IMIT CUTTACK

M.TECH 2ND SEM

EMBEDDED SYSTEM

SUJEET KUMAR NAYAK
EMAIL ID: sujeetkumarnayak.in@gmail.com

MODULE-1

DEFINATION OF Embedded System

An embedded system can be thought of as a computer hardware system having software embedded in it. An embedded system can be an independent system or it can be a part of a large system. An embedded system is a microcontroller or microprocessor based system which is designed to perform a specific task. For example, a fire alarm is an embedded system; it will sense only smoke.

An embedded system has three components −

- It has hardware.
- It has application software.
- It has Real Time Operating system (RTOS) that supervises the application software and provide mechanism to let the processor run a process as per scheduling by following a plan to control the latencies. RTOS defines the way the system works. It sets the rules during the execution of application program. A small scale embedded system may not have RTOS.

So we can define an embedded system as a Microcontroller based, software driven, reliable, real-time control system.

Features of Embedded systems

Single-functioned − An embedded system usually performs a specialized operation and does the same repeatedly. For example: A pager always functions as a pager.

Tightly constrained − All computing systems have constraints on design metrics, but those on an embedded system can be especially tight. Design metrics is a measure of an implementation's features such as its cost, size, power, and performance. It must be of a size to fit on a single chip, must perform fast enough to process data in real time and consume minimum power to extend battery life.

Reactive and Real time − Many embedded systems must continually react to changes in the system's environment and must compute certain results in real time without any delay. Consider an example of a car cruise controller; it continually monitors and reacts to speed and brake sensors. It must compute acceleration or de-accelerations repeatedly within a limited time; a delayed computation can result in failure to control of the car.

Microprocessors based − It must be microprocessor or microcontroller based.

Memory − It must have a memory, as its software usually embeds in ROM. It does not need any secondary memories in the computer.

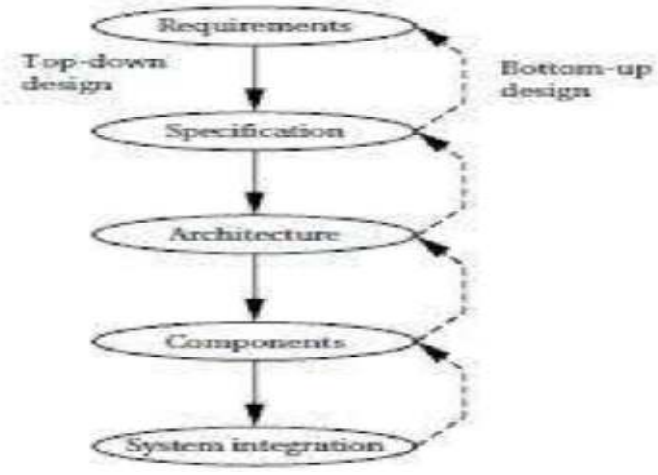
Connected − It must have connected peripherals to connect input and output devices.

HW-SW systems − Software is used for more features and flexibility. Hardware is used for performance and security.

Advantages

- Easily Customizable
- Low power consumption
- Low cost
- Enhanced

Embedded system Design flow steps:
There are different steps involved in Embedded system design flow. These steps depend on the design methodology. Design methodology is important for optimizing performance, and developing computer aided design tools.
. They are requirements gathering, specification formulation, architecture design, building of components, and system integration.



The major goals of the design to be considered are :

- Manufacturing cost;
- Performance (both overall speed and deadlines) and
- Power consumption.

The tasks which need to be performed at each step are the following.

● We must analyze the design at each step to determine how we can meet the specifications.

● We must then refine the design to add details.

● We must verify the design to ensure that it still meets all system goals, such as cost, speed, and so on.

Requirements

Informal descriptions gathered from the customer are known as requirements. The requirements are refined into a specification to begin the designing of the system architecture. Requirements can be functional or non-functional requirements. Functional requirements need output as a function of input. Non-functional requirements includes performance, cost, physical size, weight, and power consumption. Performance may be a combination of soft performance metrics such as approximate time to perform a user-level function and hard deadlines by which a particular operation must be completed. Cost includes the manufacturing, nonrecurring engineering(NRE) and other costs of designing the system. Physical size and weight are the physical aspects of the final system. These can vary greatly depending upon the application. Power consumption can be specified in the requirements stage in terms of battery life.

Specification

Requirements gathered is refined into a specification. Specification serves as the contract between the customers and the architects. Specification is essential to create working systems with a minimum of designer effort. It must be specific, understandable and accurately reflect the customer's requirements.

Architecture Design

The specification describes only the functions of the system. Implementation of the system is described by the Architecture. The architecture is a plan for the overall structure of the system. It will be used later to design the components.The architecture will be illustrated using block diagrams as shown below.

System integration

After the components are built, they are integrated. Bugs are typically found during the system integration. Good planning can help us to find the bugs quickly. By debugging a few modules at a time, simple bugs can be uncovered. By fixing the simple bugs early, more complex or obscure bugs can be uncovered. System integration is difficult because it usually uncovers problems. The debugging facilities for embedded systems are usually much more limited than the desktop systems. Careful attention is needed to

insert appropriate debugging facilities during design which can help to ease system integration problems.

System on Chip:-

SoC acronym for system on chip is an **IC** which integrates all the components into a single chip. It may contain analog, digital, mixed signal and other radio frequency functions all lying on a single chip substrate. Today, SoCs are very common in electronics industry due to its low power consumption. Also, **embedded system** applications make great use of SoCs.

SoCs consists of:

- Control Unit: In SoCs, the major control units are **microprocessors**, **microcontrollers**, digital signal processors etc.
- **Memory** Blocks: ROM, RAM. Flash memory and EEPROM are the basic memory units inside a SoC chip.
- Timing Units: **Oscillators** and **PLLs** are the timing units of the System on chip.
- Other peripherals of the SoCs are counter timers, real-time timers and power on reset generators.
- Analog interfaces, external interfaces, **voltage regulators** and power management units form the basic interfaces of the SoCs.

Advantages of SoC

- Low power.
- Low cost.
- High reliability.
- Small form factor.
- High integration levels.
- Fast operation.
- Greater design.
- Small size.

Disadvantages of SoC

- Fabrication cost.
- Increased complexity.
- Time to market demands.

- More verification.
  - Very-large-scale integration
    Very-large-scale integration (**VLSI**) is the process of creating an integrated circuit (IC) by combining thousands of transistors into a single chip. **VLSI** began in the 1970s when complex semiconductor and communication technologies were being developed. The microprocessor is a **VLSI** device.

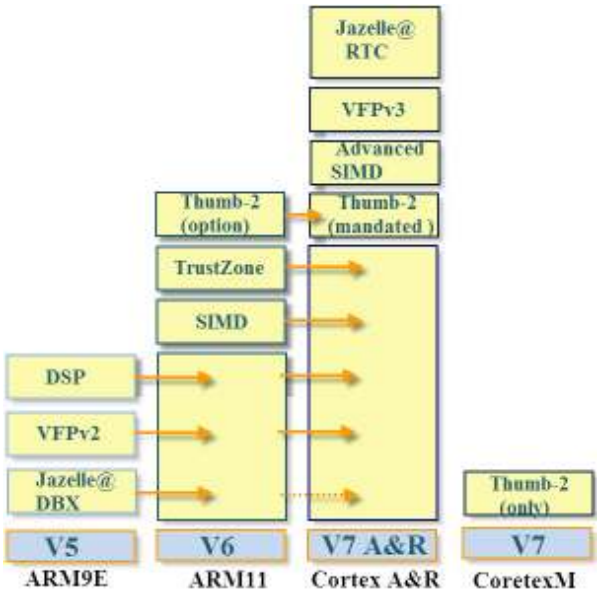
advantages of VLSI

The increase in density happens through multiple developments. Some of which would be a reduction in size, management in power consumption among others,

- Reduces the size of circuits
- Reduces the effective cost of the devices
- Increases the operating speed of circuits
- Requires less power than discrete components
- Higher reliability
- Occupies a relatively smaller area.

ARM Microcontroller

The ARM stands for Advanced RISC machine and it is a 32-bit reduced instructions set computer (RISC) microcontroller. It was first introduced by the Acron computers' organization in 1987. The ARM is a family of the microcontroller developed by the different manufacturers such as ST microelectronics, Motorola and so on. The ARM microcontroller architecture come with a few different versions such as ARMv1, ARMv2 etc



History of ARM

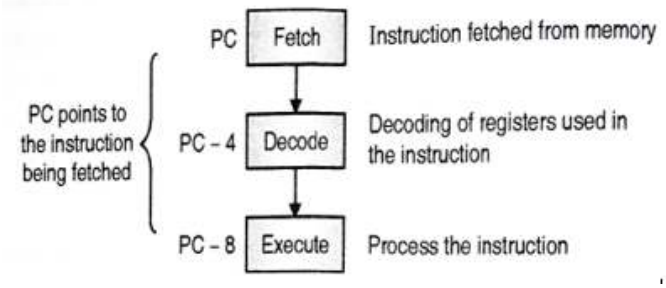
Advanced RISC machine (ARM) is the first reduced instruction set computer (RISC) processor for commercial use, which is currently being developed by ARM Holdings. The history of ARM processor dates back to 1983 in England when Acorn Computers Ltd officially launched an Acorn RISC Management project after being inspired to design its own processor by Berkeley RISC, one of the high-impact projects under ARPA's (Advanced Research Projects Agency, now converted to DARPA) VLSI project, dealing with RISC-based microprocessor design led by David Patterson who coined the term 'RISC.' As the name suggests, it does not mean that the processors with less than 100 instructions are qualified to RISC category, but instead they should have an highly optimised instruction set.

ARM in the beginning was known as Acorn RISC machine. With VLSI Technology Inc. as its silicon partner, ARM came up with ARM1, the first ARM silicon on April 26, 1985, which was used as a second processor to the BBC Micro to develop the simulation software to finish work on the support chips (VIDC, IOC and MEMC) and to increase the operating speed of the CAD software used in development of ARM2. Apple, whilst developing an entirely new computing platform for its Newton, a personal digital assistant, found that only Acorn RISC machine was close to the requirements needed for implementation, but since ARM had no integral memory management unit, Apple collaborated with Acorn to develop ARM.

ARM pipeline:-

The Process of fetching the next instruction while the current instruction is being executed is called as "pipelining". Pipelining is supported by the processor to increase the speed of program execution. Increases throughput. Several operations take place simultaneously, rather than serially in pipelining. The Pipeline has three stages fetch, decode and execute as

Shown below



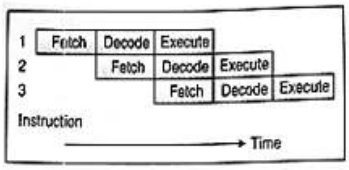
The three stages used in the pipeline are:

(i) Fetch : In this stage the ARM processor fetches the instruction from the memory.

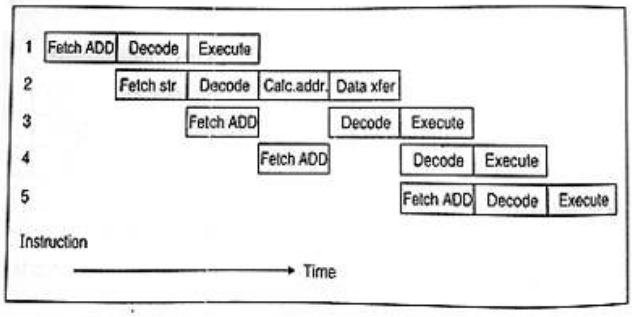
(ii) Decode : In this stage recognizes the instruction that is to be executed.

(iii) Execute 2 In this stage the processor processes the instruction and writes the result back to desired register.

If these three stages of execution are overlapped, we will achieve higher speed of execution. Such pipeline exists in version 7 of ARM processor. Once the pipeline is filled, each instructions require s one cycle to complete execution. Below fig shows three staged pipelined instruction.



(a) Single cycle instruction execution for a 3-stage pipeline in ARM



In first cycle, the processor fetches instruction 1 from the memory In the second cycle the processor fetches instruction 2 from the memory and decodes instruction 1. In the third cycle the processor fetches instruction 3 from memory, decodes instruction 2 and executes instruction 1. In the fourth cycle the processor fetches instruction 4, decodes instruction 3 and executes instruction 2. The pipeline thus executes an instruction in three cycles i.e. it delivers a throughput equal to one instruction per cycle.

In case of a multi-cycle instruction as shown 1n Fig. 9.10.2(b), instruction 2 (i. e. STR of the store instruction) requires 4 clock cycles and hence the pipeline stalls for one clock pulse. The first instruction completes execution in the third clock pulse, while the second instruction instead of completing execution in fourth clock pulse completes the same in fifth clock pulse. Thereafter every instruction completes execution in one clock pulse as seen in this figure.

The amount of work done at each stage can be reduced by increasing the number of stages in the pipeline. To improve the performance, the processor then can be operated at higher operating frequency.

As more number of cycles are required to fill the pipeline, the system latency also increases. The data dependency between the stages can also be increased as the stages of pipeline increase. So the instructions need to be schedule while writing code to decrease data dependency.

A five-stage (five clock cycle) ARM state pipeline is used, consisting of Fetch, Decode, Execute, Memory, and Write back stages. The program counter (PC) points to the instruction being fetched rather than to the instruction being executed.

During normal operation:

- one instruction is being fetched from memory
- the previous instruction is being decoded
- the instruction before that is being executed
- the instruction before that is performing data accesses (if applicable)
- the instruction before that is writing its data back to the register bank.

A six-stage (six clock cycle) pipeline is used in Jazelle state, consisting of Fetch, Jazelle/Decode (two clock cycles), Execute, Memory, and Write back stages.

1. The six stages are as follows:

- Fetch instruction (FI):
- Decode instruction ((DI):
- Calculate operand (CO):
- Fetch operands (FO):
- Execute Instruction (EI):
- Write operand (WO):

  Fetch instruction: Instructions are fetched from the memory into a temporary buffer before it gets executed.

  Decode instruction: The instruction is decoded by the CPU so that the necessary op codes and operands can be determined.

  Calculate operand: Based on the addressing scheme used, either operands are directly provided in the instruction or the effective address has to be calculated.

  Fetch Operand: Once the address is calculated, the operands need to be fetched from the address that was calculated. This is done in this phase.

Execute Instruction: The instruction can now be executed.

Write operand: Once the instruction is executed, the result from the execution needs to be stored or written back in the memory.

An Instruction Set Architecture (ISA) is part of the abstract model of a computer. It defines how software controls the CPU.

The Arm ISA family allows developers to write software and firmware that conforms to the Arm specifications, secure in the knowledge that any Arm-based processor will execute it in the same way. This is the foundation of the Arm portability and compatibility promise.

Arm Instruction Set Architecture

The Arm architecture supports three instruction sets: A64, A32 and T32.

- The A64 and A32 instruction sets have fixed instruction lengths of 32-bits.
- The T32 instruction set was introduced as a supplementary set of 16-bit instructions that supported improved code density for user code. Over time, T32 evolved into a 16-bit and 32-bit mixed-length instruction set. As a result, the compiler can balance performance and code size trade-off in a single instruction set.

The ARM Registers has following characteristics:

ARM has 37 registers in total, all of which are 32-bits long. and it consist of

1 dedicated program counter

1 dedicated current program status register

5 dedicated saved program status registers

30 general purpose registers.

When the processor is executing in ARM state:All instructions are 32 bits in length

All instructions must be word aligned

Therefore the PC value is stored in bits [31:2] with bits [1:0] equal to zero (as instruction cannot be half word or byte aligned).

R14 is used as the subroutine link register (LR) and stores the return address when Branch with Link operations are performed, calculated from the PC.

Data-processing instructions ARM has 16 data-processing instructions. Most data-processing instructions take two source operands, though Move and Move Not take only one. The

compare and test instructions only update the condition flags. Other data-processing instructions store a

result to a register and optionally update the condition flag.

CMP, CMN, TST and TEQ always update the condition code flags. The assembler automatically sets the S bit in

the instruction for them, and the corresponding instruction with the S bit.

Instruction encoding

<opcode1>{<cond>}{S} <Rd>, <shifter_operand>

<opcode1> := MOV | MVN

<opcode2>{<cond>} <Rn>, <shifter_operand>

<opcode2> := CMP | CMN | TST | TEQ

<opcode3>{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

<opcode3> := ADD | SUB | RSB | ADC | SBC | RSC | AND | BIC | EOR | ORR

I bit Distinguishes between the immediate and register forms of <shifter_operand>.

S bit Signifies that the instruction updates the condition codes.

Rn Specifies the first source operand register.

Rd Specifies the destination register.

shifter_operand Specifies the second source operand. See *Addressing Mode 1 - Data-processing operands* on page A5-2 for details of the shifter operands.

List of data-processing instructions

ADC Add with Carry.

ADD Add.

AND Logical AND.

BIC Logical Bit Clear.

CMN Compare Negative.

CMP Compare.

EOR Logical EOR.

MOV Move.

MVN Move Not.

ORR Logical OR.

RSB Reverse Subtract.

RSC Reverse Subtract with Carry.

SBC Subtract with Carry.

SUB Subtract.

TEQ Test Equivalence.

Multiply instructions

ARM has several classes of Multiply instruction:

Normal 32-bit x 32-bit, bottom 32-bit result

Long 32-bit x 32-bit, 64-bit result

Halfword 16-bit x 16-bit, 32-bit result

Word ∞ halfword 32-bit x 16-bit, top 32-bit result

Most significant word

32-bit x 32-bit, top 32-bit result

Dual halfword dual 16-bit x 16-bit, 32-bit result.

All Multiply instructions take two register operands as the input to the multiplier. The ARM processor does not directly support a multiply-by-constant instruction because of the efficiency of shift and add, or shift and reverse subtract instructions.

Normal multiply

There are two 32-bit x 32-bit Multiply instructions that produce bottom 32-bit results:

MUL Multiplies the values of two registers together, truncates the result to 32 bits, and stores the result in a third register.

MLA Multiplies the values of two registers together, adds the value of a third register, truncates the result to 32 bits, and stores the result in a fourth register. This can be used to perform multiply-accumulate operations.

Both Normal Multiply instructions can optionally set the N (Negative) and Z (Zero) condition code flags. No distinction is made between signed and unsigned variants. Only the least significant 32 bits of the result are stored in the destination register, and the sign of the operands does not affect this value.

SMMUL Multiplies the 32-bit values of two registers together, and stores the top 32 bits of the signed 64-bit result in a third register.

SMMLA Multiplies the 32-bit values of two registers together, extracts the top 32 bits, adds the 32-bit value from a third register, and stores the signed 32-bit result in a fourth register.

SMMLS Multiplies the 32-bit value of two registers together, extracts the top 32 bits, subtracts this from a 32-bit value from a third register, and stores the signed 32-bit result in a fourth register.

Examples

MUL R4, R2, R1 ; Set R4 to value of R2 multiplied by R1

MULS R4, R2, R1 ; R4 = R2 x R1, set N and Z flags

MLA R7, R8, R9, R3 ; R7 = R8 x R9 + R3

SMULL R4, R8, R2, R3 ; R4 = bits 0 to 31 of R2 x R3 ; R8 = bits 32 to 63 of R2 x R3

UMULL R6, R8, R0, R1 ; R8, R6 = R0 x R1

UMLAL R5, R8, R0, R1 ; R8, R5 = R0 x R1 + R8, R5

Load and store instructions

The ARM architecture supports two broad types of instruction which load or store the value of a single register, or a pair of registers, from or to memory:

• The first type can load or store a 32-bit word or an 8-bit unsigned byte.

• The second type can load or store a 16-bit unsigned halfword, and can load and sign extend a 16-bit halfword or an 8-bit byte. In ARMv5TE and above, it can also load or store a pair of 32-bit words.

Load and Store Multiple instructions

Load Multiple instructions load a subset, or possibly all, of the general-purpose registers from memory.

Store Multiple instructions store a subset, or possibly all, of the general-purpose registers to memory.

Load and Store Multiple instructions have a single instruction format:

LDM{<cond>}<addressing_mode> Rn{!}, <registers>{^}

STM{<cond>}<addressing_mode> Rn{!}, <registers>{^}

where: <addressing_mode> = IA | IB | DA | DB | FD | FA | ED | EA

simple

Branch or Branch with Link instruction:

_ B{condition} <address>

_ BL{condition} <address>

_ Bits [27:25] identify this as a B or BL instruction – they have values 101
only for these instructions

Conditional execution in ARM state

To execute ARM instructions conditionally you can either append a two letter suffix to the mnemonic, or you can use a conditional branch instruction. Almost all ARM instructions can be executed conditionally on the value of the condition flags in the APSR. You can either add a condition code suffix to the instruction or you can conditionally skip over the instruction using a conditional branch instruction. Using conditional branch instructions to control the flow of execution can be more efficient when a series of instructions depend on the same condition.

Example conditional instructions to control execution

```
    LSLEQ r0, r0, #24
  ADDEQ r0, r0, #2
  ;…
```

Example conditional branch to control execution

```
    BNE over
  LSL r0, r0, #24
  ADD r0, r0, #2
over
  ;…
```

The Thumb instruction set consists of 16-bit instructions that act as a compact shorthand for a subset of the 32-bit instructions of the standard ARM. Every Thumb instruction could instead be executed via the equivalent 32-bit ARM instruction. However, not all ARM instructions are available in the Thumb subset; for example, there's no way to access status or coprocessor registers. Also, some functions that can be accomplished in a single ARM instruction can only be simulated with a sequence of Thumb instructions. The Thumb instruction set provides most of the functionality required in a typical application. Arithmetic and logical operations, load/store data movements, and conditional and unconditional branches are supported. Based upon the available instruction set, any code written in C could be executed successfully in Thumb state. However, device drivers and exception handlers must often be written at least partly in ARM state.

they are identical to regular ARM instructions but generally have limitations, including that they:

Access only the bottom eight registers

Reuse a register as both a source and destination

Support shorter immediates

Lack conditional execution

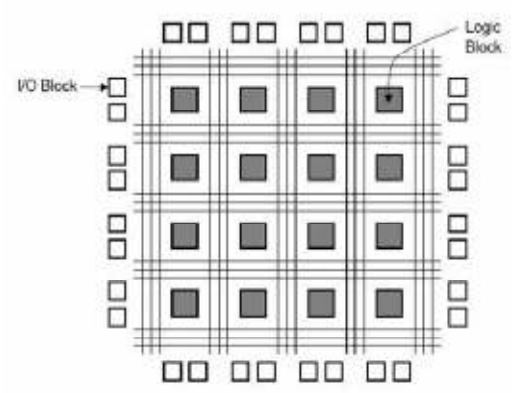
Always write the status flags

FPGA

The full form of **FPGA** is "**Field Programmable Gate Array**". It contains ten thousand to more than a million logic gates with programmable interconnection. Programmable interconnections are available for users or designers to perform given functions easily. A typical model FPGA chip is shown in the given figure. There are I/O blocks, which are designed and numbered according to function. For each module of logic level composition, there are **CLB's (Configurable Logic Blocks)**.

CLB performs the logic operation given to the module. The inter connection between CLB and I/O blocks are made with the help of horizontal routing channels, vertical routing channels and PSM (Programmable Multiplexers).

The number of CLB it contains only decides the complexity of FPGA. The functionality of CLB's and PSM are designed by VHDL or any other hardware descriptive language. After programming, CLB and PSM are placed on chip and connected with each other with routing channels.



Advantages

It requires very small time; starting from design process to functional chip.

No physical manufacturing steps are involved in it.

The only disadvantage is, it is costly than other styles.

FPGA Architecture

FPGAs are prefabricated silicon chips that can be programmed electrically to implement digital designs. The first static memory based FPGA called SRAM is used for configuring both logic and interconnection using a stream of configuration bits. Today's modern EPGA contains approximately 3,30,000 logic blocks and around 1,100 inputs and outputs.

The FPGA Architecture consists of three major components

- Programmable Logic Blocks, which implement logic functions
- Programmable Routing (interconnects), which implements functions
- I/O blocks, which are used to make off-chip connections

Programmable Logic Blocks

The programmable logic block provides basic computation and storage elements used in digital systems. A basic logic element consists of programmable combinational logic, a flip-flop, and some fast carry logic to reduce area and delay cost.

Programmable Routing

The programmable routing establishes a connection between logic blocks and Input/Output blocks to complete a user-defined design unit.

It consists of multiplexers pass transistors and tri-state buffers. Pass transistors and multiplexers are used in a logic cluster to connect the logic elements.

Programmable I/O

The programmable I/O pads are used to interface the logic blocks and routing architecture to the external components. The I/O pad and the surrounding logic circuit form as an I/O cell.

These cells consume a large portion of the FPGA's area. And the design of I/O programmable blocks is complex, as there are great differences in the supply voltage and reference voltage.

The selection of standards is important in I/O architecture design. Supporting a large number of standards can increase the silicon chip area required for I/O cells.

With advancement, the basic FPGA Architecture has developed through the addition of more specialized programmable function blocks.

The special functional blocks like ALUs, block RAM, multiplexers, DSP-48, and microprocessors have been added to the FPGA, due to the frequency of the need for such resources for applications.

MODULE-2

Device And Device Drivers :

A device driver is a program that controls a particular type of device that is attached to your computer. There are device drivers for printers, displays, CD-ROM readers, diskette drives, and so on.

A driver provides a software interface to hardware devices, enabling operating systems and other computer programs to access hardware functions without needing to know precise details of the hardware being used.

A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware connects. When a calling program invokes a routine in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware-dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.

Purpose of a Device Driver

The purpose of a device driver is to handle requests made by the kernel with regard to a particular type of device. There is a well-defined and consistent interface for the kernel to make these requests. By isolating device-specific code in device drivers and by having a consistent interface to the kernel, adding a new device is easier.

Types of Device Drivers

A device driver is a software module that resides within the kernel and is the software interface to a hardware device or devices. A hardware device is a peripheral, such as a disk controller, tape controller, or network controller device. In general, there is one device driver for each type of hardware device. Device drivers can be classified as:

- Block device drivers
- Character device drivers (including terminal drivers)
- Network device drivers

A device driver is a program that controls a particular type of device that is attached to your computer. There are device drivers for printers, displays, CD-ROM readers, diskette drives, and so on.

A driver provides a software interface to hardware devices, enabling operating

systems and other computer programs to access hardware functions without needing to know precise details of the hardware being used.

A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware connects. When a calling program invokes a routine in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware-dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.

Purpose of a Device Driver

The purpose of a device driver is to handle requests made by the kernel with regard to a particular type of device. There is a well-defined and consistent interface for the kernel to make these requests. By isolating device-specific code in device drivers and by having a consistent interface to the kernel, adding a new device is easier.

Types of Device Drivers

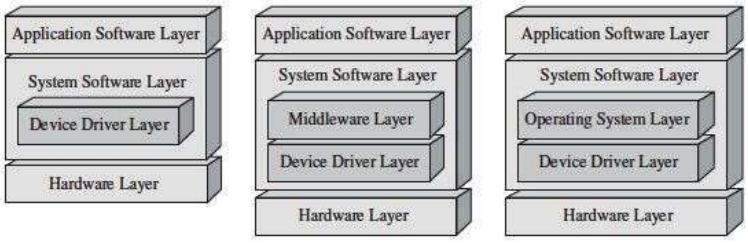
A device driver is a software module that resides within the kernel and is the software interface to a hardware device or devices. A hardware device is a peripheral, such as a disk controller, tape controller, or network controller device. In general, there is one device driver for each type of hardware device. Device drivers can be classified as:

- Block device drivers
- Character device drivers (including terminal drivers)
- Network device drivers

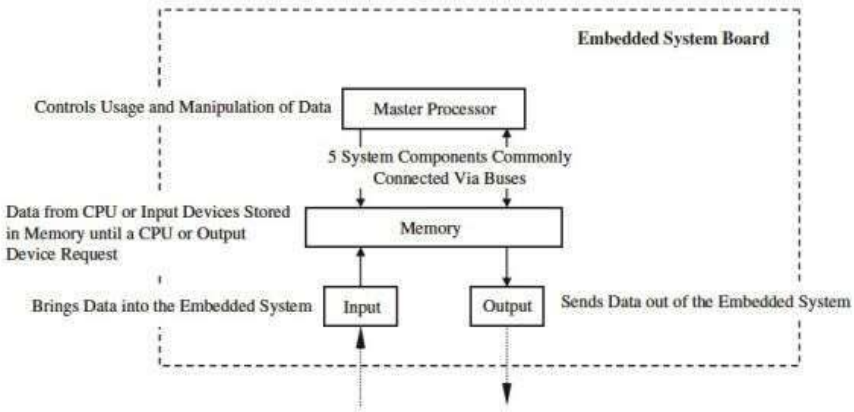
Most embedded hardware requires some type of software initialization and management. The software that directly interfaces with and controls this hardware is called a device driver. All embedded systems that require software have, at the very least, device driver software in their system software layer. Device drivers are the software libraries that initialize the hardware and manage access to the hardware by higher layers of software.

Device drivers are the liaison between the hardware and the operating system, middleware, and application layers. Different types of hardware will have different device driver requirements that need to be met.

Even the same type of hardware, such as Flash memory, that are created by different manufacturers can require substantially different device driver software libraries to support within the embedded device.



Embedded Systems Model and Device Drivers.

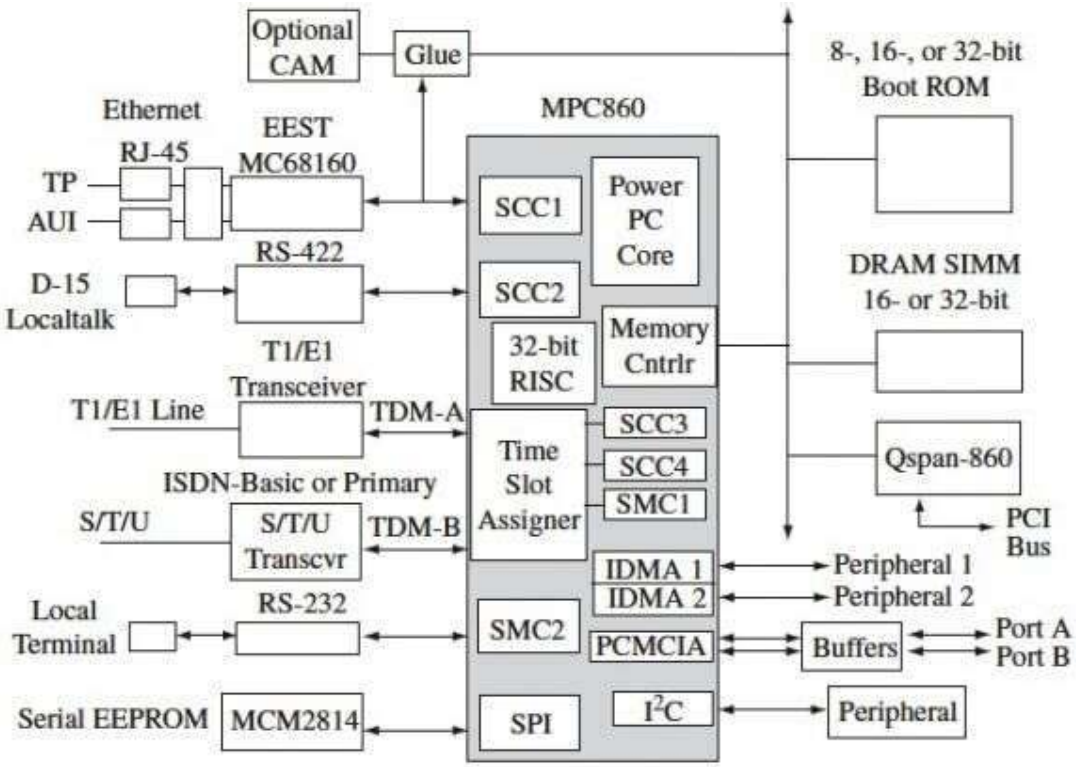


Embedded System Board Organization.[1]. Based upon the von Neumann architecture model (also referred to as the Princeton architecture).

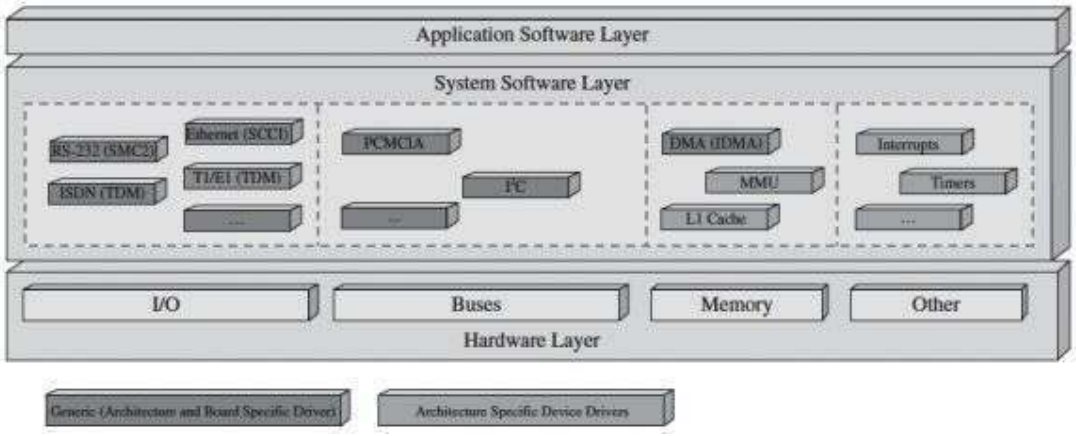
Specifically, this can include drivers for the master processor architecture-specific functionality, memory and memory management drivers, bus initialization and transaction drivers, and I/O (input/output) initialization and control drivers (such as for networking, graphics, input devices, storage devices, or debugging I/O) both at the board and master CPU level.

Device drivers are typically considered either architecture-specific or generic. A device driver that is architecture-specific manages the hardware that is integrated into the master processor (the architecture). Examples of architecture-specific drivers that initialize and enable components within a master processor include on-chip memory, integrated memory managers (memory management units (MMUs)), and floating-point hardware. A device driver that is generic manages hardware that is located on the board and not integrated onto the master processor. In a generic driver, there are typically architecture-specific portions of source code, because the master processor is the central control unit and to gain access to anything on the board usually means going through the master processor. However, the generic driver also manages board hardware that is not specific to that particular processor, which means that a generic driver can be configured to run on a variety of architectures that contain the related board hardware for which the driver is written. Generic drivers include code that initializes and manages access to the remaining major components of the board,

including board buses (I2C, PCI, PCMCIA, etc.), off-chip memory (controllers, level 2+ cache, Flash, etc.), and off-chip I/O (Ethernet, RS-232, display, mouse, etc.).



MPC860 Hardware Block Diagram. Freescale Semiconductor, Inc. Used by permission.



. MPC860 Architecture-Specific Device Driver System Stack. © Freescale Semiconductor, Inc. Used by permission.

Figure shows a hardware block diagram of an MPC860-based board and shows a systems diagram that includes examples of MPC860 processor-specific device drivers, as well as generic device drivers.

Regardless of the type of device driver or the hardware it manages, all device drivers are generally made up of all or some combination of the following functions:

Hardware Startup: initialization of the hardware upon PowerON or reset.

Hardware Shutdown: configuring hardware into its PowerOFF state.

Hardware Disable: allowing other software to disable hardware on-the-fly.

Hardware Enable: allowing other software to enable hardware on-the-fly.

Hardware Acquire: allowing other software to gain singular (locking) access to hardware.

Hardware Release: allowing other software to free (unlock) hardware.

Hardware Read: allowing other software to read data from hardware.

Hardware Write: allowing other software to write data to hardware.

Hardware Install: allowing other software to install new hardware on-the-fly.

Hardware Uninstall: allowing other software to remove installed hardware on-the-fly.

Hardware Mapping: allowing for address mapping to and from hardware storage devices when reading, writing, and/or deleting data.

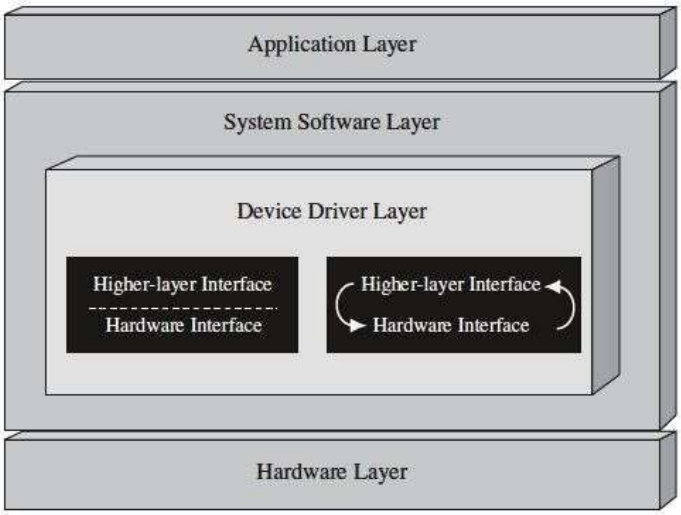
Hardware Un mapping: allowing for un mapping (removing) blocks of data from hardware storage devices.

Of course, device drivers may have additional functions, but some or all of the functions shown above are what device drivers inherently have in common. These functions are based upon the software's implicit perception of hardware, which is that hardware is in one of three states at any given time—inactive, busy, or finished. Hardware in the inactive state is interpreted as being either disconnected (thus the need for an install function), without power (hence the need for an initialization routine), or disabled (thus the need for an enable routine). The busy and finished states are active hardware states, as opposed to inactive; thus the need for uninstall, shutdown, and/or disable functionality. Hardware that is in a busy state is actively processing some type of data and is not idle, and thus may require some type of release mechanism. Hardware that is in the finished state is in an idle state, which then allows for acquisition, read, or write requests,

for example. Again, device drivers may have all or some of these functions, and can integrate some of these functions into single larger functions. Each of these driver functions typically has code that interfaces directly to the hardware and code that interfaces to higher layers of software. In some cases, the distinction between these layers is clear, while in other drivers, the code is tightly integrated

On a final note, depending on the master processor, different types of software can execute in different modes, the most common being supervisory and user modes. These modes essentially differ in terms of what system components the software is allowed access to, with software running in

supervisory mode having more access (privileges) than software running in user mode. Device driver code typically runs in supervisory mode.



Driver Code Layers. I/O Devices :

Input and output devices allow the computer system to interact with the outside world by moving data *into* and *out of* the system. An *input device* is used to bring data into the system. Some input devices are:

Keyboard

Mouse

Microphone

Bar code reader

Graphics tablet

An *output device* is used to send data out of the system. Some output devices are:

Monitor

Printer

Speaker

Input/output devices are usually called I/O devices. They are directly connected to an electronic module inside the systems unit called a **device controller**. For example, the speakers of a multimedia computer system are directly connected to a device controller called an audio card (such as a Sound blaster), which in turn is connected to the rest of the system.

Sometimes secondary memory devices like the hard disk are called I/O devices (because they move data in and out of main memory.) What counts as an I/O device depends on context. To a user, an I/O device is something outside of the system box. To a programmer, everything outside of the processor and main memory looks like an I/O devices. To an engineer working on the design of a processor, everything outside of the processor is an I/O device.

A computer that is dedicated to running a program that controls another device is an **embedded system**. An embedded system is usually embedded inside the device it controls. Usually they run just one program that is permanently kept in a special kind of main memory called ROM (for Read Only Memory). More processor chips are sold per year for embedded systems than for all other purposes.

Serial Peripheral interface :

The **Serial Peripheral Interface** (**SPI**) is a synchronous serial communication interface specification used for short-distance communication, primarily in embedded systems. The interface was developed by Motorola in the mid-1980s and has become a *de facto* standard. Typical applications include Secure Digital cards and liquid crystal displays.

SPI devices communicate in full duplex mode using a master-slave architecture with a single master. The master device originates the frame for reading and writing. Multiple slave-devices are supported through selection with individual slave select (SS), sometimes called chip select (CS), lines.

The SPI bus specifies four logic signals:

SCLK: Serial Clock (output from master)

MOSI: Master Output Slave Input, or Master Out Slave In (data output from master)

MISO: Master Input Slave Output, or Master In Slave Out (data output from slave)

SS: Slave Select (often active low, output from master)

While the above pin names are the most popular, in the past alternative pin-naming conventions were sometimes used, and so SPI port pin-names for older IC products may differ from those depicted in these illustrations:

Serial Clock:

SCLK: SCK

Master Output → Slave Input (MOSI):

SIMO, MTSR - correspond to MOSI on both master and slave devices, connects to each other

SDI, DI, DIN, SI - on slave devices; connects to MOSI on master, or to below connections

SDO, DO, DOUT, SO - on master devices; connects to MOSI on slave, or to above connections

Master Input ← Slave Output (MISO):

SOMI, MRST - correspond to MISO on both master and slave devices, connects to each other

SDO, DO, DOUT, SO - on slave devices; connects to MISO on master, or to below connections

SDI, DI, DIN, SI - on master devices; connects to MISO on slave, or to above connections

Slave Select:

SS: $\overline{\text{SS}}$, SSEL, CS, $\overline{\text{CS}}$, CE, nSS, /SS, SS#

In other words, MOSI (or SDO on a master) connects to MOSI (or SDI on a slave). MISO (or SDI on a master) connects to MISO (or SDO on a slave). Slave Select is the same functionality as chip

select and is used instead of an addressing concept. Pin names are always capitalized as in Slave Select, Serial Clock, and Master Output Slave Input.

IIC :

I²C (**Inter-Integrated Circuit**), pronounced *I-squared-C*, is a synchronous, multi-master, multi-slave, packet switched, single-ended, serial computer bus invented in 1982 by Philips Semiconductor (now NXP Semiconductors). It is widely used for attaching lower-speed peripheral ICs to processors and microcontrollers in short-distance, intra-board communication. Alternatively, I²C is spelled **I2C** (pronounced I-two-C) or **IIC** (pronounced I-I-C).

I²C is appropriate for peripherals where simplicity and low manufacturing cost are more important than speed. Common applications of the I²C bus are:

Describing connectable devices via small ROM configuration tables to enable "plug and play" operation, such as Serial Presence Detect (SPD) EEPROMs on dual in-line memory modules (DIMMs), and Extended Display Identification Data (EDID) for monitors via VGA, DVI and HDMI connectors. System management for PC systems via SMBus; SMBus pins are allocated in both Conventional PCI and PCI Express connectors.

Accessing real-time clocks and NVRAM chips that keep user settings.

Accessing low-speed DACs and ADCs.

Changing contrast, hue, and color balance settings in monitors (via Display Data Channel).

Changing sound volume in intelligent speakers.

Controlling small (e.g. feature phone) OLED or LCD displays.

Reading hardware monitors and diagnostic sensors, e.g. a fan's speed.

Turning on and turning off the power supply of system components.[4]

A particular strength of I²C is the capability of a microcontroller to control a network of device chips with just two general-purpose I/O pins and software. Many other bus technologies used in similar applications, such as Serial Peripheral Interface Bus (SPI), require more pins and signals to connect multiple devices.

RS 232C :

RS-232C is the interface that your computer uses to talk to and exchange data with your modem and other serial devices. RS-232C is the interface between your Communication networks and other communication networks.

Somewhere in your PC, typically on a Universal Asynchronous Receiver/Transmitter (UART) chip on your motherboard, the data from your computer is transmitted to an internal or external modem (or other serial device) from its Data Terminal Equipment (DTE) interface. Since data in your computer flows along parallel circuits and serial devices can handle only one bit at a time, the UART chip converts the groups of bits in parallel to a serial stream of bits.

As your PC's DTE agent, it also communicates with the modem or other serial device, which, in accordance with the RS-232C standard, has a complementary interface called the Data Communications Equipment (DCE) interface. RTS/CTS is the way the DTE indicates that it is ready to transmit data and the way the DCW indicates that it is ready to accept data

RS232C, a standard interface approved by the Electronic Industries Alliance (EIA) for connecting serial devices. In 1987, the EIA released a new version of the standard and changed the name to EIA-232-D. And in 1991, the EIA teamed up with Telecommunications Industry association (TIA) and issued a new version of the standard called EIA/TIA-232-E. Many people, however, still refer to the standard as RS-232C, or just RS-232.

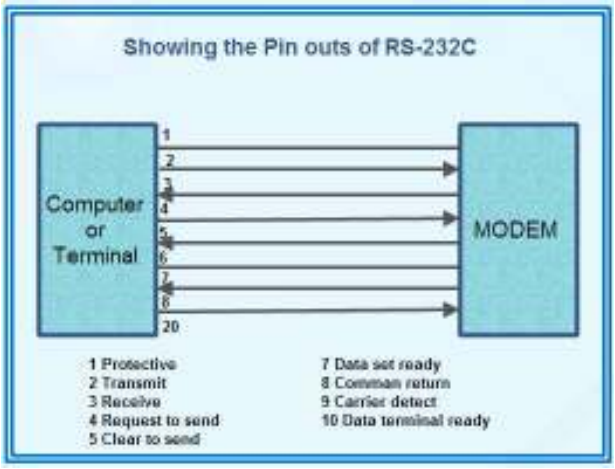
Almost all modems conform to the EIA-232 standard and most personal computers have an EIA-232 port for connecting a modem or other device. In addition to modems, many display screens, mice, and serial printers are designed to connect to a EIA-232 port. In EIA-232 parlance, the device that connects to the interface is called a Data Communications Equipment (DCE) and the device to which it connects (e.g., the computer) is called a Data Terminal Equipment (DTE).

The essential feature ofRS-232 is that the signals are carried as single voltages referred to a common ground on pin 7. In its simplest form, the RS-232:C interface consis;9'of only two wires for data and ground. The ground is the absolute voltage reference for all the interface circuitry, the point in the circuit from which all voltages are measured. Data on pin 2 of the DTE is transmitted, while the same data on pin 2 of a DCE (modem) is received data as shown in Figure.

Data is transmitted and received on pins 2 and 3, respectively. Data Set Ready (DSR) is an indication from the Data set (the modem or DSU/CSU) that it is on. Similarly, DTR indicates that the DTE is on. Carrier Detect (CD) indicates that carrier for the transmission data is on.

Pins 4 and 5 carry the Request to Send (RTS) and Clear to Send (CTS) signals. In most situations, RTS and CTS are constantly on the communication session. However, where the DTE is connected to a multipoint line, RTS is used to turn the carrier on the modem on and off. On a multipoint line, it is imperative that only one station is transmitting at a time. When a station wants to transmit, it raises RTS. The modem turns on carrier, typically waits a few milliseconds for carrier to stabilize, and raises CTS.

The DTE transmits when it sees CTS up. When the station has finished its transmission, it drops RTS and the modem drops CTS and carrier together.



RS 422 :

RS-422, also known as **TIA/EIA-422**, is a technical standard originated by the Electronic Industries Alliance that specifies electrical characteristics of a digital signaling circuit. It was intended to replace the older RS-232C standard with a standard that offered much higher speed, better immunity from noise, and longer cable lengths. RS-422 systems can transmit data at rates as high as 10 Mbit/s, or may be sent on cables as long as 1,500 meters at lower rates. It is closely related to RS-423, which used the same signaling systems but on a different wiring arrangement.

RS-422 specifies differential signaling, with every data line paired with a dedicated return line. It is the voltage difference between these two lines that define the mark and space, rather than, as in RS-232, the difference in voltage between a data line and a local ground. As the ground voltage can differ at either end of the cable, this required RS-232 to use large +5 and -5 voltages. Moving to dedicated return lines and always defining ground in reference to the sender allowed RS-422 to use 0.4 V, allowing it to run at much higher speeds. RS-423 differed primarily in that it had a single return pin instead of one for each data pin.

RS-422 and RS-423 had originally planned to use the same DB25 connector as RS-232, but over time the number of required pins grew and the standards split out the definition into the RS-449 effort. This produced an unwieldy system and later returned to DB25 in the RS-530 standard.

RS 485 :

RS-485, also known as **TIA-485(-A)** or **EIA-485**, is a standard defining the electrical characteristics of drivers and receivers for use in serial communications systems. Electrical signaling is balanced, and multipoint systems are supported. The standard is jointly published by the Telecommunications Industry Association and Electronic Industries Alliance (TIA/EIA). Digital communications networks implementing the standard can be used effectively over long distances and in electrically noisy environments. Multiple receivers may be connected to such a network in a linear, multidrop bus. These characteristics make RS-485 useful in industrial control systems and similar applications.

RS-485 supports inexpensive local networks and multidrop communications links, using the same differential signaling over twisted pair as RS-422. It is generally accepted that RS-485 can be used with data rates up to 10 Mbit/s[a] or, at lower speeds, distances up to 1,200 m (4,000 ft).[2] As a rule of thumb, the speed in bit/s multiplied by the length in metres should not exceed 10^8. Thus a 50-meter cable should not signal faster than 2 Mbit/s.

In contrast to RS-422, which has a driver circuit which cannot be switched off, RS-485 drivers use three-state logic allowing individual transmitters to be deactivated. This allows RS-485 to implement linear bus topologies using only two wires. The equipment located along a set of RS-485 wires are interchangeably called nodes, stations or devices. The recommended arrangement of the wires is as a connected series of point-to-point (multi dropped) nodes, i.e. a line or bus, not a star, ring, or multiply connected network. Star and ring topologies are not recommended because of signal reflections or excessively low or high termination impedance. If a star configuration is unavoidable, special RS-485 repeaters are available which bi directionally listen for data on each span and then retransmit the data onto all other spans.

Ideally, the two ends of the cable will have a termination resistor connected across the two wires. Without termination resistors, signal reflections off the un terminated end of the cable can cause data corruption. Termination resistors also reduce electrical noise sensitivity due to the lower impedance,The value of each termination resistor should be equal to the cable characteristic impedance (typically, 120 ohms for twisted pairs). The termination also includes pull up and pull down resistors to establish fail-safe bias for each data wire for the case when the lines are not being driven by any device. This way, the lines will be biased to known voltages and nodes will not interpret the noise from un driven lines as actual data; without biasing resistors, the data lines float in such a way that electrical noise sensitivity is greatest when all device stations are silent or unpowered.

Universal Serial Bus :

A Universal Serial Bus (USB) is basically a newer port that is used as a common interface to connect several different types of devices such as keyboards, printers, media devices, cameras, scanners, and mice. It is designed for easy installation, faster transfer rates, higher quality cabling and hot swapping. It has conclusively replaced the bulkier and slower serial and parallel ports.

One of the greatest features of the USB is hot swapping. This feature allows a device to be removed or replaced without the past prerequisite of rebooting and interrupting the system. Older ports required that a PC be restarted when adding or removing a new device. Rebooting allowed the device to be reconfigured and prevented electrostatic discharge (ESD), an unwanted electrical current

capable of causing serious damage to sensitive electronic equipment such as integrated circuits. Hot swapping is fault tolerant, i.e. able to continue operating despite a hardware failure. However, care should be taken when hot swapping certain devices such as a camera; damage can occur to the port, camera or other devices if a single pin is accidently shorted.

Another USB feature is the use of direct current (DC). In fact, several devises use a USB power line to connect to DC current and do not transfer data. Example devices using a USB connector only for DC current include a set of speakers, an audio jack and power devices like a miniature refrigerator, coffee cup warmer or keyboard lamp.

USB Version 1 allowed for two speeds: 1.5 Mb/s (megabits per second) and 12 Mb/s, which work well for slow I/O devices. USB Version 2 allows up to 480 Mb/s and is backward compatible with slower USB devices. USB supports three.

IrDa :

Short for ***Infrared Data Association,*** a group of device manufacturers that developed a standard for transmitting data via infrared light waves. Increasingly, computers and other devices (such as printers) come with IrDA ports. This enables you to transfer data from one device to another without any cables. For example, if both your laptop computer and printer have IrDA ports, you can simply put your computer in front of the printer and output a document, without needing to connect the two with a cable.

IrDA ports support roughly the same transmission rates as traditional parallel ports. The only restrictions on their use is that the two devices must be within a few feet of each other and there must be a clear line of sight between them.

CAN Bus :

A Controller Area Network (CAN bus) is a robust vehicle bus standard designed to allow microcontrollers and devices to communicate with each other's applications without a host computer. It is a message-based protocol, designed originally for multiplex electrical wiring within automobiles to save on copper, but can also be used in many other contexts. For each device the data in a frame is transmitted sequentially but in such a way that if more than one device transmits at the same time the highest priority device is able to continue while the others back off. Frames are received by all devices, including by the transmitting device.

Development of the CAN bus started in 1983 at Robert Bosch GmbH. The protocol was officially released in 1986 at the Society of Automotive Engineers (SAE) conference in Detroit, Michigan. The first CAN controller chips were introduced by Intel in 1987, and shortly thereafter by Philips. Released in 1991, the Mercedes-Benz W140 was the first production vehicle to feature a CAN-based multiplex wiring system.[2][3]

Bosch published several versions of the CAN specification and the latest is CAN 2.0 published in 1991. This specification has two parts; part A is for the standard format with an 11-bit identifier, and part B is for the extended format with a 29-bit identifier. A CAN device that uses 11-bit identifiers is commonly called CAN 2.0A and a CAN device that uses 29-bit identifiers is commonly called CAN 2.0B.

In 1993, the International Organization for Standardization (ISO) released the CAN standard ISO 11898 which was later restructured into two parts; ISO 11898-1 which covers the data link layer, and ISO 11898-2 which covers the CAN physical layer for high-speed CAN. ISO 11898-3 was released later and covers the CAN physical layer for low-speed, fault-tolerant CAN. The physical layer standards ISO 11898-2 and ISO 11898-3 are not part of the Bosch CAN 2.0 specification. These standards may be purchased from the ISO.

Bosch is still active in extending the CAN standards. In 2012, Bosch released CAN FD 1.0 or CAN with Flexible Data-Rate. This specification uses a different frame format that allows a different data length as well as optionally switching to a faster bit rate after the arbitration is decided. CAN FD is compatible with existing CAN 2.0 networks so new CAN FD devices can coexist on the same network with existing CAN devices.[5]

CAN bus is one of five protocols used in the on-board diagnostics (OBD)-II vehicle diagnostics standard. The OBD-II standard has been mandatory for all cars and light trucks sold in the United States since 1996. The EOBD standard has been mandatory for all petrol vehicles sold in the European Union since 2001 and all diesel vehicles since 2004.

Bluetooth :

Bluetooth is a wireless technology standard used for exchanging data between fixed and mobile devices over short distances using short-wavelength UHF radio waves in the industrial, scientific and medical radio bands, from 2.400 to 2.485 GHz, and building personal area networks (PANs). It was originally conceived as a wireless alternative to RS-232 data cables.

Bluetooth is managed by the Bluetooth Special Interest Group (SIG), which has more than 35,000 member companies in the areas of telecommunication, computing, networking, and consumer electronics. The IEEE standardized Bluetooth as **IEEE 802.15.1**, but no longer maintains the standard. The Bluetooth SIG oversees development of the specification, manages the qualification program, and protects the trademarks. A manufacturer must meet Bluetooth SIG standards to market it as a Bluetooth device.[4] A network of patents apply to the technology, which are licensed to individual qualifying devices. As of 2009, Bluetooth integrated circuit chips ship approximately 920 million units annually.

Real Time Operating System :

A real-time operating system (RTOS) is an operating system (OS) intended to serve real-time applications that process data as it comes in, typically without buffer delays. Processing time requirements (including any OS delay) are measured in tenths of seconds or shorter increments of time. A real-time system is a time-bound system which has well-defined, fixed time constraints. Processing must be done within the defined constraints or the system will fail. They either are event-driven or time-sharing. Event-driven systems switch between tasks based on their priorities, while time-sharing systems switch the task based on clock interrupts. Most RTOSs use a pre-emptive scheduling algorithm.

A key characteristic of an RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application's task; the variability is 'jitter'.[1] A 'hard' real-time operating system has less jitter than a 'soft' real-time operating system. The chief design goal is not high throughput, but rather a guarantee of a soft or hard performance category. An RTOS that can usually or generally meet a deadline is a soft real-time OS, but if it can meet a deadline deterministically it is a hard real-time OS.

An RTOS has an advanced algorithm for scheduling. Scheduler flexibility enables a wider, computer-system orchestration of process priorities, but a real-time OS is more frequently dedicated to a narrow set of applications. Key factors in a real-time OS are minimal interrupt latency and minimal thread switching latency; a real-time OS is valued more for how quickly or how predictably it can respond than for the amount of work it can perform in a given period of time.

a task has three states:

Running (executing on the CPU);

Ready (ready to be executed);

Blocked (waiting for an event, I/O for example).

Most tasks are blocked or ready most of the time because generally only one task can run at a time per CPU. The number of items in the ready queue can vary greatly, depending on the number of tasks the system needs to perform and the type of scheduler that the system uses. On simpler non-preemptive but still multitasking systems, a task has to give up its time on the CPU to other tasks, which can cause the ready queue to have a greater number of overall tasks in the ready to be executed state (resource starvation).

Usually the data structure of the ready list in the scheduler is designed to minimize the worst-case length of time spent in the scheduler's critical section, during which preemption is inhibited, and, in some cases, all interrupts are disabled, but the choice of data structure depends also on the maximum number of tasks that can be on the ready list.

If there are never more than a few tasks on the ready list, then a doubly linked list of ready tasks is likely optimal. If the ready list usually contains only a few tasks but occasionally contains more, then the list should be sorted by priority. That way, finding the highest priority task to run does not require iterating through the entire list. Inserting a task then requires walking the ready list until reaching either the end of the list, or a task of lower priority than that of the task being inserted.

Care must be taken not to inhibit preemption during this search. Longer critical sections should be divided into small pieces. If an interrupt occurs that makes a high priority task ready during the insertion of a low priority task, that high priority task can be inserted and run immediately before the low priority task is inserted.

The critical response time, sometimes called the flyback time, is the time it takes to queue a new ready task and restore the state of the highest priority task to running. In a well-designed RTOS, readying a new task will take 3 to 20 instructions per ready-queue entry, and restoration of the highest-priority ready task will take 5 to 30 instructions.

In more advanced systems, real-time tasks share computing resources with many non-real-time tasks, and the ready list can be arbitrarily long. In such systems, a scheduler ready list implemented as a linked list would be inadequate.

Hard real-time software systems have a set of strict deadlines, and missing a deadline is considered a system failure. Examples of hard real-time systems: airplane sensor and autopilot systems, spacecrafts and planetary rovers.

Soft real-time systems try to reach deadlines but do not fail if a deadline is missed. However, they may degrade their quality of service in such an event to improve responsiveness. Examples of soft real-time systems: audio and video delivery software for entertainment (lag is undesirable but not catastrophic).

Firm real-time systems treat information delivered/computations made after a deadline as invalid. Like soft real-time systems, they do not fail after a missed deadline, and they may degrade QoS if a deadline is missed (1). Examples of firm real-time systems: financial forecast systems, robotic assembly lines .

Task Periodicity :

The system is subjected to real time, i.e. response should be guaranteed within a specified timing constraint or system should meet the specified deadline. For example flight control system, real-time monitors etc.

There are two types of tasks in real-time systems:

Periodic tasks

Dynamic tasks

Periodic Tasks: In periodic task, jobs are released at regular intervals. A periodic task is one which repeats itself after a fixed time interval.

A periodic task is denoted by five tuples: $\mathbf{T_i = < \Phi_i, P_i, e_i, D_i >}$

Where,

$\mathbf{\Phi_i}$ – is the phase of the task. Phase is release time of the first job in the task. If the phase is not mentioned then release time of first job is assumed to be zero.

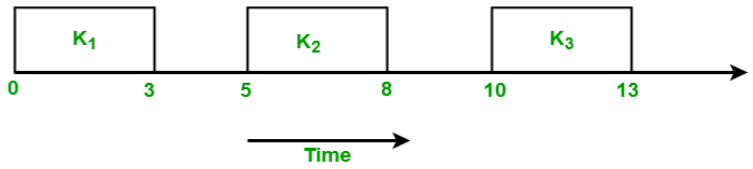
$\mathbf{P_i}$ – is the period of the task i.e. the time interval between the release times of two consecutive jobs.

$\mathbf{e_i}$ – is the execution time of the task.

$\mathbf{D_i}$ – is the relative deadline of the task.

For example: Consider the task T_i with period = 5 and execution time = 3

Phase is not given so, assume the release time of the first job as zero. So the job of this task is first released at t = 0 then it executes for 3s and then next job is released at t = 5 which executes for 3s and then next job is released at t = 10. So jobs are released at t = 5k where k = 0, 1, . . ., n



Hyper period of a set of periodic tasks is the least common multiple of periods of all the tasks in that set. For example, two tasks T_1 and T_2 having period 4 and 5 respectively will have a hyper period, H = lcm(p1, p2) = lcm(4, 5) = 20. The hyper period is the time after which pattern of job release times starts to repeat.

Dynamic Tasks: It is a sequential program that is invoked by the occurrence of an event. An event may be generated by the processes external to the system or by processes internal to the system. Dynamically arriving tasks can be categorized on their criticality and knowledge about their occurrence times.

Aperiodic Tasks: In this type of task, jobs are released at arbitrary time intervals i.e. randomly. Aperiodic tasks have soft deadlines or no deadlines.

Sporadic Tasks: They are similar to aperiodic tasks i.e. they repeat at random instances. The only difference is that sporadic tasks have hard deadlines. A sporadic task is denoted by three tuples: $\mathbf{T_i = (e_i, g_i, D_i)}$

Where

$\mathbf{e_i}$ – the execution time of the task.

g_i – the minimum separation between the occurrence of two consecutive instances of the task.

D_i – the relative deadline of the task.

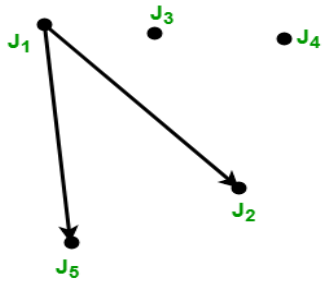
Jitter: Sometimes actual release time of a job is not known. Only know that r_i is in a range [r_i-, r_i+]. This range is known as release time jitter. Here r_i– is how early a job can be released and r_i+ is how late a job can be released. Only range [e_i-, e_i+] of the execution time of a job is known. Here e_i– is the minimum amount of time required by a job to complete its execution and e_i+ the maximum amount of time required by a job to complete its execution.

Precedence Constraint of Jobs: Jobs in a task are independent if they can be executed in any order. If there is a specific order in which jobs in a task have to be executed then jobs are said to have precedence constraints. For representing precedence constraints of jobs a partial order relation < is used. This is called precedence relation. A job J_i is a predecessor of job J_j if $J_i < J_j$ i.e. J_j cannot begin its execution until J_i completes. J_i is an immediate predecessor of J_j if $J_i < J_j$ and there is no other job J_k such that $J_i < J_k < J_j$. J_i and J_j are independent if neither $J_i < J_j$ nor $J_j < J_i$ is true.

An efficient way to represent precedence constraints is by using a directed graph G = (J, <) where J is the set of jobs. This graph is known as the precedence graph. Jobs are represented by vertices of graph and precedence constraints are represented using directed edges. If there is a directed edge from J_i to J_j then it means that J_i is immediate predecessor of J_j. For example: Consider a task T having 5 jobs J_1, J_2, J_3, J_4 and J_5 such that J_2 and J_5 cannot begin their execution until J_1 completes and there are no other constraints.

The precedence constraints for this example are:

$J_1 < J_2$ and $J_1 < J_5$



Precedence Graph

Set representation of precedence graph:

< (1) = { }

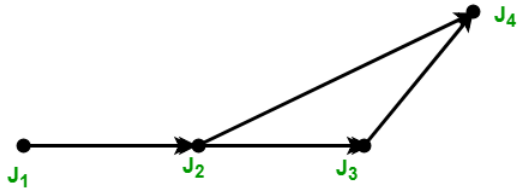
< (2) = {1}

< (3) = { }

< (4) = { }

< (5) = {1}

Consider another example where precedence graph is given and you have to find precedence constraints



From the above graph, we derive the following precedence constraints:

$J_1 < J_2$

$J_2 < J_3$

$J_2 < J_4$

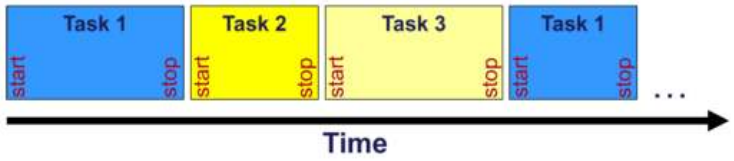
$J_3 < J_4$

Sporadic task :

A **sporadic** task is characterized by three positive integers; an execution time e, a deadline d (relative to the release time), and a minimum separation p, with e <= d and e <= p. **Sporadic tasks** may make a request at any time, but two successive requests must be separated in time by at least p "time units."

Real-time Schedulers :

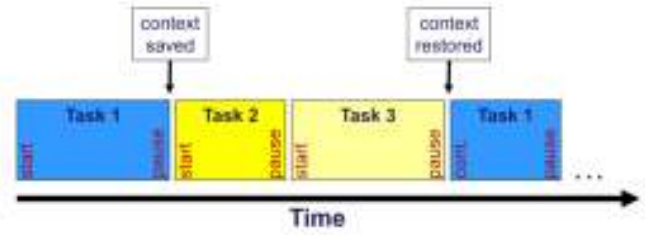
An RTC scheduler is very simple. Indeed, I have previously [and only slight inaccurately] referred to one as a "one line RTOS". The idea is that one task runs until it has completed its work, then terminates. Then the next task runs similarly. And so forth until all the tasks have run, when the sequence starts again.



The simplicity of this scheme is offset by the drawback that each task's allocation of time is totally affected by all the others. The system will not be very deterministic. But, for some applications, this is quite satisfactory. An added level of sophistication might be support for task suspend, which means that one or more tasks may be excluded from the execution sequence until they are required again.

Round robin [RR]

An RR scheduler is the next level of complexity. Tasks are run in sequence in just the same way [with task suspend being a possibility], except that a task does not need to complete its work, it just relinquishes the CPU when convenient to do so. When it is scheduled again, it continues from where it left off.

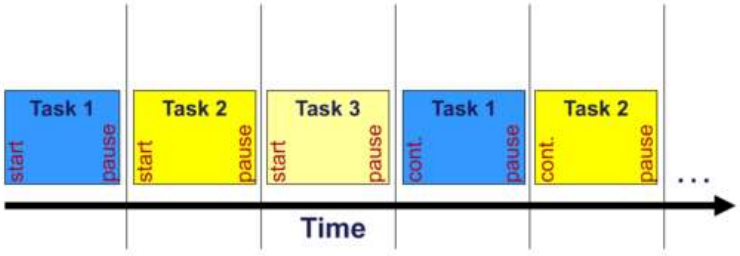


The greater flexibility of an RR scheduler comes at the cost of complexity. When a task relinquishes the CPU, its context [basically machine register values] needs to be saved so that it can be restored next time the task is scheduled. This process is required for all the other scheduler varieties that I will discuss.

As with RTC, an RR scheduler still relies of each task behaving well and not hanging on to the processor for too long. Both RTC and RR are "cooperative multitasking".

Time slice [TS]

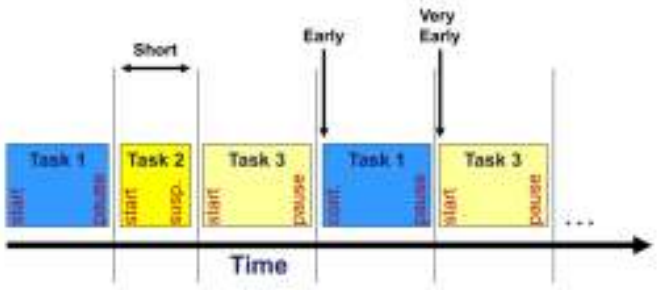
A TS scheduler is a straightforward example of "preemptive multitasking". The idea is to divide time into "slots", each of which might be, say, 1mS. Each task gets to run in a slot. At the end of its allocated time, it is interrupted and the next task run.



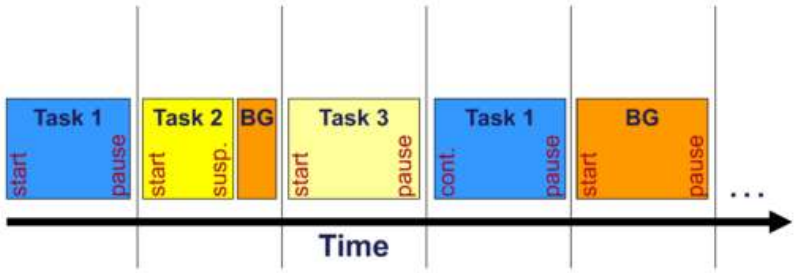
The scheduling is not now dependent on tasks being "good citizens", as time utilization is managed fairly. A system built with a TS scheduler may be fully deterministic [i.e. predictable] – it is truly real time.

Time slice with background task [TSBG]

Although a TS scheduler is neat and tidy, there is a problem. If a task finds that it has no work to do, its only option is to loop – burning CPU time – until it can do something useful. This means that it might waste a significant proportion of its slot and an indefinite number of further slots. Clearly, the task might suspend itself [to be woken again when it is needed], but this messes up the timing of the other tasks.



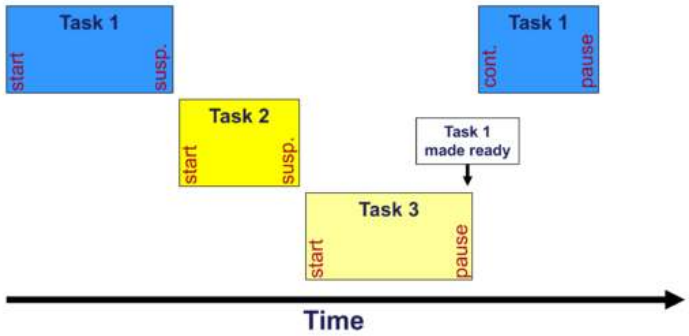
This is unfortunate, as the determinism of the system is compromised. A solution is to enhance the scheduler so that, if a task suspends itself, the remainder of its slot is taken up by a "background task"; this task would also use the full slots of any suspended tasks. This restores the timing integrity.



What the background task actually does depends on the application, but broadly it must be non-time-citical code – like self-testing. There is, of course, the possibility that the background task will never get scheduled. Also, this special task cannot be suspended.

Priority [PRI]

A common, more sophisticated scheduling scheme is PRI, which is used in many [most] commercial RTOS products. The idea is that each task has a priority and is either "ready" [to run] or "suspended". The scheduler runs the task with the highest priority that is "ready". When that task suspends, it runs the one with the next highest priority. If an event occurs, which may have readied a higher priority task, the scheduler is run.



Although more complex, a PRI scheduler give most flexibility for many applications.

Commercial RTOS products, like our own Nucleus RTOS, tend to use a priority scheduling scheme, but allow multiple tasks at each priority level. A time slice mechanism is then employed to allocate CPU time between multiple "ready" tasks of the same priority.

Types Of Tasks :

Types of Real-Time Tasks

Based on the way real-time tasks recur over a period of time, it is possible to classify them into three main categories: periodic, sporadic, and aperiodic tasks. In the following, we discuss the important characteristics of these three major categories of real-time tasks.

Periodic Task: A periodic task is one that repeats after a certain fixed time interval. The precise time instants at which periodic tasks recur are usually demarcated by clock interrupts. For this reason, periodic tasks are sometimes referred to as clock-driven tasks. The fixed time interval after which a task repeats is called the period of the task. If Ti is a periodic task, then the time from 0 till the occurrence of the first instance of Ti (i.e. Ti(1)) is denoted by φi, and is called the phase of the task. The second instance (i.e. Ti(2)) occurs at φi + pi. The third instance (i.e. Ti(3)) occurs at φi + 2pi and so on.

Formally, a periodic task Ti can be represented by a 4 tuple (φi, pi, ei, di)

where pi is the period of task,

ei is the worst case execution time of the task,

and di is the relative deadline of the task.

We shall use this notation extensively in future discussions.

To illustrate the above notation to represent real-time periodic tasks, let us consider the track correction task typically found in a rocket control software. Assume the following characteristics of the track correction task. The track correction task starts 2000 milliseconds after the launch of the rocket, and recurs periodically every 50 milliseconds then on. Each instance of the task requires a processing time of 8 milliseconds and its relative deadline is 50 milliseconds. Recall that the phase of a task is defined by the occurrence time of the first instance of the task. Therefore, the phase of this task is 2000 milliseconds.

This task can formally be represented as (2000 mSec, 50 mSec, 8 mSec, 50 mSec).

When the deadline of a task equals its period (i.e. pi=di), we can omit the fourth tuple. In this case, we can represent the task as Ti= (2000 mSec, 50 mSec, 8 mSec). This would automatically mean pi=di=50 mSec. Similarly, when φi = 0, it can be omitted when no confusion arises.

So, Ti = (20mSec; 100mSec) would indicate a task with φi = 0, pi=100mSec, ei=20mSec, and di=100mSec. Whenever there is any scope for confusion, we shall explicitly write out the parameters Ti = (pi=50 mSecs, ei = 8 mSecs, di = 40 mSecs), etc. Φ Φ + pi Φ + 2*pi Φ = 2000 di ei 0

A vast majority of the tasks present in a typical real-time system are periodic. The reason for this is that many activities carried out by real-time systems are periodic in nature, for example monitoring certain conditions, polling information from sensors at regular intervals to carry out certain action at regular intervals (such as drive some actuators). We shall consider examples of such tasks found in a typical chemical plant. In a chemical plant several temperature monitors, pressure monitors, and chemical concentration monitors periodically sample the current temperature, pressure, and chemical concentration values which are then communicated to the plant controller. The instances of the temperature, pressure, and chemical concentration monitoring tasks normally get generated through the interrupts received from a periodic timer. These inputs are used to compute corrective actions required to maintain the chemical reaction at a certain rate. The corrective actions are then carried out through actuators.

Sporadic Task: A sporadic task is one that recurs at random instants. A sporadic task Ti can be is represented by a three tuple: Ti = (ei, gi, di) where ei is the worst case execution time of an instance of the task, gi denotes the minimum separation between two consecutive instances of the task, di is the relative deadline. The minimum separation (gi) between two consecutive instances of the task implies that once an instance of a sporadic task occurs, the next instance cannot occur before gi time units have elapsed. That is, gi restricts the rate at which sporadic tasks can arise. As done for periodic tasks, we shall use the convention that the first instance of a sporadic task Ti is denoted by Ti(1) and the successive instances by Ti(2), Ti(3), etc. Many sporadic tasks such as emergency message arrivals are highly critical in nature. For example, in a robot a task that gets generated to handle an obstacle that suddenly appears is a sporadic task. In a factory, the task that handles fire conditions is a sporadic task. The time of occurrence of these tasks can not be predicted. The criticality of sporadic tasks varies from highly critical to moderately critical.

For example, an I/O device interrupt, or a DMA interrupt is moderately critical. However, a task handling the reporting of fire conditions is highly critical.

Aperiodic Task: An aperiodic task is in many ways similar to a sporadic task. An aperiodic task can arise at random instants. However, in case of an aperiodic task, the minimum separation gi between two consecutive instances can be 0. That is, two or more instances of an aperiodic task might occur at the same time instant. Also, the deadline for an aperiodic tasks is expressed as either an average value or is expressed statistically. Aperiodic tasks are generally soft real-time tasks. It is easy to realize why aperiodic tasks need to be soft real-time tasks. Aperiodic tasks can recur in quick succession. It therefore becomes very difficult to meet the deadlines of all instances of an aperiodic task. When several aperiodic tasks recur in a quick succession, there is a bunching of the task instances and it might lead to a few deadline misses. As already discussed, soft real-time tasks can tolerate a few deadline misses. An example of an aperiodic task is a logging task in a distributed system. The

logging task can be started by different tasks running on different nodes. The logging requests from different tasks may arrive at the logger almost at the same time, or the requests may be spaced out in time. Other examples of aperiodic tasks include operator requests, keyboard presses, mouse movements, etc. In fact, all interactive commands issued by users are handled by aperiodic tasks.

Clock-Driven Scheduling :

Clock-driven schedulers make their scheduling decisions regarding which task to run next only at the clock interrupt points. Clock-driven schedulers are those for which the scheduling points are determined by timer interrupts. Clock- driven schedulers are also called off-line schedulers because these schedulers fix the schedule before the system starts to run. That is, the scheduler pre-determines which task will run when. Therefore, these schedulers incur very little run time overhead. However, a prominent shortcoming of this class of schedulers is that they can not satisfactorily handle aperiodic and sporadic tasks since the exact time of occurrence of these tasks can not be predicted. For this reason, this type of schedulers is also called static scheduler. In this section, we study the basic features of two important clock-driven schedulers: table-driven and cyclic schedulers.

Table-Driven Scheduling :

Table-driven schedulers usually pre-compute which task would run when, and store this schedule in a table at the time the system is designed or configured. Rather than automatic computation of the schedule by the scheduler, the application programmer can be given the freedom to select his own schedule for the set of tasks in the application and store the schedule in a table (called schedule table) to be used by the scheduler at run time. An example of a schedule

if a set ST = {Ti} of n tasks is to be scheduled, then the entries in the table will replicate themselves after LCM (p1, p2, … ,pn) time units, where p1, p2, …, pn are the periods of T1, T2, ..., Tn. For example, if we have the following three tasks: (e1=5 msecs, p1=20 msecs), (e2=20 msecs, p2=100 msecs), (e3=30 msecs, p3=250 msecs); then, the schedule will repeat after every 1000 msecs. So, for any given task set, it is sufficient to store entries only for LCM (p1, p2, … ,pn) duration in the schedule table. LCM (p1, p2, …, pn) is called the major cycle of the set of tasks ST.

MODULE-3

Software and programming concept

Processor selection for an embedded system:-

Selecting the right processor for the given application is one of the main challenges and an important task for the design of efficient embedded system. Three types of processors are generally used in embedded systems namely: general-purpose processors, microcontrollers, and Digital Signal Processors (DSP).

General purpose processors, also known as microprocessors, are available with a bit range of 4 bit to 64 bit from various vendors and are characterized by distinct parameters such as: data lines, address lines, speed, and cost. These are used in many embedded systems from Keyboard controllers to Robots. Microcontrollers, on the other hand, are used in controlled applications and playa key role in the design of embedded systems. These are available from many vendors for low-end applications to high-end applications with a bit range of 4 bit to 64 bit. The third type of processors namely DSP processors are basically designed for highly mathematical intensive applications. They are available in floating point (32 bit) and fixed point (16 bit) form from various vendors and with different types in each category. These are used to handle the complex applications like signal processing, image processing and communications etc.

For selection of the suitable processor.

There are two aspects about finding the right processor for a given application. The first aspect is to choose the processor. The second is to evaluate whether the chosen processor is a good match for the application. There are many techniques for both processor selection and processor evaluation for embedded systems.

With numerous kinds of processors with various design philosophies available at our disposal for using in our design, following considerations need to be factored during processor selection for an embedded system.

- Performance Considerations
- Power considerations
- Peripheral Set
- Operating Voltage
- Specialized Processing Units

Performance considerations

The first and foremost consideration in selecting the processor is its performance. The performance speed of a processor is dependent primarily on its architecture and its silicon design. Evolution of fabrication techniques helped packing more transistors in same area there by reducing the propagation delay. Also presence of cache reduces instruction/data fetch timing. Pipelining and super-scalar architectures further improves the performance of the processor. Branch prediction, speculative execution etc are some other techniques used for improving the execution rate. Multi-cores are the new direction in improving the performance.

Rather than simply stating the clock frequency of the processor which has limited significance to its processing power, it makes more sense to describe the capability in a standard notation. MIPS (Million Instructions Per Second) or MIPS/MHz was an earlier notation followed by Dhrystones and latest EEMBC's ***CoreMark***. CoreMark is one of the best ways to compare the performance of various processors.

Processor architectures with support for extra instruction can help improving performance for specific applications. For example, SIMD (Single Instruction/Multiple Data) set and Jazelle – Java acceleration can help in improving multimedia and JVM execution speeds.

So size of cache, processor architecture, instruction set etc has to be taken in to account when comparing the performance.

Power Considerations

Increasing the logic density and clock speed has adverse impact on power requirement of the processor. A higher clock implies faster charge and discharge cycles leading to more power consumption. More logic leads to higher power density there by making the heat dissipation difficult. Further with more emphasis on greener technologies and many systems becoming battery operated, it is important the design is for optimal power usage.

Techniques like ***frequency scaling*** – reducing the clock frequency of the processor depending on the load, ***voltage scaling*** – varying the voltage based on load can help in achieving lower power usage. Further asymmetric multiprocessors, under near idle conditions, can effectively power off the more powerful core and load the less powerful core for performing the tasks. SoC comes with advanced power gating techniques that can shut down clocks and power to unused modules.

Peripheral Set

Every system design needs, apart from the processor, many other peripherals for input and output operations. Since in an embedded system, almost all the processors used are SoCs, it is better if the necessary peripherals are available in the chip itself. This offers various benefits compared to peripherals in external IC's such as optimal power architecture, effective data communication using DMA, lower BoM etc. So it is important to have peripheral set in consideration when selecting the processor.

Operating Voltages

Each and every processor will have its own operating voltage condition. The operating voltage maximum and minimum ratings will be provided in the respective data sheet or user manual.

While higher end processors typically operate with 2 to 5 voltages including 1.8V for Cores/Analogue domains, 3.3V for IO lines, needs specialized PMIC devices, it is a deciding factor in low end micro-controllers based on the input voltage. For example it is cheaper to work with a 5V micro-controller when the input supply is 5V and a 3.3 micro-controllers when operated with Li-on batteries.

Specialized Processing

Apart from the core, presence of various co-processors and specialized processing units can help achieving necessary processing performance. Co-processors execute the instructions fetched by the

primary processor thereby reducing the load on the primary. Some of the popular co-processors include

Floating Point Co-processor:

RISC cores supports primarily integer only instruction set. Hence presence of a FP co-processor can be very helpful in application involving complex mathematical operations including multimedia, imaging, codecs, signal processing etc.

Graphic Processing Unit:

GPU(Graphic Processing Unit) also called as Visual processing unit is responsible for drawing images on the frame buffer memory to be displayed. Since human visual perception needed at-least 16 Frames per second for a smooth viewing, drawing for HD displays involves a lot of data bandwidth. Also with increasing graphic requirements such as textures, lighting shaders etc, GPU's have become a mandatory requirements for mobile phones, gaming consoles etc.

Various GPU's like ARM's MALI, PowerVX, OpenGL etc are increasing available in higher end processors. Choosing the right co-processor can enable smooth design of the embedded application.

Digital Signal Processors

DSP is a processor designed specifically for signal processing applications. Its architecture supports processing of multiple data in parallel. It can manipulate real time signal and convert to other domains for processing. DSP's are either available as the part of the SoC or separate in an external package. DSP's are very helpful in multimedia applications. It is possible to use a DSP along with a processor or use the DSP as the main processor itself.

Price

Various considerations discussed above can be taken in to account when a processor is being selected for an embedded design. It is better to have some extra buffer in processing capacities to enable enhancements in functionality without going for a major change in the design. While engineers (especially software/firmware engineers) will want to have all the functionalities, price will be the determining factor when designing the system and choosing the right processor.

StateChart

State Charts are among the widely used specification techniques, suitable particularly for control-dominated embedded system specifications. It is a modification of finite state machine (FSM) commonly used in controller specification. It may be noted that the normal FSM specification suffers from the following drawbacks.

1. Complexity of the state diagram increases dramatically as the number of possible states increases. This poses a major bottleneck in modelling large systems.

2. In traditional state machines, there is no concept of hierarchy. This, in turn, leads to the previous problem of state-explosion.

3. There is no support for concurrent constructs. Traditional state machine modelling is based on sequential transitions from one state to the next. Concurrent systems cannot be modelled in this manner, as various portions of the system may be in different states.

State Charts introduced by David Harel in 1987, have some special features to overcome these problems. It allows the system to be simultaneously in more than one state. The concurrency can be modelled easily. The concept will become more clear when we discuss about the super states. The states can communicate and synchronize between themselves. Some of the important features are as follows.

1. All orthogonal regions of the chart (that is, different subsystems within the whole system) accept events sent to the object.

2. One region may create an event as a result of a transition that is consumed by another orthogonal region.

3. A guard may be used to test if another region is in a certain state before allowing a transition to occur to the guarded state.

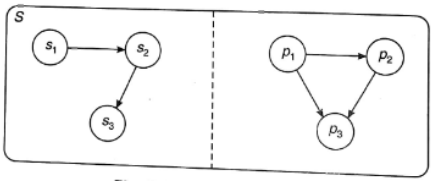
There can be two types of super states.

1. OR-super state: This is commonly available in FSMs. The system, at time, is in one of the constituent sub states of an OR-super state. For example, FSM, it is always in one of its sub states.

2. AND-super state: An AND-super state encompasses a number of sub states, the can be in all these states simultaneously. Though this apparently seems counter-intuitive, it can be utilized to model systems with different subsystems. For example, as shown in Fig., the state S is an AND-super state. It has two sub states, each of which OR-super state. Thus, the system can be

simultaneously in one of the basic states{s1,s2, S3, ...} for the first subsystem, and in one of {P1, P2, P3, ...} for the second subsystem. If the number

of states in the first subsystem is n1 and that in the second subsystem n2, then the total number of states needed in the AND-super state is n1+n2 . How in the absence of such a feature, a total of n1 X n2 states will be needed to represent the system S. The AND-super states are also said to consist of orthogonal states. Each of these orthogonal states can be used to model different regions of the system.



(Figure of AND-super state.)

1. Transitions from super states: Figure shows a hierarchical State Chart in which the

state S is an OR-super state. While the system is in state S, it is in either of the states P or Q. From state P, on occurrence of event C, it transitions to state Q. In whatever sub state the system be, on occurrence of event E, it will switch from state S to state T.

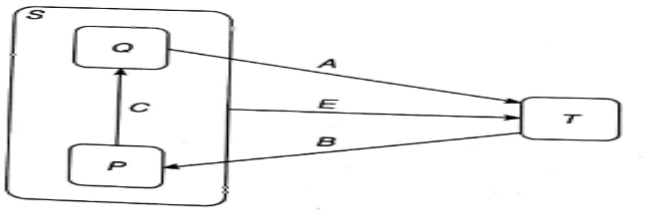


Fig. 6.2 Transitions from superstate.

2. Start and stop states: Like FSM, state chart may also have explicit start and stop state. Start state indicated by simple large dots, whereas the stop states are encircled dot. example, the overall State Chart has S as the start state. Within S,P is the start state.

That is whenever the state S is entered, unless otherwise stated. The state P will be reached. Thus P is also called the default state of S.

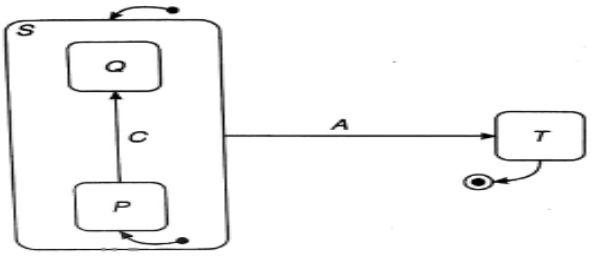


Fig. 6.3 Start and stop states.

3. History mechanism: Sometimes, it may be required that the system when transitioning into a superstate, should enter into the substate it was in the last time. This can be accomplished by the history mechanism (as shown in Fig). Here, while transitioning out of state S, the system remembers the substate in which it was. Later on, while transitioning from T to S, the arrow reaching H indicates that it should restart from the state the system remembered.

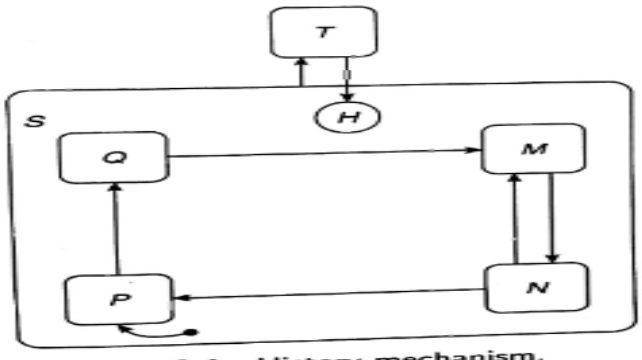


Fig. 6.4 History mechanism.

4. Timers: These are very important for real-time embedded systems. A timer is also represented as a state with a time value. Once the state has been entered, if no event occurs to create a transition out of the state, the timer will time out and marked timeout from this state will take place. This has been shown which the state P is a timer with a timeout period of 10 ms. If the event x does not occur within this 10 ms time, the transition labeled timeout to state S takes place.

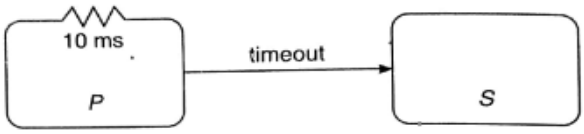


Fig. 6.5 Timer state.

5. Edge labels: The transitions in a StateChart are labeled as,

< event > < condition >][/ < reaction>]

Thus, there must be an event associated with a transition. The < condition> act as a Boolean guard. It is an expression involving values of variables. The condition is enclosed within.

'[' and ']'. The < reaction > part sets some variables to new values. Thus some typical edge labels can be,

key-pressed / on := 1

key-pressed (on = 1]/ on := 0

key-pressed (on = 0]/ on := 1; light = 1

6. State Chart simulation: StateChart execution consists of three phases.

In the first phase,the events and conditions are evaluated from the current state of the system. The transitions that are enabled by the process are identified.

In the second phase, for the enabled transitions, the reactions are evaluated, but are not immediately assigned to variables, rather, they are kept in temporaries.

In the third and final step, the transition to the new state takes place and the variables are assigned the values evaluated in the temporaries. This three-step semantics helps in simulating situations in which a variable is being modified in a transition, whereas, in another simultaneous transition, the variable appears in the condition part.

Specification and Description Language (SDL)

SDL is a premier language for specification, design, and development of real-time systems, in particular, the telecommunication applications. The currently available version is SDL2000, proposed in 1999 by ITU-T and some modifications later on. It is a graphical language and is based on the concept of Extended Finite State Machine (EFSM). An SDL system consists of one or more communicating agents, with the outermost agent communicating with the environment. An agent may in turn contain other agents in a hierarchy, definition of behaviour by EFSM, data variables of value or references data types and communication based on asynchronous message exchange.

SDL specification consists of a set of diagrams. The top-level diagram is the out most agent, it shows the connection between the system components and the environment. Each diagram has one or more pages, each page having the following:

• a frame with some information attached to the outside.

• the diagram heading showing the kind and identity of the item describe. In the top left corner.

• *page name and number of pages in the top right corner*

The following symbols are used in SDL:-

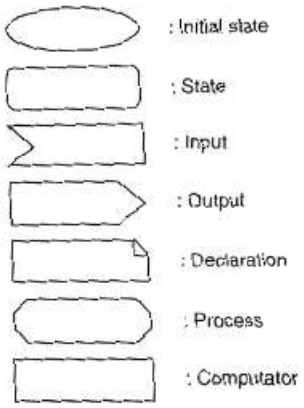


Fig. 6.9 Symbols in SDL.

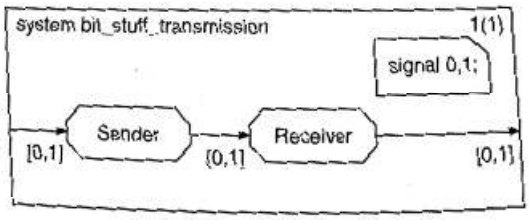


Fig. 6.10 SDL system for bit-stuffed transmission.

A system diagram can contain process agents or block agents. Instance with in a block agent The work concurrently and asynchronously, whereas, instances within a process are scheduled one at a time. A block agent can contain within it other block agent or process, where as a process can contain other process agents only within it.

SDL extends the concept of FSM in the following two ways.

1. Each agent has got a queue associated with it that stores all the signals received in a first-in-first-out manner.

2. Data can be received in signals, stored in variables, used in expressions, used to decide the behaviour of the agent, and passed out in output signals.

The FSM waits in its current state until one of the signals that can be consumed in that state is available.

Petri Nets

Petri Nets (also called Place/Transition Net or P/T Net) is one of the several mathematical representations of discrete distributed systems that have been widely used for system modelling in

many fields of science. It was invented by Carl Adam Petri in 1962 in his Ph.D. thesis. Two important intrinsic features of Petri Nets are concurrency and asynchronous nature. These, together with generality and flexibility have stimulated their applications in several areas including specification and verification of real-time embedded systems.

Basic Petri Nets

A Petri Net consists of places, transitions and directed arcs. Arcs can run between places and transitions. These can never be between places, or between transitions. Graphically, the places are represented by circles and transitions by solid lines.

The places from which an arc runs to a transition are called input places for the transition. Similarly, the places to which arcs run from a transition are called output places for the transition. Each place may contain any number of tokens.

In basic Petri Net structure, a token is represented by a small filled circle. The distribution of tokens among the places, at any point of time, is called marking.

A transition is enabled when all its input places have sufficient number of tokens. An enabled transition may fire, consuming the tokens from its input places, and placing a specified number of tokens at its output places.

The entire firing operation is atomic, that is, once a transition fires, all the subtasks, as listed above, are performed together. It may further be noted that the execution of Petri Nets is non-deterministic. That is, given a number of enabled transitions at a time, zero or any one of them can fire.

It is not necessary for an enabled transition to fire till the infinite time. Due to this non-determinism, Petri Nets are suitable for modelling concurrent behaviour of distributed systems. Figure shows a very simple Petri Net consisting of two paces pl and p2, and a transition tl. Initially, only the place pl has a token.

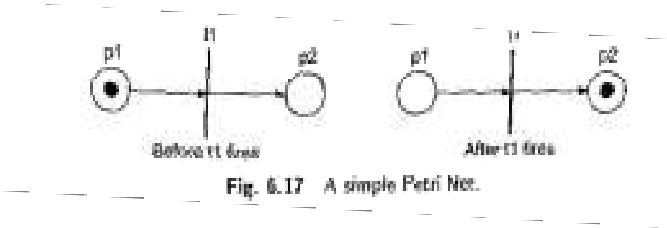


Fig. 6.17 A simple Petri Net.

Properties of a Petri Net On a system modelled using Petri Net, we can verify several properties. The following are some of those. The process allows us to verify many physical properties of the system.

1. **Termination:** Whether the net terminates.

2. **Immediate reachability:** Is a state reachable when a transition fires?
3. **Reachability:** Is a state eventually reachable?
4. **Liveness** in all states: Is there one transition that can fire?
5. **Partial dead**lock: Is there a state in which there is at least one transition that can never fire?

6. Deadlock: Is there a state in which none of the transitions can fire?

7. Safety in all states: Does each place contain at most one token?

8 **Bounded ness in all states:** Is there a limit to the number of tokens that can be in one place?

9 Conservativeness: Is the total number of tokens in the Petri Net constant?

Unified Modelling Language (UML)

The Unified Modelling Language (UML) is originally proposed for modelling complex, software intensive systems. It provides quite a few advantages over SDL. It is less formal, and thus can be used at an early stage to structure and analyse the concepts of an application domain before the functional design is made. SDL does not support relations in object modelling. thus, UML can be used to develop the formal functional design in SDL using the object models of UML.

UML is a graphical language consisting of different types of diagrams that can be used to scribe a model from different angles.

The diagrams can be classified into three categories.

• **Behaviour diagrams**- describing behavioural features of a system. This include activity, state machine, use case, as well as the following four interaction diagrams.

Interaction diagrams-a subset of behaviour diagrams emphasizing object interaction

This includes communication, interaction, overview, sequence, and timing diagram

• **Structure diagra**ms- depict the elements of a specification that are irrespective of time.

The constituents are class, composite structure, component, deployment, object package diagrams.

Activity diagram

This represents the operational step-by-step workflows of components in a system. It gives the overall flow of control. UML activity diagrams may be considered as object-oriented equivalent of flow charts and data flow diagrams. Thus, it often consists of different types of components as shown in figure.

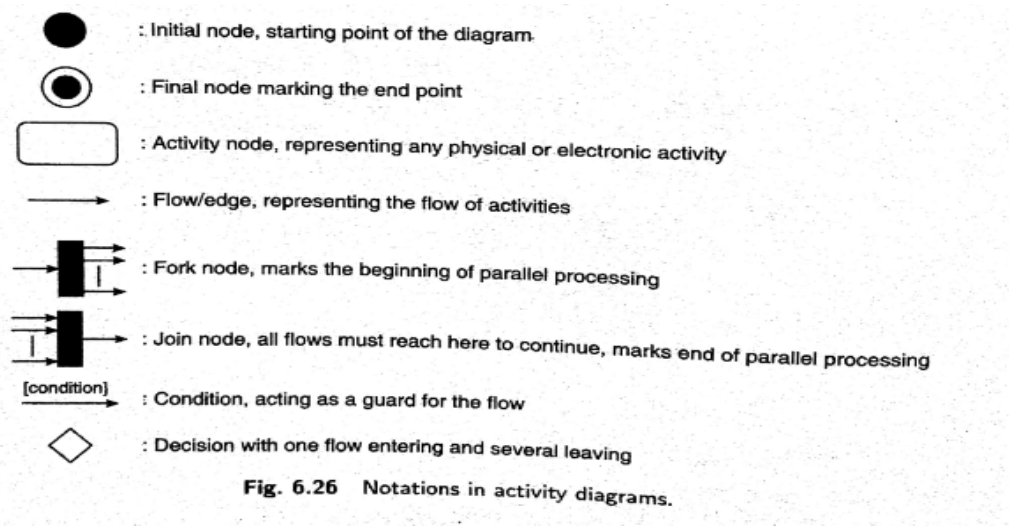


Fig. 6.26 Notations in activity diagrams.

Class diagram is a static diagram. It represents the static view of an application. Class diagram is not only used for visualizing, describing, and documenting different aspects of a system but also for constructing executable code of the software application.

Class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagrams are widely used in the modelling of objectoriented systems because they are the only UML diagrams, which can be mapped directly with object-oriented languages.

Class diagram shows a collection of classes, interfaces, associations, collaborations, and constraints. It is also known as a structural diagram.

Purpose of Class Diagrams

The purpose of class diagram is to model the static view of an application. Class diagrams are the only diagrams which can be directly mapped with object-oriented languages and thus widely used at the time of construction.

UML diagrams like activity diagram, sequence diagram can only give the sequence flow of the application, however class diagram is a bit different. It is the most popular UML diagram in the coder community.

The purpose of the class diagram can be summarized as −

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.

- Forward and reverse engineering.

Component diagrams are different in terms of nature and behavior. Component diagrams are used to model the physical aspects of a system. Now the question is, what are these physical aspects? Physical aspects are the elements such as executables, libraries, files, documents, etc. which reside in a node.

Component diagrams are used to visualize the organization and relationships among components in a system. These diagrams are also used to make executable systems.

Purpose of Component Diagrams

Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.

Thus from that point of view, component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files, etc.

Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment.

A single component diagram cannot represent the entire system but a collection of diagrams is used to represent the whole.

The purpose of the component diagram can be summarized as −

- Visualize the components of a system.
- Construct executables by using forward and reverse engineering.
- Describe the organization and relationships of the components.

the use case diagram is dynamic in nature, there should be some internal or external factors for making the interaction.

These internal and external agents are known as actors. Use case diagrams consists of actors, use cases and their relationships. The diagram is used to model the system/subsystem of an application. A single use case diagram captures a particular functionality of a system.

Hence to model the entire system, a number of use case diagrams are used.

Purpose of Use Case Diagrams

The purpose of use case diagram is to capture the dynamic aspect of a system. However, this definition is too generic to describe the purpose, as other four diagrams (activity, sequence, collaboration, and Statechart) also have the same purpose. We will look into some specific purpose, which will distinguish it from other four diagrams.

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. Hence, when a system is analyzed to gather its functionalities, use cases are prepared and actors are identified.

When the initial task is complete, use case diagrams are modelled to present the outside view.

In brief, the purposes of use case diagrams can be said to be as follows −

- Used to gather the requirements of a system.
- Used to get an outside view of a system.
- Identify the external and internal factors influencing the system.
- Show the interaction among the requirements are actors.

Sequence Diagrams –

A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place. We can also use the terms event diagrams or event scenarios to refer to a sequence diagram. Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.

Sequence Diagram Notations –

1. **Actors –** An actor in a UML diagram represents a type of role where it interacts with the system and its objects. It is important to note here that an actor is always outside the scope of the system we aim to model using the UML diagram.
   We use actors to depict various roles including human users and other external subjects. We represent an actor in a UML diagram using a stick person notation. We can have multiple actors in a sequence diagram.

2. **Lifelines –** A lifeline is a named element which depicts an individual participant in a sequence diagram. So basically each instance in a sequence diagram is represented by a lifeline. Lifeline elements are located at the top in a sequence diagram. The standard in UML for naming a lifeline follows the following format – Instance Name : Class Name

We display a lifeline in a rectangle called head with its name and type. The head is located on top of a vertical dashed line (referred to as the stem) as shown above. If we want to model an unnamed instance, we follow the same pattern except now the portion of lifeline's name is left blank.

Low Power Embedded System

Design

Power consumption is a very important issue in embedded system design. Many of the embedded applications are built around batteries, and thus, battery life is a critical concern in judging their acceptance. Some of the important issues to consider include power consumption limits, size restrictions, I/O requirements, operational duty cycle, etc. There are several practical considerations that are involved in the process.

1. Recharging facility*: It is not always possible to recharge/replace the batteries of embedded system*. This is particularly true for systems employed at remote and difficult-to reach places for example, a data collection centre for wildlife.

2. Device size: Unfortunately, battery technology has not scaled down at the same rate as IC technology. Thus, the weight and size of the device is often guided by the required battery capacity and its associated size. This is a very practical problem for mobile/portable *devices.*

3. Duration of operation*: The duration for which a device needs to be active, is another* major issue determining the total energy consumption. If the device is idle for majority of the time, a power-down mode may be incorporated into the design to save power.

4. Power required by the device: This is the most important issue and needs to be estimated even at the stage of initial design . A better estimation helps in the design process to try out alternatives.

5. I/O device types : The 1/0 interfaces, like optically isolated I/O and electromechanical relays consume high power. Thus, the system designer needs to avoid them altogether or minimize their active periods.

6. Speed of operation: As we will see later, power consumption is directly proportional to the frequency of operation. The system designer needs to identify the optimum speed for each component, so that the performance target is met, at the same time, individual are not operated at a speed higher than the required one.

Sources of Power Dissipation

Since majority of the systems designed are built around CMOS, in this section we will look into the different sources of power dissipation in CMOS. The total power consumed can broadly be divided into dynamic power and static power.

Dynamic power dissipation

This refers to the power consumed by a circuit/system due to some activities within it. For a static CMOS realization, at steady-state, either the pull-up (p) or the pull-down (n) network is OFF. Thus, when there are no activities in the system, power consumption is expected to very low. During circuit activities, the power consumed depend upon the following two mechanisms

1.Both p- and n- networks are ON simultaneously, shorting the power supply line to the ground. If V is the supply voltage and I mean is the average current drawn during the input transition, then the short-circuit power is given by.

P short circuit = I_{mean} x V DD

For properly sized and ratioed gates, the contribution to the overall dynamic power due

To P shortcircuit is of the order of 10-20%.

2. Switching power dissipation: This is the power consumed due to charging and discharging of capacitive loads when the circuit has some activities due to change in inputs. The capacitive load at different circuit gates depends upon the fan out of the gate, output capacitance, and wiring capacitances. It may be noted that a node with load capacitance might not switch when the clock is switching. To take care of this, a quantity called switching activity (a) is often used. It determines how often switching occurs on a node with load capacitance. If VDD is the supply voltage, V swing is the change in voltage level of the switched capacitance, C is the capacitance being switched and f is the frequency of operation, the switching power is given by,

P switching = C x V_{DD} x Vswing × α x f

Since in most of the cases,

Vswing = V_{DD}

Pswitching = C x V^2_{DD} × α × f

Static power dissipation

1. This is the power consumed when the circuit is not in active mode of operation. In such a situation, there is still some power dissipation due to various leakage mechanisms.

2. The situation is aggravated with the scaling of supply voltages. As the supply voltage is reduced, to keep the delay of a gate unchanged, the transistors need to be turned ON early by reducing their threshold voltages.

3. This, in turn, increases the leakage current in the sub threshold range of operation of the circuit. Due to the exponential nature of leakage current in the sub threshold range time of the transistor, it can no longer be ignored. .

4. The International Technology Roadmap for Semiconductors (ITRS) has projected an exponential increase in leakage power with minimization of devices.

5. Also, leakage increases with temperature. Thus, the Increased heat dissipation resulting from increase in leakage power consumption has a positive feedback on leakage.

6. There are three major leakage mechanisms sub threshold leakage gate direct tunneling and junction band-to-band tunneling.

. **Sub threshold leakage**: When gate voltage is below threshold voltage but very close to it , sub threshold conduction current flows between source and drain. It is caused by

The diffusion of minority carriers. It depends exponentially on the threshold voltage of the transistor . in nano - scaled devices, short channel effect (SCE) reduces the threshold voltage, There by increasing the sub threshold current.

• **Gate direct tunneling leakage**: Due to ultra-thin gate oxide, a high electric field can cause electrons to tunnel through the gate-oxide.This results in large gate leakage in nano scale transistors. An increase in the supply voltage and/or reduction in oxide thickness result in an exponential increases in gate tunneling current.

• **Junction band-to-band tunneling leakage**: Application of reverse bias across the highly doped p-n junction results in tunneling unction results in tunneling of electrons from valence band of p-side to the conduction band to the n-side .This is called band to band tunneling. Due to the use of high junction doping, large junction BTBT occurs at OFF state with the drain at V_{DD} and substrate at ground .The junction BTBT increases exponentially with an increase in junction doping and supply

voltage.

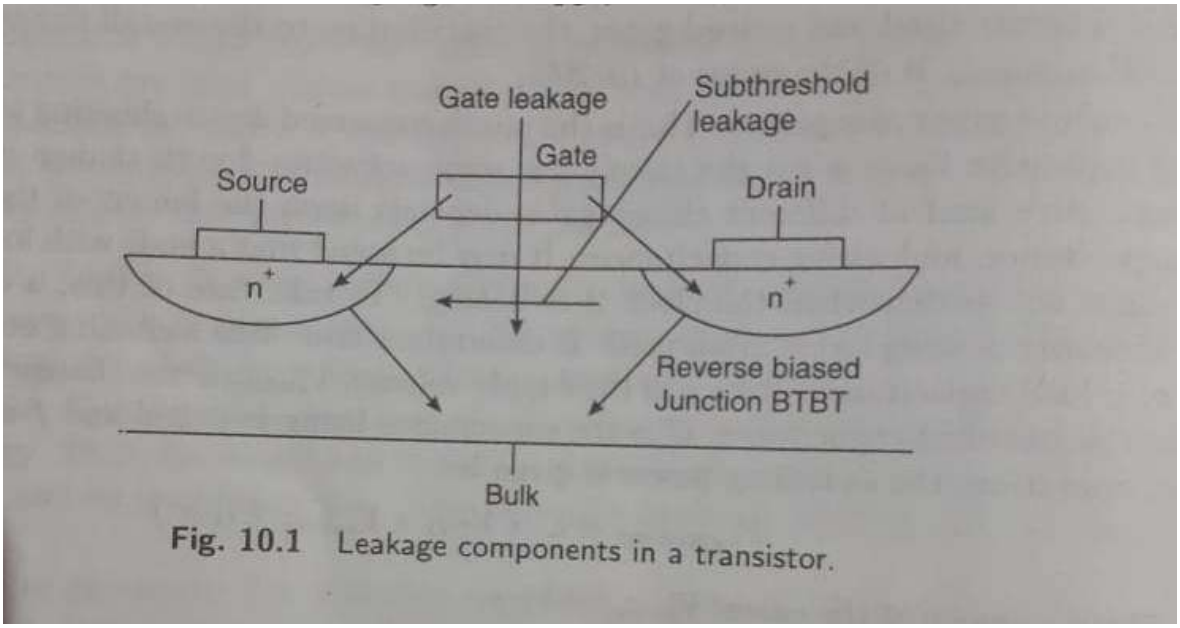


Fig. 10.1 Leakage components in a transistor.

Power Reduction Techniques

Power reduction can be attempted at all levels of design hierarchy - algorithm, architecture, logic, and device levels. In the following we give an overview of each of these (except the device level on which the embedded system designer often does not have any control). Higher the level at which power minimization is addressed, higher is the expected power saving

Algorithmic power minimization

.It mainly focuses on reducing the number of operations requiring larger power in a target implementation. For example, in many processors, the cost of an addition/subtraction operation may be different from a logical operation.

.Thus, to check" whether x is equal to y" one may first perform a subtraction operation followed by checking the status register for zero-bit.

.On the other hand, if the logical operation takes lesser power, x may be directly compared with y using a comparison instruction. The following are some of the important issues to be judged for selecting a particular algorithm from alternatives.

1. **Memory reference**: This is very important as memory is normally off-chip from the **processor. A large number of accesses to the memory means good amount of activity in the address/data bus lines. The memory access pattern is also important. If the access** pattern is sequential, only the least significant bits of address bus change, whereas, for random access through the memory, most of the address bits will switch, thus creating higher power dissipation

.2. Presence of cache memory: The presence and structure of cache memory plays an important role. Cache can be fruitfully utilized to reduce both execution time and power of an implementation if the underlying algorithm has got locality in its behavior . The locality may be both temporal and spatial in nature. While a temporal locality refers to the fact that a memory location accessed at some time is also likely to be accessed in near future, spatial locality means if a memory location is accessed at some time, its neighbouring locations are also likely to be accessed in near future. Thus, caching them inside the CPU cache saves not only the memory access time, but also the bus energy consumption is reduced.

3 . Recomputation vs. memory load /store: Normal power minimization techniques at algorithm level attempt to reduce the number of arithmetic operations. However, it may so happen that to reduce the number of operations, some repeatedly performed computation is done only once and stored at a memory location. Later, as and when necessary it is reloaded from the memory. This may lead to increased power consumption due to extra memory accesses. If the operands are already available in CPU registers or on-chip cache, it may be better to recompute the value, instead of loading it from memory, from power consumption point of view.

4. Compiler optimization technique: The typical techniques used by an optimizing compiler can be used to reduce power consumption of a piece of code. The strategies involve strength reduction, common subexpression elimination, minimizing memory traffic, etc. Loop unrolling is also often beneficial as it reduces loop overhead.

5. Number representation: This is another area for algorithmic power trade-off. The following points may be noted.

• ***Fixed vs floating point representation***: Fixed point operations are much simpler than floating point ones. Thus, it normally leads to power saving, through accuracy may suffer.

• **Sign-magnitude 2's complement**: Selection of sign-magnitude representation may have significant power saving over 2's complement, if input samples are uncorelated and range is minimized

.

• **Precision of operations**: This is important, since, having lower precision allows one to reduce the size of space needed to store the values. A typical example of this is to reduce the number of bits in mantissa portion in several signal processing applications including speech and image to improve circuit delay and power.

Architectural power minimization

1. The architectural level transformations can be used to introduce power saving in a design.

2. There are to generic techniques to save power at the cost of extra area , keeping the performance of the system unaltered .these are parallelism and pipelining. Suppose we have to design a system with supply voltage v and operating frequency f.

3. The system is expected to operate on a sequence of input data arriving at rate of f. Now, if we duplicate to have n such similar modules and the input data is processed by the modules in an inter leaved fashion, the blocks may operate at a lower speed-ideally, at a rate of f/n. Since the system is capable of running frequency f and the speed of a system is proportional to the supply voltage, we reduce the supply from v to v/n. This will effectively reduce the power consumed by the individual blocks by a factor of n (as power consumption is proportional to the square of supply voltage). Since all such systems are operating simultaneously, total power saving is in of the original power.

4. A problem with the scheme is that the hardware is duplicated with other necessary multiplexing an demultiplexing logic. Another possible architectural modification often suggested is pipelining. In this scheme, the functional block is divided into a sequence of sub-blocks, each of approximately same delay. Thus, if the number of sub-blocks be n, from pipelining principle, the overall system can produce output at a rate of about nix f. Now, if the supply voltage of individual stages is reduced by a factor of r. power reduces by a factor of 1/n. However, we need to accommodate extra latches between the stages for proper synchronization

between them. This introduces some overhead in terms of area, performance and power as well.

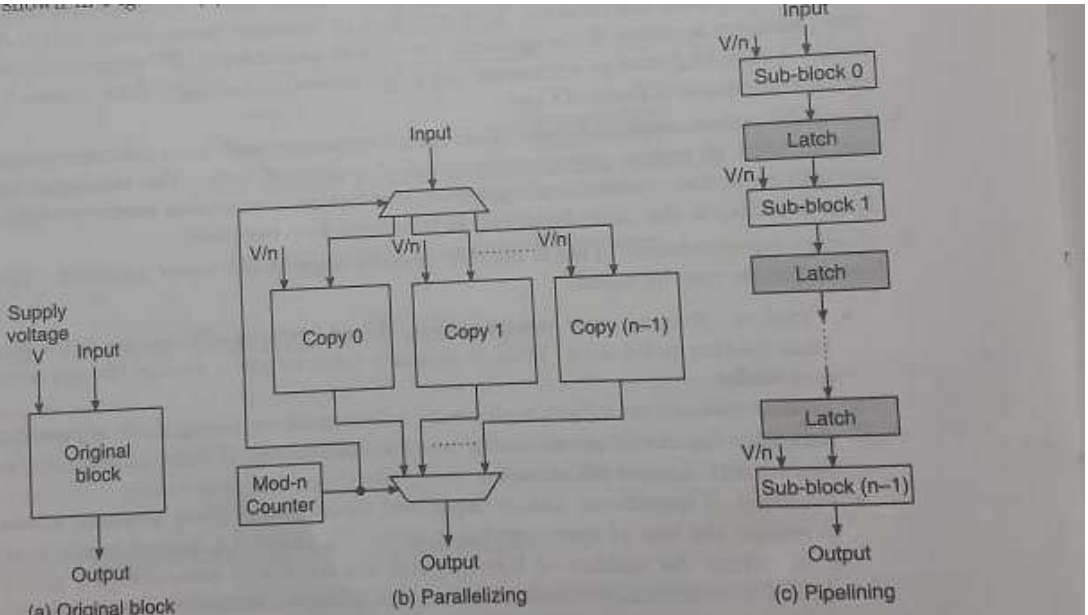


Fig. 10.2 Architectural transformations for low power.

Logic and circuit level power minimization

1. Static vs dynamic logic families : CMOS logic can be realized as static or dynamic Depending upon the signal transition probabilities, one of the design styles may edge over the other. For example, for a two-input NAND gate with uniform distribution for the inputs, the probability that the output is zero is 0.25, the probability of output being one is 0.75. Thus, the probability of a power0 → 1 transition is 0.25 X 0.75 = 0.1875. On the other hand, for a dynamic input is always precharged to 1. Thus, power will be consumed whenever output is 0. Hence, the probability of a power consuming transition is 0.25, which is higher than a static gate. However, dynamic gate has lower input capacitance (almost factor of 2 to 3) compared to static gate, as the p-network is absent. Hence, the **effective** capacitance that a dynamic gate sees is much lower. But, the power consumed in distributing the precharging signal also needs to be considered.

2. Glitches and hazards: This is another potential source of power consumption, particularly static CMOS circuits. A glitch at the output of a gate can come due to the differences in arrival times of input signals. A typical example of it is AND-OR-INVERT based implementation of the function f =

ab + āc . Assume that, b and c are always 1, while a is making a transition from 1 to 0. Ideally, output should remain fixed at 1, however, due to the delay introduced by the inverter, the output z will continue to be 0, when y is also 0. This changes the output f to 0. Thus, y, instead of remaining fixed at 1, will show a transition 1 0 1, introducing a glitch. The glitch can be removed by adding a redundant AND gate for the term be and feeding the output to the OR gate. It may be noted that dynamic logic does not suffer from any glitch power consumption since all inputs must be valid before the gate evaluates.

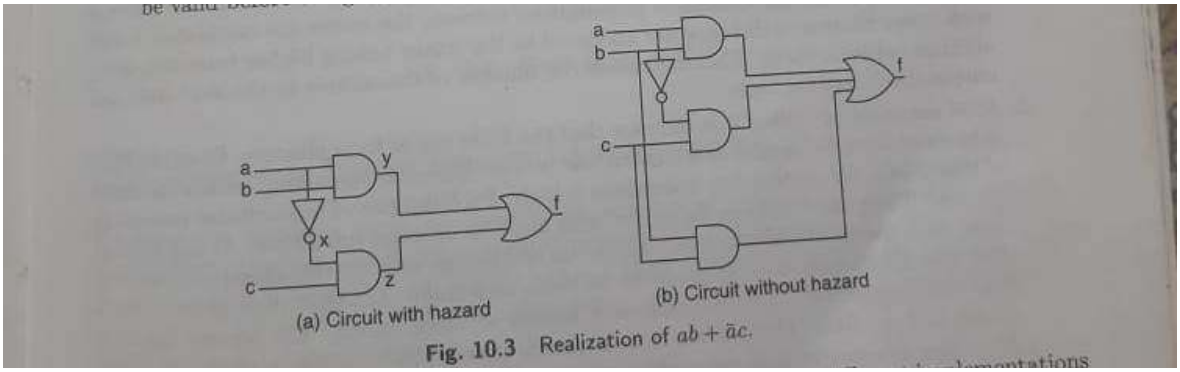


Fig. 10.3 Realization of $ab + \bar{a}c$.

3. Technology mapping: The logic synthesis library often contains different implementations of the same logic module. They normally differ in terms of area, delay, power, etc. A logic synthesis procedure targeted to power minimization may choose implementations that require higher area or delay, but score better in terms of power. For example ,consider a four –input AND function. The ON-probabilities of the gates are also shown. Total 0 1 transition probability of the first implementation is 0.25 x 0.75 +0.25 X 0.75 +0.9375 x 0.0625 = 0.4336. For the second implementation, it is, 0.25 x 0.75 + 0.125 x 0.875

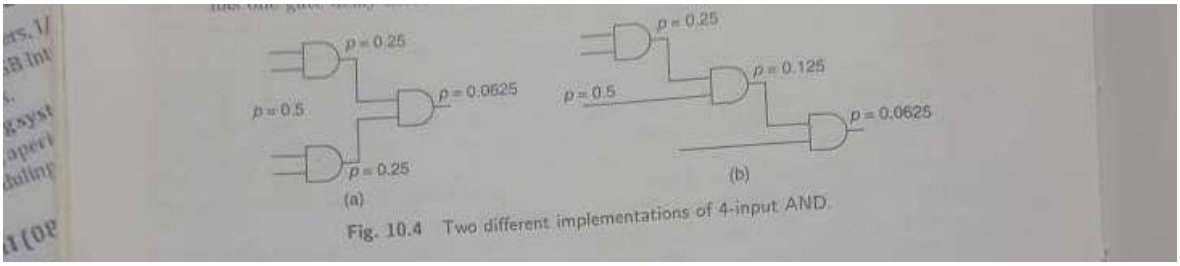


Fig. 10.4 Two different implementations of 4-input AND.

Control logic power minimization

Power optimization of controller circuitry is very important. This is because while the data path of a design can be selectively turned OFF when not in use, controller is always active. A controller is often specified and realized as a Finite State Machine (FSM).

1. Storage element design: Several flip-flop design have been reported in the literature with various area-power trade-offs.

2. State assignment: This is the assignment of binary codes to the FSM states to realize it.**For low power state encoding, first the steady-state probability for each of the states is determined. Next, the transition probabilities between the states are calculated. Codes with lesser Hamming distance are allocated to the states having higher transition probabilities between them. This minimizes the number of transitions in the next-state and** Output combinational logic .

3. FSM partitioning: Often it is the case that the FSM states form clusters. Once the FSM is in some state belonging to a cluster, the probability of its remaining within the states of this cluster for quite a few transitions is high. Probabilities of inter-cluster transitions are low. Thus, the FSM can be partitioned into two or more sub-FSMs. At any point of time, only one sub-FSM is active. We can do two things with the remaining sub machines The first alternative is to stop clocks to them and make sure that their primary input lines are masked-off. This is commonly known as clock gating. Thus, there is no dynamic power consumption in these sub-FSMS The other alternative is to use power gating by introducing sleep transistors to turn OFF power supply to them. Figure 10.6 shows introduction of sleep transistors. Power rating reduces leakage power as well, however, wake-up takes time. Thus, either the circuit performance suffers, or we have to turn ON probable active sub-

FSMS early--this lead to wastage of power as more than one sub-FSMs are active.

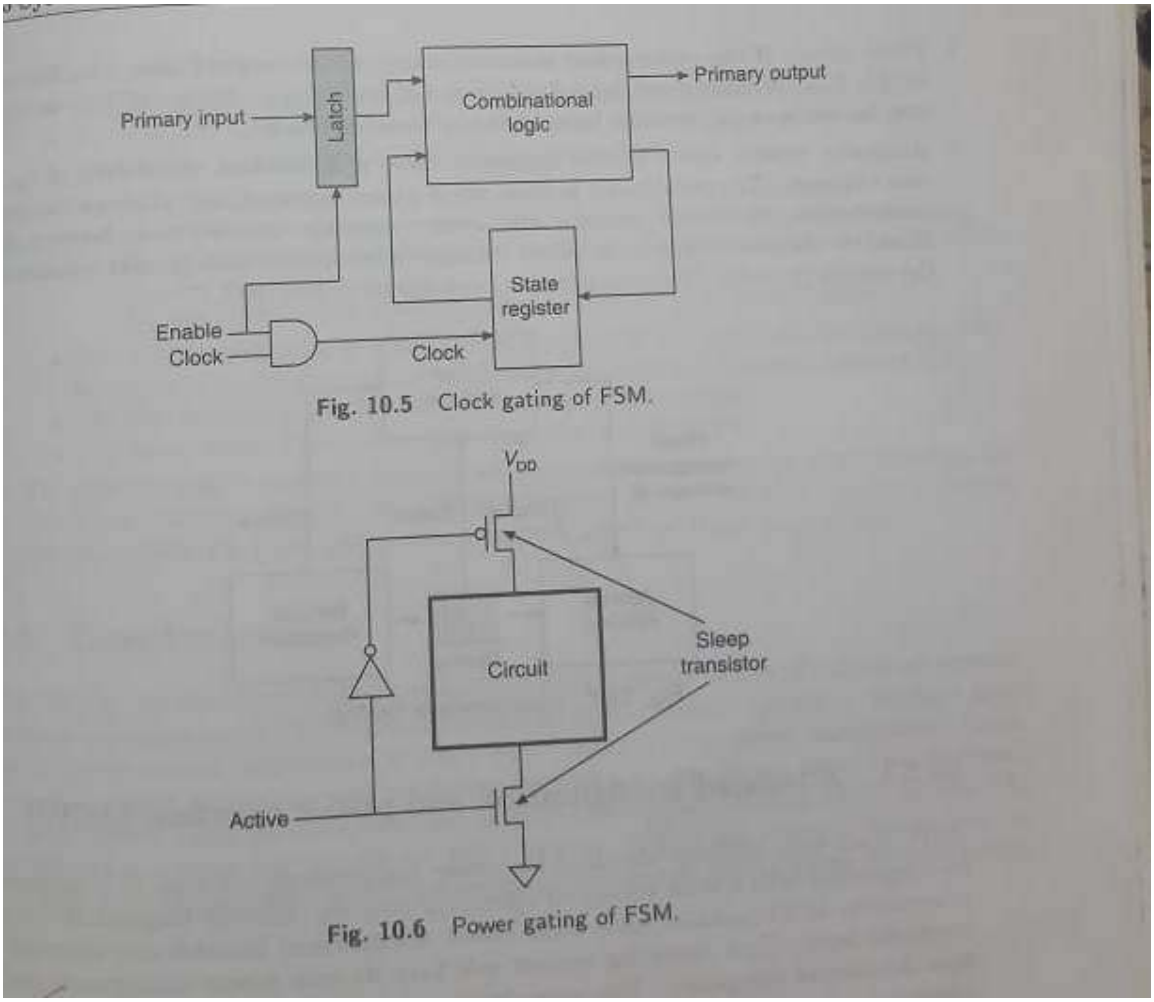


Fig. 10.5 Clock gating of FSM.

Fig. 10.6 Power gating of FSM.

System Level Power Management

The techniques discussed so far are good to the extent that they deal with local subsystems In an embedded system, consisting of various subsystems. It may be possible that all of them are not needed simultaneously to remain active. Thus, it needs a system level management policy to turn OFF and turn ON the subsystems. A suitable power management policy is needed for the following purposes

• going to a low power state takes time. The longer the duration for which we want to shutdown a system, higher is the time taken during reactivation

• Avoiding a power-down mode will cost unnecessary power

• Frequent power-down mode will affect system performance.

A naive approach may be to power-down a system whenever there is no request. This will definitely affect performance severely. A more sophisticated method is to use predictive shutdown. In this

approach, the goal is to predict the next arrival of service request and wake up the system just before that. Prediction can be made in several different ways as follows.

1. **Fixed time** : If the system does not receive any service request during an interval of length $\mathbf{T_{ON}}$ it shuts down for a fixed period of time T_{OFF} .Choice of T_{ON} and T_{OFF} may be made experimentally by studying system behavior.
2. **Analyzing System State :** In this approach, there is a constant monitoring of the service requests .The monitoring is done via a power manager that observes the system components – the service provider ,the service requestor ,and the queue between them.

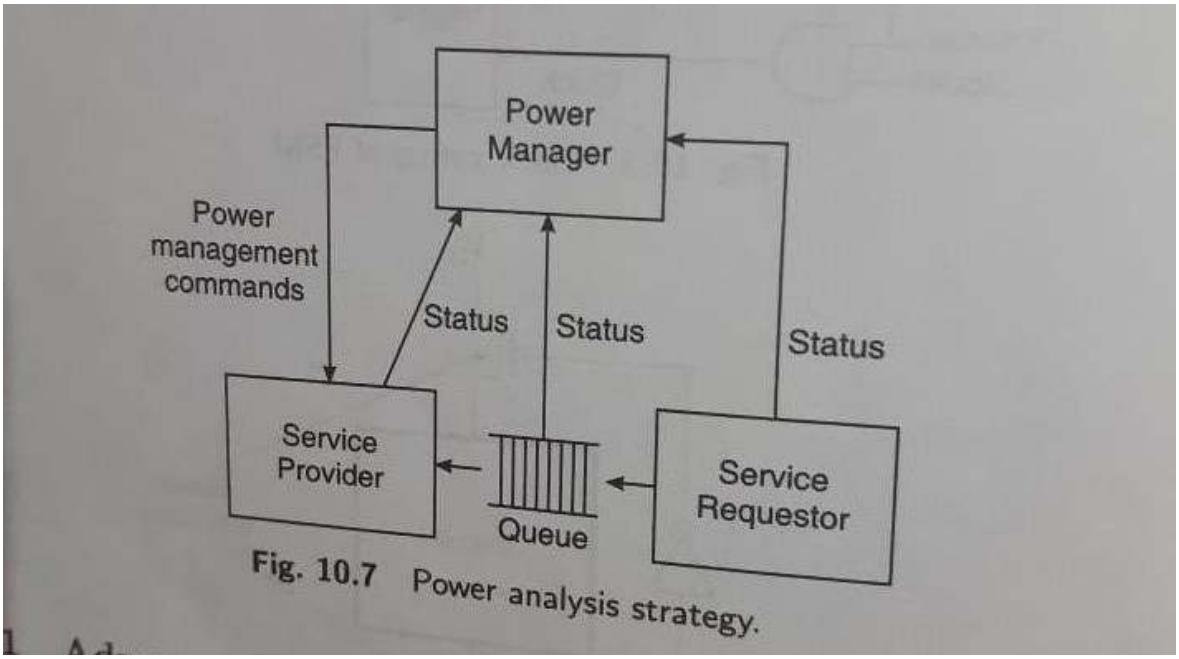


Fig. 10.7 Power analysis strategy.

MODULE-4

Hardware-Software Partitioning

Embedded systems typically consist of application-specific hardware part (containing ASIC, FPGA etc.) and software part running on general purpose processors, digital signal processors (DSPs) etc

1- Advancement on both the fronts-high level synthesis to produce custom hardware, and software development and compilation techniques to exploit the features of general processors to their limits, has ushered in a new challenge in embedded system design.

2-A compiler for embedded system cannot only be a silicon compiler (converting a specification to hardware) or a highly optimized language compiler for a target processor , it should also e able to partition the specification into a number of software and hardware modules.

3- Such practitioners must be extremely fast as they have to explore a good amount of design Space in terms of hardware, software, and communications between them.

4-The criteria to optimize may be implementation area, performance, power consumption, etc., or a mix of a number of such factors.

There are quite a few approaches to solve this partitioning problem. Some of these are as follows.

1. Integer Linear Programming (ILP)

2. Heuristic approaches, such as, Kernighan-Lin algorithm

3. Meta search techniques, like

(a) Genetic algorithm (GA)

(b) Simulated annealing (SA)

(c) Particle swarm optimization

Partitioning Using Integer Programming

Integer programming is a mathematical tool to solve constrained optimization problems. It has been used by many researchers to solve the hardware software partitioning problem as well

This section introduces a formulation of the hardware software partitioning problem using integer programming .

For this, the following definitions are needed.

the target architecture consists of an ASIC h, a set of processors P =(P1, ..., Pnp) . external memory and buses between them. The set of target architecture components is defined as

TA= {h} UP.

Extending KL-heuristic for hardware software partitioning

KL heuristic to hardware-software partitioning, the following points are to be noted

1.The objective of partitioning a task graph into hardware and software modules is to

achieve better performance. From this angle, the cut-size metric is conveniently replaced by execution time metric. Execution time of node n. represented as n.et (execution time), is given by,

$n.et = n.ict + n.ct$

n.ict is the internal computation time of node n. It may be noted that it can represent either the hardware time or the software time based upon the mapping of the node n to hardware or software respectively. n.ct encompasses the computation times of the tasks called by task n.

2. Basic k-l algorithm assumes a balanced partition. each move is a swap of a pair or nodes belonging to two partitions .thus the partitioned size are always balanced .in hw- sw partitioning we have two parts software and hardware.

3.Use better data structure for faster move selection .in k-l huristic to select the move the resulting in maximum gain ,we need to try out all possible moves and then select from them .this involves good amount of computation.

2 Partitioning Using Genetic Algorithm

. Genetic Algorithm (GA) is a stochastic search algorithm based on operations of natural genetics

. Here a fixed-sized Population of chromosomes evolves over a number of generations following the principle of natural selection.

. Each chromosome identifies a potential solution .

. A chromosome has got an associated fitness measure using the operators similar to mutation in nature, the population evolves through generations.

. In the hardware software partitioning problem, a chromosome is a bit-string. If there are n nodes in the task graph to be partitioned, a chromosome is also taken to be n-bit long with the i_{th} bit identifying the partition for the i_{th} node. It can be conveniently assumed to be o if the node is put into the hardware partition and 1 for software (or vice versa). A fixed number of such chromosomes can be created randomly to represent the initial population.

. **As the software and hardware cost metrics (like area, time delay. power of all nodes are known, for a chromosome, we can evaluate its fitness by considering the overall delay, hardware area, power, etc.**

.To evolve a new generation, the top small percentage of chromosomes are directly copied to the next generation. Rest of the population is created by two operators---crossover and mutation.

.The crossover operator selects two parent chromosomes to participate in the operation. Their parts are exchanged to create new offspring. The crossover may be single-point, multipoint, etc.

. The mutation operator may be implemented by selecting a parent chromosome and randomly complementing some of its bits (that is, changing partitions). The mutation rate can be controlled to control the rate of convergency to local or global minima.

.The main drawback of such genetic approach is the slow rate of convergence. It often requires the GA to evolve a large number of generations to converge to a solution.

.The termination criteria is often set to be "no improvement in last few generations" or a maximum number of generations for which the GA has run. The best solution at the end is taken to be the solution of the partitioning problem. To accelerate the rate of convergence, the mutation Tate can be increased, however, it mostly converges to local best solutions, rather than finding the global best.

Partitioning Using Particle Swarm

Optimization (PSO)

Particle Swarm Optimization (PSO) is a population based stochastic technique developed by Eberhart and Kennedy in 1995. The PSO algorithm is inspired by social behaviour of bird flocking or fish schooling. In PSO. the potential solutions, called particles, fly through the problem **Space by following the current optimum particles. PSO has been successfully applied in many areas, and has been found to outperform GA in Cost function and execution time.**

.Consider the following scenario: a group of birds are randomly searching for food in an area. There is only one piece of food in the area being searched. None of the birds know exactly where the food is. During each iteration, they learn via their in how far the food is. So the best strategy to find where the bird is nearest to the food.

.PSO is learnt from this bird-flocking scenario, and used to solve the optimization problems. Each single solution (particle) is perceived as a bird in the search space Each particle has a fitness value which is evaluated by the fitness function (the cost function is to be optimized and has a velocity which directs its flight. The particles fly through the following the current optimum particles.

.PSO is initialized with a group of random particles (solutions) and then searches for optima by updating through generations. During every generation (Iteration), each particle is updated by following two best values. The first one is the position vector of the best solution this particle has achieved so far. This fitness value is also stored along with the particles this position is called ***pbest***.

Another best position that is tracked by the particle swarm optimizer is the global best position, obtained so far, by any particle in the population. This hi is the current global best and is called ***gbest.***

Functional Partitioning and *Optimization*

.In the previous chapter, we have seen various strategies for distributing the tasks of an application among the available hardware and software platforms.

. This has essentially resulted in various schemes for hardware software partitioning to optimize different design goal area, delay, power, etc. However at the specification stage of a system.

. The main emphasis clean goal description of the system. understandable by the system designers. This helps in efficient management of the design in terms of its testing, validation, up gradation, modification so on.

.The issue of a good implementable of the system is not a major concern.

. As a result it is very much possible that the initial specification is not amenable to a good design and necessitates several refinements to lead to another specification that results in good implementation.

. For example, the initial specification may be given in terms of a set of user-defined procedures. These procedures identify the conceptual module that the user visualizes the system to be consisting of. There may be commonality between the procedures may be quite large or very small. Thus, taking each procedure as a task in the task-graph and trying to perform hardware software partitioning over the set may lead to poor solutions Though this initial set of procedures may act as a good starting point, the set needs to be regrouped and reconsidered for various optimizations possible.

. In this chapter, we will first look into the problem of functional partitioning of a large behavioural process into a set of procedures. Next, we will look into a set of optimizations that may be carried out over a behavioural specification to lead to better implementation.

Functional partitioning:

.The process of functional partitioning divides a given functional specification of a system into a set of sub-specifications.

. Each sub-specification corresponds to a task that is ultimately mapped onto a custom hardware or to a software module to execute on a general purpose processor. Thus, it divides a large functional specification into a set of smaller ones.

.The approach has got several advantages as enumerated below.

1. It often results in order of magnitude reduction in the runtimes for logic synthesis. This **happens because most of the heuristics used in logic synthesis work well for small- to medium-sized inputs. For large inputs, the quality of the solution degrades also the runtime increases**

disproportionately. The tools are also not linear in runtime, in the sene that.sum of runtimes for synthesizing several small processes can be much less than the run time for one large process.

2. The overall system performance may also improve, if the smaller processes can be synthesized in the custom hardware with small clock periods than the large hardware needed for the large specification and when the shorter periods outweigh the overhead of inter processor communication.

3. The partitioning also brings the scope of further parallelization between the tasks.

4. This may result in improved satisfaction of input-output and size capacity constraints on a package such as FPGA. It can significantly reduce the number of inter-package signals and may lead to fewer packages, smaller board size, reduced overall cost.

Model

.The input to the functional to the functional partitioning process is a single behavioural process X, which may for example, be a C or VHDL code.

. The process describes the behavioural of a complex system and the description consists of numerous operations, requiring many hundreds or thousands of lines of sequential program code.

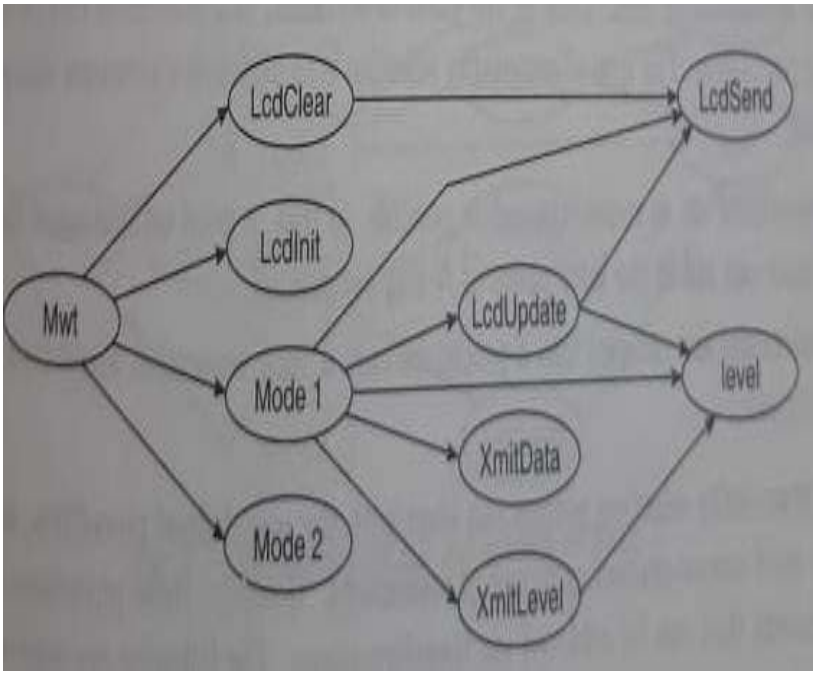
.The process X can be viewed to be consisting of a set of procedures F=(f1,f2,……fn) , with one being the main procedure which in turn calls other procedures. Any procedure can call others excepting the main procedure. A variable in the program is also treated as a simple procedure, with reads and writes being the procedure calls.

. functional partitioning of F creates a partition P consisting a set of parts {P1,P2 ... Pm. } such that every procedure fi is assigned to a single part Pi, that is, P1UP2U...UP = F and op . = for all i E {1,2, ... n} and j e {1,2,...m},

i j. Each part pj will be implemented either by a custom hardware or as a software process running on a general purpose processor.

The model used can be the call graph of the process. Figure shows an example model for a process Mwt consisting of a global variable level of type byte. It also has a number of other procedures as noted next

functional Partitioning and optimization



High-level Optimizations

In this section, we will discuss about the optimization techniques that can be applied to a process description at higher level. We will mainly look into three powerful techniques used in enhancing embedded specifications. These are,

1.Loop optimization

2.Float to fixed-point conversion

3.Data structures such as array optimizations

it can be applied to a e powerful optimization

loop optimization

.Among the various parts in aspecification loop optimization plays a vital role in improving the overall system performance .

.This is mainly because of the fact that the loop bodies are executed repeatededly within the operation of anembedded system .

.Thus a small saving in terms of delay or power in the execution of a single iteration of a loop can be very much effective in producing a good saving of the quantity.

.There have been several strategies proposed for loop optimization.

The commonly used techniques are the following

1. Loop interchange/Permutation

2. Loop splitting/peeling

3. Loop fusion/combining

4. Loop fission/distribution

5. Loop unrolling

6. Loop unswitching

7. Loop-invariant code motion

8. Loop tilling/blocking

Loop interchange/permutation

It is the process of exchanging the order of two loops .one major utility of such a transformation is to improve the cache performance for accessing array elements .For example considering the following the code sequence

for I =1 to 1000 do

for j =1to 1000 do

a [I,j] =b[I,j]+c[I,j]

in this case the array elements are accessed in the order of indices (1,1),(1,2),(1,3),……..For a row major organization of the arrays. the memory locations are accessed successively.

for I =1 to 1000 do

for j =1to 1000 do

a [I,j] =b[I,j]+c[I,j]

however it is not mandatory that interchanging of loops will always lead to better cache utilization. For example considering the following code fragment *for*

I =1 to 1000 do

for j =1to 1000 do

a [I,j] =b[I,j]+c[I,j]

For a column major organization of the b-array ,a loop interchange will bring locality into the accessing pattern of it .it will ruin the locality behavior of a-and c-arrays. Also it may not always be safe to exchange the iteration variables due to dependencies between the statements for the order in which they must execute.

Loop Splitting/Peeling

very often, in the specification of a process, we have some condition checks nested deep within a set of loops. A typical example is in the field of image filtering. The filtered value of a pixel is often computed as a weighted sum of itself and its neighbours. Though for inner pixels, all the neighbours exist, for the boundary pixels, a number of neighbours may be absent. Thus the typical processing runs over two loops as shown next.

for I =1 to num_rows do

for j = 1 to num.cols do

Check for boundaries

if boundary pixel then

do special processing

else do normal processing

In this code, the check for boundary conditions is repeated for each non-boundary pixels as well. The loop can be split as follows.

For I = 2 to num_rows -1 do

for j = 2 to num_cols -1 do

Normal processing

Special processing for all boundary pixels

A useful special case of loop splitting is loop peeling, that simplifies a loop with problematic first or few iterations separately before starting the loop. For example, consider the following code fragment

P= 10

for I =1 to 10 do

y[i] = x[i] + x[p]

p=i

Here, only for the first iteration p is equal to 10, for the rest it is i-1. Thus, the loop be modified as follows.

Y[1] = x[1] +x [10]

for i=2 to 10 do

y[i] = x[i] + x[i-1]

High-level Optimizations

Loop fusion and fission

Loop fusion/jamming is a transformation that replaces multiple loops by a single one . the reverse process is loop fission/distribution. Fission may be **advantageous if the target processor provides zero-overhead loop instructions that can be only used in**

association with small loop bodies. On the other hand, loop fusion may lead to improved cache behaviour and possibility of parallelism within the

loop body:

Loop fusion

for i=1 to n do pl) = ... for j = 1 to n do pl] =pO] + ...

Loop fission

for j = 1 to n do pl] =... pl] = pi + ...

Loop unwinding/unrolling

This is a loop transformation technique that attempts to optimize the performance of a process at the expense of its size.

The goal is to reduce speed by reducing (or eliminating) the end-of-ioop test on each iteration. Loops can be rewritten as a sequence of statements and thus eliminate the loop controller overhead. The speed gained may offset the performance degradation due to the increased size of the code, and thus result in significant saving of processing time. Moreover, if the execution of the statements of the loop in an iteration are independent of previous iterations, all these statements can be executed in parallel. Thus. loop unrolling opens up the avenue of large-scale parallelism. The following example shows the case in which the loop has been unrolled once. for j = 1 to n do p[j]

for j = 1 to n step 2 do

P[j] =

P[j + 1] = ... The number of copies of the loop is called the unrolling factor. For the above example, the unrolling factor is two. Higher unrolling factors are also possible. In the extreme case, loops can be completely unrolled, removing control overhead and branches altogether. Unrolling is normally restricted to loops with a constant number of iterations.

Loop unswitching This essentially means moving a conditional from inside a loop body to the outside, by duplicating the loop body. A version of the loop body is placed inside each of the if- and else clauses of the conditional. Since modern processors can operate fast on vectors, it can improve the overall speed of execution, though the code size is doubled, as the loop body is duplicated. The following shows the case of loop unswitching.

For i=1 to 1000 do

X[i]=x[i]+y[i]

If(w)theny[i]=0

if (w) then

for i = 1 to 1000 do

x[i]=x[i]+y[i]

y[i]= 0

else

for i = 1 to 1000 do

x[i] = x[i] + y[i]

Loop-invariant code motion

Loop-invariant code refers to the statements which can be moved outside the body of the loop without with out affecting the semantics of the process description. The process of removing such code outside is called loop-invariant code motion, hoisting, or scalar promotion. The hoisted out code is executed less often providing a speedup. The following example shows the removal of two loop-invariant code fragments outside the loop.

For i = 1 to n do

X=y+z

*A[i]=6*i+x*x*

X=y+z

*T1=x*x*

for i=1 to n do

*a[i] = 6*i + t1*

Loop tiling

It is also known as loop blocking, strip mine and interchange, unroll and jam, and super node partitioning. This optimization makes the execution of certain types of loops more efficient particularly from the cache utilization point of view. For example, for the routines that involving handling large arrays, the cache size may not be sufficient to hold enough data for a single iteration of loops. In such cases, the loop indices are divided into sub-ranges. Each sub range is processed by the iteration of inner loops. The outer loop indices change in the terms sub-ranges. A typical example of this is the matrix-vector multiplication algorithm show below.

For i=1 to N do

for j = 1 to N do

c[i] = c[i] +a[I,J] * b[J]

After a 2 x 2 tiling of the loops, the structure becomes,

for i = 1 to N step 2

for j = 1 to N step 2

for ii = i to min(i +2, N)

for jj = j to min(j+2, N)

c[i] = c[ii] + a[ii, jj]*b[jj]

B x B tiling for the full matrix multiplication.

for j = 1 to N do

for k = 1 to N do

r = X[I,k]

for j = 1 to N

Z[I,j]=z[I,j]+r*Y[K,j]

for kk = 1 to N do step B do

for j = 1 to N step B do

for i = 1 to N do

for k= kk to min(kk+B-1,N) do

r=x[I,k]

for j= jj to min(jj+B-1,N) do

z[i.j]=z[I,j]+r*y[k,j]

Hardware Software *Cosimulation*

.simulation is one of the most effective measures to ensure system correctness, since an embedded system design consists of interacting software and hardware components it is very much essential that a simulation of the system includes simulating both, along with interactions.

. cosimulation has been done late in the process, after the hardware has been deemed to be mostly working, that is stable, such that it can interact properly software.

. Software developers used to develop their code with limited facility to absence of the hardware platform. Painful integration efforts were needed towards the design cycle, and minor miscommunication became major design flaws.

. Most of these problems would be patched up in software at the cost of performance or even expensive of hardware/software.

The typical usage of cosimulation during code sign are as follows .

• Verification of system specification to check whether the specification is as per the system designer.

•Verification of system implementation.

• Performance estimation to be used during system partitioning.

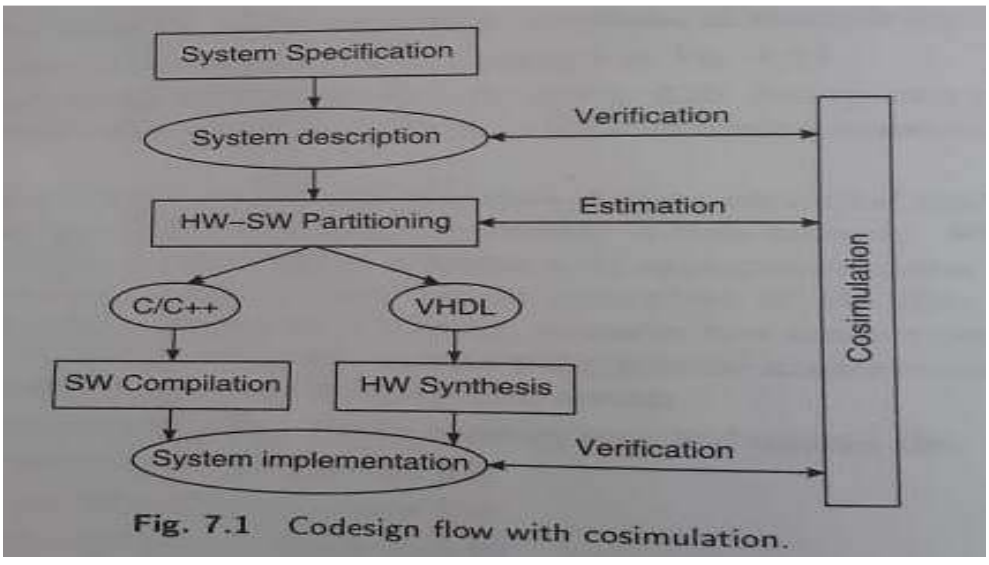


Fig. 7.1 Codesign flow with cosimulation.

Co simulation Approaches

There can be several approaches to hardware Software cosimulation ranging from very detailed gate-level model) to very abstract (such as instruction-level model) while the hardware simulation can be performed using Hardware Description Language (HDL)simulators soft ware is typically simulated using Instruction Set Simulator (ISS). A very simplistic approach for co simulation could be as follows.

. Use an HDL model of the microprocessor that will run the software part.

. Using HDL models of specific hardware synthesized for the hardware part.

.Integrating all models to perform a full simulation.

This approach is known as homogeneous simulation as all the modules are simulated using the same simulator.

. *A more generic model is the heterogeneous cosimulation working on the following principle.*

.Use an instruction set simulator model of microprocessor that runs the software.

. Use HDL model for simulating the hardware.

. Create communication between the simulators.

. Simulators run separately excepting when transferring data.

In some sense, a heterogeneous simulator networks different types of simulators as shown

Heterogeneous simulators incur high overhead due to the voluminous data transfer between hardware and software simulators. The percentage of time that the software accesses

hardware is called the hardware density of simulation. Heterogeneous simulation affords better ability to match a task with a tool. A typical example of such a simulator is Synopsis's Eaglei that lets the hardware run in many simulators and the software on the native PC/workstation or in instruction-set simulator (ISS). For the hardware portion, HDL simulators are generally used. However, simulation runs at a much slower rate. Thus, it may be difficult to catch many of the practical problems. For such a situation, in the absence of real hardware, a prototype realized on FPGA may be used. Such a concept is known as emulation. It is a special simulation environment with hardware. The following are some of its features.

. Runs whole design as compared to HDL simulators where only a part may be simulated.

• Expensive compared to pure software simulation.

• may run at 10% of real-time, thus much closer to the actual hardware speed. The software simulation runs rather slowly.

• allows designers of large products to find problems that cannot be found in simulator.

• Real devices can be attached with the FPGA based hardware emulators. This helps in **visualizing entire system and check its functional correctness.**

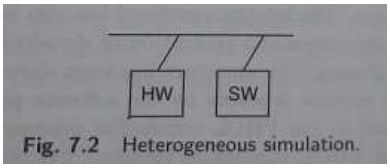


Fig. 7.2 Heterogeneous simulation.

Heterogeneous simulation