# Zaawansowane typy danych

# Program wykładu

- **Typy predefiniowane (*Predefined Types*)**
- **Typy rozszerzone (*Extended Types*)**
  - **Typy wyliczeniowe (*Enumerated Types*)**
  - **Podtypy (*Subtypes*)**
- **Typy złożone (*Composite Types*)**
  - **Tablice jednowymiarowe i wielowymiarowe (*Arrays*)**
  - **Rekordy (*Records*)**
- **Inne typy predefiniowane**
  - **Pliki (*Files*)**
  - **Linie (*Lines*)**

## Predefiniowane typy danych

# Pakiet standard

```
package standard is
  type boolean is (false, true);
  type bit is ('0', '1');
  type character is (
       nul, soh, stx, etx, eot, enq, ack, bel,...
       ...'ř', 'ů', 'ú', 'ű', 'ü', 'ý', 'ţ', '˙' );
  type severity_level is (note, warning, error, failure);
  type integer is range -2147483647 to 2147483647;
  type real is range -1.0E308 to 1.0E308;
  type time is range -2147483647 to 2147483647
    units
       fs;
       ps = 1000 fs;
       ns = 1000 ps;
       us = 1000 ns;
       ms = 1000 us;
       sec = 1000 ms;
       min = 60 sec;
       hr = 60 min;
    end units;
```

## Predefiniowane typy danych

# Pakiet standard

```
  subtype delay_length is time range 0 fs to time'high
  impure function now return delay_length;
  subtype natural is integer range 0 to integer'high;
  subtype positive is integer range 1 to integer'high;
  type string is array (positive range <>) of character;
  type bit_vector is array (natural range <>) of bit;
  type file_open_kind is (
    read_mode,
    write_mode,
    append_mode);
  type file_open_status is (
    open_ok,
    status_error,
    name_error,
    mode_error);
  attribute foreign : string;
end standard;
```

# Typy wyliczeniowe

**W deklaracji występuje lista nazw lub wartości definiujących nowy typ. Jest dogodnym środkiem np. dla symbolicznej reprezentacji kodów.**

## Składnia:

**type** *identifier* **is** **(***item* {**,** *item*}**) ;**

*item:* {*character_literal* | *identifier*}

## Przykłady:

*literals:*
```
type fiveval is ('?', '0', '1', 'Z', 'X');
```
*identifiers:*
```
type light is (red, yellow, green);
type instr is (load, store, add, sub);
```

# Typy wyliczeniowe

```
architecture behave of cpu is
type instr is (lda, sta, add);
begin process
variable a, b, data: integer;
variable opcode: instr;
begin
process (...)
..........
  case opcode is
     when lda => a := data;
     when sta => data := a;
     when add => a := a + data;
  end case;
  wait on data;
end process;
end behave;
```

**Kodowanie elementów listy dla syntezy:**

**00** add

**01** lda

**10** sta

**11** ---

```
type instr is
(add,lda,ldb,invalid);
```

## Synteza typów
# Typy wyliczeniowe

**Deklaracja:** **Jest syntezowana jako:**

```
type direction is (left, right, up, down);          -- two wires
type cpu_op is (execute, load, store);              -- two wires
                        -- the encoding of 11 is a "don't care"

subtype mem_op is cpu_op range load to store;       -- two wires
                -- the encodings of 00 and 11 are "don't cares"

-- User Defined Encoding
attribute enum_encoding: string;
attribute enum_encoding of cpu_op: type is
  "001" &     -- execute
  "010" &     -- load
  "100";      -- store
```

# Typy wyliczeniowe

- Typy wyliczeniowe w procesie syntezy domyślnie kodowane są binarnie. Do poszczególnych elementów (L) typu asygnowane są kolejne wartości, przy czym pierwszy od lewej otrzymuje wartość zero.

- Liczba elementów (N) obiektu reprezentującego typ wyliczeniowy będzie najmniejszą z możliwych liczbą, spełniającą warunek: $L <= 2^N$

# Wyrażenia kwalifikujące

**W niektórych przypadkach nie można rozróżnić typu literałów np.: `char'1'`, `bit'1'` czy `std_logic'1'`. Są one niejednoznaczne. Stosuje się wówczas tzw. rzutowanie typów (*type casting*).**

**Składnia:**

*type'* **(***literal* | *expression***) ;**

**Przykłady:**

```
bit'('1')

function ToInt (d: bit_vector) return (integer);
function ToInt (d: std_logic_vector) return (integer);

ToInt("1010");                    -- ???

ToInt(bit_vector'("1010"));       -- OK!
```

# Konwersje typów

**VHDL jest rygorystyczny pod względem typów, tzn. nie pozwala na przypisywanie danemu obiektowi wartości właściwych innym typom, niż zadeklarowany dla obiektu.**

**Funkcje konwersji typów mogą być przydatne przy dopasowywaniu dwóch projektów, które używają różnych typów.**

**Przykład:**

```
process ..........;
   variable abc: fourval;
   variable xyz: value4;
begin
   xyz := convert4val (abc); -- wywołanie funkcji
end process;
```

# Konwersje typów

```
type fourval is ('X', 'L', 'H', 'Z');
type value4 is ('X', '0', '1', 'Z');
function convert4val (s: fourval) return value4 is
begin
   case s is
      when 'X' => return 'X';
      when 'L' => return '0';
      when 'H' => return '1';
      when 'Z' => return 'Z';
   end case;
end convert4val;
```

# Konwersje typów

**Często zdarza się potrzeba konwersji pomiędzy typem `integer` a typem `std_logic_vector`. Pakiety `IEEE.std_logic_unsigned` i `IEEE.std_logic_arith` zawierają odpowiednie funkcje konwersji:**

```
function conv_integer (arg: std_logic_vector)
  return integer;
function conv_std_logic_vector (arg: integer; size: integer)
  return std_logic_vector;
```

**Przykład:**

```
entity sel is
  port (a,b,s: in integer range 0 to 15;
              q: out std_logic_vector (3 downto 0));
end;

architecture good of sel is
begin
  q <= conv_std_logic_vector(a,4) when conv_integer(s) = 8 else
       conv_std_logic_vector(b,4);
end;
```

*dummy function*

⇩

# Konwersje typów

**Gdyby wszystkie sygnały były typu `std_logic_vector`, kod byłby bardziej zwięzły:**

```
architecture better of sel is
begin
  q <= a when conv_integer(s) = 8 else b;
end;
```

**Jeżeli narzędzie do syntezy pozwala na tzw. *operator overloading*, to kod można zapisać jeszcze zwięźlej:**

```
architecture best of sel is
begin
  q <= a when s = 8 else b;
end;
```

**Z punktu widzenia syntezy fakt używania funkcji konwersji nie ma znaczenia, ponieważ nie wymaga syntezy dodatkowej logiki (ale dla symulacji ma!). Dobre narzędzia do syntezy (j.w.) pozwalają na napisanie kodu bez funkcji konwersji.**

**W kodzie syntezowalnym należy używać typu std_logic_vector.**

# Podtypy skalarne

**Są uściśleniem wcześniej zdefiniowanych typów przez zawężenie ich zakresu. Przydatne przy deklarowaniu większej liczby takich samych obiektów oraz dla utrzymania czytelności i zwięzłości kodu.**

**Przykłady:**

```
subtype digit is integer range 0 to 9;
variable msd, lsd: digit;
```

**jest równoważny:**

```
variable msd, lsd: integer range 0 to 9;
```

```
type instr is (add, sub, mul, div, sta, stb, outa, xfr);
subtype arith is instr range add to div;
subtype pos is integer range 1 to 2147483647;
subtype nano is time range 0 ns to 1 us;
```

## Złożone typy danych
# Tablice

- Tablice składają się z elementów tego samego typu.
- Są one używane do opisu magistral, rejestrów i innych zbiorów elementów sprzętowych.
- Elementy tablic mogą być skalarami lub elementami złożonymi. (Nie można tworzyć np. tablic plików !!)
- Dostęp do poszczególnych elementów dzięki użyciu wskaźników.

Jedynymi predefiniowanymi typami tablicowymi są:

- `bit_vector (package STANDARD)`
- `string (package STANDARD)`
- `std_logic_vector (package STD_LOGIC_1164)`

Użytkownik musi sam deklarować nowe typy tablic dla elementów `real` i `integer`.

# Deklaracja typu tablicowego

**Aby zadeklarować typ tablicowy należy wyspecyfikować: nazwę typu, typ elementów, liczbę indeksów, typ indeksów oraz zakres indeksów (opcjonalnie).**

**Składnia:**

```
type name is array [index_constraint] of element_type ;
index_constraint:  [range_spec]
                   index_type range [range_spec]
                   index_type range <>
```

**Przykłady:**

```
type word8 is array (1 to 8) of bit;
type word8 is array (integer range 1 to 8) of bit;
type word is array (integer range <>) of bit;
type ram is array (1 to 8, 1 to 10) of bit;
```

**<> oznacza zakres nieograniczony**

# Deklaracja typu tablicowego

**Po deklaracji typu może on zostać użyty do deklaracji zmiennej lub sygnału.**

**Przykład:**

```
variable data_bus: word8;
variable register: word (1 to 10);
```

**Typ wyliczeniowy albo podtyp może również być użyty do oznaczenia zakresu zmienności indeksów.**

**Przykłady:**

```
type instruction is (add, sub, mul, div, lda, sta, xfr);
subtype arithmetic is instruction range add to div;
subtype digit is integer range 1 to 9;

type Ten_bit is array (digit) of bit;
type Inst_flag is array (instruction) of digit;
```

# Tablice wielowymiarowe

**Szczególnie użyteczne dla opisu pamięci RAM lub ROM.**

**Przykład:**

```
type memory is array (0 to 7, 0 to 3) of bit;
constant rom: memory := (('0', '0', '0', '0'),
                         ('0', '0', '0', '1'),
                         ('0', '0', '1', '0'),
                         ('0', '0', '1', '1'),
                         ('0', '1', '0', '0'),
                         ('0', '1', '0', '0'),
                         ('0', '1', '1', '0'),
                         ('0', '1', '0', '1'));
```

```
data_bit := rom(5,3); -- słowo 5, bit 3
```

# Tablice tablic

**Przykład:**

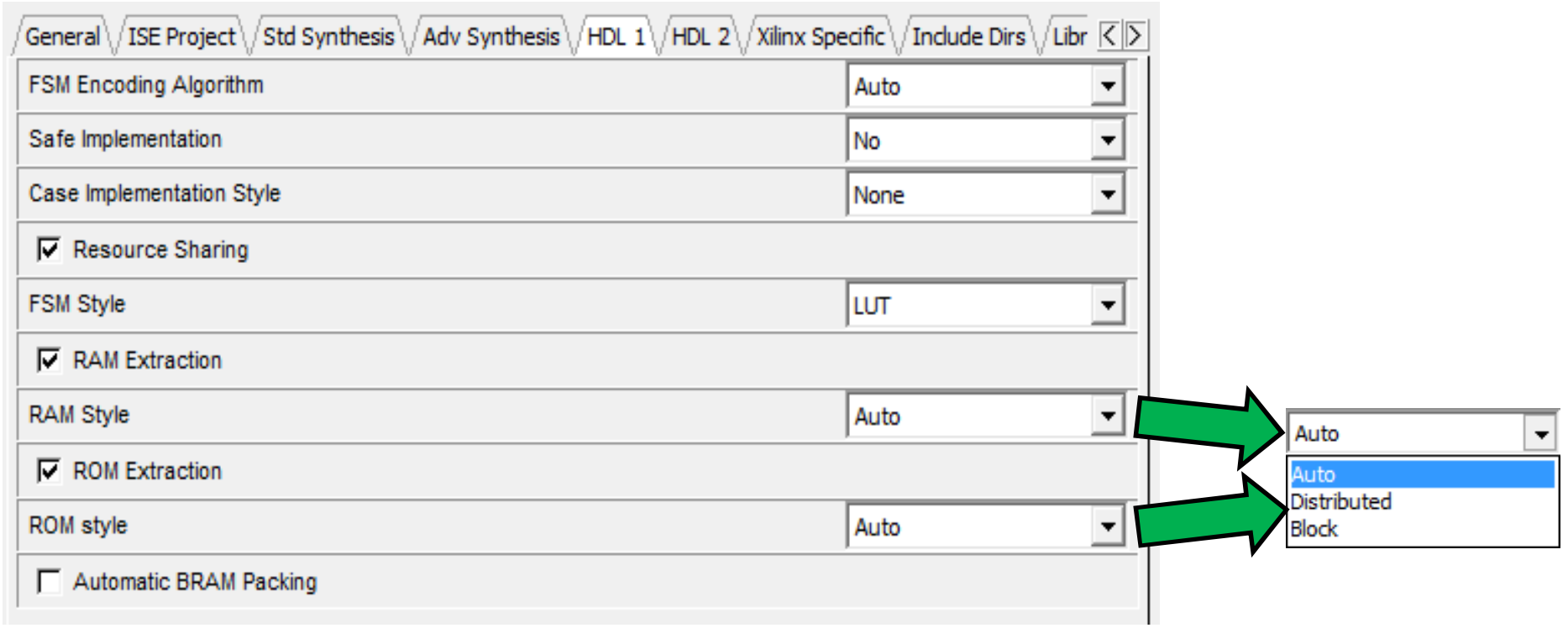
```
type word is array (0 to 3) of bit;
type memory is array (0 to 4) of word;
variable addr, index: integer;
variable data: word;
constant rom_data: memory := (('0', '0', '0', '0'),
                              ('0', '0', '0', '1'),
                              ('0', '0', '1', '0'),
                              ('0', '1', '1', '1'),
                              ('0', '1', '1', '1'));

data := rom_data(addr);

rom_data(addr)(index) --dostęp do pojedynczego bitu
```

# Pamięci

- implikowane (*inferred*) lub podstawiane (*instantiated*)
- implementowane jako rozproszone lub blokowe
  (w zal. od rozmiaru, szybkości i zajętego obszaru)
- synchroniczne (ew. z asynchronicznym odczytem – rozproszone)
- RAM (także inicjalizowane) lub ROM
- wykorzystywane także dla logiki kombinacyjnej i FSM

| Method | Advantages | Disadvantages |
|---|---|---|
| Inference | • Most generic way to incorporate RAMs into the design, allowing easy/automatic design migration from one FPGA family to another<br>• FAST simulation | • Requires specific coding styles<br>• Not all RAMs modes are supported<br>• Gives you the least control over implementation |
| CORE Generator software | Gives more control over the RAM creation | • May complicate design migration from one FPGA family to another<br>• Slower simulation comparing to Inference |
| Instantiation | Offers the most control over the implementation | • Limit and complicates design migration from one FPGA family to another<br>• Requires multiple instantiations to properly create the right RAM configuration |

# *XST Synthesis Options*



## Złożone typy danych
# Rekordy

**Grupują obiekty różnych typów. Elementy rekordów mogą być skalarami lub typami złożonymi i są dostępne przez nazwę.**

**Przykład:**

```
type two_digit is record
    sign: bit;
     msd: integer range 0 to 9;
     lsd: integer range 0 to 9;
end record;

process
variable acntr, bcntr: two_digit;
    begin
        acntr.sign := '1';
        acntr.msd := 1;
        acntr.lsd := acntr.msd;
        bcntr := two_digit'('0',3,6);
end process;
```

# Instrukcja alias

**Umożliwia wprowadzenie alternatywnej nazwy dla części obiektu, ułatwiającej dostęp.**

**Przykład:**

```
signal count: bit_vector (1 to 9);
   alias sign: bit is count (1);
   alias  msd: bit_vector (1 to 4) is count (2 to 5);
   alias  lsd: bit_vector (1 to 4) is count (6 to 9);

count := "1_1001_0000";
sign := '1';
msd := "1001";
lsd := msd;
```

## Predefiniowane typy tekstowe

# Typy `text` i `line`

**Należą do nich typy `text` oraz `line`. Używane są do operacji wejścia i wyjścia podczas symulacji. Występują w deklaracji `file` wraz z funkcjami odczytu, zapisu i pomocniczymi.**

**STD library - TEXTIO Package:**

```
readline, read, writeline, write
```

**Przykład:**

```
readline (F: in text; L: out line);
read (L: inout line; ITEM: integer);
```

**IEEE library – STD_LOGIC_TEXTIO Package:**

```
read, write
oread, owrite
hread, hwrite
```

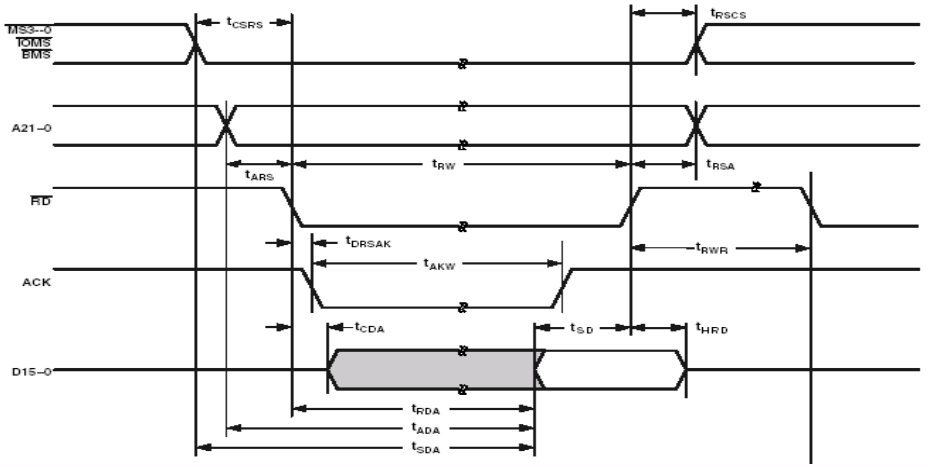
**Wbudowane:**

`endfile` (*filename*) , `endline` (*linename*)

# Przykład – procedura modelująca transfer na magistrali

```
procedure read_cycle (…) is

constant T_CSRS: TIME := H_clk_per/2 - 3ns;
constant T_ARS:  TIME := H_clk_per/2 - 3ns;
constant T_RW:   TIME := H_clk_per - 2ns + (WS * H_clk_per);
constant T_RSA:  TIME := H_clk_per/2 - 2ns;

begin
    ms3 <= '0';
    addr <= address;
    wait for T_ARS;
    rd <= '0';
    wait for T_RW;
    rd <= '1';
    data_read := data;
    wait for T_RSA;
    addr <= (others => 'Z');
    ms3 <= '1';
end procedure read_cycle;
```

| Parameter[1, 2] | | Min | Max | Unit |
|---|---|---|---|---|
| *Switching Characteristics* | | | | |
| $t_{CSRS}$ | Chip Select Asserted to $\overline{RD}$ Asserted Delay | $0.5t_{HCLK}-3$ | | ns |
| $t_{ARS}$ | Address Valid to $\overline{RD}$ Setup and Delay | $0.5t_{HCLK}-3$ | | ns |
| $t_{RSCS}$ | $\overline{RD}$ Deasserted to Chip Select Deasserted Setup | $0.5t_{HCLK}-2$ | | ns |
| $t_{RW}$ | $\overline{RD}$ Strobe Pulsewidth | $t_{HCLK}-2+W^3$ | | ns |
| $t_{RSA}$ | $\overline{RD}$ Deasserted to Address Invalid Setup | $0.5t_{HCLK}-2$ | | ns |
| $t_{RWR}$ | $\overline{RD}$ Deasserted to $\overline{WR}$, $\overline{RD}$ Asserted | $t_{HCLK}$ | | |
| *Timing Requirements* | | | | |
| $t_{AKW}$ | ACK Strobe Pulsewidth | $t_{HCLK}$ | | ns |
| $t_{RDA}$ | $\overline{RD}$ Asserted to Data Access Setup | | $t_{HCLK}-4+W^3$ | ns |
| $t_{ADA}$ | Address Valid to Data Access Setup | | $t_{HCLK}+W^3$ | ns |
| $t_{SDA}$ | Chip Select Asserted to Data Access Setup | | $t_{HCLK}+W^3$ | ns |
| $t_{SD}$ | Data Valid to $\overline{RD}$ Deasserted Setup | 7 | | ns |
| $t_{HRD}$ | $\overline{RD}$ Deasserted to Data Invalid Hold | 0 | | ns |
| $t_{DRSAK}$ | ACK Delay from $\overline{RD}$ Low | 0 | | ns |



## Predefiniowane typy tekstowe

# Przykład – metajęzyk do modelowania transferów na magistrali

adsp2191.txt ->

```
N 0028 0000 #czekaj na inicjalizację
W 4011 0001 #uruchom DCM
R 4011 0001 #sprawdź DCM
W 4001 01FF #ustaw długość akwizycji
W 4002 00FF #ustaw liczbę akumulacji
W 4000 0001 #ustaw 'Run' @ OneShot
N 0035 0000 #czekaj n * 12.5ns
W 4003 0000 #kasuj 'Int'
W 4000 0000 #kasuj 'Run'
R 0000 0200 #czytaj bufor danych
```

```
begin
...
  READ_CODE: process
  begin

  while not (endfile(transfers)) loop -- end file checking
    readline(transfers,IN_LINE);       -- read line of a file
    read(IN_LINE,code);                -- read in operation code
    hread(IN_LINE,address);            -- read in address
    hread(IN_LINE,data);               -- read in data
    code_array(i) := code;             -- put code in code table
    address_array(i) := address;       -- put address in address table
    data_wr_array(i) := data;          -- put data in data table
    i := i + 1;
  end loop;
```

## Predefiniowane typy tekstowe
# Przykład – wyniki symulacji w pliku raportów

```
....
for index in 0 to max loop
  decode: case code_array(index) is
  when 'W' =>
    ...
    write_cycle(...);                          -- write cycle timing model
    write(OUT_LINE, NOW, right, 12, ns);  -- current simulation time
    write(OUT_LINE, code_array(index), right);     -- current code
    write(OUT_LINE, address_array(index), right); -- current address
    write(OUT_LINE, data_wr_array(index), right); -- current data
    writeline(results,OUT_LINE);
  when 'R' => ...
....
```

simulation.txt ->

```
   ...
  6939 ns W 0100000000000000 0000000000000000
  6964 ns N
6994.5 ns R 0000000000000000 0000000100000000
  7025 ns R 0000000000000001 0000000100100000
7055.5 ns R 0000000000000010 0000000101000000
  7086 ns R 0000000000000011 0000000101100000
   ...
```

Dziękuję za uwagę!