Dataflow modelling

Lecture 4

Dataflow modelling

- Specifies the functioning of a circuit without explicitly refer to its structure
- Functioning is described by the flow of information through the circuit, using mostly concurrent signal assignment statements and BLOCK statements
- Statements:
  - Concurrent signal assignments
  - Concurrent ASSERT statements
  - Concurrent procedure calls
  - BLOCK

Concurrent signal assignments

•Concurrent signal assignments are triggered by events (they are event triggered), i.e. they execute when an event takes place for the expression that is assigned to the signal.

•Sequential signal assignment execute when the statement is executed within the process => they are not triggered by the events from the expression that is assigned to the signal.

Example:

```
ARCHITECTURE sequential_assignment OF example IS

SIGNAL a, b, z : some_type;

BEGIN

PROCESS(b)

BEGIN

        a<=b;

        z<=a;

END PROCESS;

END ARCHITECTURE;
```

Concurrent signal assignments

```
ARCHITECURE concurrent_assignment OF example IS
SIGNAL a, b, z: some_type;
BEGIN
        a<=b;
        z<=a;
END ARCHITECTURE;
```

First architecture (sequential_assignment):

- When there is an event on b, the process will resume
- Signal a is scheduled to take the new value of b;
- Signal z is scheduled to take the value of a (which is the previous value of b).

Concurrent signal assignments

Second architecture (concurrent_assignment):

•If an event takes place on b at the moment Tc, then the statement a<=b is executed

•Hence a is scheduled a new value at the moment Tc+Δ;

•At Tc+ Δ an event takes place on a => the statement z<=a is executed;

•At Tc+2 Δ signal z will receive the value of a, which is also the new value of b.

Second architecture is equivalent with:

```
PROCESS(b)
BEGIN
        a<=b;
END PROCESS;
PROCESS(a)
BEGIN
        z<=a;
END PROCESS;
```

Concurrent signal assignments

A concurrent statement like:

s<=x+y+z; -- x, y, z, s signals

Is equivalent with:

```
PROCESS
BEGIN
        s<=x+y+z;
        WAIT ON x,y,z;
END PROCESS;
```

Conditional signal assignment statement

concurrent_signal_assignment::=

[POSTPONED] conditional_signal_assignment |

[POSTPONED] selected_signal_assignment;

conditional_signal_assignment::=

target<= [TRANSPORT|[REJECT time_expression] INERTIAL] conditional_waveforms;

conditional_waveforms::=

{waveform WHEN condition ELSE}

[waveform WHEN condition]

•Conditional signal assignment is executed every time when an event takes place on any one of the signals from the waveforms or from conditions.

•Conditions are evaluated one by one, in the order in which they are written.

•For the first condition found true, the target signal is assigned the waveform from that condition.

Conditional signal assignment statement: the equivalent process

Example. In an architecture we have the statement:

```
ARCHITECTURE ex OF ex IS
BEGIN
s<=x+y-2 AFTER 10ns WHEN i0='0' AND i1='0' ELSE
   x-y+2 AFTER 10ns WHEN i0='1' AND i1='0' ELSE
   x+y AFTER 7ns WHEN i0='0' AND i1='1' ELSE
   x-y AFTER 5ns;
END ARCHITECUTE;
```

Conditional signal assignment statement: the equivalent process

The statement is equivalent with the following process:

```
PROCESS
BEGIN
       IF i0='0' AND i1='0' THEN
              s<= x+y-2 AFTER 10ns ;
       ELSIF i0='1' AND i1='0' THEN
              s<= x-y+2 AFTER 10ns;
       ELSIF i0='0' AND i1='0' THEN
              s<=x+y AFTER 7ns;
       ELSE
              s<=x-y AFTER 5ns;
       END IF;
       WAIT ON i0,i1,x,y;
END PROCESS;
```

Selected signal assignment

selected_signal_assignment::=

WITH expression SELECT

target <= [TRANSPORT|[REJECT time_expression]INERTIAL] selected_waveforms;

Selected_waveforms::=

{waveform WHEN choices,}

waveform WHEN choices

choices::= {choice,}

choice::=simple_expression | discrete_range | OTHERS

•When an event takes place on any one of the signals from *expression* or from waveforms, the statement is executed, which means:

- •expression is evaluated
- •The target signal is assigned the waveform from the branch which contains the value of the expression.

•Choices must be different and must cover all the values of the expression.

Selected signal assignment: the equivalent process

The statement:

```
WITH a+b SELECT
        s<= x+y WHEN 0|1|2,
            x-y AFTER 5ns WHEN 3 TO 10,
            UNAFFECTED WHEN OTHERS;
```

Is equivalent with:

```
PROCESS
BEGIN
        CASE a+b IS
                WHEN 0|1|2 => s<=x+y;
                WHEN 3 TO 10 => s<= x-y AFTER 5ns;
                WHEN OHTERS=> NULL;
        END CASE;
        WAIT ON a,b,x,y;
END PROCESS;
```

The UNAFFECTED value

- There are situations when we want that the value of a signal will remain unchanged
- This can be done using UNAFFECTED
- Using UNAFECTED there will be no changes on the signal driver
  - If we write x <= x; there will be changes !
- Sequential equivalent for UNAFFECTEFD is the NULL statement;

Concurrent ASSERT statement

•Has the same form like sequential ASSERT statement.

•It is executed every time when there is an event on any of the signals from conditions.

•Example:

```
ASSERT s1/=s2 -- /= means different
REPORT “error, s1=s2” SEVERITY ERROR;
```

•The equivalent process:

```
PROCESS
BEGIN
        ASSERT s1/=s2 REPORT “eror, s1=s2” SEVERITY ERROR;
        WAIT ON s1,s2;
END PROCESS;
```

Concurrent procedure calls

- In VHDL there exist procedures and functions.
  - A FUNCTION returns one value
  - A PROCEDURE performs some computations
- Function calls appear in expressions, hence they cannot be separate statements.
- Procedure calls can be sequential or concurrent statements.
- A concurrent procedure is equivalent with a process that contains
  - The sequential procedure call
  - And a WAIT statement containing in the sensitivity clause all signals that are parameters of mode IN or INOUT of that procedure
- A concurrent procedure call does not allow the procedure to have parameters of class VARIABLE.

Concurrent procedure calls

```
-- procedure declaration

PROCEDURE compute(SIGNAL a,b: IN INTEGER; SIGNAL res: OUT
INTEGER; SIGNAL x: INOUT INTEGER)

BEGIN

…

END PROCEDURE;

--a concurrent procedure call (inside an architecture):

ARCHITECTURE call_proc OF ex IS

      SIGNAL siga, sigb,sigres, sigx: INTEGER;

BEGIN

      …

      compute(siga, sigb, sigres, sigx);

END ARCHITECTURE;
```

Concurrent procedure calls

-- the equivalent process:

```
ARCHITECTURE call_proc OF ex IS
        SIGNAL siga, sigb, sigres, sigx : INTEGER;
PROCESS
BEGIN
        compute(siga, sigb, sigres, sigx);
        WAIT ON siga, sigb, sigx;
END PROCESS;
END ARCHITECTUTRE;
```

BLOCK statement [Bha95]

- It is a concurrent statement
- It can have three utilizations:
  1. to deactivate the driver of a signal (in guarded blocks)
  2. To limit the visibility of some declarations (including signal declarations)
  3. To partition a project in order to increase the clarity of the program:
     - E.g. we describe a microprocessor and we have the registers block, the ALU block, the control unit block, etc.; in the registers block we can have a block for each register

BLOCK

BNF description:

Block_statement::=

Block_label: BLOCK [(guard_expression)][IS]

[Block_header]

[Block_declarative_part]

BEGIN

concurrent_statements

END BLOCK [Block_label];

BLOCK

- Block_label –the label is mandatory.
- Block_header – block header
  - If it exists, it describe the block interface with the outside word, i.e. ports and generics (like for components)
  - since blocks cannot be instantiated, the header is less useful like at components (used during the elaboration phase !!)
- Block_declarative_part
  - Declarative part is optional
  - May contain declarations of:
    - Types, subtypes, constants, signals
  - MAY NOT contain: variable declarations
  - Anything declared in the declarative part of the block is visible only inside the block
- The body of the block: between BEGIN and END BLOCK:
  - May contain any number of concurrent statements, including none
  - May conain other BLOCK statements, on any number of levels.
- The label from the end of the block is optional but, if appears, it must be the same as the label from the beginning of the block.
- The symbol ; at the end is mandatory

Blocks and guards

Guard_expression: -

•If the block contains a guard expression, then a signal named GUARD, of type BOOLEAN, will be implicitly declared.

•The value of the signal GUARD is given by the guard expression, which hasto be of type BOOLEAN.

• In the guard expression we can have signals, but not variables.

•The value of the signal GUARD is updated when the guard value changes.

•Signal assignment from inside the block can use the GUARD signal in order to activate / de-activate their drivers.

•Example:

```
b1: BLOCK (strobe='1')
BEGIN
      z <= GUARDED NOT a;
END BLOCK b1;
```

Blocks and guards

- The keyword GUARDED can be optionally used at concurrent signal assignments inside the block
- In the previous example
  - The guard expression is (strobe=‘1’)
  - GUARD <= (strobe=‘1’);
  - When strobe is modified strobe, the GUARD signal will be modified as well
  - When the GUARD signal has the value TRUE, the signal z is assigned the expression (NOT a)
  - If GUARD is FALSE then the driver of z is de-activated, i.e. :
    - The events that appear on signal *a* do not affect the value of z
    - Signal z maintains its previous value.

Guarded signal assignment

- It is the only statement whose semantics is affected by its presence inside a guarded block
- Models sequential logic (hardware elements triggered by certain events)
- Functioning of the statement:
  - Every time where an event takes place on any of the signals from the expression and if the guard has the value TRUE or ot changes its value from FALSE to TRUE, the guarded signal assignment statement will be executed (new values are scheduled to the target signal)
  - If the value of the guard signal is FALSE, then the driver of the target signal remains unchanged (the value of the target signal remains also unchanged)

The equivalent process

```
bl: BLOCK (guard_expression)
       --SIGNAL GUARD: BOOLEAN;
BEGIN
       sig<= GUARDED waveform_expression;
END BLOCK b1;
```

Is equivalent with

```
bl:BLOCK (guard_expression)
       --SIGNAL GUARD: BOOLEAN;
BEGIN
PROCESS
BEGIN
       IF GUARD THEN
              sig<= waveform_expression;
       END IF;
       WAIT ON GUARD, signals_in_waveform_expression;
END PROCESS;
END BLOCK;
```

Other examples

Signal GUARD, although not explicitly declared, can be used inside the block. Example:

```
b2:BLOCK (set='1' AND reset='0')

BEGIN

        q<= '1' WHEN GUARD ELSE '0';

END BLOCK b2;
```

The signal assignment statement is not guarder => the driver of signal q will never be deactivated.

Other examples

In a block it is possible to declare explicitly a BOOLEAN signal named GUARD, to assign a logic expression to it and then to use it for guarded signal assignments inside the block. :

```
B3: BLOCK
        SIGNAL GUARD: BOOLEAN;
BEGIN
        GUARD <= reset='0' AND set='1';
        q <= GUARDED d;
END BLOCK;
```

Example of D flip-flop

```
ENTITY dff IS
       PORT(clk, d: IN BIT; q, qbar : OUT BIT);
END dff;
ARCHITECTURE garda OF dff IS
BEGIN
B: BLOCK(clk='1' AND NOT clk'STABLE)
-- had the guard been (clk='1'), D would have been level-
-- triggered (i.e., a latch), now is edge-triggered (flip-flop)
       SIGNAL temp: BIT;
BEGIN
       temp<=GUARDED d;
       q<=temp;
       qbar<=NOT temp;
END BLOCK;
END ARCHITECTURE;
```

• signal temp is visible only inside the block

•clk'STABLE is a BOOLEAN signal which is TRUE if clk did not have any events in the current simulation cycle

Example of visibility limitation

```
ARCHITECTURE ex OF ex IS
        SIGNAL s1, s2: INTEGER;
BEGIN
b1: BLOCK
        SIGNAL s3, s4: INTEGER;
BEGIN
        b2: BLOCK
                SIGNAL s1, s4, s5: INTEGER; -- s1 and s4
-- overwritten
        BEGIN
                s4<=...- refers to s4 from the block b2
                b1.s4 <= ...—in order to refer to s4 from b1
        END BLOCK b2;
END BLOCK b1;
END ARCHITECTURE;
```