Course Introduction

Hardware Software CoDesign

August 2011

Agenda

Course Introduction

1. Go through the Pre-requisite Embedded Systems Handout (Download from BITS).
2. Go through the Hardware Software CoDesign Handout
3. Some Historical Background

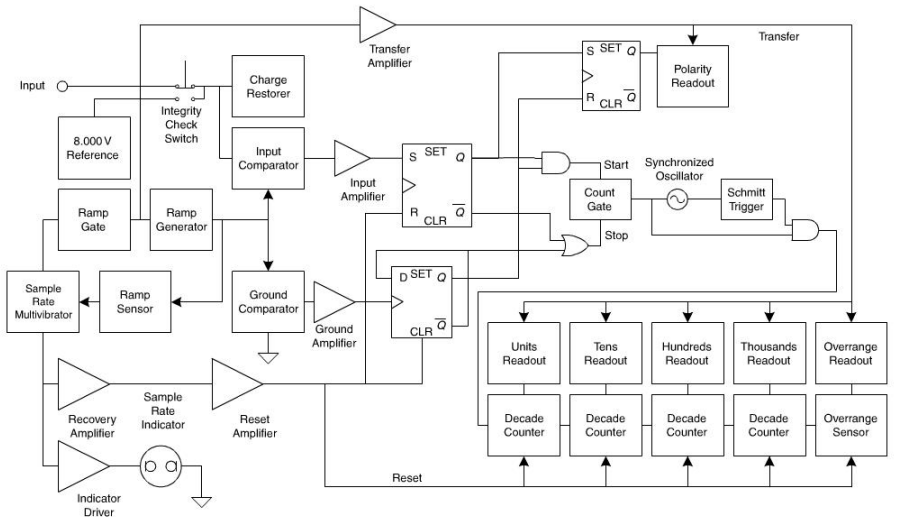
Course Introduction

Historical Background

1. The course of electronics system design changed irreversibly on November 15, 1971, when Intel introduced the first commercial microprocessor, the 4004.
2. Before that date, system design consisted of linking many hardwired blocks, some analog and some digital, with point-to-point connections.
3. The change :-
   (a) The injection of sofware / firmware into the system-design
   (b) The use of buses to interconnect major system blocks

Course Introduction

Historical Background



■ **FIGURE**

A digital voltmeter block diagram adapted from the design of a HP 3440A, circa 1963.

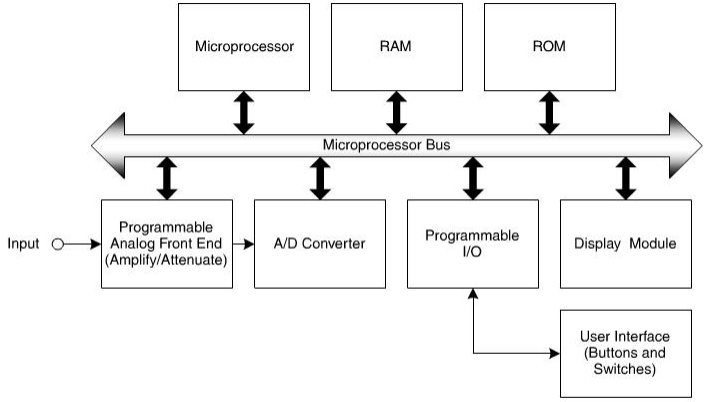
Course Introduction

Historical Background

1. The block diagram shows mix of analog & digital elements interconnected with point-to-point connections.
2. Even all-digital measurement counter, which stores ADC output consists of counters. Each counter drives its own numeric display and communicates to the next counter over one wire.
3. There are no busses in the design, none are needed.
4. There are no microprocessors, it would not appear for another 8 years.

Course Introduction

Historical Background → Enter the Processor



■ **FIGURE**

A version of the HP 3440A block diagram of Figure 1.1, adapted for microprocessor-based implementation. This system-design style is still quite popular.

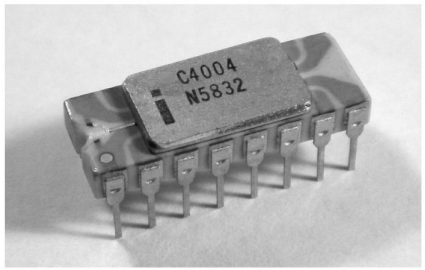
Course Introduction

Historical Background

1. The figure illustrates how a system designer might implement a digital voltmeter like the HP3440A today.
2. A Microprocessor controls all of the major system components in the modern design implementation.
3. The processor communicates with other components over a common bus → the microprocessorś main bus.
4. Now, what are the other differences :-
   - Massive amount of parallelism in early design **Vs** the one-operation-at-a-time over the microprocessor bus.
   - It was more expensive to routed multiple point-to-point connections **Vs** less expensive to route buses to move data into and out of packaged microprocessors.
   - Nothing need to be multiplexed due to full parallelism **Vs** Limitations on the frequency of operation due to multiplexing the shared bus (resource).
5. What happens NeXT :-

Course Introduction

Historical Background → Few Pins = Massive Multiplexing



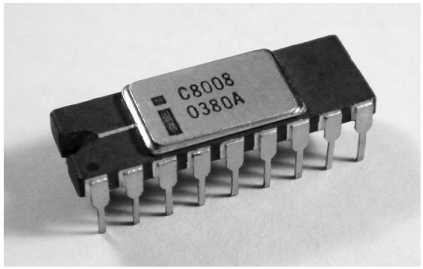
■ **FIGURE 1.3**

The Intel 4004 microprocessor, introduced in 1971, was packaged in a 16-pin package that severely limited the processor's I/O bandwidth and restricted its market accordingly. Photo Courtesy of Stephen A. Emery Jr., www.ChipScapes.com.

1. The i-4004 was packaged in a 16-pin DIP.
2. The 4bit bus had multiplexed access to the various components in the system. The architecture had multiplexed address and data bus.
3. It took 3 cycles to pass a 12bit address and another 2 or 4 more to read back an 8 or 16bit instruction.
4. With a maximum operating frequency of 740kHz and long, multi-cycle bus operations, the i-4004 was too slow to take on many system control tasks and was largely ignored.

Course Introduction

Historical Background → Few Pins = Massive Multiplexing



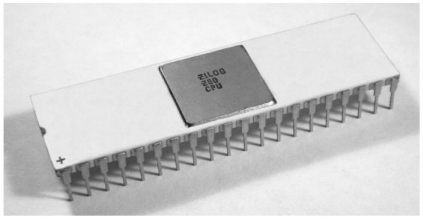
■ FIGURE 1.4

Intel got more bus bandwidth from the 8008 microprocessor by squeezing it into an unconventional 18-pin package. Photo Courtesy of Stephen A. Emery Jr., www.ChipScapes.com.

1. The i-8008 was introduced in April, 1972 packaged in a 18-pin DIP.
2. The 8bit bus had multiplexed access to the various components in the system. The architecture had multiplexed address and data bus.
3. It took 2 cycles to pass a 14bit address and another 1 or 2 more to read back an 8 or 16bit instruction.
4. With a maximum operating frequency of 800kHz, the i-8008 was still too slow to fire the imagination of many system designers.

Course Introduction

Historical Background → More Pins Better Bus Performance



■ FIGURE 1.5

Intel finally crossed the bus-bandwidth threshold into usability by packaging its third-generation 8080 microprocessor in a 40-pin package. Many competitors swiftly followed suit (shown is Zilog's 8-bit Z80 microprocessor) and the microprocessor quickly became a standard building block for system designers. Photo Courtesy of Stephen A. Emery Jr., www.ChipScapes.com.

1. The i-8080 was introduced in April, 1974 packaged in a 40-pin DIP.
2. The separate address bus was 16bits and data bus was 8bits
3. With a maximum operating frequency of 2MHz, the i-8080 finally got system designers to adopt the microprocessor as key system building block.
4. Other Microprocessor Vendors like Motorola & Zilog also introduced at about the same time in 40pin DIP (shown is the Z80 but mentioned i-8080).
5. Since then, microprocessor based design has become the nearly universal approach to system design.

Course Introduction

Historical Background → Microprocessor = Universal Building Block !

1. Economics :: Standard uP provide abilities at a modest cost.
2. Hardware is more difficult to change than Software / Firmware.
3. Hardware designer can design a uP based system and build it before the systems function is fully defined.
4. Adding the software / firmware into h/w finalizes the design and this event can occur after the h/w has been designed, prototyped, verified, tested, manufactured and even fielded. This allows for h/w and firmware to be developed concurrently which reduces the project schedule (ideally yes).
5. NOW, systems needed more tasks to be executed on processors... What happened...

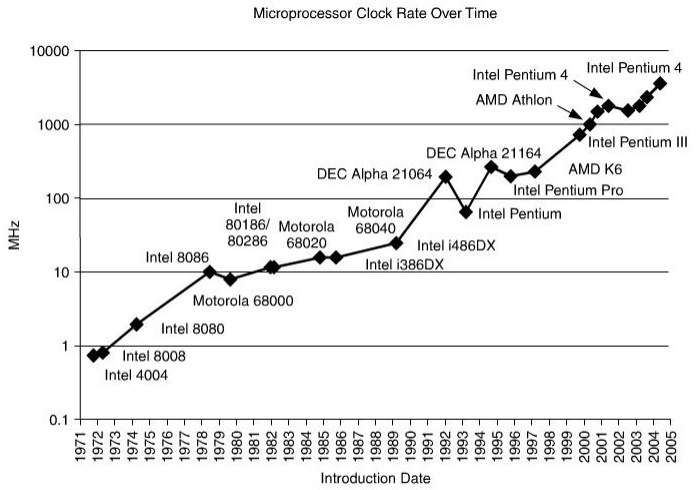
Course Introduction

Historical Background

1. NOW, systems needed more tasks to be executed on processors... What happened... uP Vendors adopted the following...
   - Increase the clock rate... rush by uP vendors to do this...
   - The uP data-word and buses widened... 16bit, 32bit, 64bit..
   - Add more buses to the processor architecture...
2. WHAT IS the fallout of all these techniques ?

Course Introduction

Historical Background



■ **FIGURE 1.6**

Microprocessor clock rates have risen dramatically over time due to the demand of system designers for ever more processor performance.

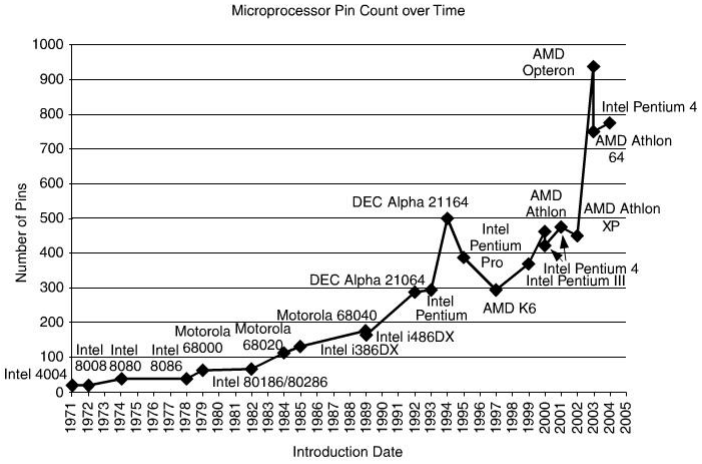
Course Introduction

Historical Background

1. NOW, systems needed more tasks to be executed on processors... What happened... uP Vendors adopted the following...
   - Increase the clock rate... rush by uP vendors to do this...
   - The uP data-word and buses widened... 16bit, 32bit, 64bit..
   - Add more buses to the processor architecture...
2. WHAT IS the fallout of all these techniques ?
   - Device pin size increase.
   - Device power dissipation increase.

Course Introduction

Historical Background

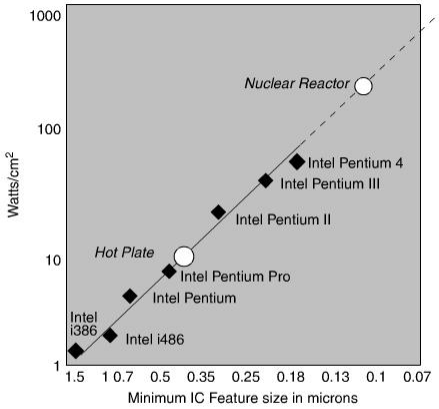


■ **FIGURE 1.7**

Microprocessor pin counts have also risen dramatically over time due to the demand of system designers for ever more processor performance.

Course Introduction

Historical Background



■ **FIGURE 1.8**

Packaged microprocessor power density has risen exponentially for decades. (*Source*: F. Pollack, keynote speech, "New microarchitecture challenges in the coming generations of CMOS process technologies," MICRO-32, Haifa, Israel, 1999.)

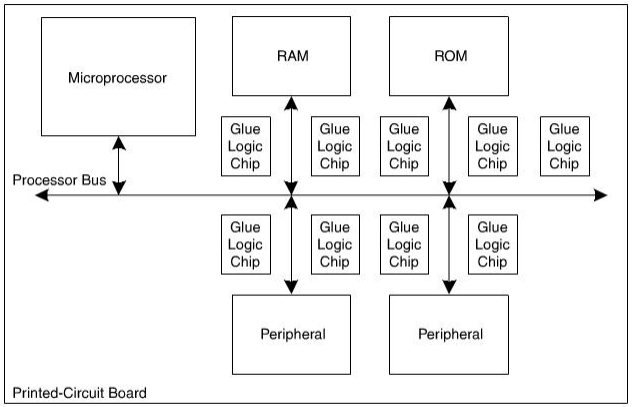
Course Introduction

Historical Background

1. APART from the technological challenges, system design evolved...
   - First with uPs and components like RAM, ROM, Glue, Peripherals on a system PCB.
   - Then with uPs and system components on a single die (SoC), with fewer components on system PCB.
   - The Glue logic evolved from simple 74LS devices, to Glue ASICs (PLAs), to FPGAs

Course Introduction

Historical Background



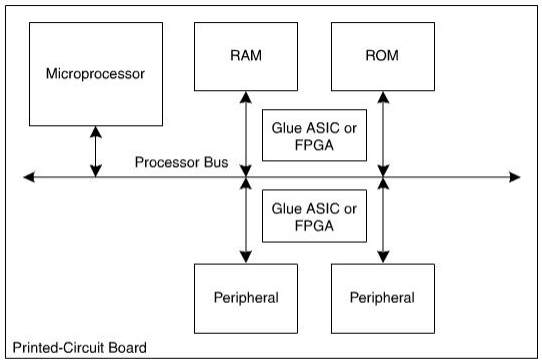
■ **FIGURE 1.11**

By 1985, microprocessor-based system design using standard LSI parts and printed-circuit boards was common.

1. First with uPs and components like RAM, ROM, Glue, Peripherals on a system PCB.

Course Introduction

Historical Background



■ **FIGURE 1.12**

By 1990, system design was still mostly done at the board level but system designers started to use ASICs and FPGAs to consolidate glue logic into one or two chips.

1. Then with uPs and system components on a single die (SoC), with fewer components on system PCB.
2. The Glue logic evolved from simple 74LS devices, to Glue ASICs (PLAs), to FPGAs

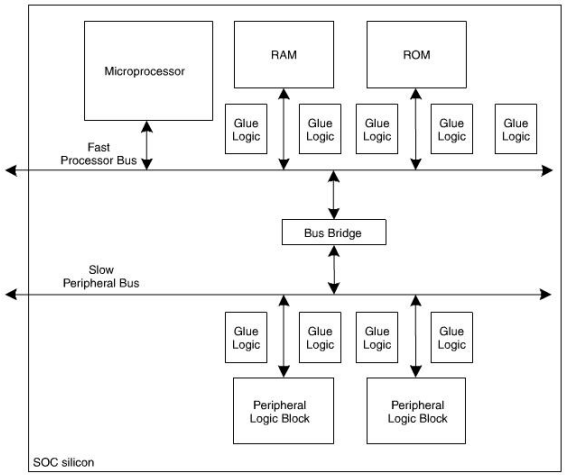
Course Introduction

Historical Background

1. By 1995, ASIC capabilities advanced enough to include a processor core. Thus began the SoC design era.
2. However, the SoC block diagrams looked similar to the system block diagrams.
3. By 2000, technology allowed processor cores to be instantiated in SoCś with increasingly high clock rates.
4. This led to the de-coupling of fast processor subsystems from the slower processor subsystems (slower peripherals). This means there were on-chip bus hierarchies like fast buses and slow buses separated by a bus bridge.

Course Introduction

Historical Background

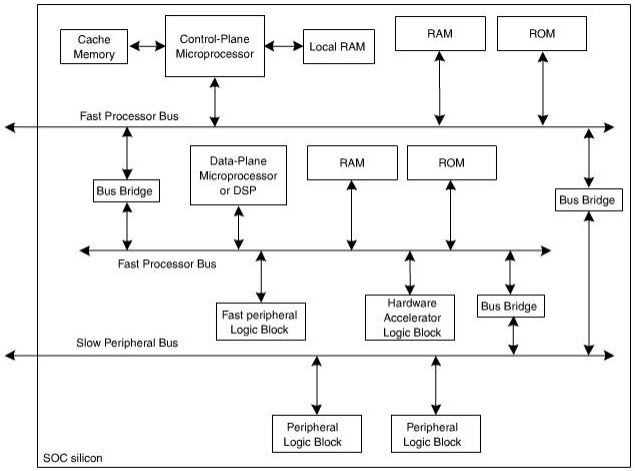


■ **FIGURE 1.14**

By the year 2000, system designers were splitting the on-chip bus into a small, fast processor-memory bus and a slower peripheral bus. A hierarchical bus topology allowed chip designers to greatly reduce the capacitance of the high-speed bus by making it physically smaller. Even so, the SOC's logical block diagram continued to strongly resemble single-processor, board-level systems designed 15 years earlier.

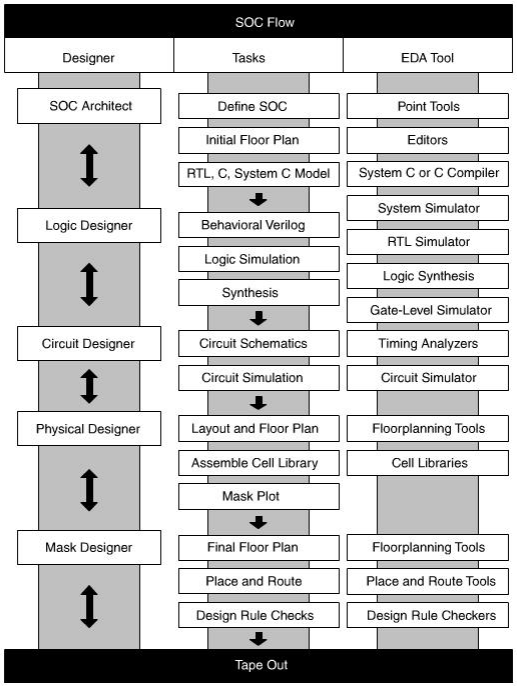
Course Introduction

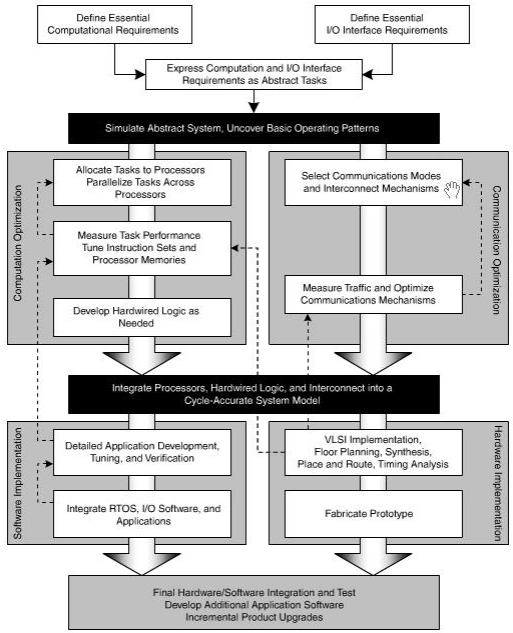
Historical Background → Present Day SoCś



■ **FIGURE 1.15**

Present-day SOC design has started to employ multiple processors instead of escalating clock rates to achieve processing goals.





Course Introduction – Class Example

An Example → Simple 4bit Multiplier

Example of a 4bit Multiplier								
Multiplier	12d				1	1	0	0
Multiplicand	5d				0	1	0	1
					1	1	0	0
				0	0	0	0	.
			1	1	0	0	.	.
		0	0	0	0	.	.	.
Result	60d	0	1	1	1	1	0	0

1. Assume for now that you only have an 8085 processor at 1MHz. How would you code it.
2. Assume you have a 8bit multiplier hardware block, attached to the 8085 processor. How would you code it.
3. What would you do if you do have h/w flexibility, but not as much as a 8bit multiplier block. AND s/w flexibility that it's ok if it is not a single cycle. In other words, its ok to have a multiplier throughput of say 500kHz or 256kHz (two flavors).
4. Assume external h/w also works at 1MHz max.

DATA TRANSFER GROUP	
MOV	Move
MVI	Move Immediate
LDA	Load Accumulator Directly from Memory
STA	Store Accumulator Directly in Memory
LHLD	Load H & L Registers Directly from Memory
SHLD	Store H & L Registers Directly in Memory
An X in the name indicates a register pair (16bits)	
LXI	Load Register Pair with Immediate data
LDAX	Load Accumulator from Address in Register Pair
STAX	Store Accumulator in Address in Register Pair
XCHG	Exchange H & L with D & E
XTHL	Exchange Top of Stack with H & L
Logical Group	
ANA	Logical AND with Accumulator
ANI	Logical AND with Accumulator Using Immediate Dat
ORA	Logical OR with Accumulator
OR	Logical OR with Accumulator Using Immediate Data
XRA	Exclusive Logical OR with Accumulator
XRI	Exclusive OR Using Immediate Data
CMP	Compare
CPI	Compare Using Immediate Data
RLC	Rotate Accumulator Left
RRC	Rotate Accumulator Right
RAL	Rotate Left Through Carry
RAR	Rotate Right Through Carry

Arithmetic Group	
ADD	Add to Accumulator
ADI	Add Immediate Data to Accumulator
ADC	Add to Accumulator Using Carry Flag
ACI	Add Immediate data to Accumulator Using Carry
SUB	Subtract from Accumulator
SUI	Subtract Immediate Data from Accumulator
SBB	Subtract from Accumulator Using Borrow (Carry) Flag
INR	Increment Specified Byte by One
DCR	Decrement Specified Byte by One
INX	Increment Register Pair by One
DCX	Decrement Register Pair by One
DAD	Double Register Add; Add Content of Register Pair to H & L Register Pair
Branch Group	
JMP	Jump
CALL	Call
RET	Return
Conditions for Branching	
NZ	Not Ze 0)
Z	Zero (Z = 1)
NC	No Car 0)
C	Carry (C = 1)
PO	Parity = 0)
PE	Parity = 1)
P	Plus (S = 0)
M	Minus (S = 1)

A B C D E H L	Registers (8-bit)
BC DE HL	Register pairs (16-bit)
PC	Program Counter register (16-bit)
PSW	Processor Status Word (8-bit)
SP	Stack Pointer register (16-bit)

Course Introduction – Class Example

An Example → Simple 4bit Multiplier → HINTS

1. Assume for now that you only have an 8085 processor at 1MHz. How would you code it.
   - Think of left shift multiplicand, conditional add
2. Assume you have a 8bit multiplier hardware block, attached to the 8085 processor. How would you code it.
   - Think of sending operands to hardware block, read results.
   - Cost of Area at the gain of higher throughput.
3. What would you do if you do have h/w flexibility, but not as much as a 8bit multiplier block. AND s/w flexibility that it's ok if it is not a single cycle. In other words, its ok to have a multiplier throughput of say 500kHz or 256kHz (two flavors).
   - Think of sending operands to hardware block, read results.
   - What will reduce Area at the expense of throughput.
   - What is the system requirements, What is the sweet spot.
4. Assume external h/w also works at 1MHz max..
5. Throw power into the equation on a more complex example.

Course Introduction

Acknowledgements

1. Designing SoCs with Configured Cores: Unleashing the Tensilica Xtensa and Diamond Cores :: Steve Leibson :: Ch 1
2. Principles of CMOS VLSI Design - A System Perspective :: N. H. E. Weste and K. Eshranghian :: Multipliers
3. Foils on multiplier implementations :: EE5324 - VLSI Design II :: Kia Bazargan (U. Minnesota)

Course Introduction

Hardware Software CoDesign

September 2011

Agenda

Course Introduction

1. Basic Themes that came out through the Historical Background
2. Basic Pitfalls – what to guard against
3. Problem Statement of a touch screen system

Course Introduction

Basic Themes

1. **Modeling** :: Because parallelism is important, the choice of an appropriate modeling paradigm is also important. Different modeling formalisms need to be developed that capture the various aspects of system behavior.
2. **Analysis and Estimation** :: Different models of the same system behavior need to be analyzed and estimated for performance. Performance would include tradeoffs for Area, Speed, Power. Cost to build, Time to Market are other factors.
3. **System-level partitioning, synthesis and interfacing** :: The basic steps of co-synthesis. A range of methodologies are applied for this.
4. **Implementation generation** :: Once the architecture has been generated and trade-offs known, the designs for the hardware and software components must be created.
5. **Co-simulation and Emulation** :: This helps designers to evaluate architectures and validate assumptions on implementations. Emulation uses FPGA / other emulation techniques to further speed up execution of system models.

Course Introduction

Basic Pitfalls – What to guard against

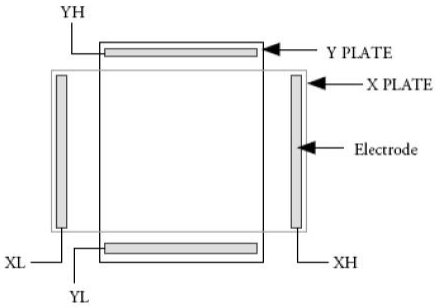
1. **Transient Overloads** :: Transient Overloads on multiple aspects like power, memory, out of bandwidth, data loss, missed deadlines of critical tasks.
2. **Analysis Paralysis** :: Since there are many ways of doing it, there could be too many factors which do not allow the system designer to take decisions. A few ways of taking "judgement" decisions are based on fixing a set of topmost criteria and then working backwards.
3. **Simulation & Complexity** :: Simulation based performance verification, has a conceptual disadvantage that it becomes disabling as complexity increases. There is a great lot of dependence on the pattern provided for finding corner cases.
4. **Applications of the SoC** :: Generally an SoC would be implemented with a certain fixed application in focus. In some cases, the SoC could be used in non-typical or non-designed for applications. This would break some concurrency conditions which were not forseen earlier. Basically, for each new application, one needs to re-run the scenarios in case of using an existing SoC.

Course Introduction

Problem Statement of a touch screen system

Resistive Touch Screen (4-wire)

Resistive touch screens consist of 2 resistive plates that are separated by a small gap. Each plate has an electrode at each end and when the screen is touched, the two plates are shorted together at that point.



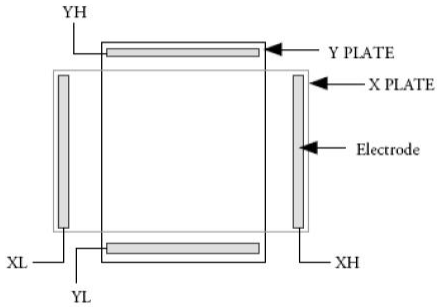
If a voltage is applied, for example, between XL and XH, then a voltage divider is formed on the X PLATE. When the Y PLATE is touched to the X PLATE, a voltage will be developed on the Y PLATE that is proportional to distance of the touch from XL and XH. By accurately measuring this voltage, the position of the touch can be determined.

Course Introduction

Problem Statement of a touch screen system – Requirements

Resistive Touch Screen (4-wire)

Resistive touch screens consist of 2 resistive plates that are separated by a small gap. Each plate has an electrode at each end and when the screen is touched, the two plates are shorted together at that point.



If a voltage is applied, for example, between XL and XH, then a voltage divider is formed on the X PLATE. When the Y PLATE is touched to the X PLATE, a voltage will be developed on the Y PLATE that is proportional to distance of the touch from XL and XH. By accurately measuring this voltage, the position of the touch can be determined.

1. Level 0 :: It should work at lowest cost.

Course Introduction

Problem Statement of a touch screen system – Requirements

1. Level 0 :: It should work at lowest cost.
2. Level 1 :: Some more details :-
   (a) System should continuously monitor for a touch.
   (b) When screen is touched, find out the X and Y coordinates.
   (c) Based on the event(s) of touch and location / pattern, tasks must be performed / activated.

Course Introduction

Problem Statement of a touch screen system – Requirements

1. Level 0 :: It should work at lowest cost.
2. Level 1 :: Some more details :-
   (a) System should continuously monitor for a touch.
   (b) When screen is touched, find out the X and Y coordinates.
   (c) Based on the event(s) of touch and location / pattern, tasks must be performed / activated.
3. Level 2 :: Some more details :-
   (a) Automated Screen Calibration
   (b) Automatically wakes up and goes back to standby to save power
   (c) Simple to write software and drivers
   (d) Programmable conversion rate

Course Introduction

Problem Statement of a touch screen system – Requirements

1. Level 0 :: What are the basic components required
   (a) An ADC which can do a "voltage sensing"
   (b) A uP which can take in the Digital Input from ADC and do processing.
   (c) Some memory to store successive touch(s) and do a pattern recognition !
   (d) A ROM memory to store program, calibration related data.

Course Introduction

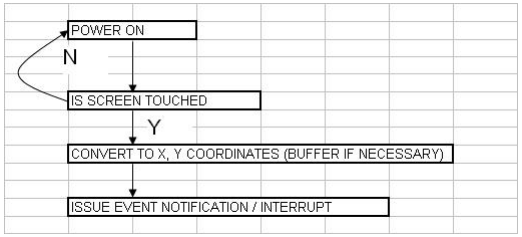
Problem Statement of a touch screen system – Requirements

1. Level 0 :: What are the basic components required
   (a) An ADC which can do a "voltage sensing"
      i. What should be the accuracy, precision of the ADC? 8b, 12b
      ii. Should the ADC be a differential mode or single ended mode.
      iii. What should be the minimum acceptable frequency for the ADC? Depends on how "Human Machine Interface" like touch screen works.
      iv. KBD controller does key debounce at 10 ĩ5ms. Should this be a programmable frequency. In what steps should it be programmable. What complexity will be added to the system. Can we make the system more simple. What is the tradeoff.
   (b) A uP which can take in the Digital Input from ADC and do processing.
   (c) Some memory to store successive touch(s) and do a pattern recognition !
   (d) A ROM memory to store program, calibration related data.

Course Introduction

Problem Statement of a touch screen system – Requirements

1. Level 0 :: What are the basic components required
   (a) An ADC which can do a "voltage sensing"
   (b) A uP which can take in the Digital Input from ADC and do processing.
      i. Write a small flow chart on how the data and control would flow from ADC to uP and control the events based on touch.

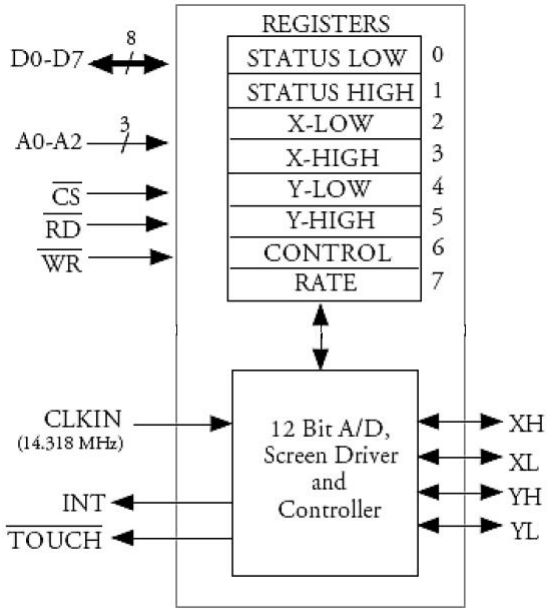


      ii. Write the software as if it was processor agnostic.
      iii. What is the complexity of the program.
      iv. Is an 8bit general purpose processor ok. What will require an upgrade to 16bit or 32bit general purpose processor.
      v. Is there anything in the program which can be hardware accelerated at low cost. Will a DSP suffice for this application.

Course Introduction

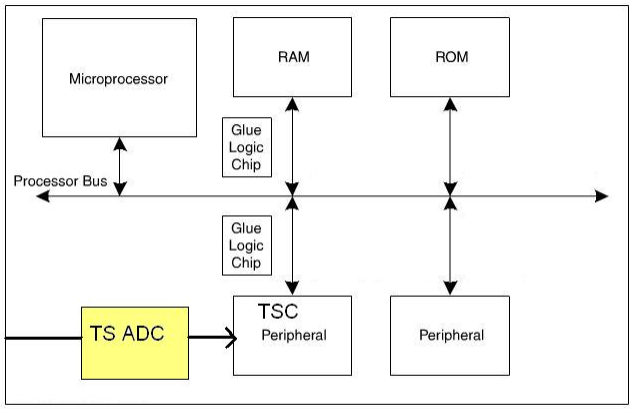
Problem Statement of a touch screen system – Design

1. Level 1 :: What are the basic components which we can fix (pin down)
   (a) The technology node based on requirements
   (b) The ADC
   (c) The uP or DSP
2. Level 2 :: Can we prototype this system using off the shelf components
   (a) Create the IP required to interface ADC with the uP/DSP busses. Add any hardware acceleration required.



Course Introduction

Problem Statement of a touch screen system – Design Block Diagram



Course Introduction

Problem Statement of a touch screen system – Software Design

At the top level, the software provides for the following :-

1. Configure the controller hardware.
2. Determine if the screen is touched.
3. Acquire Stable, debounced position measurements.
4. Calibrate the touch screen.
5. Send changes in touch status and position to the higher level graphics software.

Course Introduction

Problem Statement of a touch screen system – Software Design

Configure the controller hardware.

1. Create a function named `TouchConfigureHardware()`
2. Decide – Should the driver be interrupt driven or polling driven.
3. In case of interrupts, the driver would actually use two :-
   (a) An interrupt to wake up when the screen is initially touched, known as the `PEN_DOWN` interrupt.
   (b) A second interrupt to signal when the ADC is available with the set of data conversions (X, Y).

Determine if the screen is touched.

1. Create a function named `WaitForTouchState ()`
2. When the controller is in the detection mode and a touch is detected, an internal interrupt can be generated called `PEN_DOWN IRQ`.
3. This detection is based on Y-axis touch plane tied high, X-axis touch plane tied low, and on the basis of touch the planes are shorted together and Y-axis plane is pulled low.
4. The driver task would not consume any CPU time until the `PEN_DOWN IRQ` event occurs. It would wake up and go into conversion mode only once the user touches screen.

Course Introduction

Problem Statement of a touch screen system – Software Design

Acquire Stable, debounced position measurements – Reading touch data

1. Create a function named `TouchScan()`. The outline of the procedure would be :-
   (a) Check to see if the screen is touched.
   (b) Take several raw readings on each axis for later filtering.
   (c) Check to see if the screen is still touched.
   (d) (Depending on H/W) store the readings to a memory block, or use FIFO entries.
   (e) (Depending on H/W) if the ADC output is 12bit, either pack data into 16bit (aligned) locations or chop/dither them to 8bit.
   (f) Taking Stable readings (filtering by using oversampling) is necessary for higher level drivers to act appropriately. NOTE :: This filtering could be a h/w function if CPU bandwidth is critical factor.

Calibrate the touch screen.

1. In an ideal scenario, the calibration would be run once during initial product power up and reference values saved in "non-volatile" memory.
2. Create a function name `CalibrateTouchScreen ()` in case the user wants to calibrate using a graphical target on screen.

Course Introduction

Problem Statement of a touch screen system – Software Design

Send changes in touch status and position to the higher level graphics software.

1. Create a function named `GetScaledTouchPosition()`. This is a routine to read raw values and convert them to screen co-ordinates.
2. Create a function named `TouchTask ()`. This routine calls the actual task the user intended to be run while using the touch screen.

Course Introduction

Acknowledgements

1. Datasheet of IDT MK712 Touch Screen Controller
2. Application Bulletin "Burr Brown" now TI (public domain information) :: Touch Screen Controller Tips.
3. http://www.eetimes.com/design/embedded/4006455/Writing-drivers-fo common-touch-screen-interface-hardware

Modeling

Hardware Software CoDesign

September 2011

Agenda

Modeling

1. Basic Themes that came out through the Historical Background (repeat)
2. Basic Modeling Requirements
3. SystemC Language based Modeling

Modeling

Basic Themes

1. **Modeling** :: Because parallelism is important, the choice of an appropriate modeling paradigm is also important. Different modeling formalisms need to be developed that capture the various aspects of system behavior.
2. **Analysis and Estimation** :: Different models of the same system behavior need to be analyzed and estimated for performance. Performance would include tradeoffs for Area, Speed, Power. Cost to build, Time to Market are other factors.
3. **System-level partitioning, synthesis and interfacing** :: The basic steps of co-synthesis. A range of methodologies are applied for this.
4. **Implementation generation** :: Once the architecture has been generated and trade-offs known, the designs for the hardware and software components must be created.
5. **Co-simulation and Emulation** :: This helps designers to evaluate architectures and validate assumptions on implementations. Emulation uses FPGA / other emulation techniques to further speed up execution of system models.

Modeling

Basic Modeling Requirements

1. **Concurrency** :: H/W :: The ability to have multiple processes run simultaneously.
2. **State Transitions** :: H/W & S/W :: Conceptualize the system into different modes or states of operation (behavior). The states change based on input and the state transitions imply certain output characterestics.
3. **Hierarchy** :: H/W :: Conceptualize a system into building blocks or subsystems.
4. **Programing Constructs** :: S/W :: The ability to describe sequential algorithms.
5. **Exception Handling** :: H/W & S/W :: The ability to respond to events external and internal.
6. **Timing** :: H/W & S/W :: The ability to create a protocol and model it not only behavioraly but also by specifying a timing constraint or delay in signals.
7. **Communication** :: H/W & S/W :: The ability to describe a channel of communication. It could be as complete as a protocol, or as behaviroral as just transfer of data through a global variable.
8. **Process Synchronization** : H/W & S/W :: The ability to generate data and events to be used by other processes.

Modeling

Basic Modeling Requirements – Concurrency

1. In computer science, concurrency is a property of systems in which several computations are executing **simultaneously**, and potentially interacting with each other.

2. From the point of System level modeling, one can further divide concurrency into different types based on concepts of hardware modeling.

   (a) **Data-driven Concurrency** :: As in a data flow graph (DFG) model, execution depends on availability of data.

   (b) **Control-driven Concurrency** :: There are constructs that specify multiple threads of control, each of which can execute in parallel. eg. fork-join.

   (c) **Pipeline-driven Concurrency** :: Specific to signal processing and driven by throughput / latency requirements of systems.

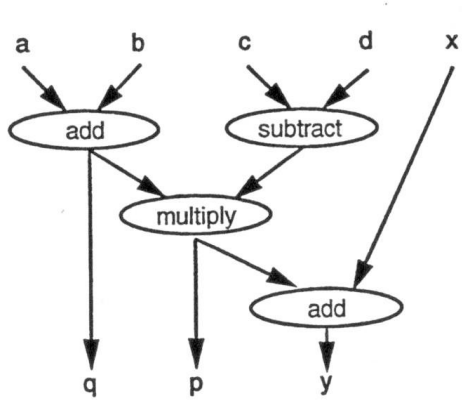
3. Concurrency can occur at Task level, Statement level or Operation level.

Modeling

Basic Modeling Requirements – Concurrency - Data-Driven

1: q = a + b
2: y = x + p
3: p = (c – d) * q

(a)



(b)

Modeling

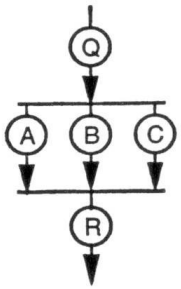
Basic Modeling Requirements – Concurrency - Control-Driven

```
sequential behavior X
begin
   Q();
   fork  A(); B(); C(); join;
   R();
end behavior X;
```

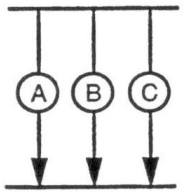
(a)

```
concurrent behavior X
begin
   process A();
   process B();
   process C();
end behavior X;
```

(b)



(c)

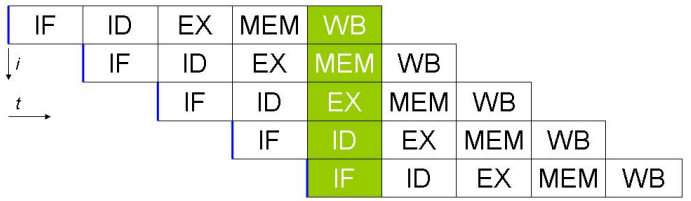


(d)

Modeling

Basic Modeling Requirements – Concurrency - Pipeline-Driven

Below is example of a typical five-stage pipeline in a RISC machine.



1. IF = InstructionFetch
2. ID = InstructionDecode
3. EX = EXecute
4. MEM = MEMory access
5. WB = register WriteBack

Modeling

Basic Modeling Requirements – State Transitions

1. Systems are best conceptualized as having various modes of operation or states, of behavior.
2. State transitions are triggered by the detection of certain events or certain conditions.
3. In software :: Would an `if-then-else` indicate a state transition?
4. In software :: Would a `goto` statement indicate a state transition?
5. Actions can be associated with each transition, and a particular mode or state can have an arbitrarily complex behavior or computation associated with it.
6. In case of the touchScreen example, the detection of a touch, followed by the invokation of the said task is an event driven State Transition.

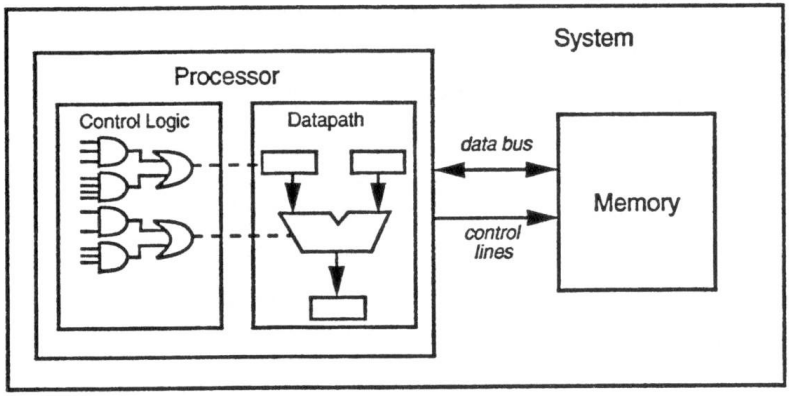
Modeling

Basic Modeling Requirements – Hierarchy

1. Hierarchical models enable system to become modular and one can look at it as a set of smaller subsystems.
2. These subsystems further down can be looked upon as another system with its own characterestics.
3. This conceptual view helps development, due to inherent modularity and making a complex system look as blocks of easily understood subsystems.
4. Hierarchy can be used to scope objects.
5. What would happen if there is no hierarchy in hardware or software?
6. Hierarchy can be of two types basically coming from the h/w & s/w backgrounds :: Structural hierarchy & Behavioral hierarchy

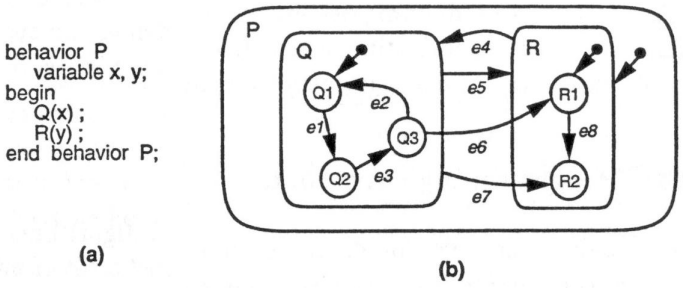
Modeling

Basic Modeling Requirements – Hierarchy - Structural



Modeling

Basic Modeling Requirements – Hierarchy - Behavioral



1. Is defined as the process of decomposing a behavior into distinct sub-behaviors, which can be either sequential or concurrent.
2. Behavioral hierarchy can further be of different types like sequential, parallel, or pipelined.
3. The Transitions can be simple (as within the blocks Q, R)
4. The Transitions can be grouped (as e4, e5) which looks like a request, grant pair.
5. The Transitions can be hierarchical (as e6, e7)

Modeling

Basic Modeling Requirements – Programing Constructs

1. Many behaviors may be best described using programming constructs : it allows the computations to be expressed explicitly.
2. Behaviors can be described as Sequential Algorithms
3. Typically, the construct includes assignment statements, branching statements, iteration statements and procedures.
4. In addition, data types are used like records (pascal?), arrays, linked lists and other complex data structures (or structure of structures).

```
type      buffer_type  is array (1 to 10) of integer;
variable  buf : buffer_type;
variable  i, j : integer;

for i = 1 to 10
    for j = i to 10

        if (buf(i) > buf(j)) then
            Swap(buf(i), buf(j));
        end if;

    end for;
end for;
```

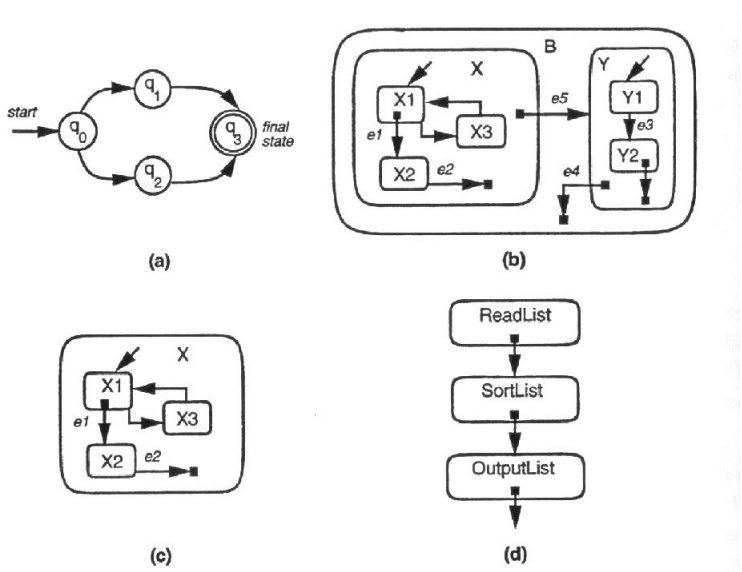

Modeling

Basic Modeling Requirements – Behavioral Completion

1. Ability of the behavior to indicate that it has completed.
2. Other behaviors should be able to detect it.
3. Behavioral completion is required because :-
   (a) In hierarchical specification, it allows to view a subsystem as an independent module and allows for its independent developement, once the boundaries (input, output) are specified
   (b) Allows natural breakup of system behavior into sub-behaviors which can then be sequenced by the completion of transition arcs.
4. Behavioral Completion are represented by
   (a) Final state(s) in a FSM.
   (b) return statement in procedures.

Modeling

Basic Modeling Requirements – Behavioral Completion



Modeling

Basic Modeling Requirements – Exception Handling

1. Occurance of a certain event can require that a behavior or mode be interrupted immediately, thus prohibiting the behavior from updating the values further. The NEXT behavior is specified explicitly.
2. It sometimes becomes crucial that occurance of an event should terminate current behavior immediately.
3. FSMs assume zero time computation in a state – hence an exception activates an appropriate transition.
4. Depending on the direction of the transferred control Exceptions can be further divided into :-
   (a) **Abort** :: the current behavior is terminated completly, and control transferred to event handling.
   (b) **Interrupt** :: the current behavior is context-switched, control transferred to event handling, and once the event handling is done, behavior is again context-switched to the completion of the previous tasks (from where it was interrupted).

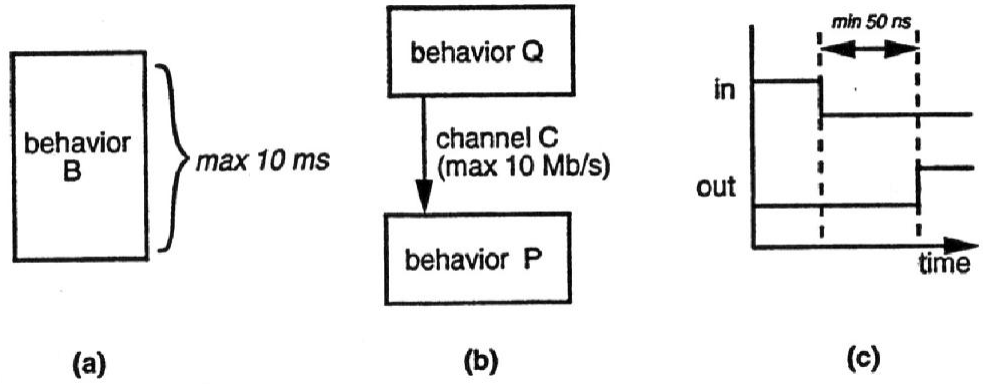
Modeling

Basic Modeling Requirements – Timing

1. Timing constraints denote performance constraints, to be used by synthesis tools.
2. It can be checked through verification tools.
3. In general, a timing relation can be described as $(e1, e2, min, max)$ where $e1$ preceeds $e2$ by at least min time units and at most max time units.
4. The different types of timing constraint are :-
   (a) Execution time constraints
   (b) Data transfer **rate** constraints
   (c) Inter event timing constraints
5. The timing information is important in **real time** embedded systems, whose performance is measured in terms of how well the implementation respects the timing constraints
6. From a specification point of view, constraints need to be specified as :-
   (a) Timing diagrams for IO event constraints
   (b) Statement labels for statement level constraints

Modeling

Basic Modeling Requirements – Timing



Modeling

Basic Modeling Requirements – Communication

1. Systems consist of several interacting behaviors which need to communicate with each other.
2. Traditionally (say the C programming language) Communication was through
   (a) For Functions :: communicate through global variables (or ports) which share the same memory space or via parameter passing.
   (b) For Local procedure calls :: information exchange is through a stack or through processor registers.
   (c) For Remote Procedure calls (RPC) :: parameters are passed through a complex protocol of sending/receiving data through network.
3. For embedded systems more is required in the form of :-
   (a) Separation of description of computation and communication
   (b) Declaration of abstract communication functions
   (c) Definition of a custom communication implementation
4. Crystalize these requirements into two basic types of communication
   (a) Shared Memory Model
   (b) Message Passing (Channel) Model

Modeling

Basic Modeling Requirements – Communication – Shared Memory Model

1. Uses shared medium such as global variable or port
2. Synchronization must be explicit like flag setting
3. May be broadcast mode – change sensed by all
4. Persistent mode – registered
5. Non Persistent – wired only (data is available only at the instant when it is written)

Modeling

Basic Modeling Requirements – Communication – Message Passing (Channel) Model

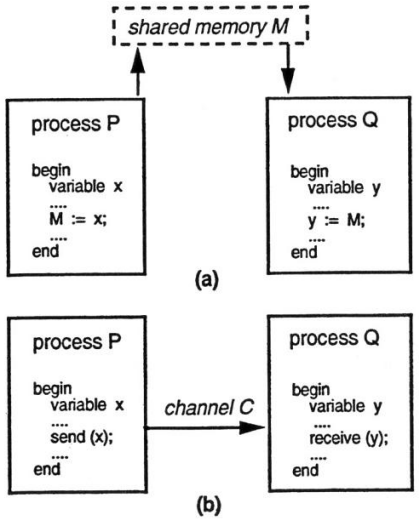
1. Messages sent through channels (an abstract medium, free from implementation details)
2. Requires send and receive primitives
3. "Send" destination may be explicit or implicit (specified by the interconnection)
4. Uni/Bi directional
5. Point to Point or multiway (more than two processes may communicate)
6. Communication can be blocking or non-blocking :-
   (a) In Blocking Mode :: The sending process blocks itself until the receiving process is ready to receive data.
   - No extra storage required for storing data
   - Forced synchronization
   - Degrades performance

   (b) In Non-Blocking Mode :: storage is associated with the channel
   - For insufficient queue length, process may still block

7. A channel itself can be hierarchical. A channel may implement a high level protocol which breaks a stream of data packets into a byte stream, and in turn uses a lower level channel (eg. a synchronous bus, which transfers the byte stream one bit at a time)

Modeling

Basic Modeling Requirements – Communication



Modeling

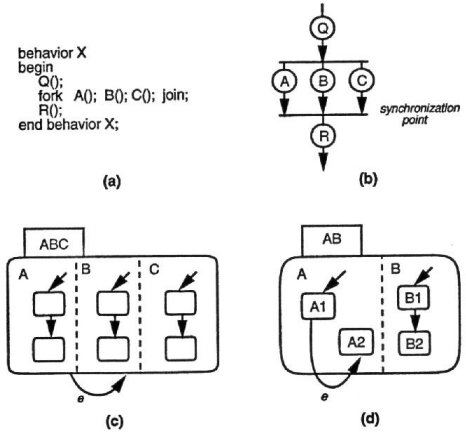
Basic Modeling Requirements – Process Synchronization

1. Concurrent processes may generate data and events that need to be recognised by ohter processes. Hence Synchronization is a necessary requirement.
2. There are two basic types of Synchronization :-
   (a) Control Dependent Synchronization
   (b) Data Dependent Synchronization

Modeling

Basic Modeling Requirements – Process Synchronization – Control Dependent

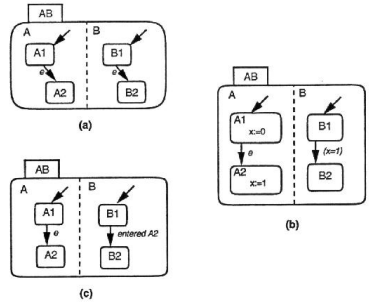
1. Control structure is responsible for synchronization eg. fork-join
2. Initialization



Modeling

Basic Modeling Requirements – Process Synchronization – Data Dependent

1. Synchronized through Shared Memory
   - One process is suspended till the other updates the shared memory to a particular value.
   - Synchronization through a common event (a data value or event)
   - Synchronization through common data (variable)
   - Synchronization through status detection
2. Synchronized through Message Passing (blocking)



Modeling

Basic Modeling Requirements – Summary

1. State Transitions
2. Behavior Hierarchy
3. Concurrency
4. Exception Handling
5. Programming constructs
6. Behavior completion
7. Timing representation

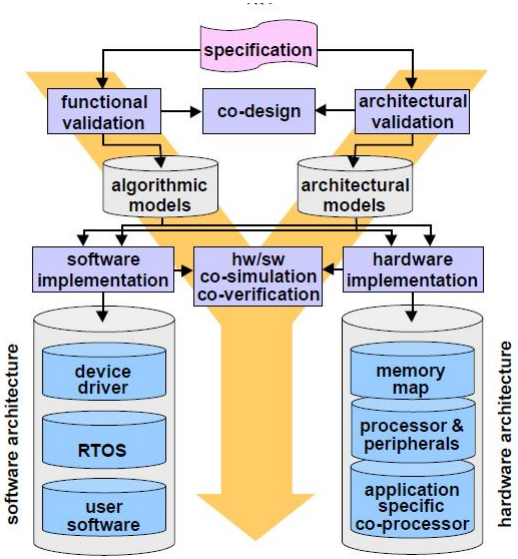
Modeling

Modeling Using SystemC

LOGICAL BREAK

Modeling

Modeling Using SystemC – (Restate) System Level Design Flow



Modeling

Modeling Using SystemC – Benefits of a C/C++ Based Design Flow

1. Productivity aspect
   - Specification between architect and implementer is executable
   - High speed and High Level simulation and prototyping capibility
   - Refinement, no translation into hardware (i.e, no "semantic gap")
2. System level aspect
   - Tomorrowś systems designers will be designing mostly software and less hardware (these are Joachim Gerlachś view and motivation factor)
   - Co-design, co-simulation, co-verification, co-debugging, co-....
3. Re-use aspect
   - Optimum re-use support by "Object Oriented" techniques
   - Efficient testbench re-use (for both software, hardware and system)
4. Especially C/C++ is widespread and commonly used

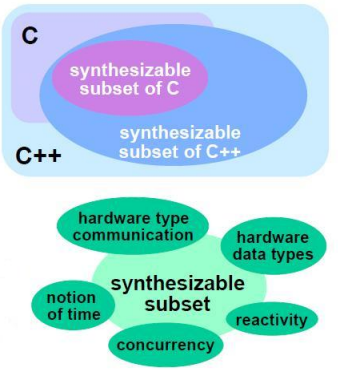
Modeling

Modeling Using SystemC – **Drawbacks** of a C/C++ Based Design Flow

1. C/C++ is `not` created to design hardware !!
2. C/C++ does `not` support
   - Hardware style communication :: Signals, protocols
   - Notion of time :: Clocks, time sequenced operations
   - Concurrency :: Hardware is inherently concurrent, operates in parallel
   - Reactivity :: Hardware responds to stimulii, interacts with its environment (→ requires handling of exceptions)
   - Hardware data types :: bit, bit-vector, multi-valued logic, signed and unsigned integer types, fixed-point types

Modeling

Modeling Using SystemC – How does one get there using C/C++



1. Restrict to synthesizable subset
2. Extend the language with hardware related components
3. The major requirements which need to be satisfied are :-

- Allow hardware/software co-design and co-verification
- Fast simulation for validation and optimization
- Smooth path to hardware and software
- Support of design and architectural reuse

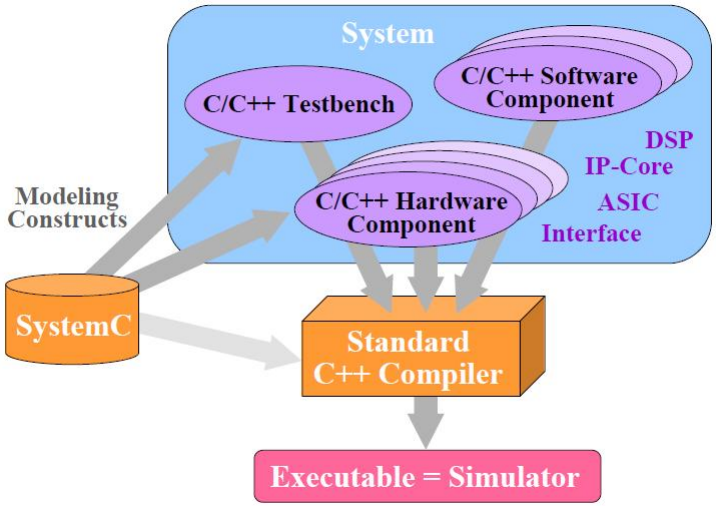
Modeling

Modeling Using SystemC – What is SystemC

1. A library of C++ classes
   - Processes (for concurrency)
   - Clocks (for time)
   - Modules, ports, signals (for hierarchy)
   - Waiting, watching (for events)
   - Hardware data types
2. A modeling style :: for modeling systems consisting of multiple design domains, abstraction levels, architectural components, real-life constraints
3. A light weight simulation kernel
   - Allow hardware/software co-verification

Modeling

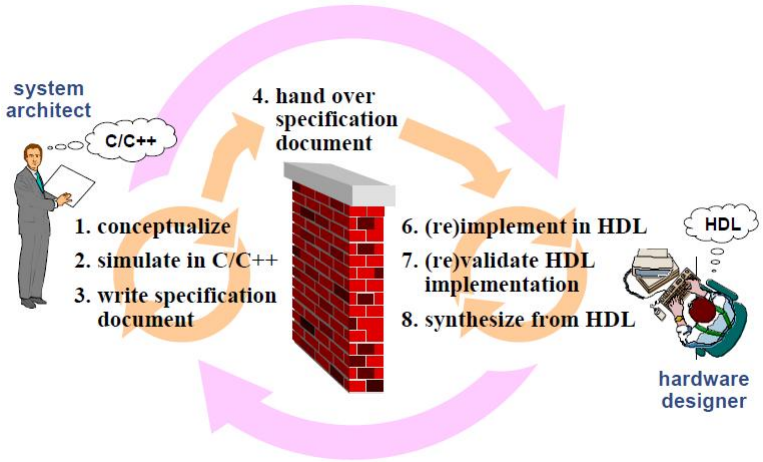
Modeling Using SystemC – How Does SystemC Work?



1. The approach is to promote a standard C/C++ modeling platform to model and exchange system level components and IP as also to build inter-operable tools infrastructure.

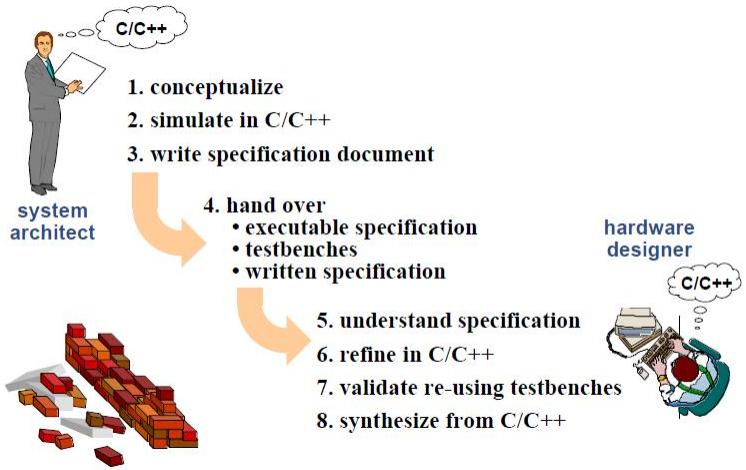
Modeling

Modeling Using SystemC – How Did it Work earlier ?



Modeling

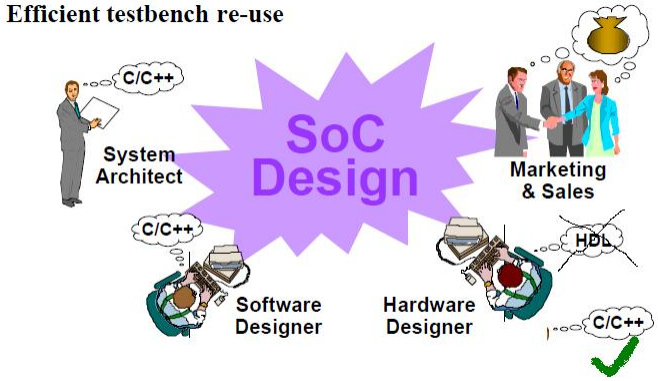
Modeling Using SystemC – What is the new Methodology?



1. The approach is to promote a standard C/C++ modeling platform to model and exchange system level components and IP as also to build inter-operable tools infrastructure.

Modeling

Modeling Using SystemC – What is the new Methodology?

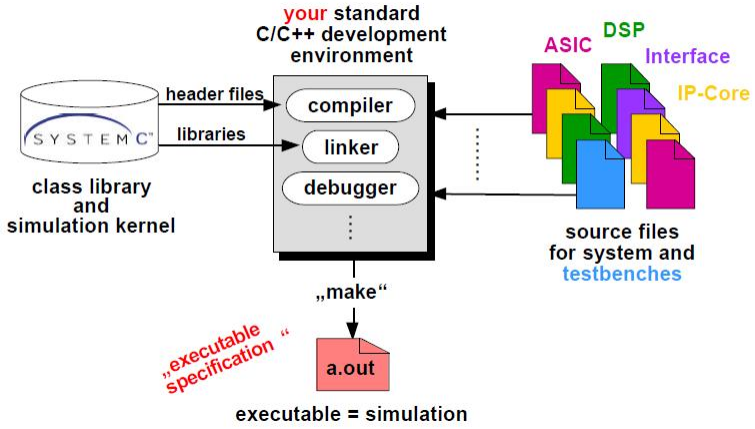


1. **Everyone talks the same language**

2. The approach is to promote a standard C/C++ modeling platform to model and exchange system level components and IP as also to build inter-operable tools infrastructure.

Modeling

Modeling Using SystemC – SystemC Design Methodology



Modeling

Modeling Using SystemC – Modules

1. Modules are basic building blocks of a SystemC design

2. A module contains processes (functionality) and / or sub-modules (structural hierarchy)

```
SC_MODULE( module_name ) {
    // Declaration of module ports
    // Declaration of module signals
    // Declaration of processes
    // Declaration of sub-modules
    SC_CTOR( module_name ) {        // Module constructor
        // Specification of process type and sensitivity
        // Sub-module instantiation and port mapping
    }
    // Initialization of module signals
};
```

3. A module corresponds to a C++ class
   - class data members :: ports
   - class member functions :: processes
   - class constructor :: process generation

Modeling

Modeling Using SystemC – Ports

1. Ports are the external interface(s) of a module
2. Passes data from and to processes / sub-modules
3. Used to trigger actions within the module
4. A port has a **mode** (direction) and a **type** (in, out, inout)

type: C++ type, SystemC type, user-defined type

```
// input port declaration
sc_in< type >  in_port_name;

// output port declaration
sc_out< type >  out_port_name;

// bidirectional port declaration
sc_inout< type >  inout_port_name;
```

Vector port / port array:

```
sc_out< int > result [32];
```


Modeling

Modeling Using SystemC – Signals

1. Signals connect port of one module to the port of another module
2. Signals are local to a module
3. Signals semantics are similar to VHDL and Verilog assignment semantics
4. A signal has a **type**

type: C++ type, SystemC type, user-defined type

```
// signal declaration
sc_signal< type >  signal_name;
```

Vector signal / signal array:

```
sc_signal< double > a[4];
```

5. NOTE :: Internal Storage of data is through local variables (not signals)

Modeling

Modeling Using SystemC – Ports & Signals Binding

1. Ports and Signals to be bound need to have the same type
2. A signal connects two ports
3. A port is bound to one signal (port-to-signal) or to one sub-module port (port-to-port)

Modeling

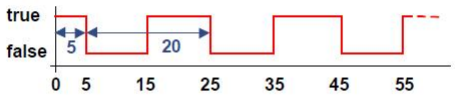
Modeling Using SystemC – Clocks

1. SystemC provides a special object `sc_clock`
2. Clocks generate timing signals to synchronize events
3. Multiple clocks with arbitrary phase relationships are supported
4. Clock generation & binding

```
sc_clock clock_name ("label", period, duty_ratio, offset, start_value);
```

Example:

```
sc_clock my_clk ("CLK", 20, 0.5, 5, true);
```



binding:

Example:

```
my_module.clk( my_clk.signal() );
```

Modeling

Modeling Using SystemC – Data Types

1. SystemC supports
   - Native C/C++ types
   - SystemC types
   - User-defined types
2. SystemC types
   - 2-value ('0', '1') logic / logic vector
   - 4-value ('0', '1', 'Z', 'X') logic / logic vector
   - Arbitrary sized integer (signed / unsigned)
   - Fixed point types (signed/unsigned, templated/untemplated)

Modeling

Modeling Using SystemC – Data Types

1. SystemC types

Type	Description
sc_bit	2-value single bit
sc_logic	4-value single bit
sc_int	1 to 64 bit signed integer
sc_uint	1 to 64 bit unsigned integer
sc_bigint	arbitrary sized signed integer
sc_biguint	arbitrary sized unsigned integer
sc_bv	arbitrary length 2-value vector
sc_lv	arbitrary length 4-value vector
sc_fixed	templated signed fixed point
sc_ufixed	templated unsigned fixed point
sc_fix	untemplated signed fixed point
sc_ufix	untemplated unsigned fixed point

2. Native C/C++ types
   - Integer types :: char, unsigned char, short, unsigned short, int, unsigned int long, unsigned long
   - Floating point types :: float, double, long double

Modeling

Modeling Using SystemC – Data Types

1. sc_bit :: 2-value single bit type :: '0' = false, '1' = true
2. sc_logic :: 4-value single bit type :: '0' = false, '1' = true, 'X' = unknown / indeterminate value, 'Z' = high-impedence / floating value
3. SystemC allows for mixed use of operand types sc_bit and sc_logic. Use of character literals for constant assignments is allowed.

sc_bit / sc_logic operators

Bitwise	& (and)	| (or)	^ (xor)	~ (not)
Assignment	=	&=	|=	^=
Equality	==	!=		

Modeling

Modeling Using SystemC – Data Types

1. sc_int$< n >$ :: Signed Fixed point integer type :: $(n : wordLength, 1 \leq n \leq 64)$
2. sc_uint$< n >$ :: UnSigned Fixed point integer type :: $(n : wordLength, 1 \leq n \leq 64)$
3. sc_bigint$< n >$ :: Signed Fixed point integer type :: $(n : wordLength, n \geq 64)$
4. sc_biguint$< n >$ :: UnSigned Fixed point integer type :: $(n : wordLength, n \geq 64)$
5. Features ::
   - Mixed use of operand types sc_int, sc_uint, sc_bigint, sc_biguint and C++ integer types
   - Truncation and / or sign extension can be done if required
   - 2ś complement representation

sc_int / sc_uint / sc_bigint / sc_biguint operators

Bitwise	&	|	^	~	>>	<<			
Arithmetic	+	-	*	/	%				
Assignment	=	+=	-=	*=	/=	%=	&=	|=	^=
Equality	==	!=							
Relational	<	<=	>	>=					
Auto-Ink/Dek	++	--							
Bit/Part Select	[]	range()							
Concatenation	(,)								

Modeling

Modeling Using SystemC – Data Types

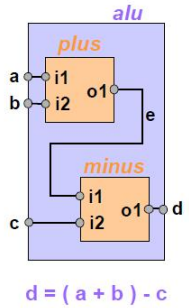
1. sc_bv< n > :: Aribitrary length bit vector :: ($n : vectorLength$)
2. sc_lv< n > :: Aribitrary length logic vector :: ($n : vectorLength$)
3. Features ::
   - Assignment between sc_bv and sc_lv
   - Use of string literals for vector constant assignments
   - Conversions between sc_bv/ sc_lv ans SystemC integer types
   - No arithmetic operation available

sc_bv / sc_lv								
Bitwise	&	|	^	~	>>	<<		
Assignment	=	+=	-=	*=	/=	%=	&=	|=	^=
Equality	==	!=						
Bit/Part Select	[]	range()						
Concatenation	(,)							
Reduction	and_reduction()	or_reduction()	xor_reduction()					
Conversion	to_string()							

Modeling

Modeling Using SystemC – Modules and Hierarchies

1. It is possible to have a module containing sub-modules to form an hierarchical structure.




```
SC_MODULE( plus ) {
  sc_in<int>  i1;
  sc_in<int>  i2;
  sc_out<int> o1;
  .....
};

SC_MODULE( minus ) {
  sc_in<int>  i1;
  sc_in<int>  i2;
  sc_out<int> o1;
  .....
};
```

```
SC_MODULE( alu ) {
  sc_in<int>  a;
  sc_in<int>  b;
  sc_in<int>  c;
  sc_out<int> d;

  plus  *p;
  minus *m;

  sc_signal<int> e;

  SC_CTOR( alu ) {
    p = new plus ( "PLUS" );
    p->i1 (a);
    p->i2 (b);
    p->o1 (e);

    m = new minus ( "MINUS" );
    (*m) (e,c,d);
  }
};
```


Modeling

Modeling Using SystemC – Processes

1. Process
   - Encapsulates functionality
   - Basic unit of concurrent execution
   - Not Hierarchical
2. Process Activation
   - Processes have sensitivity lists
   - Processes are triggered by events on sensitive signals
3. Process Types
   - Method (SC_METHOD) :: asynchronous block, like a sequential function
   - Thread (SC_THREAD) :: asynchronous process
   - Clocked Thread (SC_CTHREAD) :: synchronous process

	SC_METHOD	SC_THREAD	SC_CTHREAD
triggered	by signal events	by signal events	by clock edge
infinite loop	no	yes	yes
execution suspend	no	yes	yes
suspend & resume	-	wait()	wait() wait_until()

construct & sentisize method	SC_METHOD(*p*);
	
	sensitive(*s*);
	sensitive_pos(*s*);
	sensitive_neg(*s*);
		SC_THREAD(*p*);
		
		sensitive(*s*);
		sensitive_pos(*s*);
		sensitive_neg(*s*);
			SC_CTHREAD (*p*,*clock*.pos());
			
			SC_CTHREAD (*p*,*clock*.neg());

modeling example (hardware)	combinational logic	sequential logic at RT level (asynchronous reset, etc.)	sequential logic at higher design levels

Modeling

Modeling Using SystemC – Processes

Example: SC_METHOD

```
SC_MODULE( plus ) {
  sc_in<int> i1;
  sc_in<int> i2;
  sc_out<int> o1;

  void do_plus();

  SC_CTOR( plus ) {
    SC_METHOD( do_plus );
    sensitive << i1 << i2;
  }
};
```

```
void plus::do_plus() {
  int arg1;
  int arg2;
  int sum;

  arg1 = i1.read();
  arg2 = i2.read();
  sum = arg1 + arg2;
  o1.write(sum);
}
```

```
void plus::do_plus() {
  o1 = i1 + i2;
}
```


Modeling

Modeling Using SystemC – Processes

Example: SC_THREAD

```
SC_MODULE( plus ) {
  sc_in<int> i1;
  sc_in<int> i2;
  sc_out<int> o1;

  void do_plus();

  SC_CTOR( plus ) {
    SC_THREAD( do_plus );
    sensitive << i1 << i2;
  }
};
```

```
void plus::do_plus() {
  int arg1;
  int arg2;
  int sum;

  while ( true ) {
    arg1 = i1.read();
    arg2 = i2.read();
    sum = arg1 + arg2;
    o1.write(sum);

    wait();
  }
}
```


Modeling

Modeling Using SystemC – Processes

Example: SC_CTHREAD

```
SC_MODULE( plus ) {
  sc_in_clk  clk;

  sc_in<int> i1;
  sc_in<int> i2;
  sc_out<int> o1;

  void do_plus();

  SC_CTOR( plus ) {
    SC_CTHREAD( do_plus, clk.pos() );
  }
};
```

```
void do_plus() {
  int arg1;
  int arg2;
  int sum;

  while ( true ) {
    arg1 = i1.read();
    arg2 = i2.read();
    sum = arg1 + arg2;
    o1.write(sum);

    wait();
  }
}
```


Modeling

Modeling Using SystemC – Waiting and Watching

1. Suspend / reactivate process execution :: `wait()` :: SC_THREAD, SC_CTHREAD
2. Halt process execution until an event occurs :: `wait_until()` :: SC_CTHREAD ONLY
3. Transfer control to a special code sequence if a specific condition occurs :: `watching(reset.delayed() == true)`

Modeling

Modeling Using SystemC – Simulation Kernel Scheduler Steps

Step 1: All clock signals that change their valueat the current time are assigned their new value.

Step2: All *SC_METHOD* / *SC_THREAD* processes with inputs that have changed are executed. The entire bodies of *SC_METHOD* processes are executed. *SC_THREAD* processes are executed until the next *wait()* statement suspends execution. *SC_METHOD* / *SC_THREAD* processes are not executed in a fixed order.

Step3: All *SC_CTHREAD* processes that are triggered have their outputs updated and are saved in a queue to be executed in step 5. All outputs of *SC_METHOD* / *SC_THREAD* processes that were executed in step 1 are also updated.

Step 4: Step 2 and step 3 are repeated until no signal changes ist value.

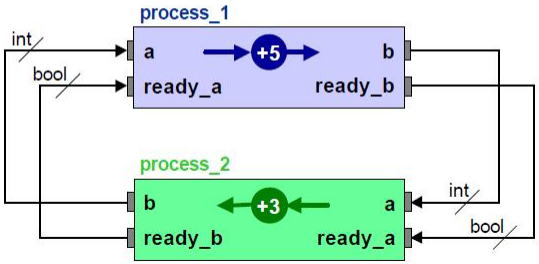
Step 5: All *SC_CTHREAD* processes that were triggered and queued in step 3 are executed. There is no fixed execution order of these processes. Their outputs are updated at the next active edge (when step 3 is executed), and therefore are saved internally.

Step6: Simulation time is advanced to the next clock edge and the scheduler goes back to step 1.

Modeling

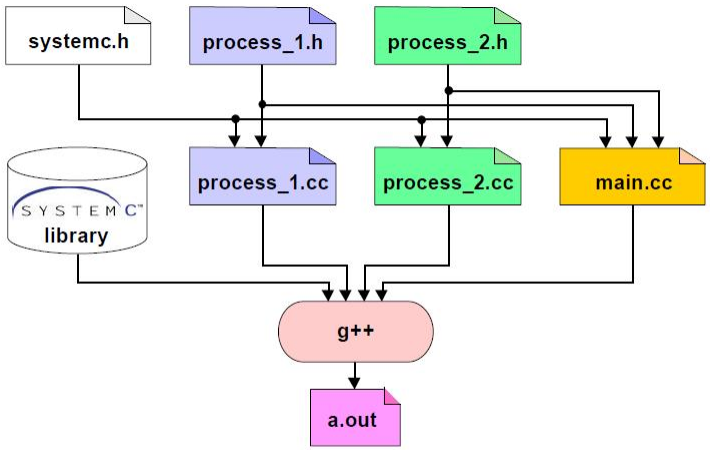
Modeling Using SystemC – Example

Two processes (process_1 and process_2) alternately incrementing an integer value



Modeling

Modeling Using SystemC – Example – Typical File Structure



Modeling

Modeling Using SystemC – Example

```
// header file: process_1.h
SC_MODULE( process_1 ) {

    // Ports
    sc_in_clk clk;
    sc_in<int> a;
    sc_in<bool> ready_a;
    sc_out<int> b;
    sc_out<bool> ready_b;

    // Process functionality
    void do_process_1();

    // Constructor
    SC_CTOR( process_1 ) {
        SC_CTHREAD( do_process_1 , clk.ps() );
    }

};
```

Module: process_1

```
// implementation file: process_1.cc
#include "systemc.h"
#include "process_1.h"

void process_1::do_process_1()
{
    int v;

    while ( true )
    {
        wait_until( ready_a.delayed() == true );
        v = a.read();
        v += 5;
        cout << "P1: v = " << v << endl;
        b.write( v );

        ready_b.write( true );
        wait();
        ready_b.write( false );
    }
}
```

Modeling

Modeling Using SystemC – Example

```
// header file: process_2.h
SC_MODULE( process_2 ) {

    // Ports
    sc_in_clk clk;
    sc_in<int> a;
    sc_in<bool> ready_a;
    sc_out<int> b;
    sc_out<bool> ready_b;

    // Process functionality
    void do_process_2();

    // Constructor
    SC_CTOR( process_2 ) {
        SC_CTHREAD( do_process_2 , clk.ps() );
    }

};
```

Module: process_2

```
// implementation file: process_2.cc

#include "systemc.h"
#include "process_2.h"

void process_2::do_process_2()
{
    int v;

    while ( true )
    {
        wait_until( ready_a.delayed() == true );
        v = a.read();
        v += 3;
        cout << "P2: v = " << v << endl;
        b.write( v );

        ready_b.write( true );
        wait();
        ready_b.write( false );
    }
}
```


Modeling

Modeling Using SystemC – Example

```
// implementation file: main.cc

#include "systemc.h"
#include "process_1.h"
#include "process_2.h"

int  sc_main  (int ac,char *av[])
{
    sc_signal<int> s1 ( "Signal-1" );
    sc_signal<int> s2 ( "Signal-2" );
    sc_signal<bool> ready_s1 ( "Ready-1" );
    sc_signal<bool> ready_s2 ( "Ready-2" );

    sc_clock clock( "Clock" , 20 , 0.5 , 0.0 );

    process_1 p1 ( "P1" );
    p1.clk( clock );
    p1.a( s1 );
    p1.ready_a( ready_s1 );
    p1.b( s2 );
    p1.ready_b( ready_s2 );
```

Top-Level Module: main

```
    process_2 p2 ( "P2" );
        p2.clk( clock );
        p2.a( s2 );
        p2.ready_a( ready_s2 );
        p2.b( s1 );
        p2.ready_b( ready_s1 );

        s1.write(0);
        s2.write(0);
        ready_s1.write(true);
        ready_s2.write(false);

        sc_start(100000);

        return 0;
}
```


Modeling

Acknowledgements

1. Hardware / Software Co-Design - Principles and Practice :: J. Staunstrup & W. Wolf :: Ch 1.3

2. Lecture Notes :: http://www.facweb.iitkgp.ernet.in/ãnupam/language_ab.ppt

3. Lecture Notes :: Joachim Gerlach :: System-on-Chip Design with System C :: University of Tubingen (Dept of Computer Engg.)

4. IEEE Standard SystemC Language Reference Manual :: 1666^{TM} - 2005

Analysis and Estimation

Hardware Software CoDesign

September 2011

Agenda

Analysis and Estimation

1. Basic Themes that came out through the Historical Background (repeat)
2. Analysis and Estimation
3. Continue on "Answering Machine" Assignment Discussions

Analysis and Estimation

Basic Themes

1. **Modeling** :: Because parallelism is important, the choice of an appropriate modeling paradigm is also important. Different modeling formalisms need to be developed that capture the various aspects of system behavior.
2. **Analysis and Estimation** :: Different models of the same system behavior need to be analyzed and estimated for performance. Performance would include tradeoffs for Area, Speed, Power. Cost to build, Time to Market are other factors.
3. **System-level partitioning, synthesis and interfacing** :: The basic steps of co-synthesis. A range of methodologies are applied for this.
4. **Implementation generation** :: Once the architecture has been generated and trade-offs known, the designs for the hardware and software components must be created.
5. **Co-simulation and Emulation** :: This helps designers to evaluate architectures and validate assumptions on implementations. Emulation uses FPGA / other emulation techniques to further speed up execution of system models.

Analysis and Estimation

Introduction

1. **Analysis** :: Assumptions that which are formally dervived. Other than simulation, formal analysis covers all possible behaviors of a system that lead to more reliable verification results. Analysis precision is crucial to avoid costly, overly conservative designs that might not be competitive.
2. **Estimation** :: Assumptions that are based on "educated guesses" possibly based on analysis steps.

Analysis and Estimation

Analysis – Process Path

1. **Process Path** :: A sequence of process statements that is executed for given input data.
   - If there are loops, statements can be executed several times on a path.
   - Data dependent control statements in the process decide which of the possible path is taken.
   - In the most general case :: the determination of the set of possible process paths and the frequency of execution of each single statements is a **known non-computable problem**
   - Practical embedded system processes are bounded in their running time and so, path length.
   - GROUP DISCUSSION How does one determine upper bounds on a set of possible process paths?

Analysis and Estimation

Analysis – Process Path – Path Enumeration Technique

1. GROUP DISCUSSION How does one determine upper bounds on a set of possible process paths?

   One approach was the introduction of **the implicit path enumeration technique.**

   (a) Divide the program into basic blocks :: short sequences of statements in a program that include only one entry point at the first statement and one exit point at the end.

   (b) Determine the running time for these basic blocks (by architecture analysis).

   (c) The path constraints are implicitly given by equations and inequalities that describe the relative frequency of basic block execution.

   (d) Basic block running times and equations / inequalities form an ILP problem (IntegerLinearProgramming).

2. This technique is good for **Single-Processor Systems with Simple Memory Architectures** and is widely used.

3. This technique can be easily extended to cost functions other than timing, by changing the data provided by the model from timing to say power consumption.

4. Besides worst case analysis, it could be possible to add statistical methods that assume branching probabilities to obtain "statistical timing".

5. GROUP DISCUSSION :: What are the scenarios where the process path technique can get complicated.

Analysis and Estimation

Analysis – Process Path – Path Enumeration Technique

1. GROUP DISCUSSION :: What are the scenarios where the process path technique can get complicated.

   (a) The behavior of instruction cache does not fall under the ILP approach since dependencies between statements depend on their memory address and not on the control flow.

   (b) Power analysis is more complicated even for smaller architectures, because in CMOS technology, most of the power consumption is caused by transistor and wire switching that depend on the "sequence of operations" rather than on a single instruction.

   (c) However, for processor architectures like DSP's, the enabling / disabling of datapaths in the instruction sequence can effectively enable power analysis. Even when doing so there is "data dependency" on power analysis.

   (d) A simplified Process path technique can get complicated during analysis of parallel process execution.

Analysis and Estimation

Analysis Of Parallel Process Execution

1. A simplified Process path technique can get complicated during analysis of parallel process execution. While single processes run un-interrupted, the execution of parallel processes can be delayed.

2. Reasons for delay are ::

   (a) Process p_i waits for the output of another process p_j

   (b) Process p_j blocks a shared resource (shared memory, IO channel..) that p_i requires for continue execution.

   (c) Another reason for delay is processes running on one component can interrupt each other.

3. Thus, a problem statement would be :: given a set of tasks, to decide which task gets the CPU time at a given instance in time

4. The Goals are :-

   (a) Minimize turnaround time

   (b) Maximize throughput

   (c) Maximize CPU utilization

   (d) Minimize response time

   (e) Fairness :: avoid starvation

Analysis and Estimation

Analysis Of Parallel Process Execution – Running on One CPU

1. Discuss the following New terms :-

   (a) **Computation Time** :: of a task (or process) – the execution time of a process when it is run without interruption.

   (b) **Response Time** :: of a task – the time between the request for process execution, to the instant when a process has finished execution, including all delays by interrupts and dependencies.

   (c) **Deadline** :: of a task – the instant in which a task must have finished.

   (d) **Overflow** :: of a task – occurs at a time t if t is the deadline of an unfulfilled request.

   (e) **Critical instant** :: of a task – an instant at which a request for the process will have the largest response time.

   (f) **Scheduling** :: of a task – the order in which tasks are executed.

   (g) **Pre-Emptive Scheduling** :: of a task – interrupts running tasks to execute other tasks.

   (h) **Non-Pre-Emptive Scheduling** :: of a task – does not support interrupting executing tasks.

   (i) **Static Scheduling** :: of a task – is when the scheduling algorithm determines a fixed order of process execution.

(j) **Dynamic Scheduling** :: of a task – is when the scheduling is done during run time and requires a scheduler task (itself).

(k) **Scheduling Algorithm** :: is a set of rules that determine the task to be executed at a particular moment.

Analysis and Estimation

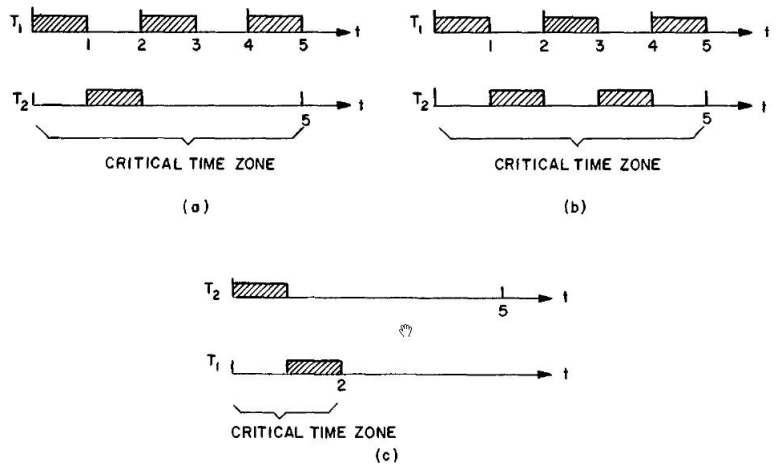
Analysis Of Parallel Process Execution – Running on One CPU – Rate Monotic Analysis

1. Let $(T_1, T_2, ...T_m)$ be the request Time periods of processes (tasks) $(p_1, p_2, ...p_m)$
2. Let $(C_1, C_2, ...C_m)$ be their Computation Times.
3. So, the request Rate $= (1/T_n)$ for process n.
4. Take an example where $T_1 = 2, T_2 = 5$ and $C_1 = 1, C_2 = 1$
5. Static prioritization was optimal, with the highest priority going to the process with the shortest period. This is known as **Rate Monotonic Priority Assignment**

Analysis and Estimation

Analysis Of Parallel Process Execution – Running on One CPU

1. The timing of parallel processes running on one cpu is a problem in "real time computing"
2. In this problem, many applications (signal processing, control) require "periodic process execution", or they can be mapped to periodic process execution.
3. Assume you know the process deadlines (equal to the individual process periods).
4. And the process maximum computation time.
5. **GROUP DISCUSSION** :: Can you derive an algorithm to show that the system of processes is schedulable?



(a) (Process p_1 has higher Priority) – Feasible

(b) (Process p_1 has higher Priority) – C_2 can be increased to 2

(c) (Process p_2 has higher Priority) – C_1, C_2 can be at most to 1

6. Processor Utilization $= \sum_{i=1}^{m} (C_i/T_i)$

Analysis and Estimation

Analysis Of Parallel Process Execution – Running on One CPU – Rate Monotic Analysis

Theorem 1

For a set of n independent periodic tasks

if $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{\frac{1}{n}}-1)$

where
C_i = execution time
P_i = period of task i

then the set is schedulable by rate monotonic algorithm for all task phasings.

Analysis and Estimation

Analysis Of Parallel Process Execution – Running on One CPU – Rate Monotic Analysis

1. **GROUP DISCUSSION**

Task Set — **Schedulable?**

Process	C	T	U= C/P
P1	20	100	0.2
P2	40	150	0.267
P3	100	350	0.286
		Total	0.753

Analysis and Estimation

Analysis Of Parallel Process Execution – Running on One CPU – Deadline Driven Analysis

1. While RMA is a static priority scheme, the deadline driven analysis is a dynamic scheme.
2. Priorities are assigned to tasks according to the deadlines of their current requests.
3. A task will be assigned a highest priority if the deadline of its current request is the nearest, and will be assigned the lowest priority if the deadline of its current request is the furthest.
4. At any moment, the task with the highest priority and yet unfulfilled request will be executed.
5. For a given set of m tasks, the deadline driven scheduling algorithm is feasible if and only if $(C_1/T_1) + (C_2/T_2) + ... + (C_m/T_m) \leq 1$
6. Total demand cannot exceed the available processor time.
7. The deadline driven scheduling algorithm is optimum in the sense that if a set of tasks can be scheduled by any algorithm, it can be scheduled by the deadline driven scheduling algorithm.

Analysis and Estimation

Analysis Of Parallel Process Execution – Running on One CPU – Mixed Scheduling Algorithm

1. Implement a deadline driven scheduler for the slower paced tasks.
2. Let tasks $1, 2, ..., k$ be the k tasks of shortest periods. Schedule these according to the fixed priority rate-monotonic scheduling algorithm.
3. Let the remaining tasks $k+1, k+2, ..., m$ be scheduled according to the deadline driven scheduling algorithm when the processor is not occupied by tasks $1, 2, ..., k$
4. **Example** With 3 tasks.
   - $T_1 = 3, T_2 = 4, T_3 = 5$; $C_1 = 1, C_2 = 1, C_3 = 1(ratemonotonic), 2(mixed)$
   - Fixed Priority $U = 1/3 + 1/4 + 1/5 = 78.3\%$
   - Mixed Scheduling $U = 1/3 + 1/4 + 2/5 = 98.3\%$

Analysis and Estimation

Acknowledgements

1. Readings in Hardware / Software Co-design ::G. D. Micheli, Rolf Ernst, Wayne Wolf :: Ch 3

2. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment :: C. L. Liu, James W. Layland :: Journal of ACM (vol 20, January 1973).

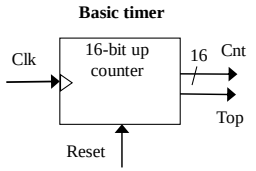
3. Hardware / Software Co-Design - Principles and Practice :: J. Staunstrup & W. Wolf :: Ch 2.3

Introduction

- Single-purpose processors
  - Performs specific computation task
  - Custom single-purpose processors
    - Designed by us for a unique task
  - *Standard* single-purpose processors
    - "Off-the-shelf" -- pre-designed for a common task
    - a.k.a., peripherals
    - serial transmission
    - analog/digital conversions

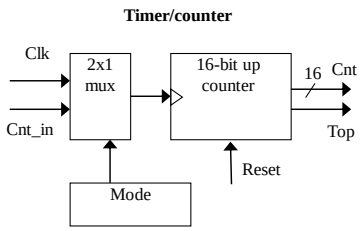
Timers, counters, watchdog timers

- Timer: measures time intervals
  - To generate timed output events
    - e.g., hold traffic light green for 10 s
  - To measure input events
    - e.g., measure a car's speed
- Based on counting clock pulses
  - E.g., let Clk period be 10 ns
  - And we count 20,000 Clk pulses
  - Then 200 microseconds have passed
  - 16-bit counter would count up to 65,535*10 ns = 655.35 microsec., resolution = 10 ns
  - Top: indicates top count reached, wrap-around



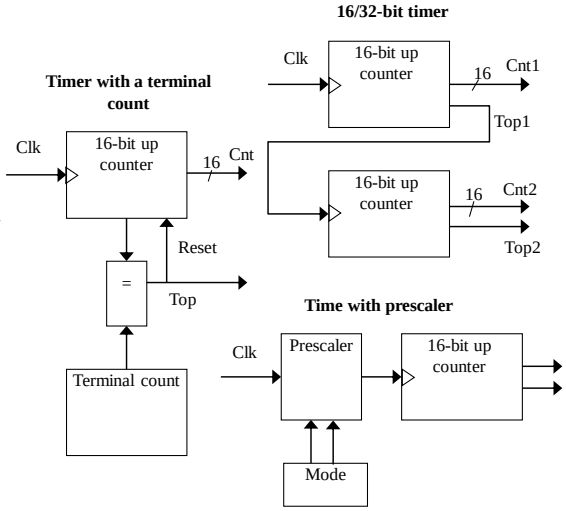
Counters

- Counter: like a timer, but counts pulses on a general input signal rather than clock
  - e.g., count cars passing over a sensor
  - Can often configure device as either a timer or counter

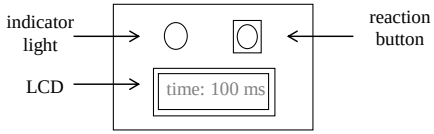


Other timer structures

- Interval timer
  - Indicates when desired time interval has passed
  - We set terminal count to desired interval
    - *Number of clock cycles = Desired time interval / Clock period*
- Cascaded counters
- Prescaler
  - Divides clock
  - Increases range, decreases resolution



Example: Reaction Timer

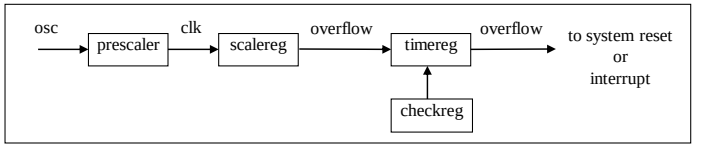


- Measure time between turning light on and user pushing button
  - 16-bit timer, clk period is 83.33 ns, counter increments every 6 cycles
  - Resolution = 6*83.33=0.5 microsec.
  - Range = 65535*0.5 microseconds = 32.77 milliseconds
  - Want program to count millisec., so initialize counter to 65535 – 1000/0.5 = 63535

```
/* main.c */

#define MS_INIT      63535
void main(void){
  int count_milliseconds = 0;

  configure timer mode
  set Cnt to MS_INIT

  wait a random amount of time
  turn on indicator light
  start timer

while (user has not pushed reaction button){
    if(Top) {
      stop timer
      set Cnt to MS_INIT
      start timer
      reset Top
      count_milliseconds++;
    }
  }
  turn light off
  printf("time:  %i ms", count_milliseconds);
}
```


Watchdog timer

- Must reset timer every X time unit, else timer generates a signal
- Common use: detect failure, self-reset
- Another use: timeouts
  - e.g., ATM machine
  - 16-bit timer, 2 microsec. resolution
  - *timereg* value = $2*(2^{16}-1)–X = 131070–X$
  - For 2 min., X = 120,000 microsec.

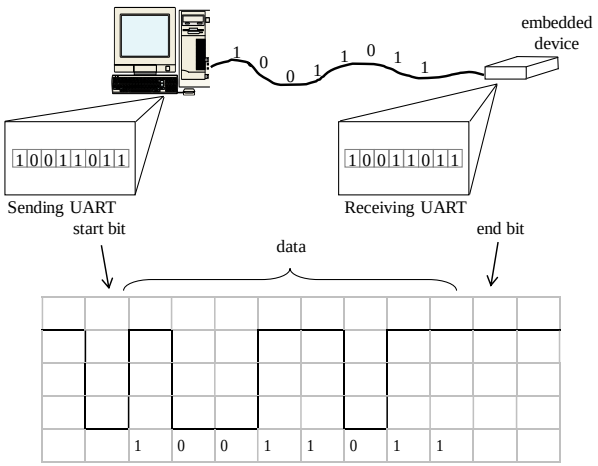



```
/* main.c */

main(){
  wait until card inserted
  call watchdog_reset_routine

  while(transaction in progress){
    if(button pressed){
      perform corresponding action
      call watchdog_reset_routine
    }

/* if watchdog_reset_routine not called every
< 2 minutes, interrupt_service_routine is
called */
}
```

```
watchdog_reset_routine(){
/* checkreg is set so we can load value into
timereg.  Zero is loaded into scalereg and
11070 is loaded into timereg */

  checkreg = 1
  scalereg = 0
  timereg  = 11070
}

void interrupt_service_routine(){
  eject card
  reset screen
}
```

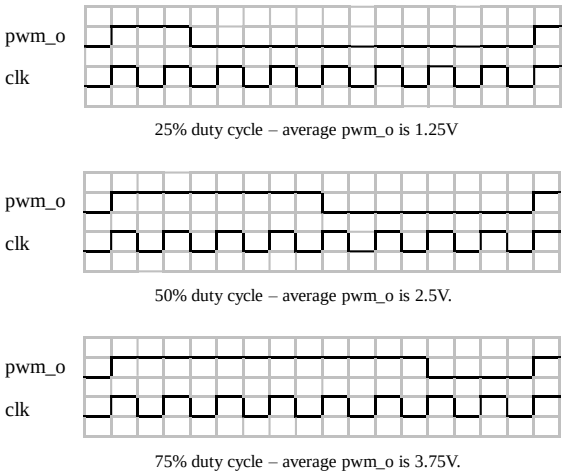
Serial Transmission Using UARTs

- UART: Universal Asynchronous Receiver Transmitter
  - Takes parallel data and transmits serially
  - Receives serial data and converts to parallel
- Parity: extra bit for simple error checking
- Start bit, stop bit
- Baud rate
  - signal changes per second
  - bit rate usually higher

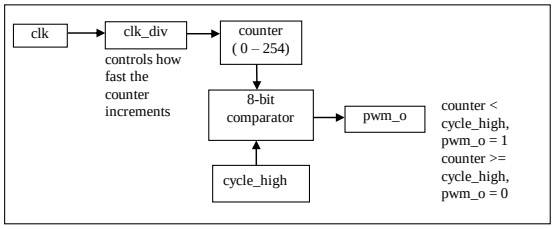


Pulse width modulator

- Generates pulses with specific high/low times
- Duty cycle: % time high
  - Square wave: 50% duty cycle
- Common use: control average voltage to electric device
  - Simpler than DC-DC converter or digital-analog converter
  - DC motor speed, dimmer lights
- Another use: encode commands, receiver uses timer to decode



Controlling a DC motor with a PWM



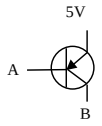
Internal Structure of PWM

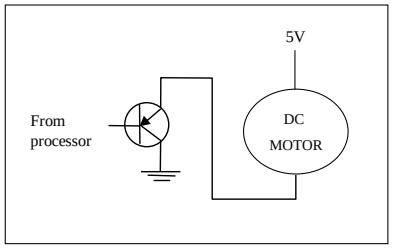
Input Voltage	% of Maximum Voltage Applied	RPM of DC Motor
0	0	0
2.5	50	1840
3.75	75	6900
5.0	100	9200

Relationship between applied voltage and speed of the DC Motor

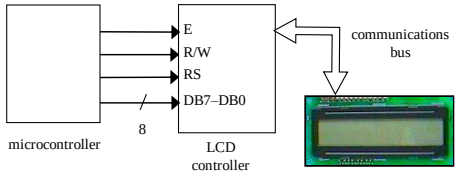
```
void main(void){

   /* controls period */
   PWMP = 0xff;
   /* controls duty cycle */
   PWM1 = 0x7f;

   while(1){};
 }
```

The PWM alone cannot drive the DC motor, a possible way to implement a driver is shown below using an MJE3055T NPN transistor.





LCD controller




```
void WriteChar(char c){

  RS = 1;              /* indicate data being sent */
  DATA_BUS = c;        /* send data to LCD */
  EnableLCD(45);       /* toggle the LCD with appropriate delay */
}
```

CODES	
I/D = 1 cursor moves left	DL = 1 8-bit
I/D = 0 cursor moves right	DL = 0 4-bit
S = 1 with display shift	N = 1 2 rows
S/C =1 display shift	N = 0 1 row
S/C = 0 cursor movement	F = 1 5x10 dots
R/L = 1 shift to right	F = 0 5x7 dots
R/L = 0 shift to left	

RS	R/W	DB_7	DB_6	DB_5	DB_4	DB_3	DB_2	DB_1	DB_0	Description
0	0	0	0	0	0	0	0	0	1	Clears all display, return cursor home
0	0	0	0	0	0	0	0	1	*	Returns cursor home
0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and/or specifies not to shift display
0	0	0	0	0	0	1	D	C	B	ON/OFF of all display(D), cursor ON/OFF (C), and blink position (B)
0	0	0	0	0	1	S/C	R/L	*	*	Move cursor and shifts display
0	0	0	0	1	DL	N	F	*	*	Sets interface data length, number of display lines, and character font
1	0	WRITE DATA								Writes Data

System Level Partitioning, Synthesis and Interfacing

Hardware Software CoDesign

October 2011

Agenda

Analysis and Estimation

1. Basic Themes that came out through the Historical Background (repeat)
2. System Level Partitioning, Synthesis and Interfacing
3. TextBook example of a single-purpose processor implementing the GCD program
4. Continue on "Answering Machine" Assignment Discussions. Give clear milestones.

Analysis and Estimation

Basic Themes

1. **Modeling** :: Because parallelism is important, the choice of an appropriate modeling paradigm is also important. Different modeling formalisms need to be developed that capture the various aspects of system behavior.
2. **Analysis and Estimation** :: Different models of the same system behavior need to be analyzed and estimated for performance. Performance would include tradeoffs for Area, Speed, Power. Cost to build, Time to Market are other factors.
3. **System-level partitioning, synthesis and interfacing** :: The basic steps of co-synthesis. A range of methodologies are applied for this.
4. **Implementation generation** :: Once the architecture has been generated and trade-offs known, the designs for the hardware and software components must be created.
5. **Co-simulation and Emulation** :: This helps designers to evaluate architectures and validate assumptions on implementations. Emulation uses FPGA / other emulation techniques to further speed up execution of system models.

System Level Partitioning, Synthesis and Interfacing

Introduction

1. A hardware / software partitioning problem can be stated as finding those parts of the model best implemented in hardware and those best implemented in software.
   - Partitioning can be decided by the designer, with successive refinement of initial model
   - It can also be determined by a CAD tool.
2. In the case of embedded systems, a hardware/software partition represents a physical partition of system functionality into application-specific hardware and software on one or more processor(s).
3. There are multiple techniques... We will discuss the paper by Kalavade and Lee.

System Level Partitioning, Synthesis and Interfacing

Kalavade and Lee (Kal97)

1. Take a global view of the partitioning problem.
2. Assume that a homogeneous procedural model is compiled into task graphs.
3. Then determine the implementation choice (hw/sw) for each task graph node.
4. Also, scheduling these nodes at the same time so that real-time constraints are met.
5. Note that there is an intimate relation between partitioning and scheduling. This is caused by wide variation in timing properties of the hw/sw implementations of a task which affects the overall latency significantly.

System Level Partitioning, Synthesis and Interfacing

Kalavade and Lee (Kal97)

1. The basic premises is that typically the designer needs to explore the possible options, tools and architectures.
2. A **design methodology management** framework helps to manage the design space exploration process. A software environment called **Design Assistant** addresses :-
   (a) Specific tools for partitioning, synthesis and simulation that are configured for a particular hw/sw codesign flow.
   (b) An underlying design methodology management infrastructure for design space exploration.
3. Each node in the task-level specification can be implemented in several ways in both hw/sw mappings.
4. At hw implementation it can be :-
   (a) *Algorithm level* :: Several algorithms can be used to describe the same task. Example a FIR filter can be implemented either as an inner product or using the FFT in a shif-and-add algorithm.
   (b) *Transformation level* :: For a particular algorithm, several transforamtions can be applied on the original task description.
   (c) *Resource level* :: A task, for a specified algorithm and transformation set, can be implemented using varying numbers of resource units.

System Level Partitioning, Synthesis and Interfacing

Kalavade and Lee (Kal97)

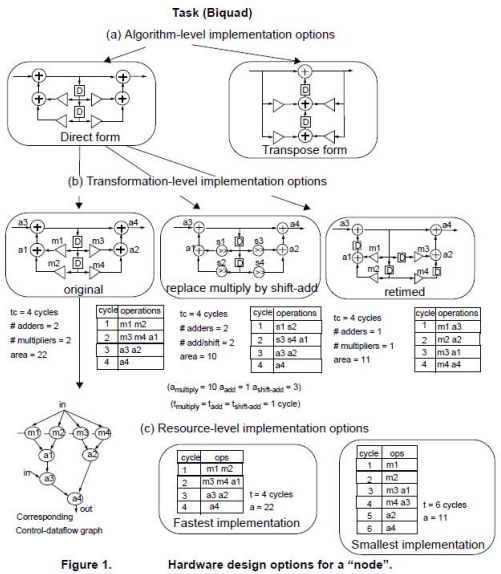


Figure 1. **Hardware design options for a "node".**

System Level Partitioning, Synthesis and Interfacing

Kalavade and Lee (Kal97)

1. The goal of partitioning is to determine three parameters :: mapping (hw/sw), implementation (type wrt algorithm, transformation and area-time value) and schedule (when it executes wrt other tasks).
2. Partitioning is non-trivial problem :: consider a task-level specification, typically in the order of 50 to 100 tasks. Each task can be mapped into hw/sw. Given a mapping say there are 5 design options for each node, then there will be $(2 \cdot 5)^{100}$ design options. Say there are p preferred implementations, there are still a large number of design alternatives wrt the remaining nodes $(2 \cdot 5)^{100-p}$.
3. Partitioning again can be divided into two stages ::
   (a) *Binary partitioning* :: problem of determining hw/sw mapping and schedule.
   (b) *Extended partitioning* :: problem of selecting an appropriate implementation over and above binary partitioning.
4. We will discuss Binary partitioning in detail and Extended partitioning briefly.

System Level Partitioning, Synthesis and Interfacing

Kalavade and Lee (Kal97)

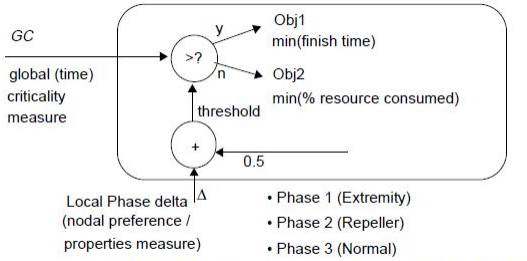
The Partitioning Method

1. Serially traverse a node list (usually from the source node to the sink node in the DAG)
2. For each node select a mapping that minimizes an objective function. The objective function can be to minimize the finish time, or minimize the area (i.e, hw area or software size).
1. There are limitations :: The CAD algorithms tend to be greedy and hence can be suboptimal because they can either go for only finish time or only area.
2. To overcome the limitations :: adaptively select an appropriate mapping objective at each step to determine the mapping and schedule.
3. *Global Criticality GC* :: GC is a global look-ahead measure that estimates the time criticality at each step of the algorithm. GC is compared to a threshold to determine if time is critical. If time is critical, an objective function that minimizes finish time is selected, otherwise one that minimizes area is selected. GC can change at every step (node) of the algorithm.
4. *Local Phase LP* :: LP is a classification of nodes based on their heterogeneity and intrinsic properties. Each node is classified as an extermity (local phase 1), repeller (local phase 2) or normal (local phase 3) node. A measure called "local phase delta" quantifies the local mapping preferences of the node under consideration and accordingly modifies the threshold used in the GC comparison.

System Level Partitioning, Synthesis and Interfacing

Kalavade and Lee (Kal97)

The Partitioning Method



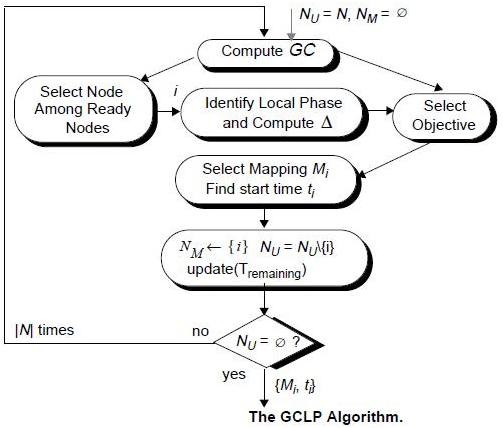
Selection of the mapping objective at each step of GCLP.

System Level Partitioning, Synthesis and Interfacing

Kalavade and Lee (Kal97)

The Partitioning Method

1. N_U = set of unmapped nodes at the current step. It is initialized to N the total nodes in the DAG.
2. N_M = set of mapped nodes at the current step.
3. Unmapped nodes whose predecessors have already been mapped and scheduled are called "ready" nodes.
4. A node is selected for mapping from the set of ready nodes using an urgency criteria. ie., a ready node that lies on the critical path is selected for mapping.
5. The local phase of the selected node is identified and the corresponding local phase delta is computed.

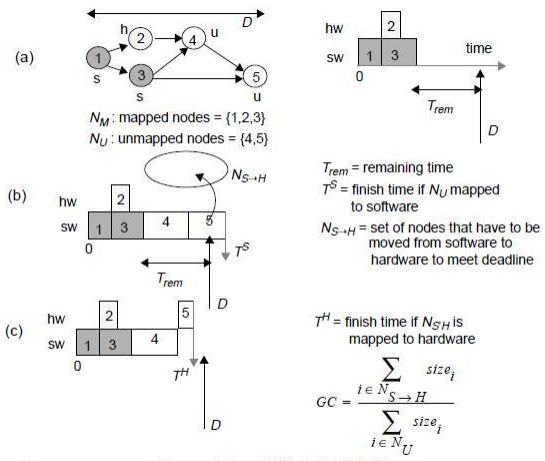


The GCLP Algorithm.

System Level Partitioning, Synthesis and Interfacing

Kalavade and Lee (Kal97)

The Partitioning Method



Computation of Global Criticality

System Level Partitioning, Synthesis and Interfacing

Kalavade and Lee (Kal97)

The Partitioning Method – Assumptions

1. Requirement of a DAG. The throughput constraint on the graph translates to a deadline D ie., the execution time of the DAG should not exceed D clock cycles. The DAG itself is a SDF (Synchronous Data Flow) graph.

2. The target architecture consists of a single programmable processor, which executes the software component and a custom datapath (the hardware component). The software and hardware components have capacity constraints – the software (program and data) size and should not exceed AS (memory requirements) and the hardware size should not exceed AH.

3. The area and time estimates for the hw/sw implementation bins are assumed to be known.

4. There is no reuse between nodes mapped to hw.

System Level Partitioning, Synthesis and Interfacing

Kalavade and Lee (Kal97)

Extended Partitioning

1. The GCLP algorithm solves the binary partititon problem.

2. The Extended Partitioning problem is to jointly optimize the mapping as well as implementation bin for each node.

3. Consider an implementation bin set (usually a graph with implementation points) with L = fastest implementation and H = slowest implementation.

4. This is far more complex than the binary partition problem. The binary partition problem has $2^{|N|}$ mapping possibilities. Given B implementation bins within a mapping, the extended partitioning problem has $(2B)^{|N|}$ possibilities in the worst case.

5. It is not enough to decompose the extended partitioning problem into two isolated steps viz, mapping followed by implementation bin selection. WHY?
   - The serial traversal of nodes in the DAG means that the implementation bin of a particular node affects the mapping of as-yet-unmapped nodes.
   - There is a correlation between mapping and implementation-bin selection, they cannot be optimized in isolation.

System Level Partitioning, Synthesis and Interfacing

Kalavade and Lee (Kal97)

Extended Partitioning Method (MIBS)

1. Let each node have attributes :: mapping, implementation bin, schedule.

2. As the algorithm progresses, depending on the extent of information that has been generated, each node goes through a sequence of three states :: (1) free, (2) tagged, (3) fixed.

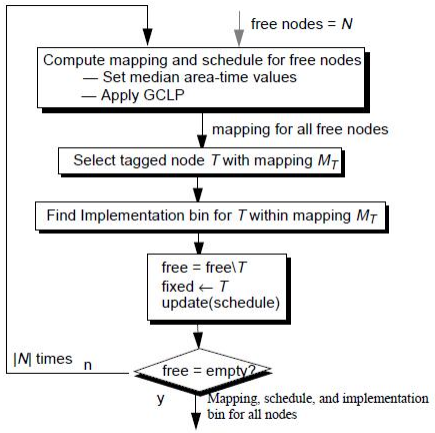
3. GCLP is first applied to get a mapping and schedule for all the free nodes in the DAG.

4. A particular free node (called a "tagged" node) is then selected.

5. Assume its mapping to be that determined by GCLP, an appropriate implementation bin is then chosen for the tagged node. HOW ??

6. Once the mapping and implementation bin are known, the tagged node becomes a "fixed" node.

7. GCLP is applied on the remaining nodes and this process is repeated until all nodes in the DAG become "fixed".



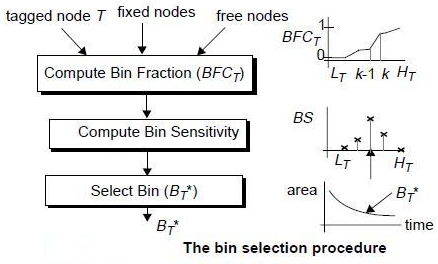
MIBS approach to solving extended partitioning

System Level Partitioning, Synthesis and Interfacing

Kalavade and Lee (Kal97)

Extended Partitioning Method (MIBS) – Implementation Bin Selection

1. Assume its mapping to be that determined by GCLP, an appropriate implementation bin is then chosen for the tagged node. HOW ??
   (a) Let the free nodes mapped to hw at the current step be called $free^h$ nodes. (Same can be extended to software implementation bin also).
   (b) Select the most responsive bin in this respect as the implementation bin for the tagged node.
   (c) The key idea is to use a look-ahead measure to correlate the implementation bin of the tagged node with the hardware area required for the $free^h$ nodes.
   (d) The look-ahead measure (called bin fraction BT_T^j) computes, for each bin j of the tagged node T, the fraction of $free^h$ nodes that need to be moved from H bins to the L bins in order to meet timing constraints.
   (e) A high value of BT_T^j indicates that if the tagged node T were to be implemented in bin j, a large fraction of $free^h$ nodes would likely get mapped to their fast implementations (L bins), hence increasing the overall area (or memory).
   (f) The bin fraction curve BFC_T is the collection of all bin fraction values of the tagged node T
   (g) The bin sensitivity is the gradient of BFC_T, which reflects the responsiveness of the bin fraction to the bin motion of node T.



The bin selection procedure

System Level Partitioning, Synthesis and Interfacing

Summary of other Methodologies / Algorithms

Polis System :: Hardware-Software Codesign of Embedded Systems

1. Chiodo et. all (Chi94) describe the Polis System where designs are described as networks of co-design finite state machines (CFSMs).
2. A CFSM designś components are assigned to implementation in either hw or sw.
3. Hardware units are fully synchronous.
4. Each software component is implemented as a standalone C program.

System Level Partitioning, Synthesis and Interfacing

Summary of other Methodologies / Algorithms

Heterogeneous MPU systems :: SOS Synthesis of Application-Specific Heterogeneous Multiprocessor Systems

1. Prakash and Parker (Pra92) describe a formal method of design, based on mixed-ILP programming model. The basics include :-
2. Synthesis of hardware units involves several subtasks.
3. First and foremost is operation scheduling, which may or maynot be combined with system level partitioning.
4. Different scheduling approaches are used, often borrowed from the OperatingSystems and Real-Time Systems concepts.
5. The method provides a static schedule, as well as an assignment of tasks to processors.

System Level Partitioning, Synthesis and Interfacing

Summary of other Methodologies / Algorithms

Wolf (Wol97) :: An Architectural co-Synthesis Algorithm for Distributed, Embedded Computing Systems

1. This paper addresses hw/sw co-synthesis in distributed systems.
2. Describes a method to simultaneously synthesize the hardware and software architectures of a distributed system to satisfy performance requirements and minimize cost.
3. The hardware consists of a network of processors with an arbitary communication topology.
4. The software consists of an allocation of processes to processors and a schedule for the processes.

Example of single-purpose processor implementing the GCD program

Problem Statement

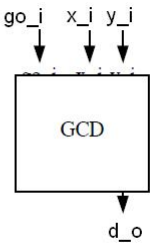
1. Take a break for 10mins.

Example of single-purpose processor implementing the GCD program

Problem Statement

1. The system's functionality :: the output should represent the GCD of the inputs
2. i.e, if 12, 8 are inputs, output should be 4. If 13 and 5 are inputs output should be ??
3. GROUP ACTIVITY :: Is the below algorithm actually for GCD. Are there better ways ??
4. GCD algorithm and entity description of the implementation

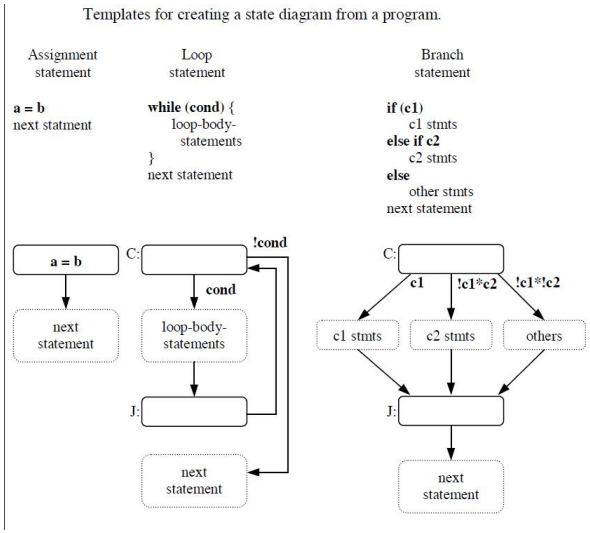
```
0: vectorN x, y;
1: while (1) {
2:   while (!go_i);
3:   x = x_i;
4:   y = y_i;
5:   while (x != y) {
6:     if (x < y)
7:       y = y - x;
       else
8:       x = x - y;
     }
9:   d_o = x;
   }
```



Example of single-purpose processor implementing the GCD program

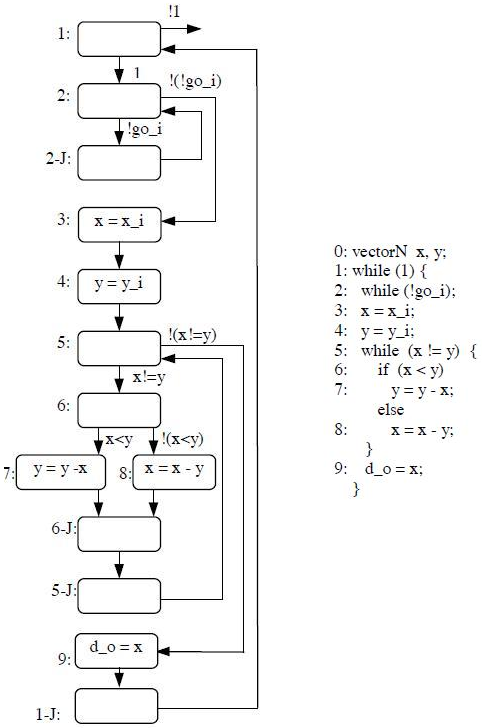
Building Blocks

1. First convert the program into a (complex) state diagram, in which states and arcs may include arithmetic expressions, and these expressions may use external inputs, outputs, or variables.



Example of single-purpose processor implementing the GCD program

Use Building Blocks to convert to a State Diagram



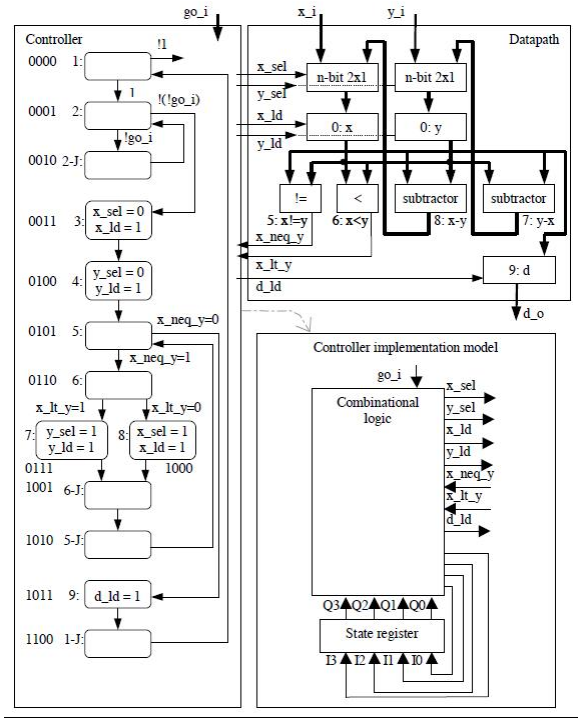

```
0: vectorN  x, y;
1: while (1) {
2:   while (!go_i);
3:   x = x_i;
4:   y = y_i;
5:   while  (x != y)  {
6:     if  (x < y)
7:       y = y - x;
       else
8:       x = x - y;
     }
9:   d_o = x;
   }
```

Example of single-purpose processor implementing the GCD program

From State Diagram – BreakUp the functionality into datapath and a controlpath

1. Create a register for any declared variable. (x, y). For registered outputs create d as register
2. Create a functional unit for each arithmetic operation in the state diagram. ie., 2 subtractors, 2 comparators one for lessThan (lt), one for nonEq (neq).
3. Connect the ports, registers and functional units.
   (a) For each write to a variable in the state diagram, draw a connection from the writeś source (input port, functional unit, or another register) to the variableś register.
   (b) For each arithmetic and logical operation, connect sources to an input of the operationś corresponding functional unit.
   (c) When more than one source is connected to a register, add an appropriate mux.
4. Create a unique identifier for each control input and output of the data components.

Example after splitting into a controller and a datapath.



Example of single-purpose processor implementing the GCD program

From State Diagram – Build the StateTable for registers and IO's

State table for the GCD example.

State table

Inputs							Outputs								
Q3	Q2	Q1	Q0	x_neq_y	x_lt_y	go_i	I3	I2	I1	I0	x_sel	y_sel	x_ld	y_ld	d_ld
0	0	0	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	0	1	*	*	0	0	0	1	0	X	X	0	0	0
0	0	0	1	*	*	1	0	0	1	1	X	X	0	0	0
0	0	1	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	1	1	*	*	*	0	1	0	0	0	X	1	0	0
0	1	0	0	*	*	*	0	1	0	1	X	0	0	1	0
0	1	0	1	0	*	*	1	0	1	1	X	X	0	0	0
0	1	0	1	1	*	*	0	1	1	0	X	X	0	0	0
0	1	1	0	*	0	*	1	0	0	0	X	X	0	0	0
0	1	1	0	*	1	*	0	1	1	1	X	X	0	0	0
0	1	1	1	*	*	*	1	0	0	1	X	1	0	1	0
1	0	0	0	*	*	*	1	0	0	1	1	X	1	0	0
1	0	0	1	*	*	*	1	0	1	0	X	X	0	0	0
1	0	1	0	*	*	*	0	1	0	1	X	X	0	0	0
1	0	1	1	*	*	*	1	1	0	0	X	X	0	0	1
1	1	0	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	0	1	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	1	*	*	*	0	0	0	0	X	X	0	0	0

* - indicates all possible combinations of 0's and 1's
X - indicates don't cares

Example of single-purpose processor implementing the GCD program Other Optimizations ??

1. Could the algorithm be modified to yield lower / predictable latency (better algorithm)
2. Can you re-use subtractors by adding muxes and scheduling them (resource sharing)
3. Optimizing the FSM (state minimization)

Example of Dedicated H/W for matrix multiplication

Discuss the design of a Matrix Multiplication H/W $A(3x2) \cdot B(2x3) = C(3x3)$

```
main () {
  int A[3][2] = { {1,2}, {3,4}, {5,6}};
  int A[2][3] = { {7,8,9}, {10,11,12}};
  int C[3][3];
  int i, j, k;
  for (i=0; i < 3; i++) {
    for (j=0; i < 3; j++) {
      C[i][j] = 0;
      for (k=0; i < 2; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```


Continue on "Answering Machine" Assignment Discussions

Marking Milestones

1. Take a break for 5mins.
2. Implementation Documentation – Hardware and Software components, with all IPś in place, assumptions et. all → October 30 Deadline (10 marks)
3. Actual Implementation, documentation updates and enabling me to replay with my own set of tests → December 30 Deadline (10 marks)
4. Selection of key persons from the team (groups)
   (a) IP creation / reuse
   (b) System Hardware Integration
   (c) System Software Integration
   (d) TestBench Creation and System Checkout → Hardware & Software
   (e) Total Hardware / software → bringing it all together

Continue on "Answering Machine" Assignment Discussions

Requirements Till date

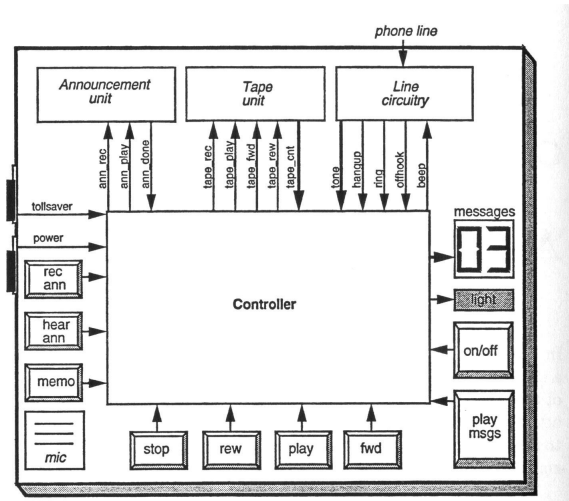
1. System C model of the system.
2. Use models for analog components.
3. Should have System C testbench as a part of SoC verification.
4. ReUse preferred to creation of already existing components.

Solution components / discussion

1. Decide Microprocessor.
2. How to write software program.
3. Memory requirements.
4. Model of ADC required (file input and digital output).
5. Input Buttons.
6. On/Off Button - special case.
7. Model of LEC (7 segment display).
8. Some real time clock – keep date and time (simplification is ok)
9. Each message has upper bound (10sec) – min 16 messages.
10. Announcement = single 10 seconds.
11. Announcement hear out through microphone (DAC)
12. Light keeps blinking if there is a new message.

"Answering Machine" Assignment

Diagram of Use Case



The answering machine controller's environment.

"Answering Machine" Assignment

Next Steps for Implementation document

1. Entity Definition
2. Processor / Dedicated Hardware decision
3. System State Machine or StateChart diagram (gross functionality)
4. Individual functionality State Macine or StateChart diagram

System Level Partitioning, Synthesis and Interfacing

Acknowledgements

1. Readings in Hardware / Software Co-design ::G. D. Micheli, Rolf Ernst, Wayne Wolf :: Ch 4
2. Embedded System Design - A Unified Hardware / Software Introduction :: Ch 2.4
3. The Extended Partitioning Problem : Hw/Sw Mapping, Scheduling, and Implementation bin selection :: Asawaree Kalavade, Edward Lee :: Proceedings of the 6th International Workshop on Rapid Systems Prototyping (1997).
4. Reference :: An efficient Hardware Design Tool for Scalable Matrix Multiplication :: Semih Aslan, Christophe Desmouliers, Erdal Oruklu and Jafar Saniie :: Electrical & Computer Engineering Dept (Illinois Institute of Technology).

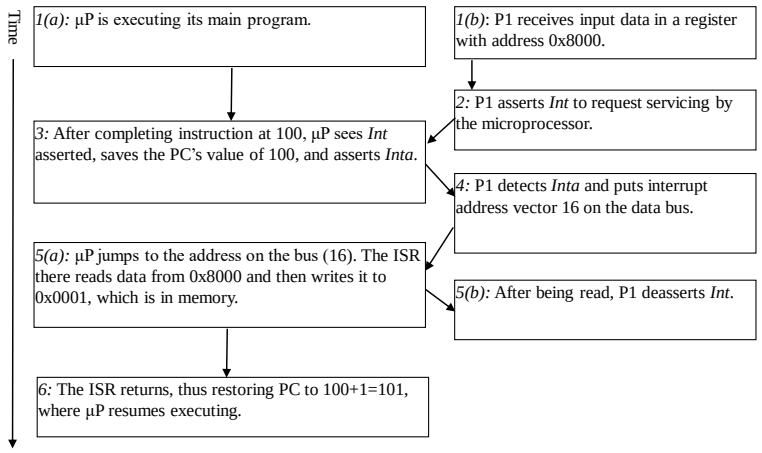
Embedded Systems Design: A Unified Hardware/Software Introduction

DMA

Direct memory access

- Buffering
  - Temporarily storing data in memory before processing
  - Data accumulated in peripherals commonly buffered
- Microprocessor could handle this with ISR
  - Storing and restoring microprocessor state inefficient
  - Regular program must wait
- DMA controller more efficient
  - Separate single-purpose processor
  - Microprocessor relinquishes control of system bus to DMA controller
  - Microprocessor can meanwhile execute its regular program
    - No inefficient storing and restoring state due to ISR call
    - Regular program need not wait unless it requires the system bus
      - Harvard archictecture – processor can fetch and execute instructions as long as they don't access data memory – if they do, processor stalls

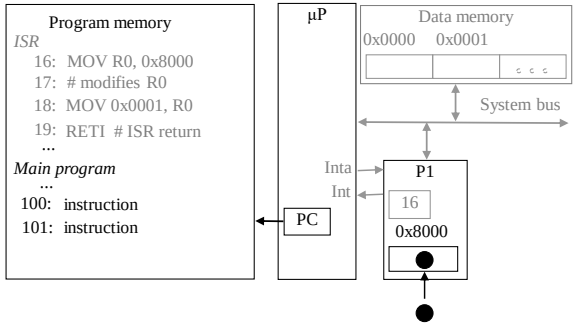
Peripheral to memory transfer *without* DMA, using vectored interrupt



Peripheral to memory transfer *without* DMA, using vectored interrupt

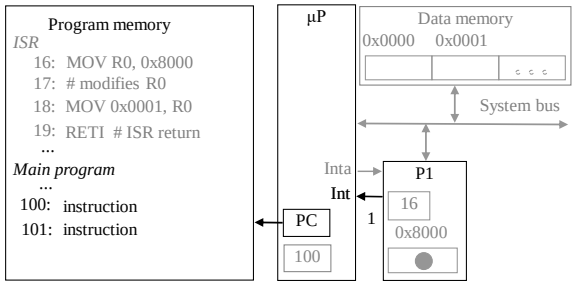
1(a): µP is executing its main program

1(b): P1 receives input data in a register with address 0x8000.



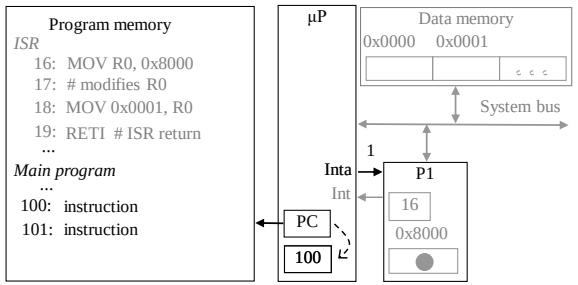
Peripheral to memory transfer *without* DMA, using vectored interrupt

2: P1 asserts *Int* to request servicing by the microprocessor



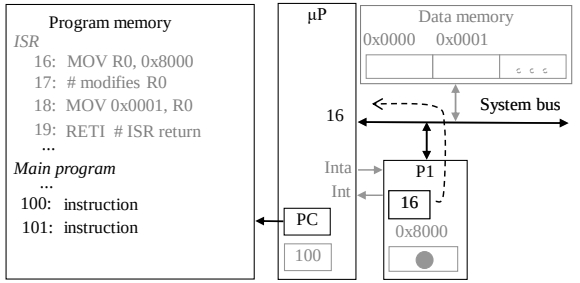
Peripheral to memory transfer *without* DMA, using vectored interrupt

3: After completing instruction at 100, µP sees *Int* asserted, saves the PC's value of 100, and asserts *Inta*.



Peripheral to memory transfer *without* DMA, using vectored interrupt (cont')

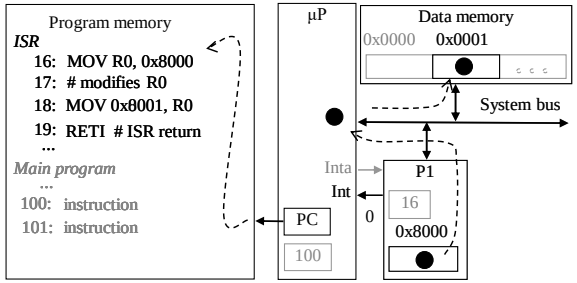
4: P1 detects *Inta* and puts interrupt address vector 16 on the data bus.



Peripheral to memory transfer *without* DMA, using vectored interrupt (cont')

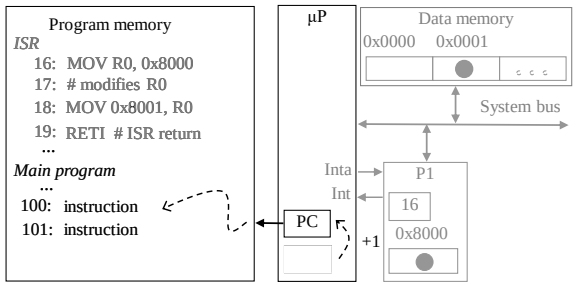
5(a): µP jumps to the address on the bus (16). The ISR there reads data from 0x8000 and then writes it to 0x0001, which is in memory.

5(b): After being read, P1 de-asserts *Int*.

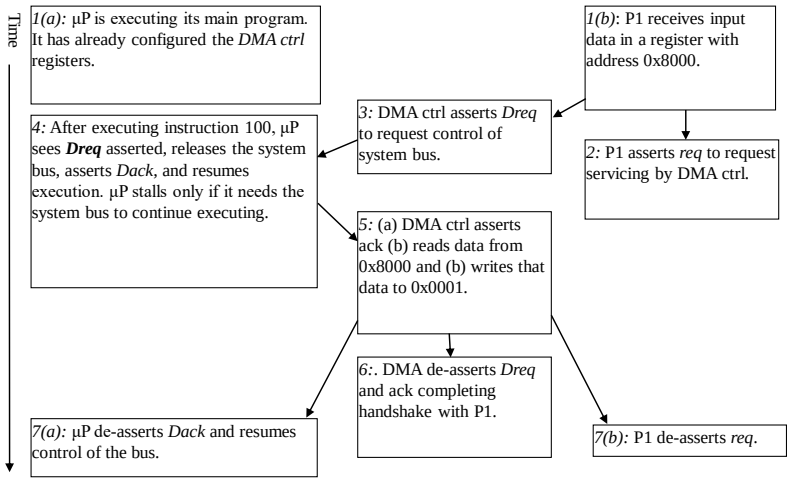


Peripheral to memory transfer *without* DMA, using vectored interrupt (cont')

6: The ISR returns, thus restoring PC to 100+1=101, where µP resumes executing.



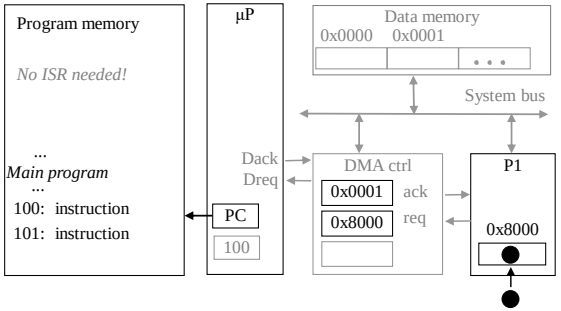
Peripheral to memory transfer with DMA



Peripheral to memory transfer with DMA (cont')

1(a): µP is executing its main program. It has already configured the DMA ctrl registers

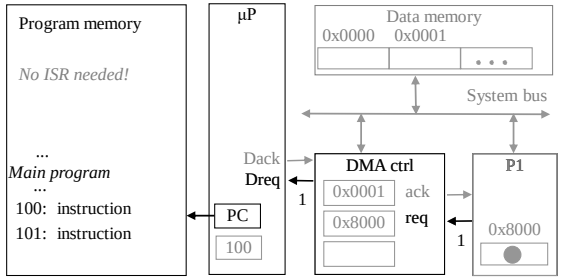
1(b): P1 receives input data in a register with address 0x8000.



Peripheral to memory transfer with DMA (cont')

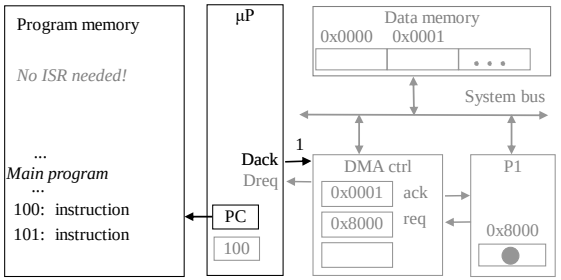
2: P1 asserts *req* to request servicing by DMA ctrl.

3: DMA ctrl asserts *Dreq* to request control of system bus



Peripheral to memory transfer with DMA (cont')

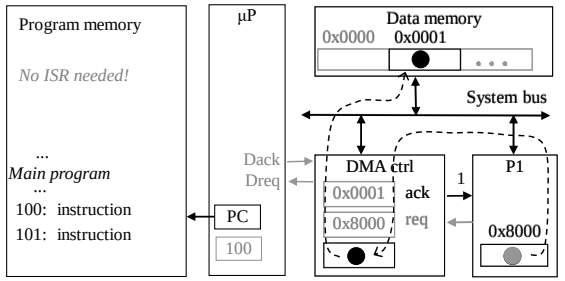
4: After executing instruction 100, µP sees *Dreq* asserted, releases the system bus, asserts *Dack*, and resumes execution, µP stalls only if it needs the system bus to continue executing.



Peripheral to memory transfer with DMA (cont')

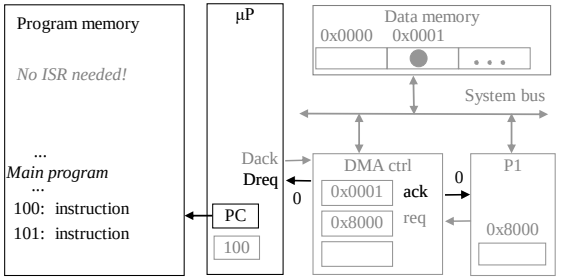
5: DMA ctrl (a) asserts ack, (b) reads data from 0x8000, and (c) writes that data to 0x0001.

(Meanwhile, processor still executing if not stalled!)

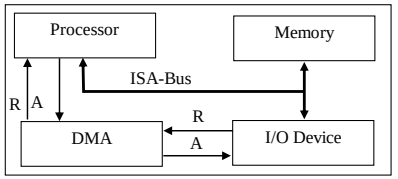


Peripheral to memory transfer with DMA (cont')

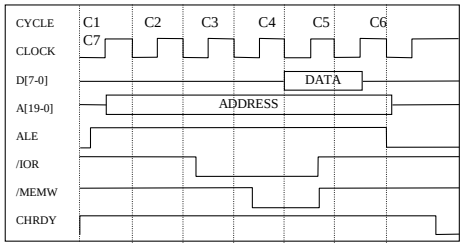
6: DMA de-asserts *Dreq* and *ack* completing the handshake with P1.



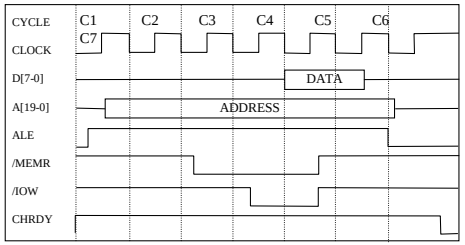
ISA bus DMA cycles



DMA Memory-Write Bus Cycle

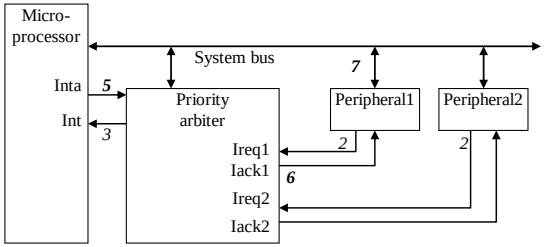


DMA Memory-Read Bus Cycle

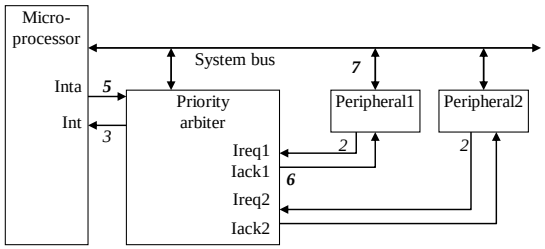


Arbitration: Priority arbiter

- Consider the situation where multiple peripherals request service from single resource (e.g., microprocessor, DMA controller) simultaneously - which gets serviced first?
- Priority arbiter
  - Single-purpose processor
  - Peripherals make requests to arbiter, arbiter makes requests to resource
  - Arbiter connected to system bus for configuration only



Arbitration using a priority arbiter



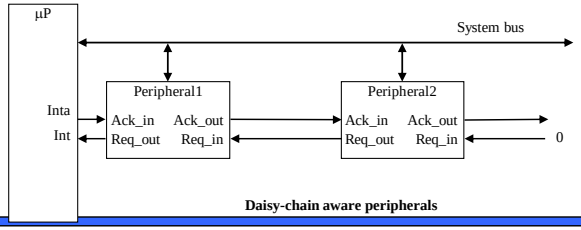
1. 1. Microprocessor is executing its program.
2. 2. Peripheral1 needs servicing so asserts *Ireq1*. Peripheral2 also needs servicing so asserts *Ireq2*.
3. 3. Priority arbiter sees at least one *Ireq* input asserted, so asserts *Int*.
4. 4. Microprocessor stops executing its program and stores its state.
5. 5. Microprocessor asserts *Inta*.
6. 6. Priority arbiter asserts *Iack1* to acknowledge Peripheral1.
7. 7. Peripheral1 puts its interrupt address vector on the system bus
8. 8. Microprocessor jumps to the address of ISR read from data bus, ISR executes and returns
9. (and completes handshake with arbiter).
10. 9. Microprocessor resumes executing its program.

Arbitration: Priority arbiter

- Types of priority
  - Fixed priority
    - each peripheral has unique rank
    - highest rank chosen first with simultaneous requests
    - preferred when clear difference in rank between peripherals
  - Rotating priority (round-robin)
    - priority changed based on history of servicing
    - better distribution of servicing especially among peripherals with similar priority demands

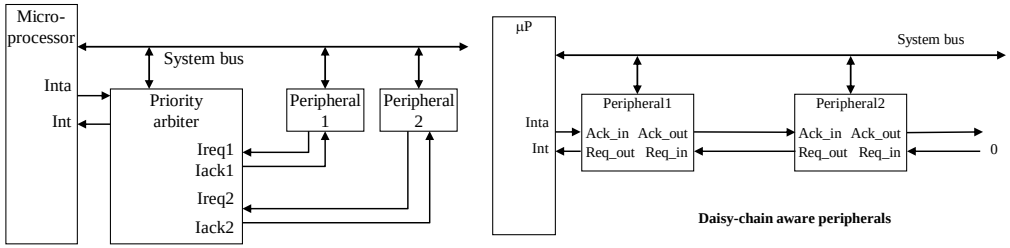
Arbitration: Daisy-chain arbitration

- Arbitration done by peripherals
  - Built into peripheral or external logic added
    - *req* input and *ack* output added to each peripheral
- Peripherals connected to each other in daisy-chain manner
  - One peripheral connected to resource, all others connected "upstream"
  - Peripheral's *req* flows "downstream" to resource, resource's *ack* flows "upstream" to requesting peripheral
  - Closest peripheral has highest priority



Arbitration: Daisy-chain arbitration

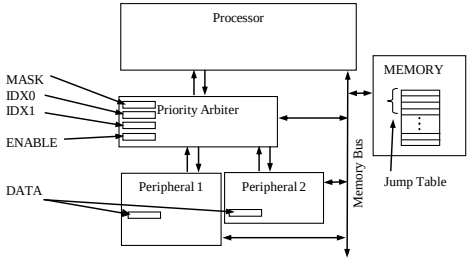
- Pros/cons
  - Easy to add/remove peripheral - no system redesign needed
  - Does not support rotating priority
  - One broken peripheral can cause loss of access to other peripherals



Network-oriented arbitration

- When multiple microprocessors share a bus (sometimes called a network)
  - Arbitration typically built into bus protocol
  - Separate processors may try to write simultaneously causing collisions
    - Data must be resent
    - Don't want to start sending again at same time
      - statistical methods can be used to reduce chances
- Typically used for connecting multiple distant chips
  - Trend – use to connect multiple on-chip processors

Example: Vectored interrupt using an interrupt table

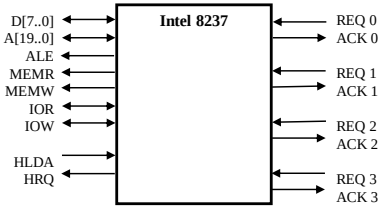


- Fixed priority: i.e., Peripheral1 has highest priority
- Keyword "_at_" followed by memory address forces compiler to place variables in specific memory locations
  - e.g., memory-mapped registers in arbiter, peripherals
- A peripheral's index into interrupt table is sent to memory-mapped register in arbiter
- Peripherals receive external data and raise interrupt

```
unsigned char ARBITER_MASK_REG              _at_ 0xfff0;
unsigned char ARBITER_CH0_INDEX_REG         _at_ 0xfff1;
unsigned char ARBITER_CH1_INDEX_REG         _at_ 0xfff2;
unsigned char ARBITER_ENABLE_REG            _at_ 0xfff3;
unsigned char PERIPHERAL1_DATA_REG          _at_ 0xffe0;
unsigned char PERIPHERAL2_DATA_REG          _at_ 0xffe1;
unsigned void* INTERRUPT_LOOKUP_TABLE[256] _at_ 0x0100;

void main() {
       InitializePeripherals();
       for(;;) {} // main program goes here
}
```

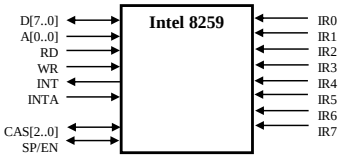
```
void Peripheral1_ISR(void) {
       unsigned char data;
       data = PERIPHERAL1_DATA_REG;
       // do something with the data
}
void Peripheral2_ISR(void) {
       unsigned char data;
       data = PERIPHERAL2_DATA_REG;
       // do something with the data
}
void InitializePeripherals(void) {
       ARBITER_MASK_REG = 0x03; // enable both channels
       ARBITER_CH0_INDEX_REG = 13;
       ARBITER_CH1_INDEX_REG = 17;
       INTERRUPT_LOOKUP_TABLE[13] = (void*)Peripheral1_ISR;
       INTERRUPT_LOOKUP_TABLE[17] = (void*)Peripheral2_ISR;
       ARBITER_ENABLE_REG = 1;
}
```


Intel 8237 DMA controller



Signal	Description
D[7..0]	These wires are connected to the system bus (ISA) and are used by the microprocessor to write to the internal registers of the 8237.
A[19..0]	These wires are connected to the system bus (ISA) and are used by the DMA to issue the memory location where the transferred data is to be written to. The 8237 is
ALE*	This is the address latch enable signal. The 8237 use this signal when driving the system bus (ISA).
MEMR*	This is the memory write signal issued by the 8237 when driving the system bus (ISA).
MEMW*	This is the memory read signal issued by the 8237 when driving the system bus (ISA).
IOR*	This is the I/O device read signal issued by the 8237 when driving the system bus (ISA) in order to read a byte from an I/O device
IOW*	This is the I/O device write signal issued by the 8237 when driving the system bus (ISA) in order to write a byte to an I/O device.
HLDA	This signal (hold acknowledge) is asserted by the microprocessor to signal that it has relinquished the system bus (ISA).
HRQ	This signal (hold request) is asserted by the 8237 to signal to the microprocessor a request to relinquish the system bus (ISA).
REQ 0,1,2,3	An attached device to one of these channels asserts this signal to request a DMA transfer.
ACK 0,1,2,3	The 8237 asserts this signal to grant a DMA transfer to an attached device to one of these channels.
*See the ISA bus description in this chapter for complete details.	

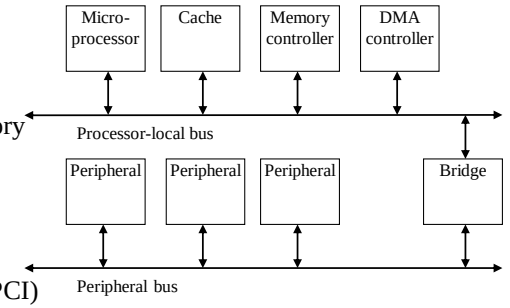
Intel 8259 programmable priority controller



Signal	Description
D[7..0]	These wires are connected to the system bus and are used by the microprocessor to write or read the internal registers of the 8259.
A[0..0]	This pin actis in cunjunction with WR/RD signals. It is used by the 8259 to decipher various command words the microprocessor writes and status the microprocessor wishes to read.
WR	When this write signal is asserted, the 8259 accepts the command on the data line, i.e., the microprocessor writes to the 8259 by placing a command on the data lines and asserting this signal.
RD	When this read signal is asserted, the 8259 provides on the data lines its status, i.e., the microprocessor reads the status of the 8259 by asserting this signal and reading the data lines.
INT	This signal is asserted whenever a valid interrupt request is received by the 8259, i.e., it is used to interrupt the microprocessor.
INTA	This signal, is used to enable 8259 interrupt-vector data onto the data bus by a sequence of interrupt acknowledge pulses issued by the microprocessor.
IR 0,1,2,3,4,5,6,7	An interrupt request is executed by a peripheral device when one of these signals is asserted.
CAS[2..0]	These are cascade signals to enable multiple 8259 chips to be chained together.
SP/EN	This function is used in conjunction with the CAS signals for cascading purposes.

Multilevel bus architectures

- Don't want one bus for all communication
  - Peripherals would need high-speed, processor-specific bus interface
    - excess gates, power consumption, and cost; less portable
  - Too many peripherals slows down bus
- Processor-local bus
  - High speed, wide, most frequent communication
  - Connects microprocessor, cache, memory controllers, etc.
- Peripheral bus
  - Lower speed, narrower, less frequent communication
  - Typically industry standard bus (ISA, PCI) for portability
- Bridge
  - Single-purpose processor converts communication between busses



Co-Simulation and Emulation

Hardware Software CoDesign

October 2011

Agenda

Co-Simulation and Emulation

1. Basic Themes that came out through the Historical Background (repeat)
2. Co-Simulation and Emulation
3. Continue on "Answering Machine" Assignment Discussions. Give clear milestones.
4. Taking Stock

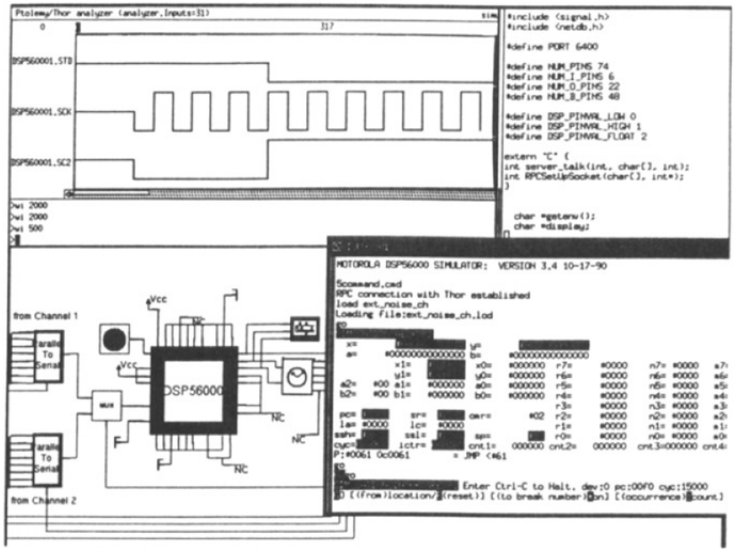
Co-Simulation and Emulation

Basic Themes

1. **Modeling** :: Because parallelism is important, the choice of an appropriate modeling paradigm is also important. Different modeling formalisms need to be developed that capture the various aspects of system behavior.
2. **Analysis and Estimation** :: Different models of the same system behavior need to be analyzed and estimated for performance. Performance would include tradeoffs for Area, Speed, Power. Cost to build, Time to Market are other factors.
3. **System-level partitioning, synthesis and interfacing** :: The basic steps of co-synthesis. A range of methodologies are applied for this.
4. **Implementation generation** :: Once the architecture has been generated and trade-offs known, the designs for the hardware and software components must be created.
5. **Co-simulation and Emulation** :: This helps designers to evaluate architectures and validate assumptions on implementations. Emulation uses FPGA / other emulation techniques to further speed up execution of system models.

Co-Simulation and Emulation

Screen Shot of CoSimulation System



Co-Simulation and Emulation

Introduction

1. *Hardware-software co-simulation* is the combination of simulation of software running on a processor hardware with the fixed-function hardware components / sub-systems.

2. From here on.. there can be various methods for co-simulation.

   (a) Do a detailed processor simulation (full RTL) → too time consuming when simulating large software programs, and too slow to bring existing components to work with simulated components. Thus there is a requirement for a set of models for existing components (called test-bench). The creation, maintenance and usage of testbench adds to the cost of the device which is being designed.

   (b) For simulating larger software programs, abstract processor models are used. (usage of behavioral models for processors).

   - *Bus Functional Model (BFM)* :: abstracts from program execution and describes the processor bus interface function and timing only (think of a donut).
   - *Cycle-accurate Model* :: executes the program instructions with the accurate number of processor clock cycles, but without detailed interface timing. Such a model allows the designer to analyze the system timing and to validate the cooperation of hardware components and processors.
   - *Instruction Set Simulator Model* :: executes the program instructions preserving the program function but completly abstracts from timing. Main applications are program validation and debugging the software program.

3. Cycle-accurate and Instruction Set simulator are well suited to compiled mode (say nc)simulation.

4. Abstract Models are simpler, faster to execute, and can be used in early phases of a design where implementation details are still open.

5. Co-simulation uses abstract models to form a virtual prototype, *co-emulation* provides a real prototype by function implementation in hardware. This prototype can be used to accelerate co-simulation, but it can also be installed in the real environment *like a hardware in the loop* to investigate the system function under real conditions.

6. The holy grail for co-simulation and emulation :: Investigation of various possibilities with different tradeoffs in hw/sw is only sensible if different hw partitions can be implemented as fast as software can be compiled into machine code. High Level of Synthesis, different abstraction models are therefore enablers for hw/sw co-design.

Co-Simulation and Emulation

Emulation System

1. An Hardware Emulator would allow the full design to be imported onto the Emulator (through a said flow) onto a board which has real components.

2. FPGAs, QuickTurn (Palladium) are different emulation system (system components) which are used for prototyping.

3. Advantage of emulation are ::
   - Higher speed, in cases it is just 100x slower than real time components.
   - Allows for Higher Level of Hardware Verification or Application Level Verification.
   - Allows for Development of Firmware.
   - Allows for testing the design and the software before silicon. (With a speed of 500,000 cycles per second, it was possible to test the boot sequence of Solaris on a UltraSparc Architecture)
   - Simulation aims at the general reduction of the number of functional and timing errors, whereas emulation in general serves realistic loads in demonstrating functional correctness.

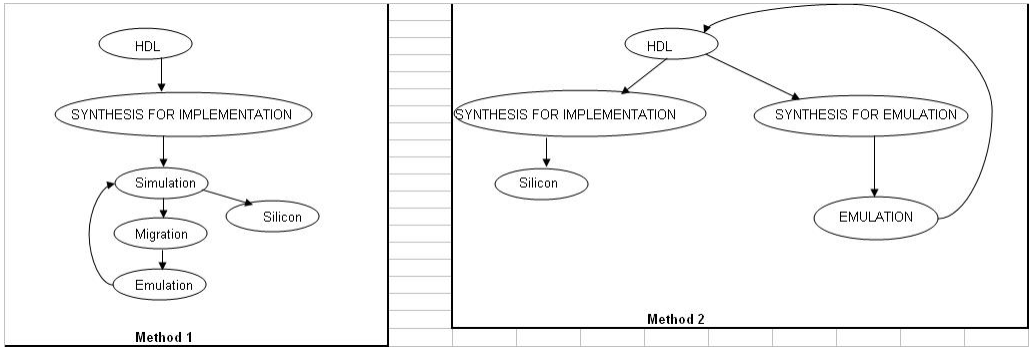
Co-Simulation and Emulation

Emulation System

1. Dis-Advantage of emulation are ::
   - Timing errors in the actual device are hard to detect on emulation platforms (this is possible in simulation system using sdf timing annotations).
   - In case of FPGAs, a separate effort has to be done to meet timing requirements for the FPGA itself.
   - Cost is higher for emulation.
   - Emulation is generally only used after thorough simulations so that there is low expectation of design errors.

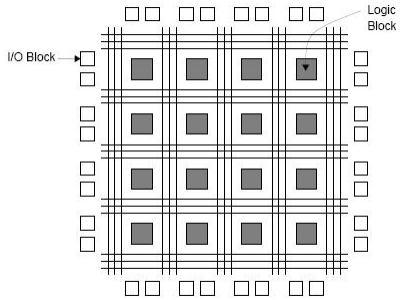
Co-Simulation and Emulation

Methods in Deploying Emulation System



Co-Simulation and Emulation

FPGA Based System

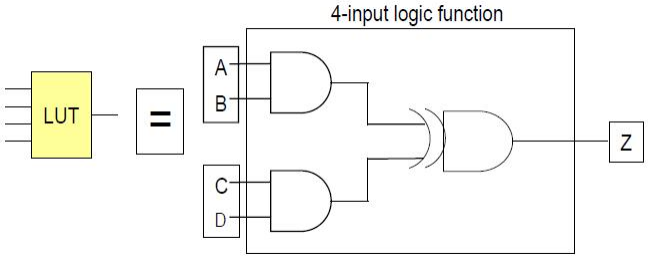


1. The Static RAMs based implementation provides unlimited programmability to FPGAs.
2. FPGAs use building blocks called CLB (configurable logic block),
3. Add Dedicated Memory Blocks,
4. Clocking Blocks,
5. With Interconnects to connect one CLB, Memories, Clocks, IO to each another, and

6. An IO interface to talk to external (real) components.
7. GROUP DISCUSSION :: Discuss that a 4 input LUT can serve as a universal gate.
8. There can be architectures where the CLB itself can serve as a Memory Block.

Co-Simulation and Emulation

FPGA Based System

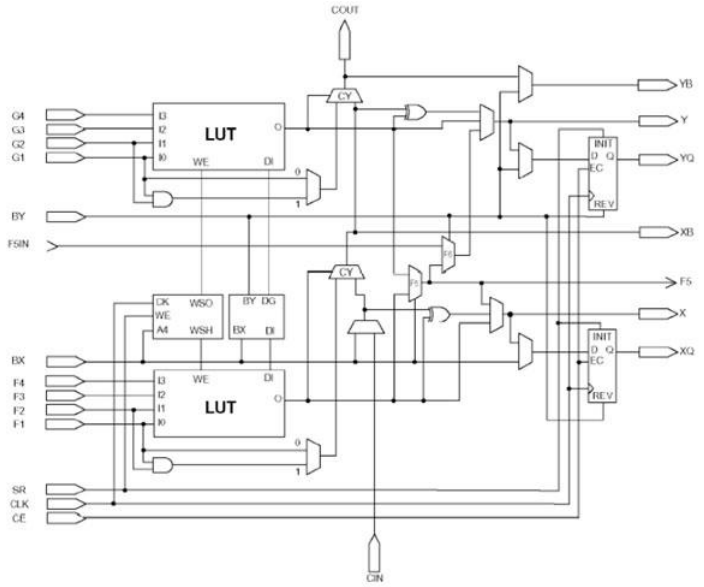


Inputs(ABCD)	Output(Z)
0000	0
0001	0
0010	0
0011	1
.....	..
1110	1
1111	0

1. GROUP DISCUSSION :: Discuss that a 4 input LUT can serve as a universal combo gate.

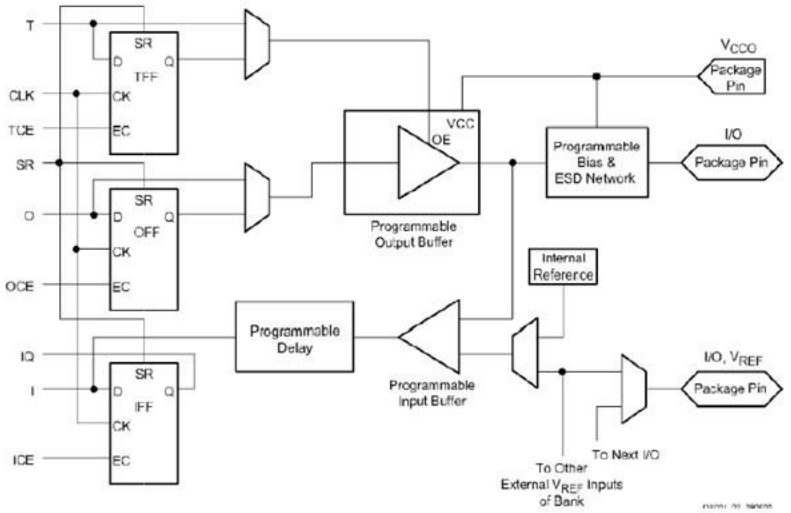
Co-Simulation and Emulation

FPGA Based System – CLB of a Xilinx FPGA



Co-Simulation and Emulation

FPGA Based System – IO of a Xilinx FPGA



Co-Simulation and Emulation

FPGA Based System

1. FPGAs also come with various capacity tags, which are suitable for different ranges of devices being built till date.

2. There exists systems and software which use "arrays" of FPGAs for larger SoC's or "chip-sets"

3. As FPGAs evolve through the technology nodes, capacity naturally increases and increasingly FPGA vendors are able to integrate common CPUs (like ARM), common blocks like DSP MAC units, Communication physical layer (Phy) like Serdes, Ethernet Phy, USB Phy, differential IO's.

4. Unlike Emulation systems, FPGAs are also being used for doing field trials i.e, in an environment where the actual ASIC would be. Thus portability is also become a requirement for prototyping.

Co-Simulation and Emulation

Emulation Based System – Example of Palladium XP – From Cadence Website

Co-Simulation and Emulation

Emulation Based System – Example of Palladium XP – From Cadence Website

1. Unparalleled operational efficiency and user flexibility with highly scalable systems
2. Delivers high performance of up to 4MHz
3. Allows flexible configurations from 4 million gates to 2 billion gates
4. Supports up to 512 users simultaneously
5. Advanced compiler and runtime capabilities
6. Integrates seamlessly with Incisive products
7. Offers a unified, advanced debug environment for HW/SW co-verification
8. Supports SCE-MI and SystemVerilog DPI for third-party models/tools integration
9. Unique platform extensions
10. Supports metric-driven verification acceleration
11. Supports industry-standard hardware design and verification languages and the Open Verification Methodology

12. Enables Dynamic Power Analysis and verification by integrating with the Encounter RTL Compiler power estimation engine
13. SystemC-to-emulation flow allows users to integrate high-level abstraction models into the system verification environment
14. Integrates with the comprehensive SpeedBridge family of rate adapters and the Cadence Verification IP portfolio

Co-Simulation and Emulation

Emulation Based System – Example of Palladium XP – From Cadence Website

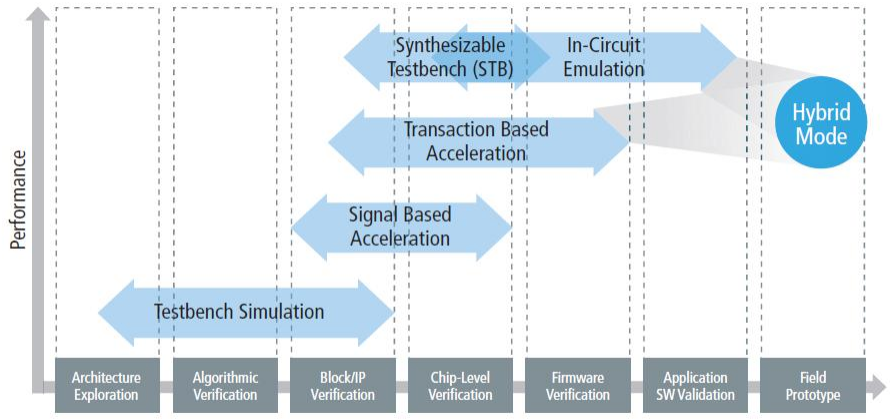
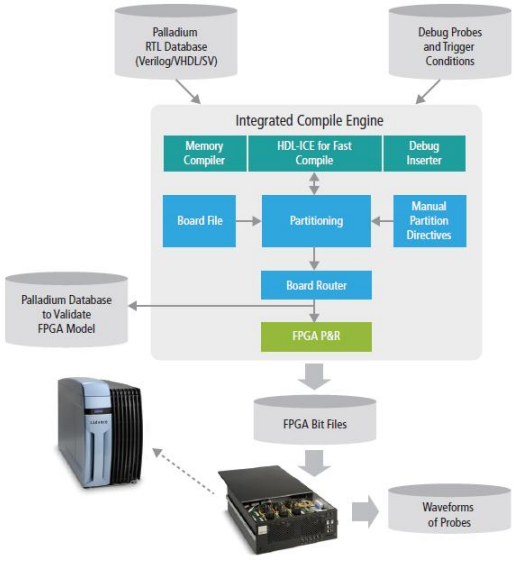


Figure 7: The Palladium XP Verification Computing Platform stretches through much of the hardware/software development cycle

1. **GROUP DISCUSSION** :: Compare an Emulator Vs FPGA

Co-Simulation and Emulation

Emulation Based System – From Palladium XP – Rapid FPGA Based System – From Cadence Website



Continue on "Answering Machine" Assignment Discussions

Marking Milestones

1. Take a break for 5mins.
2. Implementation Documentation – Hardware and Software components, with all IPś in place, assumptions et. all → November 13 Deadline (10 marks)
3. Actual Implementation, documentation updates and enabling me to replay with my own set of tests → December 30 Deadline (10 marks)
4. Selection of key persons from the team (groups)
   (a) IP creation / reuse
   (b) System Hardware Integration
   (c) System Software Integration
   (d) TestBench Creation and System Checkout → Hardware & Software
   (e) Total Hardware / software → bringing it all together

Continue on "Answering Machine" Assignment Discussions

Requirements Till date

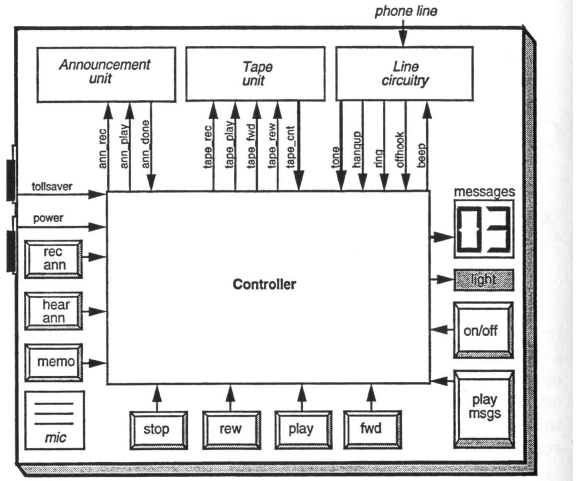
1. System C model of the system.
2. Use models for analog components.
3. Should have System C testbench as a part of SoC verification.
4. ReUse preferred to creation of already existing components.

Solution components / discussion

1. Decide Microprocessor.
2. How to write software program.
3. Memory requirements.
4. Model of ADC required (file input and digital output).
5. Input Buttons.
6. On/Off Button - special case.
7. Model of LEC (7 segment display).
8. Some real time clock – keep date and time (simplification is ok)
9. Each message has upper bound (10sec) – min 16 messages.
10. Announcement = single 10 seconds.
11. Announcement hear out through microphone (DAC)
12. Light keeps blinking if there is a new message.

"Answering Machine" Assignment

Diagram of Use Case



The answering machine controller's environment.

"Answering Machine" Assignment

Next Steps for Implementation document

1. Entity Definition
2. Processor / Dedicated Hardware decision
3. System State Machine or StateChart diagram (gross functionality)
4. Individual functionality State Macine or StateChart diagram

"Answering Machine" Assignment

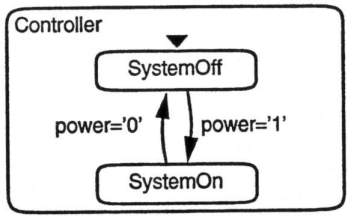
Next Steps for Implementation document – Entity Definition

```
entity Controller_E is
(
    -- Interface to Announcement unit
    ann_rec      : out bit;      -- 1 causes unit to record
    ann_play     : out bit;      -- 1 causes unit to play
    ann_done     : in bit;       -- 1 indicates end of announcement

    -- Interface to Tape unit
    tape_rec     : out bit;      -- 1 causes unit to record
    tape_play    : out bit;      -- 1 causes unit to play
    tape_fwd     : out bit;      -- 1 causes unit to forward
    :
    :
);
end entity;

architecture Controller_A of Controller_E is
  ...
```

The answering machine controller's interface.

"Answering Machine" Assignment

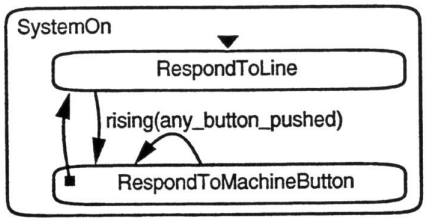
Next Steps for Implementation document – Function Level State Charts – ON/OFF



Highest-level view of the controller

"Answering Machine" Assignment

Next Steps for Implementation document – Function Level State Charts



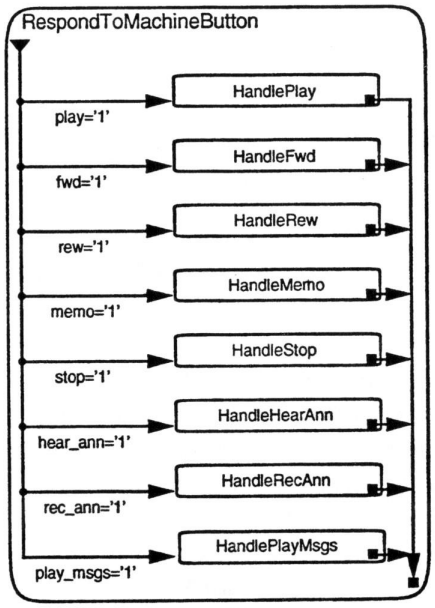
The SystemOn behavior

"Answering Machine" Assignment

Next Steps for Implementation document – Function Level State Charts

```
behavior RespondToMachineButton
  type code is
begin
   if (play='1') then
      HandlePlay;
   elsif (fwd='1') then
      HandleFwd;
   elsif (rew='1') then
      HandleRew;
   elsif (memo='1') then
      HandleMemo;
   elsif (stop='1') then
      HandleStop;
   elsif (hear_ann='1') then
      HandleHearAnn;
   elsif (rec_ann='1') then
      HandleRecAnn;
   elsif (play_msgs='1') then
      HandlePlayMsgs;
   end if;
end;
```

(a)



(b)

The RespondToMachineButton behavior: (a) sequential statements, (b) equivalent subbehavior decomposition.

Taking Stock

1. Historical perspective – from microprocessor based designs to current SoCs.
2. A Multiplier example to illustrate the tradeoffs.
3. Typical HW/SW co-design flow.
4. 5-Basic Themes for hw/sw co-design.
5. Example of a TouchScreen System. Defining different parts of the system.
6. Modeling – Basic Modeling Requirements for co-design.
7. Modeling – Modeling using SystemC.
8. Analysis and Estimation – Path based enumeration technique.
9. Analysis and Estimation – Rate Monotonic Analysis for Single CPU, Deadline Driven analysis, Mixed Scheduling Algorithm.
10. Partitioning, Synthesis and Interfacing – GCLP, MIBS algorithms (Kalavade & Lee)
11. Example of converting a C based algorithm to HW (GCD, Matrix Multiplication)
12. CoSimulation and Emulation – FPGA based, Emulator based

Co-Simulation and Emulation

Acknowledgements

1. Readings in Hardware / Software Co-design ::G. D. Micheli, Rolf Ernst, Wayne Wolf :: Ch 6
2. Hardware / Software Co-Design - Principles and Practice :: J. Staunstrup & W. Wolf :: Ch 3
3. Fpga Prototyping Methodology Manual :: Ch 3
4. http://www.cadence.com/products/sd/palladium_xp/Pages/default.aspx

Compiler, Linker, Loader

Hardware Software CoDesign

November 2011

Agenda

Compiler, Linker, Loader

1. Basic Definitions of a Compiler, Linker, Loader
2. Challenges in Compiler design
3. Basic requirements a compiler must fulfill
4. Brief on Compiler Architecture
5. Challenges for Compiler design for Embedded processors
6. Concept of a ReTargettable Compiler
7. Discussion on the Answering Machine Assignment (Group wise)
8. If time permits, open up the ARM CortexM3 processor documentation and readup.

Compiler, Linker, Loader

Introduction to Compiler

1. **What is a Compiler?** :: A *compiler* is a program that transforms a source program written in some high-level programming language (such as C) into machine code for some computer architecture (such as ARM).
2. The generated machine code can be later executed many times against different data each time.
3. The translation (transformation) of the source code from a high-level programming language to an executable or machine code can happen in steps :-
   - conversion into the target language, often having a binary form called object code.
   - conversion from object code into machine code (executable)
4. If the compiled program can run on a computer whose CPU or OS is different from the one on which the compiler runs, the compiler is known as a *cross-compiler*.
5. Compare this with an *interpreter* which reads an executable source program written in a high-level programming language as well as data for this program, and it runs the program against the data to produce some results. One example is the Unix Shell intrepretor, which runs operating systems commands interactively.
6. GROUP DISCUSSION ::
   - Enumerate available compilers.
   - Why do we need compilers in the first place.

Compiler, Linker, Loader

What are the Challenges? – Many Variations

1. many programming languages (eg. FORTRAN, C, C++, Java)
2. many programming paradigms (eg. object-oriented, procedural, functional, logic)
3. many computer architectures (eg. MIPS, ARM, SPARC, Intel, SHARC)
4. many operating systems (eg. Linux, Solaris, Windows, Android)

Compiler, Linker, Loader

What are the Qualities of a Compiler?

1. the compiler itself must be bug-free
2. it must generate correct machine code
3. the generated machine code must run fast
4. the compiler itself must run fast (compilation time must be proportional to program size)
5. the compiler must be portable (ie., modular, supporting separate compilation)
6. it must print good diagnostics and error messages
7. the generated code must work well with existing debuggers
8. must have consistent and predictable optimization

Compiler, Linker, Loader

What is requirement for design of a Compiler?

Knowledge of :-

1. programming languages (parameter passing, variable scoping, memory allocation...)
2. automata theory
3. algorithms and data structures (hash tables, graph algorithms, dynamic programming...)
4. computer architecture (assembly programming)
5. software engineering

Compiler, Linker, Loader

Brief on Compiler Architecture

1. Suppose you want to build compilers for m programming languages and you want to run them on n different architectures.
2. If it is done natively, there is a requirement to write $m * n$ compilers, one for each language-architecture combination.
3. The HOLY GRAIL of portability in compilers is to do the same thing by writing $m+n$ programs only. HOW ?
4. Define and use a universal *Intermediate Representation (IR)* and make the compiler two phases. An IR is typically a tree-like data structure that captures the basic features of most computer architectures.
5. Eg. representation of say an instruction $d \leftarrow s_1 + s_2$, that gets two source numbers and produces one destination number.
6. The front-end of the compiler maps the source code into IR and the second phase, called back-end, maps IR into machine code.
7. If this is followed, ideally there will be only $m + n$ components.
8. The challenge is to define the IRs such that all language and machine features are captured properly.

Compiler, Linker, Loader

Brief on Compiler Architecture – Multiple Phases in Front-End and Back-End

A typical real-world compiler has multiple phases. This increases the compiler's portability and simplifies re-targetting.

The following phases in front-end :-

1. *scanning:* a scanner groups input characters into tokens;
2. *parsing:* a parser recognizes sequences of tokens according to some grammar and generates *Abstract Syntax Trees (ASTs)*;
3. *Semantic analysis:* performs *type checking* (ie., checking whether the variables, functions etc in the source program are used consistently with their definitions and with the language semantics0 and translates ASTs into IRs;
4. *Optimization:* optimization IRs.

The following phases in the back-end :-

1. *instruction selection:* maps IRs into assembly code;
2. *code optimization:* optimizes the assembly code using control-flow and data-flow analysis, register allocation, etc;
3. *code emission:* generates machine code from assembly code.

Compiler, Linker, Loader

Brief on Compiler Architecture – Multiple Phases in Front-End and Back-End

The generated machine code in written in an object file. This file is not executable since it may refer to external symbols (such as system calls). The operating system provides the following utilities to execute the code (in a computer system – this is slightly different in an embedded system).

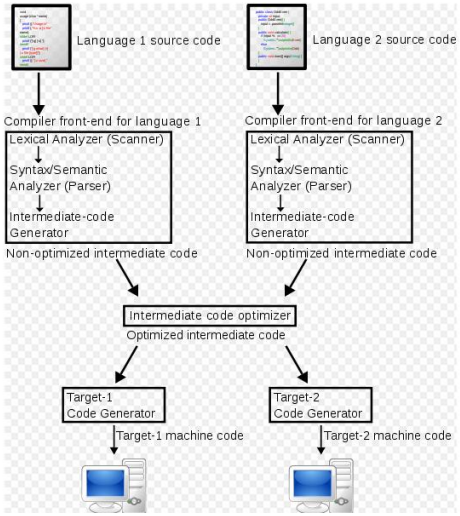
1. *linking:* A linker takes several object files and libraries as input and produces one executable object file.

   → It retrieves from the input files (and puts them together in the executable object file) the code of all the referenced functions / procedures and it resolves all external references to real addresses. The libraries include the operating system libraries, the language- specific libraries, and user-created libraries.

2. *loading:* A loader loads an executable object file into memory, initializes the registers, heap, data, etc and starts the execution of the program.

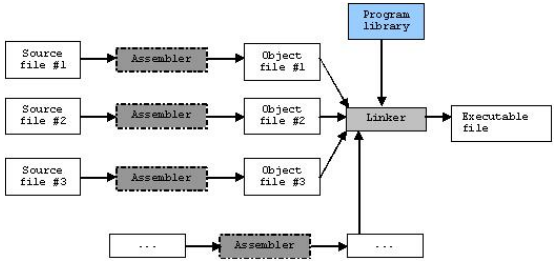
Compiler, Linker, Loader

Compiler Architecture – Multiple Phases in Front-End and Back-End



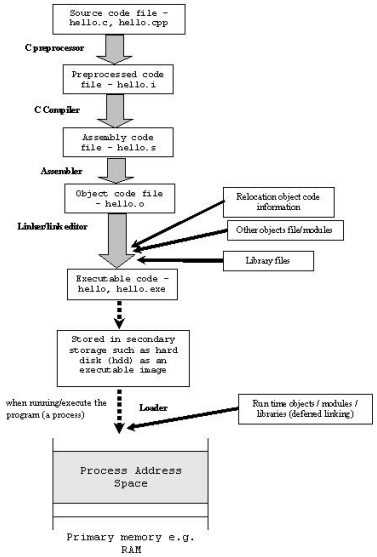
Compiler, Linker, Loader

Compiler Architecture – Multiple Phases in Front-End and Back-End



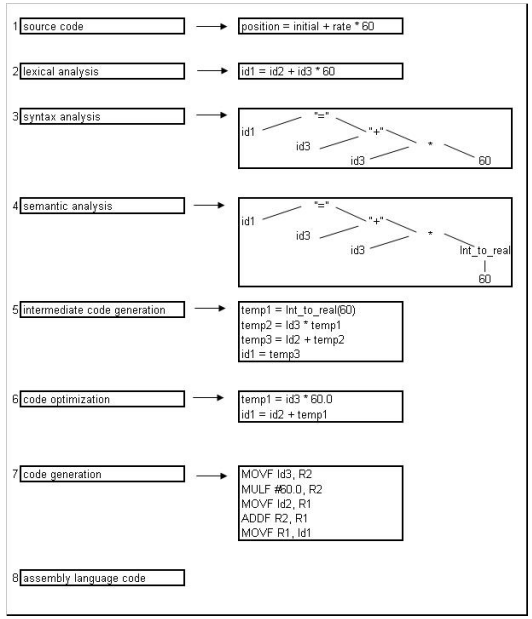
Compiler, Linker, Loader

Compiler Architecture – Multiple Phases in Front-End and Back-End



Compiler, Linker, Loader

Brief on Compiler Architecture – Steps in Compilation



Compiler, Linker, Loader

Compiler Architecture – Problem for Compilation to Embedded Processors

1. *ReTargetability:* Typically, retargetting to a new architecture is confined to the final code generation phase. This means that the IR (intermediate representation) must closely represent the final target machine to produce efficient code.

   → In a scenario where there are "Application Specific Instruction Sets" and where the embedded processor instruction sets vary widely in composition, a general (generic) form of IR can be difficult to conceptualize.

2. *Register Constraints:* Embedded processors contain a number of special purpose registers. Registers are reserved for special functionality to maintain low instruction widths. The instruction width directly correlates with the program space. The register constraints can affect all the phases of compilation.

3. *Arithmetic Specialization:* The typical 3-tupule code artificially decomposes data-flow operation into smaller pieces. Arithmetic operations which require more than 3 operands are not naturally handled with 3-tupule code. However, greater than 3 operands occur frequently in DSP architecture.

4. *Instruction-Level Parallelizm:* Architectures with parallel executing engines require different compilation techniques. eg. A DSP typically has both Data Calculation Unit (DCU) and Address Calculation Unit (ACU).

   → A compiler should take into account the possibility to perform operations on different functional units, as well as choose the most compact solution.

5. *Optimization:* Real Time embedded software cannot afford to have performance penalties as a result of poor compilation. Efficient compilation is only arrived upon by many optimization algorithms.

Compiler, Linker, Loader

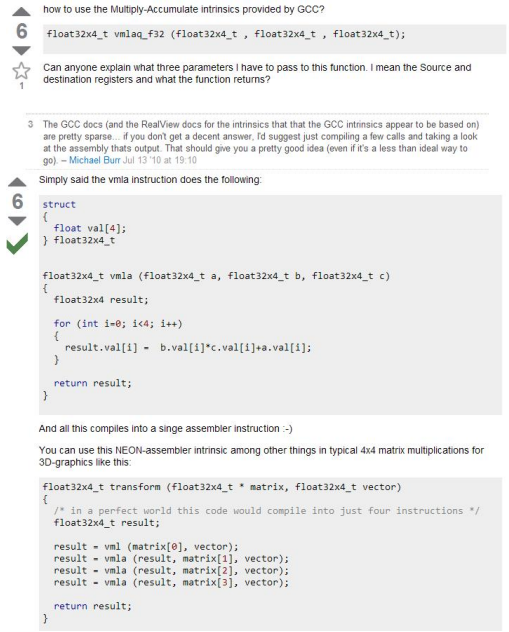
Compiler ReTargetability

Take an Example of a Typical compiler (say gcc)

1. Freely distributed through the FSF (Free Software Foundation)
2. With free C source code, its been ported to many machines – Intel, Sun..
3. Also its been retargetted to several DSPs like ADI2101, Lucent1610, Motorola56001, STD950,...
4. GCC has become a de-facto approach to develop compilers quickly from freely available sources.
5. GCC strengts lie in the complete set of architecture independent optimizations: common subexpression removal, dead code elimination, constant folding, constant propagation, basic code execution and others.
6. HOWEVER, for embedded processing, it is important that optimizations be applied according to the character of target architectures (and ability to exploit the architecture). GCC has little provisions for enabling optimizations to the target machine. Hence performance falls short of acceptable code quality.
7. Embedded processors have usually few registers, heterogeneous register structures, unusual wordlengths, and other architectural specializations.
8. Till Recently, a formal re-targetability model was not fully adopted primarily because general purpose CPUs have long lifetimes that do not justify the effort required to make the compilers re-targetable.

Compiler, Linker, Loader

Compiler ReTargetability – Example from recent times



how to use the Multiply-Accumulate intrinsics provided by GCC?

6 1

```
float32x4_t vmlaq_f32 (float32x4_t , float32x4_t , float32x4_t);
```

Can anyone explain what three parameters I have to pass to this function. I mean the Source and destination registers and what the function returns?

3 The GCC docs (and the RealView docs for the intrinsics that that the GCC intrinsics appear to be based on) are pretty sparse... if you don't get a decent answer, I'd suggest just compiling a few calls and taking a look at the assembly thats output. That should give you a pretty good idea (even if it's a less than ideal way to go). – Michael Burr Jul 13 '10 at 19:10

6

Simply said the vmla instruction does the following:

```
struct
{
  float val[4];
} float32x4_t


float32x4_t vmla (float32x4_t a, float32x4_t b, float32x4_t c)
{
  float32x4 result;

  for (int i=0; i<4; i++)
  {
    result.val[i] =  b.val[i]*c.val[i]+a.val[i];
  }

  return result;
}
```

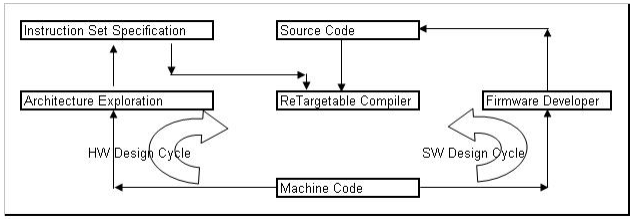
And all this compiles into a singe assembler instruction :-)

You can use this NEON-assembler intrinsic among other things in typical 4x4 matrix multiplications for 3D-graphics like this:

```
float32x4_t transform (float32x4_t * matrix, float32x4_t vector)
{
  /* in a perfect world this code would compile into just four instructions */
  float32x4_t result;

  result = vml (matrix[0], vector);
  result = vmla (result, matrix[1], vector);
  result = vmla (result, matrix[2], vector);
  result = vmla (result, matrix[3], vector);

  return result;
}
```

Compiler, Linker, Loader

ReTargetable Compiler – concept



1. Retargetability allows the rapid setup of a compiler to a specific processor. This can be an enormous boost for algorithm developers wishing to evaluate the efficiency of application code on different existing architectures.
2. Retargetability permits architecture exploration. The processor designer is able to tune his architecture to run efficiently for a set of source applications in a particular domain.
3. This however, requires the need for *instruction set specification languages* where the user can describe the functionality of a processor in a formal fashion.
4. Then, the transformations of a compiler may be returned according to the architecture model (or definition).
5. One example of a retargetable compiler is MIMOLA - Machine Independent Microprogramming LAnguage.

Compiler, Linker, Loader

Example of Register Allocation – concept

Example - For the following 2 statement program segment, determine a smart register allocation scheme:

$A = B + C * D$

$B = A - C * D$

Simple Register Allocation
LOD (R1,C)
MUL (R1,D)
STO (R1,Temp)
LOD (R1,B)
ADD (R1,Temp)
STO (R1,A)
LOD (R1,C)
MUL (R1,D)
STO (R1,Temp2)
LOD (R1,A)
SUB (R1,Temp2)
STO (R1,B)
Net Result
12 instructions and memory ref.

Smart Register Allocation	
LOD (R1,C)	
MUL (R1,D)	**C*D**
LOD (R2,B)	
ADD (R2,R1)	**B+C*D**
STO (R2,A)	
SUB (R2,R1)	**A-C*D**
STO (R2,B)	
Net Result	
7 instruc. And 5 mem. refs.	

Compiler, Linker, Loader

Acknowledgements

1. Hardware / Software Co-Design - Principles and Practice :: J. Staunstrup & W. Wolf :: Ch 5
2. http://lambda.uta.edu/cse5317/notes/notes.html
3. A retargetable compiler for a high-level microprogramming language :: Peter Marwedel :: Proceedings of the 17th annual workshop on Microprogramming, 1984.
4. ARM Assembler Reference :: www.arm.com

ARM CortexM3 – Programmers view and Development Environment

Hardware Software CoDesign

November 2011

Agenda

ARM CortexM3 – Programmers view and Development Environment

1. Basic information on ARM CortexM3
2. Programmers view of CortexM3 (refer as M3)
3. Discussion on the Answering Machine Assignment (Group wise 1, 2, 3, 4)
4. Mid Semester Paper distribution

ARM CortexM3 – Programmers view

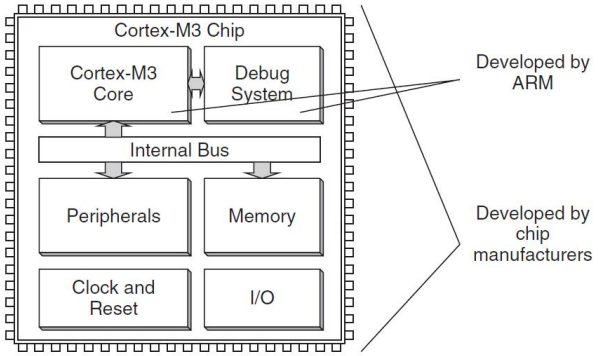
Introduction to ARM CortexM3

1. Primarily designed to target the 32bit Microcontoller market
2. Great performance at low cost and many new features available only in high-end processors
3. Enhanced determinism, guaranteeing that critical tasks and interrupts are serviced as quickly as possible, but in a "known" number of cycles.
4. Improve code density, ensuring that code fits even the smallest memory footprints
5. Ease of use, providing debugability and easy programmability for those applications which are migrating from 8, 16bit to 32bit.
6. Can be used in "device aggregation", where multiple traditional 8bit devices can get replaced by a single 32bit high performance device.
7. Through the compilers, the amount of code reuse across other ARM systems can take place.

ARM CortexM3 – Programmers view

Introduction to ARM CortexM3

1. The use model for ARM and other integrators is :-

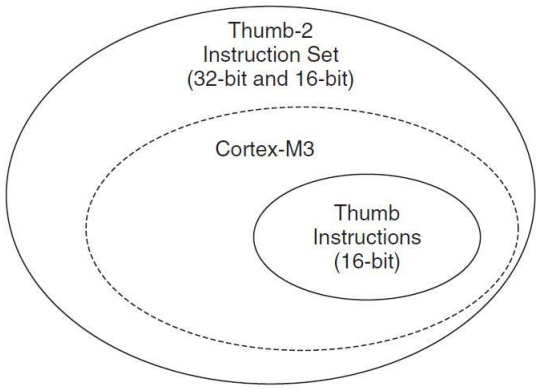


2. In general, ARM has come up with Cortex family like :-
   - **A family** :: designed for high-performance *application* platforms
   - **R family** :: designed for high-end embedded systems in which *Real-Time* performance is needed
   - **M family** :: designed for deeply embedded *Microcontroller* systems

ARM CortexM3 – Programmers view

Introduction to ARM CortexM3 – ARM and Thumb Instruction Sets

1. There are two different types of instruction sets
   (a) 32bit called *ARM* instruction set
   (b) 16bit called *Thumb* instruction set
2. During program execution, the processor can be dynamically switched between the ARM state or Thumb state to use either of the instruction sets
3. The Thumb instruction set provides only a subset of the ARM instructions, but can provide high code density.
4. M3 processor supports only the Thumb-2 (and traditional Thumb) instruction set. It uses Thumb-2 instruction set for all operations



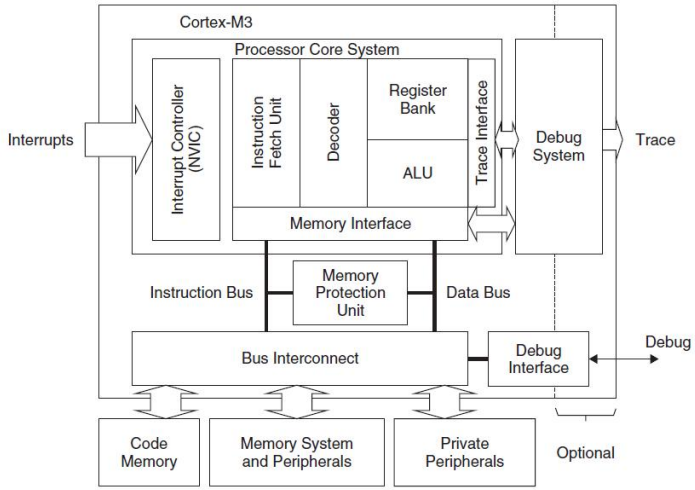
The Relationship Between the Thumb-2 Instruction Set and the Thumb Instruction Set

1. The processor has a Harvard architecture, ie., has a separate instruction bus and data bus. However, the instruction and data buses share the same memory space (unified memory system). i.e, Cannot get 8GB space just because there are separate bus interfaces.

2. *GROUP DISCUSSION* :: How does Harvard architecture help M3 ?

ARM CortexM3 – Programmers view

Introduction to ARM CortexM3 – Basic Block Diagram



A Simplified View of the Cortex-M3

ARM CortexM3 – Programmers view

Introduction to ARM CortexM3 – General Purpose Register Set

1. The M3 has GP registers *R0 to R15*

2. *General Purpose Registers* :: R0 to R12

3. *Stack Pointer* :: R13 → banked R13 register
   (a) *Main Stack Pointer MSP* - Default StackPointer, used by the OS kernel and exception handlers
   (b) *Process Stack Pointer PSP* - Used by the user application code

4. *Link Register* :: R14 → When a subroutine is called, the return address is stored in the link register.

5. *Program Counter* :: R15 → The current program address. This register can be written to control the program flow

ARM CortexM3 – Programmers view

Introduction to ARM CortexM3 – Operation Modes

1. The M3 has 2 Operation modes and 2 privilege levels

2. *Operation Modes* :: What kind of operation the M3 is doing. Whether the processor is running a normal program or running an exception handler like an interrupt handler or system exception handler.

   (a) *Thread Mode* :: Thread mode is entered on reset, and can be entered as a result of an exception return. Privileged and User code can run in Thread mode.

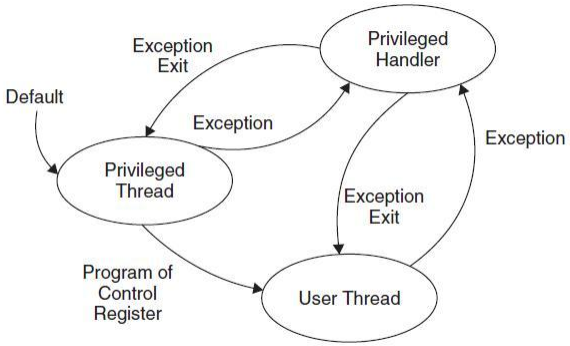
   (b) *Handler Mode* :: Handler mode is entered as a result of an exception. All code is privileged in Handler mode.

3. *Privilege Levels* :: provide a mechanizm for safeguarding memory acces to critical regions as well as provide a basic security model.

   (a) *Privilege Level* ::

   (b) *User Level* ::

	Privileged	*User*
When running an exception	Handle Mode	
When running main program	Thread Mode	Thread Mode



Allowed Operation Mode Transitions

ARM CortexM3 – Programmers view

Introduction to ARM CortexM3 – BuiltIn Nested Vectored Interrupt Controller

1. The M3 Processor includes an Interrupt Controller called the *NVIC (Nested Vectored Interrupt Controller* with following main features :-

   (a) *Nested Interrupt Support* :: All interrupts and most of the system exceptions can be programmed to different priority levels. When an interrupt occurs, the NVIC compares the priority of this interrupt to the current running priority level. If the priority of the new interrupt is higher than the current level, the interrupt handler of the new interrupt will override the current running task.

   (b) *Vectored Interrupt Support* :: When an interrupt is accepted, the starting address of the ISR is located from a vector table in memory. There is no need to use software to determine and branch to the starting address of the ISR. Thus it takes less time to process the interrupt request.

   (c) *Dynamic Priority changes Support* :: Priority levels of interrupts can be changed by software during run time. Interrupts that are being serviced are blocked from further activation until the ISR is completed, so their priority can be changed without risk of accidental re-entry.

   (d) *Reduction of Interrupt Latency* :: M3 includes automatic saving and restoring some register contents, reducing delay in switching from one ISR to another and handling late arrival interrupts.

   (e) *Interrupt Masking* :: Interrupts and system exceptions can be masked based on their priority level or masked completly using interrupt masking registers BASEPRI, PRIMASK, FAULTMASK. They can be used to ensure that time-critical tasks can be finished on time without being interrupted.

Cortex-M3 Exception Types

Exception Number	Exception Type	Priority (Default to 0 if Programmable)	Description
0	NA	NA	No exception running
1	Reset	−3 (Highest)	Reset
2	NMI	−2	Nonmaskable interrupt (external NMI input)
3	Hard fault	−1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus fault	Programmable	Bus error (Prefetch Abort or Data Abort)
6	Usage fault	Programmable	Exceptions due to program error
7–10	Reserved	NA	Reserved
11	SVCall	Programmable	System service call
12	Debug monitor	Programmable	Debug monitor (break points, watchpoints, or external debug request)
13	Reserved	NA	Reserved
14	PendSV	Programmable	Pendable request for system device
15	SYSTICK	Programmable	System tick timer
16	IRQ #0	Programmable	External interrupt #0
17	IRQ #1	Programmable	External interrupt #1
...	...	...	...
255	IRQ #239	Programmable	External interrupt #239

ARM CortexM3 – Programmers view

Introduction to ARM CortexM3 – Memory Map

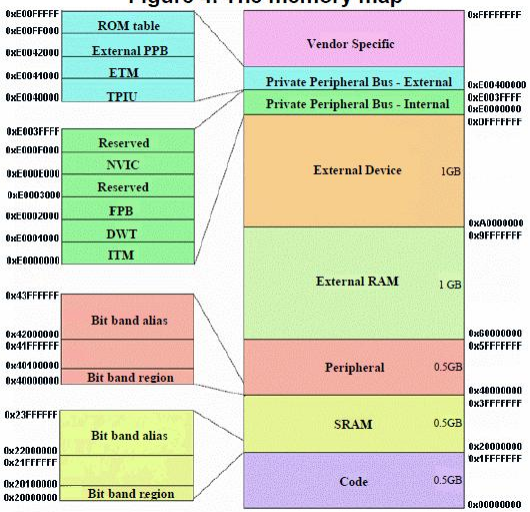
1. M3 has a predefined memory map. This allows the built-in peripherals, such as NVIC, and debug components to be accessed by simple memory access instructions.

2. This allows most system features through normal C program code.

3. Allows optimization for ease of integration and re-use

4. M3 has an optional Memory Protection Unit (MPU). The MPU is setup by an OS, allowing data used by privileged code to be protected from User programs. The MPU can be used to make memory regions ReadOnly to prevent accidental erasing of data, or to isolate memory regions between different tasks in a multi-tasking system. Overall, helps in making systems more robost and reliable.

Figure 4. The memory map

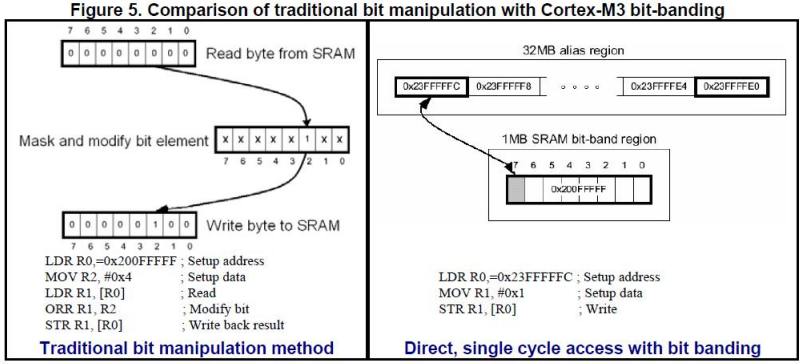


ARM CortexM3 – Programmers view

Introduction to ARM CortexM3 – Memory Map – Bit Banding

The Cortex-M3 processor enables direct access to single bits of data in simple systems by implementing a technique called bit-banding (Figure 5). The memory map includes two 1MB bit-band regions in the SRAM and peripheral space that map on to 32MB of alias regions. Load/store operations on an address in the alias region directly get translated to an operation on the bit aliased by that address. Writing to an address in the alias region with the least-significant bit set writes a 1 to the bit-band bit and writing with the least-significant bit cleared writes a 0 to the bit. Reading the aliased address directly returns the value in the appropriate bit-band bit. Additionally, this operation is atomic and cannot be interrupted by other bus activities.

Figure 5. Comparison of traditional bit manipulation with Cortex-M3 bit-banding



ARM CortexM3 – Programmers view

Introduction to ARM CortexM3 – Instruction Set

1. Basic syntax used is :: `opcode operand1, operand2, ...  ; Comments`
2. Bringup the TRM of M3 or GuideToArmCortexM3 (pg 71). Following categories :-
3. 16-bit Data processing instructions
4. 16-bit Branch instructions
5. 16-bit Load and Store instructions
6. Other 16-bit instructions
7. Same as above with 32-bit instructions

ARM CortexM3 – Programmers view and Development Environment

Acknowledgements

1. ARM Assembler Reference :: www.arm.com
2. The Definitive Guide To ARM Cortex-M3 :: Joseph Yiu

ARM CortexM3 – Programmers view and Development Environment

Hardware Software CoDesign

November 2011, December 2011

Agenda

ARM CortexM3 – Programmers view and Development Environment

1. Continued... Programmers view of CortexM3 (refer as M3)
2. Discussion on the Answering Machine Assignment (Group wise completion)
3. Mid Semester Paper distribution (left-overs and any doubts)
4. Introduction to Image Processing – Pixelization, Quantization

ARM CortexM3 – Programmers view

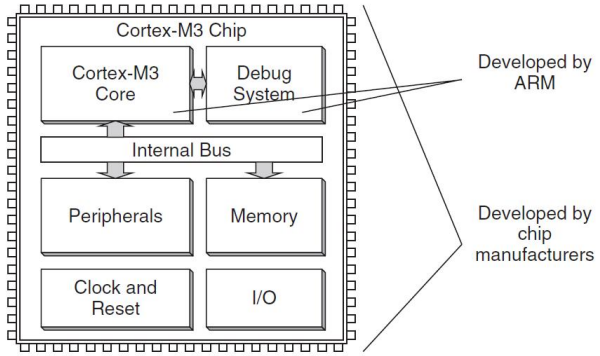
Introduction to ARM CortexM3

1. Primarily designed to target the 32bit Microcontoller market
2. Great performance at low cost and many new features available only in high-end processors
3. Enhanced determinism, guaranteeing that critical tasks and interrupts are serviced as quickly as possible, but in a "known" number of cycles.
4. Improve code density, ensuring that code fits even the smallest memory footprints
5. Ease of use, providing debugability and easy programmability for those applications which are migrating from 8, 16bit to 32bit.
6. Can be used in "device aggregation", where multiple traditional 8bit devices can get replaced by a single 32bit high performance device.
7. Through the compilers, the amount of code reuse across other ARM systems can take place.

ARM CortexM3 – Programmers view

Introduction to ARM CortexM3

1. The use model for ARM and other integrators is :-

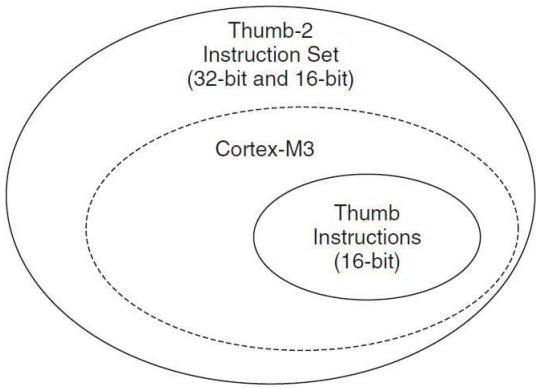


2. In general, ARM has come up with Cortex family like :-
   - **A family** :: designed for high-performance *application* platforms
   - **R family** :: designed for high-end embedded systems in which *Real-Time* performance is needed
   - **M family** :: designed for deeply embedded *Microcontroller* systems

ARM CortexM3 – Programmers view

Introduction to ARM CortexM3 – ARM and Thumb Instruction Sets

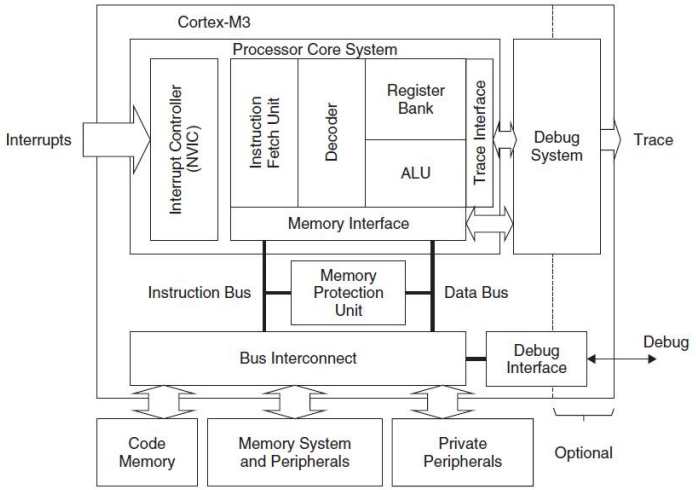
1. There are two different types of instruction sets
   (a) 32bit called *ARM* instruction set
   (b) 16bit called *Thumb* instruction set
2. During program execution, the processor can be dynamically switched between the ARM state or Thumb state to use either of the instruction sets
3. The Thumb instruction set provides only a subset of the ARM instructions, but can provide high code density.
4. M3 processor supports only the Thumb-2 (and traditional Thumb) instruction set. It uses Thumb-2 instruction set for all operations



The Relationship Between the Thumb-2 Instruction Set and the Thumb Instruction Set

ARM CortexM3 – Programmers view

Introduction to ARM CortexM3 – Basic Block Diagram



A Simplified View of the Cortex-M3

1. The processor has a Harvard architecture, ie., has a separate instruction bus and data bus. However, the instruction and data buses share the same memory space (unified memory system). i.e, Cannot get 8GB space just because there are separate bus interfaces.

2. *GROUP DISCUSSION* :: How does Harvard architecture help M3 ?

ARM CortexM3 – Programmers view

Introduction to ARM CortexM3 – General Purpose Register Set

1. The M3 has GP registers *R0 to R15*

2. *General Purpose Registers* :: R0 to R12

3. *Stack Pointer* :: R13 → banked R13 register
   (a) *Main Stack Pointer MSP* - Default StackPointer, used by the OS kernel and exception handlers
   (b) *Process Stack Pointer PSP* - Used by the user application code

4. *Link Register* :: R14 → When a subroutine is called, the return address is stored in the link register.

5. *Program Counter* :: R15 → The current program address. This register can be written to control the program flow

ARM CortexM3 – Programmers view

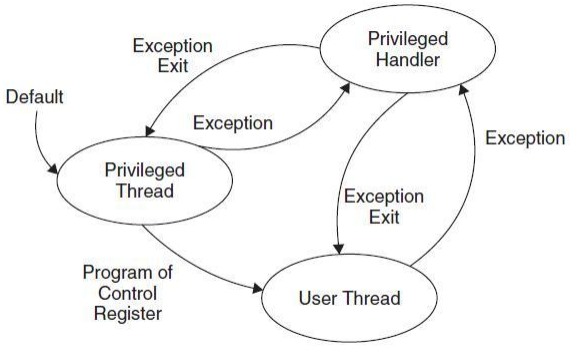
Introduction to ARM CortexM3 – Operation Modes

1. The M3 has 2 Operation modes and 2 privilege levels

2. *Operation Modes* :: What kind of operation the M3 is doing. Whether the processor is running a normal program or running an exception handler like an interrupt handler or system exception handler.
   (a) *Thread Mode* :: Thread mode is entered on reset, and can be entered as a result of an exception return. Privileged and User code can run in Thread mode.
   (b) *Handler Mode* :: Handler mode is entered as a result of an exception. All code is privileged in Handler mode.

3. *Privilege Levels* :: provide a mechanizm for safeguarding memory acces to critical regions as well as provide a basic security model.
   (a) *Privilege Level* ::
   (b) *User Level* ::

	Privileged	*User*
When running an exception	Handle Mode	
When running main program	Thread Mode	Thread Mode



Allowed Operation Mode Transitions

PRIMASK, FAULTMASK. They can be used to ensure that time-critical tasks can be finished on time without being interrupted.

ARM CortexM3 – Programmers view

Introduction to ARM CortexM3 – BuiltIn Nested Vectored Interrupt Controller

1. The M3 Processor includes an Interrupt Controller called the *NVIC (Nested Vectored Interrupt Controller* with following main features :-

   (a) *Nested Interrupt Support* :: All interrupts and most of the system exceptions can be programmed to different priority levels. When an interrupt occurs, the NVIC compares the priority of this interrupt to the current running priority level. If the priority of the new interrupt is higher than the current level, the interrupt handler of the new interrupt will override the current running task.

   (b) *Vectored Interrupt Support* :: When an interrupt is accepted, the starting address of the ISR is located from a vector table in memory. There is no need to use software to determine and branch to the starting address of the ISR. Thus it takes less time to process the interrupt request.

   (c) *Dynamic Priority changes Support* :: Priority levels of interrupts can be changed by software during run time. Interrupts that are being serviced are blocked from further activation until the ISR is completed, so their priority can be changed without risk of accidental re-entry.

   (d) *Reduction of Interrupt Latency* :: M3 includes automatic saving and restoring some register contents, reducing delay in switching from one ISR to another and handling late arrival interrupts.

   (e) *Interrupt Masking* :: Interrupts and system exceptions can be masked based on their priority level or masked completly using interrupt masking registers BASEPRI,

Cortex-M3 Exception Types

Exception Number	Exception Type	Priority (Default to 0 if Programmable)	Description
0	NA	NA	No exception running
1	Reset	−3 (Highest)	Reset
2	NMI	−2	Nonmaskable interrupt (external NMI input)
3	Hard fault	−1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus fault	Programmable	Bus error (Prefetch Abort or Data Abort)
6	Usage fault	Programmable	Exceptions due to program error
7–10	Reserved	NA	Reserved
11	SVCall	Programmable	System service call
12	Debug monitor	Programmable	Debug monitor (break points, watchpoints, or external debug request)
13	Reserved	NA	Reserved
14	PendSV	Programmable	Pendable request for system device
15	SYSTICK	Programmable	System tick timer
16	IRQ #0	Programmable	External interrupt #0
17	IRQ #1	Programmable	External interrupt #1
...	...	...	...
255	IRQ #239	Programmable	External interrupt #239

ARM CortexM3 – Programmers view

Introduction to ARM CortexM3 – Memory Map

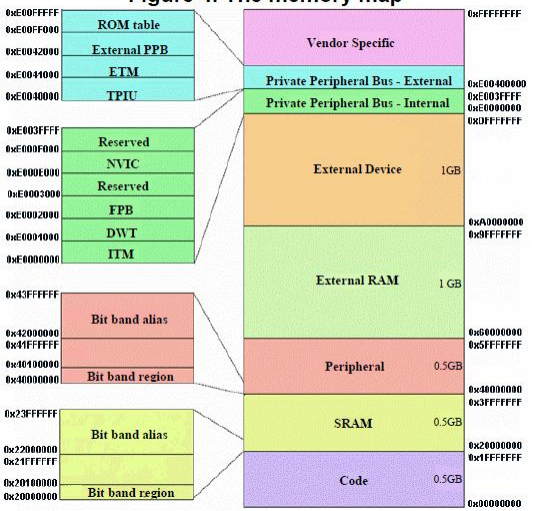
1. M3 has a predefined memory map. This allows the built-in peripherals, such as NVIC, and debug components to be accessed by simple memory access instructions.

2. This allows most system features through normal C program code.

3. Allows optimization for ease of integration and re-use

4. M3 has an optional Memory Protection Unit (MPU). The MPU is setup by an OS, allowing data used by privileged code to be protected from User programs. The MPU can be used to make memory regions ReadOnly to prevent accidental erasing of data, or to isolate memory regions between different tasks in a multi-tasking system. Overall, helps in making systems more robost and reliable.

Figure 4. The memory map

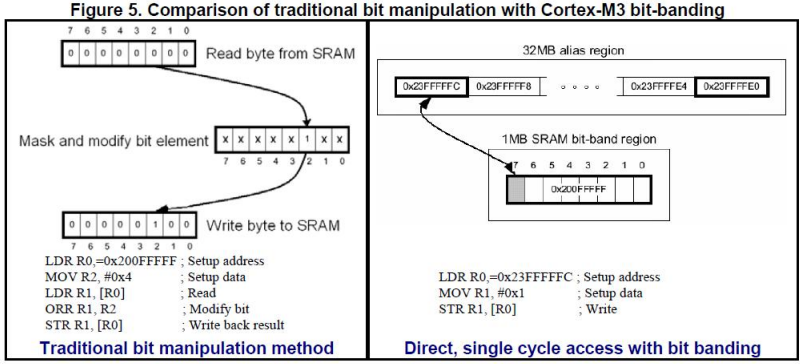


ARM CortexM3 – Programmers view

Introduction to ARM CortexM3 – Memory Map – Bit Banding

The Cortex-M3 processor enables direct access to single bits of data in simple systems by implementing a technique called bit-banding (Figure 5). The memory map includes two 1MB bit-band regions in the SRAM and peripheral space that map on to 32MB of alias regions. Load/store operations on an address in the alias region directly get translated to an operation on the bit aliased by that address. Writing to an address in the alias region with the least-significant bit set writes a 1 to the bit-band bit and writing with the least-significant bit cleared writes a 0 to the bit. Reading the aliased address directly returns the value in the appropriate bit-band bit. Additionally, this operation is atomic and cannot be interrupted by other bus activities.

Figure 5. Comparison of traditional bit manipulation with Cortex-M3 bit-banding



ARM CortexM3 – Programmers view

Introduction to ARM CortexM3 – Instruction Set

1. Basic syntax used is :: `opcode operand1, operand2, ... ; Comments`

2. Bringup the TRM of M3 or GuideToArmCortexM3 (pg 71). Following categories :-

3. 16-bit Data processing instructions

4. 16-bit Branch instructions

5. 16-bit Load and Store instructions

6. Other 16-bit instructions

7. Same as above with 32-bit instructions

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q	Application PSR										
IPSR	Interrupt PSR											Exception Number				
EPSR	Execution PSR					ICI/IT	T			ICI/IT						

Program Status Registers (PSRs) in the Cortex-M3

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	ICI/IT	T			ICI/IT		Exception Number				

Combined Program Status Registers (xPSR) in the Cortex-M3

Bit	Description
N	Negative
Z	Zero
C	Carry/borrow
V	Overflow
Q	Sticky saturation flag
ICI/IT	Interrupt-Continuable Instruction (ICI) bits, IF-THEN instruction status bit
T	Thumb state, always 1; trying to clear this bit will cause a fault exception
Exception Number	Indicates which exception the processor is handling

Bit Fields in Cortex-M3 Program Status Registers

- Z (Zero) flag: This flag is set when the result of an instruction has a zero value or when a comparison of two data returns an equal result.
- N (Negative) flag: This flag is set when the result of an instruction has a negative value (bit 31 is 1).
- C (Carry) flag: This flag is for unsigned data processing—for example, in add (ADD) it is set when an overflow occurs; in subtract (SUB) it is set when a borrow did not occur (borrow is the invert of carry).
- V (Overflow) flag: This flag is for signed data processing; for example, in an add (ADD), when two positive values added together produce a negative value, or when two negative values added together produce a positive value.

16-Bit Data Processing Instructions

Instruction	Function
ADC	Add with carry
ADD	Add
AND	Logical AND
ASR	Arithmetic shift right
BIC	Bit clear (Logical AND one value with the logic inversion of another value)
CMN	Compare negative (compare one data with two's complement of another data and update flags)
CMP	Compare (compare two data and update flags)
CPY	Copy (available from architecture v6; move a value from one high or low register to another high or low register)
EOR	Exclusive OR
LSL	Logical shift left
LSR	Logical shift right
MOV	Move (can be used for register-to-register transfers or loading immediate data)
MUL	Multiply
MVN	Move NOT (obtain logical inverted value)
NEG	Negate (obtain two's complement value)
ORR	Logical OR
ROR	Rotate right
SBC	Subtract with carry
SUB	Subtract
TST	Test (use as logical AND; Z flag is updated but AND result is not stored)
REV	Reverse the byte order in a 32-bit register (available from architecture v6)
REVH	Reverse the byte order in each 16-bit half word of a 32-bit register (available from architecture v6)
REVSH	Reverse the byte order in the lower 16-bit half word of a 32-bit register and sign extends the result to 32 bits. (available from architecture v6)
SXTB	Signed extend byte (available from architecture v6)
SXTH	Signed extend half word (available from architecture v6)

16-Bit Branch Instructions

Instruction	Function
B	Branch
B<cond>	Conditional branch
BL	Branch with link; call a subroutine and store the return address in LR
BLX	Branch with link and change state (BLX <reg> only)[1]
CBZ	Compare and branch if zero (architecture v7)
CBNZ	Compare and branch if nonzero (architecture v7)
IT	IF-THEN (architecture v7)

16-Bit Load and Store Instructions

Instruction	Function
LDR	Load word from memory to register
LDRH	Load half word from memory to register
LDRB	Load byte from memory to register
LDRSH	Load half word from memory, sign extend it, and put it in register
LDRSB	Load byte from memory, sign extend it, and put it in register
STR	Store word from register to memory
STRH	Store half word from register to memory
STRB	Store byte from register to memory
LDMIA	Load multiple increment after
STMIA	Store multiple increment after
PUSH	Push multiple registers
POP	Pop multiple registers

Other 16-Bit Instructions

Instruction	Function
SVC	System service call
BKPT	Breakpoint; if debug is enabled, will enter debug mode (halted), or if debug monitor exception is enabled, will invoke the debug exception; otherwise it will invoke a fault exception
NOP	No operation
CPSIE	Enable PRIMASK (CPSIE i)/FAULTMASK (CPSIE f) register (set the register to 0)
CPSID	Disable PRIMASK (CPSID i)/ FAULTMASK (CPSID f) register (set the register to 1)

32-Bit Data Processing Instructions

Instruction	Function
ADC	Add with carry
ADD	Add
ADDW	Add wide (#immed_12)
AND	Logical AND
ASR	Arithmetic shift right
BIC	Bit clear (logical AND one value with the logic inversion of another value)
BFC	Bit field clear
BFI	Bit field insert

(Continued)

Instruction	Function
CMN	Compare negative (compare one data with two's complement of another data and update flags)
CMP	Compare (compare two data and update flags)
CLZ	Count lead zero
EOR	Exclusive OR
LSL	Logical shift left
LSR	Logical shift right
MLA	Multiply accumulate
MLS	Multiply and subtract
MOV	Move
MOVW	Move wide (write a 16-bit immediate value to register)
MOVT	Move top (write an immediate value to the top half word of destination reg)
MVN	Move negative
MUL	Multiply
ORR	Logical OR
ORN	Logical OR NOT
RBIT	Reverse bit
REV	Byte reserve word
REVH/REV16	Byte reverse packed half word
REVSH	Byte reverse signed half word
ROR	Rotate right register
RSB	Reverse subtract
RRX	Rotate right extended
SBFX	Signed bit field extract
SDIV	Signed divide
SMLAL	Signed multiply accumulate long
SMULL	Signed multiply long
SSAT	Signed saturate
SBC	Subtract with carry
SUB	Subtract
SUBW	Subtract wide (#immed_12)
SXTB	Sign extend byte
TEQ	Test equivalent (use as logical exclusive OR; flags are updated but result is not

Instruction	Function
TST	Test (use as logical AND; Z flag is updated but AND result is not stored)
UBFX	Unsigned bit field extract
UDIV	Unsigned divide
UMLAL	Unsigned multiply accumulate long
UMULL	Unsigned multiply long
USAT	Unsigned saturate
UXTB	Unsigned extend byte
UXTH	Unsigned extend half word

32-Bit Load and Store Instructions

Instruction	Function
LDR	Load word data from memory to register
LDRB	Load byte data from memory to register
LDRH	Load half word data from memory to register
LDRSB	Load byte data from memory, sign extend it, and put it to register
LDRSH	Load half word data from memory, sign extend it, and put it to register
LDM	Load multiple data from memory to registers
LDRD	Load double word data from memory to registers
STR	Store word to memory
STRB	Store byte data to memory
STRH	Store half word data to memory
STM	Store multiple words from registers to memory
STRD	Store double word data from registers to memory
PUSH	Push multiple registers
POP	Pop multiple registers

32-Bit Branch Instructions

Instruction	Function
B	Branch
BL	Branch and link
TBB	Table branch byte; forward branch using a table of single byte offset
TBH	Table branch half word; forward branch using a table of half word offset

Other 32-Bit Instructions

Instruction	Function
LDREX	Exclusive load word
LDREXH	Exclusive load half word
LDREXB	Exclusive load byte
STREX	Exclusive store word
STREXH	Exclusive store half word
STREXB	Exclusive store byte
CLREX	Clear the local exclusive access record of local processor
MRS	Move special register to general-purpose register
MSR	Move to special register from general-purpose register
NOP	No operation
SEV	Send event
WFE	Sleep and wake for event
WFI	Sleep and wake for interrupt
ISB	Instruction synchronization barrier
DSB	Data synchronization barrier
DMB	Data memory barrier

Examples of Arithmetic Instructions

Instruction	Operation

`ADD Rd, Rn, Rm ; Rd = Rn + Rm`
`ADD Rd, Rm ; Rd = Rd + Rm`
`ADD Rd, #immed ; Rd = Rd + #immed`
	ADD operation

`ADC Rd, Rn, Rm ; Rd = Rn + Rm + carry`
`ADC Rd, Rm ; Rd = Rd + Rm + carry`
`ADC Rd, #immed ; Rd = Rd + #immed + ; carry`
	ADD with carry

`ADDW Rd, Rn,#immed ; Rd = Rn + #immed`	ADD register with 12-bit immediate value

`SUB Rd, Rn, Rm ; Rd = Rn - Rm`
`SUB Rd, #immed ; Rd = Rd - #immed`
`SUB Rd, Rn,#immed ; Rd = Rn -#immed`
	SUBTRACT

`SBC Rd, Rm ; Rd = Rd - Rm - ; carry flag`
`SBC.W Rd, Rn, #immed ; Rd = Rn - #immed - ; carry flag`
`SBC.W Rd, Rn, Rm ; Rd = Rn - Rm - ; carry flag`
	SUBTRACT with borrow (carry)

`RSB.W Rd, Rn, #immed ; Rd = #immed -Rn` `RSB.W Rd, Rn, Rm ; Rd = Rm - Rn`	Reverse subtract
`MUL Rd, Rm ; Rd = Rd * Rm` `MUL.W Rd, Rn, Rm ; Rd = Rn * Rm`	Multiply
`UDIV Rd, Rn, Rm ; Rd = Rn /Rm` `SDIV Rd, Rn, Rm ; Rd = Rn /Rm`	Unsigned and signed divide

Logic Operation Instructions

Instruction	Operation

`AND Rd, Rn ; Rd = Rd & Rn`
`AND.W Rd, Rn,#immed; Rd = Rn & #immed`
`AND.W Rd, Rn, Rm ; Rd = Rn & Rd`
	Bitwise AND

`ORR Rd, Rn ; Rd = Rd | Rn`
`ORR.W Rd, Rn,#immed; Rd = Rn | #immed`
`ORR.W Rd, Rn, Rm ; Rd = Rn | Rd`
	Bitwise OR

`BIC Rd, Rn ; Rd = Rd & (~Rn)`
`BIC.W Rd, Rn,#immed; Rd = Rn &(~#immed)`
`BIC.W Rd, Rn, Rm ; Rd = Rn &(~Rd)`
	Bit clear

`ORN.W Rd, Rn,#immed; Rd = Rn |(~#immed)` `ORN.W Rd, Rn, Rm ; Rd = Rn |(~Rd)`	Bitwise OR NOT

`EOR Rd, Rn ; Rd = Rd ^ Rn`
`EOR.W Rd, Rn,#immed; Rd = Rn | #immed`
`EOR.W Rd, Rn, Rm ; Rd = Rn | Rd`
	Bitwise Exclusive OR

Shift and Rotate Instructions

Instruction	Operation

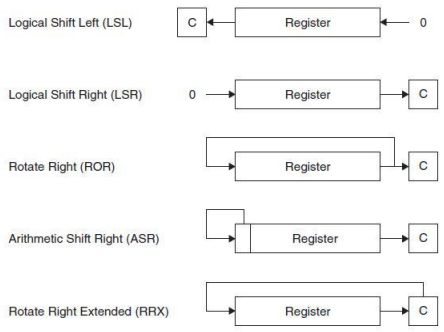
`ASR Rd, Rn,#immed; Rd = Rn >> immed`
`ASR Rd, Rn ; Rd = Rd >> Rn`
`ASR.W Rd, Rn, Rm ; Rd = Rn >> Rm`
	Arithmetic shift right

Instruction	Operation

`LSL Rd, Rn,#immed; Rd = Rn << immed`
`LSL Rd, Rn ; Rd = Rd << Rn`
`LSL.W Rd, Rn, Rm ; Rd = Rn << Rm`
	Logical shift left

`LSR Rd, Rn,#immed; Rd = Rn >> immed`
`LSR Rd, Rn ; Rd = Rd >> Rn`
`LSR.W Rd, Rn, Rm ; Rd = Rn >> Rm`
	Logical shift right

`ROR Rd, Rn ; Rd rot by Rn` `ROR.W Rd, Rn, Rm ; Rd = Rn rot by Rm`	Rotate right
`RRX.W Rd, Rn ; {C, Rd} = {Rn, C}`	Rotate right extended

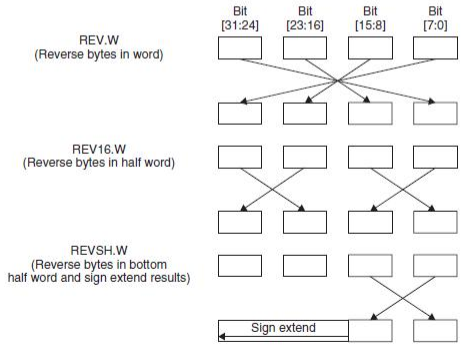


Sign Extend Instructions

Instruction	Operation
`SXTB.W Rd, Rm ; Rd = signext(Rn[7:0])`	Sign extend byte data into word
`SXTH.W Rd, Rm ; Rd = signext(Rn[15:0])`	Sign extend half word data into word

Data Reverse Ordering Instructions

Instruction	Operation
`REV.W Rd, Rn ; Rd = rev(Rn)`	Reverse bytes in word
`REV16.W <Rd>, <Rn> ; Rd = rev16(Rn)`	Reverse bytes in each half word
`REVSH.W <Rd>, <Rn> ; Rd = revsh(Rn)`	Reverse bytes in bottom half word and sign extend the result



ARM CortexM3 – Programmers view

Bit Field Processing and Manipulation Instructions

Instruction	Operation
`BFC.W Rd, Rn, #<width>`	Clear bit field within a register
`BFI.W Rd, Rn, #<lsb>, #<width>`	Insert bit field to a register
`CLZ.W Rd, Rn`	Count leading zero
`RBIT.W Rd, Rn`	Reverse bit order in register
`SBFX.W Rd, Rn, #<lsb>, #<width>`	Copy bit field from source and sign extend it
`UBFX.W Rd, Rn, #<lsb>, #<width>`	Copy bit field from source register

ARM CortexM3 – C to Assembly examples

```
i = 5;
while (i != 0 ){
func1(); ; call a function
i--;
}
```

This can be compiled into:

```
      MOV  R0, #5          ; Set loop counter
loop1 CBZ  R0, loop1exit   ; if loop counter = 0 then exit the lo
      BL   func1           ; call a function
      SUB  R0, #1          ; loop counter decrement
      B    loop1           ; next loop
loop1exit
```


ARM CortexM3 – C to Assembly examples

```
if (R1<R2) then
   R2=R2-R1
   R2=R2/2
else
   R1=R1-R2
   R1=R1/2
```

In assembly:

```
   CMP      R1, R2          ; If R1 < R2 (less then)
   ITTEE    LT              ; then execute instruction 1 and 2
                            ; (indicated by T)
                            ; else execute instruction 3 and 4
                            ; (indicated by E)
   SUBLT.W  R2,R1           ; 1st instruction
   LSRLT.W  R2,#1           ; 2nd instruction
   SUBGE.W  R1,R2           ; 3rd instruction (notice the GE is
                            ; opposite of LT)
   LSRGE.W  R1,#1           ; 4th instruction
```


Logical Break

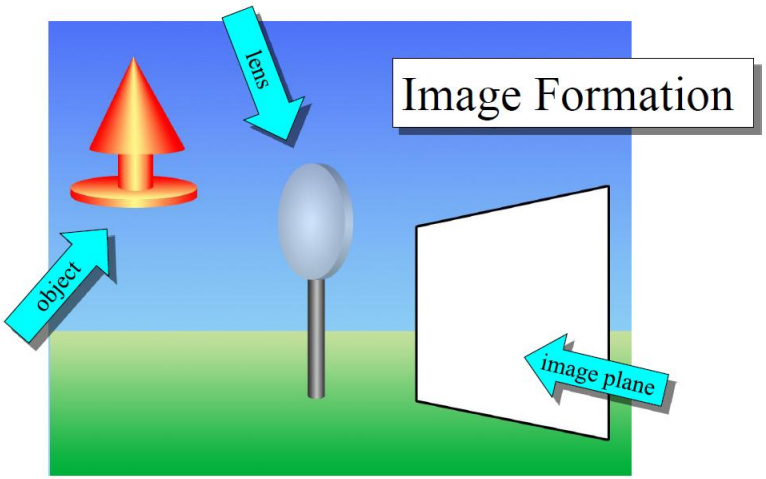


Image Formation
lens
object
image plane

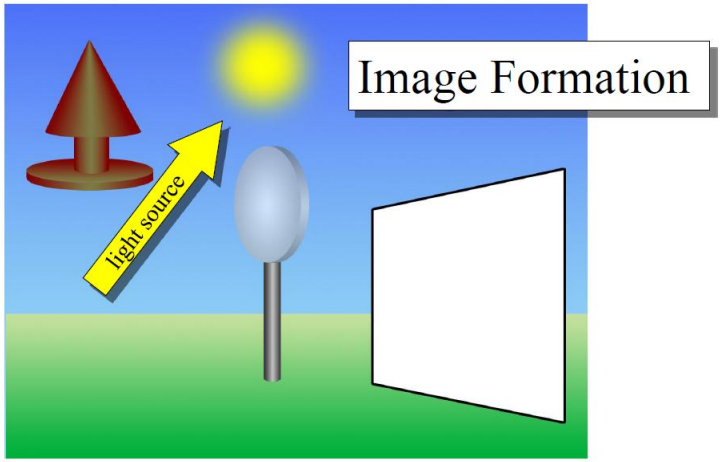


Image Formation
light source

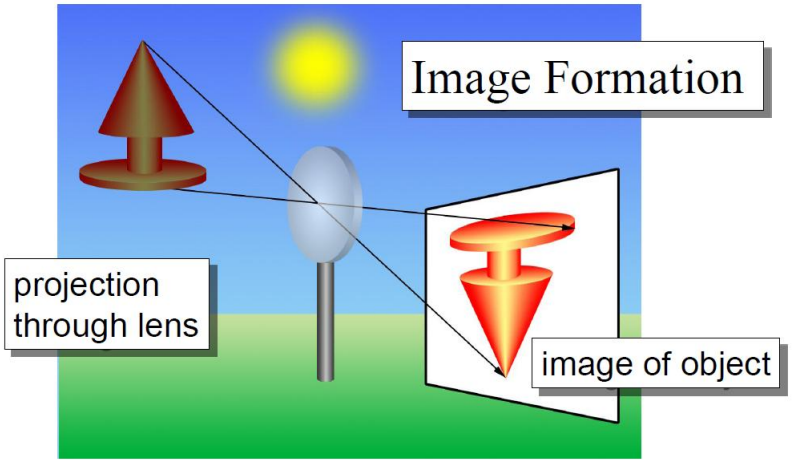


Image Formation
projection through lens
image of object

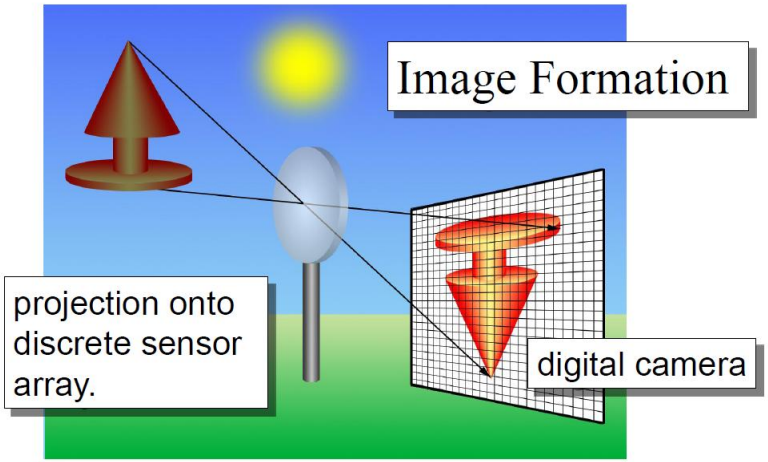
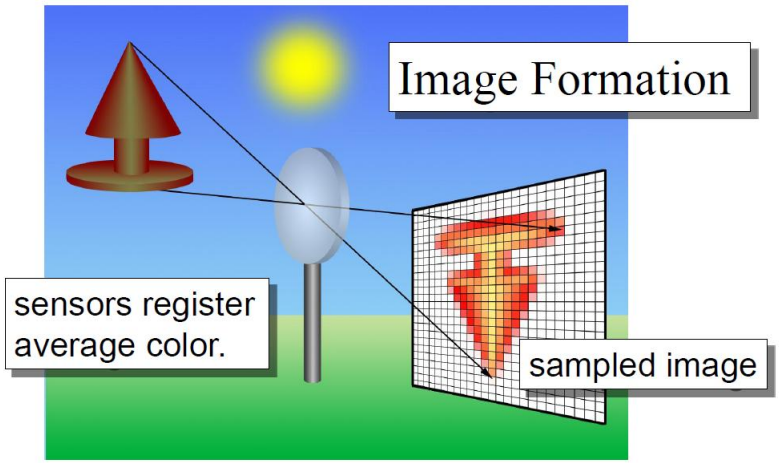
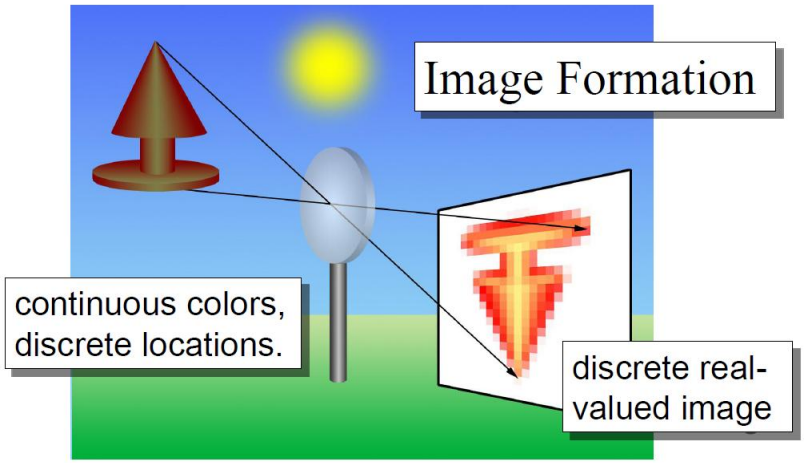
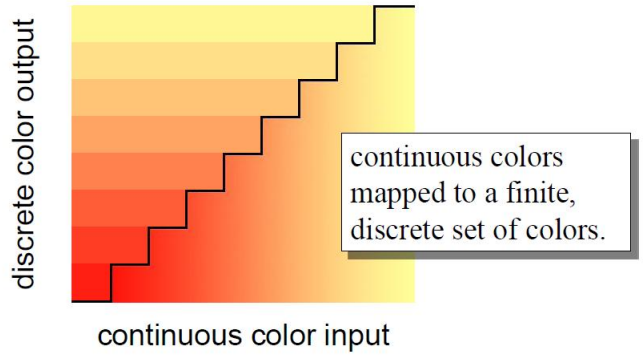


Image Formation
projection onto discrete sensor array.
digital camera

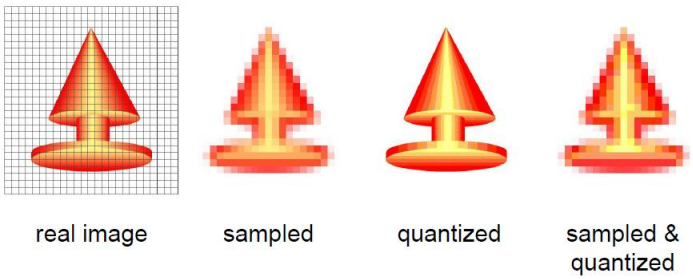




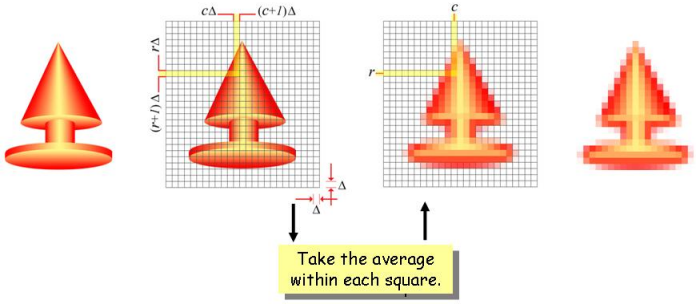
Digital Image Formation: Quantization



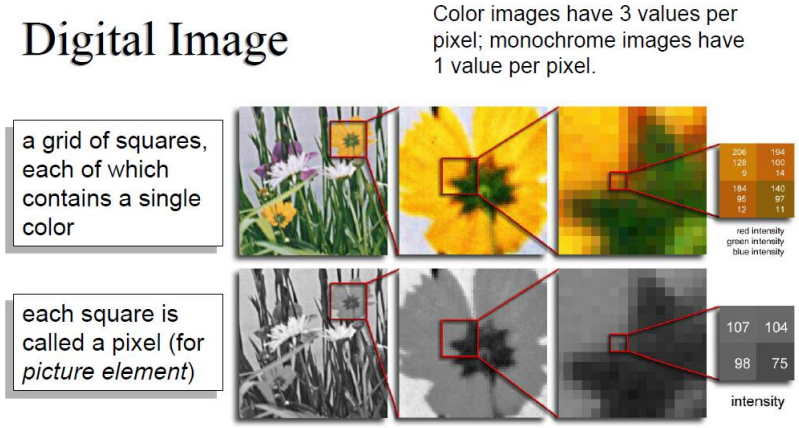
Sampling and Quantization



Pixelization :-



Pixelization :-



ARM CortexM3 – Programmers view and Development Environment

Introduction to Image Processing

Acknowledgements

1. ARM Assembler Reference :: www.arm.com
2. The Definitive Guide To ARM Cortex-M3 :: Joseph Yiu
3. http://www.archive.org/details/Lectures_on_Image_Processing :: Richard Alan Peters II :: Dept of Electrical Engg & CS :: Vanderbilt University School of Engineering.
   (a) EECE253_01_Intro.pdf
   (b) EECE253_03_PointProcessing.pdf
   (c) EECE253_10_PixelizationQuantization.pdf
4. YCbCr to RGB Considerations :: YCbCr_Intersil_AppNote.pdf
5. Embedded System Design - A Unified Hardware / Software Introduction :: Ch 7

Image Processing

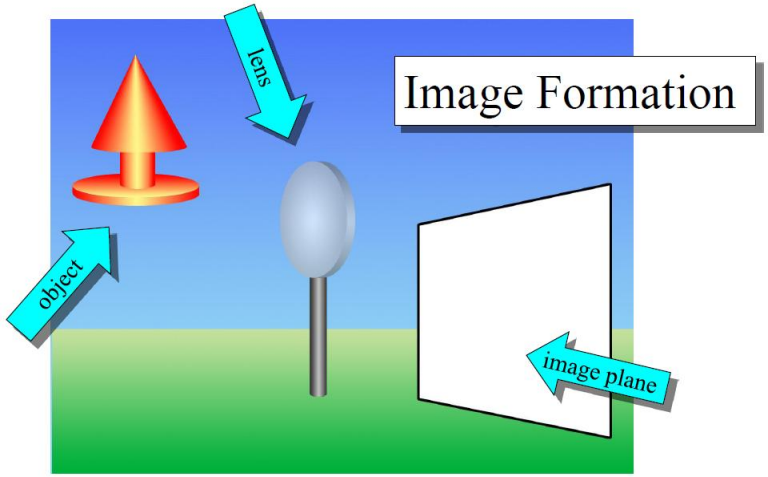
Hardware Software CoDesign

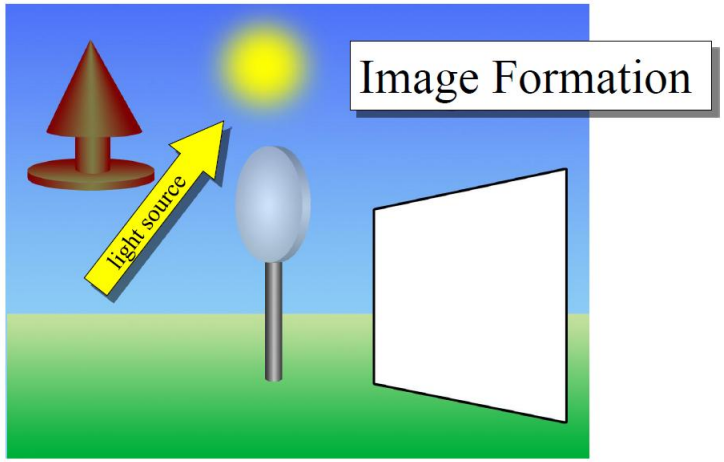
December 2011

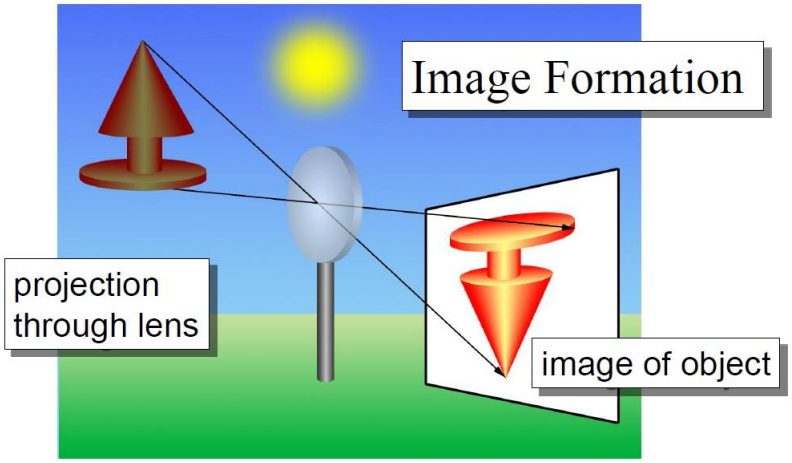
Agenda

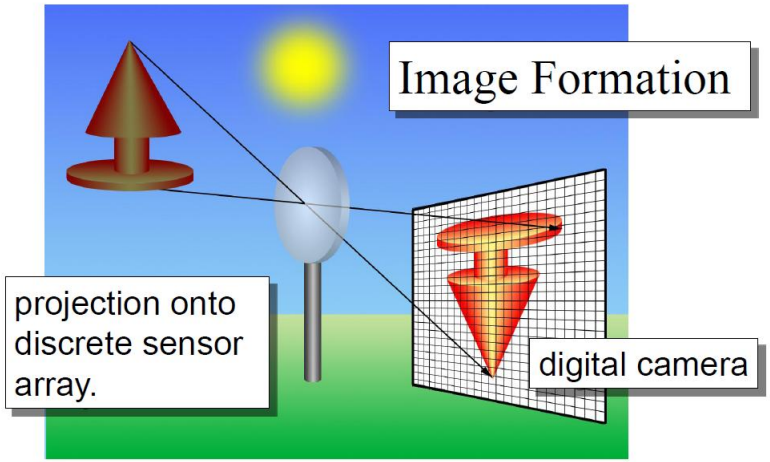
Image Processing

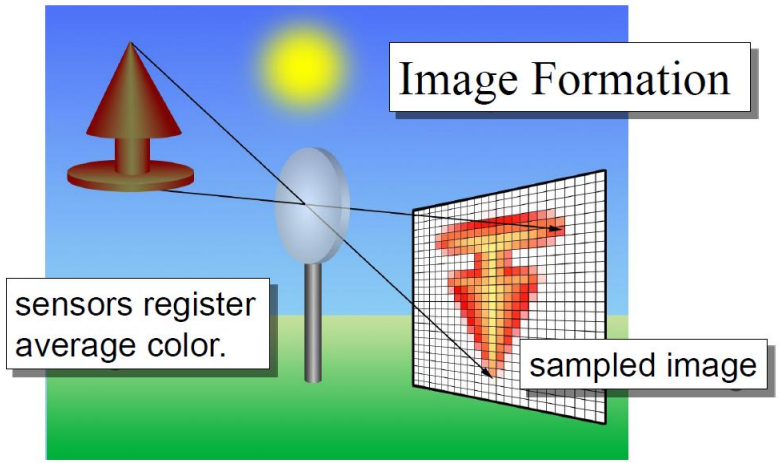
1. Introduction to Image Processing – Pixelization, Quantization (repeat)
2. The Discrete Cosine Transform
3. Why Compress or Encode Images ?
4. The Steps in Image Compression
5. Digital Camera Example
6. Discussion on the Answering Machine Assignment (Group wise completion)
7. Mid Semester Paper distribution (left-overs and any doubts)

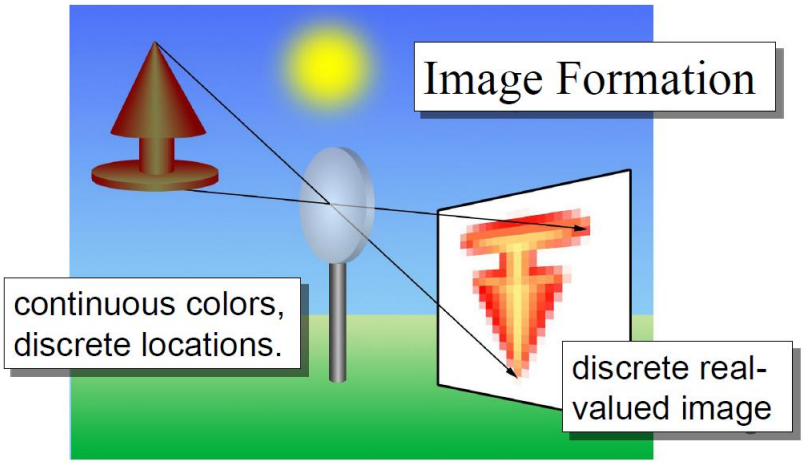




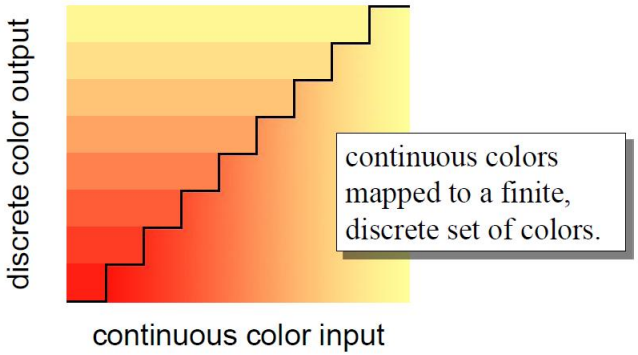




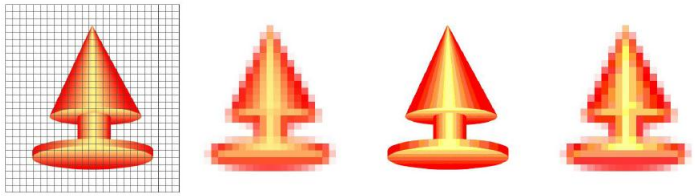




Digital Image Formation: Quantization

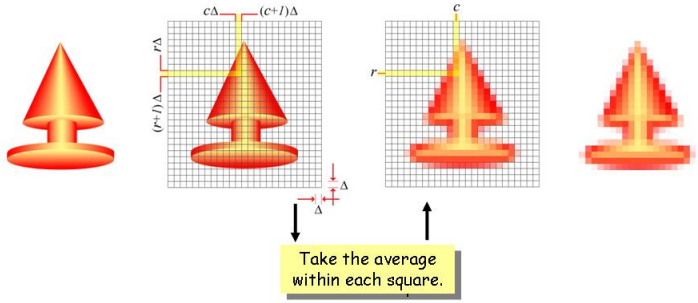


Sampling and Quantization



real image | sampled | quantized | sampled & quantized

Pixelization :-



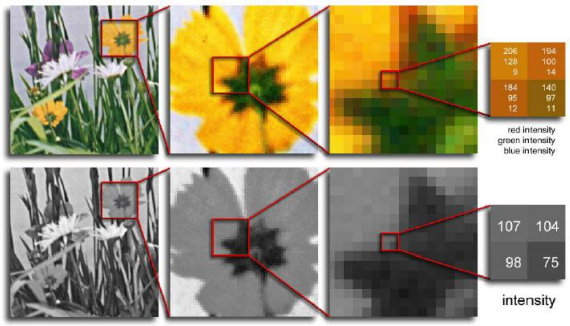
Pixelization :-

Digital Image

Color images have 3 values per pixel; monochrome images have 1 value per pixel.

a grid of squares, each of which contains a single color

each square is called a pixel (for *picture element*)



Why Compress / Encode Images

Original Image :: 352 x 288
24bits per pixel.

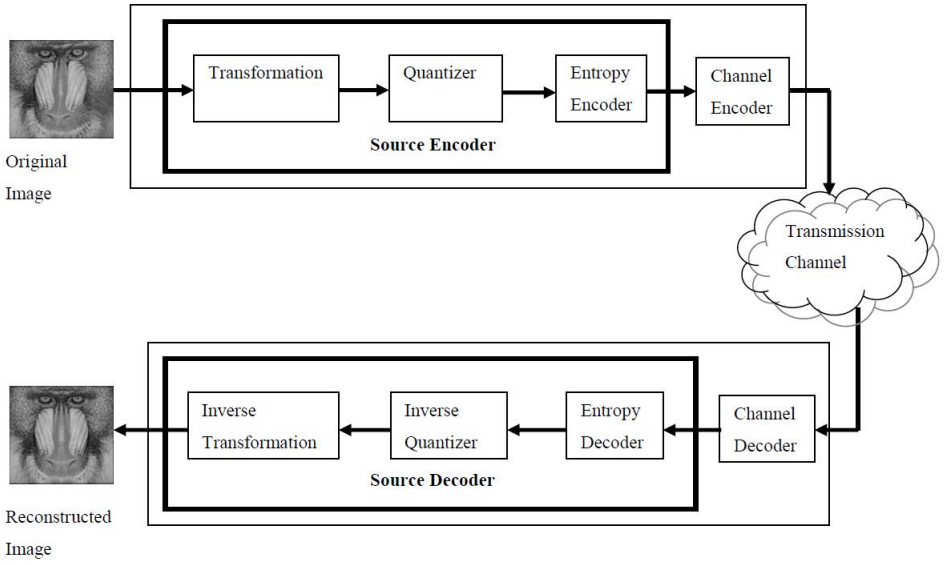
JPG FILE :: LT 20KB

Why Compress / Encode Images

Video Source	Output Data Rate[Kbits/sec]
Quarter VGA @20 frames/sec	36 864.00
CIF camera @30 frames/sec	72 990.72
VGA @30 frames/sec	221 184.00

Transmission Medium	Data Rate [Kbits/sec]
Wireline modem	56
GPRS (estimated average rate)	30
3G/WCDMA (theoretical maximum)	384

Steps in Image Compression



Components of a typical image/video transmission system

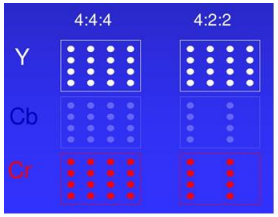
Steps in Image Compression

1. If the color is represented by RGB mode, translate it to YCrCb.
2. Divide the file into 8 x 8 blocks.
3. Transform the pixel information from the spatial domain to the frequency domain with the Discrete Cosine Transform.
4. Quantize the resulting values by dividing each coefficient by an integer value and rounding off to the nearest integer.
5. Look at the resulting coefficients in a zig-zag order. Do a run-length encoding of the coefficients ordered in this manner.
6. Follow by Huffman coding.

Steps in Image Compression

Converting RGB to YCbCr

1. YCbCr color mode stores color in terms of its luminance (brightness) and chrominance (hue)
2. The human eye is less sensitive to chrominance than luminance.
3. ?? Compression then can be achieved by storing more luma details than chroma details. ?? Called chroma-sub-sampling.
4. Formats 4:4:4, 4:2:2
5. When Converted from RGB → 4:4:4 → 4:2:2 the bandwidth is reduced by 50% (achieving compression).



Steps in Image Compression

Converting RGB to YCbCr

The BT.601 equations are used by many video ICs to convert between digital R′G′B′ data and YCbCr are:

$$Y = (77/256)R' + (150/256)G' + (29/256)B'$$

$$Cb = -(44/256)R' - (87/256)G' + (131/256)B' + 128$$

$$Cr = (131/256)R' - (110/256)G' - (21/256)B' + 128$$

$$R' = Y + 1.371(Cr - 128)$$

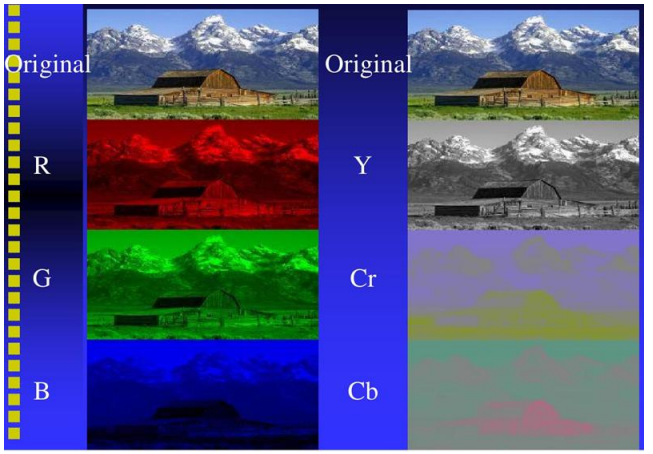
$$G' = Y - 0.698(Cr - 128) - 0.336(Cb - 128)$$

$$B' = Y + 1.732(Cb - 128)$$

1. From a Hardware - Software point of view :: This is a computation requiring bandwidth.
2. GROUP DISCUSSION :: How does one implement in HW.

Steps in Image Compression

Converting RGB to YCbCr



Steps in Image Compression

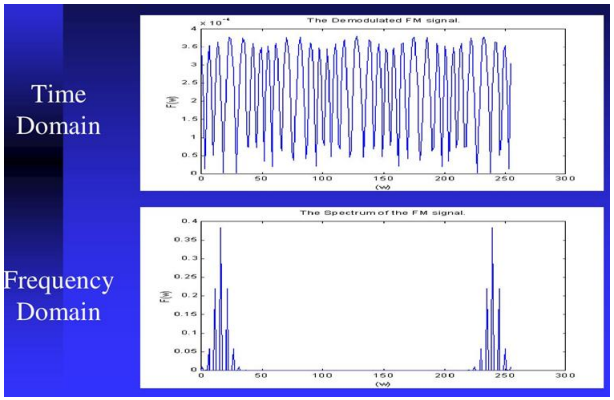
Divide into 8x8 blocks and Transform to Frequency Domain :: DCT

1. The DCT transforms the data from the spatial domain to the frequency domain.
2. The spatial domain shows the amplitude of the color as you move through space.
3. The frequency domain shows how quickly the amplitude of the color is changing from one pixel to the next in an image file.
4. For the 8 x 8 matrix of color data, we'll get an 8 x 8 matrix of coefficients for the frequency components.
5. GROUP DISCUSSION :: What is being done here actually?

Steps in Image Compression

Divide into 8x8 blocks and Transform to Frequency Domain :: DCT

1. GROUP DISCUSSION :: What is being done here actually?
2. "Use Time Domain –¿ Frequency Domain" transformation.
3. "Use the fact that human eye is imperfect"
4. "Achieve lossy compression, which satisfies the bandwidth / size consideration"



Steps in Image Compression

Divide into 8x8 blocks and Transform to Frequency Domain :: DCT

1. Discrete Cosine Transform (DCT) is a Fourier-related transform similar to the Discrete Fourier Transform (DFT), but using only real numbers.
2. The most common variant of the DCT is the type II DCT; its inverse is called IDCT (inverse DCT)
3. The frequency domain is a better representation for the data because it makes it possible to separate out → and throw away → information that isnt very important for human perception.
4. The human eye is not very sensitive to high frequency changes → especially in photographic images, so the high frequency data, to some extent, can be discarded

Steps in Image Compression

Divide into 8x8 blocks and Transform to Frequency Domain :: DCT

The most common DCT definition of a 1-D sequence of length N is

$$C(u)=\alpha(u)\sum_{x=0}^{N-1} f(x)\cos\left[\frac{\pi(2x+1)u}{2N}\right], \tag{1}$$

for $u=0,1,2,\ldots,N-1$. Similarly, the inverse transformation is defined as

$$f(x)=\sum_{u=0}^{N-1}\alpha(u)C(u)\cos\left[\frac{\pi(2x+1)u}{2N}\right], \tag{2}$$

for $x=0,1,2,\ldots,N-1$. In both equations **(1)** and **(2)** $\alpha(u)$ is defined as

$$\alpha(u)=\begin{cases}\sqrt{\frac{1}{N}} & for \quad u=0\\ \sqrt{\frac{2}{N}} & for \quad u\neq 0.\end{cases} \tag{3}$$

It is clear from **(1)** that for $u=0$, $C(u=0)=\sqrt{\frac{1}{N}}\sum_{x=0}^{N-1} f(x)$. Thus, the first transform coefficient is the average value of the sample sequence. In literature, this value is referred to as the *DC Coefficient*. All other transform coefficients are called the *AC Coefficients*[4].

Steps in Image Compression

Divide into 8x8 blocks and Transform to Frequency Domain :: DCT

$$C(u,v)=\alpha(u)\alpha(v)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)\cos\left[\frac{\pi(2x+1)u}{2N}\right]\cos\left[\frac{\pi(2y+1)v}{2N}\right], \tag{4}$$

for $u,v=0,1,2,\ldots,N-1$ and $\alpha(u)$ and $\alpha(v)$ are defined in **(3)**. The inverse transform is defined as

$$f(x,y)=\sum_{u=0}^{N-1}\sum_{v=0}^{N-1}\alpha(u)\alpha(v)C(u,v)\cos\left[\frac{\pi(2x+1)u}{2N}\right]\cos\left[\frac{\pi(2y+1)v}{2N}\right], \tag{5}$$

for $x,y=0,1,2,\ldots,N-1$.

Steps in Image Compression

Divide into 8x8 blocks and Transform to Frequency Domain :: DCT

- An Example 8x8 block of pixel data is shown here >
- The next step is to transform the 8x8 matrix from a positive range to one centered around zero.
- Since 8 bits are used for each value, each value is subtracted by 128.
- The resultant block is shown here ->

52	55	61	66	70	61	64	73
63	59	55	90	109	85	69	72
62	59	68	113	144	104	66	73
63	58	71	122	154	106	70	69
67	61	68	104	126	88	68	70
79	65	60	70	77	68	58	75
85	71	64	59	55	61	65	83
87	79	69	68	65	76	78	94

$x \longrightarrow$

−76	−73	−67	−62	−58	−67	−64	−55
−65	−69	−73	−38	−19	−43	−59	−56
−66	−69	−60	−15	16	−24	−62	−55
−65	−70	−57	−6	26	−22	−58	−59
−61	−67	−60	−24	−2	−40	−60	−58
−49	−63	−68	−58	−51	−60	−70	−53
−43	−57	−64	−69	−73	−67	−63	−45
−41	−49	−59	−60	−63	−52	−50	−34

$y \downarrow$

Steps in Image Compression

Divide into 8x8 blocks and Transform to Frequency Domain :: DCT

- Original 8x8 Pixel block

52	55	61	66	70	61	64	73
63	59	55	90	109	85	69	72
62	59	68	113	144	104	66	73
63	58	71	122	154	106	70	69
67	61	68	104	126	88	68	70
79	65	60	70	77	68	58	75
85	71	64	59	55	61	65	83
87	79	69	68	65	76	78	94

- Corresponding DCT coefficient block

$u \longrightarrow$

−415	−30	−61	27	56	−20	−2	0
4	−22	−61	10	13	−7	−9	5
−47	7	77	−25	−29	10	5	−6
−49	12	34	−15	−10	6	2	2
12	−7	−13	−4	−2	2	−3	3
−8	3	2	−6	−2	1	4	2
−1	0	0	−2	−1	−3	4	−1
0	0	−1	−4	−1	0	1	2

$v \downarrow$

Steps in Image Compression

Divide into 8x8 blocks and Transform to Frequency Domain :: DCT

1. The DCt is lossless in the sense that the inverse DCt will give back exactly the same initial information.
2. The values from the DCT are initially floating point.
3. They are changed to integers by "quantization".
4. The roundoff errors that happen will continue to happen in a say embedded system though.
5. "GROUP DISCUSSION" : How do you want the DCT step be done HW or SW?

Steps in Image Compression

Quantization

1. Quantization involves dividing each coefficient by an integer between 1 and 255 (if 8bits are used) and rounding off.
2. The quantization table is chosen to reduce the precision of each coefficient to no more than necessary.
3. The quantization table is carried along with the compressed file.
4. $B_{j,k} = round(\frac{A_{j,k}}{Q_{j,k}})$ for $j = 0, 1, 2, ...N_1 - 1$ and $k = 0, 1, 2, ...N_2 - 1$

Steps in Image Compression

Quantization

- Quantization Table

16	11	10	16	24	40	51	61
12	12	14	16	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

- Resultant 8x8 DCT coefficients after each coefficient is divided by corresponding Quantization factor

-26	-3	-6	2	2	-1	0	0
0	-2	-4	1	1	0	0	0
-3	1	5	-1	-1	0	0	0
-3	1	2	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Steps in Image Compression

Entropy Coding

1. This is done so that the coefficients are in order of increasing frequency
2. The higher frequency coefficients are more likely to be 0 after quantization
3. This improves the compression of run-length encoding
4. Later, do run-length encoding and Huffman coding

- Quantization Table

16	11	10	16	24	40	51	61
12	12	14	16	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

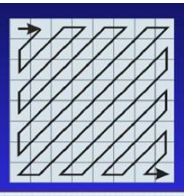
- Resultant 8x8 DCT coefficients after each coefficient is divided by corresponding Quantization factor

-26	-3	-6	2	2	-1	0	0
0	-2	-4	1	1	0	0	0
-3	1	5	-1	-1	0	0	0
-3	1	2	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Steps in Image Compression

Entropy Coding

- Zig Zag scan Pattern



- Resultant data

-26,
-3, 0,
-3, -2, -6,
2, -4, 1, -3,
1, 1, 5, 1, 2,
-1, 1, -1, 2, 0, 0,
0, 0, 0, 0, -1, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

1. The Output of the Zig-Zag scan is further encoded using Run Length Encoding procedure
2. In Run Length Encoding, consecutive pixels with the same value are encoded using a run-length and value pair
3. Example :: if 0x000 comes 10 times → 0x0A 0x000

Steps in Image Compression

Huffman EnCoding

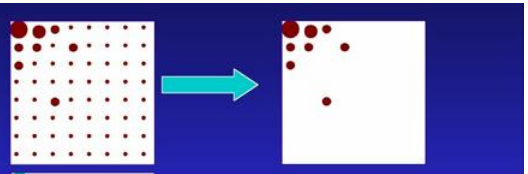
- The Output of the RLE data is further encoded using Huffman encoding
- Huffman coding refers to the use of a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol.
- An EOB (End Of Block) marker is put at the end and the data is written to a file

Char	Freq	Code
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110
l	1	11001
o	1	00110
p	1	10011
r	1	11000
u	1	00111
x	1	10010

Steps in Image Compression

Small Summary of Coding Methods

- Weighted scalar quantization: loss of precision, few non-zero coefficients are left
- Zig-zag scan: non-zero coefficients tend to be grouped together
- Run-Level encoding: encode each coefficient value as a (run,level) pair
  - run: number of zeros preceding values
  - lenght: non-zero value



Introduction to Image Processing

Acknowledgements

1. http://www.archive.org/details/Lectures_on_Image_Processing :: Richard Alan Peters II :: Dept of Electrical Engg & CS :: Vanderbilt University School of Engineering.
   (a) EECE253_01_Intro.pdf
   (b) EECE253_03_PointProcessing.pdf
   (c) EECE253_10_PixelizationQuantization.pdf
2. YCbCr to RGB Considerations :: YCbCr_Intersil_AppNote.pdf
3. http://www.slideshare.net/sanjivmalik/video-compression-basics :: Sanjiv Malik
4. Direct Use :: digicam.ppt :: http://www.facweb.iitkgp.ernet.in/ anupam/ppts.html :: Lecture Notes from Prof. Anupam Basu IIT KGP.
5. Embedded System Design - A Unified Hardware / Software Introduction :: Ch 7
6. Huffman_Coding.ppt :: Vida Movahedi

Image Processing

Hardware Software CoDesign

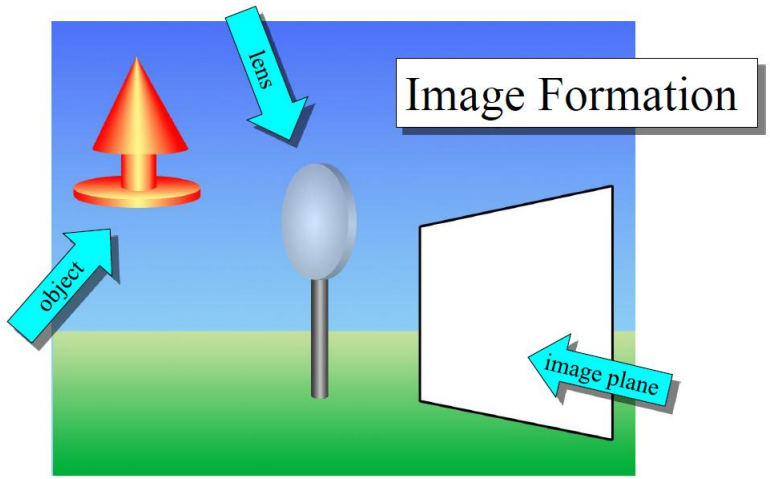
June 2013

Agenda

Image Processing

1. Introduction to Image Processing – Pixelization, Quantization
2. The Discrete Cosine Transform
3. Why Compress or Encode Images ?
4. The Steps in Image Compression
5. Digital Camera Example
6. Discussion on the Assignment

Introduction to Image Processing



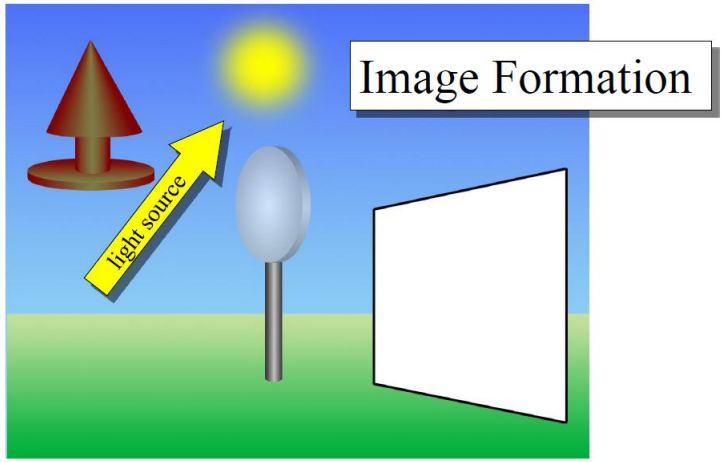


Image Formation
light source

3

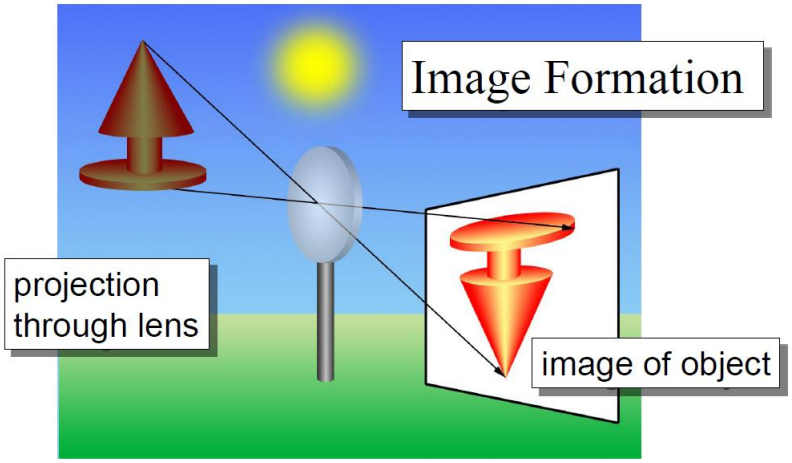


Image Formation
projection through lens
image of object

4

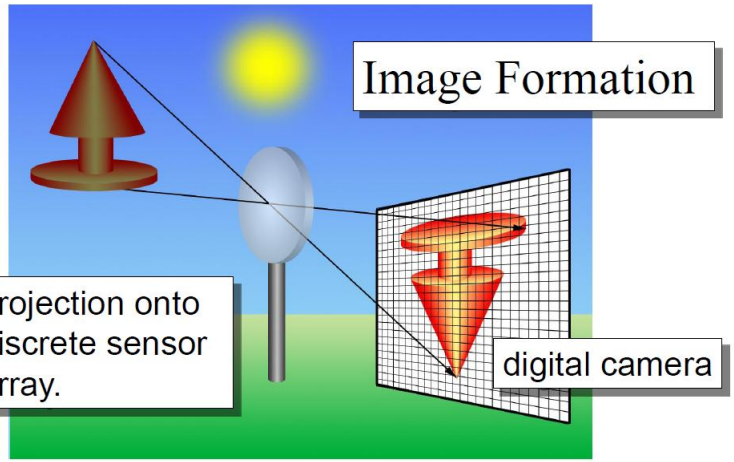


Image Formation
projection onto discrete sensor array.
digital camera

5

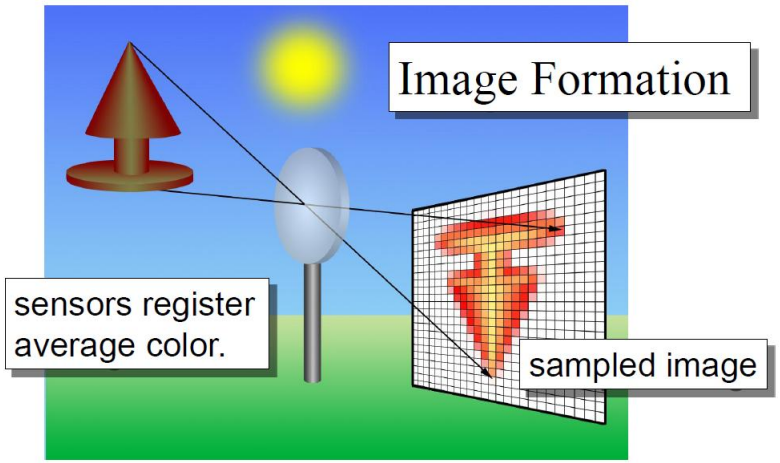
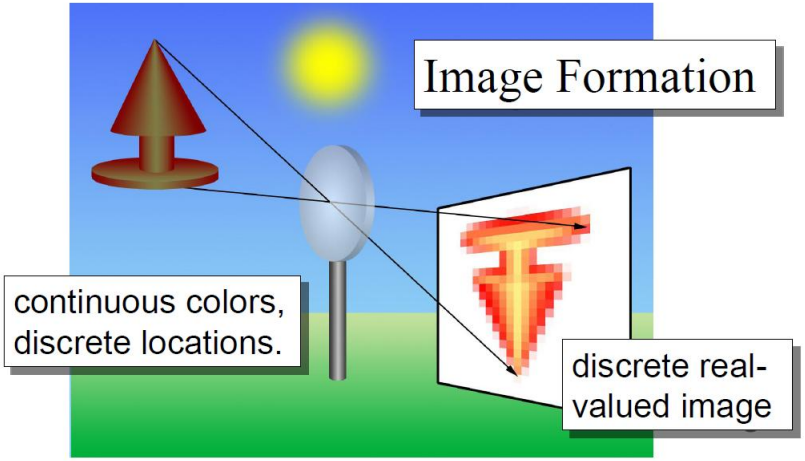
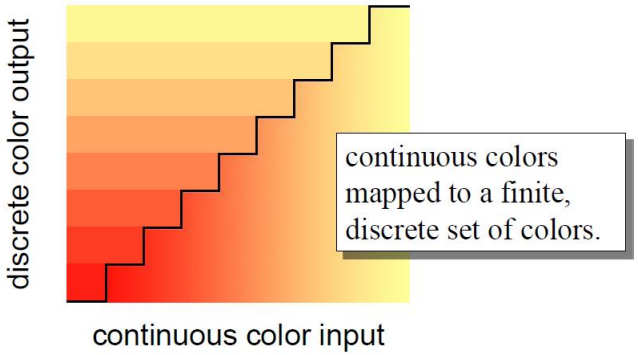


Image Formation
sensors register average color.
sampled image

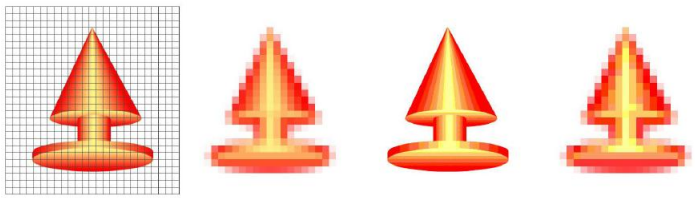
6



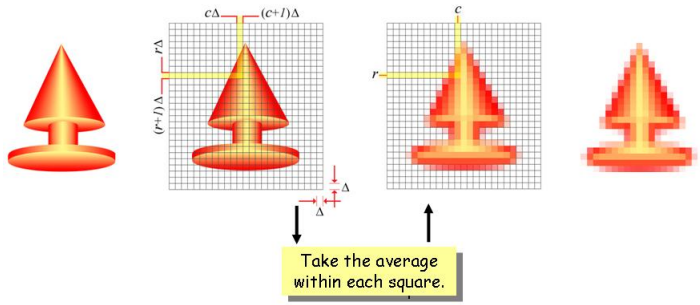
Digital Image Formation: Quantization



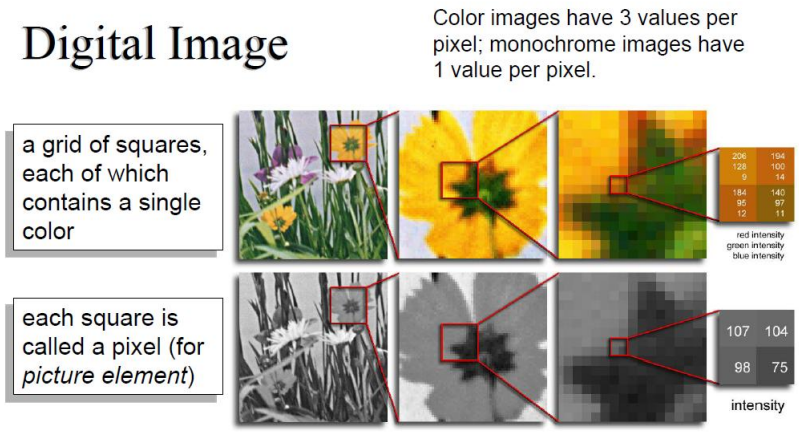
Sampling and Quantization



Pixelization :-



Pixelization :-



Why Compress / Encode Images

Original Image :: 352 x 288
24bits per pixel.

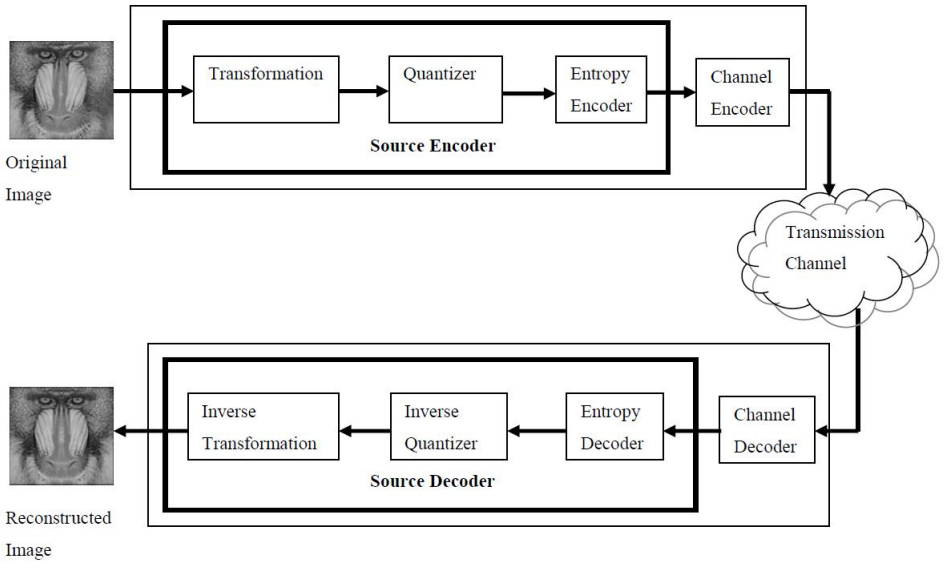
JPG FILE :: LT 20KB

Why Compress / Encode Images

Video Source	Output Data Rate[Kbits/sec]
Quarter VGA @20 frames/sec	36 864.00
CIF camera @30 frames/sec	72 990.72
VGA @30 frames/sec	221 184.00

Transmission Medium	Data Rate [Kbits/sec]
Wireline modem	56
GPRS (estimated average rate)	30
3G/WCDMA (theoretical maximum)	384

Steps in Image Compression



Components of a typical image/video transmission system

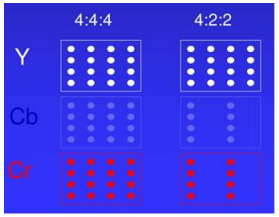
Steps in Image Compression

1. If the color is represented by RGB mode, translate it to YCrCb.
2. Divide the file into 8 x 8 blocks.
3. Transform the pixel information from the spatial domain to the frequency domain with the Discrete Cosine Transform.
4. Quantize the resulting values by dividing each coefficient by an integer value and rounding off to the nearest integer.
5. Look at the resulting coefficients in a zig-zag order. Do a run-length encoding of the coefficients ordered in this manner.
6. Follow by Huffman coding.

Steps in Image Compression

Converting RGB to YCbCr

1. YCbCr color mode stores color in terms of its luminance (brightness) and chrominance (hue)
2. The human eye is less sensitive to chrominance than luminance.
3. ?? Compression then can be achieved by storing more luma details than chroma details. ?? Called chroma-sub-sampling.
4. Formats 4:4:4, 4:2:2
5. When Converted from RGB $\rightarrow$ 4:4:4 $\rightarrow$ 4:2:2 the bandwidth is reduced by 50% (achieving compression).



Steps in Image Compression

Converting RGB to YCbCr

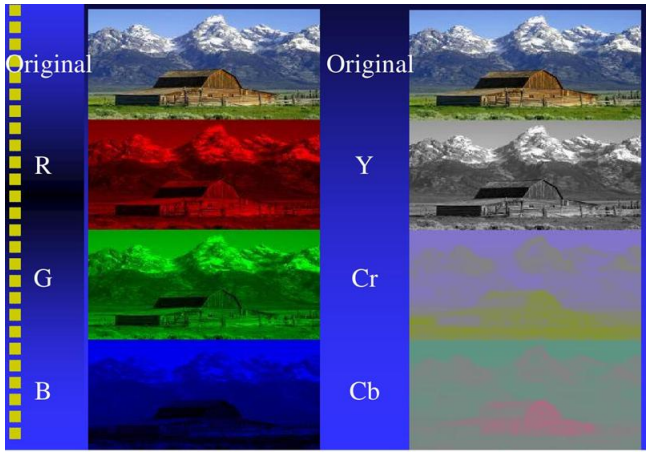
The BT.601 equations are used by many video ICs to convert between digital R′G′B′ data and YCbCr are:

$$Y = (77/256)R' + (150/256)G' + (29/256)B'$$
$$Cb = -(44/256)R' - (87/256)G' + (131/256)B' + 128$$
$$Cr = (131/256)R' - (110/256)G' - (21/256)B' + 128$$
$$R' = Y + 1.371(Cr - 128)$$
$$G' = Y - 0.698(Cr - 128) - 0.336(Cb - 128)$$
$$B' = Y + 1.732(Cb - 128)$$

1. From a Hardware - Software point of view :: This is a computation requiring bandwidth.
2. GROUP DISCUSSION :: How does one implement in HW.

Steps in Image Compression

Converting RGB to YCbCr



Steps in Image Compression

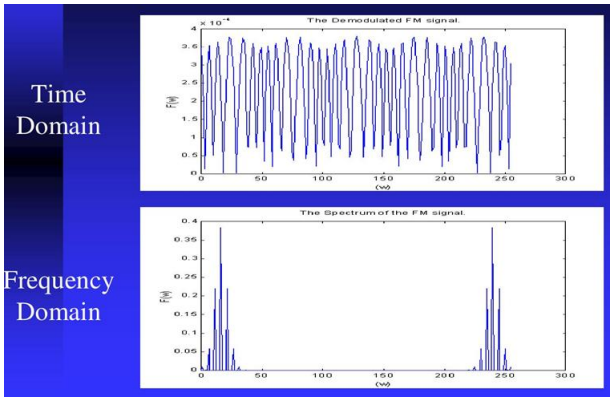
Divide into 8x8 blocks and Transform to Frequency Domain :: DCT

1. The DCT transforms the data from the spatial domain to the frequency domain.
2. The spatial domain shows the amplitude of the color as you move through space.
3. The frequency domain shows how quickly the amplitude of the color is changing from one pixel to the next in an image file.
4. For the 8 x 8 matrix of color data, we'll get an 8 x 8 matrix of coefficients for the frequency components.
5. GROUP DISCUSSION :: What is being done here actually?

Steps in Image Compression

Divide into 8x8 blocks and Transform to Frequency Domain :: DCT

1. GROUP DISCUSSION :: What is being done here actually?
2. "Use Time Domain –¿ Frequency Domain" transformation.
3. "Use the fact that human eye is imperfect"
4. "Achieve lossy compression, which satisfies the bandwidth / size consideration"



Steps in Image Compression

Divide into 8x8 blocks and Transform to Frequency Domain :: DCT

1. Discrete Cosine Transform (DCT) is a Fourier-related transform similar to the Discrete Fourier Transform (DFT), but using only real numbers.
2. The most common variant of the DCT is the type II DCT; its inverse is called IDCT (inverse DCT)
3. The frequency domain is a better representation for the data because it makes it possible to separate out → and throw away → information that isn't very important for human perception.
4. The human eye is not very sensitive to high frequency changes → especially in photographic images, so the high frequency data, to some extent, can be discarded

Steps in Image Compression

Divide into 8x8 blocks and Transform to Frequency Domain :: DCT

The most common DCT definition of a 1-D sequence of length N is

$$c(u) = \alpha(u)\sum_{x=0}^{N-1} f(x)\cos\left[\frac{\pi(2x+1)u}{2N}\right], \quad (1)$$

for $u = 0,1,2,\ldots,N-1$. Similarly, the inverse transformation is defined as

$$f(x) = \sum_{u=0}^{N-1} \alpha(u)c(u)\cos\left[\frac{\pi(2x+1)u}{2N}\right], \quad (2)$$

for $x = 0,1,2,\ldots,N-1$. In both equations **(1)** and **(2)** $\alpha(u)$ is defined as

$$\alpha(u) = \begin{cases} \sqrt{\frac{1}{N}} & for \quad u = 0 \\ \sqrt{\frac{2}{N}} & for \quad u \neq 0. \end{cases} \quad (3)$$

It is clear from **(1)** that for $u = 0$, $c(u=0) = \sqrt{\frac{1}{N}}\sum_{x=0}^{N-1} f(x)$. Thus, the first transform coefficient is the average value of the sample sequence. In literature, this value is referred to as the *DC Coefficient*. All other transform coefficients are called the *AC Coefficients*[4].

Steps in Image Compression

Divide into 8x8 blocks and Transform to Frequency Domain :: DCT

$$C(u,v) = \alpha(u)\alpha(v)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)\cos\left[\frac{\pi(2x+1)u}{2N}\right]\cos\left[\frac{\pi(2y+1)v}{2N}\right], \quad (4)$$

for $u,v = 0,1,2,\ldots,N-1$ and $\alpha(u)$ and $\alpha(v)$ are defined in **(3)**. The inverse transform is defined as

$$f(x,y) = \sum_{u=0}^{N-1}\sum_{v=0}^{N-1} \alpha(u)\alpha(v)C(u,v)\cos\left[\frac{\pi(2x+1)u}{2N}\right]\cos\left[\frac{\pi(2y+1)v}{2N}\right], \quad (5)$$

for $x,y = 0,1,2,\ldots,N-1$.

Steps in Image Compression

Divide into 8x8 blocks and Transform to Frequency Domain :: DCT

- An Example 8x8 block of pixel data is shown here >
- The next step is to transform the 8x8 matrix from a positive range to one centered around zero.
- Since 8 bits are used for each value, each value is subtracted by 128.
- The resultant block is shown here ->

52	55	61	66	70	61	64	73
63	59	55	90	109	85	69	72
62	59	68	113	144	104	66	73
63	58	71	122	154	106	70	69
67	61	68	104	126	88	68	70
79	65	60	70	77	68	58	75
85	71	64	59	55	61	65	83
87	79	69	68	65	76	78	94

$x \longrightarrow$

−76	−73	−67	−62	−58	−67	−64	−55
−65	−69	−73	−38	−19	−43	−59	−56
−66	−69	−60	−15	16	−24	−62	−55
−65	−70	−57	−6	26	−22	−58	−59
−61	−67	−60	−24	−2	−40	−60	−58
−49	−63	−68	−58	−51	−60	−70	−53
−43	−57	−64	−69	−73	−67	−63	−45
−41	−49	−59	−60	−63	−52	−50	−34

$y \downarrow$

Steps in Image Compression

Divide into 8x8 blocks and Transform to Frequency Domain :: DCT

- Original 8x8 Pixel block

52	55	61	66	70	61	64	73
63	59	55	90	109	85	69	72
62	59	68	113	144	104	66	73
63	58	71	122	154	106	70	69
67	61	68	104	126	88	68	70
79	65	60	70	77	68	58	75
85	71	64	59	55	61	65	83
87	79	69	68	65	76	78	94

- Corresponding DCT coefficient block

$u \longrightarrow$

−415	−30	−61	27	56	−20	−2	0
4	−22	−61	10	13	−7	−9	5
−47	7	77	−25	−29	10	5	−6
−49	12	34	−15	−10	6	2	2
12	−7	−13	−4	−2	2	−3	3
−8	3	2	−6	−2	1	4	2
−1	0	0	−2	−1	−3	4	−1
0	0	−1	−4	−1	0	1	2

$v \downarrow$

Steps in Image Compression

Divide into 8x8 blocks and Transform to Frequency Domain :: DCT

1. The DCt is lossless in the sense that the inverse DCt will give back exactly the same initial information.
2. The values from the DCT are initially floating point.
3. They are changed to integers by "quantization".
4. The roundoff errors that happen will continue to happen in a say embedded system though.
5. "GROUP DISCUSSION" : How do you want the DCT step be done HW or SW?

Steps in Image Compression

Quantization

1. Quantization involves dividing each coefficient by an integer between 1 and 255 (if 8bits are used) and rounding off.
2. The quantization table is chosen to reduce the precision of each coefficient to no more than necessary.
3. The quantization table is carried along with the compressed file.
4. $B_{j,k} = round(\frac{A_{j,k}}{Q_{j,k}})$ for $j = 0, 1, 2, ...N_1 - 1$ and $k = 0, 1, 2, ...N_2 - 1$

Steps in Image Compression

Quantization

- Quantization Table

16	11	10	16	24	40	51	61
12	12	14	16	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

- Resultant 8x8 DCT coefficients after each coefficient is divided by corresponding Quantization factor

-26	-3	-6	2	2	-1	0	0
0	-2	-4	1	1	0	0	0
-3	1	5	-1	-1	0	0	0
-3	1	2	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Steps in Image Compression

Entropy Coding

1. This is done so that the coefficients are in order of increasing frequency
2. The higher frequency coefficients are more likely to be 0 after quantization
3. This improves the compression of run-length encoding
4. Later, do run-length encoding and Huffman coding

- Quantization Table

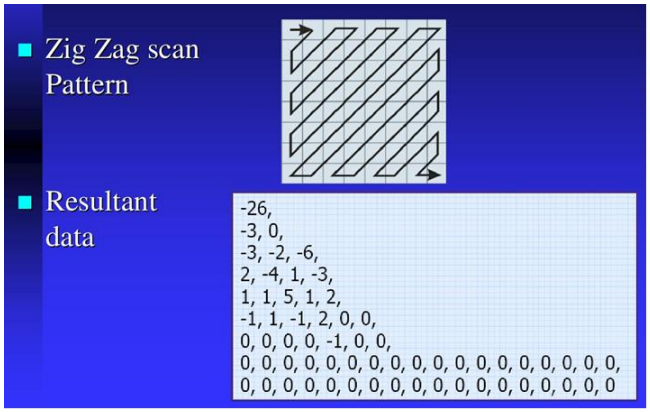
16	11	10	16	24	40	51	61
12	12	14	16	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

- Resultant 8x8 DCT coefficients after each coefficient is divided by corresponding Quantization factor

-26	-3	-6	2	2	-1	0	0
0	-2	-4	1	1	0	0	0
-3	1	5	-1	-1	0	0	0
-3	1	2	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Steps in Image Compression

Entropy Coding



1. The Output of the Zig-Zag scan is further encoded using Run Length Encoding procedure
2. In Run Length Encoding, consecutive pixels with the same value are encoded using a run-length and value pair
3. Example :: if 0x000 comes 10 times → 0x0A 0x000

Steps in Image Compression

Huffman EnCoding

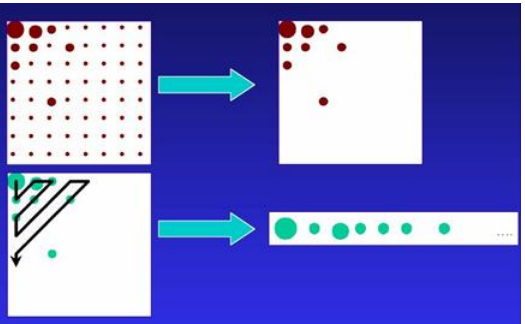
- The Output of the RLE data is further encoded using Huffman encoding
- Huffman coding refers to the use of a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol.
- An EOB (End Of Block) marker is put at the end and the data is written to a file

Char	Freq	Code
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110
l	1	11001
o	1	00110
p	1	10011
r	1	11000
u	1	00111
x	1	10010

Steps in Image Compression

Small Summary of Coding Methods

- Weighted scalar quantization: loss of precision, few non-zero coefficients are left
- Zig-zag scan: non-zero coefficients tend to be grouped together
- Run-Level encoding: encode each coefficient value as a (run,level) pair
  - run: number of zeros preceding values
  - lenght: non-zero value



Introduction to Image Processing

Acknowledgements

1. http://www.archive.org/details/Lectures_on_Image_Processing :: Richard Alan Peters II :: Dept of Electrical Engg & CS :: Vanderbilt University School of Engineering.
   (a) EECE253_01_Intro.pdf
   (b) EECE253_03_PointProcessing.pdf
   (c) EECE253_10_PixelizationQuantization.pdf
2. YCbCr to RGB Considerations :: YCbCr_Intersil_AppNote.pdf
3. http://www.slideshare.net/sanjivmalik/video-compression-basics :: Sanjiv Malik
4. Direct Use :: digicam.ppt :: http://www.facweb.iitkgp.ernet.in/ anupam/ppts.html :: Lecture Notes from Prof. Anupam Basu IIT KGP.
5. Embedded System Design - A Unified Hardware / Software Introduction :: Ch 7
6. Huffman_Coding.ppt :: Vida Movahedi

Device Drivers

Hardware Software CoDesign

January 2012

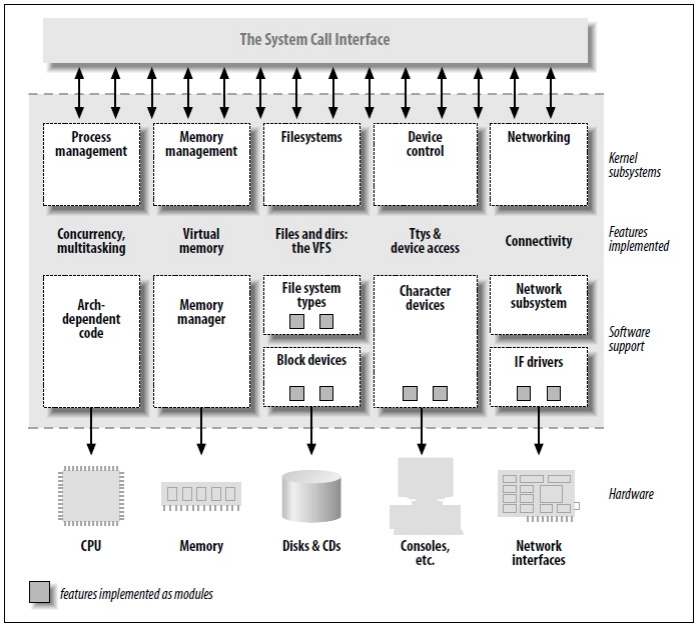
Agenda

Device Drivers

1. Introduction to Device Drivers (quick repeat for completness)
2. More on Device Drivers
3. Taking stock before EndSem Examination
4. Collection of Answering Machine Assignment

Device Drivers

Kernel view of Things



A split view of the kernel

Device Drivers

Kernel view of Things

1. The Kernel is the big chunk of executable code in charge of handling all requests → computing power, memory, network connectivity, any other resource allocation.
2. The distinction between the different kernel tasks isn't always clearly marked, the kernel's role can be split into :-
3. *Process Management* :: Creating and destroying processes and handling their connection to the outside world (IO). Communication among different processes (through signals, pipes, IPC). The scheduler, which controls how processes share the CPU, is part of process management.
4. *Memory Mangement* :: The computer's memory is a major resource, and the policy to deal with it is a critical one for overall system performance. The kernel builds up a virtual address space for all processes. The different parts of the kernel interact with the memory-management subsystem through simple `malloc / free` and other more complex memory functions.
5. *FileSystems* :: Unix is heavily based on the filesystem concept; almost everything in Unix can be treated as a file. The kernel builds a structured filesystem on top of unstructured hardware, and the resultant file abstraction is heavily used throughout the whole system.
6. *Device Control* :: Almost every system operation eventually maps to a physical device. With the exception of the processor, memory and a very few other entities; any and

all device control operations are performed by code that is specific to the device being addressed. That code is called *Device Driver*. The kernel must have embedded in it a device driver for every peripheral present on a system (hard drive, printer, kbd, tape..)

7. *Networking* :: Networking must be managed by the operating system, because most network operations are not specific to a process. Incoming packets are asynchronous events. These packets must be collected, identified, and dispatched before a process takes care of them. The system is in charge of delivering data packets across program and network interfaces, and it must control execution of programs according to their network activity.

Device Driver

Device Driver Types

1. Service Device Driver :: If the device driver will be communicating directly with the hardware device, this interface will be the memory ranges, registers and/or ports through which I/O to the device takes place.

2. System Device Driver :: If the device driver will be communicating with its device via an intermediate device, this interface will be whatever APIs the driver for the intermediate device exposes. Eg. a GPS device driver will use the serial port driver API – the stream interface functions – to communicate with a GPS receiver.

Device Driver

What is a Device Driver

1. A set of software procedures or apiś which enable higher level of programs (applications) to interact with hardware device

2. A Device Driver "Controls", "Manages" and "Configures" devices connected to the system

3. The driver ha few functions and provides the logic to initialize and communicate with the hardware

Device Driver

Device Driver Need & Challenges

1. By providing a clear abstraction between the driver and the application, one can essentially free the application of the specifics of a certain peripheral and port it more easily to new hardware

2. Device Drivers have a tight connection to the target device and the development environment, these are usually not portable. Eg., device drivers across microcontroller families.

3. Device Drivers consist of a lot of "bit bashing" and register programming. One needs to get all the details right → the bits, sequences of initialization and exit, timing (say flash, ddr), or else the hardware would malfunction or not provide the desired output.

4. Device Drivers also enable Debug of system malfunction (due to modularization). E.g. 85% of failures in WinXP were from buggy device drivers.

5. GROUP DISCUSSION :: So, How would you go about developing a Device Driver?

Device Driver

Device Driver Development → HardWare Side

1. Reading the hardware manual and learning the chip internals
2. Learn about the hardware / platform. Identify the interface between the driver and the device
   (a) How the device is reset
   (b) How the device is "address mapped"
   (c) Return codes and software protocols recognized by the device
   (d) What are the types of DMA transfers possible with the device
   (e) How the device reports hardwar failures
   (f) How the device sends / responds to interrupts
3. Test the hardware to make sure it is functioning. This becomes important for a newly developed device.

Device Driver

Device Driver Development → SoftWare Side

1. What kind of device driver library will enable the application ?
   (a) Identify the APIs that the driver must expose
   (b) Initialization and De-initialization routines (eg. setting up of baud rate or timer periods)
   (c) Run time control (Read, Write, Interrupts, Signal Responses..)
   (d) Notification and any additional interfaces that the Device Driver needs to use or other OS APIs or services
   (e) Version Control for Software
   (f) Bug Tracking and closure utility
   (g) Suite of testcases to test scenarios where the Device Driver is used
   (h) Integration into the System and the OS Kernel

Device Driver

Device Driver Development → SoftWare Side

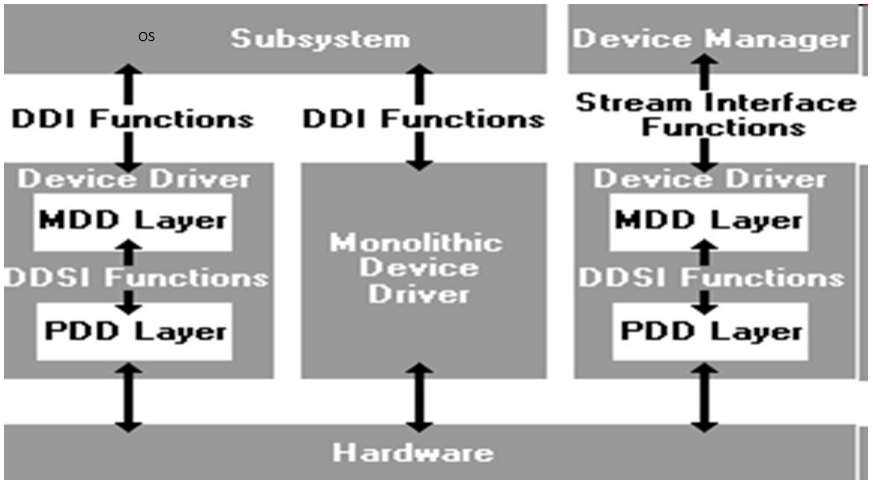
1. Monolithic device driver library
   (a) Based on a single piece of code that exposes hardware devices functionality directly to the OS
   (b) Accesses device(s) directly
2. Layered device driver library
   (a) A model device driver based on an "upper" layer which is logical and "lower" layer which is physical
   (b) RTOS abstraction layer
   (c) Compiler abstraction layer ?
   (d) Hardware abstraction layer ?

Device Driver

Device Driver Development → SoftWare Side PROs and CONs of Monolithic Vs Layered

1. Monolithic :: Performance is better as it avoids calling many functions.
2. Layered ::
   (a) Because of modularity, the structure of software is easy to understand.
   (b) It is easy to add or modify features of the overall application as it evolves and gets deployed.
   (c) Because there is one module that ever interacts directly with the peripherals registers, the state of the hardware device can be more accurately tracked.
   (d) Software changes that result from hardware changes are localized to the device driver, making software more portable.
   (e) Enhance the reusability, but bit of extra effort up front, at design time, in order to realize the savings.

Device Driver



1. PDD :: Platform Dependent Driver (physical layer)
2. MDD :: Model Device Driver (logical layer)
3. DDI :: Device Driver Interface
4. DDSI:: Device Driver Service Provider Interface

Device Driver

Concepts → Mutual Exclusion of Device Access

1. One of the most basic requirements of a Device Driver is the need to ensure that only one application task at a time can request an input or output operation on a specific device
2. Example :: Same hardware I2C is shared between multiple devices like FM tuner, LCD, Temperature sensor on Hardware board
3. Semaphores can be used in the open and close function to ensure mutual exclusion
4. In case the target allows multiple simultaneous accesses, like in some complex file IO, it can be through system of arbitration logic

Device Driver

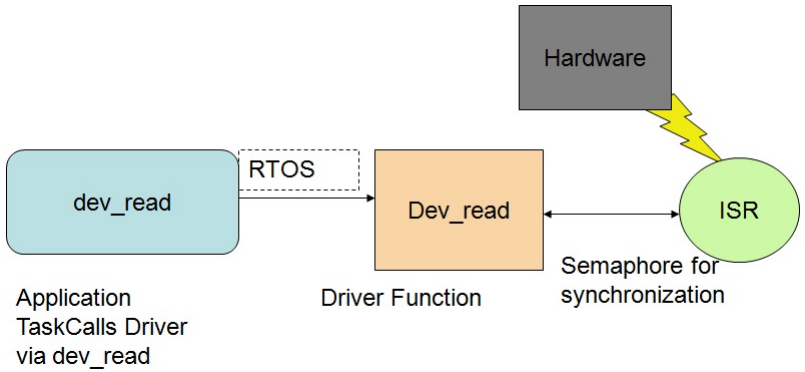
Concepts → Synchronous Vs Asynchronous driver operation

1. Do you want the application task that called the device driver to wait for the result of the IO operation that it asked for? → Blocking (Synchronous)
2. OR do you want the application task to continue to run while the device driver is doing its IO operation? → Not Blocking (Asynchronous)
3. Question to Ask (Software Side) :: If my task requests an IO operation through a driver, then what work can it usefully do before that IO operation is done?
4. Question to Ask (Hardware Side) :: Does the completion of the IO operation through driver provide a logical status only or it also provides an interrupt?
5. Based on such questions, one can decide Synchronous Vs Asynchronous drivers
6. GROUP DISCUSSION :: if a task calls a driver via a "read" call, the driver function implementing the call would act as a subroutine of the caller task. And if this driver attempts to get a semaphore token that is not present, the driver function would be blocked from continuing exectuion; and together with it, the requesting task would be in waiting state → Synchronous or Asynchronous

Device Driver

Concepts → Synchronous Vs Asynchronous driver operation

1. GROUP DISCUSSION :: if a task calls a driver via a "read" call, the driver function implementing the call would act as a subroutine of the caller task. And if this driver attempts to get a semaphore token that is not present, the driver function would be blocked from continuing exectuion; and together with it, the requesting task would be in waiting state → Synchronous* or Asynchronous



2. the read call is dev_read() from the OS or RTOS side.
3. this in turn calls the device driver routine named dev_read()

4. the drivers read function will request the device hardware to perform a read operation, and then it will attempt to get a token from the semaphore to its right.

5. Since the semaphore initially has no tokens, the driver "read" function, and hence all the related tasks will be "BLOCKED".

6. The hardware interrupt triggers the execution of the ISR.

7. When device hardware completes the previous read operation, it would trigger this ISR, that will put a token into the semaphore.

8. this is the semaphore token for which the Task is waiting, and will become unblocked and proceed to fetch the newly-read data from hardware

```
dev_read() {
  Start IO Device Read Operations;
  Get Synchronizer Semaphore Token /*Wait for Semaphore token */;
  Get Device Status and Data;
  Give Device Information to Requesting Task;
}

dev_isr {
  Calm down the hardware device;
  Put a token into synchronizer semaphore;
}
```

1. GROUP DISCUSSION :: Write a similar code for Asynchronous Drivers (NON Blocking)

Device Driver

Concepts → Synchronous Vs Asynchronous driver operation

1. In an "Asynchronous" driver, the application task that called the device driver may continue executing, without waiting for the result of the IO operation it requested

2. it is more complex than synchronous drivers

3. GROUP DISCUSSION :: Write a similar code for Asynchronous Drivers (NON Blocking)

```
dev_read_async() {
  Get Message from the Queue /* Wait only if Queue is Empty */;
  Start new IO device read operation;
  Give old Device Information to the Requesting Task;
}

dev_read_async_isr() {
  Calm down the hardware device;
  Get Data/Status information from Hardware;
  Package this information into a Message;
  Put Message into the Queue;
}
```

1. Complexity is :: How much buffer is required, need to create an memory management structure around it → stack, fifo

Device Driver

Concepts → Miscellaneous

1. Compiler Selected :: for code optimization, selection of variable types (global, volatile, loops..)

2. Memory Mangement :: usage of stacks, fifos

3. Interrupt Vs Polling

4. Sampling Frequency :: How often to read from ADC?

5. DMA Vs Byte/Word/Double Mode

6. How much to buffer

7. Data structure

Device Driver

Concepts → Performance Analysis

1. Interrupt Latency :: Response time, very key to RTOS

2. Performance and Throughput :: Memory read /write speed from a SD Card

3. Memory Foot Print :: For a Network Interface Controller (NIC) what is the memory requirement on per packet basis, or the max number of packets if burst modes are supported

4. CPU Load :: "GROUP DISCUSSION :: give examples"

5. Hardware Resource Used :: Some device drivers may use multiple hardware resources. Say a higher level CodecDriver would use DMA, SPI, Memory interfaces including DSP Core bandwidth.

Device Driver

Example → LoopBack (codec) driver

1. Writing a codec driver for a DSP involves programming three different peripherals :: the codec itself, the SPI, and the DMA controller

```
void main() {
  void* buf0, buf1, buf2, buf3;
        /*Allocate buffers for the SIO buffer exchange*/
buf0 = (void*) MEM_calloc(0, BUFSIZE, BUFALIGN);
buf1 = (void*) MEM_calloc(0, BUFSIZE, BUFALIGN);
buf2 = (void*) MEM_calloc(0, BUFSIZE, BUFALIGN);
buf3 = (void*) MEM_calloc(0, BUFSIZE, BUFALIGN);

      /*Create the task and open the I/O streams */
      TSK_create(echo);
      inStream = SIO_create("/codec", SIO_INPUT, BUFSIZE);
      outStream= SIO_create("/codec", SIO_OUTPUT, BUFSIZE);

      /*Start the scheduler when main() exits*/
}
```


Device Driver

Example → LoopBack (codec) driver

1. The application uses the SIO_create() call to create the channels.
2. The SIO_create() function arguments indicate the design decisions when implementing the driver.
3. The SIO stream can be opened either for reading or writing, but not both. Example :: LCD is only write and KBD is only read.
4. If BiDi communication is required, the application opens two channels (as above example).
5. Most Codecs operate of fixed-sized frames of data, so BUFSIZE can be a constant.

Device Driver

Example → LoopBack (codec) driver

```
void echo(){
  int sizeRead; // Number of buffer units read
  unsigned short *inbuf, *outbuf;

  /* Issue the first & second empty buffers to input stream.*/
  SIO_issue(inStream, buf0, SIO_bufsize(inStream), NULL);
  SIO_issue(inStream, buf1, SIO_bufsize(inStream), NULL);

  /* Issue the first & second empty buffers to output stream. */
  SIO_issue(outStream, buf2, SIO_bufsize(outStream), NULL);
  SIO_issue(outStream, buf3, SIO_bufsize(outStream), NULL);

  for (;;) {
  /* Reclaim full buffer from input stream
     and empty from output stream. */
     sizeRead = SIO_reclaim(inStream, (void**)&inbuf, NULL);
     SIO_reclaim(outStream, (void**)&outbuf, NULL);

  /* Copy data from input buffer to output buffer.*/
  for (int i = 0; i < sizeRead; i++) {
    outbuf[i] = inbuf[i];
  }

  /* Issue full buffer to output stream
     and empty to input stream. */
  SIO_issue(outStream, outbuf, nmadus, NULL);
  SIO_issue(inStream, inbuf, SIO_bufsize(inStream), NULL);
  }
}
```


Device Driver

Example → LoopBack (codec) driver

1. mdCreateChan() :: Create a channel. Add data structure which can include channel state information, information about the current IO packet being processed, a linked list of packets queued for processing, and the call back function that is to be used to notify the driver that a packet processing is complete.
2. mdBindDev() :: Initializes the devices
3. mdUnBindDev() :: Freez any resources allocated by the driver
4. mdSubmitChan() :: Processes IO request (read / write). The function will receive an IO packet from the driver and either put the packet in queue if the function is already working on a previous job, or start working on the packet right away. All of the state driver information required to accomplish this is contained in the channel-object strucutre. Interrupts is disabled in this function (by design) to maintain the coherency of the channel state. Assumption is that the period is short for proper driver functioning.
5. mdControlChan() :: Enables the application to perform device-specific control, such as device reset, volume change.

Device Driver

Example → LoopBack (codec) driver

```
//data structure used & created in mdCreateChan()
typedef struct {
     bool inuse; // TRUE => channel has been opened
     int imode; // IOM_INPUT or IOM_OUTPUT
     IOM_Packet *dataPacket; // current active I/O pkt
     QUE_Obj pendList; // list of packets forI/O
     unsigned int *bufptr; // pointer *within* current buf
     unsigned int bufcnt; // remaining samples
     IOM_TiomCallback cbFxn; // used to notify client
     void* cbArg; // arg passed with callback function
   } ChanObj, *ChanHandle;

static int mdSubmitChan(void* chanp, IOM_Packet *packet) {
   ChanHandle chan = (ChanHandle) chanp;
   unsigned int imask;

   imask = HWI_disable(); // disable interrupts
   if (chan->dataPacket == NULL) {
     /* Start I/O job. */
     chan->bufptr = (unsigned int *)packet->addr;
     chan->bufcnt = packet->size;

     // dataPacket must be set last, to synchronize with ISR.
     chan->dataPacket = packet;
  } else {
```



```
     /* There is an I/O job already pending; queue packet. */
     QUE_put(&chan->pendList, packet);
  }
  HWI_restore(imask); // restore interrupts
  return (IOM_PENDING);
}
```

Device Driver

Example → LoopBack (codec) driver

1. GROUP DISCUSSION :: Write a state diagram of the use of the driver by the application.

Device Driver

General look of a Driver The structure of a driver is similar for a peripheral device (say TouchScreen)

1. There is an init routine for initializing the hardware, setting memory from the kernel and hooking the driver-routines into the kernel.
2. There is a data structure that is initialized with routines that are provided with the device. This strucutre is key and is used by the applications.
3. Mostly there are open and release routines.
4. Mostly there are routines for reading and writing data to or from the driver, a ioctl routine that can perform special commands to driver like config requests or options.
5. Mostly there are routines for interrupt handling if hardware supports.
6. The Driver itself, can be either "Compiled as part of Kernel" or "Dynamically Loaded" at runtime. The only differences between a loadable "Module" and a kernel linked driver are a special `init()` routine that is called when the module is loaded into the kernel and a cleanup routine that is called when the module is removed.

Device Driver

ReCall → Touch Screen System Device Driver

1. Configure the controller hardware.
2. Determine if the screen is touched.
3. Acquire Stable, debounced position measurements.
4. Calibrate the touch screen.
5. Send changes in touch status and position to the higher level graphics software.

Device Driver

ReCall → Touch Screen System Device Driver

Configure the controller hardware.

1. Create a function named `TouchConfigureHardware()`
2. Decide – Should the driver be interrupt driven or polling driven.
3. In case of interrupts, the driver would actually use two :-
   (a) An interrupt to wake up when the screen is initially touched, known as the `PEN_DOWN` interrupt.
   (b) A second interrupt to signal when the ADC is available with the set of data conversions (X, Y).

Determine if the screen is touched.

1. Create a function named `WaitForTouchState ()`
2. When the controller is in the detection mode and a touch is detected, an internal interrupt can be generated called `PEN_DOWN IRQ`.
3. This detection is based on Y-axis touch plane tied high, X-axis touch plane tied low, and on the basis of touch the planes are shorted together and Y-axis plane is pulled low.
4. The driver task would not consume any CPU time until the `PEN_DOWN IRQ` event occurs. It would wake up and go into conversion mode only once the user touches screen.

Device Driver

ReCall → Touch Screen System Device Driver

Acquire Stable, debounced position measurements – Reading touch data

1. Create a function named `TouchScan()`. The outline of the procedure would be :-
   (a) Check to see if the screen is touched.
   (b) Take several raw readings on each axis for later filtering.
   (c) Check to see if the screen is still touched.
   (d) (Depending on H/W) store the readings to a memory block, or use FIFO entries.
   (e) (Depending on H/W) if the ADC output is 12bit, either pack data into 16bit (aligned) locations or chop/dither them to 8bit.
   (f) Taking Stable readings (filtering by using oversampling) is necessary for higher level drivers to act appropriately. NOTE :: This filtering could be a h/w function if CPU bandwidth is critical factor.

Calibrate the touch screen.

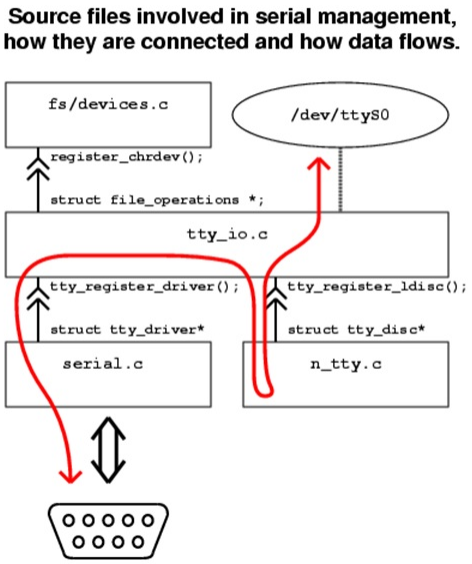
1. In an ideal scenario, the calibration would be run once during initial product power up and reference values saved in "non-volatile" memory.
2. Create a function name `CalibrateTouchScreen ()` in case the user wants to calibrate using a graphical target on screen.

Send changes in touch status and position to the higher level graphics software.

1. Create a function named `GetScaledTouchPosition()`. This is a routine to read raw values and convert them to screen co-ordinates.
2. Create a function named `TouchTask ()`. This routine calls the actual task the user intended to be run while using the touch screen.

Device Driver

Serial Device Driver Example from Linux



Taking Stock

1. Introduction of the course – Harware developments from pure function based to cpu based design. HW - SW codesign flow.
2. Simple example of multiplier for 8085
3. Basic themes of HSCD – Modeling, Analysis and Estimation, System Level Partitioning, synthesis and interfacing, Implementation Generation, Co-simulation and Emulation.
4. Basic Pitfalls – Transient Overloads, Analysis Paralysis, Simulation & complexity, Applications of SoC.
5. Example of Touch Screen Controller
6. Basic Modeling Requirements, leading to SystemC
7. State Chart Diagrams
8. Modeling using SystemC
9. Analysis using Process Path Algorithm(s) and parallel process execution on one CPU.
10. Task Scheduling on one CPU – Rate Monotonic Priority Assignment algorithm, Deadline driven analysis, Mixed Scheduling.

11. Partitioning paper by Kalavade and Lee – Binary Partitioning, Extended Partitioning, Global Criticality Local Phase (GCLP) algorithm.
12. Example of GCD program C → Hardware method.
13. FPGA and Emulation systems :: descriptions, differences and their usage
14. Compiler, Linker, Loader :: Basic Steps, Intermediate Format Generation, Challenges for ASISP and ReTargetable Compilers. With examples on point items.
15. Programmers View and SW Development Environment of ARM CortexM3
16. Static Image Processing – Introduction, DCT, Quantization, Huffman Encoding, Run Length encoding. With point examples.
17. Video Image Processing – Motion Vector, Motion Estimation, Frame Types.
18. Device Drivers – Introduction and basic Methodology. Example of Touch Screen, Serial Loopback.
19. Answering Machine Assignment – Alloted time for group discussions
20. Could not cover :: ReConfigurable Computing :: Heterogeneous and Homogeneous Multiprocessor architectures, On Chip Communication Architectures (Network On Chip NoC) concepts

Device Drivers

Acknowledgements

1. Class Notes and Slides from Puneetha Mukherjee
2. Linux Device Drivers :: 3rd Edition