` statement

wait on Statement

- "Termination" of circuit not sensible
  - Stays active as long as powered
  - Instead: Model circuit as "sequential routine" for input changes

$\Rightarrow$ `wait on sensitivity_list`

- Last element in **synthesizable** process (like `wait`)
- Process suspended when reaching `wait on` statement
- Starts from top when signal in list changes

wait on Statement

- "Termination" of circuit not sensible
  - Stays active as long as powered
  - Instead: Model circuit as "sequential routine" for input changes

⇒ `wait on sensitivity_list`

- Last element in **synthesizable** process (like `wait`)
- Process suspended when reaching `wait on` statement
- Starts from top when signal in list changes

```
1 process begin
2  case c is
3   when false & false => o <= i(0);
4   [...]
5  end case;
6  wait on c, i;
7 end process;
```

wait on Statement

- "Termination" of circuit not sensible
  - Stays active as long as powered
  - Instead: Model circuit as "sequential routine" for input changes

⇒ `wait on sensitivity_list`

- Last element in **synthesizable** process (like `wait`)
- Process suspended when reaching `wait on` statement
- Starts from top when signal in list changes

```
1 process begin
2  case c is
3   when false & false => o <= i(0);
4   [...]
5  end case;
6  wait on c, i;
7 end process;
```

"Process inputs"

Sensitivity List

wait on Statement

- "Termination" of circuit not sensible
  - Stays active as long as powered
  - Instead: Model circuit as "sequential routine" for input changes

⇒ `wait on sensitivity_list`

- Last element in **synthesizable** process (like `wait`)
- Process suspended when reaching `wait on` statement
- Starts from top when signal in list changes

```
1 process begin
2  case c is
3   when false & false => o <= i(0);
4   [...]
5  end case;
6  wait on c, i;
7 end process;
```

"Process inputs"

Sensitivity List

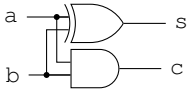
Example: Half-adder

Completeness of sensitivity list matters!

Example: Half-adder

Completeness of sensitivity list matters!

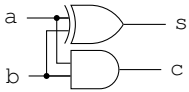
```
1 entity ha is
2  port (
3   a, b : in boolean;
4   s, c : out boolean
5  );
6 end entity;
```



Example: Half-adder

Completeness of sensitivity list matters!

```
1 entity ha is
2  port (
3   a, b : in boolean;
4   s, c : out boolean
5  );
6 end entity;
```

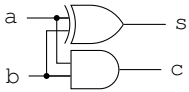



```
1 architecture arch1 of ha is
2 begin
3  process begin
4   s <= a xor b;
5   c <= a and b;
6   wait on a, b;
7  end process;
8 end architecture;
```

Example: Half-adder

Completeness of sensitivity list matters!

```
1 entity ha is
2  port (
3   a, b : in boolean;
4   s, c : out boolean
5  );
6 end entity;
```

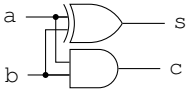



```
1 architecture arch1 of ha is
2 begin
3  process begin
4   s <= a xor b;
5   c <= a and b;
6   wait on a, b;
7  end process;
8 end architecture;
```

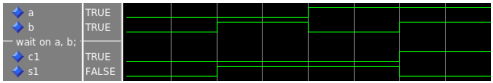
Example: Half-adder

Completeness of sensitivity list matters!

```
1 entity ha is
2  port (
3   a, b : in boolean;
4   s, c : out boolean
5  );
6 end entity;
```



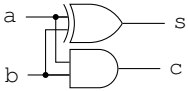

```
1 architecture arch1 of ha is
2 begin
3  process begin
4   s <= a xor b;
5   c <= a and b;
6   wait on a, b;
7  end process;
8 end architecture;
```



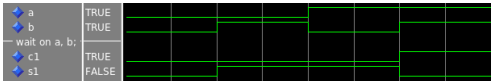
Example: Half-adder

Completeness of sensitivity list matters!

```
1 entity ha is
2  port (
3   a, b : in boolean;
4   s, c : out boolean
5  );
6 end entity;
```



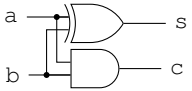

```
1 architecture arch2 of ha is
2 begin
3  process begin
4   s <= a xor b;
5   c <= a and b;
6   wait on a;
7  end process;
8 end architecture;
```



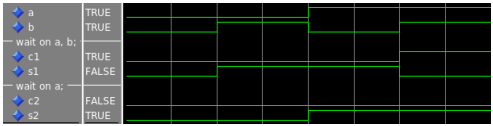
Example: Half-adder

Completeness of sensitivity list matters!

```
1 entity ha is
2  port (
3   a, b : in boolean;
4   s, c : out boolean
5  );
6 end entity;
```



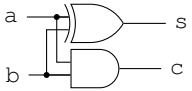

```
1 architecture arch2 of ha is
2 begin
3  process begin
4   s <= a xor b;
5   c <= a and b;
6   wait on a;
7  end process;
8 end architecture;
```



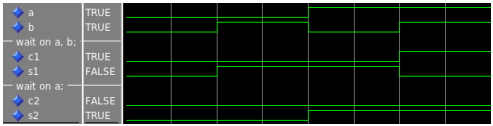
Example: Half-adder

Completeness of sensitivity list matters!

```
1 entity ha is
2  port (
3   a, b : in boolean;
4   s, c : out boolean
5  );
6 end entity;
```




```
1 architecture arch2 of ha is
2 begin
3  process begin
4   s <= a xor b;
5   c <= a and b;
6   wait on a;
7  end process;
8 end architecture;
```



Sensitivity List

Sensitivity List

- Use sensitivity list in process declaration
  - Implicit `wait on` $\Rightarrow$ semantically equivalent

Sensitivity List

- Use sensitivity list in process declaration
  - Implicit `wait on` ⇒ semantically equivalent

```
1 process (a, b)
2 begin
3   [...]
4 end process;
```

Sensitivity List

- Use sensitivity list in process declaration
  - Implicit `wait on` ⇒ semantically equivalent

```
1 process (a, b)
2 begin
3   [...]
4 end process;
```

```
1 process
2 begin
3   [...]
4   wait on a, b;
5 end process;
```

Sensitivity List

- Use sensitivity list in process declaration
  - Implicit `wait on` ⇒ semantically equivalent

```
1 process (a, b)
2 begin
3   [...]
4 end process;
```

```
1 process
2 begin
3   [...]
4   wait on a, b;
5 end process;
```

  - No wait statements in process (sim. only)

Sensitivity List

- Use sensitivity list in process declaration
  - Implicit `wait on` ⇒ semantically equivalent

```
1 process (a, b)
2 begin
3   [...]
4 end process;
```

⟺

```
1 process
2 begin
3   [...]
4   wait on a, b;
5 end process;
```

  - No wait statements in process (sim. only)

- Explicit sensitivity list for combinational logic can be

Sensitivity List

- Use sensitivity list in process declaration
  - Implicit `wait on` ⇒ semantically equivalent

```
1 process (a, b)
2 begin
3   [...]
4 end process;
```

⇔

```
1 process
2 begin
3   [...]
4   wait on a, b;
5 end process;
```

  - No wait statements in process (sim. only)

- Explicit sensitivity list for combinational logic can be
  - error-prone

Sensitivity List

- Use sensitivity list in process declaration
  - Implicit `wait on` ⇒ semantically equivalent

```
1 process (a, b)
2 begin
3   [...]
4 end process;
```

```
1 process
2 begin
3   [...]
4   wait on a, b;
5 end process;
```

  - No wait statements in process (sim. only)

- Explicit sensitivity list for combinational logic can be
  - error-prone
  - hard to maintain

Sensitivity List

- Use sensitivity list in process declaration
  - Implicit `wait on` ⇒ semantically equivalent

```
1 process (a, b)
2 begin
3   [...]
4 end process;
```

⟺

```
1 process
2 begin
3   [...]
4   wait on a, b;
5 end process;
```

  - No wait statements in process (sim. only)

- Explicit sensitivity list for combinational logic can be
  - error-prone
  - hard to maintain

⇒ `all` keyword for sensitivity lists

Sensitivity Lists Using *all*

- Sensitivity list is constructed at compile-time
  - Consider all statements inside the process
  - Apply rules to determine sensitive signals

Sensitivity Lists Using *all*

- Sensitivity list is constructed at compile-time
  - Consider all statements inside the process
  - Apply rules to determine sensitive signals

```
1 process (all)
2 begin
3   [...]
4 end process;
```

```
1 process (c, i)
2 begin
3   [...]
4 end process;
```

Sensitivity Lists Using *all*

- Sensitivity list is constructed at compile-time
  - Consider all statements inside the process
  - Apply rules to determine sensitive signals

```
1 process (all)
2 begin
3   [...]
4 end process;
```

```
1 process (c, i)
2 begin
3   [...]
4 end process;
```

- **Caveat:** Some circuits should not change for *any* input change

Sensitivity Lists Using *all*

- Sensitivity list is constructed at compile-time
  - Consider all statements inside the process
  - Apply rules to determine sensitive signals

```
1 process (all)
2 begin
3   [...]
4 end process;
```

```
1 process (c, i)
2 begin
3   [...]
4 end process;
```

- **Caveat:** Some circuits should not change for *any* input change
  - Synchronous logic **only** sensitive to clock and reset
  - More on that in chapter II

Sensitivity Lists Using *all*

- Sensitivity list is constructed at compile-time
  - Consider all statements inside the process
  - Apply rules to determine sensitive signals

```
1 process (all)
2 begin
3   [...]
4 end process;
```

```
1 process (c, i)
2 begin
3   [...]
4 end process;
```

- **Caveat:** Some circuits should not change for *any* input change
  - Synchronous logic **only** sensitive to clock and reset
  - More on that in chapter II
- Rule of thumb: Use `all` for comb. processes

Process Simulation

- Process is a *sequential* description
  - Hardware is highly concurrent

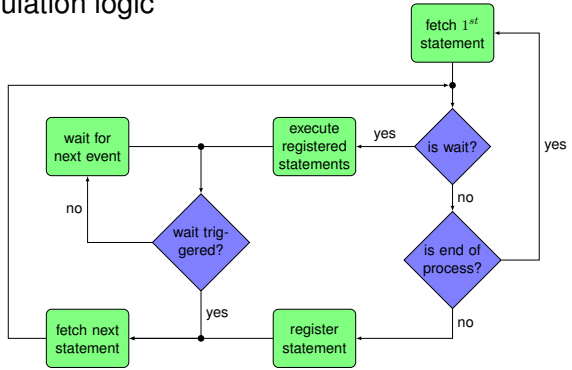
Process Simulation

- Process is a *sequential* description
  - Hardware is highly concurrent
  - What does the simulator do?

Process Simulation

- Process is a *sequential* description
  - Hardware is highly concurrent
  - What does the simulator do?

$\Rightarrow$ Process simulation logic



Example: Process Simulation

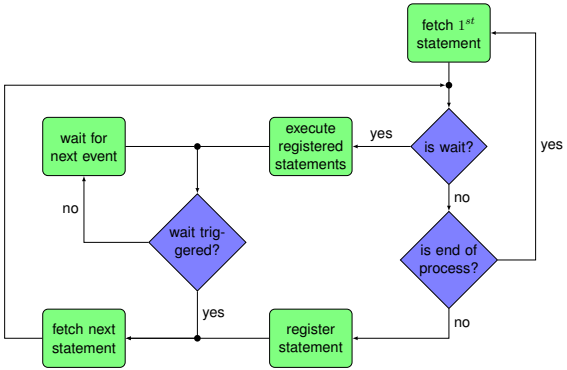

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```

Example: Process Simulation


```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```

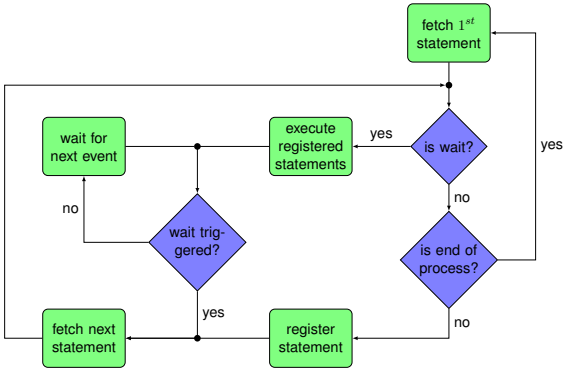
Example: Process Simulation


```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



Example: Process Simulation


```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```

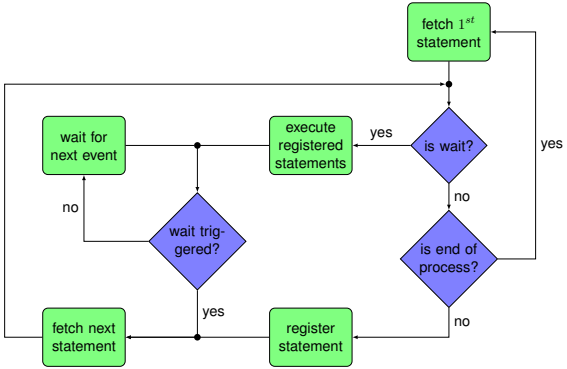


a: false
b: false
c: false

Example: Process Simulation

Input `a` changes

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```

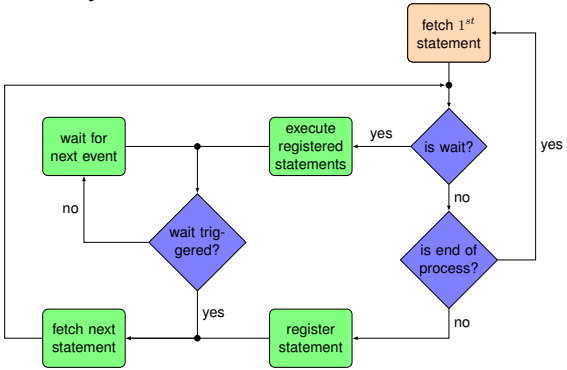


a: `false` → `true`
b: `false`
c: `false`

Example: Process Simulation

Fetch first statement of the process body

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```

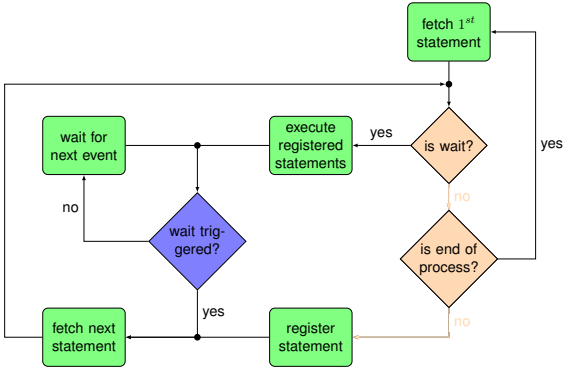


a: `true`
b: `false`
c: `false`

Example: Process Simulation

Neither `wait` nor `end process`

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```

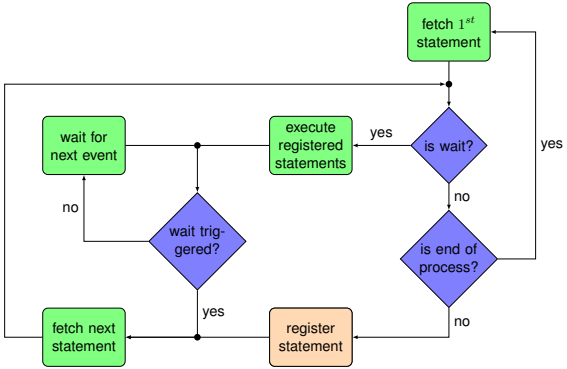


a: `true`
b: `false`
c: `false`

Example: Process Simulation

Register statement for **future** execution

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



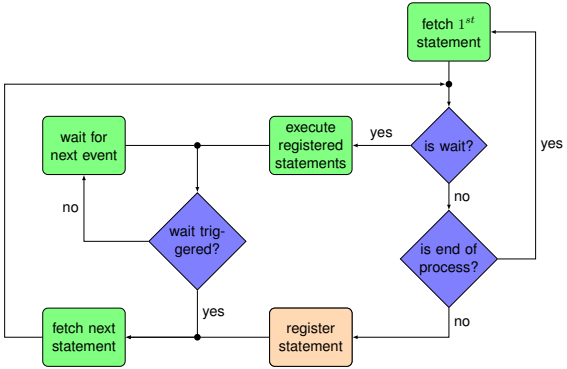
a: `true`
b: `false`
c: `false`

Registered: `b <= true;`

Example: Process Simulation

Register statement for **future** execution

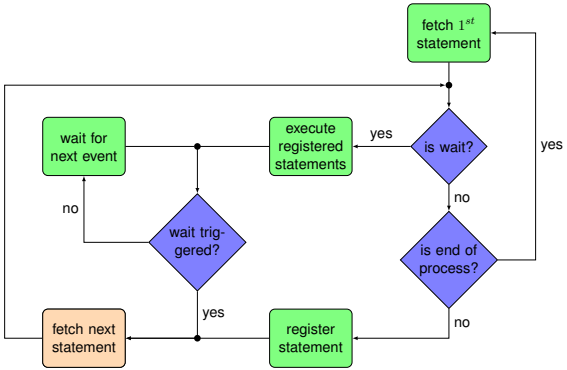
```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



Example: Process Simulation

Fetch the next statement

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



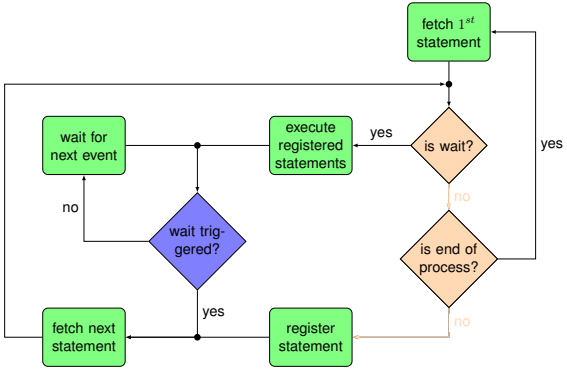
a: `true`
b: `false`
c: `false`

Registered: `b <= true;`

Example: Process Simulation

Continue until `wait` nor `end process` encountered

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



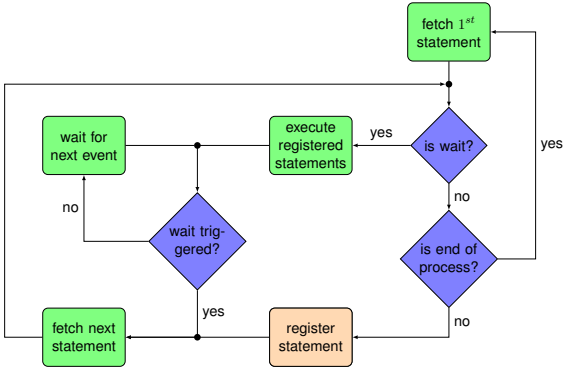
a: `true`
b: `false`
c: `false`

Registered: `b <= true;`

Example: Process Simulation

Continue until `wait` nor `end process` encountered

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



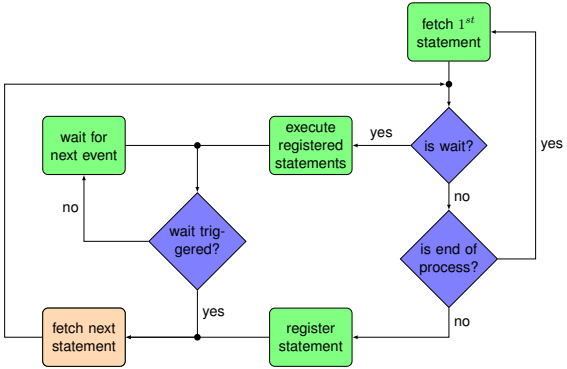
a: `true`
b: `false`
c: `false`

Registered: `b <= true;`
`c <= false;`

Example: Process Simulation

Continue until `wait` nor `end process` encountered

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



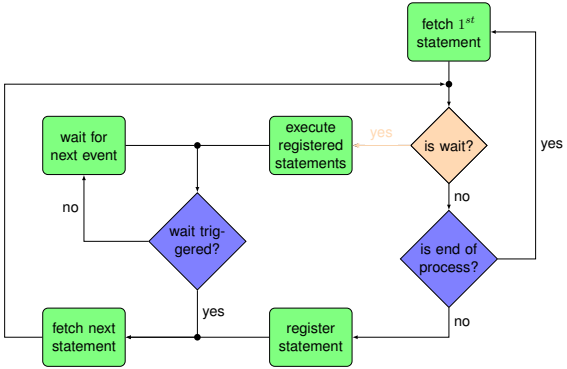
a: `true`
b: `false`
c: `false`

Registered: `b <= true;`
`c <= false;`

Example: Process Simulation

`wait` statement encountered

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



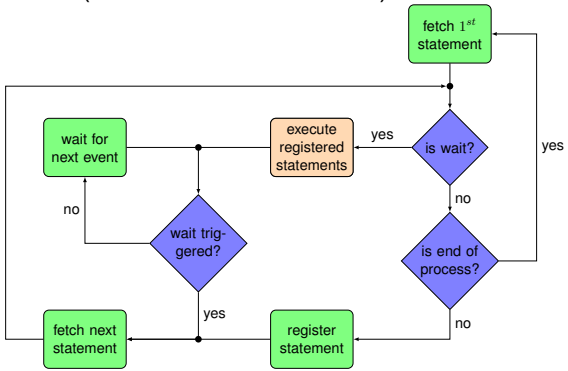
a: `true`
b: `false`
c: `false`

Registered: `b <= true;`
`c <= false;`

Example: Process Simulation

Now execute **all** registered statements (in zero simulation time)

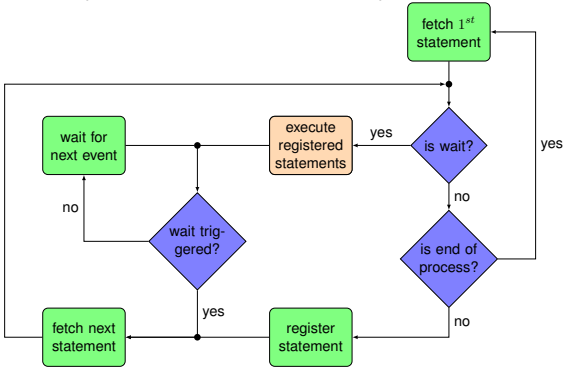
```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



Example: Process Simulation

Now execute **all** registered statements (in zero simulation time)

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



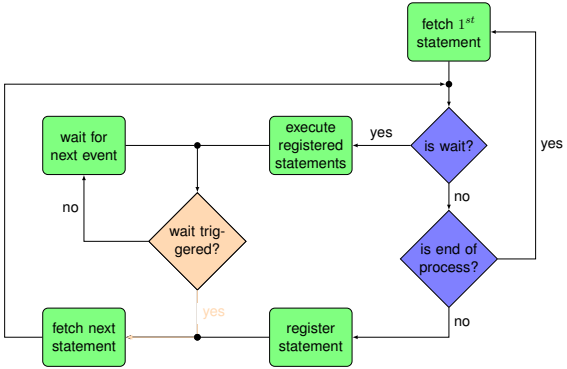
a: `true`
b: `true`
c: `false` → `false`

Registered: `c <= false;`

Example: Process Simulation

Change of `b` triggered `wait on a,b`

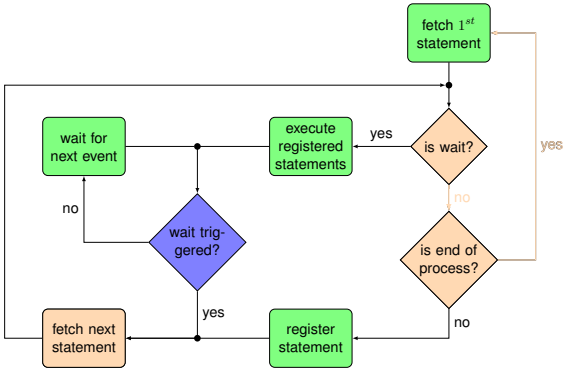
```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



Example: Process Simulation

Fetch next statement

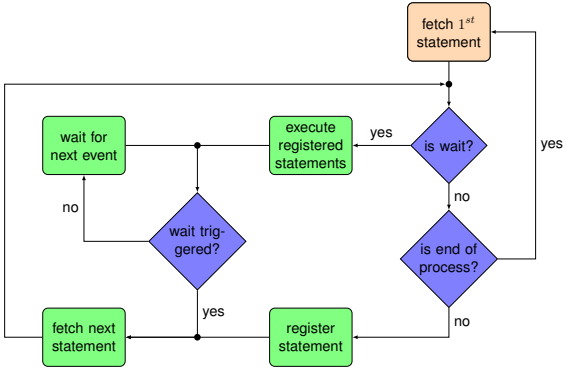
```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



Example: Process Simulation

End of process ⇒ Repeat

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



a: `true`
b: `true`
c: `false`

Registered:

Observations: Process Simulation

Observation I

Statements are not executed immediately, but rather gathered until a wait is encountered. Then they are all executed without consuming simulation time. This mimics a concurrent execution.

Observations: Process Simulation

Observation I

Statements are not executed immediately, but rather gathered until a wait is encountered. Then they are all executed without consuming simulation time. This mimics a concurrent execution.

Observation II

Without `b` in the sensitivity list in the example, `c` would end remaining `false` until the next change of `a` $\Rightarrow$ proper sensitivity list paramount.

Observations: Process Simulation

Observation I

Statements are not executed immediately, but rather gathered until a wait is encountered. Then they are all executed without consuming simulation time. This mimics a concurrent execution.

Observation II

Without `b` in the sensitivity list in the example, `c` would end remaining `false` until the next change of `a` $\Rightarrow$ proper sensitivity list paramount.

Observation III

A process without (implicit) `wait` statement loops endlessly.

Variables

- Signal assignments executed at `wait`

Variables

- Signal assignments executed at `wait`

$\Rightarrow$ Variables

Variables

- Signal assignments executed at `wait`

$\Rightarrow$ Variables

  - In process declarative part

```
variable x : integer := 10;
```

Variables

- Signal assignments executed at `wait`

⇒ Variables

- In process declarative part

```
variable x : integer := 10;
```

- Assignments (`:=`) have *immediate* effect

Variables

- Signal assignments executed at `wait`

⇒ Variables

- In process declarative part

```
variable x : integer := 10;
```

- Assignments (`:=`) have *immediate* effect
- **Not** in sensitivity list

Variables

- Signal assignments executed at `wait`

⇒ Variables

- In process declarative part

```
variable x : integer := 10;
```

- Assignments (`:=`) have *immediate* effect
- **Not** in sensitivity list
- Name and reuse intermediate expressions

Variables

- Signal assignments executed at `wait`

⇒ Variables

  - In process declarative part
    ```
    variable x : integer := 10;
    ```
  - Assignments (`:=`) have *immediate* effect
  - **Not** in sensitivity list
  - Name and reuse intermediate expressions

- Example

```
1  entity wide_and is
2  port(
3   i : in std_ulogic_vector;
4   o : out std_ulogic
5  );
```

```
1  process (all) is
2   variable result : std_ulogic := '1';
3  begin
4   for x in i'range loop
5    result := result and i(x);
6   end loop;
7   o <= result;
8  end process;
```

Variables

- Signal assignments executed at `wait`

⇒ Variables
  - In process declarative part
    `variable x : integer := 10;`
  - Assignments (`:=`) have *immediate* effect
  - **Not** in sensitivity list
  - Name and reuse intermediate expressions

- Example

```
1  entity wide_and is
2  port(
3   i : in std_ulogic_vector;
4   o : out std_ulogic
5  );
```

```
1  process (all) is
2   variable result : std_ulogic := '1';
3  begin
4   for x in i'range loop
5    result := result and i(x);
6   end loop;
7   o <= result;
8  end process;
```

Example: Variables vs Signals

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Example: Variables vs Signals

```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	`x='1'`
2	

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	`x='1', y='1'`
2	

Example: Variables vs Signals

```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	x='1', y='1' z='1'
2	

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	x='1', y='0' z='1'
2	

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	`x='1', y='0'` `z='1', w='1'`
2	

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	`x='1', y='0'` `z='1', w='1'`
2	`w='0', x='1'` `y='0', z='0'`

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	x='1', y='0' z='1', w='1'
2	w='0', x='1' y='0', z='0'

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	`x='1', y='0'` `z='1', w='1'`
2	`w='0', x='1'` `y='0', z='0'`

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	`x='1', y='0'` `z='1', w='1'`
2	`w='0', x='1'` `y='0', z='0'`

Iteration #	signal values at end
1	
2	

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	x='1', y='0' z='1', w='1'
2	w='0', x='1' y='0', z='0'

Iteration #	signal values at end
1	x='1', y='1'
2	

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	x='1', y='0' z='1', w='1'
2	w='0', x='1' y='0', z='0'

Iteration #	signal values at end
1	x='1', y='1' z='1'
2	

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	x='1', y='0' z='1', w='1'
2	w='0', x='1' y='0', z='0'

Iteration #	signal values at end
1	x='1', y='0' z='1'
2	

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	x='1', y='0' z='1', w='1'
2	w='0', x='1' y='0', z='0'

Iteration #	signal values at end
1	x='1', y='0' z='1', w='0'
2	

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	x='1', y='0' z='1', w='1'
2	w='0', x='1' y='0', z='0'

Iteration #	signal values at end
1	x='1', y='0' z='1', w='0'
2	-

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Not the same circuit!

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

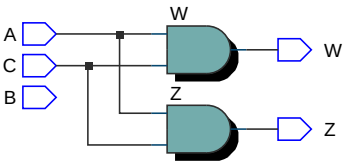
Iteration #	signal values at end
1	x='1', y='0' z='1', w='1'
2	w='0', x='1' y='0', z='0'

Iteration #	signal values at end
1	x='1', y='0' z='1', w='0'
2	-

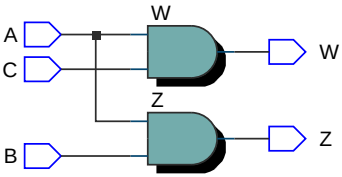
Variables vs Signals (Cont'd)

⇒ Use of `variable` and `signal` not equivalent!

Signals



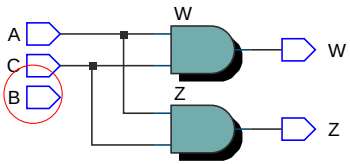
Variables



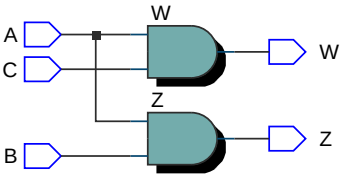
Variables vs Signals (Cont'd)

⇒ Use of `variable` and `signal` not equivalent!

Signals



Variables



Remarks

Remarks

- `process` declaration (simplified)

```
1 [label] : process designator [(sensitivity_list)] [is]
2   [declarative_part]
3 begin
4   [statement part]  -- process body
5 end process;
```

Remarks

- `process` declaration (simplified)

```
1 [label] : process designator [(sensitivity_list)] [is]
2   [declarative_part]
3 begin
4   [statement part]  -- process body
5 end process;
```

- Every concurrent signal has an equivalent process

Remarks

- `process` declaration (simplified)

```
1 [label] : process designator [(sensitivity_list)] [is]
2   [declarative_part]
3 begin
4   [statement part]  -- process body
5 end process;
```

- Every concurrent signal has an equivalent process
  - Examples: MUX41, halfadder, wide AND-gate

Remarks

- `process` declaration (simplified)

```
1 [label] : process designator [(sensitivity_list)] [is]
2   [declarative_part]
3 begin
4   [statement part]  -- process body
5 end process;
```

- Every concurrent signal has an equivalent process
  - Examples: MUX41, halfadder, wide AND-gate
  - The other direction is not true (e.g., sync. logic)

Remarks

- `process` declaration (simplified)

```
1 [label] : process designator [(sensitivity_list)] [is]
2   [declarative_part]
3 begin
4   [statement part]  -- process body
5 end process;
```

- Every concurrent signal has an equivalent process
  - Examples: MUX41, halfadder, wide AND-gate
  - The other direction is not true (e.g., sync. logic)
- Arbitrary many processes possible

Remarks

- `process` declaration (simplified)

```
1 [label] : process designator [(sensitivity_list)] [is]
2   [declarative_part]
3 begin
4   [statement part]  -- process body
5 end process;
```

- Every concurrent signal has an equivalent process
  - Examples: MUX41, halfadder, wide AND-gate
  - The other direction is not true (e.g., sync. logic)
- Arbitrary many processes possible
  - Executed concurrently

Remarks

- `process` declaration (simplified)

```
1 [label] : process designator [(sensitivity_list)] [is]
2   [declarative_part]
3 begin
4   [statement part]  -- process body
5 end process;
```

- Every concurrent signal has an equivalent process
  - Examples: MUX41, halfadder, wide AND-gate
  - The other direction is not true (e.g., sync. logic)
- Arbitrary many processes possible
  - Executed concurrently
  - Order of actual execution *undefined* $\Rightarrow$ do not rely on it

Lecture Complete!