Embedded systems are key elements of our society today
  ✓ An adequate supply of competent designers is a rate limiting factor in our economies ability to grow

As processing power increases, we’ll be able to do incredible things that we haven’t begun to imagine.

It is a discipline of Computer Science that has traditionally been:
  ✓ Ignored
  ✓ Conceded to Electrical Engineering Departments
  ✓ Treated as a cast-off because of its intimacy with “hardware”
  ✓ Not well understood because of its special niche

Engineers who are experienced embedded system designers are in short supply and high demand

It’s a really fun topic!
Embedded System

- Dedicated to a specific task or tasks
- Rich variety of microprocessors (over 300 types)
- Designs are cost-sensitive
- Have real-time performance constraints
- Used with Real-Time Operating Systems (RTOS)
- Software failure can be life-threatening
- May have constraints on power consumption
- Operate over a wide-range of environmental conditions
- Fewer system resources than a desktop system
- All code might be stored in ROM
- Require specialized design tools
- May have on-chip debugging resources
Embedded System

- In general, there is no architectural link to standard platforms
  - PC (Win9X, NT, XP, Linux), MAC, HP, Sun are considered the standard platforms
  - Almost every embedded design (hardware and software) is unique
  - The hardware and software are highly integrated and interdependent

- Typically, have moderate to severe real-time constraints
  - Real-time means system must be able to respond to the outside world

- May or may not have Operating System (O/S) services available
  - No printf() for debugging when there is no terminal!

- Tolerance for bugs is 1000X (or more) lower in embedded systems than in desktop computers.
  - May be life-threatening consequences if system fails
  - Often engineered for the highest possible performance at the lowest cost
  - Performance may not be an important consideration

- Most likely to have power constraints
Co-Design

- Define the IP
- Partition IP between HW and SW

SW Process
- Design algorithm
- Write C/C++ code

Co-design Phase
- Design algorithm
- Write HDL code (dialect of C)

HW Process
- Write test vectors
- Run simulations

Integrate
- Compile to object code

Iterate Software
- Compile to Si foundry database

Re-spin the ASIC
- The differences between the process of designing hardware (ASIC) and the process of designing software are disappearing
Processor Selection Phase

- Packaged benchmarks
- Past Experience
- Other Similar Designs
- Applications support
- Instruction Set Simulator
- Benchmarking tools

Design Phase

- Software Design Team
- Compiler Tool Chain
- Instruction Set Simulator
- Evaluation Board
- Out-of-Circuit Emulation
- Initial Software
- Software running on Hardware
- Software Performance Analysis
- In-Circuit Emulation
- Logic Analyzer/Preprocessor
- JTAG Emulation
- ROM Emulation
- MiniMON29K

Design Win

- Literature
- Compiler Tool Chain
- Architectural Simulator
- Evaluation Board
- Customer Benchmarks

Hardware Designer(s)

- Logic Simulator
- Oscilloscope
- Initial hardware without working memory system
- In-Circuit Emulator
- Oscilloscope

SW/HW Integration Phase

- Maintenance and upgrade of existing products

Courtesy of Daniel Mann
- ROM Emulator
- ICE (In Circuit Emulator)
- (on-chip)
Power Budget
Cost of Goods
Real-time Constraints
Legacy Code
Time to Market
Performance
Landmines
Tool Support
- Issue 1: 
  - Width of data path
  - Clock speed: RAW MIPS
  - Processor architecture issues
  - Single or multiple processors
Issue 2: Integration of functions

✓ Microprocessor or micro-controller?
✓ Review:
  ▪ A microprocessor contains the basic CPU functionality, and little more
  ▪ A micro-controller combines the CPU core with peripheral devices
✓ The microprocessor is usually the leading edge of performance
  ▪ Lowest level of integration
  ▪ Highest cost
✓ Higher levels of integration imply
  ▪ Lower system costs
  ▪ Greater reliability
✓ As uP matures the core moves into the uC families
Higher level of integration (continued)
- Less power
- Faster
- Higher processor costs

Issue 3: Use a microcontroller
- Peripheral choices (timers, ports, serial comm., A/D, etc.)
- On-chip, RAM, ROM, Flash
- Power requirements
- Sleep modes
- Commercially available or build to order (Motorola 683XX)
Micro-controller vs System-on-Silicon

- Application Specific Integrated Circuit (ASIC)
  - Processor is soft, the processor exists as an encoded HDL
    - Licensed foundries can fabricate CPU core design into actual
  - Intellectual Property from ARM, MIPS, Motorola MoCore, ARC
    - Companies do not build ASICs, called Fabless vendors
  - Customizable CPUs

- Multiple CPU cores
  - Mix RISC and DSPs
  - Designs out with 64 (and more) 32-bit RISC and DSP cores
Issues 4 and up: Software considerations

- Legacy code base for existing architecture
  - C code may or may not be portable
  - Assembly code is definitely not portable

- Instruction set architecture issues
  - Certain ISAs may be better for certain problems
  - Engineers may be more familiar with certain instruction sets
Not so obvious considerations:

- Compatibility with existing tools
- Processor vendor's roadmap for the future and long-term support
- Design assistance, availability of IP
- Legacy code
  - "C is portable", assembly code is not
- Pricing and availability
- Availability of third-party tools (Emulators, debuggers)
- Power consumption (power budget)

And... There are lots to choose from!

- Currently there are over 100 32-bit embedded processors
- About 1000 different total devices
Recall the beginning phase of the embedded lifecycle

A significant effort usually goes into partitioning the “algorithm” between the hardware components and the software components

Critical part of the design because mistakes made here can cause an inferior product to result

- Software team usually tasked with picking up the pieces

Today, partitioning is becoming an almost seamless process

Tools can be used to create a generalized algorithmic solution (UML) and then partitioned into software and hardware components
The hardware and software in an embedded system work together to solve a problem (algorithm).

The decision about how to partition the software components and the hardware components is usually dictated by speed and cost:

- Dedicated hardware is fast, inflexible and expensive
- Reconfigurable hardware is fast, flexible and more expensive
- Software is slower, more flexible and cheaper
There has been a revolution in hardware design called **Hardware Description Languages (HDL)**

- Commonly known as Verilog and VHDL
- Like C, but includes extensions for real time and other hardware realities
- *Abstracts the design of hardware away from gates and wires to algorithms and state machines*

Hardware design is described in HDL and then compiles to a recipe for the silicon FAB to build the circuit

- Called **Silicon Compilation**

A single HW designer can now develop an IC that required entire teams several years ago

- This has led to whole new concept of **Systems-on-a-chip, or SOC**
**Key Point**

The differences between the process of designing hardware (ASIC) and the process of designing software are disappearing.
Situation Worse in S/W

DoD Embedded System Costs

Billion $/Year

Hardware

Software

On-going Paradigm Shift in Embedded System Design

- Change in business model due to SoCs
  ✓ Currently many IC companies have a chance to sell devices for a single board
  ✓ In future, a single vendor will create a System-on-Chip
  ✓ But, how will it have knowledge of all the domains?

- Component-based design
  ✓ Components encapsulate the intellectual property

- Platforms
  ✓ Integrated HW/SW/IP
  ✓ Application focus
  ✓ Rapid low-cost customization
What is IP?

- *Predesigned, preverified silicon circuit block*, usually containing 5000 gates, that can be used in building larger application on a semiconductor chip
IP-based Design

- Which DSP Processor? C50? Can DSP be done on Microcontroller?
- Which Microcontroller? ARM? HC11?
- How fast will my User Interface Software run? How Much can I fit on my Microcontroller?
- Can I Buy an MPEG2 Processor? Which One?
- Do I need a dedicated Audio Decoder? Can decode be done on Microcontroller?
UML and Co-design

- Two new technologies have emerged that address this desire to focus on the algorithm and not the partition
  - Unified Modeling Language (UML)
    - iLogix, ObjecTime, Cadence, CoWare
  - HW/SW Codesign
    - Synopsys, Mentor Graphics, VAST

- UML products, such as Statemate and Rhapsody (iLogix) and Co-design products, such as Seamless (Mentor Graphics) blur the distinction between hardware and software

- Output of UML tool is C++ application code and/or VHDL code
  - Partition decisions are made by which code is generated

- Objective: Design the algorithm in an implementation independent way
  - Let the automatic code generator do the grunt work
Unified Modeling Language

- UML is a way to represent the finite-state behavior of an embedded system:
  - A finite state machine is an abstract machine that defines a set of conditions of existence (called “states”), a set of behaviors or actions performed in each of these states, and a set of events that cause changes in states according to a finite and well-defined rule set.

- Consider some examples of a finite state machine
  - Vending machine
  - Gas pump
  - Flight control system
HW/SW Co-verification

- After a design is partitioned and separate HW and SW development is proceeding, how do you know that you’ll get it right?
  - Re-spinning an ASIC is costly
  - Fixing a problem in software will reduce performance

- HW/SW Co-verification focuses on validating the correctness of the design across the HW/SW interface
Embedded System Lab. II

- 0% [page]
- 60% [page]

Page 29
System Specification & Design

HW & SW Design/Debug

Prototype Debug

System Test

37%

20%

31%

12%

51% of Time

Source: Dataquest

Cost to fix a problem

Maintenance and upgrade

Cost to fix a problem

51% of Time
Integration

- Hardware/Software integration
  - The separate hardware designers, firmware designers and software designers have completed unit testing their code
  - The hardware has been built and passes the *smoke test*

- The hardware is turned over to the software team
  - Load the software image
  - Validate the correctness of the hardware performance against the software design
  - Integrate their software with the hardware
  - Find the bugs
  - Test for performance and reliability
Debugging embedded systems is made more difficult because of the added dimension brought about by untested hardware.

The process of debugging an embedded system is usually referred to as **Hardware/Software Integration**.

- Identifies this as a unique process.

---

**The Integration “Loop”**

- **Start**
  - New Test? (Yes) → **Run Test**
  - No → **Stop**
  - Yes → **Pass Test?**
    - Yes → **Debug**
    - No → **Re-design physical h/w and/or s/w**
Use an oscilloscope to look at the parametric performance of the hardware system

- Is the power supply voltage stable?
  - Minimal AC ripple
  - Within limits

- Do the bus signals look clean?
  - Ringing on the waveform is within bounds
  - System noise is within acceptable bounds
  - All busses are properly terminated, no reflections

- Is the clock(s) distribution clean?
  - Stable waveform without noise, undershoot or overshoot
  - Clock skew within acceptable limits
- Verify that the processor to memory interface is stable
  - Memory set-up times are within acceptable limits
  - Hold times are long enough
  - Proper timing on reads and writes
  - Use an In-Circuit Emulator to force simple memory test programs
    - No need to load a program to test memory
  - Use a Logic Analyzer to look at bus activity to verify that all signals are understood
    - Verify that the memory decoding logic is working properly
- Use emulator to force accesses to peripheral devices
  - Memory access conflicts
  - All glue logic PAL equations are correct
  - Read and Write to ASIC registers
  - Read and write to registers of other peripheral devices

- Use an ICE (In Circuit Emulator) to load simple HW test programs *(diagnostics)*
  - Diagnostics will exercise hardware and verify reliability
  - Will not verify functionality
An RTOS enables a design team to create a complex system that responds to every event with the right action at the right time, every time.

- Provides mechanisms to precisely synchronize a large number of tasks
- Reliable

An RTOS provides mechanisms to handle faults, bugs, and other unexpected events that inevitably affect any system of embedded software

When the level of complexity of an embedded system is such that partitioning the software into smaller, independently executing modules significantly lowers the overall design complexity and leads to greater system reliability

- Time-to-market pressure balances cost of the RTOS
RTOS - Pre-emptive schedule

- General purpose operating systems are very democratic
  - Every task or process has equal access to the CPU
  - Scheduling decisions are based upon time sharing
    - Tasks get to execute without interruption until their time slice is up
    - Problem: If a less important task is running and a more important, deadline dependent task is pending, then the higher priority task must wait its turn

- RTOSes are different!
  - Pre-emptive scheduling allows the higher priority task to take over the CPU from lower priority tasks
  - Extends from application level code to drivers and interrupts
RTOS - Predictability

- RTOSes must provide the designers with predictable performance data for system latencies
- Need to know best case and worst case scenarios for
  - Task switching times
    - Elapsed time from the completion of the last instruction of the prior task to the beginning of execution of the first task of the replacement process
    - Measure of the overhead caused by the RTOS
  - Interrupt handling
    - Elapsed time from the arrival of the interrupt signal to the beginning of execution of the first instruction of the interrupt handler
    - Worst case timing for a lower priority interrupt must take into account the time required to process all higher priority interrupts
Generally, we view RTOSes as intellectual property
- Pay a royalty to RTOS vendor for each copy of the RTOS that you deliver in your application (annuity)
  - Example: WinCE royalty ~$15 per copy volumes > 10000

Cost of tools
- Some RTOS vendors supply complete development and debug environments for their customers
  - One-stop shopping
  - Products are well-integrated and work together

Board support packages (BSP)
- Integrating an RTOS with your hardware platform (target system) is a non-trivial exercise
- Requires expert knowledge of the O/S and hardware
- Is there an existing BSP for your hardware?
  - Pay the O/S vendor to do it? ( $50K - $1M )
RTOS - CPU issues

- Has an RTOS been written and fine-tuned to your CPU choice?
  - Take advantage of specialized performance hardware on CPU?
  - Example:
    - May contain shadow registers for rapid task switching
    - May contain special registers for storing task tables context switching.
  - Key kernel functions written in assembly language for maximum performance

- Has performance of RTOS been fully characterized with your CPU?

- Does the RTOS support the peripheral register set on your microcontroller?
  - Example: PowerPC 860 contains 200 registers for peripheral devices (timers, ethernet ports, other O/S relevant I/O devices) require support

- Support for virtual and protected memory management?
RTOS - Application support

- How many simultaneous tasks can the RTOS support?
- How many interrupts can be supported?
- Does RTOS vendor provide support services for specialized applications?
  - FAA and FDA certification of O/S for mission critical applications
  - Protocol stacks for telecomm and datacomm (TCP/IP)

- Architecture of RTOS
  - Flat memory model
    - Applications and RTOS are built and linked as a single executable
    - Entire address space of processor is available to O/S and all applications
    - Generally fast, but very susceptible to crashes due to buggy code
      - Errant pointers can overwrite critical kernel code