EE497 Senior Design

FPGA based design
Grzegorz Chmaj
Logic design process:
- Functional/ Non-functional requirements
- Mapping into an FPGA
- Hardware description languages
- Physical design
FPGA

Functional requirements

- Inputs
- Outputs
- Logic networks
Non-functional requirements

- **Performance**
  - Clock speed = primary requirement

- **Size**
  - Determines manufacturing cost

- **Power/Energy**
  - Related to battery life
  - Related to the heat
FPGA selection

- The FPGA you choose must meet many requirements:
  - Capacity
  - Pinout/package type
  - Speed
FPGA

Two ways to go:

- Hardware SOPC design
  - Schematic design
  - HDL design
- Soft processor
Two ways to go:

- Hardware SOPC design
  - Schematic design
  - HDL design
- Soft processor
FPGA

Hardware design flow:

1. Start
2. Integrate SOPC Builder System with Quartus II Project
3. Assign Pin Locations, Timing Requirements, and Other Design Constraints
4. Compile Hardware for Target
5. Ready to Download
FPGA

Schematic design:

- Good for small projects
- Could be hard for bigger projects
- Problems with maintenance and extending more complicated designs
- Not suggested for the Senior Design projects
FPGA

HDL languages:

- Suitable for both small and big designs
- Produce correct design in less time
- Variety of languages to choose
- Easy to maintain and extend even big and complicated systems
- Entry problem: one must know the HDL already, or spend the time to learn
HDL languages:

- **VHDL** – Very High Speed Integrated Circuit Hardware Description Language -> VHSIC Hardware Description Language

- **Verilog** – its syntax is somewhat similar to C, and is considered as simpler than VHDL

- Other…
FPGA – VHDL vs Verilog

Compilation:

- **VHDL:**
  - Multiple design-units (entity/architecture pairs), that reside in the same system file, may be separately compiled if so desired.
  - However, it is good design practice to keep each design unit in its own system file in which case separate compilation should not be an issue.

- **Verilog:**
  - The Verilog language is still rooted in its native interpretative mode.
  - Compilation is a means of speeding up simulation, but has not changed the original nature of the language.
  - As a result care must be taken with both the compilation order of code written in a single file and the compilation order of multiple files.
  - Simulation results can change by simply changing the order of compilation.
FPGA – VHDL vs Verilog

Data types:

- **VHDL**
  - A multitude of language or user defined data types can be used. This may mean dedicated conversion functions are needed to convert objects from one type to another.
  - The choice of which data types to use should be considered wisely, especially enumerated (abstract) data types.
  - This will make models easier to write, clearer to read and avoid unnecessary conversion functions that can clutter the code.
  - VHDL may be preferred because it allows a multitude of language or user defined data types to be used.

- **Verilog**
  - Compared to VHDL, Verilog data types are very simple, easy to use and very much geared towards modeling hardware structure as opposed to abstract hardware modeling.
  - Unlike VHDL, all data types used in a Verilog model are defined by the Verilog language and not by the user.
  - There are net data types, for example wire, and a register data type called reg.
  - A model with a signal whose type is one of the net data types has a corresponding electrical wire in the implied modeled circuit.
  - Objects, that is signals, of type reg hold their value over simulation delta cycles and should not be confused with the modeling of a hardware register.
  - Verilog may be preferred because of it's simplicity.
Design reusability:

- **VHDL**: Procedures and functions may be placed in a package so that they are available to any design-unit that wishes to use them.

- **Verilog**: There is no concept of packages in Verilog. Functions and procedures used within a model must be defined in the module. To make functions and procedures generally accessible from different module statements the functions and procedures must be placed in a separate system file and included using the `include` compiler directive.
FPGA – VHDL vs Verilog

Easy to learn:

- **VHDL:**
  - Strongly typed
  - Many ways to model one circuit

- **Verilog:**
  - More intuitive
  - Similar to other languages
Libraries:

- VHDL:
  - Library can contain:
    - Architectures
    - Compiled entities
    - Packages
    - Configurations
  - Easier to maintain larger projects

- Verilog:
  - No concept of libraries
  - Harder to maintain larger projects
FPGA – VHDL vs Verilog

Procedure calls:

- **VHDL**:
  - Available
  - VHDL allows concurrent procedure calls

- **Verilog**:
  - No concept of procedure calls
FPGA – Soft Processor

Soft processor:

- Processor implemented in Verilog, VHDL, or other similar technology
- Uploaded to FPGA
- Many soft processors can be implemented in one FPGA
- Can use additional FPGA resources
- Programmed in C/assembly languages
FPGA – Soft Processor

Altera Nios II processor:
- Is an equivalent of a microcontroller
- Is a **soft processor**
- Includes processor and peripherals
- Uses consist
FPGA – Soft Processor

Nios II design flow:
Altera Nios II processor:
- Memory Management Unit
- Memory protection unit (MPU)
- External Vector Interrupt Controller with up to 32 interrupts per controller
- Advanced exception support
- Separate instruction and data caches (configurable from 512 bytes to 64 KB)
- Access to up to 2 GB of external address space
- Optional tightly-coupled memory for instructions and data
- Up to six-stage pipeline to achieve maximum MIPS* (*Dhrystones 2.1 benchmark) per MHz
- Single-cycle hardware multiply and barrel shifter
- Hardware divide option
- Dynamic branch prediction
- Up to 256 custom instructions and unlimited hardware accelerators
- Configurable JTAG debug module
- Optional JTAG debug module enhancements, including hardware breakpoints, data triggers, and real-time trace
Altera Nios II processor:

- General purpose RISC processor core
  - Full 32-bit instruction set, data path, and address space
  - 32 general-purpose registers
  - 32 external interrupt sources
  - ALU supports:
    - Single-instruction 32x32 multiply and divide producing a 32-bit result (Dedicated instructions for computing 64-bit and 128-bit products of multiplication)
    - Floating-point instructions for single-precision floating-point operations
    - Single Instruction barrel shifter
  - I/O capabilities:
    - Access to on-chip peripherals
    - Interfaces to off-chip memories and peripherals
Altera NiosII

Programming the NiosII:
Assembly program:

```
.include "nios_macros.s"
.global _start
_start:
  movia r2, AVECTOR /* Register r2 is a pointer to vector A */
  movia r3, BVECTOR /* Register r3 is a pointer to vector B */
  movia r4, N
  ldw r4, 0(r4) /* Register r4 is used as the counter for loop iterations */
  add r5, r0, r0 /* Register r5 is used to accumulate the product */
  LOOP:
    ldw r6, 0(r2) /* Load the next element of vector A */
    ldw r7, 0(r3) /* Load the next element of vector B */
    mul r8, r6, r7 /* Compute the product of next pair of elements */
    add r5, r5, r8 /* Add to the sum */
    addi r2, r2, 4 /* Increment the pointer to vector A */
    addi r3, r3, 4 /* Increment the pointer to vector B */
    subi r4, r4, 1 /* Decrement the counter */
    bgt r4, r0, LOOP /* Loop again if not finished */
    stw r5, DOT_PRODUCT(r0) /* Store the result in memory */
STOP:
```

N:
```
.word 6 /* Specify the number of elements */
AVECTOR:
.word 5, 3, -6, 19, 8, 12 /* Specify the elements of vector A */
BVECTOR:
.word 2, 14, -3, 2, -5, 36 /* Specify the elements of vector B */
DOT_PRODUCT:
.skip 4
```

Figure 6. A program that computes the dot product of two vectors.