16 - System Design

CEG 4330/6330 - Microprocessor-Based Embedded Systems

Max Gilson

System Design

- In the lab you’ve been given a hardware platform and a set of requirements
    - Easy: we know the hardware platform can satisfy the requirements
- In embedded systems engineering you are only given requirements
    - Hard: what hardware satisfies these requirements?
    - Some systems may be impossible to build
- An embedded system engineer must be capable of designing either the entire system or parts of the system for integration

Challenges in Embedded System Design

- How much hardware or type of hardware?
  - How fast of a CPU? How much memory? I/O?
  - How much is too much?
    - E.g. $500 CPU for a thermostat necessary?
- How can we meet deadlines?
  - Faster hardware or more clever software?
  - Is a RTOS required?
  - Might not be strict, customer may ask for a thermostat that sends a temperature reading once per minute
- How do we minimize power consumption?
  - Turn off unused components? Choose different hardware?

Before Starting the Design…

- Before starting our design we have to look at:
    - Requirements
        - The strict requirement that the design must satisfy
        - Some requirements may be impossible to meet!
            - E.g. a smartphone that never dies and doesn’t need to be plugged in

After Completing the Design…

- After completing our design we have to look at:
  - Specification
    - What our design “says” it’s capable of
  - Implementation
    - Our actual physical implementation, hardware, software, and all components

After Completing the Design… (cont.)

- Does our design actually work?
  - Are the requirements realistic and what the customer really wanted?
  - Does the specification actually satisfy the requirements?
  - Does the implementation meet the specification?
  - How do we test the device in real time?
    - When you have a prototype or the final product how can we make sure it does what it says?
  - How do we test using real data?
    - E.g. how do we ensure the thermostat reads temperature and sets temperatures properly?

After Completing the Design… (cont.)

- How do we work with our system?
  - Observability
    - How can we observe what's going on inside the system?
    - Battery voltage reading? Data inside memory?
  - Controllability
    - How can we control our device?
  - What is our development platform?
    - Ex: Arduino IDE, Raspberry Pi OS, etc.

Design Methodologies

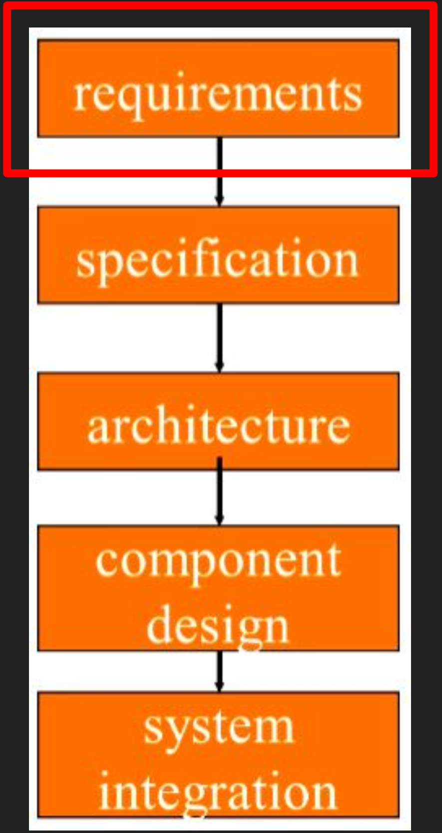
- A design methodology is a process for designing the system
  - Sometimes called a workflow
- It is important to have a process so that no steps are skipped
- Some software tools can help automate steps in the process or keep track of the process itself
  - Compilers, Jira/Trello, CAD tools, etc.
  - Your boss, customers, or certification organizations might need to see you’ve tracked your process accurately

Design Goals

- Your requirements are most likely going to involve one or more of these goals:
  - Performance
  - Functionality and user interface
  - Manufacturing cost
  - Power consumptions
  - Other requirements
    - Physical size, weight, durability, water submerging, etc.

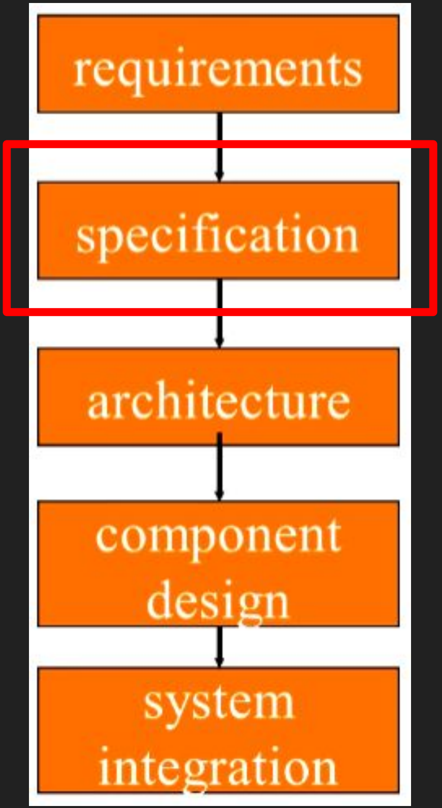
Layers of Design Methodology

- Requirement
  - An outline of what the end product needs to be capable of
  - Example:
    - A thermostat that reads and controls temperature
    - Connects to your phone
    - Can be controlled via the Internet



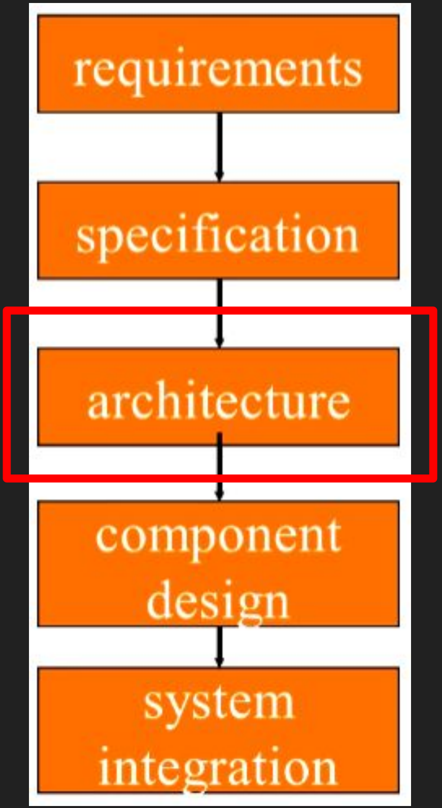
Layers of Design Methodology

- Specification
  - A technical outline that will meet the requirements
  - The specification is for the engineers to follow and adhere to
  - Example:
    - The thermostat will be powered off of 120V AC
    - The temperature reading should have a range of 50F to 90F and an accuracy of ± 1F
    - Wireless internet connectivity



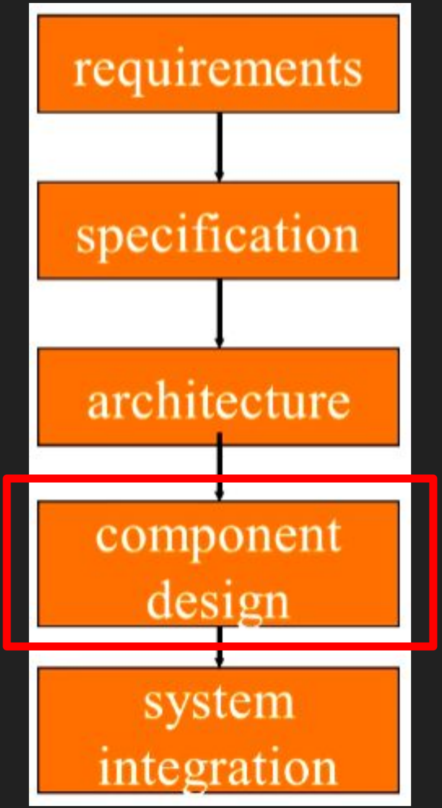
Layers of Design Methodology

- Architecture
  - The high level connectivity of all the major pieces
  - Example:
    - A voltage regulator section to convert 120V AC to DC
    - A microcontroller to read and control temperature
    - A PID loop in software needed for temperature control
    - A WiFi section



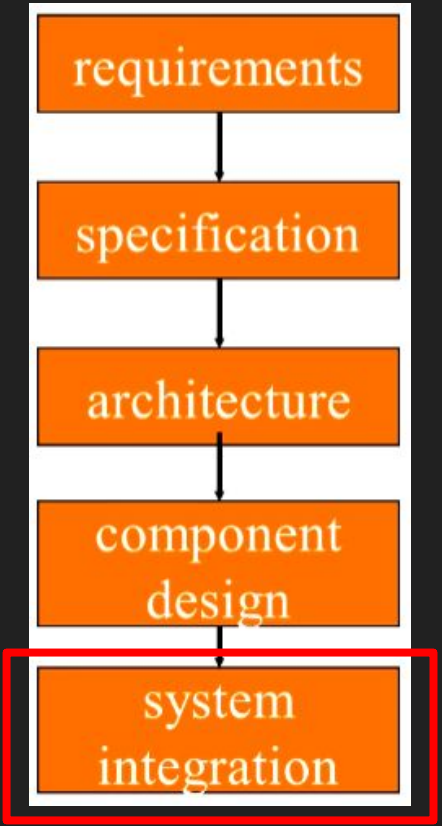
Layers of Design Methodology

- Component Design
  - The selection of the actual components
  - Example:
    - Selecting all of the resistors, capacitors, regulators for converting 120V AC to DC
    - An Atmega328p for the microcontroller
    - A u-blox NINA-W102 for WiFi receiver



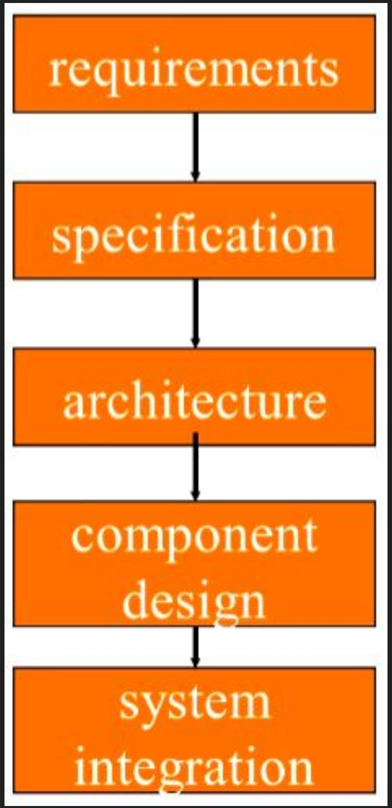
Layers of Design Methodology

- System Integration
  - The actual physical combination of all the components
  - Example:
    - The manufactured circuit board housing all the components and with software installed



Top Down vs Bottom Up

- Top Down Design
  - Start with requirements
  - Build out each step along the way
  - Ensures no steps are skipped and requirements are met
- Bottom Up Design
  - Start with a component(s) then build out into bigger system
  - Could involve development boards or reverse engineering
  - Demonstrates hard to measure values like power consumption, memory usage, speed, etc.
- In practice, both approaches are used



Where to Get Requirements

- Requirements are plain language descriptions of what the user wants and expects
- Requirements can be developed from:
    - Talking directly to your customers
    - Talking to marketing representatives or doing your own market research
    - Providing prototypes to users for comment

Functional vs Non-Functional Requirements

- Functional requirements:
  - Output as a function of input
- Non-functional requirements:
  - Performance
  - Size and/or weight
  - Power consumption
  - Reliability
  - Etc.

Design Process Example (Requirements)

- The marketing team at your company asks you (the engineer) to build a new product
- The product is a dash cam requiring:
  - A “real-time” camera
  - Powered by a car battery
  - Holds onto the last 2 hours of video
  - Mounts to a windshield

Design Process Example (Specifications)

- We can establish some specifications from the requirements
- Requirement: A “real-time” camera
  - Spec: A 480p 25 FPS camera controlled via microcontroller
  - Basis: Seems attainable based on some bottom up design (more later)
- Requirement: Powered by a car battery
  - Spec: Input voltage of 15V to 12V DC
  - Basis: A car’s battery voltage ranges from 15V to 12V DC
- Requirement: Holds onto the last 2 hours of video
  - Spec: 4 GB SD card
  - Basis: https://www.dr-lex.be/info-stuff/videocalc.html
- Requirement: Mounts to a windshield
  - Spec: A special case design that encloses the electronics and has a suction cup
  - Basis: A suction cup is a great mount for a windshield

Design Process Example (Architecture)

- Next, we can create a block diagram to outline our architecture
- We will most likely need the following blocks based on our specification:
  - Camera
  - Microcontroller or Microprocessor
    - Source code outline
  - Power regulation
  - SD card

Design Process Example (Component Design)

- Some bottom up design may prove useful in deciding our components
- First, we can survey different dash cams (reverse engineering) and camera systems to see if there already exists an open source or "solved" system
  - The ESP32-CAM satisfies the "real-time" camera requirement and can run at 480p at 25 FPS
  - Source: https://github.com/jameszah/ESP32-CAM-Video-Recorder-junior
- Next, we can start breaking down our architecture into components that make them up
  - Microcontroller circuits
  - Power regulation circuits
  - Source code

Design Process Example (Implementation)

- Next is building the actual implementation
  - Purchase the components and assemble it yourself OR ask a manufacturer to assemble it for you
  - Write and install any software required (easier said than done)
  - Test the implementation and verify that it meets the specification
    - If yes: success!
    - If no: iterate, fix issues, revise the architecture or components, etc.

Design Verification

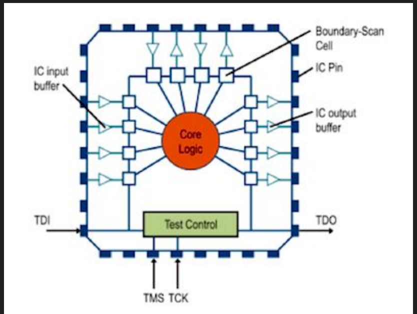
- It is important to make sure the design meets the specification
  - You should verify your design
- If you have not tested it how do you know?
  - For example, if we say:
    “Our device can take an input voltage from 15V to 12V DC.”
    Does the device always work in this range? Is there performance degradation at 12V vs 15V?

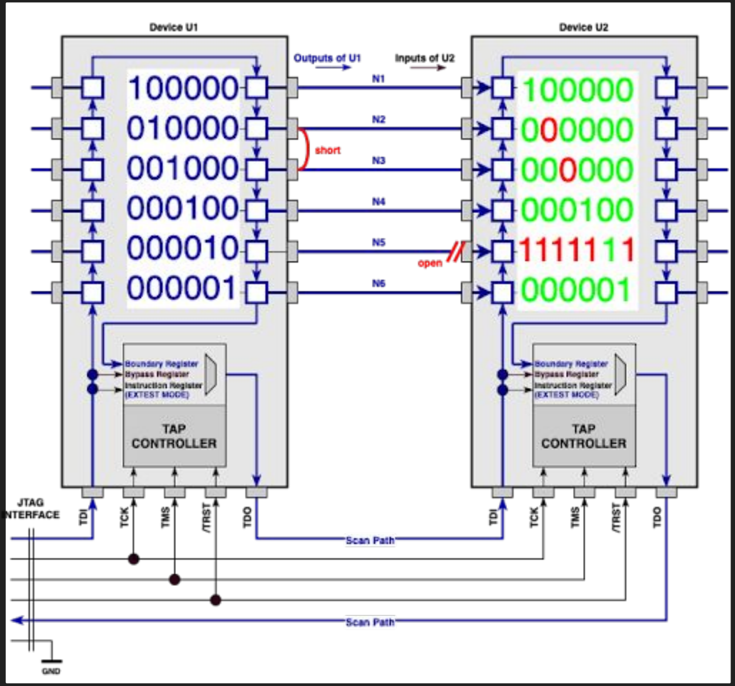
Built-In Self Test (BIST)

- A built-in self test is a great method for verifying the design works as intended
  - For example, if you built a drone, you may want it to check some things when it is powered up
    - Verify the microcontroller can communicate with sensors
    - Verify battery is healthy
    - Verify memory with a checksum
- This can be done by a built-in test that automatically runs
  - This concept applies not just to embedded systems, but any system
    - Example: could apply to the anti-lock brake system in a car

JTAG Testing

- Joint Test Action Group (JTAG) is an industry standard method for testing hardware
  - Replaced bed of nails testing
- JTAG is hardware inside the microcontroller itself that allows you to programm, debug, and verify pin functionality inside of the microcontroller





The Inevitable Moving Goal Posts

- After you complete your design or prototype, the marketing team might say:

  "We can't sell a 480p dash cam! The quality is too bad! Let's revise the requirements to require 1080p!"
- Do we have to change anything?
  - Hint: yes, almost everything
- Moral of the story: make sure you have ALL the requirements before you design

Designing for Reliability

- How can we quantify if a system is reliable or not?
  - Mean Time to Failure (MTTF)
  - Mean Time Between Failures (MTBF)
  - Availability is equal to MTTF / MTBF
- There exists some probability that a system will fail
  - This probability changes over time
- Repairing failures can be done by:
  - Mask errors (hide the error)
  - Hot spare (swap with spare part)

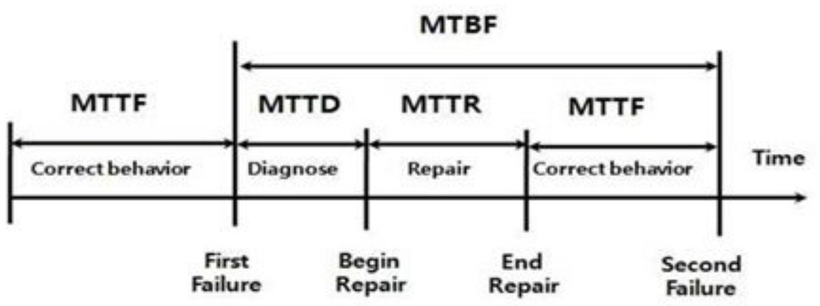


Fig. 1 MTTD, MTTR, MTTF & MTBF

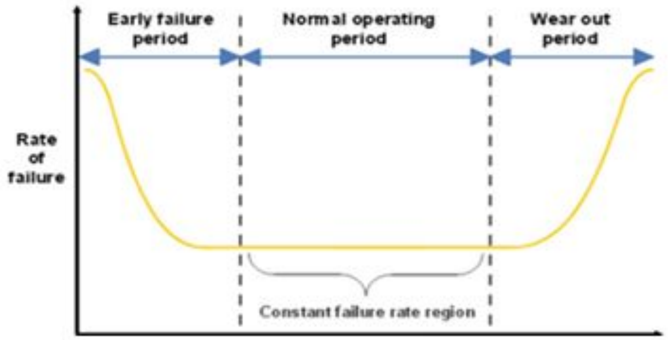


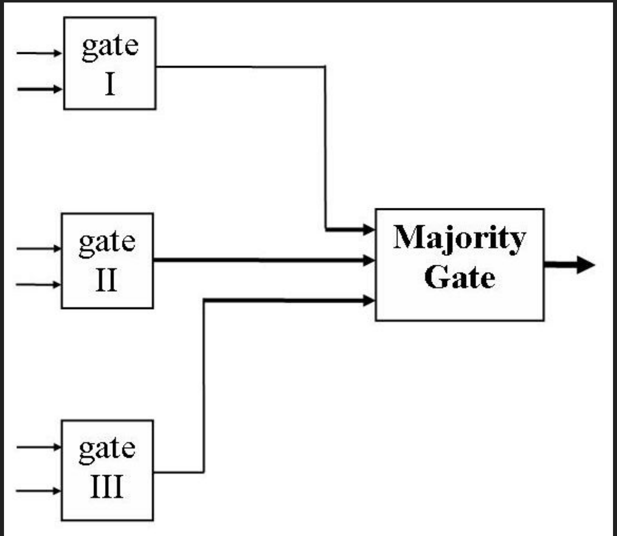
Fig. 2: Bathtub curve illustrate consistent rate of failures

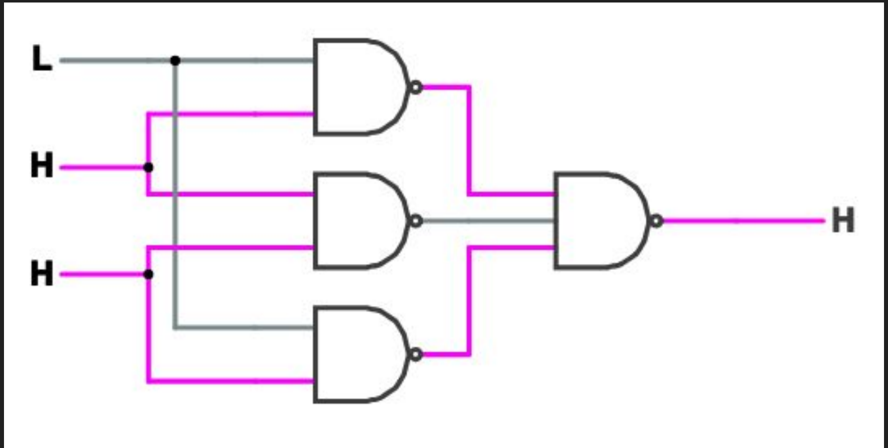
DMR and TMR

- Dual modular redundancy (DMR) is having two copies of the same hardware or software component working in parallel
  - Software components should be developed by separate teams using the same specification
  - Provides error detection but not error correction
  - If my components both give me a different output, which is correct?

DMR and TMR (cont.)

- Triple modular redundancy (TMR) is having three copies of the same hardware or software component working in parallel
  - Provides error detection AND error correction
  - When using 3 parallel components we can perform error correction
  - If 1 component is outputting something different than the other 2 it can be replaced
  - A gate can be developed to take the majority of outputs
- $R_{TMR} = R_v\,(3\,R_m^{\ 2} - 2\,R_m^{\ 3})$
  - R_v is probability the majority gate (voter) fails
  - R_m is probability the node fails
  - R_{TMR} is the probability the whole system fails





ECC Memory

- Error Checking and Correction (ECC) is a method for verifying that memory is operating properly
  - A memory chip might have 64 data pins or 72 data pins
  - If using 72 pins: 64 are data bits, 8 are check bits
- Using a Hamming code encoder, there are 4 data bits and 3 check bits, then we have 3 even-parity equations
  - Check bits are produced to an even parity for each equation
  - If any of the equations output non-zero, an error is detected and can be corrected
    - If output is (1, 1, 1) then D3 must be wrong
    - If output is (1, 0, 0) then C2 must be wrong
  - Fails when two bits are wrong (but this is unlikely)

$$C2 \oplus D3 \oplus D2 \oplus D1 = 0$$

$$C1 \oplus D3 \oplus D2 \oplus D0 = 0$$

$$C0 \oplus D3 \oplus D1 \oplus D0 = 0$$

Burst Errors

- Hamming code is useful when a noisy environment might corrupt communication
  - A major downfall is we can only detect 1 bit errors
- Burst errors are going to cause a burst of errors to occur in short periods of time
  - Can result in multiple bits being wrong
- Reed-Solomon error correction can mitigate this issue but is significantly more complicated

Cyclic Redundancy Check (CRC)

- A cyclic redundancy check module can create a signature for any piece of data sent
- This signature can be recreated on both the receiver and the transmitter side of the communication
- If there is an inconsistent signature on either side, the data received contained errors
- Any size of data can be sent either 1 bit or 1000's of bits
  - A signature may not always be unique