Advanced Digital Logic Design – EECS 303

http://ziyang.eecs.northwestern.edu/eecs303/

Teacher:	Robert Dick
Office:	L477 Tech
Email:	dickrp@northwestern.edu
Phone:	847–467–2298

NORTHWESTERN
UNIVERSITY

System design languages

- Software-oriented languages
- Graph-based languages
- Hardware-oriented languages

Software oriented specification languages

- ANSI-C
- SystemC
- Other SW language-based

ANSI-C

Advantages
- Huge code base
- Many experienced programmers
- Efficient means of SW implementation
- Good compilers for many SW processors

Disadvantages
- Little implementation flexibility
  - Strongly SW oriented
  - Makes many assumptions about platform
- Poor support for fine-scale HW synchronization

SystemC

Advantages
- Support from big players
  - Synopsys, Cadence, ARM, Red Hat, Ericsson, Fujitsu, Infineon Technologies AG, Sony Corp., STMicroelectronics, and Texas Instruments
- Familiar for SW engineers

Disadvantages
- Extension of SW language
  - Not designed for HW from the start
- Compiler available for limited number of SW processors
  - New

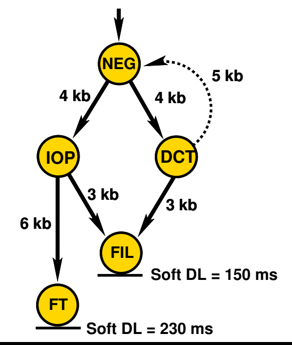
Other SW language-based

- Numerous competitors
- Numerous languages
  - ANSI-C, C++, and Java are most popular starting points
- In the end, few can survive
- SystemC has broad support

Graph-based specification languages

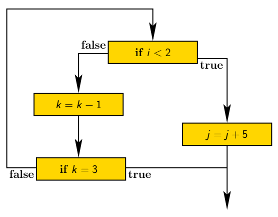
- Dataflow graph (DFG)
- Synchronous dataflow graph (SDFG)
- Control flow graph (CFG)
- Control dataflow graph (CDFG)
- Finite state machine (FSM)
- Petri net
- Periodic vs. aperiodic
- Real-time vs. best effort
- Discrete vs. continuous timing
- Example from research

Dataflow graph (DFG)



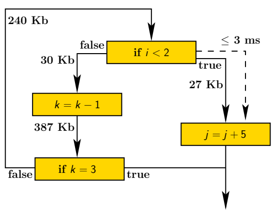
- Nodes are tasks
- Edges are data dependencies
- Edges have communication quantities
- Used for digital signal processing (DSP)
- Often acyclic when real-time
- Can be cyclic when best-effort

Control flow graph (CFG)



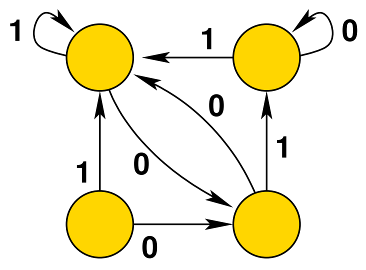
- Nodes are tasks
- Supports conditionals, loops
- No communication quantities
- SW background
- Often cyclic

Control dataflow graph (CDFG)



- Supports conditionals, loops
- Supports communication quantities
- Used by some high-level synthesis algorithms

Finite state machine (FSM)



Finite state machine (FSM)

	input	
	0	1
00	10	00
01	01	00
10	00	01
11	10	00
current	next	

- Normally used at lower levels
- Difficult to represent independent behavior
  - State explosion
- No built-in representation for data flow
  - Extensions have been proposed
- Extensions represent SW, e.g., co-design finite state machines (CFSMs)

Design representations

- VHDL
- Verilog

VHDL

Advantages

- Supports abstract data types
- System-level modeling supported
- Better support for test harness design

Disadvantages

- Requires extensions to easily operate at the gate-level
- Difficult to learn
- Slow to code

Verilog

Advantages

- Easy to learn
- Easy for small designs

Disadvantages

- Not designed to handle large designs
- Not designed for system-level

Verilog vs. VHDL

- March 1995, Synopsys Users Group meeting
- Create a gate netlist for the fastest fully synchronous loadable 9-bit increment-by-3 decrement-by-5 up/down counter that generated even parity, carry, and borrow
- 5 / 9 Verilog users completed
- 0 / 5 VHDL users completed

Does this mean that Verilog is better?
Maybe, but maybe it only means that Verilog is easier to use for simple designs. VHDL has better system-level support.

Active HDL debate

- Synopsys CEO pushes System Verilog
  - No new VHDL project starts
- However, many FPGA designers prefer VHDL
- Many places replacing ASICs with FPGAs
- A lot of controversy recently
  - End result unknown

VHDL

- We'll be introducing VHDL
- This will be helpful for later courses
- This course will only introduce the language
- If you know VHDL and C, learning Verilog will be easy
- Still has better support for system-level design
- Learn VHDL now but realize that you will probably need to know more than one system design language in your career, e.g., System Verilog, SystemC, or both

System-level representations summary

- No single representation has been decided upon
- Software-based representations becoming more popular
- System-level representations will become more important
- Substantial recent changes in the VHDL/Verilog argument

Introduction to VHDL

- This is an overview and introduction only!
  - You may need to use reference material occasionally
- VHDL basics
- Interface
- Architecture body
- Process
- Signal assignment and delay models
- Sequential statements

Modeling

- VHDL designed to model any digital circuit that processes or stores information
- Model represents relevant information, omits irrelevant detail
- Should support
  - Specification of requirements
  - Simulation
  - Formal verification
  - Synthesis

VHDL roots

- Very High Speed Integrated Circuits (VHSIC)
- VHSIC Hardware Description Language (VHDL)
  1. Model digital systems
  2. Simulate the modeled systems
  3. Specify designs to CAD tools for synthesis
- VHDL provides a blackboard for designing digital systems
- An initial design is progressively expanded and refined

Functional and structural specification

- VHDL capable of functional and structural specification
- Functional: What happens
- Structural: How components are connected together
- Supports different levels, from algorithmic to gate

Example functional specification

```
entity XOR2_OP is
-- IO ports
port (
   A, B : in bit;
   Z  :  out bit
);
end XOR2_OP;

-- Body
architecture EX_DISJUNCTION of XOR_OP2 is
begin
   Z <= A xor B;
end EX_DISJUNCTION;
```

VHDL interface and body

A VHDL entity consists of two parts

1. Interface denoted by keyword *entity*
   - Describes external view
2. Body denoted by keyword *architecture*
   - Describes implementation

Interface

```
entity [identifier] is
port ([name]: in/out/inout bit/[type]);
end [identifier];
-- lines beginning with two dashes are comments
```

Body

- Architecture body describes functionality
- Allows for different implementations
- Can have behavioral, structural, or mixed representations

Body

```
architecture [identifier] of
   [interface identifier] is
begin
   [code]
end [identifier];
```

Data types

- The type of a data object defines the set of values that object can assume and set of operations on those values
- VHDL is strongly typed
  - Operands not implicitly converted
- Four classes of objects
  1. Constants
  2. Variables
  3. Signals
  4. Files

Constants

- The value of a constant cannot be changed
- constant [identifier] : [type] (:= expression)
- Examples
  - constant number_of_bytes : integer := 4;
  - constant prop_delay : time := 3ns;
  - constant e : real := 2.2172;

Variable declaration

- The value of a variable can be changed
- variable [identifier] [type] (:= [expression])
- Examples
  - variable index: integer := 0;
  - variable sum, average, largest : real;
  - variable start, finish : time := 0 ns;

Variable assignment

- Once a variable is declared, its value can be modified by an assignment statement
- ([label] :) [name] := [expression];
- Examples
  - program_counter := 0;
  - index := index + 1;
- Variable assignment different from signal assignment
- A variable assignment immediately overrides variable with new value
- A signal assignment schedules new value at later time

Scalar types

- Variable can only assign values of nominated type
- Default types: integer, real, character, boolean, bit
- User defined types: type small_int is range 0 to 255;
- Enumerated types: type logiclevel is (unknown, low, driven, high);

Sub-types

- A type defines a set of values
- Sub-type is a restricted set of values from a base type
    - subtype [identifier] is [name] range [simple expression] to/downto [simple expression]
- Examples
    - subtype small_int is integer range -128 to 127;
    - subtype bit_index is integer range 31 downto 0;

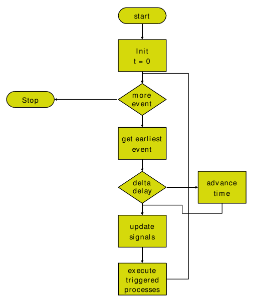
Operators

Operator	Operation	Operand types
**	exponentiation	integer, real
abs	absolute value	numeric
*, /, mod, rem	mult, div, modulus, remainder	integer, real
and, nand, or, nor, xor, xnor, not	logical ops	bit, boolean, or 1-D array
sll, srl, sla,sra	Shift left/right	1-D array of bit/boolean
+, -	add, subtract	integer, real
=, / =, <, <=, >, >=	equal, greater	scalar

VHDL modeling concepts

- Meaning is heavily based on simulation
- A design is described as a set of interconnected modules
- A module could be another design (component) or could be described as a sequential program (process)

VHDL simulator



Process statements

```
[process label]: process
-- declarative part declares functions, procedures,
-- types, constants, variables, etc.
begin
-- Statement part
sequential statement;
sequential statement;
-- E.g., Wait for 1 ms; or wait on ALARM_A;
wait statement;
sequential statement;
...
wait statement;
end process;
```

Sequential statements

- Sequential statements of various types are executed in sequence within each VHDL process
- Variable statement
    - [variable] := [expression];
- Signal Assignment
- If statement
- Case statement
- Loop statement
- Wait statement

Variable and sequential signal assignment

- Variable assignment
    - New values take effect immediately after execution

  variable LOGIC_A, LOGIC_B : BIT;
  LOGIC_A := '1';
  LOGIC_B := LOGIC_A;
- Signal assignment
    - New values take effect after some delay (delta if not specified)

  signal LOGIC_A : BIT;
  LOGIC_A <= '0';
  LOGIC_A <= '0' after 1 sec;
  LOGIC_A <= '0' after 1 sec, '1' after 3.5 sec;

Signal declaration and assignment

- Signal declaration: Describes internal signal
- signal [identifier] : [type] [:= expression]
- Example: signal and_a, and_b : bit;
- Signal Assignment: name <= value_expression [after time_expression];
- Example: y <= not or_a_b after 5 ns;
- This specifies that signal y is to take on a new value at a time 5 ns later statement execution.
- Difference from variable assignment, which only assigns some values to a variable

Inertial delay model

- Reflects inertia of physical systems
- Glitches of very small duration not reflected in outputs
  - Logic gates exhibit low-pass filtering
- SIG_OUT <= not SIG_IN after 7 ns –implicit
- SIG_OUT <= inertial (not SIG_IN after 7 ns)

Transport delay model

- Under this model, ALL input signal changes are reflected at the output
- SIG_OUT <= transport not SIG_IN after 7 ns;

If statement

```
if [boolean expression] then
   [sequential statement]
elsif [boolean expression] then
   [sequential statement]
else
   [sequential statement]
endif;

if sel=0 then
   result <= input_0; -- executed if sel = 0
else
   result <= input_1; -- executed if sel = 1
endif;
```

Case statement

Example of an ALU operation

```
case func is
when pass1 =>
   result := operand1;
when pass2 =>
   result := operand2;
when add =>
   result := operand1 + operand2;
when subtract =>
   result := operand1 - operand2;
end case;
```

While

```
while condition loop
   [sequential statements]
end loop;

while index > 0 loop
   index := index -1;
end loop;
```

For

```
for identifier in range loop
   [sequential statements]
end loop;

for count in 0 to 127 loop
   count_out <= count;
   wait for 5~ns;
end loop;

for i in 1 to 10 loop
   count := count + 1;
end loop;
```

Wait statement

- A wait statement specifies how a process responds to changes in signal values.
  - wait on [signal name]
  - wait until [boolean expression]
  - wait for [time expression]

Wait statement example

```
half_add: process is
begin
   sum <= a xor b after T_pd;
   carry <= a and b after T_pd;
   wait on a, b;
end process;
```

Equivalent process sensitivity list

```
half_add: process (a,b) is
begin
   sum <= a xor b after T_pd;
   carry <= a and b after T_pd;
end process;
```

Clock generator

```
clock_gen: process (clk) is
begin
   if clk = ’0’ then
      clk <= ’1’ after T_pw, ’0’ after 2*T_pw;
   endif;
end process clock_gen;
```

MUX example

```
mux:  process (a, b, sel) is
begin
   case sel is
   when ’0’ =>
      z <= a after prop_delay;
   when ’1’ =>
      z <= b after prop_delay;
end process mux;
```

XOR2 functional example

```
-- Interface
entity XOR2_OP is
-- IO
        port (
                a, b: in bit;
                z: out bit
        );
end XOR2_OP;
```

XOR2 functional example (cont.)

```
-- Body
architecture EX_DISJUNCTION of XOR2_OP is
begin
        z <= a xor b;
end EX_DISJUNCTION;
```

XOR3 structural example

```
entity XOR3_OP is
        port (
                a, b, c: in bit;
                z: out bit
        );
end XOR3_OP;
```

XOR3 structural example (cont.)

```
architecture DISJ_STRUCT of XOR3_OP is
component XOR2_OP
        port (a, b: in bit; z: out bit);
end component;
signal a_int: bit;
begin
x1: XOR2_OP port map (a, b, a_int);
x2: XOR2_OP port map (c, a_int, z);
end DISJ_STRUCT;
```

Test bench for XOR2

```
entity test_bench is
end;

architecture test1 of test_bench is
signal a, b, z : BIT := ’0’;
component XOR2_OP
port (a, b: in BIT; z : out BIT);
end component;
for U1: XOR2_OP use
   entity work.XOR2_OP(EX_DISJUNCTION);
begin
U1: XOR2_OP port map (a, b, z);
```

Test bench for XOR2 (cont.)

```
input_changes: process
begin
a <= ’0’ after 0 ns,
’1’ after 10 ns;
b <= ’0’ after 0 ns,
’1’ after 5 ns,
’0’ after 10 ns,
’1’ after 15 ns;
wait;
end process;
end test1;
```

Test bench for XOR3

```
architecture test2 of test_bench is
signal a, b, c, z : BIT := ’0’;
component XOR3_OP
port (a, b, c: in BIT; z : out BIT);
end component;
for U1: XOR3_OP use
   entity work.XOR3_OP(DISJ_STRUCT);
begin
U1: XOR3_OP port map (a, b, c, z);
```

Test bench for XOR3 (cont.)

```
a_change: process
begin loop
        a <= ’0’;
        wait for 5 ns;
        a <= ’1’;
        wait for 5 ns;
end loop;
end process;
```

Test bench for XOR3 (cont.)

```
b_change: process
begin loop
        b <= ’0’;
        wait for 10 ns;
        b <= ’1’;
        wait for 10 ns;
end loop;
end process;
```

Test bench for XOR3 (cont.)

```
c_change: process
begin loop
        c <= ’0’;
        wait for 20 ns;
        c <= ’1’;
        wait for 20 ns;
end loop;
end process;

end test2;
```

Introduction to VHDL sequential system design

- Fundamental meaning of state variables
- AFSM solution to latch problem
- Use of asynchronous reset
- Multiple output sequence detector
- Multi-output pattern recognizers
- Laboratory four walk-through
- VHDL examples

Fundamental meaning of state variables

- They are not remembering something specific about the inputs
- Every state transition is a function of the current state and input only
- However, multiple cycles of memory are possible because the current state is a function of the state before it
- When designing an FSM, consider the meaning of each state
- Example: Design a recognizer for any sequence that ends with 01 and observed 1101 at any time.

Lab assignment four

- Use VHDL to specify and synthesize a FSM
- Design a pattern recognizer FSM
- Specify it in VHDL
- Simulate it with Mentor Graphics ModelSim
- Synthesize it with Synopsys Design Compiler

Multiple-output sequence detector

- If the last two inputs were 00, *G* is high
- If the last three inputs were 100, *H* is high

Lab example interface

```
entity RECOG is
        port (
                clk, a, reset: in bit;
                h: out bit
        );
end RECOG;
```

Lab example body

```
architecture STATE_MACHINE of RECOG is
        type state_type is (s0, s1, s2, s3);
        signal ps, ns : state_type;
        begin

        STATE: process (reset, clk)
        begin
                if (reset = '1') then
                        ps <= s0;
                elsif (clk'event and clk = '1') then
                        ps <= ns;
                end if;
        end process STATE;
```

Lab example body

```
NEW_STATE: process (ps, a)
begin
    case ps is
        when s0 =>
            case a is
                when '0' => ns <= s1;
                when '1' => ns <= s0;
            end case;

        when s1 =>
            case a is
                when '0' => ns <= s2;
                when '1' => ns <= s0;
            end case;
```

Lab example body

```
        when s2 =>
            case a is
                when '0' => ns <= s2;
                when '1' => ns <= s3;
            end case;

        when s3 =>
            case a is
                when '0' => ns <= s1;
                when '1' => ns <= s0;
            end case;
    end case;
end process NEW_STATE;
```

Lab example body

```
        OUTPUT: process (ps)
        begin
                case ps is
                        when s0 => h <= '0';
                        when s1 => h <= '0';
                        when s2 => h <= '0';
                        when s3 => h <= '1';
                end case;
        end process OUTPUT;
end STATE_MACHINE;
```

Test bench

```
entity test_bench is
end;

architecture test_recog of test_bench is
signal clk, input, reset, output : bit := '0';

component RECOG
port (clk, a, reset: in bit; h : out bit);
end component;

for U1: RECOG use entity work.RECOG(STATE_MACHINE);
begin
U1: RECOG port map (clk, input, reset, output);
```

Test bench

```
CLK_CHANGE: process
begin
        loop
                clk <= '1';
                wait for 5 ns;
                clk <= '0';
                wait for 5 ns;
        end loop;
end process CLK_CHANGE;
```

Test bench

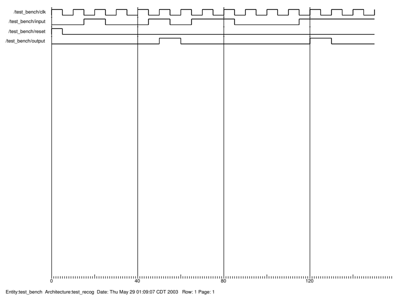
```
RESET_CHANGE: process
begin
        reset <= '1' after 0 ns,
                '0' after 5 ns;
        wait;
end process RESET_CHANGE;
```

Test bench

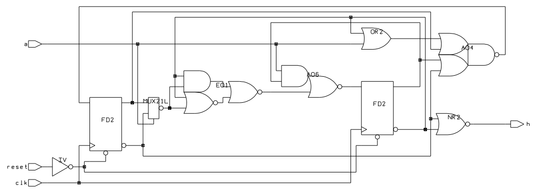
```
INPUT_CHANGE: process
begin
        input <=
                '0' after 5 ns,
                '1' after 15 ns,
                '0' after 25 ns,
                '0' after 35 ns,
                '1' after 45 ns,
                '0' after 55 ns,
                '1' after 65 ns,
                '1' after 75 ns,
                '0' after 85 ns,
                '0' after 95 ns,
                '0' after 105 ns,
                '1' after 115 ns;
        wait;
end process INPUT_CHANGE;

end test_recog;
```

Timing diagram



Optimized implementation



design: RECOG	designer:	date: 5/28/2003
technology: lsi_10k	company:	sheet: 1 of 1

Behavioral and structural specification

- Can either specify behavior or structure of circuit
- May use both styles in a single design
- Also have control over detail of specification
- For example, can keep states abstract and allow synthesis tool to do assignment

Detail of specification

- Could manually specify states
- Could describe entire circuit's connectivity
- Abstract specifications allow synthesis software more freedom
  - Have more potential for automatic optimization
- Detailed specification doesn't rely on as much intelligence in synthesis

VHDL behavioral modeling example

```
architecture primitive of and_or_inv is
signal and_a, and_b, or_a_b : bit;
begin
and_gate_a : process (a1,a2) is
begin
    and_a <= a1 and a2;
end process and_gate_a;
and_gate_b : process (b1,b2) is
begin
    and_b <= b1 and b2;
end process and_gate_b;
```

Behavioral modeling example (cont.)

```
or_gate: process (and_a, and_b) is
begin
    or_a_b <= and_a or and_b;
end process or_gate;
inv : process (or_a_b) is
begin
    y <= not or_a_b;
end process inv;
end architecture primitive;
```

High-level algorithmic specification

```
cpu: process is
   variable instr_reg, PC : word;
 begin
    loop
       address <= PC;
       mem_read <= 1;
      wait until mem_ready = 1;
      PC := PC + 4; -- variable assignment, not a signal;
         --- execute instruction
    end loop;
end process cpu;
```

Memory specification

```
memory: process is
  type memory_array is array (0 to 2**14 - 1) of word;
  variable store: memory_array := ();
begin
     wait until mem_read = 1 or mem_write = 1;
     if mem_read = 1 then
         read_data <= store(address/4);
         mem_ready <= 1;
         wait until mem_ready = 0;
     else
         . --- perform write access;
 end process memory;
```

Example of component instantiation

Structural specification requires connecting components

```
entity DRAM_controller is
port (rd, wr, mem: in bit;
    ras, cas, we, ready: out bit);
end entity DRAM_controller;
```

Example of component instantiation

We can then perform a component instantiation as follows assuming that there is a corresponding architecture called "fpld" for the entity.

```
main_mem_cont : entity work.DRAM_controller(fpld)
port map(rd=>cpu_rd, wr=>cpu_wr,
    mem=>cpu_mem, ready=>cpu_rdy,
    ras=>mem_ras, cas=>mem_cas, we=>mem_we);
```

VHDL synthesis quirks

- Given a statement
  - y <= a + b + c + d;
- Synthesis tool will create a tree of adders by adding a + b, then adding to c, and then to c;
- Instead if specified as
  - y <= (a +b) + (c +d);
- The synthesis tool will be forced to synthesize a tree of depth 2 by adding (a+b), and (c+d) in parallel, then adding results together.

VHDL synthesis quirks

- Another possible mistake
  - y <= a or b or c and d;
- Instead write as
  - y <= (a or b) or (c and d);

Reading assignment

- Zvi Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill Book Company, NY, 1978
- Chapter 11

Next lecture

- More on VHDL
- Introduction to asynchronous FSM design