Introduction to Programmable Logic

LISHA/UFSC

Prof. Dr. Antônio Augusto Fröhlich
Fauze Valério Polpeta
Lucas Francisco Wanner
Danillo Moura Santos
Tiago de Albuquerque Reis
Tiago Rogério Mück

http://www.lisha.ufsc.br/~guto
Why Programmable Logic in a Microprocessors course?

- Microcontrollers are the workhorses of embedded systems world, however:

- Necessary to optimize HW/SW

Peter Marwedel, Univ. Dortmund, Informatik 12, 2006/7
Application-Specific Integrated Circuits (ASIC)

- Silicon chips designed for special Apps.
- ASICs are fabricated on a silicon wafer
  - The transistor and the wires are made from many layers on top of each other
- Interesting when:
  - Max speed needed with Energy efficiency
  - Large scale production
- However
  - Long development time with High NRE Costs
  - Very expensive process
  - Final design is “frozen in silicon”
Application-specific Instruction-set Processor (ASIP)

- The instruction set of an ASIP is tailored to benefit a specific application
- Tradeoff between the flexibility of a general purpose CPU and the performance of an ASIC
- Complex development
- Only feasible when used for multiple applications in the same domain
Programmable Logic Devices (PLD)

- An integrated circuit chip that can be configured by end use to implement different digital hardware
  - Also known as “Field Programmable Logic Device (FPLD)”
Types of PLDs

- **Simple Programmable Logic Device (SPLD)**
  - LSI device with Thousands of Transistors

- **Complex Programmable Logic Device (CPLD)**
  - VLSI device with Higher logic capacity than SPLDs

- **Field Programmable Gate Array (FPGA)**
  - VLSI device, Higher logic capacity than CPLDs
  - Created to allow ASIC prototyping before masks
Basic FPGA Architecture

Programmable Interconnections

Programmable Logic Blocks
Logic Block Architecture
Hardware Description Languages

- “Programming languages” that have been designed and optimized for digital circuit design and modeling
  - VHDL, Verilog, SystemC ...
- Two purposes
  - Writing a model for the expected behavior of a circuit before that circuit is designed and built (Testing)
  - Writing detailed descriptions of circuits that are fed into a computer program called a logic compiler (Synthesis)
Hardware Description Languages

- Digital systems are about signals and their values
- Events, propagation delays, concurrency
  - Signal value changes at specific points in time
Hardware Description Languages

- Timing: computation of events takes place at specific points in time
- Need to “wait for” an event: in this case the clock
- Timing is an attribute of both synchronous and asynchronous systems
Hardware Description Languages

- Example: Asynchronous communication
- No global clock
- Still need to wait for events on specific signals

![Graph showing asynchronous communication](image)
HDL vs Programming Languages

- So, why not use software programming languages for hardware design?
  - HDL's syntax and semantic include explicit notations for expressing time and concurrency

Super-sets of traditional programming languages have been proposed to support timing constraints
HDL Trends

- As occurred with software programming languages, HDLs have been level-increased
HDLs Review

- VHDL
- Verilog
- SystemC
VHDL birth

- Very High Scale Integrated Circuit Hardware Description Language
- Created by US Department of Defense
- Born to document ASIC's
- Similar do ADA
- Evolved to Hardware Synthesis
- In 1887 became a IEEE standard, consolidated in 1993
Basic concepts

- **Entities**
  - Defines the interface

- **Architecture**
  - Describes behavior and structure
  - A entity can have more than one architecture

- **Process**
  - Basic unit of execution

- **Signal**
Entity Declaration

- Describes the input/output ports of a module

```vhdl
entity Counter is
  Generic ( DOWN : boolean := false);
  Port ( clk_in    :in  std_logic;
        rst_in    :in  std_logic;
        counter_out :out std_logic_vector (7 downto 0)
      );
end Counter;
```

- Module name
- Parameters
- I/O ports name
- I/O ports type

Microprocessors Laboratory

June 2007

http://www.lisha.ufsc.br/
Architecture definition

architecture ARCH_NAME of ENTITY is

   Declaration section

begin

   Architecture body

end RTL;
architecture RTL of Counter is

    signal counter : integer range 0 to 255;

begin

    counter_out <= counter;

    counter_proc: process (clk_in)
    begin

        if clk_in'event and clk_in = '1' then
            if rst_in = '1' then
                counter <= 0;
            else
                if DOWN = true then
                    counter <= counter - 1;
                else
                    counter <= counter + 1;
                end if;
            end if;
        else
            if clk_in'event and clk_in = '1' then
                if rst_in = '1' then
                    counter <= 0;
                else
                    if DOWN = true then
                        counter <= counter - 1;
                    else
                        counter <= counter + 1;
                    end if;
                end if;
            end if;
        end if;
    end process;

end RTL;
Modeling Structure

- Implementing the module as a composition of submodules

architecture RTL of Counter_Toplevel is

component Counter is
    Generic ( DOWN : boolean := false);
    Port ( clk_in : in std_logic;
        rst_in : in std_logic;
        counter_out : out std_logic_vector (7 downto 0)
    );
end component Counter;

begin
    counter_instance: component Counter
        generic map ( DOWN => true)
        port map ( clk_in => clk,
                    rst_in => rst,
                    counter_out => leds
        );
end RTL;

Redefines the entity as a component
Creates a instance of the component an defines its connections
Verilog birth

- Created in 1983 by Gateway Design Automation
- Born to design HW for simulation
- C-style syntax
- Evolved to Hardware Synthesis
- In 1995 became a IEEE standard
Verilog Overview

■ Modules
  • Defines the interface, behavior, and structure

■ Process
  • Initial statement
    • Defines behavior executed at system initialization
      (simulation-only)
  • Always statement
    • Defines behavior executed when signals change

■ Wires and registers
Module declaration

module MODULE_NAME ( 

    Declaration of input/outputs 

); 

Module body 

d endmodule
module Counter (
  input wire clk_in,
  input wire rst_in,
  output reg [7:0] counter_out
);

parameter DOWN = 1'b0;

always@(posedge clk_in)
  if (rst_in) begin
    counter_out <= 0;
  end
  else begin
    if (DOWN)
      counter_out <= counter_out - 8'd1;
    else
      counter_out <= counter_out + 8'd1;
  end
endmodule
Modeling Structure

```verilog
module Counter_Toplevel ( 
  ...
);
...
Counter #( .DOWN(0))
counter(
  .clk_in(clk),
  .rst_in(rst),
  .counter_out(leds)
);
...
endmodule
```
VHDL X Verilog

- High-level data types
  - VHDL
    - std_logic
    - integer
    - user-defined types
    - etc
  - Verilog
    - reg
    - wire
Register inference in VHDL

```vhdl
signal C : std_logic;

process (A, B)
  C <= A and B;
end process;

Or

C <= A and B;

process (clk_in)
  if clk_in'event and clk_in = '1' then
    C <= A and B;
  end process;
```

Assignment on clk edge automatically creates a register.
VHDL X Verilog (cont.)

- No register inference in Verilog

```vhdl
wire C;
always@ (A, B)
  C <= A & B;
Or
assign C = A & B;
```

```verilog
reg C;
always@ (posedge clk_in)
  C <= A & B;
Register must be explicitly defined
```

**ERROR**

```
reg C;
assign C = ...
```

```
wire C;
always@ (posedge clk_in)
  C <= ...
```

**ERROR**
VHDL X Verilog (cont.)

- VHDL is strongly typed

**VHDL**

```vhdl
signal A : std_logic_vector (7 downto 0);
signal B : std_logic_vector (3 downto 0);
...
B <= A;          ERROR
B <= A(3 downto 0)  OK
```

**Verilog**

```verilog
wire [7:0] A;
wire [3:0] B;
...
assign B = A;     A is truncated. May generate a warning
```
VHDL X Verilog (cont.)

- Advantages and disadvantages
  - VHDL is less error-prone
  - Verilog have a smaller learning curve
  - VHDL $\rightarrow$ more "verbose" code
  - Verilog $\rightarrow$ more "packed" code
  - Both languages are semantically equivalent
  - 50/50 market share

- It is important to learn both languages!
HDLs Review

- VHDL  OK
- Verilog  OK
- SystemC
SystemC

- Developed by the Open SystemC Initiative (OSCI) and approved as a IEEE Standard in 2005.
- SystemC is a C++ library which provides an event-driven simulation kernel in C++
- Support for System-level and RT-level modeling.
- A synthesizable sub-set is defined, but support differs in each synthesis tool
Basic features

- Modules
  - Basic building block. Similar to Entity/Module of VHDL/Verilog

- Process
  - Implements behavior

- Channels
  - Communication between processes and modules
Module declaration

SC_MODULE (module_name) {

Declaration of I/O channels

SC_CTOR(module_name){

Definition of hierarchy, processes and sensitivity list

};

Processes are implemented as class methods
void process0();
...
void process1();
};
Module declaration (without macros)

```cpp
class module_name : public sc_module {

  // Declaration of I/O channels
  SC_HAS_PROCESS(module_name);
  module_name(sc_module_name nm) : sc_module(nm) {

    // Definition of processes and sensitivity list
  }

  // Processes are implemented as class methods
  void process0();
  ...
  void process1();
};
```
Example

template <bool DOWN = false>
SC_MODULE(Counter) {
    sc_in<bool> clk_in;
    sc_in<bool> rst_in;
    sc_out<sc_uint<8> > counter_out;

    SC_CTOR(Counter){
        SC_METHOD(counter);
        sensitive << clk_in.pos();
    }

    void counter(){
        if (rst_in.read())
            counter_out = 0;
        else {
            if (DOWN)
                counter_out = counter_out.read() - 1;
            else
                counter_out = counter_out.read() + 1;
        }
    }
};

parameters

module name

IO channels

Defines a method as a process

Method implementation

Equivalent to:

counter_out.write(counter_out.read() + 1)
SystemC channels

- **sc_in<...>** and **sc_out<...>**
  - For interfacing with other modules.
  - Equivalents to input/output ports of Verilog/VHDL

- **sc_signal<...>**
  - Communication between processes and module interconnection.
  - Equivalent to VHDL's signal

- Other high-level channels
  - **sc_fifo, sc_buffer, sc_mutex**, etc ...
Modeling Structure

```
SC_MODULE(Counter_Toplevel) {

    ... 

    Counter<false> counter;

    SC_CTOR(Counter_Toplevel) :counter("Counter"){
        ...
        counter.clk_in(clk);
        counter.rst_in(rst);
        counter.counter_out(leds);
        ...
    }

    ...

};
```

- Submodule instantiated as a class attribute
- Initialization
- Connects module's IO channels to another channels
SystemC process and wait()

- Three kinds of processes
  - SC_METHOD
    
    ```
    ...
    SC_METHOD(counter);
    sensitive << clk_in.pos();
    ..
    void counter () {...}
    ```
  
  - Executes every time a signal on the sensitivity list changes
  - `wait` statements are not supported
  - Equivalent to VHDL's `process` and Verilog's `always`
SystemC process and wait()

- **SC_CTHREAD**

  ```
  ...
  SC_CTHREAD(counter, clk_in.pos());
  reset_signal_is(rst_in, true);
  ...
  void counter () {
    //Reset behavior
    wait();
    while(true) {
      //Normal behavior
      wait();
    }
  }
  ```

- Process synchronized by a clock with synchronous reset
  - Process is restarted when reset signal is asserted
  - **wait()** is used to wait for the next cycle and synchronize the process execution
SystemC process and wait()

- **SC_THREAD**

  ```
  ...
  SC_THREAD(counter);
  ..
  void counter () {
    while(true) {
      ...
      wait(clk_in.posedge_event());
    }
  }
  ```

  - Similar to a common software thread
  - `wait(...)` can wait for any kind of event
  - Mostly used for system-level design and testbench implementation
Disadvantages

- Not really appropriated for RTL
- Simple things get complicated in SystemC
  - Example: create a process for a simple combinational assignment

```cpp
signal A : sc_logic;
signal B : sc_logic;
signal C : sc_logic;
...
C <= A and B;
```

```cpp
sc_signal<bool> A;
signal B : sc_logic;
signal C : sc_logic;
...
SC_CTOR(...) {
  ...
  SC_METHOD(and_assignment)
      sensitive << A << B;
  ...
}
...
void and_assignment() {
  C = A.read() && B.read();
}```
Advantages

- System-level modeling
- Everything C++ has to offer to write testbenches
- Object-oriented and other C++ features also available for RT level:
  - Namespaces, classes, inheritance, better communication abstraction through channels, etc.
- Some (expensive) tools support its use for high-level synthesis
Testbenches

- Testing a design by simulation
- Use a *test bench* model
  - an module that includes an instance of the design under test
  - applies sequences of test values to inputs
  - monitors values on output signals
    - either using simulator
    - or with a process that verifies correct operation
Test Bench Waveforms

**Figure 8:** Test Bench Waveform

**Figure 10:** Simulation Results
HDLs study case

- See exercise
FPGA Design Flow

wire C;
always@ (A, B)
    C <= A & B;

[Diagram showing FPGA design flow]
Study case design Flow

- Design entry (VHDL/Verilog)
- RTL simulation (Modelsim)
- Synthesis (Xilinx XST)
- Implementation
  - Translate (Xilinx NGDBUILD)
  - Map (Xilinx MAP)
  - Place and route (Xilinx PAR)
- Design timing and placement constraints (*.ucf)
- Bitstream gen. (Xilinx bitgen)
- Timing and resource information
Synthesis study case

- See exercise
Deployment of HDLs

- HDLs are largely used to implement Intellectual Property Components (Soft IPs)
  - A pre-designed and pre-verified hardware description module written in order to physically instantiate a hardware component

- Interconnecting soft IPs in order to deliver a customized hardware system known as SoC
  - A single chip pastille usually employed in dedicated computing systems to deliver the same functionality that a traditional hardware platform does when connecting several electronic components together on a circuit board
System-on-Chip

System-on-Board (SoB)  System-on-Chip (SoC)

Microprocessors Laboratory

June 2007  http://www.lisha.ufsc.br/
An SoC may contain:
- Portable / reusable IP
- Embedded CPU
- Embedded Memory
- Real World Interfaces
- (USB, PCI, Ethernet)
- Software (both on-chip and off)
- Mixed-signal Blocks
- Programmable HW (FPGAs) > 500K gates

A to Z of SoCs, Reinaldo Bergamaschi, IBM
System-on-Chip Design

- Initially, IP cores were interconnected through custom interface logics
  - The glue logic of the SoC is implicitly defined by the interconnections of the IPs
- The SoC and the IPs become very specifics

- IP reusability decreasing
- Time-consuming
- Labor-intensive
- Error-prone design
- ...
System-on-Chip Design

- More recent approaches are based on standard on-chip buses
  - Avalon from Altera
  - Wishbone from Silicore
  - AMBA from ARM
  - CoreConnect from IBM

- Libraries of pre-designed and pre-verified IP soft cores can be now easily built
  - Users can mix-and-match IPs from libraries to assist in the design of the required system
Sample IP: aeMB

- Open source 32-bit “Soft” Microcontroller
- Implements Xilinx's Microblaze ISA
  - RISC architectures
- Wishbone-compliant
- Support for interrupts exceptions or bus errors
- Optional parameterised multiplier and barrel shifter
- Software implementation of division and floating point
- GCC available
SoC bus example: Wishbone

- Open source on-chip bus
- Created by Silicore Corporation and maintained by Opencores.org
- Defines a "logic bus".
  - Does not specify electrical information or the bus topology.
  - Specification is defined in terms of "signals", clock cycles, and high and low levels.
Wishbone signals
# Basic signals description

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADR_O</td>
<td>Master → Slave</td>
<td>Transaction address</td>
</tr>
<tr>
<td>CYC_O</td>
<td>Master → Slave</td>
<td>High during a valid bus cycle</td>
</tr>
<tr>
<td>STB_O</td>
<td>Master → Slave</td>
<td>High during a bus transaction</td>
</tr>
<tr>
<td>DAT_I</td>
<td>Slave → Master</td>
<td>Data read from the slave</td>
</tr>
<tr>
<td>DAT_O</td>
<td>Master → Slave</td>
<td>Data to be written to the slave</td>
</tr>
<tr>
<td>WE_O</td>
<td>Master → Slave</td>
<td>High indicates a write transaction.</td>
</tr>
<tr>
<td>SEL_O</td>
<td>Master → Slave</td>
<td>Byte selection for writes</td>
</tr>
<tr>
<td>ACK_I</td>
<td>Slave → Master</td>
<td>Indicates that a transaction have been completed</td>
</tr>
</tbody>
</table>
Simple read transaction
Simple write transaction
Study case: Wishbone SoC

- aeMB
- 16 Kb RAM
- Firmware loader
- Timer
- Interrupt Controller
- UART
- GPIO
- Leds
- Buttons
- Switches
- RS232

aeMB

IWB

16 Kb RAM

Firmware loader

DWB

Timer

Interrupt Controller

Spartan3 FPGA

Master

Slave
Study case: Wishbone SoC

![Diagram of Wishbone SoC with Spartan3 FPGA, aeMB, 16 Kb RAM, Firmware loader, Timer, Interrupt Controller, UART, GPIO, Leds, Buttons, and Switches. Connections include IWB, DWB, Master, and Slave.]