



Descrizioni VHDL Behavioral

- In questo capitolo vedremo come la struttura di un sistema digitale è descritto in VHDL utilizzando descrizioni di tipo *comportamentale*.
- **Outline:**
 - process
 - wait statements, sensitivity list
 - variabili
 - assegnazione sequenziale
 - istruzioni sequenziali
 - assert



Convenzioni Tipografiche

- In tutte le slide si useranno le seguenti convenzioni tipografiche:
- **ENTITY, ARCHITECTURE, FOR, ...** sono comandi del linguaggio VHDL e sono parole riservate, ovvero non si possono usare come nomi assegnati dall'utente ad oggetti.
- **Nome_oggetto** è un oggetto del VHDL a cui l'utente ha assegnato il nome `nome_oggetto`.
- *Entity, architecture, etc.* fa riferimento al concetto di entity in VHDL e non al comando entity che permette di instanziare una entity.
- Il VHDL non è case sensitive.
- Per introdurre dei commenti si usano due segni '-':
-- commento



Process (1)

- Il **processo** (*process*) è l’istruzione fondamentale delle descrizioni comportamentali (behavioral) in VHDL;
- Un costrutto process rappresenta una porzione di un progetto, che viene descritta dal punto di vista algoritmico:
 - È importante il suo comportamento (cosa fa), ma non la sua implementazione (come lo fa);
- Un processo:
 - è uno *statement concorrente* (come una assegnazione o una istanziazione di un componente) che può essere usato in un architecture body e reagire “contemporaneamente” agli altri statement concorrenti,
 - contiene *solo istruzioni sequenziali* (assegnazione sequenziali, if-then-else, case, ...)
 - è un insieme di statement sequenziali, ma considerato come *un unico statement* è concorrente con altri statement concorrenti;



Process (2)

- Un process può essere identificato durante una simulazione attraverso una label (opzionale);
- Un **processo** è composto da 3 parti:
 1. una *sensitivity list* (opzionale) che contiene informazioni sugli eventi trigger
 2. una *parte dichiarativa*, che contiene le dichiarazioni dei tipi, di sottotipi, delle costanti, delle variabili, delle procedure e delle function, che potranno essere usate nel suo body (hanno visibilità locale);
 3. un *process body*, che rappresenta il comportamento del process, specificato tramite un insieme di istruzioni eseguite in maniera sequenziale fra gli statement BEGIN ed END PROCESS;

```
label: PROCESS (segnale_1, , segnale_n);  
  dichiarazioni di tipi;  
  dichiarazioni di costanti;  
  dichiarazioni di variabili;  
  dichiarazioni di procedure e functions;  
BEGIN  
  istruzione sequenziale_1;  
  
  istruzione sequenziale_N;  
END PROCESS label;
```



Process body

- Un **process body** consiste in un insieme di istruzioni sequenziali, il cui ordine di esecuzione è definito dall'utente e rappresentato dall'ordine con cui compaiono nel process body;
- Solo statement sequenziali sono leciti nel corpo di un process:
 - le assegnazioni \leq sono lecite (come vedremo l'assegnazione di segnale \leq è considerata sequenziale all'interno di un process);
 - le assegnazioni selezionate e condizionate non sono considerate concorrenti;



Timing (1)

- **Inizializzazione di una simulazione VHDL:**
- Al tempo $0+0\delta$ della simulazione viene eseguita una fase di inizializzazione in cui:
 - ai segnali vengono assegnati i loro valori iniziali (valori esplicitamente assegnati, oppure i valori di default per il loro tipo);
 - vengono eseguiti tutti i processi, finché tutti non raggiungono lo stato di sospensione;
 - anche le assegnazioni concorrenti \leq sono eseguite, essendo equivalenti a dei process, anche se poi il loro risultato si avrà nei delta successivi;
- Terminata la fase di inizializzazione, la simulazione è pilotata dagli eventi;
- **Esecuzione di un process:**
 - Ogni successiva esecuzione del processo è innescata da eventi che vanno esplicitamente indicati
 - Un process esegue tutte le istruzioni sequenziali e poi le ripete ripartendo dall'inizio, il tutto ripetuto come in un loop infinito;



Timing (2)

- In generale ci sono **due templates** per un process:
 - con *sensitivity list* (codice a sinistra);
 - con *istruzioni di tipo WAIT* (codice a destra);

```
PROCESS (segnale_1,      , segnale_n);  
  dichiarazioni  
BEGIN  
  process body  
END PROCESS;
```

```
PROCESS  
  dichiarazioni  
BEGIN  
  
  WAIT ;  
  
END PROCESS;
```

- Non è lecito usare una sensitivity list assieme ad una istruzione WAIT nel medesimo processo, quindi i due templates sono alternativi;
- La differenza fra i 2 tipi di process è nell'attivazione e nella sospensione:
 - nel caso di *sensitivity list*, il process viene attivato da un evento su un segnale che appartiene alla sensitivity list e sospeso quando raggiunge la fine del process;
 - nel caso di uso di istruzioni WAIT, quando il flusso di esecuzione incontra una WAIT, il processo viene sospeso e la sua esecuzione è ripresa quando la condizione richiesta dall'istruzione WAIT è verificata;



Wait statements (1)

- Un process esegue tutte le istruzioni sequenziali e poi le ripete ripartendo dall'inizio, tutto ripetuto come in un loop infinito;
- Nel caso di processo con istruzioni WAIT questo loop infinito:
 - può essere sospeso (suspended) attraverso una istruzione WAIT;
 - può essere riavviato (resumed) successivamente al verificarsi di una opportuna condizione;



Wait statements (2)

- **WAIT** è una istruzione sequenziale e quindi può comparire solo in un process body;
- I possibili tipi di uno statement WAIT sono:
 - WAIT FOR waiting_time;
 - WAIT ON waiting_sensitivity_list;
 - WAIT UNTIL waiting_condition;
 - WAIT;
- Esempi degli statement WAIT:

```
WAIT FOR 4ns;  
-- sospende il processo per 4ns  
  
WAIT ON s1, s2, clk;  
-- aspetta un evento su uno dei segnali s1, s2, clk  
  
WAIT UNTIL (clk EVENT and clk= 1 );  
-- aspetta il fronte 0->1 del segnale clk  
  
WAIT;  
--"forever"
```



Sensitivity List

- Una dichiarazione di processo può contenere opzionalmente una **sensitivity list**;
- La sensitivity list contiene gli identificativi dei segnali a cui il processo “reagisce”
 - un evento su questi segnali fa in modo che un processo sospeso si risvegli;
 - il processo viene eseguito fino alla fine;
 - il processo viene sospeso indefinitamente;

- Una sensitivity list è equivalente ad una istruzione del tipo:

WAIT ON sensitivity list;

alla fine del processo;

```
PROCESS (segnale_1,      , segnale_n);  
    dichiarazioni  
BEGIN  
    process body  
END PROCESS;
```

```
PROCESS  
    dichiarazioni  
BEGIN  
    process body  
    WAIT ON (segnale_1,      , segnale_n);  
END PROCESS;
```



Interazione con l'esterno

- **Visibilità di oggetti:**
 - Un processo ha visibilità di tutti gli “oggetti” definiti nella sua architettura (tipi, sottotipi, costanti, segnali, procedure, functions, ...): in altre parole lo scope di un process è lo stesso della architettura che lo contiene;
 - Le dichiarazioni della parte dichiarazione di un process sono invece locali al process;
- **Comunicazione con l'esterno:**
- L'unico modo che ha un processo per comunicare con l'esterno (e con altri process) è tramite i segnali di cui ha visibilità e che assegna e legge;
 - Non è possibile condividere variabili fra processi;
 - Non è possibile dichiarare segnali all'interno di un process;



Variabili

- Le **variabili** possono essere usate per rappresentare:
 - lo stato del sottosistema descritto dal process
 - dei risultati intermedi dell'elaborazione all'interno di un process;
- Ogni variabile ha una sua **dichiarazione** (un tipo, un valore di inizializzazione), come altri oggetti in VHDL (segnali, costanti, ...)

```
VARIABLE nome_variabile : tipo;
```

- E' possibile **assegnare un valore** ad una variabile attraverso l'operatore := (è analogo a <= per i segnali);
- Un signal e una variable sono oggetti completamente diversi dal punto di vista della tempificazione (anche se entrambi "portano" una informazione):
 - Una assegnazione di segnale <= comporta la schedulazione di un evento (al più un delta dopo), ma non ha mai l'effetto di una assegnazione immediata;
 - Una assegnazione di variabile := non comporta lo scheduling di nessun evento ed ha effetto di una assegnazione immediata;



Variabili

- La tempificazione delle assegnazioni fra segnali e delle variabili all'interno di un process è diversa.
- Consideriamo i 2 listati VHDL riportati:

```
SIGNAL ingr, tmp, out : std_logic;
```

```
PROCESS (ingr)  
BEGIN  
    tmp <= ingr;  
    out <= NOT tmp;  
END PROCESS;
```

```
SIGNAL ingr : std_logic;
```

```
PROCESS (ingr)  
VARIABLE tmp, out : std_logic;  
BEGIN  
    tmp := ingr;  
    out := NOT tmp;  
END PROCESS;
```



Variabili

```
SIGNAL ingr, tmp, out : std_logic;  
PROCESS (ingr)  
BEGIN  
    tmp <= ingr;  
    out <= NOT tmp;  
END PROCESS;
```

- Supponiamo che ci sia un evento $1 \rightarrow 0$ sul segnale *ingr* all'istante T ($ingr = 1 \rightarrow 0 @ T$):
- all'istante T il process viene attivato si schedula l'assegnazione del valore 0 a *tmp* per l'istante di simulazione $T+\delta$ ($tmp = 1 \rightarrow 0 @ T+\delta$);
- nello stesso istante T il flusso di esecuzione del process incontra la seconda assegnazione $out <= NOT tmp$ che viene schedulata per l'istante $T+\delta$;
- il valore che viene assegnato ad *out*, all'istante $T+\delta$, è $NOT tmp$, ma *tmp* vale ancora 1 (l'evento su *tmp* non è ancora avvenuto) e quindi l'evento (in realtà è una transizione) è $out = 1 @ T+\delta$;
- il processo raggiunge la fine (END PROCESS) e viene sospeso, in pratica il processo viene eseguito nell'istante T



Variabili

```
SIGNAL in : std_logic;  
PROCESS (in)  
VARIABLE tmp, out : std_logic;  
BEGIN  
    tmp := in;  
    out := NOT tmp;  
END PROCESS;
```

- A causa dello stesso evento $in = 1 \rightarrow 0 @ T$:
- il processo si attiva all'istante T, l'esecuzione incontra l'assegnazione di variabile $tmp := in$ che viene eseguita istantaneamente e quindi tmp all'istante T vale 0;
- all'istante T si incontra l'assegnazione $out := NOT tmp$ dove tmp vale già 0 e quindi il valore di out diviene 1 all'istante di simulazione T;



Assegnazione di un segnale

- I frammenti di codice riportati sono equivalenti:

```
ARCHITECTURE
BEGIN

    a <= b;

    c <= d;

END
```

```
ARCHITECTURE
BEGIN

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

    c <= d;

END
```

- Una **assegnazione di segnale** è *concorrente* nella figura di sinistra ma è *sequenziale* nel process della figura a destra;
- Sulla sinistra: un evento su b comporta una assegnazione e un evento su a;
- Sulla destra: b è la sensitivity list del processo
 - Lo statement process viene eseguito una sola volta per ogni evento su b e dopo viene sospeso fino al prossimo evento su b;



Assegnazione sequenziale

- Se una assegnazione di segnale appare dentro un processo (**assegnazione sequenziale**), questa ha effetto quando il processo sospende;
- Se si incontrano più assegnazioni di segnale all'interno dello stesso process, solo l'ultima è valida;

```
SIGNAL A, B, Y : integer;  
PROCESS (A, B)  
BEGIN  
    Y <= A * B;  
    Y <= B;  
END PROCESS;
```

- Se ad un segnale è assegnato un valore in un process, ed il segnale è uno che compare nella sensitivity list del process, allora un evento su questo segnale può riattivare il process;

```
SIGNAL A, B, C, X, Y, Z : integer;  
PROCESS (A, B, C)  
BEGIN  
    X <= A + 1;  
    Y <= A * B;  
    Z <= C * X;  
    B <= Z * C;  
END PROCESS;
```



Tempificazione di una assegnazione sequenziale (1)

- L'esecuzione del process riportato all'istante T , comporta che:
 - prima sul segnale x viene schedulata (all'istante $T+\delta$) una assegnazione del valore a ,
 - poi viene schedulata l'assegnazione di b a y (all'istante $T+\delta$);
- Le assegnazioni sono incontrate nello stesso istante di simulazione T e schedulate entrambe all'istante $T+\delta$, ma sequenzialmente;
- I segnali x e y ricevono i loro valori allo stesso istante ($T+\delta$);

```
ARCHITECTURE sequentiality_demo OF process_example IS
  BEGIN
    PROCESS (a,b)
      BEGIN

        x <= a;
        y <= b;

      END PROCESS;
    END process_example ;
```



Tempificazione di una assegnazione sequenziale (2)

- Allo stesso modo dell'esempio precedente gli eventi su x e y sono schedulati contemporaneamente, ma i segnali x e y commutano in istanti diversi:
- Il segnale y riceve il suo valore b, 4ns prima che x riceva il valore a;

```
ARCHITECTURE sequentiality_demo OF process_example IS
  BEGIN
    PROCESS
      BEGIN

        x <= a AFTER 10 NS;
        y <= b AFTER 6 NS;

      END PROCESS;
    END process_example ;
```



Tempificazione di una assegnazione sequenziale (3)

- Bisogna tenere bene a mente la tempificazione delle assegnazioni dei segnali all'interno dei body dei process;
- A volte il significato può essere diverso da quello che si potrebbe intuire usando la semantica di linguaggi come il C;
- Nell'es. successivo, l'evento '1' su x_sig a seguito dell'assegnazione $x_sig \leq 1$ è schedulato un δ dopo la valutazione della condizione dell'IF;
- Quindi se all'istante in cui è stato attivato il process $x_sig = 0$ verrà preso il branch 2;
- In altre parole l'IF non risente dell'assegnazione precedente;
- Se si fosse usata una variabile al posto del segnale x_sig , il comportamento sarebbe stato quello analogo al codice C (sarebbe stata eseguita l'istruzione_1);

```
ARCHITECTURE sequentiality_demo OF
    process_example IS
    SIGNAL x_sig : BIT := '0';
    BEGIN
        PROCESS
        BEGIN

            x_sig <= '1';
            IF x_sig = '1' THEN
                istruzione_1
            ELSE
                istruzione_2
            END IF;

        END PROCESS;
    END process_example ;
```



Esempio

- Ci sono 3 statement concorrenti:
 - Il processo dff è sensibile ai 3 segnali rst, set e clk;
 - Il segnale q propaga un evento su state dopo un delta;
 - Un evento su state si propaga su qb dopo un delta;

```
ENTITY d_sr_flipflop IS
  GENERIC (sq_delay, rq_delay, cq_delay : TIME := 6 NS);
  PORT (d, set, rst, clk : IN BIT; q, qb : OUT BIT);
END d_sr_flipflop;
--
ARCHITECTURE behavioral OF d_sr_flipflop IS
  SIGNAL state : BIT := '0';
BEGIN
  dff: PROCESS (rst, set, clk)
    BEGIN
      IF set = '1' THEN state <= '1' AFTER sq_delay;
      ELSIF rst = '1' THEN state <= '0' AFTER rq_delay;
      ELSIF clk = '1' AND clk'EVENT THEN state <= d AFTER cq_delay;
    END IF;
  END PROCESS dff;

  q <= state; qb <= NOT state;
END behavioral;
```



Istruzioni sequenziali

- All'interno del process è possibile usare un insieme completo di istruzioni sequenziali, in maniera del tutto simile a quanto è possibile fare con qualsiasi linguaggio di programmazione:
 - IF ... THEN
 - CASE ... WHEN
 - FOR ... LOOP
 - WHILE ... LOOP



If Then

- Il costrutto IF...THEN permette di controllare il flusso di esecuzione all'interno di un process body;
- Le condizioni condition sono espressioni booleane, che se vere (true) abilitano l'esecuzione del ramo relativo composto da statement sequenziali;
- Il costrutto IF...THEN può contenere anche degli ELSIF che vengono eseguiti quando le precedenti condizioni non sono verificate;
- E' possibile anche usare una clausola ELSE per raccogliere i casi esclusi da tutti i rami precedenti;
- I costrutti IF...THEN sono utili quando i casi non sono tutti mutualmente esclusivi e si vuole stabilire una "priorità" di esecuzione nei confronti;

```
IF condition THEN
    sequenza_di_istruzioni_1;
END IF;
```

```
IF condition_1 THEN
    sequenza_di_istruzioni_1;
ELSIF condition_2 THEN
    sequenza_di_istruzioni_2;
END IF;
```

```
IF condition_1 THEN
    sequenza_di_istruzioni_1;
ELSIF condition_2 THEN
    sequenza_di_istruzioni_2;

ELSIF condition_N THEN
    sequenza_di_istruzioni_N;
ELSE
    sequenza_di_istruzioni_N+1;
END IF;
```



If Then

- Ad esempio nel caso di un flip-flop D con set e reset asincroni i vari segnali hanno priorità diversa (i segnali asincroni hanno priorità maggiore);

```
dff: PROCESS (rst, set, clk)
  BEGIN
    IF set = '1' THEN state <= '1';
      ELSIF rst = '1' THEN state <= '0';
        ELSIF clk = '1' AND clk'EVENT THEN state <= d;
    END IF;
  END PROCESS dff;
```



Case When

- Quando i casi sono mutuamente esclusivi può essere utile usare uno statement di tipo CASE ... WHEN;

```
CASE segnale/variabile_di_selezione IS
  WHEN caso_1 =>
    sequenza_di_istruzioni_1;
  WHEN caso_2 =>
    sequenza_di_istruzioni_2;

  WHEN caso_N =>
    sequenza_di_istruzioni_N;
END CASE;
```

- Quando i casi enumerati non esauriscono tutti i possibili casi si può usare il caso OTHERS:

```
CASE segnale/variabile_di_selezione IS
  WHEN caso_1 =>
    sequenza_di_istruzioni_1;

  WHEN caso_N =>
    sequenza_di_istruzioni_N;
  WHEN OTHERS =>
    sequenza_di_istruzioni_N+1;
END CASE;
```



Case When

- Con un costrutto CASE...WHEN si può implementare una qualunque tabella di verità;
- In figura è riportata una implementazione comportamentale di una XOR a 3 ingressi;

```
ENTITY XOR_3 IS PORT (  
    x : IN std_logic_vector(2 DOWNTO 0);  
    y : OUT std_logic);  
END XOR_3 ;  
--  
ARCHITECTURE behavioral OF XOR_3 IS  
BEGIN  
    PROCESS (x)  
    BEGIN  
        CASE x IS  
            WHEN 000 => y <= 0 ;  
            WHEN 001 => y <= 1 ;  
            WHEN 010 => y <= 1 ;  
            WHEN 011 => y <= 0 ;  
            WHEN 100 => y <= 1 ;  
            WHEN 101 => y <= 0 ;  
            WHEN 110 => y <= 0 ;  
            WHEN 111 => y <= 1 ;  
            WHEN OTHERS => y <= x ;  
        END IF;  
    END PROCESS;  
END behavioral;
```



For Loop

- Per eseguire un insieme di istruzioni un certo numero di volte si può usare il costrutto FOR ... LOOP;

```
FOR identifier IN range LOOP  
    sequenza_di_istruzioni;  
END LOOP;
```

- Lo schema FOR...LOOP permette di eseguire la sequenza di istruzioni che ne compone il loop body, un assegnato numero di volte;
- L'identificatore identifier dichiara un oggetto che assume valori consecutivi nell'assegnato range del loop:
 - dentro il loop, questo oggetto è trattato come una costante e quindi può essere solo letto ma non assegnato;
 - l'oggetto identifier non esiste fuori dal FOR...LOOP;



For Loop

- In figura è riportata una XOR generica ad N bit descritta in maniera comportamentale:

```
ENTITY XOR_N IS
  GENERIC (N : INTEGER);
  PORT (
    x : IN std_logic_vector(N-1 DOWNT0 0);
    y : OUT std_logic);
END XOR_N ;
--
ARCHITECTURE behavioral OF Function_3 IS
BEGIN
  PROCESS (x)
    VARIABLE cnt : INTEGER;
  BEGIN
    cnt := 0;
    FOR i IN 0 to N-1 LOOP
      IF (x(i) = 1) THEN
        cnt := cnt + 1;
      END IF;
    END LOOP;
    CASE (cnt MOD 2) IS
      WHEN 0 => y <= 0 ;
      WHEN 1 => y <= 1 ;
    END CASE;
  END PROCESS;
END behavioral;
```



While Loop

- Per eseguire un insieme di istruzioni finché non si verifica una data condizione si può usare il costrutto WHILE ... LOOP;

```
WHILE condition LOOP  
    sequenza_di_istruzioni;  
END LOOP;
```

- Sia per il costrutto FOR ... LOOP che WHILE ... LOOP sono disponibili gli statement:
 - EXIT: provoca l'uscita dal loop (si salta all'istruzione dopo l'END LOOP);
 - NEXT: forza l'inizio di un nuovo loop;



Assert

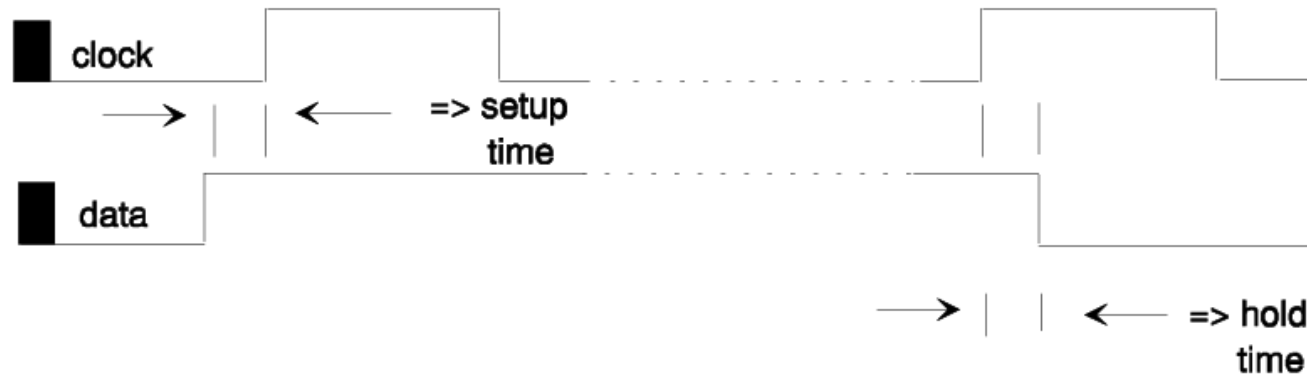
- Le istruzioni ASSERT possono essere usate sia come statement sequenziali che concorrenti;
- La sintassi è riportata in figura:

```
ASSERT condition REPORT message SEVERITY severity_level;
```

in cui:

- il messaggio “message” viene riportato dal simulatore quando si verifica la condizione NOT(condition);
- il severity_level può essere NOTE, WARNING, ERROR e FAILURE e può fermare o meno la simulazione;

Assert



- I flip-flop e gli altri elementi di memoria (e quindi anche i sistemi sequenziali) per funzionare correttamente hanno bisogno che siano verificate delle opportune relazioni fra segnali da campionare e segnali di clock;
 - vincolo di set-up: il segnale da campionare deve essere stabile per un certo tempo (tempo di set-up) prima del fronte attivo del clock;
 - vincolo di hold: il segnale di uscita deve essere stabile per un certo tempo (tempo di hold) dopo il fronte attivo del clock;
- Se uno di questi vincoli non è verificato non si può essere certi che il valore campionato sia quello presente quando c'è il fronte attivo del clock;



Assert

- Tempo di setup: quando (il clock commuta da 0 a 1), se (il data input non è stato stabile per almeno un tempo pari a setup-time) allora c'è una violazione del tempo di setup;

```
ASSERT
    NOT( (clock='1' AND clock'EVENT) AND (NOT data'STABLE(setup_time)) )
REPORT
    Violazione del tempo di setup
SEVERITY
    WARNING;
```

- Tempo di hold: quando (c'è una commutazione sul data input), se (il valore del clock è '1' ed il clock ha commutato meno di un hold-time), allora c'è una violazione del tempo di hold;

```
ASSERT
    NOT( (data'EVENT) AND (clock='1') AND (NOT clock'STABLE(hold_time)) )
REPORT
    Violazione del tempo di hold
SEVERITY
    WARNING;
```



Assert

```
ENTITY d_sr_flipflop IS
  GENERIC (sq_delay, rq_delay, cq_delay : TIME := 6 NS;
           set_up, hold : TIME := 4 NS);
  PORT (d, set, rst, clk : IN BIT; q, qb : OUT BIT);
BEGIN
  ASSERT (NOT (clk = '1' AND clk'EVENT AND NOT d'STABLE(set_up) ))
    REPORT "Set_up time violation"
    SEVERITY WARNING;

  ASSERT (NOT (d'EVENT AND clk = '1' AND NOT clk'STABLE(hold) ))
    REPORT "Hold time violation"
    SEVERITY WARNING;
END d_sr_flipflop;
--
ARCHITECTURE behavioral OF d_sr_flipflop IS
  SIGNAL state : BIT := '0';
  BEGIN
    dff: PROCESS (rst, set, clk)
      BEGIN
        ASSERT (NOT (set = '1' AND rst = '1'))
          REPORT "set and rst are both 1"
          SEVERITY NOTE;

        IF set = '1' THEN state <= '1' AFTER sq_delay;
          ELSIF rst = '1' THEN state <= '0' AFTER rq_delay;
            ELSIF clk = '1' AND clk'EVENT THEN state <= d AFTER cq_delay;
              END IF;
        END PROCESS dff;
    q <= state; qb <= NOT state;
  END behavioral;
```