Day 1

- 9:00 am  Introduction
- 9:15 am  What is an FPGA
- 9:45 am  FPGA Design Techniques
- 1:00 pm  Lab 1 – Introduction to Altera’s Quartus
- 2:30 pm  Introduction to VHDL
- 3:30 am  Lab 2 - Introduction to VHDL

Day 2

- 9:00 am  Lab 3 - VHDL Testbench Lab
- 10:00 am  Advanced VHDL
- 11:00 am  Lab 4 – Advanced VHDL
- 1:00 pm  Final Project (using DE2)
- 3:00 pm  Support System (Software/hardware)
- 3:30 pm  Implementation / Adaption Plan / Issues at schools
- 4:00 pm  Conclusions / Feedback / Survey
Introductions

- Your Name
- Your Affiliations
- Experience with VHDL and FPGA
- Your intended learning goals
- Pre-test / Pre-Survey

Workshop Goals:

- Participants will:
  - Be able to identify the importance of teaching engineering technology students relevant skills in hardware modeling and FPGA design
  - Demonstrate the understanding of the fundamental concepts of hardware description languages and gain knowledge on programmable logic devices (PLD)
  - Gain hands-on expertise on the hardware and software necessary to establish a re-configurable lab at their respective institutions
  - Gain hands-on lab experience by practicing modeling basic building blocks of digital systems and learn the FPGA design flow
  - Develop potential curricular resources to be used at their respective institutions
Introduction

• FPGA-based re-programmable logic design became more attractive as a design medium during the last decade
• only 19.5% of 4-year and 16.5% of 2-year electrical and computer engineering technology programs at US academic institutions currently have a curriculum component in hardware description language and programmable logic design
• Curriculum has not yet “caught up” to industry needs. Industry must be driving the curriculum development.

Research Background

• Respond to the Market needs of Skilled FPGA Engineers
• Historically, electrical engineering technology Associate and Baccalaureate programs have included a traditional logic design course. These topics are far from most current industry practice in logic design.
• EET two-year and four-year programs must teach digital logic using VHDL and FPGA [3] and students must be equipped with design skills that are current, relevant, and widely used in industry
• The major objectives of this curriculum shift are to give technology students at Michigan Technological University and College of Lake County the opportunity to learn and experience logic design using FPGA
FPGA

- Introduced by Xilinx in mid 1980 for implementing digital logic
  - Field Programmable Gate Array (FPGA)
  - FPGA Can be visualized as a set of programmable logic blocks embedded in programmable interconnect
  - Interconnect architecture provides the connectivity between logic blocks
  - Programming Technology determines the method of storing configuration

FPGA Re-programmable Logic Applications

- When FPGA first introduced, it was considered as another form of gate array

- SRAM-FPGA in-circuit reprogrammability feature provides more than just a standard gate array

- FPGAs have gained rapid acceptance and growth over the past decade because they can be applied to a very wide range of applications
  - random logic
  - Custom computing machine
  - device controllers
  - communication encoding and filtering

- programmable logic becomes the dominant form of digital logic design and implementation
FPGA design flow

- Design Flow is the step-by-step methodology to go through the process of FPGA design
- The design flow can be divided into 6 basic steps
  - Design Entry
  - Functional Verification and Simulation
  - FPGA Synthesis
  - FPGA Place & Route
  - Circuit Analysis (Timing, Power ...)
  - Programming FPGA devices
Description of Design steps

- Design Entry – describes the design that has to be implemented onto FPGA
- Functional Verification and Simulation – checks logical correctness of design
- FPGA synthesis – converts design entry into actual gates/blocks needed
- FPGA Place & Route – selects the optimal position and minimizes length of interconnections on device
- Time Analysis – determines the speed of the circuit which has been completely placed and routed
- Programming to FPGA – downloads bitstream codes onto FPGA devices

What projects are FPGAs good for

- **Aerospace & Defense**
  Radiation-tolerant FPGAs along with intellectual property for image processing, waveform generation, and partial reconfiguration for SDRs.
- **Automotive**
  Automotive silicon and IP solutions for gateway and driver assistance systems, comfort, convenience, and in-vehicle infotainment.
- **Broadcast**
  Solutions enabling a vast array of broadcast chain tasks as video and audio finds its way from the studio to production and transmission and then to the consumer.
- **Consumer**
  Cost-effective solutions enabling next generation, full-featured consumer applications, such as converged handsets, digital flat panel displays, information appliances, home networking, and residential set top boxes.
- **Industrial/Scientific/Medical**
  Industry-compliant solutions addressing market-specific needs and challenges in industrial automation, motor control, and high-end medical imaging.
- **Storage & Server**
  Data processing solutions for Network Attached Storage (NAS), Storage Area Network (SAN), servers, storage appliances, and more.
- **Wireless Communications**
  RF-base band, connectivity, transport and networking solutions for wireless equipment, addressing standards such as WCDMA, HSDPA, WiMAX and others.
- **Wired Communications**
  End-to-end solutions for the Reprogrammable Networking Linecard Packet Processing, Framer/MAC, serial backplanes, and more.
FPGA’s are horrendously complex (much more than computer software)
- Many tools, coding systems, etc.
- Analog (non-ideal) timing behavior
- Many types of devices
- Code synthesis/Place & Route are not deterministic

In short a design frequently
- Is theoretically correct
- Simulates OK
- Fails in a real part
Why are they important

• They have the ability to revolutionize the way that prototyping is done.
• Allows companies to get to market quicker and stay in market longer.

Xilinx

• Largest manufacturer of HW
• Develop hardware and software
• Embedded PowerPC
• University Program
Altera

- Second largest manufacturer
- Develop HW and SW
- University Program

Which is best?

- It depends
  - Time
  - Existing resources
  - Money
  - Level of effort
  - Preference
University Programs

• Major manufacturers want to provide you with the resources to be successful

AUP

• Altera University Program
  • SBROWN@altera.com (Stephen Brown)
  • Very aggressive at supporting schools
  • Pros
    – Good software and hardware
  • Cons
    – have limited educational support material
• Xilinx University Program (XUP)
• xup@xilinx.com
• Only give away what they make or have significant investment in
• Pros
  – Give away hardware and software
• Cons
  – Slow and not a positive donation

Altera DE2 Development Board
What do you really need to start?

- Quartus Software
- DE2 FPGA board

Obtaining, licensing and service contract with Altera

- If you decide to go with Altera, we can help you (with currently available resources)
  - Register with AUP
  - Get software
Quartus Software

• Quratus is the software used to make the project we are going to complete.

• We will cover this package more in the next slides and during the practical exercise.

Quartus® II Software Design Series: Foundation
Objectives

• Create a new Quartus® II project
• Choose supported design entry methods
• Compile a design into an FPGA
• Locate resulting compilation information
• Create design constraints (assignments & settings)
• Manage I/O assignments
• Perform timing analysis & obtain results
Quartus II Software – Two Editions

<table>
<thead>
<tr>
<th>Devices Supported</th>
<th>Subscription Edition</th>
<th>Web Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Features</td>
<td>All 100%</td>
<td>Selected 95%</td>
</tr>
<tr>
<td>Distribution</td>
<td>Internet &amp; DVD</td>
<td>Internet &amp; DVD</td>
</tr>
<tr>
<td>Price</td>
<td>Paid</td>
<td>Free (no license required)</td>
</tr>
</tbody>
</table>

Feature Comparison available on Altera web site

Quartus II Design Software

- Fully-integrated development tool
  - Multiple design entry methods
  - Logic synthesis
  - Place & route
  - Timing & power analysis
  - Simulation (ModelSim-Altera Edition)
  - Device programming
More Features

• MegaWizard® Plug-In Manager & SOPC Builder design tools
• TimeQuest Timing Analyzer
• Incremental compilation feature
• PowerPlay Power Analyzer
• NativeLink® 3rd-party EDA tool integration
• Debugging capabilities - From HDL to device-in-system
• 32 & 64-bit Windows & Linux support
• Multi-processor support
• Node-locked & network licensing options

Welcome to the Quartus II Software!

Getting Started With Quartus® II Software

Start Designing
Designing with Quartus II software requires a project.

Create a New Project
New Project Wizard
Open Existing Project
Open Recent Project

Start Learning
The audio/video interactive tutorial teaches you the basic features of Quartus II software.

Open Interactive Tutorial

Turn on or off in Tools ⇒ Options
To reset views:
1. Tools ⇒ Customize ⇒ Toolbars ⇒ Reset All
2. Restart Quartus II
Provides useful instructions on using the Quartus II software & links to settings. Available sections include:

- New features in current release
- Helpful features and project settings available to designers

Built-In Help System

Web browser-based allows for easy search in page

Shortcut to Altera forums

Expand all topics
Interactive Tutorial

Detachable Windows

• Separate child windows from the Quartus II GUI frame (Window menu ⇒ Detach/Attach Window)

Help menu ⇒ Getting Started Tutorial or from the Getting Started window

Click to detach window
Click again to re-attach
Tasks Window

- Easy access to most Quartus II functions
- Organized into related tasks within three task flows

**Full Design Flow**
Perform all project tasks

**Compilation Flow**
Focus on compilation tasks

**Early Timing Estimate**
Faster compile iterations

**Custom Task Flow**
- Customize the Tasks display
- Add Tcl scripts for quick access

Custom Task Flow
Tcl Console Window

- Enter and execute Tcl commands directly in the GUI

```tcl
# Example Tcl commands
project_close
project_open pipemult
set_global_assignment -name VHDL_FILE mult.vhd
execute_flow -compile
```

- Execute from command-line using Tcl shell
  - `quartus_sh -shell`
- Run complete scripts from **Tools** menu ⇒ **Tcl Scripts**

Typical PLD Design Flow

**Design Specification**
- Design entry/RTL coding
  - Behavioral or structural description of design

**RTL simulation**
- Functional simulation
- Verify logic model & data flow

**Synthesis (Mapping)**
- Translate design into device specific primitives
- Optimization to meet required area & performance constraints
- Quartus II synthesis or 3rd party synthesis tools
- **Result**: Post-synthesis netlist

**Place & route (Fitting)**
- Map primitives to specific locations inside Target technology with reference to area & performance constraints
- Specify routing resources to be used
- Quartus II Fitter
- **Result**: Post-fit netlist
**Typical PLD Design Flow**

**Timing analysis (TimeQuest Timing Analyzer)**
- Verify performance specifications were met
- Static timing analysis

**Gate level simulation (optional)**
- Simulation with timing delays taken into account
- Verify design will work in target technology

**PC board simulation & test**
- Simulate board design
- Program & test device on board
- Use SignalTap® II Logic Analyzer or other on-chip tools for debugging

---

**Quartus II Projects**

- **Description**
  - Collection of related design files & libraries
  - Must have a designated top-level entity
  - Target a single device
  - Store settings in Quartus II Settings File (.QSF)
  - Compiled netlist information stored in `db` folder in project directory

- **Create new projects with New Project Wizard**
  - Can be created using Tcl scripts
New Project Wizard

File menu

Tasks

Tcl: project_new <project_name>

Select working directory

Name of project can be any name; recommend using top-level file name

Top-level entity does not need to be the same name as top-level file name

Add design files

- Graphic
- VHDL
- Verilog
- SystemVerilog
- EDIF
- VQM

Add library paths

- User libraries
- MegaCore®/AMPPSM
- Pre-compiled VHDL packages

Tcl: set_global_assignment –name VHDL_FILE <filename.vhd>
Tcl: set_global_assignment –name USER_LIBRARIES <library_path_name>
Tcl: set_global_assignment –name FAMILY "device family name"
Tcl: set_global_assignment –name DEVICE <part_number>

Choose device family
Filter results
Let Fitter select part or
Choose specific part from list
Show advanced devices

EDA Tool Settings

- Choose EDA tools and file formats
- Settings can be changed or added later
Done!

**Design Entry Methods**

- **Quartus II design entry**
  - Text editor
    - VHDL
    - Verilog or SystemVerilog
  - Schematic editor
    - Block Diagram File
  - State machine editor
    - HDL from state machine file
  - Memory editor
    - HEX
    - MIF
- **3rd-party EDA tools**
  - EDIF 2.0.0
  - Verilog Quartus Mapping (.VQM)
- **Mixing & matching design files allowed**
Creating New Design Files (& Others)

File ⇒ New or in Toolbar

• Quartus II Text Editor features
  – Block commenting
  – Line numbering in HDL text files
  – Bookmarks
  – Syntax coloring
  – Find/replace text
  – Find and highlight matching delimiters
  – Function collapse/expand
  – Create & edit .sdc files for TimeQuest timing analyzer (described later)
  – Preview/editing of full design and construct HDL templates

• Enter text description
  – VHDL (.vhd, .vhdl)
  – Verilog (.v, .vlg, .Verilog, .vh)
  – SystemVerilog (.sv)
Verilog & VHDL

- VHDL - VHSIC hardware description language
  - IEEE Std 1076 (1987 & 1993) supported
  - Partial IEEE Std 1076-2008 support
  - IEEE Std 1076.3 (1997) synthesis packages supported
- Verilog
  - IEEE Std 1364 (1995 & 2001) & 1800 (SystemVerilog) supported
- Use Quartus II integrated synthesis to synthesize
- View supported commands in built-in help

Text Editor Features

- Find/highlight matching delimiters
- Bookmarks (on/off/jump to)
- Insert Template (Edit menu)
- Preview window: edit before inserting & save as user template
- Collapse/expand functions
Quartus II Full Compilation Flow

Design Files

- Analysis & Elaboration
  - Constraints & settings
  - Synthesis
- Fitter
  - Constraints & settings
  - Assembler
  - TimeQuest Timing Analysis
  - EDA Netlist Writer
- Functional Netlist
  - Programming & Configuration files
  - Post-Fit Simulation Files

Functional Simulation

EDA Netlist Writer

Gate-Level Simulation

Processing Options

- Start Compilation (Full)
- Start Analysis & Elaboration
  - Checks syntax & builds hierarchy
  - Performs initial synthesis
- Start Analysis & Synthesis
  - Synthesizes & optimizes code
- Start Fitter
  - Places & routes design
  - Generates output netlists
- Start Assembler
  - Generate programming files
- Analyzers
  - I/O Assignment
  - PowerPlay
  - SSN (Switching Noise)
- Start Design Assistant

 MichiganTech
Compilation Design Flows

- Default “flat” compilation flow
  - Design compiled as a whole
  - Global optimizations performed
- Incremental flow (on by default for new projects)
  - User assigns design partitions
  - Each partition processed separately; results merged
  - Post-synthesis or post-fit netlists for partitions are reused
- Benefits
  - Decrease compilation time
  - Preserve compilation results and timing performance
  - Enable faster timing closure

Status bars indicate compilation progress

Message window displays informational, warning, & error messages

Manually flag selected messages for later review
Each compiler executable generates separate folder

Compilation Report

Netlist Viewers

- RTL Viewer
  - Schematic of design after Analysis and Elaboration
  - Visually check initial HDL before synthesis optimizations
  - Locate synthesized nodes for assigning constraints
  - Debug verification issues

- Technology Map Viewers (Post-Mapping or Post-Fitting)
  - Graphically represents results of mapping (post-synthesis) & fitting
  - Analyze critical timing paths graphically
  - Locate nodes & node names after optimizations (cross-probing)
Note: Must perform elaboration first (e.g. Analysis & Elaboration OR Analysis & Synthesis)

Tools menu ⇒ Netlist Viewers or Tasks window “Compile Design” tasks

Find in hierarchy

Schematic view

Hierachy list

• Represents design using logic blocks & nets
  – I/O pins
  – Registers
  – Muxes
  – Gates (AND, OR, etc.)
  – Operators (adders, multipliers, etc.)

Place pointer over any element in schematic to see details
  • Name
  • Internal resource count
Compilation Summary

- Compilation includes synthesis & fitting
- Compilation Report contains detailed information on compilation results
- Use Quartus II software tools to understand how design was processed
  - RTL Viewer
  - Technology Map Viewers
  - State Machine Viewer
  - Chip Planner
  - Resource Property Editors

Synthesis & Fitting Control

- Controlled using two methods
  - Settings - Project-wide switches
  - Assignments - Individual entity/node controls
- Both accessed in Assignments menu or Tasks window
- Stored in QSF file for project/revision
Settings

- Project-wide switches that affect entire design
- Examples
  - Device selection
  - Synthesis optimization
  - Fitter settings
  - Physical synthesis
  - Design Assistant
- Located in Settings dialog box
  - Assignments menu
  - Set Project and Compiler Settings task in Tasks window
Settings Dialog Box

- Change settings
  - Top-level entity
  - Add/remove files
  - Libraries
  - Compiler settings
  - EDA tool settings
  - Synthesis settings
  - Fitter settings
  - Timing Analysis settings
  - Power analysis settings
  - SSN Analyzer settings

### Tcl: `set_global_assignment -name <assignment_name> <value>`

Compilation Process Setting Examples

- **Smart compilation**
  - Skips entire compiler modules when not required
  - Saves compiler time
  - Uses more disk space
  - Generate version-compatible database

### Tcl: `set_global_assignment -name SMART_RECOMPILE ON`
Physical Synthesis

- Optimize during synthesis or re-synthesize based on Fitter output
  - Makes incremental changes that improve results for a given placement
  - Compensates for routing delays from Fitter
  - Apply globally (Settings) or only to specific design entities (Assignment Editor)

Fitter Settings – Fitter Effort

- Standard Fit
  - Highest effort
  - Longest compile time

- Fast Fit
  - Faster compile
  - Possibly lesser performance

- Auto Fit
  - Compile stops after meeting timing
  - Conserves CPU time
  - Will mimic standard fit for hard-to-fit designs
  - Default for new designs

- One fitting attempt

Tcl: set_global_assignment -name FITTER_EFFORT "<Effort Level>"
I/O Planning

Need

• I/O standards increasing in complexity
• FPGA/CPLD I/O structure increasing in complexity
  – Results in increased pin placement guidelines
• PCB development performed simultaneously with FPGA design
• Pin assignments need to be verified earlier in design cycle
• Designers need easy way to transfer pin assignments into board tools

Creating I/O-Related Assignments

• Pin Planner
• Import from spreadsheet in CSV format
• Type directly into QSF file
• Scripting
• Using synthesis attributes in HDL

Note: Other methods/tools are available in the Quartus II software to make I/O assignments. The above are the most common or recommended.
Pin Planner

- Interactive graphical tool for assigning pins
  - Drag & drop pin assignments
  - Set pin I/O standards
  - Reserve future I/O locations

- Three main sections
  - Package View
  - All Pins list
  - Groups list

Assignments menu ⇒ Pin Planner or “Assign Constraints” folder in Tasks window
Pin Planner Window (2)

- **Package View**
  - Displays graphical representation of chip package
  - Use to make or edit design I/O assignments
  - Use to locate other package pins (power & configuration pins)
- **All Pins list**
  - Displays I/O pins (signals) in design as indicated by filter
  - Use to edit pin settings/properties directly
- **Groups list**
  - Similar to All Pins list except displays only groups & buses
  - Use to make bus and group assignments
  - Use to create new user-defined groups

Assigning Pin Locations Using Pin Planner

Drag & drop single pin; tooltips provide pin information

Drag & drop multiple highlighted pins or buses

Choose one-by-one or pin alignment direction (Pin Planner toolbar or Edit menu)
Assigning Pin Locations Using Pin Planner (2)

- Filter nodes displayed
- Double-click pin or I/O bank to open Properties dialog box
- Drag & drop to I/O bank, VREF block, or device edge

Assigning Pin Locations Using Pin Planner (3)

- Select available locations from list of pins color-coded by I/O bank
Show Fitter Placements

- View I/O locations automatically selected by Fitter

View ⇒ Show or in Toolbar

Top View - Wire Bond Cyclone II - EP2C5F256C6

Back-Annotation

Assignments menu

Green/brown pattern indicates back-annotation
Verifying I/O Assignments

- I/O Assignment Analysis
  - Checks legality of all I/O assignments without full compilation
- Minimal requirements for running
  - I/O declaration
    - HDL port declaration
    - Reserved pin
  - Pin-related assignments
    - I/O standard
    - Current strength
    - Pin location (pin, bank, edge)
    - PCI clamping diode
    - Toggle rate

I/O Rules Checked

- No internal logic
  - Checks I/O locations & constraints with respect to other I/O & I/O banks
  - Each I/O bank supports a single $V_{CCIO}$
- I/O connected to logic
  - Checks I/O locations & constraints with respect to other I/O, I/O banks, & internal resources
  - PLL must be driven by a dedicated clock input pin

Note: When working with design files, synthesize design before running I/O Assignment Analysis
I/O Assignment Analysis Output

Compilation Report (Fitter section)
- Pin-out file
- I/O pin tables
- Output pin loading
- I/O rules checking*

Messages on I/O assignment issues
- Compiler assumptions
- Device & pin migration issues
- I/O bank voltages & standards

*Note: See Appendix for special reports and information generated only for Stratix II, Cyclone III, and newer devices

Please go to Lab1 – Introduction to Altera’s Quartus in the Manual
Introduction to VHDL

- Implement basic constructs of VHDL
- Implement modeling structures of VHDL

VHDL

VHSIC (Very High Speed Integrated Circuit)

Hardware

Description

Language

VHDL
What is VHDL?

• IEEE industry standard hardware description language
• High-level description language for both simulation & synthesis
  – Facilitates portability and productivity as design logic and verification stimulus are both vendor and tool independent

Terminology

• HDL – A hardware description language is a software programming language that is used to model a piece of hardware

• Behavioral modeling - A component is described by its input/output response

• Structural modeling - A component is described by interconnecting lower-level components/ primitives
Register Transfer Level (RTL) - A type of behavioral modeling, for the purpose of synthesis
  - Hardware is implied or inferred
  - Synthesizable

Synthesis - Translating HDL to a circuit and then optimizing the represented circuit

Process – Basic unit of execution in VHDL
  - Process executions are converted to equivalent hardware

Typical digital components per IC
  - 1960s/1970s: 10-1,000
  - 1980s: 1,000-100,000
  - 1990s: Millions
  - 2000s: Billions

1970s
  - IC behavior documented using combination of schematics, diagrams, and natural language (e.g., English)

1980s
  - Documentation was hundreds of pages for large ICs
    - Imprecise
    - Need for better documentation

“The system has four states. When in state Off, the system outputs 0 and stays in state Off until the input becomes 1. In that case, the system enters state On1, followed by On2, and then On3, in which the system outputs 1. The system then returns to state Off.”
HDLs for Simulation

- Hardware description languages (HDLs) – Machine-readable textual languages for describing hardware
  - Schematic + documentation in one
  - HDL description became the precise, concise IC documentation
  - Simulator is a tool that automatically generates outputs values of a hardware module for a given sequence of input values.
  - VHDL originally defined for simulation of ICs
    - First, IC was designed
    - Then, IC was described in HDL for documentation

VHDL

- VHDL: Very High Speed Integrated Circuit Language
  - VHSIC: Very High Speed Integrated Circuit
    - Project of the U.S. Dept. of Defense
  - VHDL defined in 1980s
  - Syntax: like Ada software programming language
    - Ada also a U.S. DoD creation
  - IEEE adopted VHDL standard ("1076") in 1987
- Other HDLs
  - Verilog: Defined in 1980s / C-like syntax / IEEE standard ("1364") in 1995
    - VHDL & Verilog very similar in capabilities, differ mostly in syntax
  - SystemC: Defined in 2000s by several companies / C++ libraries and macro routines / IEEE standard ("1666") in 2005
    - Excels for system-level; cumbersome for logic level
VHDL Versions

<table>
<thead>
<tr>
<th>Version</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>VHDL 1987</td>
<td>Initial version accepted by IEEE</td>
</tr>
<tr>
<td>VHDL 1993</td>
<td>First major update to language; Increased flexibility and added “missing” constructs &amp; operations</td>
</tr>
<tr>
<td>VHDL 2000</td>
<td>Minor update to language</td>
</tr>
<tr>
<td>VHDL 2002</td>
<td>Minor update to language</td>
</tr>
<tr>
<td>VHDL-2008*</td>
<td>Second major update to language; Check tool for support</td>
</tr>
</tbody>
</table>

* Course will cover constructs supported by the Quartus II software version 9.1.

HDLs for Design and Synthesis

- HDLs became increasingly used for designing ICs using top-down design process
  - Design: Converting a higher-level description into a lower-level one
  - Describe circuit in HDL, simulate
    - Physical design tools automatically convert to low-level IC design
  - Describe behavior in HDL, simulate
    - e.g., Describe addition as \( A = B + C \), rather than as circuit of hundreds of logic gates
      - Compact description, designers get function right first
    - Design circuit
      - Manually, or
      - Using synthesis tools, which automatically convert HDL behavior to HDL circuit
  - Simulate circuit, should match
• Use of HDLs for synthesis is growing
  – Circuits are more complex
  – Synthesis tools are maturing
• But HDLs originally defined for simulation
  – General language
  – Many constructs not suitable for synthesis
    • e.g., "wait" statements, pointers, recursive function calls
  – Behavior description may simulate, but not synthesize, or may synthesize to incorrect or inefficient circuit
• Not necessarily synthesis tool's fault!

HDL language
  – General and complex; many uses
  – But use for synthesizing circuits is greatly restricted
    • Synthesis tool understands: sensitivity lists, if statements, ...
    • Synthesis tool may not understand: wait statements, while loops, ..., even if the HDL simulates correctly
      – If the circuit is bad, don't blame the synthesis tool!
    – We will emphasize use of VHDL for design and synthesis
Behavioral Modeling

IF shift_left THEN
FOR j IN high DOWNTO low LOOP
    shft(j) := shft(j-1);
END LOOP;
output <= shft AFTER 5ns;

Structural Modeling

Higher-level component

Input 1, ..., inputn

Output 1, ..., outputn

Only the functionality of the circuit, no structure

No specific hardware intent

Functionality and structure of the circuit

Call out the specific hardware
Two sets of constructs:
  – Simulation
  – Synthesis & simulation
• The VHDL language is made up of reserved keywords
• The language is, for the most part, not CASE sensitive
• VHDL statements are terminated with a ;
• VHDL is white space insensitive
• Comment support
  – -- : End of line (EOL) comment; everything from symbol to EOL is commented
  – /* */ : Delimited comment; everything between symbols is commented
    • Supported in VHDL-2008 only

• ENTITY
  – used to define external view of a model. i.e. symbol
• ARCHITECTURE
  – used to define the function of the model. i.e. schematic
• PACKAGE
  – Collection of information that can be referenced by VHDL models. i.e. LIBRARY
  – Consists of two parts: PACKAGE declaration and PACKAGE body
ENTITY <entity_name> IS
Port Declarations
END ENTITY <entity_name> ; (1076-1993 version)

- Analogy: symbol
  - <entity_name> can be any alpha/numerical name
- Port declarations
  - Used to describe the inputs and outputs i.e. pins
- Generic declarations
  - Used to pass information into a model
- Close entity in one of 3 ways
  - END ENTITY <entity_name>; -- VHDL '93 and later
  - END ENTITY; -- VHDL '93 and later
  - END; -- All VHDL versions

ENTITY : Port Declarations

ENTITY <entity_name> IS
PORT (  
  SIGNAL clk : IN bit;  
  --Note: SIGNAL is assumed and is not required  
  q : OUT bit  
);  
END ENTITY <entity_name> ;

- Structure: <class> object_name : <mode> <type> ;
  - <class>: what can be done to an object
  - object_name: identifier (name) used to refer to object
  - <mode>: directional
    - IN (input)  OUT (output)
    - INOUT (bidirectional) BUFFER (output w/ internal feedback)
  - <Type>: what can be contained in the object (discussed later)
ARCHITECTURE

- Analogy: schematic
  - Describes the functionality and timing of a model
- Must be associated with an ENTITY
- ENTITY can have multiple architectures
- ARCHITECTURE statements execute concurrently (processes)
- ARCHITECTURE styles
  - Behavioral: how designs operate
    - RTL: designs are described in terms of registers
    - Functional: no timing
  - Structural: netlist
    - Gate/component level
  - Hybrid: mixture of the two styles
- End architecture with
  - END ARCHITECTURE <architecture_name>; -- VHDL '93 & later
  - END ARCHITECTURE; -- VHDL '93 & later
  - END; -- All VHDL versions

ARCHITECTURE <identifier> OF <entity_identifier> IS

-- ARCHITECTURE declaration section (list does not include all)
  SIGNAL temp: INTEGER := 1; -- signal declarations with optional default values
  CONSTANT load: boolean := true; -- constant declarations
  -- Type declarations (discussed later)
  -- Component declarations (discussed later)
  -- Subprogram declarations (discussed later)
  -- Subprogram body (discussed later)
  -- Subtype declarations
  -- Attribute declarations
  -- Attribute specifications

BEGIN
  PROCESS statements
  Concurrent procedural calls
  Concurrent signal assignment
  Component instantiation statements
  Generate statements

END ARCHITECTURE <architecture_identifier> ;
**ENTITY** entity_name **IS**
   port declarations
**END ENTITY** entity_name;

**ARCHITECTURE** arch_name **OF** entity_name **IS**
   internal signal declarations
   enumerated data type declarations
   component declarations
**BEGIN**
   signal assignment statements
   PROCESS statements
   component instantiations
**END ARCHITECTURE** arch_name;

**ENTITY** cmpl_sig **IS**
   PORT (
      a, b, sel : IN BIT;
      x, y, z : OUT BIT
   );
**END ENTITY** cmpl_sig;
**ARCHITECTURE** logic **OF** cmpl_sig **IS**
**BEGIN**
   -- simple signal assignment
   x <= (a AND NOT sel) OR (b AND sel);
   -- conditional signal assignment
   y <= a WHEN sel='0' ELSE b;
   -- selected signal assignment
   WITH sel SELECT
      z <= a WHEN '0', b WHEN '1', '0' WHEN OTHERS;
**END ARCHITECTURE** logic;
• A **LIBRARY** is a directory that contains a package or a collection of packages

• Two types of libraries
  - Working library
    - Current project directory
  - Resource libraries
    - **STANDARD** package
    - IEEE developed packages
    - Altera component packages
    - Any **LIBRARY** of design units that is referenced in a design

---

**Example**

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY cmpl_sig IS
  PORT ( a, b, sel : IN STD_LOGIC;
         x, y, z : OUT STD_LOGIC);
END ENTITY cmpl_sig;

ARCHITECTURE logic OF cmpl_sig IS
BEGIN
  -- Simple signal assignment
  X <= (a AND NOT sel) OR (b AND sel);
  -- Conditional signal assignment
  Y <= a WHEN sel='0' ELSE B;
  -- Selected signal assignment
  WITH sel SELECT
    Z <= a WHEN '0',
         B WHEN '1',
         '0' WHEN OTHERS;
END ARCHITECTURE logic;
```

• **LIBRARY** `<name>`, `<name>` ;
  - Name is symbolic and defined by compiler tool
  - Note: Remember that WORK and STD do not need to be defined.

• **USE** `lib_name.pack_name.object`;
  - **ALL** is a reserved word for object name

• Placing the library/use clause first will allow all following design units to access it
Types Defined in STANDARD Package

• Type BIT
  — 2 logic value system (‘0’, ‘1’)
    `SIGNAL a_temp : BIT;`
  — Bit_vector array of bits
    `SIGNAL temp : BIT_VECTOR (3 DOWNTO 0);`
    `SIGNAL temp : BIT_VECTOR (0 TO 3);`

• Type BOOLEAN
  — (False, true)

• Type INTEGER
  — Positive and negative values in decimal
    `SIGNAL int_tmp : INTEGER; -- 32-bit number`
    `SIGNAL int_tmp1 : INTEGER RANGE 0 TO 255; -- 8 bit number`

Other Types Defined in Standard Package

• Type NATURAL
  — Integer with range 0 to $2^{32}$

• Type POSITIVE
  — Integer with range 1 to $2^{32}$

• Type CHARACTER
  — ASCII characters

• Type STRING
  — Array of characters

• Type TIME
  — Value includes units of time (e.g. ps, ns, ms, sec, min, hr)

• Type REAL
  — Double-precision floating point numbers
• **LIBRARY IEE**
  
  Contains the following packages
  
  • **STD_LOGIC_1164 (STD_LOGIC types & related functions)**
  • **NUMERIC_STD** (unsigned arithmetic functions using standard logic vectors defined as SIGNED and UNSIGNED data type)
  • **STD_LOGIC_ARITH** (arithmetic functions using standard logic vectors as SIGNED or UNSIGNED)
  • **STD_LOGIC_SIGNED** (signed arithmetic functions directly using standard logic vectors)
  • **STD_LOGIC_UNSIGNED** (unsigned arithmetic functions directly using standard logic vectors)
  • **STD_LOGIC_TEXTIO** (file operations using std_logic)

  * Packages actually developed by Synopsys but accepted and supported as standard VHDL by most synthesis and simulation tools.

---

**Type STD_LOGIC**

- 9 logic value system ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-')
  
  - '1': Logic high
  - '0': Logic low
  - 'X': Unknown
  - 'Z': (not 'z') Tri-state
  - '-': Don't Care
  - 'U': Undefined
  - 'H': Weak logic high
  - 'L': Weak logic low
  - 'W': Weak unknown

  - Resolved type: supports signals with multiple drivers
  - Driving multiple values onto same signal results in known value

**Type STD_ULOGIC**

- Same 9 value system as STD_LOGIC
- Unresolved type: Does not support multiple signal drivers
  
  - Driving multiple values onto same signal results in error
• **ENTITY** – Declares a new type of component
  — Named “And2” in this example
• **PORT( );** – List of inputs and outputs of entity
  — “x: IN std_logic” → port “x” is an *input* port of type *std_logic*
    • *std_logic* short for “standard logic,” a logical *bit* type (versus say an integer type), which can be ‘0’ or ‘1’ (more later)
  — Ports in list separated by semicolons “;”

**Description:**

- **AND/NOT Gates**
- **Entities and Ports**

**Diagram:**

- **ENTITY And2 IS**
  - PORT (x: IN std_logic;
      y: IN std_logic;
      F: OUT std_logic);
   END And2;

- **ENTITY Or2 IS**
  - PORT (x: IN std_logic;
      y: IN std_logic;
      F: OUT std_logic);
   END Or2;

- **ENTITY Inv IS**
  - PORT (x: IN std_logic;
      F: OUT std_logic);
   END Inv;

**VHDL**:

- **VHDL has several dozen reserved words**
  — ENTITY, IS, PORT, IN, OUT, and END are reserved words above
  — Our convention: Reserved words are UPPER CASE
  — User cannot use reserved words when naming items like ports or entities

**User-defined names**

- Begin with letter, optionally followed by sequence of letters, numbers, or underscore
  - No two underscores in a row, and can’t end in underscore
  - Valid names: *A, x, Hello, JXYZ, B14, Sig432, Wire_23, Index_counter_1, In1, test_entity*
  - Invalid names: *IN (reserved word), Wire_ (ends in underscore), Wire__23 (two underscores in a row), 4x1Max (doesn’t start with a letter), __in4 (doesn’t start with a letter)*

**Note:** VHDL is case **insensitive**. ENTITY, Entity, entity, even EnTiTy, all mean the same
• Constants
• Signals
• Signal Assignments
• Operators
• Processes
• Variables
• Sequential Statements
• Subprograms
• Types

Constants

• Associates value to name
• Constant declaration
  – Can be declared in ENTITY, ARCHITECTURE or PACKAGE
    CONSTANT <name> : <DATA_TYPE> := <value>;
  CONSTANT bus_width : INTEGER := 16;
• Cannot be changed by executing code
  – Remember generics are constants (parameters) that can be overwritten by passing new values into the entity at compile time, not during code execution
• Improves code readability
• Increases code flexibility
Signals

• Signals represent physical interconnect (wire) that communicate between processes (functions)

• Signal declaration
  – Can be declared in PACKAGE, ENTITY and ARCHITECTURE

```vhdl
SIGNAL temp : STD_LOGIC_VECTOR (7 DOWNTO 0);
```

Assigning Values to Signals

```vhdl
SIGNAL temp : STD_LOGIC_VECTOR (7 DOWNTO 0);
```

• Signal assignments are represented by <=

• Examples
  – All bits
    temp <= "10101010";
    temp <= 'x"aa"'; (1076-1993)
    – VHDL also supports ‘o’ for octal and ‘b’ for binary
  – Bit-slicing
    temp (7 DOWNTO 4) <= "1010";
  – Single bit
    temp(7) <= '1';

• Use double-quotes (" "') to assign multi-bit values and single-quotes (' ') to assign single-bit values
**Simple Signal Assignments**

- **Format:**
  \[ <signal_name> \leq <expression> ; \]

- **Example:**
  \[
  qa \leq r \text{ OR } t ; \\
  qb \leq (qa \text{ AND NOT } (g \text{ XOR } h));
  \]

- Expressions use VHDL operators to describe behavior

---

**Standard VHDL Operators**

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator Name/Symbol</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>NOT AND OR NAND NOR XOR XNOR</td>
<td>Low</td>
</tr>
<tr>
<td>Relational</td>
<td>= /= &lt; &lt;= &gt; &gt;=</td>
<td>Low</td>
</tr>
<tr>
<td>Shifting (1) (2)</td>
<td>SLL SRL SLA SRA ROL ROR</td>
<td>Low</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>+ −</td>
<td>High</td>
</tr>
<tr>
<td>Concatenation</td>
<td>&amp;</td>
<td>High</td>
</tr>
<tr>
<td>Multiplication</td>
<td>* / MOD REM</td>
<td>High</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>** ABS</td>
<td>High</td>
</tr>
</tbody>
</table>

**Notes:**
- **(1)** Not supported in VHDL '87
- **(2)** Supported in NUMERIC_STD package for SIGNED/UNSIGNED data types

**Operators:**
- **NOT**
- **AND**
- **OR**
- **NAND**
- **NOR**
- **XOR**
- **XNOR**
- **=**
- **/=**
- **<**
- **<=**
- **>**
- **>=**
- **SLL**
- **SRL**
- **SLA**
- **SRA**
- **ROL**
- **ROR**
- **+**
- **−**
- **&**
- ****
- **ABS**

**Other Notes:**
- **** = exponentiation
- **abs** = absolute value
The VHDL compiler can understand this operation because an arithmetic operation is defined for the built-in data type `INTEGER`.

```vhdl
ENTITY opr IS
  PORT ( a : IN INTEGER RANGE 0 TO 16;
         b : IN INTEGER RANGE 0 TO 16;
         sum : OUT INTEGER RANGE 0 TO 32 );
END ENTITY opr;
ARCHITECTURE example OF opr IS
BEGIN
  sum <= a + b;
END ARCHITECTURE example;
```

Note: remember the library STD and the package `STANDARD` do not need to be referenced.

Conditional Signal Assignments

**Format:**

```
<signal_name> <= <signal/value> WHEN <condition_1> ELSE
<signal/value> WHEN <condition_2> ELSE
... 
<signal/value> WHEN <condition_n> ELSE <signal/value>;
```

**Example:**

```
q <= a WHEN sela = '1' ELSE b WHEN selb = '1' ELSE c;
```

\*Implied process\*
WITH <expression> SELECT
<signal_name> <= <signal/value> WHEN <condition_1>,
<signal/value> WHEN <condition_2>,
...
<signal/value> WHEN OTHERS;

Example:

WITH sel SELECT
q <= a WHEN “00”,
b WHEN “01”,
c WHEN “10”,
d WHEN OTHERS;

• All possible conditions must be considered
• WHEN OTHERS clause evaluates all other possible conditions that are not specifically stated
What are the values for a STD_LOGIC data type

Answer: {'0','1','X','Z'…}

Therefore, is the WHEN OTHERS clause necessary?

Answer: YES
Process Statements

• Implicit process
  – Types
    • Concurrent signal assignments
    • Component instantiations
  – Process sensitive to all inputs
    • e.g. Read side of signal assignment

• Explicit process
  – PROCESS keyword defines process boundary

Explicit PROCESS Statement

• Process sensitive to explicit (optional) sensitivity list
  – Events (transitions) on signals in sensitivity list trigger process
    • 0→1, X→1, 1→Z, etc
• Declaration section allows declaration of local objects and names
• Process contents consist of sequential statements and simple signal assignments

-- Explicit PROCESS statement
label : PROCESS (sensitivity_list)
  Constant declarations
  Type declarations
  Variable declarations
BEGIN
  Sequential statement #1;
  ...
  Sequential statement #n ;
END PROCESS;
**Execution of PROCESS Statement**

- Process statement is executed infinitely unless broken by a WAIT statement or sensitivity list
- Sensitivity list implies a WAIT statement at the end of the process
  - Due to all processes being executed once at the beginning of model execution
- Process can have multiple WAIT statements
- Process can not have both a sensitivity list and WAIT statement

⇒ Note: Logic synthesis places restrictions on the usage of WAIT statements as well as on the usage of sensitivity lists

```vhdl
PROCESS (a,b)
BEGIN
  Sequential statements
END PROCESS;
```

```vhdl
PROCESS
BEGIN
  Sequential statements
WAIT ON (a,b) ;
END PROCESS;
```

**Multi-Process Architectures**

- An architecture can have multiple process statements
- Each process executes in parallel with other processes
  - Order of process blocks does not matter
- Within a process, the statements are executed sequentially
  - Order of statements within a process does matter

A R C H I T E C T U R E

- Describes the functionality of design
Equivalent Functions??

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY simp_prc IS
PORT (a, b : IN STD_LOGIC;
      y : OUT STD_LOGIC);
END ENTITY simp_prc;

ARCHITECTURE logic OF simp_prc IS
SIGNAL c : STD_LOGIC;
BEGIN
  process1: PROCESS (a, b)
  BEGIN
    c <= a AND b;
    END PROCESS process1;
  process2: PROCESS (c)
  BEGIN
    y <= c;
    END PROCESS process2;
  END ARCHITECTURE logic;

New value of c not available for y until next process execution (requires another simulation cycle or transition on a/b)

YES

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY simp IS
PORT (a, b : IN STD_LOGIC;
      y : OUT STD_LOGIC);
END ENTITY simp;

ARCHITECTURE logic OF simp IS
SIGNAL c : STD_LOGIC;
BEGIN
  c <= a AND b;
  END ARCHITECTURE logic;

Equivalent Functions??

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY simp_prc IS
PORT (a, b : IN STD_LOGIC;
      y : OUT STD_LOGIC);
END ENTITY simp_prc;

ARCHITECTURE logic OF simp_prc IS
SIGNAL c : STD_LOGIC;
BEGIN
  process1: PROCESS (a, b)
  BEGIN
    c <= a AND b;
    END PROCESS process1;
  process2: PROCESS (c)
  BEGIN
    y <= c;
    END PROCESS process2;
  END ARCHITECTURE logic;

New value of c not available for y until next process execution (requires another simulation cycle or transition on a/b)

NO
IF-THEN Statements

- **Format:**
  
  ```
  IF <condition1> THEN 
  {sequence of statement(s)} 
  ELSIF <condition2> THEN 
  {sequence of statement(s)} 
  ... 
  ELSE 
  {sequence of statement(s)} 
  END IF;
  ```

- **Example:**
  
  ```
  PROCESS (sela, selb, a, b, c) 
  BEGIN 
  IF sela='1' THEN 
  q <= a; 
  ELSIF selb='1' THEN 
  q <= b; 
  ELSE 
  q <= c; 
  END IF; 
  END PROCESS;
  ```

Sequential Statements

- Indicate behavior and express order
- Must be used inside explicit processes

- **Sequential statements**
  - IF-THEN statement
  - CASE statement
  - Looping statements
  - WAIT statements
IF-THEN Statements

• Similar to conditional signal assignment

**Implicit Process**

```
q <= a WHEN sel_a = ‘1’ ELSE
   b WHEN sel_b = ‘1’ ELSE
   c;
```

**Explicit Process**

```
PROCESS (sel_a, sel_b, a, b, c)
BEGIN
  IF sel_a = ‘1’ THEN
    q <= a;
  ELSIF sel_b = ‘1’ THEN
    q <= b;
  ELSE
    q <= c;
  END IF;
END PROCESS;
```

---

**CASE Statement**

**Format:**

```
CASE {expression} IS
  WHEN <condition1> =>
    {sequence of statements}
  WHEN <condition2> =>
    {sequence of statements}
  ...
  WHEN OTHERS => -- (optional)
    {sequence of statements}
END CASE;
```

**Example:**

```
PROCESS (sel, a, b, c, d)
BEGIN
  CASE sel IS
    WHEN “00” =>
      q <= a;
    WHEN “01” =>
      q <= b;
    WHEN “10” =>
      q <= c;
    WHEN OTHERS =>
      q <= d;
  END CASE;
END PROCESS;
```
CASE Statement

- Similar to selected signal assignment

### Implicit Process

```vhdl
WITH sel SELECT
  q <= a WHEN "00",
     b WHEN "01",
     c WHEN "10",
     d WHEN OTHERS;
```

### Explicit Process

```vhdl
PROCESS (sel, a, b, c, d)
BEGIN
  CASE sel IS
    WHEN "00" =>
      q <= a;
    WHEN "01" =>
      q <= b;
    WHEN "10" =>
      q <= c;
    WHEN OTHERS =>
      q <= d;
  END CASE;
END PROCESS;
```

---

Sequential LOOPS

- **Infinite loop**
  - Loops forever
- **While loop**
  - Loops until conditional test is false
- **For loop**
  - Loops for certain number of iterations
  - Note: Iteration identifier not required to be previously declared

- Additional loop commands (each requires loop label)
  - NEXT / NEXT WHEN
    - Skips to next loop iteration
  - EXIT / EXIT WHEN
    - Cancels loop execution

### Infinite Loop

```vhdl
[Loop_label]: LOOP
  --Sequential statement
  EXIT loop_label;
END LOOP;
```

### While Loop

```vhdl
WHILE <condition> LOOP
  --Sequential statements
END LOOP;
```

### For Loop

```vhdl
FOR <identifier> IN <range> LOOP
  --Sequential statements
END LOOP;
```
**WAIT Statements**

- Pauses execution of process until WAIT statement is satisfied

**Types**
- **WAIT ON <signal>**
  - Pauses until signal event occurs
    ```
    WAIT ON a, b;
    ```
- **WAIT UNTIL <boolean_expression>**
  - Pauses until boolean expression is true
    ```
    WAIT UNTIL (int < 100);
    ```
- **WAIT FOR <time_expression>**
  - Pauses until time specified by expression has elapsed
    ```
    WAIT FOR 20 ns;
    ```
- Combined WAIT
  ```
  WAIT UNTIL (a = '1') FOR 5 us;
  ```

* Wait statement usage limited in synthesis

---

**Using WAIT Statements**

```vhdl
stim: PROCESS
VARIABLE error : BOOLEAN;
BEGIN
    WAIT UNTIL clk = '0';
    a <= (OTHERS => '0');
    b <= (OTHERS => '0');
    WAIT FOR 40 NS;
    IF (sum /= 0) THEN
        error := TRUE;
    END IF;
    WAIT UNTIL clk = '0';
    a <= "0010";
    b <= "0011";
    WAIT FOR 40 NS;
    IF (sum /= 5) THEN
        error := TRUE;
    END IF;
    ...
    WAIT;
END PROCESS stim;
```

- Pause execution of the process until clk transitions to a logic 0
- Pause execution of the process until the equivalent of 40 ns passes in simulation time
- Pause execution of the process indefinitely
Two Types of RTL Process Statements

- **Combinatorial process**
  - Sensitive to all inputs read by the process
  - **Example**
    ```vhdl
    PROCESS (a, b, sel)
    PROCESS (ALL) -- VHDL-2008
    ```

- **Sequential process**
  - Sensitive to select inputs (clock and asynchronous control signals)
  - **Example**
    ```vhdl
    PROCESS (clr, clk)
    ```

---

**DFF**

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY dff1 IS
    PORT (d : IN STD_LOGIC;
          clk : IN STD_LOGIC;
          q : OUT STD_LOGIC);
END ENTITY dff1;

ARCHITECTURE logic OF dff1 IS
    BEGIN
        PROCESS (clk)
            BEGIN
                IF clk'EVENT AND clk = '1' THEN
                q <= d;
                END IF;
            END PROCESS;
    END ARCHITECTURE behavior;
```
Types

• VHDL has built-in data types to model hardware (e.g. BIT, BOOLEAN, STD_LOGIC)

• VHDL also allows creation of brand new types for declaring objects (i.e. constants, signals, variables)

• Subtype
• Enumerated Data Type
• Array

Subtype

• A constrained type
• Synthesizable if base type is synthesizable
• Use to make code more readable and flexible
  -- Place in package to use throughout design

ARCHITECTURE logic OF subtype_test IS
  SUBTYPE word IS std_logic_vector (31 DOWNTO 0);
  SIGNAL mem_read, mem_write : word;

  SUBTYPE dec_count IS INTEGER RANGE 0 TO 9;
  SIGNAL ones, tens : dec_count;
BEGIN
Enumerated Data Type

- Allows user to create data type **name** and **values**
  - Must create constant, signal or variable of that type to use
- Used in
  - Making code more readable
  - Finite state machines
- Enumerated type declaration

```vhdl
TYPE <your_data_type> IS
  (data type items or values separated by commas);
```

```vhdl
TYPE enum IS (idle, fill, heat_w, wash, drain);
SIGNAL dshwshr_st : enum;
...
drain_led <= '1' WHEN dshwshr_st = drain ELSE '0';
```

Array

- Creates multi-dimensional data type for storing values
  - Must create constant, signal or variable of that type
- Used to create memories and store simulation vectors
- Array type Declaration

```vhdl
TYPE <array_type_name> IS ARRAY (<integer_range>) OF <data_type>;
```

**array depth**

**what can be stored in each array address**
Array Example

ARCHITECTURE logic OF my_memory IS

-- Creates new array data type named mem which has 64
-- address locations each 8 bits wide

TYPE mem IS ARRAY (0 to 63) OF std_logic_vector (7 DOWNTO 0);

-- Creates 2 - 64x8-bit array to use in design
SIGNAL mem_64x8_a, mem_64x8_b : mem;

BEGIN

... mem_64x8_a(12) <= x"AF";
mem_64x8_b(50) <= "11110000";
...

END ARCHITECTURE logic;

Recall - Structural Modeling

• Functionality and structure of the circuit
• Call out the specific hardware, lower-level components
• VHDL hierarchical design requires component declarations and component instantiations

Design Hierarchically - Multiple Design Files

Component Declaration and Instantiation

• Component declaration - used to declare the port types and the data types of the ports for a lower-level design

```
COMPONENT <lower-level_design_name>
    PORT ( 
        <port_name> : <port_type> <data_type>; 
        ... 
        <Port_name> : <port_type> <data_type>
    );
END COMPONENT;
```

• Component instantiation - used to map the ports of a lower-level design to that of the current-level design

```
<Instance_name> : <lower-level_design_name>
    PORT MAP(<lower-level_port_name> => <current_level_port_name>,
              ..., <lower-level_port_name> => <current_level_port_name>);
```
Hierarchical Circuits Using Entities as Components

- Entity can be used as component in a new entity
  - AND entity used as component in compCircuit entity
  - Can continue: CompCircuit entity can be used as component in another entity
    - And so on
- Hierarchy powerful mechanism for managing complexity

4-bit 2x1 mux example

2x1 mux circuit from earlier

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Mux2 IS
  PORT (i1, i0: IN std_logic;
        s0: IN std_logic;
        d: OUT std_logic);
END Mux2;

ARCHITECTURE Struct OF Mux2 IS
  COMPONENT And2 IS
    PORT (x, y: IN std_logic;
          F: OUT std_logic);
  END COMPONENT;

  COMPONENT Or2 IS
    PORT (x, y: IN std_logic;
          F: OUT std_logic);
  END COMPONENT;

  COMPONENT Inv IS
    PORT (x: IN std_logic;
          F: OUT std_logic);
  END COMPONENT;

  SIGNAL n1, n2, n3: std_logic;

  Inv_1: Inv PORT MAP (s0, n1);
  And2_1: And2 PORT MAP (i0, n1, n2);
  And2_2: And2 PORT MAP (i1, s0, n3);
  Or2_1: Or2 PORT MAP (n2, n3, d);
END Struct;
```
Multifunction Register Behavior

Now consider register with control inputs, such as load or shift

- Could describe structurally
  - Four flip-flops, four muxes, and some combinational logic (to convert control inputs to mux select inputs)

- We'll describe behaviorally

<table>
<thead>
<tr>
<th>Ld</th>
<th>Shr</th>
<th>Shl</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Maintain present value</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Shift left</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Shift right</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Parallel load</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Parallel load – ld has priority</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Parallel load – ld has priority</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Parallel load – ld has priority</td>
</tr>
</tbody>
</table>

Compact register operation table, clearly showing priorities

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY MfReg4 IS
    PORT (I: IN std_logic_vector(3 DOWNTO 0);
    Q: OUT std_logic_vector(3 DOWNTO 0);
    Ld, Shr, Shl, Shr_in, Shl_in: IN std_logic;
    Clk, Rst: IN std_logic);
END MfReg4;

ARCHITECTURE Beh OF MfReg4 IS
BEGIN
    PROCESS (Clk)
    BEGIN
        IF (Clk = '1' AND Clk'EVENT) THEN
            IF (Rst = '1') THEN
                R <= "0000";
            ELSIF (Ld = '1') THEN
                R <= I;
            ELSIF (Shr = '1') THEN
                R(3) <= Shr_in; R(2) <= R(3);
                R(1) <= R(2); R(0) <= R(1);
            ELSIF (Shl = '1') THEN
                R(0) <= Shl_in; R(1) <= R(0);
                R(2) <= R(1); R(3) <= R(2);
            END IF;
        END IF;
        Q <= R;
    END PROCESS;
END Beh;

• Use IF statement
  - ELSIF parts ensure correct priority of control inputs
    - Rst has first priority, then Ld, then Shr, and finally Shl

• Shift by assigning each bit
  - Recall that statement order doesn't matter

• Use signal R for storage
  - Can't use port Q because OUT port cannot be read by process

• Use concurrent signal assignment to update Q when R changes
  - Equivalent to:
    process(R)
    begin
      Q <= R;
    end process;

Use IF statement
- ELSIF parts ensure correct priority of control inputs
  - Rst has first priority, then Ld, then Shr, and finally Shl

Shift by assigning each bit
- Recall that statement order doesn't matter

Use signal R for storage
- Can't use port Q because OUT port cannot be read by process

Use concurrent signal assignment to update Q when R changes
- Equivalent to:
process(R)
begin
  Q <= R;
end process;
A testbench is a setup for applying test vectors to test a design. The setup creates an entity called Testbench having no inputs or outputs. The setup instantiates a component representing the entity to be simulated, CompToTest. The setup uses a process that writes to signals $x_s$ and $y_s$, which are connected to the inputs of CompToTest. The process will contain statements that set the signals with the desired test vectors.

HDL testbench
- Entity with no ports
- Declare component to test
- Declare signal for each port
- Instantiate component, map signals to ports
- Set signal values at desired times

AND/OR/NOT Gates Simulation and Testbenches
AND Gate
Simulation and Testbenches

• Process has no sensitivity list
  — Executes immediately

• "WAIT FOR 10 ns;"
  — Tells simulator to suspend this process, move simulation time forward by 10 ns before executing next statement

• "WAIT;" - no time clause
  — Waits forever — so process executes only once, doesn’t repeat

• Note: Process cannot have both a sensitivity list and WAIT statement(s)

Combinational Circuits
Component Instantiations

• Component instantiation statement

  CompToTest: And2 PORT MAP (x_s, y_s, f_s);

  Name of new instance
  Must be distinct

  Type of component
  From earlier component declarations

  Connects (maps) component ports to signals

  Note: order same as in component declaration (positional)

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Testbench IS
END Testbench;

ARCHITECTURE TBarch OF Testbench IS
COMPONENT And2 IS
  PORT (x: IN std_logic;
        y: IN std_logic;
        F: OUT std_logic);
END COMPONENT;

SIGNAL x_s, y_s, F_s: std_logic;
BEGIN
  CompToTest: And2 PORT MAP (x_s, y_s, F_s);
  PROCESS
  BEGIN
    -- Test all possible input combinations
    y_s <= '0'; x_s <= '0'; WAIT FOR 10 ns;
    y_s <= '0'; x_s <= '1'; WAIT FOR 10 ns;
    y_s <= '1'; x_s <= '0'; WAIT FOR 10 ns;
    y_s <= '1'; x_s <= '1'; WAIT;
  END PROCESS;
END TBarch;
• Provide testbench file to simulator
  – Simulator generates waveforms
  – We can then check if behavior looks correct

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Testbench IS
END Testbench;
ARCHITECTURE TBarch OF Testbench IS
  COMPONENT And2 IS
    PORT (x: IN std_logic;
          y: IN std_logic;
          F: OUT std_logic);
  END COMPONENT;
  SIGNAL x_s, y_s, F_s: std_logic;
  BEGIN
    CompToTest: And2 PORT MAP (x_s, y_s, F_s);
    PROCESS
    BEGIN
      -- Test all possible input combinations
      y_s <= '0'; x_s <= '0';
      WAIT FOR 10 ns;
      y_s <= '0'; x_s <= '1';
      WAIT FOR 10 ns;
      y_s <= '1'; x_s <= '0';
      WAIT FOR 10 ns;
      y_s <= '1'; x_s <= '1';
      WAIT;
    END PROCESS;
  END TBarch;
```

Please go to Lab2
Introduction to VHDL
in the Manual
Sylabus

Day 1
- 9:00 am  Introduction
- 9:15 am  What is an FPGA
- 9:45 am  FPGA Design Techniques
- 1:00 pm  Lab 1 – Introduction to Altera’s Quartus
- 2:30 pm  Introduction to VHDL
- 3:30 am  Lab 2 - Introduction to VHDL

Day 2
- 9:00 am  Lab 3 - VHDL Testbench Lab
- 10:00 am  Advanced VHDL
- 11:00 am  Lab 4 – Advanced VHDL
- 1:00 pm  Final Project (using DE2)
- 3:00 pm  Support System (Software/hardware)
- 3:30 pm  Implementation / Adaption Plan / Issues at schools
- 4:00 pm  Conclusions / Feedback / Survey

Please go to Lab3
VHDL Testbench
in the Manual
Advanced VHDL Design Techniques

FSM
Writing synthesizable VHDL
Create state machines and control their encoding
Optimize designs to improve resource usage and performance
• State machine states must be an enumerated data type:

    TYPE state_type IS (idle, tap1, tap2, tap3, tap4);

• Object which stores the value of the current state must be a signal of the user-defined type:

    SIGNAL filter : state_type;

Writing VHDL Code for FSM (cont.)

• To determine next state transition/logic
  – Use a CASE statement inside a sequential process

• Use 1 of 2 methods to determine state machine outputs
  1. Use a combinatorial process with a CASE statement
  2. Use conditional and/or selected signal assignments for each output
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY filter_sm IS
PORT (
    clk, reset, nw : IN STD_LOGIC;
    select : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
    nxt, first : OUT STD_LOGIC
);
END ENTITY filter_sm;

ARCHITECTURE logic OF filter_sm IS

TYPE state_type IS (idle, tap1, tap2, tap3, tap4);
SIGNAL filter : state_type;

BEGIN

FSM VHDL Code - Enumerated Data Type

FSM VHDL Code - Next State Logic
output: PROCESS (filter) BEGIN
nxt <= '0';
first <= '0';
select <= "00";
CASE filter IS
  WHEN idle =>
  WHEN tap1 =>
    first <= '1';
  WHEN tap2 =>
    select <= "01";
  WHEN tap3 =>
    select <= "10";
  WHEN tap4 =>
    select <= "11";
    nxt <= '1';
END CASE;
END PROCESS output;
END ARCHITECTURE logic;

 FSM VHDL Code - Outputs Using CASE

nxt <= '1' WHEN filter=tap4 ELSE '0';
first <= '1' WHEN filter=tap1 ELSE '0';

WITH filter SELECT
  select <= "00" WHEN tap1,
    "01" WHEN tap2,
    "10" WHEN tap3,
    "11" WHEN tap4,
    "00" WHEN OTHERS;

END ARCHITECTURE logic;

 FSM VHDL Code - Outputs Using Signal Assignments

Conditional signal assignments

Selected signal assignments
Simulation vs. Synthesis

- **Simulation**
  - Code executed in the exact way it is written
  - User has flexibility in writing
  - Initialization of logic supported

- **Synthesis**
  - Code is interpreted & hardware created
    - Knowledge of PLD architecture is important
  - Synthesis tools require certain coding to generate correct logic
    - Subset of VHDL language supported
    - Coding style is important for fast & efficient logic
  - Initialization controlled by device
  - Logic implementation can be adjusted to support initialization

- Pre- & post-synthesis logic should operate the same

Writing Synthesizable VHDL

- Synthesizable VHDL Constructs
- Sensitivity lists
- Latches vs. registers
- **IF-THEN-ELSE** structures
- **CASE** statements
- Variables
Some Synthesizable VHDL Constructs

- ENTITY
- ARCHITECTURE
- CONFIGURATION
- PACKAGE
- Concurrent signal assignments
- PROCESS
- SIGNAL
- VARIABLE (non-shared)
- CONSTANT
- IF-ELSE
- CASE
- Loops (fixed iteration)
- Multi-dimensional arrays
- PORT
- GENERIC (constant)
- COMPONENT

- Component & direct instantiation
- GENERATE
- FUNCTION
- PROCEDURE
- ASSERT (constant false)
- WAIT (one per process)
- TYPE
- SUBTYPE

- Synthesis tools may place certain restrictions on supported constructs
- See the online help in Quartus II (or your target synthesis tool) for a complete list

Some Non-Synthesizable VHDL Constructs

- ACCESS
- ASSERT
- DISCONNECT
- FILE
- GROUP
- NEW
- Physical delay types
- PROTECTED
- SHARED VARIABLE
- Signal assignment delays

- These are some of the constructs not supported by Quartus II synthesis
- See the online help in Quartus II (or your target synthesis tool) for a complete list
Two Types of RTL PROCESS Statements

- **Combinatorial PROCESS**
  - Sensitive to all signals used on right-hand side of assignment statements
- **Example**
  - \texttt{PROCESS (a, b, sel)}

- **Sequential PROCESS**
  - Sensitive to a clock and control signals
- **Example**
  - \texttt{PROCESS (clr, clk)}

Sensitivity Lists

- Incomplete sensitivity list in combinatorial PROCESS blocks may result in differences between RTL & gate-level simulations
  - Synthesis tool synthesizes as if sensitivity list complete

\texttt{PROCESS (a, b)}
\[
y \leftarrow a \text{ AND } b \text{ AND } c;
\]

\textbf{Incorrect Way} – the simulated behavior is not that of the synthesized 3-input AND gate

\texttt{PROCESS (a, b, c)}
\[
y \leftarrow a \text{ AND } b \text{ AND } c;
\]

\textbf{Correct way for the intended AND logic}!
Common Pitfall – Missing Inputs from Sensitivity List

- Pitfall – Missing inputs from sensitivity list when describing combinational behavior
  - Results in sequential behavior
  - Wrong 4x1 mux example
    - Has memory
    - No compiler error
      - Just not a mux
    - Missing i3-0 from sensitivity list
    - Recomputes \( d \) if \( s_1 \) or \( s_0 \) changes
    - Fails to recompute \( d \) if i3 (or i2-i0) changes

Reminder
- Combinational behavior: Output value is purely a function of the present input values
- Sequential behavior: Output value is a function of present and past input values, i.e., the system has memory

Latches vs. Registers
- Altera devices have registers in logic elements, not latches
- Latches are implemented using combinatorial logic & can make timing analysis more complicated
  - Look-up table (LUT) devices use LUTs in combinatorial loops
  - Product-term devices use more product-terms
- Recommendations
  - Design with registers (RTL)
  - Watch out for inferred latches
    - Latches inferred on combinatorial outputs when results not specified for set of input conditions
    - Lead to simulation/synthesis mismatches
IF-ELSE Structure

• **IF-ELSE** (like WHEN-ELSE concurrent assignment) structure implies prioritization & dependency
  - Nth clause implies all N-1 previous clauses not true
  - Beware of needlessly “ballooning” logic

Logical Equation

\[
(<\text{cond1} > \bullet A) + (<\text{cond1} > \cdot <\text{cond2} > \bullet B) + (<\text{cond1} > \cdot <\text{cond2} > \cdot \lnot \text{cond3} \bullet C) + \ldots
\]

– Consider restructuring IF statements
  - May flatten the multiplexer and reduce logic


\[
\begin{array}{|c|c|c|}
\hline
\text{IF cond1 THEN} & \text{IF cond2 THEN} & \text{IF cond1 AND cond2 THEN} \\
\hline
\end{array}
\]

• If sequential statements are mutually exclusive, individual IF structures may be more efficient

When Writing IF-ELSE Structures...

• Cover all cases
  - Uncovered cases in combinatorial processes result in latches

• For efficiency, consider
  - Using don’t cares (‘·’ or ‘X’) for final ELSE clause (avoiding unnecessary default conditions)
    - Synthesis tool has freedom to encode don’t cares for maximum optimization
  - Assigning initial values and explicitly covering only those results different from initial values
Unwanted Latches

• Combinatorial processes that do not cover all possible input conditions generate latches

```plaintext
PROCESS (sel, a, b, c)
BEGIN
    IF sel = “001” THEN
        output <= a;
    ELSIF sel = “010” THEN
        output <= b;
    ELSIF sel = “100” THEN
        output <= c;
    END IF;
END PROCESS;
```

Unwanted Latches Removed

• Close all IF-ELSE structures
  – If possible, assign “don’t care’s” to else clause for improved logic optimization

```plaintext
PROCESS (sel, a, b, c)
BEGIN
    IF sel = “001” THEN
        output <= a;
    ELSIF sel = “010” THEN
        output <= b;
    ELSIF sel = “100” THEN
        output <= c;
    ELSE
        output <= (OTHERS => 'X');
    END IF;
END PROCESS;
```
Common Pitfall – Output not Assigned on Every Pass

- Pitfall – Failing to assign every output on every pass through the process for combinational behavior
  - Results in sequential behavior
    - Referred to as inferred latch
    - Wrong 2x4 decoder example
    - Has memory
    - No compiler error
      - Just not a decoder

Referred to as inferred latch
Wrong 2x4 decoder example
Has memory
No compiler error

• Same pitfall often occurs due to not considering all possible input combinations

PROCESS(i1, i0)
BEGIN
  IF (i1='0' AND i0='0') THEN
    d3 <= '0'; d2 <= '0';
    d1 <= '0'; d0 <= '1';
  ELSIF (i1='0' AND i0='1') THEN
    d3 <= '0'; d2 <= '0';
    d1 <= '1'; d0 <= '0';
  ELSIF (i1='1' AND i0='0') THEN
    d3 <= '0'; d2 <= '1';
    d1 <= '0'; d0 <= '0';
  ELSIF (i1='1' AND i0='1') THEN
    d3 <= '1';
    END IF;
  -- Note: missing assignments
  -- to all outputs in last ELSIF
END PROCESS;
END Beh;

Last "ELSE" missing, so not all input combinations are covered (i.e., i1i0=11 not covered) – no update to the outputs
Beware of building unnecessary dependencies
- e.g. Outputs x, y, z are mutually exclusive, IF-ELSIF causes all outputs to be dependant on all tests & creates latches

```
PROCESS (sel, a, b, c)
BEGIN
    IF sel = "010" THEN
        x <= a;
    ELSIF sel = "100" THEN
        y <= b;
    ELSIF sel = "001" THEN
        z <= c;
    ELSE
        x <= '0';
        y <= '0';
        z <= '0';
    END IF;
END PROCESS;
```

Mutually Exclusive IF-ELSE Latches

Mutually Exclusive Latches Removed
Case Statements

- Case statements *usually* synthesize more efficiently when mutual exclusivity exists
- Define outputs for all cases
  - Undefined outputs for any given case generate latches
- VHDL already requires all case conditions be covered
  - Use **WHEN OTHERS** clause to close undefined cases (if any remain)

Case Statement Recommendations

- Initialize all case outputs or ensure outputs assigned in each case
- Assign initialized or default values to don’t cares (X) for further optimization, if logic allows
Unwanted Latches - Case Statements

- Conditions where output is undetermined

```
output: PROCESS (filter)
BEGIN
  CASE filter IS
    WHEN idle =>
      nxt <= '0';
      first <= '0';
    WHEN tap1 =>
      sel <= "00";
      first <= '1';
    WHEN tap2 =>
      sel <= "01";
      first <= '0';
    WHEN tap3 =>
      sel <= "10";
    WHEN tap4 =>
      sel <= "11";
      nxt <= '1';
  END CASE;
END PROCESS output;
```

- Undetermined output conditions implies memory
- Latch generated for ALL 3 outputs

Latches Removed - Case Statements

- Conditions where output is determined

```
output: PROCESS(filter)
BEGIN
  first <= '0';
  nxt <= '0';
  sel <= "00";
  CASE filter IS
    WHEN idle =>
    WHEN tap1 =>
      first <= '1';
      sel <= "01";
    WHEN tap2 =>
      sel <= "10";
    WHEN tap3 =>
      sel <= "11";
      nxt <= '1';
  END CASE;
END PROCESS output;
```

To remove latches & ensure outputs are never undetermined
- Use signal initialization at beginning of case statement (case statement only deals with changes)
- Use don’t cares (‘-’) for WHEN OTHERS clause, if design allows (for better logic optimization)
- Manually set output in each case
Variables

- May synthesize to hardware depending on use
- Advantages vs. signals
  - Variables are a more behavioral construct as they don’t have a direct correlation to hardware (like signals) and may lead to more efficient logic
  - Simulate more efficiently as they require less memory
    - Signals not updated immediately, so simulator must store two values (current and next value) for every changing signal
    - Variables updated immediately, so simulator stores single value
- Disadvantages vs. signals
  - Must be assigned to signal before process ends
    - Do not represent physical hardware unless equated with signal
  - Must be handled with care
    - Requires fully understand assigning values to variables and signals in same process and how dataflow is effected

Variables & Latches (Recommendations)

- Assign an initial value or signal to a variable unless feedback is desired
- If a variable is not assigned an initial value or signal in a combinatorial process, a latch will be generated
  - This could cause your design to not function as intended
ARCHITECTURE logic OF cmb_vari IS
BEGIN
    PROCESS(i0, i1, a)
        VARIABLE val :
            INTEGER RANGE 0 TO 1;
    BEGIN
        IF (a = '0') THEN
            val := val;
        ELSE
            val := val + 1;
        END IF;
        CASE val IS
            WHEN 0 =>
                q <= i0;
            WHEN OTHERS =>
                q <= i1;
        END CASE;
    END PROCESS;
END ARCHITECTURE logic;

ARCHITECTURE logic OF cmb_vari IS
BEGIN
    PROCESS(i0, i1, a)
        VARIABLE val :
            INTEGER RANGE 0 TO 1;
    BEGIN
        val := 0;
        IF (a = '0') THEN
            val := val;
        ELSE
            val := val + 1;
        END IF;
        CASE val IS
            WHEN 0 =>
                q <= i0;
            WHEN OTHERS =>
                q <= i1;
        END CASE;
    END PROCESS;
END ARCHITECTURE logic;
Synthesizable Subprograms

• Make code more readable/reusable
• Two types
  – Functions
    • Synthesize to combinatorial logic
  – Procedures
    • Can synthesize to combinatorial or sequential logic
      – Signal assignments in procedures called from clocked processes generate registers
      – May test for clock edges
      » May not be supported by all synthesis tools
    • Must not contain WAIT statements
• Each call generates a separate block of logic
  – No logic sharing
  – Implement manual resource sharing, if possible (discussed later)

Combinational Loops

• Common cause of instability
• Behavior of loop depends on the relative propagation delays through logic
  – Propagation delays can change
• Simulation tools may not match hardware behavior

PROCESS (clk, clrn) 
BEGIN
  IF clrn = '0' THEN
    q <= 0;
  ELSIF rising_edge (clk) THEN
    q <= d;
  END IF;
END PROCESS;
clrn <= (ctrl1 XOR ctrl2) AND q;
Combinational Loops

• All feedback loops should include registers

Logic

\[ \text{PROCESS (clk, clrn)} \]
\[ \text{BEGIN} \]
\[ \quad \text{IF clrn = '0' THEN} \]
\[ \quad \quad q <= 0; \]
\[ \quad \text{ELSIF rising_edge (clk)} \]
\[ \quad \quad q <= d; \]
\[ \quad \text{END IF;} \]
\[ \text{END PROCESS;} \]

\[ \text{PROCESS (clk)} \]
\[ \text{BEGIN} \]
\[ \quad \text{IF rising_edge (clk) THEN} \]
\[ \quad \quad \text{clrn <= (ctrl1 XOR ctrl2) AND q;} \]
\[ \quad \text{END IF;} \]
\[ \text{END PROCESS;} \]

State Machine Coding

• Enumerated data type is used to define the different states in the state machine
  – Using constants for states may not be recognized as state machine

\[ \text{TYPE state_type IS (idle, fill, heat_w, wash, drain);} \]

• One or two signals assigned to the name of the state-variable:

\[ \text{SIGNAL current_state, next_state : state_type;} \]

• Use CASE statement to do the next-state logic, instead of IF-THEN statement
  – Synthesis tools recognize CASE statements for implementing state machines

• Use CASE or IF-THEN-ELSE for output logic
• Use to verify correct coding of state machine

Tools Menu ⇒ State Machine Viewer

Highlighting State in State Transition Table Highlights Corresponding State in State Flow Diagram

State Flow Diagram

Use Drop-Down to Select State Machine

State Transition/Encoding Table

ENTITY wm IS
PORT (  
clk, reset, door_closed, full : in std_logic;  
heat_demand, done, empty : in std_logic;  
water, spin, heat, pump : out std_logic);  
END ENTITY wm;

ARCHITECTURE behave OF wm IS
  TYPE state_type IS
    (idle, fill, heat_w, wash, drain);  
  SIGNAL current_state, next_state : state_type;
BEGIN
PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    current_state <= idle;
  ELSIF rising_edge(clk) THEN
    current_state <= next_state;
  END IF;
END PROCESS;

PROCESS (current_state, door_closed, full, heat_demand, done, empty)
BEGIN
  next_state <= current_state;
  CASE current_state IS
    WHEN idle =>
      IF door_closed = '1' THEN
        next_state <= fill;
      END IF;
    WHEN fill =>
      IF full = '1' THEN
        next_state <= heat_w;
      END IF;
    WHEN heat_w =>
      spin <= '1';
      heat <= '1';
    WHEN wash =>
      spin <= '1';
      heat <= '1';
    WHEN drain =>
      spin <= '1';
      pump <= '1';
    ELSE
      next_state <= current_state;
  END CASE;
END PROCESS;

PROCESS (current_state)
BEGIN
  water <= '0';
  spin <= '0';
  heat <= '0';
  pump <= '0';
  CASE current_state IS
    WHEN idle =>
    WHEN fill =>
      water <= '1';
    WHEN next_w =>
      spin <= '1';
      heat <= '1';
    WHEN wash =>
      spin <= '1';
    WHEN drain =>
      spin <= '1';
      pump <= '1';
    ELSE
      water <= '0';
      spin <= '0';
      heat <= '0';
      pump <= '0';
  END CASE;
END PROCESS;
Quartus II default encoding styles for Altera devices
- One-hot encoding for look-up table (LUT) devices
  - Architecture features lesser fan-in per cell and an abundance of registers
- Binary (minimal bit) or grey-code encoding for product-term devices
  - Architecture features fewer registers and greater fan-in
Undefined States

- Noise and spurious events in hardware can cause state machines to enter undefined states
- If state machines do not consider undefined states, it can cause mysterious “lock-ups” in hardware
- Good engineering practice is to consider these states
  - Explicitly code for them (manual)
  - Use “safe” synthesis constraint (automatic)

```
TYPE state_type IS
  (idle, fill, heat_w, wash, drain);
SIGNAL current_state, next_state : state_type;
PROCESS (current_state, door_closed, full, heat_demand, done, empty)
BEGIN
  next_state <= current_state;
  CASE current_state IS
    WHEN idle =>
      IF door_closed = '1' THEN next_state <= fill;
      END IF;
    WHEN fill =>
      IF full = '1' THEN next_state <= heat_w;
      END IF;
    WHEN heat_w =>
      IF heat_demand = '0' THEN next_state <= wash;
      END IF;
    WHEN wash =>
      IF heat_demand = '1' THEN next_state <= heat_w;
      ELSIF done = '1' THEN next_state <= drain;
      END IF;
    WHEN drain =>
      IF empty = '1' THEN next_state <= idle;
      END IF;
    WHEN others =>
      next_state <= idle;
  END CASE;
END PROCESS;
```

'Safe' Binary State Machine?

- This code does not consider undefined states
- The “when others” statement only considers other enumerated states
- The states “101”, “110” & “111” are not considered
Creating “Safe” State Machines

- WHEN OTHERS clause does not make state machines “safe”
  - Once state machine is recognized, synthesis tool only accounts for explicitly defined states
  - Exception: Number of states equals power of 2 AND binary/grey encoding enabled
- Safe state machines created using synthesis constraints
  - Quartus II software uses
    - SAFE STATE MACHINE assignment applied project-wide and to individual FSMs
    - VHDL synthesis attribute
  - May increase logic usage

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Assignment Name</th>
<th>Value</th>
<th>Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>current_state</td>
<td>Safe State Machine</td>
<td>On</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

VHDL Logic Optimization & Performance

- Balancing operators
- Resource sharing
- Logic duplication
- Pipelining
Operators

- Synthesis tools replace operators with pre-defined (pre-optimized) blocks of logic
- Designer should control when & how many operators
  - Ex. Dividers
    - Dividers are large blocks of logic
    - Every '/', mod and rem inserts a divider block and leaves it up to synthesis tool to optimize
    - Better resource optimization usually involves cleverly using multipliers or shift operations to do divide

Generating Logic from Operators

IF (sel < 10) THEN
  y <= a + b;
ELSE
  y <= a + 10;
END IF;
Balancing Operators

- Use parenthesis to define logic groupings
  - Increases performance
  - May increase utilization
  - Balances delay from all inputs to output
  - Circuit functionality unchanged

**Unbalanced**

\[ z \leq a \times b \times c \times d \]

**Balanced**

\[ z \leq (a \times b) \times (c \times d) \]

**Balancing Operators: Example**

- \( a, b, c, d \): 4-bit vectors

**Unbalanced**

\[ z \leq a \times b \times c \times d \]

**Balanced**

\[ z \leq (a \times b) \times (c \times d) \]
Resource Sharing

- Reduces number of operators needed
  - Reduces area
- Two types
  - Sharing operators among mutually exclusive functions
  - Sharing common subexpressions
- Synthesis tools can perform automatic resource sharing
  - Feature can be enabled or disabled

process(rst, clk)
variable tmp_q : std_logic_vector(7 DOWNTO 0);
begin
  if rst = '0' then
    tmp_q := (OTHERS => '0');
  elsif rising_edge(clk) then
    if updn = '1' then
      tmp_q := tmp_q + 1;
    else
      tmp_q := tmp_q - 1;
    end if;
  end if;
end process;

Mutually Exclusive Operators

- Up/down counter
- 2 adders are mutually exclusive & can be shared (typically IF-THEN-ELSE with same operator in both choices)
process(rst, clk)
variable tmp_q : std_logic_vector(7 DOWNTO 0);
variable dir : integer range -1 to 1;
begin
  if rst = '0' then
    tmp_q := (OTHERS => '0');
  elsif rising_edge(clk) then
    if updn = '1' then
      dir := 1;
    else
      dir := -1;
    end if;
    tmp_q := tmp_q + dir;
  end if;
  q <= tmp_q;
end process;

Sharing Mutually Exclusive Operators

– Up/down counter
– Only one adder required

How Many Multipliers?

\[ y \leftarrow a \times b \times c \]
\[ z \leftarrow b \times c \times d \]
How Many Multipliers? (Answer)

\[ y \leq a \times b \times c \]
\[ z \leq b \times c \times d \]

4 Multipliers!

How Many Multipliers Again?

\[ y \leq a \times (b \times c) \]
\[ z \leq (b \times c) \times d \]
y <= a * (b * c)  
z <= (b * c) * d

How Many Multipliers Again?  
(Answer)

3 Multipliers!

- This is called sharing common subexpressions
- Some synthesis tools do this automatically, but some don’t!
- Parentheses guide synthesis tools
- If (b*c) is used repeatedly, assign to temporary signal

Please go to Lab 4 Advanced VHDL in the Manual
Please go to Final Project in the Manual

Educational Materials Available

• Project Website:
  http://www.tech.mtu.edu/NSFATE

• Altera University Program website
  http://www.altera.com/education/univ/unv-index.html
What do you really need to start?

- Quratus Software
- DE2 FPGA board

SW/HW donation

- Register with Altera website
- Request DE2 Boards
  
More workshops

We will offer one more free two-day workshop

Time: Summer 2014
Date: TBD
Location: Michigan Tech University

Post Test survey
Thank You

Thanks to
National Science Foundation
Advanced Technological Education