12. File และ I/O

รศ.ณรงค์ บวบทอง
ภาควิชาวิศวกรรมไฟฟ้า
คณะวิศวกรรมศาสตร์

File และ I/O

- File I/O Operations
  - การประกาศค่าไฟล์ (File Declarations)
  - การเปิดและปิดไฟล์ (Opening and Closing Files)
  - การเปรียบเทียบรูปแบบ VHDL_87 and VHDL_93
  - การอ่านและเขียน (Reading and Writing Files)
- TEXTIO
  - TEXTIO Package
  - ขันตอนการจัดการไฟล์ด้วย TEXTIO
  - File declaration identifies
  - การอ่านและเขียนไฟล์
  - การอ่านเขียนค่ากับตัวแปรชนิด Line
  - การใช้ TEXTIO ใน TESTBENCH

File I/O Operations

ไฟล์ใช้สำหรับเก็บค่าต่างๆ เพื่อใช้ในการออกแบบหรือการทดสอบ

ทำนองเดียวกับ Variable Signal และ Constant

ก่อนที่จะมีการใช้งานต้องมีการกำหนด Type ของออปเจ็คต่างๆเหล่านี้เสียก่อนหลังจากนั้นจึงจะสามารถนำไปใช้งานได้ การใช้งานไฟล์ก็เช่นเดียกันต้องมีการกำหนด Type ขึ้นเสียก่อน การใช้ไฟล์มีวิธีการดังนี้

- ประกาศไฟล์และชนิดของไฟล์ (File type)
- เปิดและปิดไฟล์ (Opening and Closing Files)
- อ่านและเขียนไฟล์ (Reading and Writing Files)

การประกาศไฟล์ (File Declarations)

รูปแบบการกำหนดไฟล์

type *file_type* **is file of** *type_mark* **;**

ประการ *file_type*

ตัวอย่างเช่น

type bitfile **is file of** bit_vector (0 to 31) ;

type text **is file of** string;

type intf **is file of** integer;

หลังจากนี้แล้วเราสามารถกำหนดออปเจ็คใหม่จาก type ที่กำหนดแล้วได้ เช่น

file Bit_file : bitfile;

file input_file : text;

file Int_file : intf;

เปรียบเทียบ VHDL_93 กับ VHDL-87

VHDL-93

file ***file_name*** {,...} : *subtype_indication*
[[**open** *file_open_kind_expession*] **is** *string_expression*] **;**

type file_open_kind **is (** read_mode, write_mode, append_mode);

example..

file p_file : bitfile **is** "C:\CLASS\ALU.PAT" ;

VHDL-87

file ***file_name*** {,...} : *subtype_indication*
[[**open in** | **out**] **is** *string_expression*] **;**

Note - This is not Upward Compatable with VHDL-93. Keywords **in** and **out** replaced with file_open_kind.

หลีกเลี่ยงการกำหนด Multiple File Objects กับ Physical File ตัวเดียวกัน

ถึงแม้ว่า VHDL จะมีกฎกติกาที่ชัดเจนแต่ก็ทำนายไม่ได้เสมอไป

เปิดและปิดไฟล์ (Opening and Closing Files)

```
procedure FILE_OPEN (file File_handle: FILE_TYPE;
                     File_name : in STRING;
                     Open_Kind: in FILE_OPEN_KIND := READ_MODE);

procedure FILE_OPEN (File_Status: out FILE_OPEN_STATUS;
                     file File_handle: FILE_TYPE;
                     File_name : in STRING;
                     Open_Kind: in FILE_OPEN_KIND := READ_MODE);

procedure FILE_CLOSE (file f : FILE_TYPE);
```

-file pointer

-ชื่อของไฟล์ที่จะเขียนและอ่าน

-Mode of the file

- -READ_MODE เป็น default mode
- -WRITE_MODE
- -APPEND_MODE

File_Status

- OPEN_OK - เปิดได้
- STATUS_ERROR - มีไฟล์เปิดอยู่แล้ว
- NAME_ERROR - ไม่พบไฟล์
- MODE_ERROR -ไฟล์ไม่สามารถเปิดโหมดนี้ได้

ตัวอย่างการเปิดปิดไฟล์แบบ Explicit และ Implicit file

Explicit

```
type intf is file of integer;                -- declare a file type in the architecture
process is                                   -- a template for a process
   file datain : intf;                       -- declare file handle
  variable fstatus: File_open_status;        -- declare file variable status
 ....
begin
  file_open(fstatus, dataout, “myfile.txt”, read_mode);
  ...
 end process;                                -- termination implicitly causes a call to
file_close
```

Implicit

```
type intf is file of integer;                          -- declare a file type in the architecture
process is                                             -- a template for a process
   file datain : intf open read_mode is “myfile.txt”;  -- declare file handle
   ....
  begin
  ...
 end process;                                          -- termination implicitly causes a call to file_close
```

อ่านและเขียนไฟล์ (Reading and Writing Files)

```
procedure READ (file File_handle: FILE_TYPE; value : out type);
procedure WRITE (file File_handle: FILE_TYPE; value : in type);
function ENDFILE (file File_handle: FILE_TYPE)  return boolean;
```

ตัวอย่างการเขียน ข้อความ

test 8 บรรทัด

ลงไฟล์ myfile.txt

```
-- select option 1993 Language for complier
entity filewr is
end filewr;
architecture beh of filewr is
begin
process is
type text is file of string;
file dataout : text;
variable fstatus : FILE_OPEN_STATUS;
begin
file_open(fstatus, dataout, "myfile.txt", write_mode);
For i in 1 to 8 loop
    write (dataout, "test "& cr & lf);
end loop;
wait;
end process;
end beh;
```

Text I/O

Textual input และ Output หรือ Text I/O เป็น processสำหรับการอ่านและเขียนไฟล์อักขระ (Text files)
ไฟล์อักขระนี้เป็นไฟล์ที่บรรจุด้วยรหัส ASCII ซึ่งมีคุณสมบัติดังนี้

- ประกอบด้วยอักษรของรหัส ASCII เป็นบรรทัด
- แต่ละบรรทัดปิดท้ายด้วยรหัส Carriage return
- แต่ละบรรทัดอาจมีหลายฟิลด์ได้ โดยแต่ละฟิลด์แยกกันด้วยเว้นวรรค

ในภาษาตระกูล c \r\n หรือ \n
-Line Feed – LF – \n – 0×0a
-Carriage Return – CR – \r – 0×0D

TEXTIO ถูกกำหนดไว้ใน Package TEXIO ของ IEEE std 1076 –1987
ดังนั้นเวลาจะใช้ TEXTIO ต้องเรียกใช้ use STD.TEXTO.ALL; โดยในแพกเกจจะประกอบด้วยโปรแกรมย่อยสำหรับจัดการกับไฟล์อักขระอย่างง่ายๆ

Package TEXTIO ทำอะไรได้บ้าง

Package TEXTIO of STD library
Important functions and procedures:

readline(...), read(...),
writeline(...), write(...),
endfile(...)

Additional data types (text, line)

READ / WRITE overloaded for all predefined data types:

bit, bit_vector
boolean
character, string
integer, real
time

ขันตอนการจัดการไฟล์ด้วย TEXTIO

1. กำหนด Type (Type Declarations)
   - Logical name of file
   - type of file
   - mode of file (in or out)
   - physical file by path name
2. อ่านหรือเขียนข้อมูลกับไฟล์ทีละบรรทัดด้วยคำสั่ง READLN และ WRITELN กระทำระหว่าง ไฟล์กับตัวแปรเก็บข้อมูลที่ละบรรทัด
3. อ่านหรือเขียนข้อมูลจากตัวแปรที่เก็บข้อมูลของแต่บรรทัดที่ละฟิลด์ กระทำระหว่าง ตัวแปรเก็บข้อมูลที่ละบรรทัดกับ ตัวแปรหรือสัญญาณ
4. ทำฟังก์ชั้นการจบไฟล์ ENDFILE

File declaration identifies

รูปแบบ

file logical_file_name : **TEXT is in** “..physical_file_name”;

Logical file name นี้ใช้สำหรับการอ้างถึงเท่านั้น ทำหน้าที่เป็น file handle นั้นเอง

ชื่อไฟล์ที่จะใช้งานจริง

Mode เป็น
IN สำหรับอ่านจากไฟล์
OUT สำหรับเขียนไฟล์
ไฟล์หนึ่งๆเปิดได้เพียงโหมดเดียว

ชนิดของไฟล์สำหรับ TEXTIO เป็นได้เพียง TEXT เท่านั้น

การอ่านเขียนไฟล์

รูปแบบการอ่านไฟล์

READLN (logical_file_name, line_name);

รูปแบบการเขียนไฟล์

WRITELN (logical_file_name, line_name);

การอ่านหรือเขียนต้องทำที่ละบรรทัด และต้องเป็นการกระทำระหว่างไฟล์ที่ถูกระบุด้วย logical_file_name กับ ตัวแปร line_name ที่เป็นตัวแปรที่กำหนดด้วยคำสั่ง Variable ให้เป็นชนิด Line เท่านั้น

การอ่านเขียนค่ากับตัวแปรชนิด Line

รูปแบบการอ่านที่ละฟิลด์

READ (line_name, object_name);

HREAD (line_name, object_name); -- อ่านแบบ HEXแล้วแปลงเป็น
-- binary vector

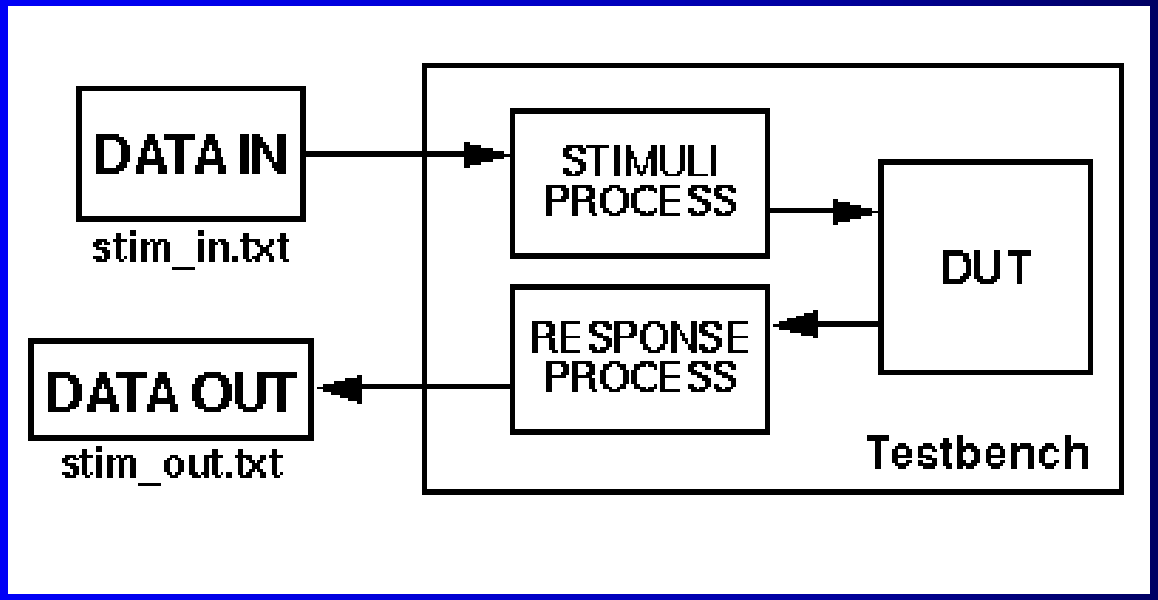
รูปแบบการเขียนที่ละฟิลด์

WRITE (line_name, object_name);

HWRITE (line_name, object_name); -- แปลง binary vector เป็น HEX

1. การอ่านหรือเขียนแต่ละครั้งจะได้ครั้งละฟิลด์ ฟิลด์ซ้ายมือสุดจะถูกกระทำก่อน
2. HREAD และ HWRITE เป็นฟังก์ชั้นใน IEEE.std_logic_textio Package
3. Object_name เป็นอะไรก็ได้เช่น signal, variable

การใช้ TEXTIO ใน TESTBENCH



ตัวอย่าง

ตัวอย่างโปรแกรมคูณขนาด 4 บิต คูณ 4 บิต

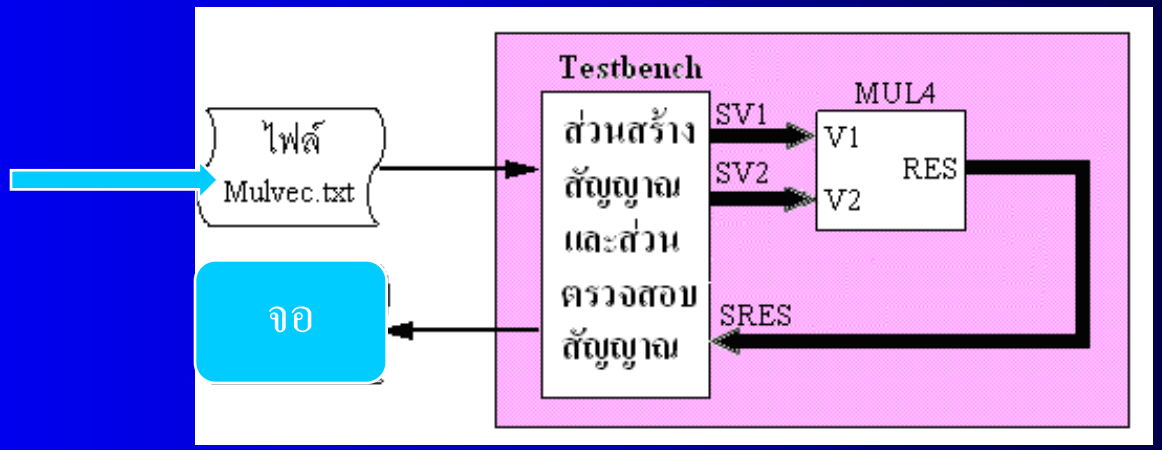
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity MUL4 is
port (V1, V2 : in std_logic_vector(3 downto 0);
      RES : out std_logic_vector(7 downto 0));
end MUL4;

architecture RTL of MUL4 is

begin
  RES <= V1 * V2;
end RTL;
```

ตัวอย่างไฟล์ข้อมูลสำหรับป้อนให้กับ V1 และ V2 ชื่อ mulvec.txt

```
0 0
0 8
0 F
1 0
1 8
1 F
4 0
4 8
4 F
8 8
8 F
F 0
F 8
F 9
```

Testbench แบบที่ 1

```
0 0
0 8
0 F
1 0
1 8
1 F
4 0
4 8
4 F
8 8
8 F
F 0
F 8
F 9
```



โปรแกรม Testbench แบบที่ 1 อ่านค่าจาก ไฟล์อินพุท Mulvec.txt แล้วส่งเข้าสู่วงจรคูณ ตรวจสอบผลการทำงานจากหน้าจอ

```
entity MUL4_TEB is
end MUL4_TEB;
architecture BEH of MUL4_TEB is
component MUL4
port (V1, V2 : in std_logic_vector(3 downto 0);
      RES : out std_logic_vector(7 downto 0));
end component;
signal SV1, SV2 : std_logic_vector(3 downto 0);
signal SRES : std_logic_vector(7 downto 0);
begin
DUT: MUL4 port map(V1 => SV1, V2 => SV2, RES => SRES);
STIMULI: process
  variable NUM_IN : line;
  variable CHAR : character;
  variable NUM1 : std_logic_vector(3 downto 0);
  variable NUM2 : std_logic_vector(3 downto 0);
  file STIMULI_IN: text is in "mulvec.txt";
begin
  wait for 20 ns;
  while not endfile(STIMULI_IN) loop
    readline(STIMULI_IN, NUM_IN);
    hread(NUM_IN,NUM1);
    SV1 <= NUM1;
    read(NUM_IN, CHAR);
    hread(NUM_IN, NUM2);
    SV2 <= NUM2;
    wait for 20 ns;
  end loop;
end process STIMULI;
end BEH;
```

ส่วน Library

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_textio.all;
use STD.textio.all;
```

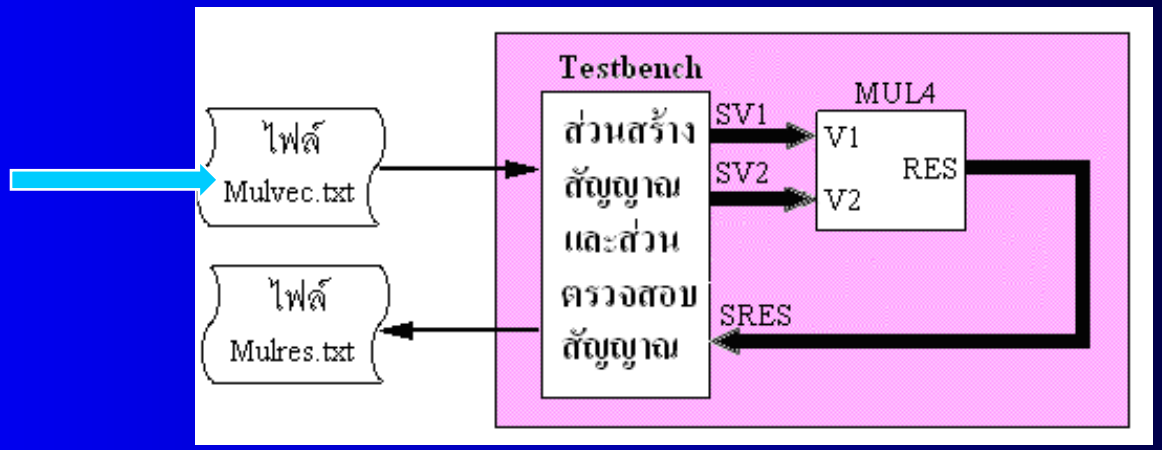
File Declaration

อ่านที่ละบรรทัด

อ่านแต่ละ field แล้วป้อนให้สัญญาณ

Testbench แบบที่ 2

```
0 0
0 8
0 F
1 0
1 8
1 F
4 0
4 8
4 F
8 8
8 F
F 0
F 8
F 9
```



โปรแกรม Testbench แบบที่ 2 อ่านค่าจากไฟล์อินพุท Mulvec.txt แล้วส่งเข้าสู่วงจรคูณ ผลการทำงานเก็บลงไฟล์ Mulres.txt

```
entity MUL4_TEB2 is
end MUL4_TEB2;

architecture BEH of MUL4_TEB2 is

component MUL4
port (V1, V2 : in std_logic_vector(3 downto 0);
      RES : out std_logic_vector(7 downto 0));
end component;

signal SV1, SV2 : std_logic_vector(3 downto 0);
signal SRES : std_logic_vector(7 downto 0);

begin
DUT: MUL4 port map(V1 => SV1, V2 => SV2, RES => SRES);

STIMULI: process
  variable NUM_IN : line;
  variable CHAR : character;
  variable NUM1 : std_logic_vector(3 downto 0);
  variable NUM2 : std_logic_vector(3 downto 0);
  file STIMULI_IN: text is in "mulvec.txt";
begin
  wait for 100 ns;
  while not endfile(STIMULI_IN) loop
    readline(STIMULI_IN, NUM_IN);
    hread(NUM_IN,NUM1);
    SV1 <= NUM1;
    read(NUM_IN, CHAR);
    hread(NUM_IN, NUM2);
    SV2 <= NUM2;
    wait for 500 ns;
  end loop;
end process STIMULI;
```

ส่วน Library

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_textio.all;
use STD.textio.all;
```

Process การส่งสัญญาณ

```
RESPONSE: process(SRES)
  variable NUM_OUT : line;
  variable CSPACE : character := ' ';
  file STIMULI_OUT: text is out "mulres.txt";
  begin
    write(NUM_OUT,now);
    write(NUM_OUT,CSPACE);
    write(NUM_OUT,SV1);
    write(NUM_OUT,CSPACE);
    write(NUM_OUT,SV2);
    write(NUM_OUT,CSPACE);
    write(NUM_OUT,SRES);
    write(NUM_OUT,CSPACE);
    hwrite(NUM_OUT,SRES);
    writeline(STIMULI_OUT,NUM_OUT);
end process RESPONSE;

end BEH;
```

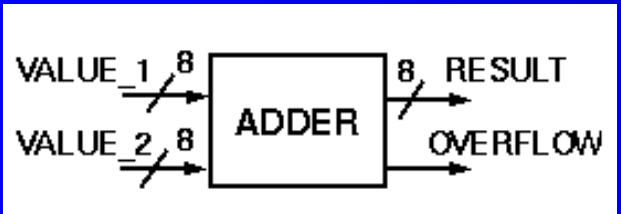
Process การตรวจสอบสัญญาณ

ไฟล์ผลการทำงาน mulres.txt

0 ns UUUU UUUU UUUUUUUU 00
0 ns UUUU UUUU XXXXXXXX 00
100 ns 0000 0000 00000000 00
2100 ns 0001 1000 00001000 08
2600 ns 0001 1111 00001111 0F
3100 ns 0100 0000 00000000 00
3600 ns 0100 1000 00100000 20
4100 ns 0100 1111 00111100 3C
4600 ns 1000 1000 01000000 40
5100 ns 1000 1111 01111000 78
5600 ns 1111 0000 00000000 00
6100 ns 1111 1000 01111000 78
6600 ns 1111 1001 10000111 87

จบ

Example for File I/O



ADDER module

- Adds two 8 bit vectors and provides an 8 bit result vector
- Generates an overflow signal

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity ADDER is
  port(VALUE_1      : in   std_logic_vector(7 downto 0);
       VALUE_2      : in   std_logic_vector(7 downto 0);
       OVERFLOW : out std_logic;
       RESULT       : out std_logic_vector(7 downto 0));
end ADDER;

architecture RTL of ADDER is
  signal INT_RES    : std_logic_vector(8 downto 0);
  signal INT_VAL_1 : std_logic_vector(8 downto 0);
  signal INT_VAL_2 : std_logic_vector(8 downto 0);
begin
  INT_VAL_1   <= '0' & VALUE_1;
  INT_VAL_2   <= '0' & VALUE_2;
  INT_RES      <= INT_VAL_1 + INT_VAL_2;
  RESULT       <= INT_RES(7 downto 0);
  OVERFLOW <= INT_RES(8);
end RTL;
```

Entity and architecture of the ADDER module

std_logic_unsigned package

- copyright by Synopsys (EDA software company)
- not standardized by IEEE
- overloaded mathematical operators where std_logic_vector is treated as unsigned number

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_textio.all;
use STD.textio.all;
```

Add package std.textio and IEEE.std_logic_textio for file I/O functions and procedures

```
entity TB_ADDER is
end TB_ADDER;

architecture BEH of TB_ADDER is
  component ADDER
    port(VALUE_1   : in  std_logic_vector(7 downto 0);
         VALUE_2   : in  std_logic_vector(7 downto 0);
         OVERFLOW : out std_logic;
         RESULT    : out std_logic_vector(7 downto 0));
  end component;

  signal W_VALUE_1   : std_logic_vector(7 downto 0);
  signal W_VALUE_2   : std_logic_vector(7 downto 0);
  signal W_OVERFLOW : std_logic;
  signal W_RESULT    : std_logic_vector(7 downto 0);

begin
  DUT : ADDER
    port map(VALUE_1   => W_VALUE_1,
             VALUE_2   => W_VALUE_2,
             OVERFLOW => W_OVERFLOW,
             RESULT    => W_RESULT);
```

Common testbench structure

- Empty entity; no external interface
- Component declaration and instantiation
- Definition of internal signals to connect the input/output ports with the stimuli/response analysis proc

```
STIMULI : process
   variable L_IN : line;
  variable CHAR : character;
  variable DATA_1 : std_logic_vector(7 downto 0);
  variable DATA_2 : std_logic_vector(7 downto 0);
   file STIM_IN : text is in "stim_in.txt";
begin
  W_VALUE_1 <= (others => '0');
  W_VALUE_2 <= (others => '0');
  wait for PERIOD;
  while not endfile (STIM_IN) loop
     readline (STIM_IN, L_IN);
     hread (L_IN, DATA_1);
    W_VALUE_1 <= DATA_1;
     read (L_IN, CHAR);
     hread (L_IN, DATA_2);
    W_VALUE_2 <= DATA_2;
    wait for PERIOD;
  end loop;
  wait;
end process STIMULI;
```

STIMULI process

- File access is limited to only one line at a certain time
- Only variables are allowed for the parameters of the read functions
- The function hread(...) is defined in the IEEE.std_logic_textio package; it reads hex values and transforms them into a binary vector

```
00 A1
FF 01
FF 00
11 55
0F 01
1F 05
AA F3
```

- Stimuli file "stim_in.txt"
- Each line contains two hex values to stimulate the inputs of the ADDER module

```
RESPONSE : process(W_RESULT)
  variable L_OUT : line;
  variable CHAR_SPACE : character := ' ';
  file STIM_OUT : text is out "stim_out.txt";
begin
   write (L_OUT, now);
   write (L_OUT, CHAR_SPACE);
  write (L_OUT, W_RESULT);
  write (L_OUT, CHAR_SPACE);
  hwrite (L_OUT, W_RESULT);
  write (L_OUT, CHAR_SPACE);
   write (L_OUT, W_OVERFLOW);
   writeline (STIM_OUT, L_OUT);
end process RESPONSE;
```

Response process of the testbench

- 'NOW' is a function returning the current simulation time
- Several write commands assemble a line
- Writeline saves this line in the file
- The function hwrite(...) is defined in the IEEE.std_logic_textio package; it transforms a binary vector to a hex value and stores it in the line

```
0 NS UUUUUUUU 00 U
0 NS XXXXXXXX 00 X
0 NS 00000000 00 0
20 NS 10100001 A1 0
40 NS 00000000 00 1
60 NS 11111111 FF 0
80 NS 01100110 66 0
100 NS 00010000 10 0
120 NS 00100100 24 0
140 NS 10011101 9D 1
```

Response file "stim_out.txt"

- 4 columns containing:
  - Simulation time
  - 8 bit result value (binary and hex)
  - Overflow bit