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

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

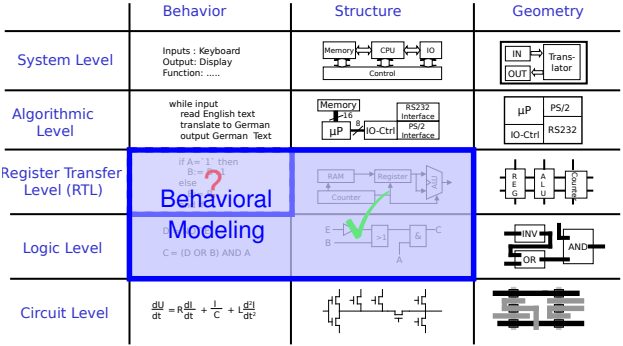
WS 2025/26

Modified: 2025-12-16, 16:07 (f8a58e9)

Introduction

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

⇒ *Behavioral Modeling*



Behavioral Modeling

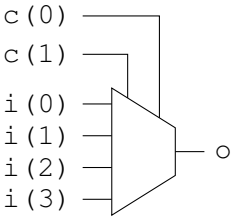
- Revolves around processes
  - Must be synthesizable
  - “single-use entity and architecture”
  - Control flow statements and *variables*
  - *Sequential* description
- **Complements** struct. modeling and concurrent assignments
- Ubiquitous in synchronous designs

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;

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;

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);
```



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

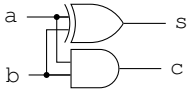
```
1 process begin
2  case c is
3   when false & false => o <= i(0);
4   [...]
5  end case;
6  wait on c, i;
7 end process;
```

"Process inputs"

Sensitivity List

Example: Half-adder

Completeness of sensitivity list matters!




```
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;
```

```
1 architecture arch2 of ha is
2 begin
```

Sensitivity List

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

  - No wait statements in process (sim. only)

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

$\Rightarrow$ `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

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

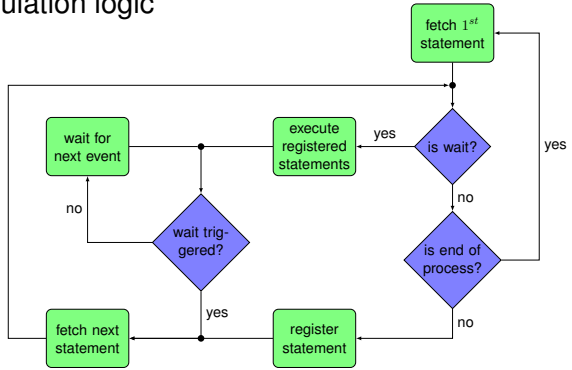
```
1 process (c, i)
2 begin
3   [...]
4 end process;
```

- **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
  - What does the simulator do?

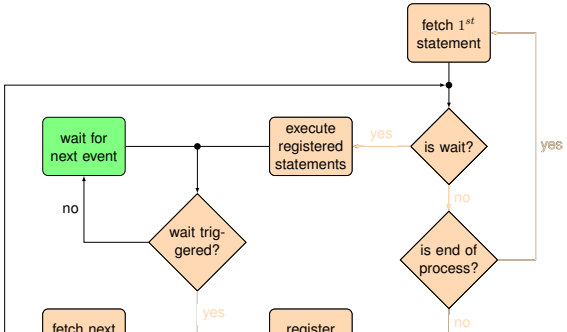
$\Rightarrow$ Process simulation logic



Example: Process Simulation

Input `a` changes Fetch first statement of the process body Neither `wait` nor `end process` *Register* statement for **future** execution Fetch the next statement Continue until `wait` nor `end process` encountered `wait` statement encountered Now execute **all** registered statements (in zero simulation time) Change of `b` triggered `wait on a,b` Fetch next statement 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;
```



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

  - 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

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

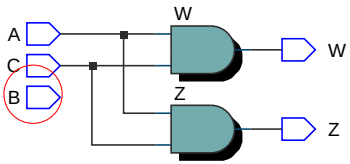
Iteration #	signal values at end
1	`x='1', y='1', y='0'` `z='1', w='1'`
2	`w='0', x='1'` `y='0', z='0'`

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

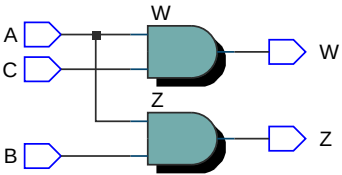
Variables vs Signals (Cont'd)

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

Signals



Variables



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!