

# ECEN 449 – Microprocessor System Design



## The Design Process

## Objectives of this Lecture Unit

- Help you to take “formal” approach to designing systems
  - Recognize when it is time to start the implementation process
- Provide you with methodologies and strategies you can use in design

## What is “Design?”

- Formally: Design is the process of taking a set of specifications for a system/object and generating specifications for a set of modules that implement the system/object. This is often done recursively until the modules are small enough for a physical implementation.
  - Each module is not ready for implementation using Verilog, for example
- Informally: Design is the process of dividing a system into chunks such that each chunk is small/simple enough that you can implement it
  - Key trick: abstraction, hierarchy and refinement

## The Design Process/Problem

- Projects often start out with very informal specifications “Let’s build a refrigerator with a net connection!”
- Phase 1 of design generally involves refining the specification enough that you can start thinking about implementation
- Phases 2-n take the specification and break it down into a *hierarchy* of implementable chunks
- Very often an iterative process
  - Specification may be impractical
  - Design often reveals things you forgot to specify, although a good specification will minimize this.

## Why do we Need a Design *Process*?

- To manage complexity
  - Human brain can only keep so much detail in mind at one time, although the amount varies wildly from person to person
  - Good design handles complexity in manageable chunks by creating abstractions
- To support parallelism in the implementation process
  - Want multiple people working on a project to reduce time-to-market
  - Good design allows people to work independently by defining modules hierarchically, with clear functions, interfaces and interactions

# Abstractions

- Goal of an abstraction is to create an interface that hides the information that you don't need, while exposing the inputs and outputs that you do.
  - Example: FFT module
    - Do need to know format of input and output data, how long the core will take to perform an FFT, how much area the core takes up
    - Don't need to know how the module implements FFT – this is an issue for the designer of the FFT module
- Advantage:
  - Separation of concerns: an engineer can design something that uses the abstraction without feeling like they need to know everything that is going on “under the hood.”
  - Freedom for experts: do not over-constrain the implementation algorithms

# Examples of Abstractions

- Logic gates
  - Can design functional systems out of logic gates without knowing how a given gate is implemented
  - Example: NMOS vs. PMOS vs. CMOS
- Can consider the I/O interfaces of the chips on the XUP board as abstractions – you'll build systems all term without knowing the details of the circuits that implement each chip
- Interrupts are another example we'll see – hide the details of how the interrupt signaling works from the programmer
  - Write a handler and install into OS
  - Generate the appropriate interrupt request signal
  - Do not need to know exactly how interrupts are implemented in the processor hardware and OS kernel

# Creating Good Abstractions

A good abstraction

- Encapsulates a useful amount of complexity
- Occurs at a “natural” point in the design
  - Performs a function that makes logical sense
  - Has an easily-defined set of inputs/outputs
  - Hides complexity that isn’t required outside the abstraction
  - Allows innovation and flexibility in the module implementation
- Promotes re-use
  - Many useful abstractions “just” avoid the need to re-implement the same function over and over again
  - Ex: library routines in programming languages

## Creating Good Abstractions

### A bad abstraction

- Frustrates the designer
- Requires more effort to use than to implement
- Hides information that the designer needs
- Requires multiple implementations of similar functionality

## Top-Down vs. Bottom-Up Design

- Bottom-up: Start with a set of low-level components. Figure out how to assemble those components into a system that does what you want.
- Top-down: Start with the description of what you want to build, decompose it into successively less-complex modules, until the modules are “simple enough” to implement directly

## What is “Small/Simple Enough?”

- A module is small enough to implement if you can keep all the details of the module in your head simultaneously.
- One good rule of thumb is that a module shouldn't be more than 1-3 pages of code in size.
  - This will vary a lot depending on how much you tend to comment code, what language you're using, etc.
  - The implementation will be in terms of HDL statements and library components
  - This is an upper bound. There are plenty of cases where you'll want to design with even smaller modules.

## Which Approach is Better?

- Both! – for real projects, you'll generally wind up using some combination of both approaches
  - The parts and components available to you can affect the strategy of partitioning the design at higher levels.
  - Ex: Don't want to start design of a microprocessor by thinking about logic gates, but do need to think about what logic gates you have available when you do the design of an adder component
- Early stages of a design almost always want a top-down approach
  - Systems are typically very complex
  - Top-down is good for dividing a task into pieces that can be distributed to team members
  - Caveat: Need to keep in mind the constraints of your implementation technology.

## How Much Design is Enough?

- Most common mistake of inexperienced engineers is moving from design to implementation too soon
  - Very easy to feel that you aren't making progress if you're not writing code.
  - Need to train yourself to think of the design process as a crucial step in getting the job done
- Signs that you're ready to start implementing:
  - You can see how you're going to code up each of the bottom-level modules from their interface description
  - You've thought through how the modules are going to connect to each other, and what data they need to pass back and forth
  - You can clearly explain the interfaces of modules that will be written by one person and used by another

## Thoughts on Design

- It's an art, and a science
- It's inevitable that you'll need to revise the design during implementation.
  - One sign of a good design is that it doesn't change much during implementation
- Similarly, the design process often requires changes to the specification
  - Design may reveal points where the specification was unclear/insufficient
  - Design may reveal requirements that are too difficult to implement
- Design needs to prepare for changes in specification
  - A good design localizes the effects of any given change

## Example Industrial Design Flow

- Starts with a set of specifications for the capabilities of a system
  - Often generated by non-technical team, typically marketing
- Initial design proposal outlining how system will provide the proposed capabilities
  - Design proposal needs to bridge gap between non-technical “what does system do” and technical “how does it do it”
  - Proposal also helps establish feasibility
- Then, design document that provides detailed breakdown of design into modules, specification of each module
  - Extremely large systems may involve multiple design documents at different levels of detail

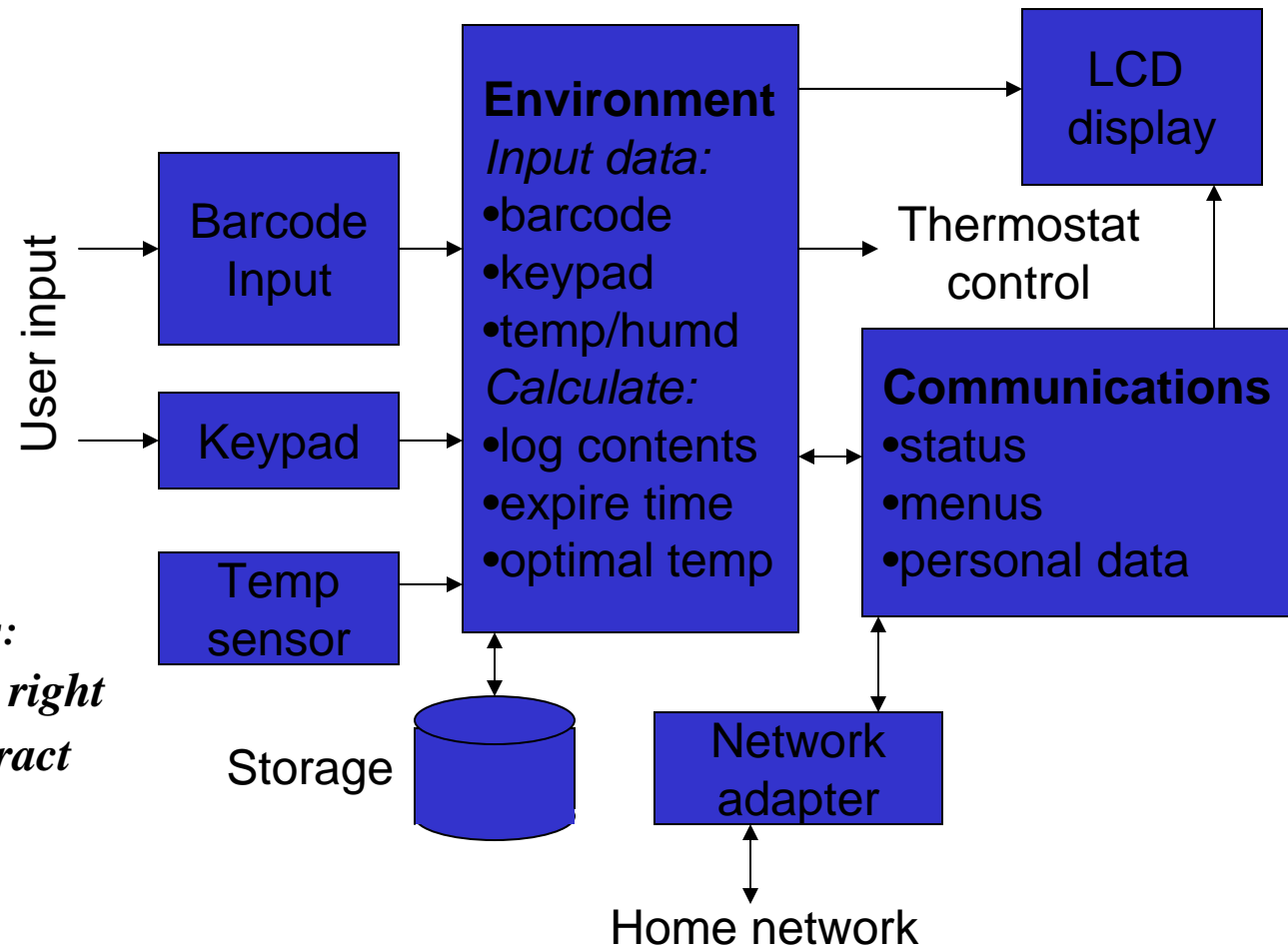
# Design Proposal Document

- Specifications; *initially presented to you, but include as good practice*
  - Listing of desired requirements for engineering team
  - Usually text; often augmented with verbal comments *Take notes and have management sign off on additions and limitations!*
  - Obtain as many specifics as possible (network bandwidth, power requirements/limits, future expansion plans, e.g. *is portability an anticipated possibility?*)
  - Expect specs to change throughout project and design accordingly
  - Establish dialog about flexible parameters with non-technical team
- Block Diagram (6-8 blocks)
  - Your first-pass response: essentially a proposed solution
  - Prepare at a purely functional level
  - High level of abstraction, suggests a natural hierarchy in the design
- Descriptive Design Proposal (1-3 pages)
  - Begin to establish technical parameters
  - Internal description of each block
  - Fully characterize interconnects
  - Be specific about both benefits and shortcomings

## “The Refrigerator of the Future” -- Specifications

- LCD display screen
  - List contents of refrigerator
  - Track items as they enter/exit
  - Warn user about expiring items
  - Adjust temperature to maximize food lifetime
- Food organizer/planner function
  - Generate shopping lists
  - Suggest recipes based on available ingredients
  - Warn user if he/she appears to be not getting enough of any nutrient
- Internet connectivity
  - Automatically order items from delivery service as they run low
  - Download software updates

# “The Refrigerator of the Future” – Block Diagram



## Diagram Characteristics:

- *Flow is left to right*
- *Contents abstract*
- *Interconnects functional in nature*

# “The Refrigerator of the Future” – Document

## Overview:

The “Refrigerator of the Future” is a networked, smart food storage unit capable of monitoring and adjusting internal conditions (temperature, humidity) and contents, as well as providing user guidance for nutrition. As each item is placed in, it is scanned and an entry made in a database. Expiration dates are inferred, and nutritional information obtained via the web. Menu combinations are suggested...and so on.

## Description of each block:

1. Barcode device – Initialize device using I2C protocol. Convert incoming serial stream to 8-bit data bytes and synchronize with central controller. Provide indication of data presence and IRQ capability.

Inputs are: signals from commercial device, system clock.

Outputs include I2C to device, 8 bit data bus to CPU, interrupt to CPU.

## “The Refrigerator of the Future” – Document (cont’d)

2. Keypad – Primary user entry facility capable of both alphabetic and numeric entry via menu-driven format. The anticipated data fields are 1) refrigerator temperature, 2) time and date set 3) personalization for up to 16 family members by name, height, weight, optional nutritional requirements, activity level, etc. 4) maintenance menu for system status and updates. This unit will debounce and decode key press, format data byte and synchronize to central clock, and provide an interrupt to main system.

Inputs are: 4 x-lines and 6 y-lines from keypad, system clock.  
Outputs are 8-bit data to CPU, interrupt to CPU, illumination signal to keypad

- Temperature sensor – Internal environmental detection for refrigerator (can include humidity sensor). Convert 0-5v analog signal to digital, scale to Fahrenheit, Celsius, or absolute, and output to system etc....