Embedded Systems Design with Platform FPGAs
Principles & Practices

Ron Sass and Andrew G. Schmidt
http://www.rcs.uncc.edu/~rsass
University of North Carolina at Charlotte
Spring 2011
Topics

- principles of system design and how to assemble systems on Platform FPGAs
- a general inventory of hardware cores
- review of software resources, conventions, and tools for embedded systems
Principles of System Design

- creating embedded *systems* is central to the book and this class
  - designing the hardware
  - finding/compiling system software
  - developing the application software
- 30 years ago → emphasis was hardware
- nowadays → most of the intellectual effort is on *software*
The big concepts covered in this chapter include:

- principles and measures of complex system design
  - what makes for ‘good’ design?
  - tools for improving system design
- Control Flow Graph: basic representation of computation
- Hardware Design: A Framework
- System Software: The Key Components
To start we ask: what is “good” and “bad” design? Designs fall into two broad classes:

- **external criteria**
  - pressing volume up button, yet volume decreases

- **internal criteria**
  - coding used in DVR impact ability to fix/maintain design

Clearly, some of these qualities can be measured quantitatively but many are very subjective.
Design Quality

Terms relates to system performing intended function.

- **correctness**

- **reliability** depends if applied to hardware or software:
  - reliable hardware:
  - reliable software:

- **resilience** whereas reliability focuses on detecting and correcting corruptions, resilience
For example:

- embedded systems used in medical systems must not have any human errors in the design
- usually accomplished by incorporating additional safety interlocks and formally proving the correctness of software-controlled, dangerous components
- with many different interacting components, one would describe all valid states mathematically and then prove that for all possible inputs, the software will never enter an invalid state
For Example:

- formally describing all valid states can be enormously taxing (and itself error prone)
- designers may resort to informal specifications
- informal specifications often unintentionally omit directions for some situations
- i.e. a camera designed to work with USB 1.1, 1.2 and 2.0 plugged into a 3.0 port
- or, an FPGA flying in SPACE is more susceptible to radiation, a more reliable design would be to use Triple Modular Redundancy on critical system hardware and periodically check/update the configuration memory to detect corruption
resilience and robustness are different from reliability

embedded systems interact with the physical world and the physical world is not as orderly as simple discrete zeroes and ones

i.e. actuators in machines wear out over time

a resilient design behaves correctly even when something that is not supposed to happen, happens

i.e. a thermometer connected to an embedded system may be expected to be in an environment that will never exceed 100° Celsius. If, because sensor has become uncalibrated, the sensor begins to report temperatures such as 101, 102, 103, etc. then the system should behave sensibly

a reliable system would try to fix the result coming from the sensor; a resilient system might treat 103 as the same as 100 and continue
Other Design Characteristics

- **verifiability**
- **maintainability**
- **repairability**
Other Design Characteristics

- **evolvability**
- **portability**
- **interoperability**

Terms can be useful during development, discussion, documentation, etc...
Modules and Interfaces

- in order to build complex systems need to first build simple components
- building on these simple components in a bottom-up approach
- or, consider the design from the top and work down (top-down)

To begin:

- a module

- can be hardware, software or a mix of both
- for now easiest to think of as a VHDL component or software subroutine
Two meanings to term *interface*:

- **formal interface**
- **general interface**

- formal interface can be mechanically inspected
- i.e., for two modules to interact, their interfaces must be compatible and some automated process can check the interfaces
- general interface is more like “how a module is intended to be used” and this cannot, in general, be automatically checked
a module can also include a functional description
the description can be:

*implicit*
Implicit Module Example

A module called “Full Adder” we do not need to say any more because the functionality of a full adder is well-known.
The functional description can be informal.

The full adder component will add three bits together, X, Y and a carry in bit. The addition will result in both sum and carry out bits.
The functional description can also be formal.

— Assign the Sum output signal
S <= A xor B xor Ci;
— Assign the Carry Out output signal
Co <= (A and B) or (A and Ci) or (B and Ci);
graphically, a module is very simple to denote
it can be as simple as just a box
or, more specifically, we can give a module a name
need to distinguish between a module versus an instance
instances are shown with the module name underlined
can name instances too in the event multiple exist
two instances of a module system, (a) the default instance format and (b) the id to give the instance a unique name
**Implementations and Instances**

- **implementation** (of a module)

- **instance**

  in software, there is generally a one-to-one relationship between an implementation and instance because the same instance is reused in time.

  in hardware, it is common to use multiple copies of an implementation; each copy is an instance.
Implementations and Instances

four formally defined modules to generate a 4-bit full adder from 1-bit full adders
Abstraction and State

- two major concepts in system design:
  - abstraction
  - state

- apply concepts to modules described earlier
Abstraction

- **abstraction**

- hence, a module is an abstraction of some functionality in a system
- what makes for a good abstraction?
  - interface and description provides some easily understood idea
  - details of implementation are more complex
  - i.e., capturing all salient features of an idea and cuts out low-level details
State

- from a hardware designer’s perspective:
  - consider sequential “state” machines
  - can point to the memory devices (where state is stored)
- from a system designer’s perspective — more complicated
- state can be stored in many different places (devices)
something “has state” or is “stateful” if it has the ability to hold information over time
identifying the state in a module is important in system design
in system design, separate functionalities into modules and abstractions
state in a module is not explicit, designer must identify states a module can be in
must also define how/what can “change” the state
Cohesion and Coupling

- the concepts are abstraction and state
- the measures are cohesion and coupling
- cohesion is a way of measuring the abstraction
  if the details inside of a module come together to implement an easily understood function, then the module is said to have **cohesion**.
- **Coupling** is a measure of how modules of a system are related to one another
  a system’s coupling is judged by the number of and types of dependencies between modules
  explicit dependence exists when one module directly communicates with another
Dependence Example

For example:

- if output of module A is input to module B, then we say that A and B depend on each other.

- in software, if A might invoke a function in module B, then we say A depends on B (but B does not necessarily depend on A).

- the rule for determining dependence is “if a change to module A requires a designer to inspect module B (to see if the change impacts B) then B depends on A.”
dependence is not always explicit

two modules can be dependent in a number of ways

if two modules share state, then there is dependence

for example, if one module uses a large table in a Block RAM to keep track of some number of events and another module will occasionally inspect that table, the latter is dependent of the former

if someone wants to make a change to the format of the table, that change will impact the latter module

this is where explicitly identifying state becomes critical to formation of modules within a system
one way to reduce coupling in a system through encapsulation.

- Encapsulation involves manipulating state and introducing formal interfaces.
- To move state into a module and make it exclusive (not shared).
Coupling Example

(a) original design; (b) modified design with lower coupling
in (a), the two inputs to the module, $x$ and $y$, are summed together in sub-module $A$ and the results are passed to the sub-modules $B$ and $C$

in (b), the summation is duplicated *inside* each of the sub-modules

in the first design, we had two dependencies — sub-module $B$ depends on sub-module $A$ and sub-module $C$ depended on sub-module $A$

in the second design there are no sub-module dependencies, so we have clearly reduced the coupling in the design
in addition to improving design quality we want reusable components

construction designs with the foresight that components or even the whole design will be reused again in the future

what is necessary to create reusable designs?

one indicator is high cohesion and low coupling which leads to reusable design components

must also consider costs in terms of:

- RCR — relative cost of reuse
- RCWR — relative cost of writing for reuse
RCR vs. RCWR

- RCR — you have to read and understand how to use the class hierarchy before you can reuse its code
- RCWR — someone has to put a lot of effort into designing a hierarchy for reuse

For Example:
- writing a C program to copy character data, say 32-Bytes worth of data, we could write our own for-loop to exactly copy the 32-Bytes of data
- could reuse this loop and augment it to support various sizes; however, we are limited to copying characters, or Bytes
- alternatively, there is already a more efficient libraries in C, such as `string` where `strcpy` already exists
- trade-off is learning how to use `strcpy` compared to the time it takes to create your own copying function is RCR
in some cases, it could be easier to generate your own in place learning a potentially complex component
this leads into RCWR (the cost associated with making your custom create component fully reusable)
one way to manage RCWR is with an incremental approach:
- design a specific component for the current design
- if it is needed again, copy-and-generalize it

in VHDL this could be as simple as adding generics to the design
Refactoring is the task of looking at an existing design and rearranging the groupings and hierarchy without changing its functionality.
- test is very IMPORTANT!
- the value of reusable components is clear
- the danger is a refactore component accidentally changes functionality

**regression testing**
Hardware Design

- turn attention towards design
- specifically hardware components available to Platform FPGA designer
- a brief description on the evolution of these components
- followed by a description of:
  - processors
  - memory
  - buses and bridges
- conclude with a description of how to use custom hardware modules
designers rarely want to build embedded systems from scratch
to be productive designer will typically begin with an existing architecture
then remove unneeded components
and add cores to meet the project requirements
one model that is commonly used as a base design is the processor-memory model
the fundamental process-memory model that is commonly used as a base system in Platform FPGA designs
the two bus process-memory model used to support parallel, independent high speed and low speed communication
Processor-Memory Model on FPGAs

- organization is a good starting point (or platform) for FPGA designs
- can utilize hard or soft processor cores,
- on-chip and off-chip memory controllers,
- and peripherals (existing) cores to interact with the system
- quickly can amass a system that resembles modern computer architecture
Why Processor-Memory Model on FPGAs?

- provides an established framework custom designs can be built on top of
- shortened development time using existing infrastructure
- the designer can focus on new/custom cores
- the “hello world” example from Chapter 1 is an example
“Why create a generic base system when we are using FPGAs?”

- in an ideal world, could create custom design from scratch
- in reality projects have deadlines and budgets
- can leverage existing software infrastructure (such as Linux)
- FPGAs being field programmable allow a less than ideal design to be released first, then updated later with a more ideal design
in Chapter 2 we discussed the FPGAs basic building blocks:
  - function generators
  - logic cells
  - logic blocks
  - on-chip memory
use these components to assemble larger systems
with the ideas of modularity, cohesion and coupling of components and designs, want to build base systems
is an obvious starting point when describing the processor-memory model

- offers the designer control and a familiar design environment
- useful in rapid prototyping
- as more of the design is offloaded to the FPGA the processor’s role may reduce
- two types of processors may exist, hard and soft
- Chapter 2 gives more details on the differences between the two
if no hard processors exist on the FPGA, a soft processor must be used

this requires sufficient reconfigurable resources

soft processors offer a great deal of flexibility

can be configured to more specifically meet the needs of the design

i.e, if a Memory Management Unit (MMU) is needed, one can be designed/added

soft processors also allow for incremental improvements

i.e, the Xilinx MicroBlaze has seen significant improvements
processors can operate in *standalone* mode offering most basic functionality (i.e, stdin/out)

with a MMU processor can support more verbose Operating Systems

multi-processor support with coherency and shared memory

however, knowing the processors role is key to the design

i.e, PicoBlaze is great for complex state machines, but cannot support Linux
Choosing a Processor

Before choosing a processor, consider the following questions:

- does the FPGA include hard processors?
- are there sufficient resources to implement a soft processor?
- what role will the processor play in your design?
- what type of software will be used on the processor?
- how much time will the processor be used vs. hard cores?
- are there future plans to using a different FPGA?
in order for the processor to do any useful work, memory must be included to store operations and data

memory hierarchies and organization is important to consider

Von Neumann

Harvard architecture

cache (L1, L2, etc.)
Choosing Memory

Before choosing memory, consider the following questions:

- what type of memory is available?
- is there on-chip and/or off-chip memory?
- how much on-chip/off-chip memory is available?
- is the memory volatile or non-volatile?
- how does the processor interface with the memory?
- how does the system interface with memory?
- how do specific cores interface with memory?
- is the memory being utilized efficiently?
Choosing Memory

- modern FPGAs include varying amounts of on-chip memory
- can also use flip-flops for short-term storage (in small quantities)
- can include memory in a core
- as part of the base system connected via a bus
- or off-chip connected through a more complex memory controller
Choosing Memory

- how memory is interfaced is important
- low latency, high bandwidth, and/or storage capacity
- efficient memory controllers are necessary
- consider how memory is connected to the compute core
- i.e., a 400 MB/s bus is not ideal for a 6.4 GB/s memory
- reuse of memory controllers is key — design once and reuse!
- Chapter 6 covers memory in much more detail
Buses

- how to connect a processor, compute core and memory?
- one common approach is to use a *bus*
- processor(s) and memory controller(s) connect to the bus via a standard bus interface
- bus interface is specific to particular bus, but at the simplest level consists of address, data, read/write requests and acknowledgment signals
- the bus also includes a bus arbiter which controls access to the bus
- a core that can request access to the bus is considered a *bus masters*
- not all cores need to be bus masters and only need to respond to requests
- these are called *bus slaves*
Choosing a Bus

Before choosing a bus, consider the following questions:

- what cores will need to directly communicate?
- do certain cores communicate more often than others?
- do specific cores require higher bandwidth between them?
- can any cores function on a slower bus?
One vs. Two Buses

- it is common to find a two-bus system in desktop computers
- isolate cores needing to communicate
- keep processor and memory tightly coupled
- keep lower speed peripherals on secondary bus
- if two or more buses are used, need to include bridges between them
Bridges

- when two cores on opposing buses need to communicate
- i.e., the processor on the system bus needs to communicate with the UART on the peripheral bus
- must include a bridge between the buses
- a bridge is a special core which resides on both buses and propagates requests from one bus to another
- functions by interfacing as a bus master on one bus and a bus slave on the other
Bridges

- slave side responds to requests that must be passed along to the other bus
- master side issues those requests on behalf of the original sender
- sometimes only a single bridge is required, if the peripheral bus only will respond to requests from the system bus, a system-to-peripheral bridge is required
- otherwise two bridges are required to allow dual communication
- common nomenclature for describing the interfaces of a bus in terms of master side and slave side is:
  - *system-to-peripheral bridge* means bridge is slave on system bus and master on peripheral bus
Building Base Systems

- let’s build a base system
- consisting of a processor, on-chip and off-chip memory, and a UART
- with on-chip and off-chip memory controllers
- goal: use this system in future designs
block diagram of the base system consisting of a processor, on-chip and off-chip memory, and a UART
impractical to use schematic capture software for this design
use hardware description languages
use bottom-up design approach
end result, one large HDL file with each component instantiated within it
must also include pin constraints for UART, clocks, off-chip memory
use existing components when available
most vendors provide tools to simplify these steps
Xilinx offers a Base System Builder (used in Chapter 1)
a base system is nice, but let’s add to it
there are many cores, which one to add next?
many designs need network (Ethernet) access
the network core will provide access to the FPGA via a web interface
other common cores may be USB to generic peripheral attachment
and I2C or SPI for low-speed peripherals on the same PCB
Adding another core typically includes:

- instantiating the core in the design
- connecting the core to a bus (or other cores)
- adding custom pin constraints for cores with I/O
- Chapter 7 goes into more details regarding FPGA designs with I/O
Modified Base System

- On-chip memory controller
- Processor
- Off-chip memory controller
- 10/100/1000 ethernet core

System bus

Bridge

Peripheral bus

USB
UART
I2C

block diagram of the base system with the additional cores: networking, USB and I2C
physically speaking the “location” of data is straight forward and easily identifiable

- off-chip memory is located outside of the FPGA’s packaging
- on-chip memory is stored in SRAM cells within the FPGA’s fabric
- from a design perspective accessing the memory is done through an address space
Address Space

- Off-chip memory may be addressable from the address range $0x30000000$–$0x3FFFFFFF$, for a total of 256 MB of addressable data.
- Globally addressable on-chip memory, located on the same bus as the off-chip memory controller, may have an address range $0xFFFF0000$–$0xFFFFFFFF$, for a total of 64 KB of addressable data.
- In a platform FPGA design it is possible to automatically generate these address ranges or to set specific address ranges.
one possible address map for the theoretical two bus system with networking, USB, I2C and UART
Assembling Custom Compute Cores

“why build custom compute cores?”

advantages?
  - speed
  - efficiency
  - predictability

disadvantages?
  - cost
  - design time / constraints
  - achievable performance
Why Build Custom Cores?

- hardware development is hard
- fewer hardware engineers than software engineers
- processors are inexpensive/cost-effective at scale
- common rebuttal: “performance” (or “speed”)
- we will also consider efficiency and predictability
- important for a hardware engineer to know when it is and is not justifiable to implement a custom core
Advantage: Speed

- custom hardware is often used to “speed up” applications
- unfortunately this can lead to a bad assumption:
  - “hardware is faster than software”
- this is a fallacy
- the processor is good at computing sequential operations quickly (typically $5 \times$ to $10 \times$ faster clock rates than FPGAs)
- FPGAs can perform operations in parallel, but must be designed to do so
- FPGAs can implement custom cores to perform specialized operations
- operations are not fixed to 32-bit and 64-bit operations
- memory operations (especially on-chip) can be parallelized
- communication between cores can occur in parallel with computation
Suppose a hardware implementation of task A takes exactly the same amount of time to complete as the equivalent task executed on a processor. We can assume the software implementation is easier to develop. Is there any reason to build a hardware implementation?

- yes, when the hardware is solution is more efficient
- by efficiency, what we are concerned with is how to accomplish a fixed task with variable amount of resources
- by resources, we could be talking about area on a chip, the number of discrete chips, or the cost of the solution
- i.e, a custom hardware core plus processor may be more efficient than two processors
Advantage: Predictability

- timing constraints
- a computation must occur within a given period of time
- custom hardware cores can be designed with predictability in mind
- processors require real-time operating systems
- can design communication path as a design constraint
- embedded systems software rapidly becoming more sophisticated
- includes memory management unit, virtual memory
- can run full operating systems on devices
- can adopt applications originally designed for general purpose desktops
- this can be good or bad – caution needed!
- section aims to cover background information for software implemented on Platform FPGAs
System Software Options

- **system software**

- ranges from simple library of routines
- to full-fledged Operating Systems that virtualizes the hardware for individual processes
- although almost no system software is needed
- standalone systems can run, albeit not always ideal
- useful for initial testing of hardware
- easier to interface with hardware (physical addresses)
a step up from “standalone” is a simple threading library includes the ability to create, schedule, and destroy multiple threads of control simplest of these just provide library calls so the developer does not have to manage context switches and the program has to explicitly yield the processor more advanced threading libraries include preemption (a thread does not have to explicitly yield the processor) and has the ability to schedule the frequency, priority, and deadlines of various tasks

Xilinx offers one integrated into the SDK as *XilKernel* others include eCos, Nucleous, and \( \mu \)C/OS-II
between light-weight threading system services and a full-fledged operating system is \( \mu C \) Linux project.

The project grew out of the Linux 2.0 kernel and was intended to provide support for processors that lacked a Memory Management Unit (MMU).

Without an MMU, there is no virtual memory.

Meaning the operating system cannot create true processes (since a process has its own virtual address space).
System Design

System Software Options

- a full-featured operating system (like Linux)
- these are operating systems that one would find on desktop PCs, workstations, and servers
- a big disadvantage of using an ordinary operating system in an embedded system is that it requires a substantial overhead — the processor has to have an MMU, the OS generally has a large memory footprint (almost always requiring external RAM), and the operating system will include a number of extra processes running concurrently with the embedded application
- however, embedded systems are able to support these OSes, making development easier and porting designs quicker
UNIX and its variants (Linux, BSD, Solaris, and many, many others) share the concept of a root filesystem

- a **filesystem**

- most filesystems organize files hierarchically
- in UNIX, files and sub-directories are grouped in directories
- one special directory called *root* which contains files and sub-directories
- the filesystem data structure is most often implemented on secondary, non-volatile storage
Root Filesystem

- when a filesystem is being manipulated it is typically referred to as a *filesystem image*
- when a filesystem is being used to manipulate files it is called a *mounted filesystem*
- in some cases the designer must create some initial filesystem called the *root filesystem*
- during boot the kernel interacts with the filesystem
- once running the kernel calls a special application called *init* which is the first process to run
- the init process finishes booting the system with the configuration files stored in the root file system
- what this means is that the embedded system designer has to know how to create a filesystem and know how to populate it
- Lab 3 goes into specific details with step-by-step directions
Linux on a Platform FPGA

We refer you to Lab 3 and the Gray Pages for specific details
principles of system design and the hardware and software background that is necessary to be able to construct embedded system designs on a Platform FPGA running with a full-fledged operating system

emphasized important design concepts to support the ability for base systems, custom hardware cores, and low-level components to be reused within a system

discussion of system software and its role in Platform FPGA design
external criteria are characteristics of a design that an end-user can observe

internal criteria are characteristics that are inherent in the structure or organization of the design, but not necessarily directly observable by the user

correctness refers to a system that has been (mathematically) shown to meet a formal specification

reliability in terms of hardware means that the system behaves correctly in presence of physical failures

resilience accepts the fact that there will be errors and the design “works around” problems even if it means working in a degraded fashion
System Design

Chapter 3 Terms

- **Verifiability** is the degree to which parts of the system can be formally verified, i.e., prove correct.
- **Maintainability** refers to the ability to fix problems that arose from unspecified behavior.
- **Repairability** refers to fixing situations where the behavior was specified but the implementation was incorrect.
- **Evolvability** refers to changes that are due to new features.
- **Portability** refers to a system design that can move to new hardware or software platforms.
- **Interoperability** refers to a system design that works well with other devices.
Chapter 3 Terms

bottom-up approach is a form of system design where a design is constructed starting with simple components that are used to build small systems and those small systems are used to build bigger systems and so on.

top-down approach is a form of system design where a design is described at the top-level as consisting of smaller subcomponents, which are described by still smaller subcomponents down to the smallest component.

module is any self-contained operation that has: a name, an interface, and some functional description; and can refer to hardware, software or even something less concrete.
formal interface

is the module's name and an enumeration of its operations including, for each operation, its inputs (if any), outputs (one or more), and name.

general interface

includes the formal interface and any additional protocol or implied communication (through shared, global memory for example).

implicit

functional description of a module refers to the a module's description that is so universal that by convention it is simply understood how it functions, such as the module “FullAdder”.

informal

functional description of a module means its intended behavior is described in comments, exposition, or narrative.
formal functional description of a module means the behavior is either described mathematically (in terms of sets and functions) or otherwise codified, such as a completed C subroutine or a behavioral description in VHDL.

implementation of a module is some realization of the module's intended functionality.

instance of a module is a use of an implementation.

instantiate means to make a copy of an instance.

abstraction is the act or an instance of abstracting or taking away; an abstract or visionary idea.
Chapter 3 Terms

- **State**: (of an application) is all of the data associated with an application's data and part of the system's storage; this includes data stored in registers, program counter, condition codes, RAM, configuration registers in I/O devices.

- **Cohesion**: is a means of measuring the abstraction, if the details inside of a module come together to implement an easily understood function, the module is said to have cohesion.

- **Coupling**: is a measure of how modules of a system are related to one another.

- **Refactoring**: is the task of looking at an existing design and rearranging the groupings and hierarchy without changing its functionality.
Chapter 3 Terms

Regression testing is a sometimes automated testing procedure that might be simulation driven (i.e. test benches) or it may be a set of systems that wraps around the component and exercises its functionality.

Software reference design is a piece of software where the functionally mimics the behavior of the whole system, even hardware modules that have not been implemented yet.

Control flow graph (CFG) is a graph \(G = (V, E)\) where the vertices (or nodes) \(V\) of the graph are basic blocks and the directed edges indicate all possible paths that the program could take at runtime.

Basic block is a maximal sequence of sequential instructions with a Single Entry and Single Exit (SESE) point.
system software refers to any software that assists the application, usually by adding a software interface to access the hardware.

standalone C is a program that runs without the support of and any additional system software.

cross-compiler is a high-level language translator that runs on one platform (a specific processor, a C library, and an operating system) but produces executable for another platform.
monitor is a primitive type of a debugger that is interrupt-driven, either the processor is interrupted or the application being debugged traps to the debugger; also, a monitor usually only supports the most basic functionality, reading/writing absolute addresses, setting breakpoints, manipulating registers.

target

see target machine

target machine

in cross-compiler terms, the processor and operating environment for the embedded system

host

see host machine

host machine

(cross-compiling) when cross-compiling a program, the system that will run the compiler tools for a particular target is called a host machine.