Session 1432

A Small, Effective VHDL Subset for the Digital Systems Course

Pong P. Chu
Department of Electrical and Computer Engineering
Cleveland State University
Cleveland, OH 44115

1. Introduction

The availability of inexpensive, capable synthesis software tool has a significant impact on the design and implementation of digital systems. Many curriculums, as well as a number of textbooks[3,12], have integrated HDL (Hardware Description Language), such as VHDL, and synthesis tool into the introductory digital systems course[1,5,9,11,13]. VHDL is a very rich language and includes many constructs resembling a conventional programming language, including variables, loop and complex branch structures[2,6,10]. These constructs are intended to describe the "behavior" of a circuit, but not its internal implementation. These constructs are frequently abused by students who use them for synthesis. Instead of thinking hardware, some students just describe their design as a C-like program and hope the synthesis software can derive the hardware for them. This usually leads to inefficient, excessively complex implementation[4].

To remedy the problem, we introduce a small, but effective, subset of VHDL for the introductory digital systems course. The constructs in this set have clear mappings to hardware components so that a VHDL description can be easily transformed into a block diagram and vice versa. This approach forces students to be conscious about hardware structure but at the same time let them exercise the modern design practice and take advantage of the synthesis software. The paper is organized as follows: Section 2 provides background information and discusses the use and abuse of VHDL; Section 3 discusses the languages constructs selected for the subset and their hardware implementation; Section 4 illustrates the application of the subset for various circuits and the last section summarizes the paper.

2. Background

2.1 Content of the Digital Systems course

Digital systems is a core undergraduate course in the curriculum of electrical engineering, computer engineering or computer science. As indicated in the proposed IEEE/ACM Computing Curricula[7], this course "covers the digital building blocks, tools, and techniques in digital design and emphasis is on a building-block approach"[8]. The main focus is on the theory and practice of using gate-level components, such as simple logic gates and FF (Flip-Flop), and module-level components, such as adder, comparator, multiplexer and register. While HDL and synthesis software may be covered, they are not the main focus for this course. As suggested in the curricula[8], only a small fraction of time is allocated for these topics.

2.2 Overview on VHDL

An HDL should faithfully and accurately model and describe a circuit, whether already built or under development, from either structural or behavioral views, at the desired level of

abstraction[6]. VHDL is one of the two widely used HDLs. It is designed to model the basic characteristics of a digital circuit, which include the concepts of entity, connectivity, concurrency and timing[2]. The semantics of VHDL is entirely different from conventional programming languages, such as C or Java. It normally needs an entire book and an advanced course to completely cover the features and uses of the VHDL language.

2.3 Use and abuse of VHDL and synthesis software

Recent advancements make HDL and EDA (Electronic Design Automation) software available to everyday engineering design. Many curriculums and some textbooks are incorporating VHDL and synthesis and simulation software into instruction. There are several advantages of this approach. First, this helps students to get familiar with modern tools and design practice. Second, simulation allows students to observe the circuit operation in computer and better understand the basic principles. Third, synthesis software can realize the circuit in FPGA (Field Programmable Gate Array) device so that students can quickly obtain a physical prototype and test their design.

One major problem of this approach is the students' "software mentality". Most students in this course have some, or even extensive, prior programming experience, normally in C or C++. Since VHDL includes languages constructs, such as variable, loop, function, procedure etc., that are similar to the conventional programming languages, it is sometimes easier for the students to think in term of software algorithms rather than hardware circuitry. However, these language constructs are intended to model abstract, high-level system behavior. While they describe the desired external behavior, the resulting code does not provide any hint about the internal hardware components and the underlying implementation. This kind of code actually works against the "a building-block approach" philosophy described in the Computing Curricula and distracts the students from the real design issues.

An example of this kind of problem is illustrated by the following code. Assume that we wish to implement a 4-bit priority encoder with the following function table:

a(2)	*a(2)*	*a(1)*	*a(0)*	*code*	*valid*
1	x	x	x	1 1	1
0	1	x	x	1 0	1
0	0	1	x	0 1	1
0	0	0	1	0 0	1
0	0	0	0	0 0	0

In this circuit, the `a(3)`, `a(2)`, `a(1)` and `a(0)` are four input request signals, in which `a(3)` has the highest priority. The output `code` is the binary code of the highest request and the output `valid` indicates whether there is an active request. With C programming in mind, the VHDL code will be written as:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY p_encoder IS
   PORT( a: IN std_logic_vector(3 DOWNTO 0);
         valid: OUT std_logic;
```

```
        code: OUT std_logic_vector(1 DOWNTO 0));
END p_encoder ;
ARCHITECTURE behavioral_arch OF p_encoder IS
BEGIN
   PROCESS (a)
   BEGIN
      valid <= '0';
      code <= "00";
      FOR i IN 3 DOWNTO 0 LOOP
         IF a(i)='1' THEN
            code <= std_logic_vector(to_unsigned(i,2));
            valid <='1';
            EXIT;
         END IF;
      END LOOP;
   END PROCESS;
END behavioral_arch;
```

This program simply uses VHDL as the C language. It uses variables, sequential execution of the code and complex loop and branch constructs. While the description is correct, the program does not show any hint about its underlying circuit or how to realize the desired behavior in hardware. In order to realize this circuit in hardware, the students have to blindly rely on synthesis software and use it as a black box. However, since synthesis is a complex process that involves many untractable problems[4], there is no guarantee that the software can obtain an efficient implementation or even can derive an implementation.

Most of the time, this kind of "C-like" VHDL code is either not synthesizable or results in an unnecessarily complex circuit. Instead of thinking hardware and circuit, some students treat the digital design as a "trial-and-error" process and wish to derive a VHDL code that can be accepted by the synthesis software.

3. Minimal synthesizable VHDL subset

3.1 Selection criteria

The goal of finding a minimal synthesizable VHDL subset is to take advantage of the modern EDA software tools and FPGA devices but at the same time to keep students "conscious" about hardware and circuits. To achieve this goal, the subset should satisfy the following criteria. First, the subset should be simple so that it can be covered in a short amount of time and thus doesn't interfere with the regular curriculum. Abstract, high-level modeling language constructs and some more difficult VHDL concepts will be avoided. There should be a clear mapping between the selected language constructs and the hardware components so that students can quickly covert a VHDL code to a block diagram and vice versa. Second, the subset should be capable and flexible enough to describe moderately complex module-level circuits so that the students can simulate and synthesize their designs and eventually obtain physical prototypes. Third, the subset should be "upward compatible". This means that the concepts and principles introduced in this course should not contradict with the concepts and practices of a full-fledged VHDL course in the future curriculum.

To satisfy the criteria, the proposed subset includes four data types that can be easily mapped to logic 1 and logic 0, a collection of operators corresponding to the gate-level and module-level components, and two concurrent signal assignment statements as routing structures. We intentionally avoid VHDL process and sequential statements since they cannot be easily mapped

into hardware components and encourage students to develop C-like codes. The only exception is the inference of a D FF (Flip-Flop) or a register, in which a "register template" is used.

With this subset, the VHDL code can only contain concurrent signal assignment statements and register template. Each statement is treated as a circuit part with a small delay (i.e., the delta delay of VHDL) and the parts are connected via the common signal names. We can easily convert a VHDL code into a block diagram and vice versa.

The constructs selected for this subset and their corresponding hardware implementations are discussed in the following subsections.

3.2 Data types and package

VHDL is a strongly type language and supports a wide variety of data t types. Four data types are used in the subset and they can be easily mapped to logic 1 and logic 0 or an array of logic 0 and logic (i.e.; a bus) of digital circuit. These types are:

- `std_logic` and `std_logic_vector`
- `unsigned`
- `boolean`

The `std_logic` and `std_logic_vector` data type defined in IEEE `1164` package are used for general signal representation. The two types are selected because they are more versatile and general than the build-in `bit` and `bit_vector` types.

The `unsigned` data type is defined in IEEE `numeric_std` package. We prefer the `unsigned` data type than the VHDL's built-in `integer` data type since the `unsigned` data type explicitly specifies a signal's width and the representation (i.e., unsigned format). Simple type casting can convert the `std_logic_vector` data type to the `unsigned` data type and vice versa.

The `boolean` data type has a value of `true` or `false`. It is mainly for the condition signals used in a conditional concurrent signal assignment statement.

3.3 Operator

Four groups of VHDL operators are selected and all the operators can be mapped to standard gate-level or module-level components. These operators are:

- Logical operator: `AND`, `OR`, `XOR`, `NAND`, `NOR`, `XOR`
- Relational operator: `=`, `/=`, `<`, `<=`, `>`, `>=`
- Arithmetic operator: `+`, `-`
- Miscellaneous operator: `&`

The logic operator corresponds to the basic logic function used in Boolean algebra and each one can be mapped to a physically gate.

The relational operators can be mapped to physical comparators, and the arithmetic operators can be mapped to an adder or a subtractor. The `&` operator combines several signals to form a single, wider signal and it involves no extra logic. One common use of the `&` operator is to perform a fixed amount of shifting or rotation.

3.4 Expression

An expression can be thought as a formula that specifies how to compute a value. An expression can be a simple signal but it normally utilizes operators to construct a more sophisticated function. If the operators are limited to the subset defined in Section 3.3,

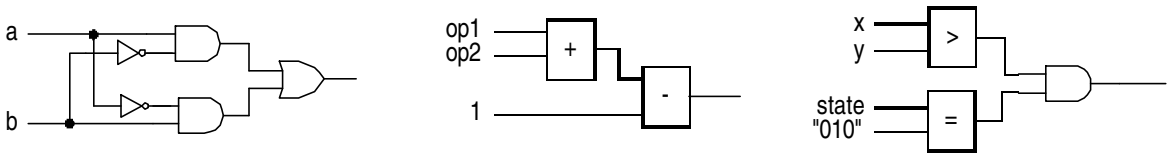


Figure 1. Circuit diagrams of simple expressions

an expression can be implemented by replacing the operators with the corresponding physical parts. Three examples are shown below:

- Example 1: `(a AND (NOT b)) OR ((NOT a) AND b)`
- Example 2: `op1 + op2 - 1`
- Example 3: `(x > y) AND (state = "010")`

The circuit diagrams of the three expressions are shown in Figure 1.

3.5 Simple concurrent signal assignment statement

The simplified syntax of a simple concurrent signal assignment statement is

```
sigal_name <= expression;
```

It is actually a special case of a conditional signal assignment statement. It is an expression assigned to a specific signal. In term of hardware, we can think that the computation result of the expression is connected to an output signal. For example, consider the statement

```
arith_result <= op1 + op2 - 1;
```

It means that the output of the `op1+op2-1` circuit is connected to an output signal named `arith_result`.

3.6 "Restricted" conditional signal assignment statement

The simplified syntax of a restricted conditional signal assignment statement is

```
sig <= exp_1 WHEN bool_1 ELSE
       exp_2 WHEN bool_2 ELSE
       exp_3 WHEN bool_3 ELSE
       . . .
       exp_n.;
```

The conditional signal assignment appears to force the order of evaluation of the expressions (`exp_i`) according to the corresponding Boolean signals (`bool_i`). However, in hardware, it actually specifies a "priority routing network". Consider a statement with three expressions:

```
sig <= exp_1 WHEN bool_1 ELSE
       exp_2 WHEN bool_2 ELSE
       exp_3.;
```

The block diagram is shown in Figure 2. Each ". . . `WHEN bool_i ELSE`" clause corresponds to a 2-to-1 multiplexer in which the `bool_i` signal functions as the selection signal. The two clauses form a priority routing network with two 2-to1 multiplexers. Three "clouds" represent the three circuits used to implement the expressions. Note that the expression circuits are operated concurrently and the routing network forms a path to pass the desired result to the output.

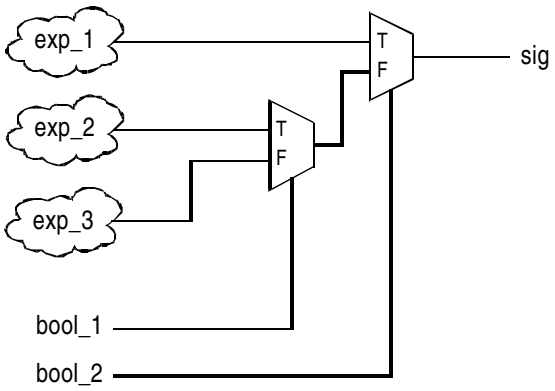


Figure 2. Block diagram of a conditional signal assignment statement

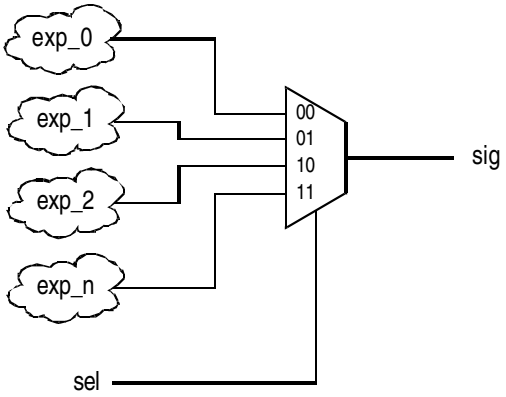


Figure 3. Block diagram of a selected signal assignment statement

3.7 "Restricted" selected signal assignment statement

The simplified syntax of a restricted selected signal assignment statement is

```
WITH sel SELECT
   sig <= exp_0 WHEN value_0,
          exp_1 WHEN value_1,
          exp_2 WHEN value_2,
          . . .
          exp_n.WHEN value_n;
```

The `sel` signal is with a data type of n+1 possible values (i.e., `value_0`, `value_1`, … , `value_n`). According to the current value of the `sel` signal, the result of the corresponding expression will be assigned to the output `sig` signal. In hardware, this statement implies an (n+1)-to-1 multiplexing network that has n+1 input ports. Each port is connected to a circuit that implements the corresponding expression (`exp_i`). The `sel` is the selection signal of a multiplexer, which has n+1 possible values. It specifies which input port will be connected to the output port. We restrict the `sel` signal to the `std_logic_vector` data type. For a k-bit `sel` signal, it has 2^k possible "valid" values and the statement specifies a 2^k-to-1 multiplexer.

Note that we need `WHEN OTHERS` clause in the end to cover the unused meta-value combinations. Consider a statement with 2-bit select signal:

```
WITH sel SELECT
   sig <= exp_0 WHEN "00",
          exp_1 WHEN "01",
          exp_2 WHEN "10",
          exp_n.WHEN OTHERS;
```

The block diagram is shown in Figure 3, using a 4-to-1 multiplexer for routing. As in Section 3.6, the "clouds" represent the circuits used to implement the expressions. Again, the expression circuits are operated concurrently although only one computation result is routed to the output.

3.8 Template for register

In VHDL, memory elements can be inferred in various mechanisms and can be easily "embedded" in code. To enforce good design practice, our approach is to separate the memory elements from the remaining circuit and describe them in explicit code segments. We use a process template, which is treated as a black box, to explicitly specify the desired D FFs or registers. The VHDL code segment is :

```
PROCESS(clk)
BEGIN
   IF (clk'EVENT AND clk='1') THEN
      q <= d;
   END IF;
END PROCESS;
```

The `clk` signal is the clock of the register and the `q` and `d` signals are the input and output respectively.

4. Application of the subset

Although the proposed VHDL subset is very simple, it is general and powerful enough to cover most circuits encountered in an introductory digital systems course. The subset can describe both gate-level and module-level circuits and can specify combinational circuits, regular sequential circuits as well as FSMs (Finite State Machine). The following subsections illustrate the usage of this subset for various circuits.

4.1 Gate-level circuit

Gate-level circuits are the designs based on basic logic gates. These circuits are normally described by logic expressions or truth tables. The logic expression can be translated to simple signal assignment with logical operators.

For example, the priority encoder can be expressed as[12]:

$h3 = a(3);\quad h2 = a(2) \bullet a(3)';\quad h1 = a(1) \bullet a(2)' \bullet a(3)'$

$code(1) = h2 + h3$

$code(2) = h1 + h3$

$valid = a(3) + a(2) + a(1) + a(0)$

This can be directly translated to VHDL code:

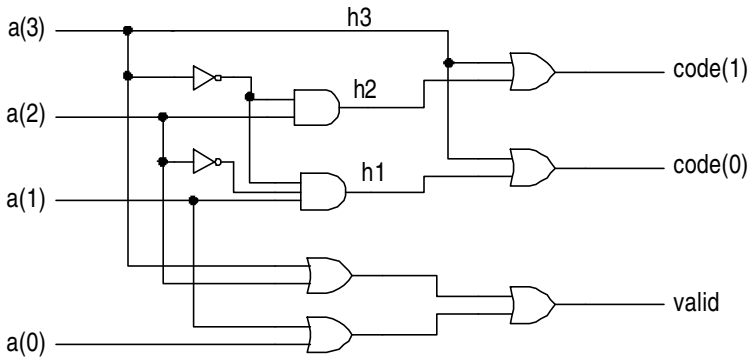


Figure 4. Gate-level implementation of 4-to-2 priority encoder

```
ARCHITECTURE gate_level_arch OF p_encoder IS
    SIGNAL h3, h2, h1: std_logic;
BEGIN
    h3 <= a(3);
    h2 <= a(2) AND (NOT a(3));
    h1 <= a(1) AND (NOT a(2)) AND (NOT a(3));
    code(1) <= h2 OR h3;
    code(0) <= h1 OR h3;
    valid <= (a(3) OR a(2)) OR (a(1) OR a(0));
END gate_level_arch ;
```

In the code, each statement represents a circuit part and the program can be easily converted to circuit diagram, as shown in Figure 4.

There is no explicit "table" construct defined in VHDL. However, the truth table or function table can be "emulated" by using the selected signal assignment statement. For example, we can describe the priority encoder based on the function table of Section 2.3. This table can be converted into the VHDL code:

```
ARCHITECTURE function_table_arch OF p_encoder IS
BEGIN
   WITH a SELECT
     code <=
        "11" WHEN "1000"|"1001"|"1010"|"1011"|
                  "1100"|"1101"|"1110"|"1111",
        "10" WHEN "0100"|"0101"|"0110"|"0111",
        "01" WHEN "0010"|"0011",
        "00" WHEN OTHERS;
    valid <= a(3) OR a(2) OR a(1) OR a(0);
END function_table_arch ;
```

Recall that any k-input function can be implemented by a 2^k-to-1 multiplexer[12]. The selected signal assignment statement essentially uses this approach to implement the truth table. This particular code needs two 1-bit 16-to-1 multiplexers. The multiplexing circuit will be simplified during the synthesis.

4.2 Module-level circuit

Module-level circuits are the designs that utilize larger components, such as comparators and adder, as well as routing structures. They are more abstract than the gate-level circuits.

The previous priority encoder can be described in more abstract VHDL code:

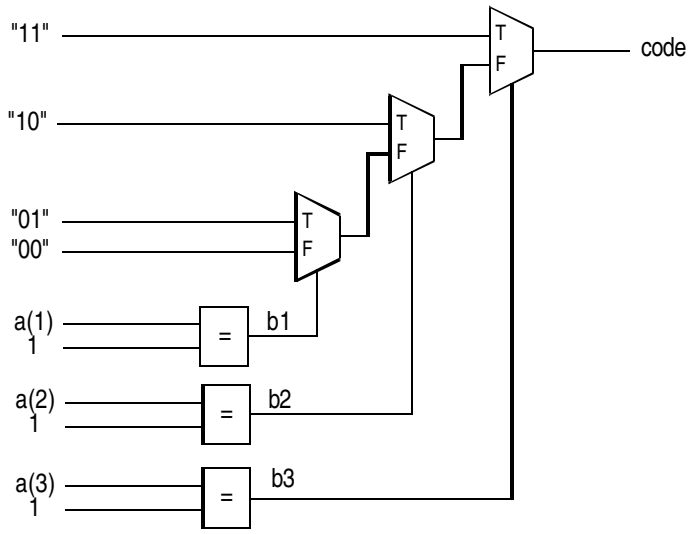


Figure 5. Module-level implementation of 4-to-2 priority encoder

```
ARCHITECTURE cond_assign_arch OF p_encoder IS
    SIGNAL b3, b2, b1: BOOLEAN;
BEGIN
    b3 <= a(3)='1';    b2 <= a(2)='1';    b1 <= a(1)='1';
    code <= "11" WHEN b3 ELSE
            "10" WHEN b2 ELSE
            "01" WHEN b1 ELSE
            "00";
    valid <= a(3) OR a(2) OR a(1) OR a(0);
END cond_assign_arch ;
```

Instead of using logic expression, the code describes the circuit using the "priority routing network" specified by the conditional signal assignment statement. The conceptual block diagram to obtain the *code* signal is shown in Figure 5.

Unlike the gate-level circuits, synthesis software is less efficient in performing optimization in module level. The simplicity of the proposed subset allows students to "visualize" the VHDL code in block diagram and to explore more efficient design. Consider a circuit that determines the difference between inputs *a* and *b*; i.e., |*a*-*b*|. A straightforward design is to first compute compare the magnitude of *a* and *b*, and then use the result to route either *a*-*b* or *b*-*a* to the output. The VHDL code is

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY difference IS
   PORT( a, b: IN unsigned(3 DOWNTO 0);
         diff: OUT unsigned(3 DOWNTO 0));
END difference;
ARCHITECTURE design1_arch OF difference IS
    SIGNAL agtb: BOOLEAN;
BEGIN
    agtb <= (a > b);
    diff <= (a - b) WHEN agtb ELSE
            (b - a);
END design1_arch;
```

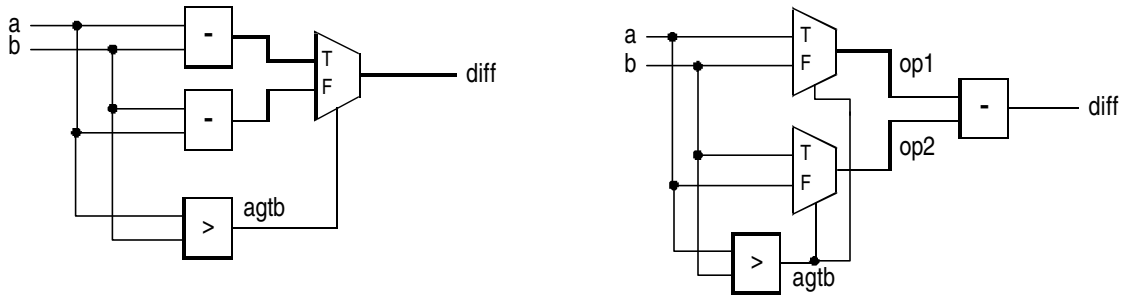


Figure 6. Block diagrams of two difference circuits

The conceptual diagram is shown in the left of Figure 6. Since the subtractor is a much more complex than a multiplexer, the circuit can be simplified by sharing the subtractor. We can first route the inputs *a* and *b* to the proper ports of subtractor and then perform the subtraction. The revised bock diagram is shown in the right of Figure 6 and the VHDL code becomes:

```
ARCHITECTURE design2_arch OF difference IS
    SIGNAL agtb: BOOLEAN;
    SIGNAL op1, op2: unsigned(3 DOWNTO 0);
BEGIN
    agtb <= (a > b);
    op1 <= a WHEN agtb ELSE b;
    op2 <= b WHEN agtb ELSE a;
    diff <= op1 - op2;
END design2_arch;
```

4.3 Regular sequential circuit

Sequential circuits are circuits with "memory". While there are a variety of sequential circuit models, the proposed subset is intended to support the most commonly used one - synchronous sequential circuit, in which all memory elements (FF or register) are controlled by the same clock signal. In this model, the memory elements are separated from other combinational circuits (next-state logic and output logic). The basic diagram is shown in Figure 7. Our VHDL code follows this model and uses the register templates for the memory elements. For the discussion purpose, we divide the sequential circuit into regular sequential circuits, such as counter, shift register etc. and FSM, which has "random" next-state logic.

Let us first examine a simple regular sequential circuit example. Consider a 4-bit free-running binary counter. The block diagram, modeled after the basic sequential circuit diagram of Figure 7, is shown in Figure 8. The next-state logic is an incrementor that increases the current register value by 1. At the rising edge of the clock, this new value is stored into the register and the process repeats. The VHDL code is:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY binary_counter IS
   PORT( clk: IN std_logic;
         count: OUT unsigned(3 DOWNTO 0));
END binary_counter;
```

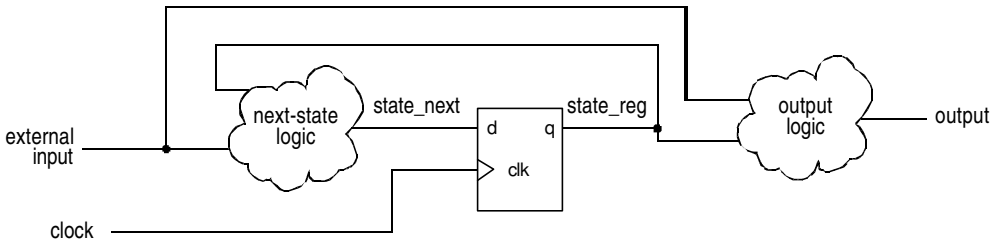


Figure 7. Conceptual diagram of a synchronous sequential circuit

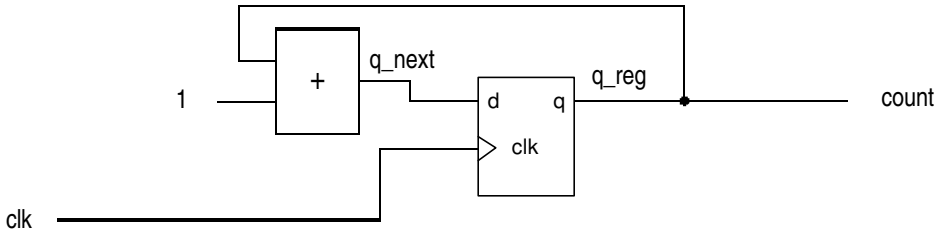


Figure 8. Conceptual diagram of a free-running binary counter

```
ARCHITECTURE arch OF binary_counter IS
    SIGNAL q_next, q_reg: unsigned(3 DOWNTO 0);
BEGIN
    -- register
    PROCESS(clk)
    BEGIN
      IF (clk'EVENT AND clk='1') THEN
         q_reg <= q_next;
      END IF;
    END PROCESS;
    -- next_state logic
    q_next <= q_reg + 1;
    -- output logic
    count <= q_reg;
END arch;
```

Note that the code follows the conceptual diagram. The first segment uses the register template to specify a 4-bit register and the next two segments describe the operation of the next-state logic and output logic (in this case, a wire) respectively.

We can expand the circuit to make it function like a 74163 counter, which has control signals *clr*, *ld* and *en*, for clear, load and enable functions, as well as an additional *rco* output pulse, which is asserted when the counter is in state "1111". The function table of this counter is shown below.

clr	*ld*	*en*	*next q*
1	x	x	*0*
0	1	x	*d*
0	0	1	*q+1*
0	0	0	*q*

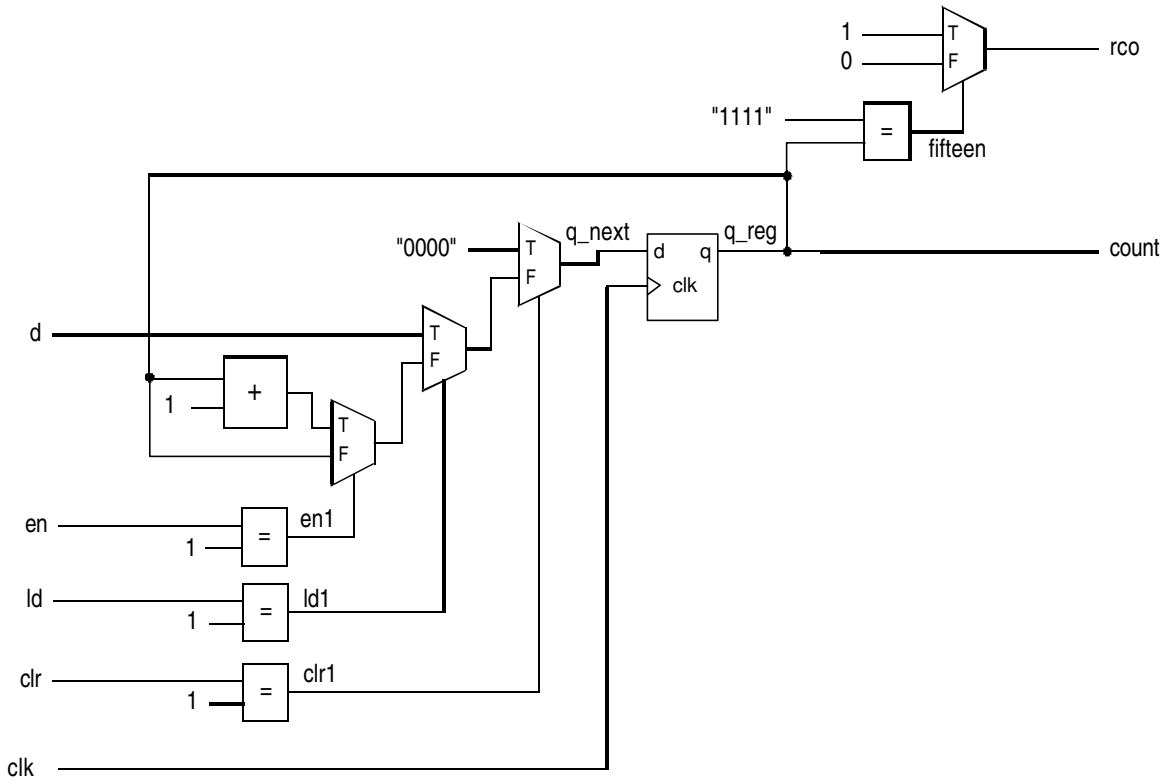
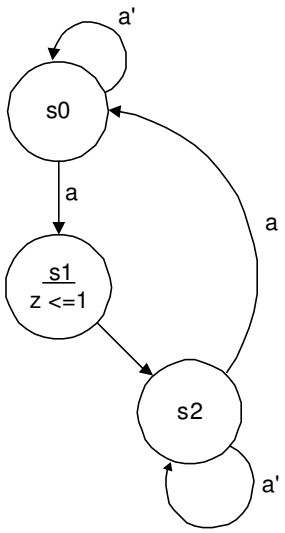


Figure 9. Conceptual diagram of a 74163-like counter

The VHDL code becomes:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY fancy_counter IS
   PORT( clk: IN std_logic;
         clr, ld, en: IN std_logic;
         d: IN unsigned(3 DOWNTO 0);
         count: OUT unsigned(3 DOWNTO 0);
         rco: OUT std_logic);
END fancy_counter;

ARCHITECTURE arch OF fancy_counter IS
   SIGNAL q_next, q_reg: unsigned(3 DOWNTO 0);
   SIGNAL clr1, ld1, en1: BOOLEAN;
   SIGNAL fifteen: BOOLEAN;
BEGIN
   -- register
   PROCESS(clk)
   BEGIN
      IF (clk'EVENT AND clk='1') THEN
         q_reg <= q_next;
      END IF;
   END PROCESS;
   -- next_state logic
   clr1 <= (clr='1');
   ld1 <= (ld='1');
   en1 <= (en='1');
```



current state	*next state*		*output (z)*
	a=0	*a=1*	
s0	s0	s1	0
s1	s2	s2	1
s2	s2	s0	0

State assignment: s0=00, s1=01, s2=10

Figure 10. State diagram and state table of a simple FSM

```
   q_next <= "0000"     WHEN clr1 ELSE
             d          WHEN ld1  ELSE
             q_reg + 1 WHEN en1 ELSE
             q_reg;
    -- output logic
    fifteen <= (q_reg="1111");
    rco <= '1' WHEN fifteen ELSE '0';
    count <= q_reg;
END arch;
```

The code still follows the basic sequential circuit diagram, with segments for register, next-state logic and output logic. The next-state logic is more involved and uses a conditional signal assignment to accommodate the control functions. The complete diagram is shown in Figure 9.

4.4 FSM

The conceptual diagram of an FSM is still modeled after the diagram of Figure 7. However, unlike the regular sequential circuit, the next-state logic does not show a specific pattern and has to be implemented by "random" logic. Let us consider the simple FSM, whose state diagram, state table and state assignment are shown in Figure 10. The VHDL code of this FSM is:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY fsm IS
   PORT( clk: IN std_logic;
         a: IN std_logic;
         z: OUT std_logic);
END fsm;

ARCHITECTURE arch OF fsm IS
   SIGNAL state_next, state_reg: std_logic_vector(1 DOWNTO 0);
   SIGNAL state_input: std_logic_vector(2 DOWNTO 0);
BEGIN
   -- state register
   PROCESS(clk)
   BEGIN
      IF (clk'EVENT AND clk='1') THEN
```

```
        state_reg <= state_next;
      END IF;
   END PROCESS;
   -- next_state logic
   state_input <= state_reg & a;
   WITH state_input SELECT
      state_next <=  "00" WHEN "000",
                     "01" WHEN "001",
                     "10" WHEN "010",
                     "10" WHEN "011",
                     "10" WHEN "100",
                     "00" WHEN OTHERS;
   -- output logic
   z <= '1' WHEN state_reg ="01" ELSE '0';
END arch;
```

The VHDL code fllows the basic diagram of Figure 7, containing segments for register, next-state logic and output logic. The next-state logic utilizes a selected signal assignment statement to obtain the next state, which is patterned after the state table. It will be optimized during synthesis.

5. Conclusion

We introduce a small VHDL subset for the introductory digital systems course. The constructs of the subset can be mapped directly into physical components or routing structures so that the students are "conscious" about hardware and circuits. Despite of its simplicity, the subset can be used to describe most circuits encountered in the courses, including both combination circuits and sequential circuits, either in gate level or in and module level. Using this subset allows the course to focus its original goal of "building block approach" but at the same time let students learn the modern design practice and EDA software tool.

Bibliography

[1] Areibi, S. "A First Course in Digital Design Using VHDL and Programmable Logic," *Proceeding of IEEE Frontiers in Education*, 2001.

[2] Ashenden, P. J., *The Designer's Guide to VHDL, 2nd ed.*, Morgan Kaufmann, 2001.

[3] Brown, S. and Vranesic, Z., *Fundamentals of* Digital *Logic with VHDL Desig*n, McGrawHill, 2000.

[4] Chu, Pong P., "Computation Theory in Digital Systems Course," *Proceeding of IEEE Frontiers in Education*, 2002.

[5] Calazans, N. L. V. and Moras, F. G., "Integrating the Teaching of Computer Organization and Architecture with Digital Hardware Design Early in Undergraduate Course," *IEEE Transaction on Education*, May 2001, pp. 109-119.

[6] Ghosh, S., *Hardware Description Language*, IEEE Press, 2000.

[7] IEEE Computer Society/ACM, *Pre-Iron Man Draft of Computing Curriculum - Computer Engineering*, http://www.eng.auburn.edu/ece/CCCE.

[8] IEEE Computer Society/ACM, *CE-DIG - Digital Logic of Pre-Iron Man Draft of Computing Curriculum - Computer Engineering*, http://www.eng.auburn.edu/ece/CCCE/Digital_Logic.pdf.

[9] Maurer, P. M., "Electrical Design Automation: An Essential Part of a Computer Engineer's Education," *Proceeding of IEEE Frontiers in Education*, 1998.

[10] Palnitkar, S., *Verilog HDL: A Guide to Digital Design and Synthesis*, Prentice Hall, 1996.

[11] Shaout, A. et al., "Enhancing the Digital Systems Courses with Modern Design Tools and Practices," *Proceeding of ASEE Annual Conference*, 2001.

[12] Wakerly, J., *Digital Design: Principles & Practices*, Prentice Hall, 2001.

[13] Zemva, A. et al., "A Rapid Prototyping Environment for Teaching Digital Logic Design," *IEEE Transaction on Education*, Nov. 1998, pp. 342-347.

Acknowledgment

Part of this effort is supported by NSF Grant1#0126752 and by instructional improvement grant from Cleveland State University Center for Teaching and Learning.