Q.1 a. Describe the domestic and industrial application of embedded system.

Answer:

(a) Application IN INDUSTRY:-

→ Telecom
→ Automative applications
→ Domestic applicaton
→ Robotics
→ Aerospace applications
→ Medical equipment
→ Defense systems.

Application In Domestic

Printers.
Satellite phones
Scanners.
Televisions
Temperatur controller.
Tv-set up Boxes.

b. Show the contents of the PSW register of after execution of the following instruction.
MOV A, #0BFH
ADD A, #1BH

Answer:

b) MOV A, #0BFH
ADD A, #1BH

PSW 7	PSW 6	PSW 5	PSW 4	PSW 3	PSW 2	PSW 1	PSW 0
CY	AC	F0	RS_1	RS_0	OV	–	P

Sol

```
  BF      10111111
+ 1B      00011011
  DA      11011010
          D7 D6 D5 D4 D3 D2 D1 D0
```

PSW7 : CY = 0 Since there is no carry beyond the D_7 bit.

PSW6 : AC = 1 Since there is a carry from the D_3 to the D_4 bit.

PSW5 : F0 = unused hence 0.

PSW4 : Register bank selector bit RS1 = 0 Since by default, Bank 0 is selected.

PSW3 : Register bank selector bit RS0 = 0 Since by default, Bank 0 is selected

PSW2 : OV = 0 No carry from D_6 to D_7

PSW1 : No used hence 0

PSW0 : P = 1 Since there is an odd No of ones (1's) in accumulator

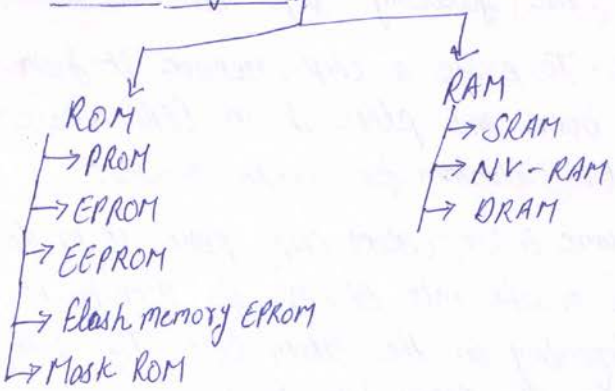
The contents of the PSW is 0100 0001 → 4 1

PSW = 41H

c. Explain briefly different types of memories used in 8051.

Answer:

Different types of memories used in 8051



ROM (Read only memory)

ROM is also called nonvolatile memory that does not lose its contents when the power is turned off.

PROM (Programmable ROM) and OTP

PROM refers to the kind of ROM that the user can burn information into. In other words, PROM is a user-programmable memory. For every bit of the PROM, there exists a fuse. PROM is programmed by blowing the fuses. If the information burned into PROM is wrong, that PROM must be discarded since its internal fuses are blown permanently. For this reason, PROM is also referred to as OTP. Programming ROM, also called burning ROM, requires special equipment called a ROM burner or ROM programmer.

EPROM (erasable Programmable ROM) and UV-EPROM

In EPROM, one can program the memory chip and erase it thousands of times. This is especially necessary during development of the prototype of a microprocessor-based project. A widely used EPROM is called UV-EPROM, where UV stands for ultraviolet. The only problem with UV-EPROM is that erasing its contents can take up to 20 minutes. All UV-EPROM chips have a window through which the programmer can shine ultraviolet (UV) radiation to erase its contents. For this reason, EPROM is also referred

to as UV-erasable EPROM or simply UV-EPROM.
To program a UV-EPROM chip, the following steps must be taken:

(1) Its contents must be erased. To erase a chip, remove it from its socket on the system board and place it in EPROM erasure equipment to expose it to UV radiation for 15-20 minutes.

(2) Program the chip. To programe a UV-EPROM chip, place it in the ROM burner. To burn code or data into EPROM, the ROM burner uses 12.5 volts or higher, depending on the EPROM type. This voltage is referred to as Vpp in the UV-EPROM data sheet.

(3) Place the chip back into its socket on the system board.

EEPROM (electrically erasable programmable ROM)

EEPROM has several advantages over EPROM, such as the fact that its method of erasure is electrical and therefore instant, as opposed to the 20 minute erasure time required for UV-EPROM. In addition, in EEPROM one can select which byte to be erased, in contrast to UV-EPROM, in which the entire contents of ROM are erased.
However, the main advantage of EEPROM is that one can program and erase its contents while still in the system board. It does not require an external erasure and programming device.

Flash Memory EPROM:

The major difference between EEPROM and flash memory is that when flash memory's contents are erased, the entire device is erased, in contrast to EEP-ROM, where one can erase a desired section or byte. Although in some flash memories recently made available the contents are divided into blocks and the erasure can be done block by block, unlike EEPROM, flash memory has no byte erasure option.

Mask Rom:

Mask Rom refers to a kind of Rom in which the contents are programmed by the IC manufacturer. In other words, it is not a user-programmable Rom. The term mask is used in IC fabrication. Since the process is costly, mask Rom is used when the needed volume is high and it is absolutely certain that the contents will not change.

RAM (random access memory)

RAM memory is called volatile memory since cutting off the power to the IC results in the loss of data. Sometimes RAM is also referred to as RAWM (read and write memory), in contrast to Rom, which cannot be written to. There are three types of RAM: Static RAM, NV-RAM and dynamic RAM. Each is explained separately.

SRAM (Static RAM)

Storage cell in static RAM memory are made of flip-flop and therefore do not require refreshing in order to keep their data. This is in contrast to DRAM.

The problem with the use of flip-flop and therefore each cell requires at least 6 transistors to build, and the cell holds only 1 bit of data.

NV-RAM (nonvolatile RAM)

Whereas SRAM is volatile. there is a new type of nonvolatile RAM called NV-RAM.

(1) It uses extremely power-efficient SRAM cells built out of cmos.
(2) It uses an internal lithium battery as a backup energy source.

DRAM (Dynamic RAM)

The major advantages of DRAM high density, cheaper cost per bit, and lower power consumption per bit.

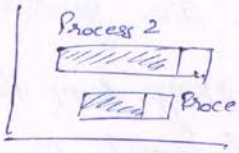
The disadvantage is that it must be refreshed periodically because the capacitor cell loses it charge; furthermore, while DRAM is being

d. Explain Scheduling Algorithms of RTOS.

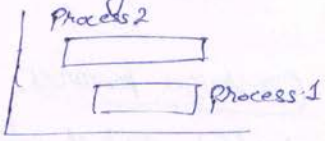
Answer:

(d) Scheduling Algorithms of RTOS

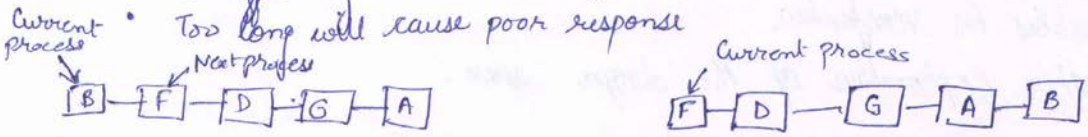
Shortest Job First Scheduling : The process that requests less CPU burst is allocated the CPU first
- Case where SJF request in failure



Earliest Deadline First Scheduling : The process that have earlier deadline is having higher priority. Priority is determined by deadline
- Case where EDF results in failure.

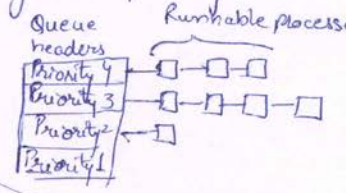


Round Robin Scheduling
- Each process is assigned time interval
- If process is still running at end of quantum CPU is preempted and given to another process
- Issue - length of quantum
  - Too short will cause too many process switches and lowers the CPU efficiency
  - Too long will cause poor response



Priority Scheduling - Each process is assigned priority and runable process with highest priority is allowed to run
Priority assign algorithms are classified as fixed priority, dynamic priority or mixed priority

Multilevel Queue Scheduling Ready queue is partitioned into separate queues: foreground and background
- Each queue has its own scheduling algorithm



Rate Monotonic Scheduling RMS is optimal fixed priority algorithm
The shorter period, the higher priority. If a task set cannot be scheduled using RM algorithms, it cannot be scheduled using any fixed priority algorithm.
The deadline of m processes can be met if processor utilization

$$U = \sum_{i=1}^{n} (C_i / P_i) \leq n\left(2^{\frac{1}{n}} - 1\right)$$

U = Processor utilization
n = no. of tasks in system
C_i = computation time
P_i = Process

e. What is Pipelining? Explain with an example.

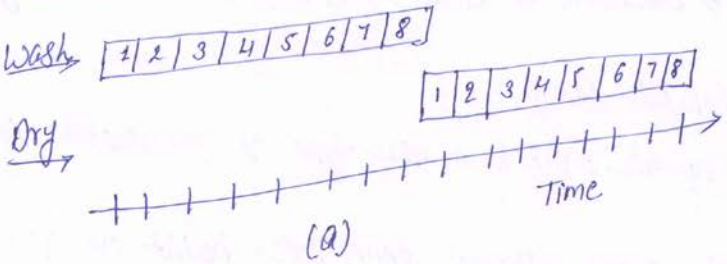
Answer:

(c)

Pipelining

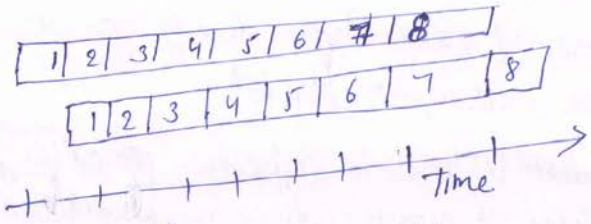
Pipelining is a common way to increase the instruction throughput of a microprocessor. We first make a simple analogy of two people approaching the chore of washing and drying eight dishes. In one approach, the first person washes all eight dishes, and then the second person dries all eight dishes. Assuming 1 minute per dish per person, this approach requires 16 minutes. The approach is clearly inefficient since at any time only one person is working and the other is idle.

Obviously, a better approach is for the second person is being drying the first dish immediately after it has been washed. This approach requires only 9 minutes - 1 minute for the first dish to be washed and then 8 more minutes until the last dish is finally dry. We refer to this latter approach as pipelined.

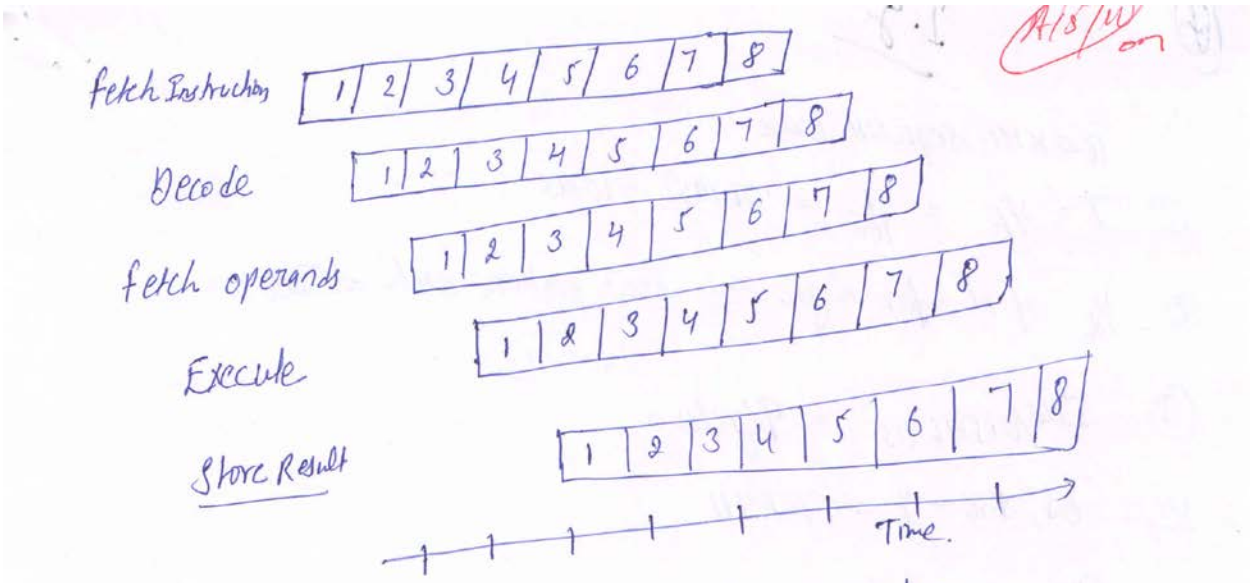


(a)

Non pipelined dish cleaning.



(b) Pipelined dish cleaning.



(c) Pipelined Instruction execution.

Each dish is like an instruction and the two task of washing and drying are like the five stages. By using a separate for each stage, we can pipeline instruction execution. after the instruction fetch unit fetches the first Instruction, the decode unit decodes it while the Instruction fetch unit simultaneously fetches the next instruction. The idea of pipelining is illustrated.

Pipelining work well, instruction execution must be decomposable into roughly equal length stages and instructions should each require the same number of cycles.

f. Explain development environment and debugging techniques.

Answer:

In this section we take a step back from the platform and consider how it is used during design. We first consider how we can build an effective means for programming and testing an embedded system using hosts. We then see how hosts and other techniques can be used for debugging embedded systems.

4.6.1 Development Environments

A typical embedded computing system has a relatively small amount of everything, including CPU horsepower, memory, I/O devices, and so forth. As a result, it is common to do at least part of the software development on a PC or workstation known as a ***host*** as illustrated in Figure 4.26. The hardware on which the code will finally run is known as the ***target***. The host and target are frequently connected by a USB link, but a higher-speed link such as Ethernet can also be used.

The target must include a small amount of software to talk to the host system. That software will take up some memory, interrupt vectors, and so on, but it should

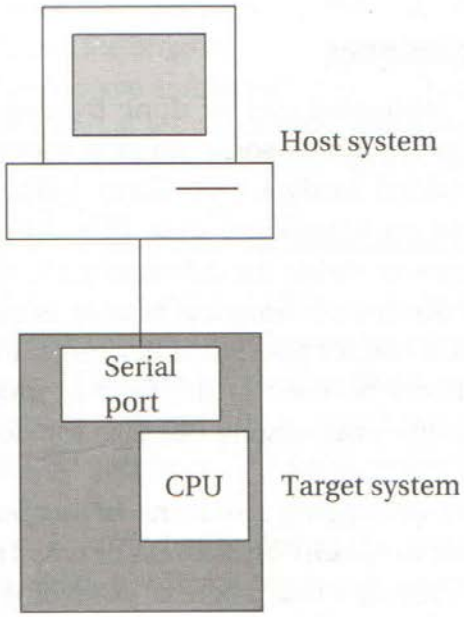


FIGURE 4.26

generally leave the smallest possible footprint in the target to avoid interfering with the application software. The host should be able to do the following:

- load programs into the target,
- start and stop program execution on the target, and
- examine memory and CPU registers.

A ***cross-compiler*** is a compiler that runs on one type of machine but generates code for another. After compilation, the executable code is downloaded to the embedded system by a serial link or perhaps burned in a PROM and plugged in. We also often make use of host-target debuggers, in which the basic hooks for debugging are provided by the target and a more sophisticated user interface is created by the host.

A PC or workstation offers a programming environment that is in many ways much friendlier than the typical embedded computing platform. But one problem with this approach emerges when debugging code talks to I/O devices. Since the host almost certainly will not have the same devices configured in the same way, the embedded code cannot be run as is on the host. In many cases, a ***testbench program*** can be built to help debug the embedded code. The ***testbench*** generates inputs to simulate the actions of the input devices; it may also take the output values and compare them against expected values, providing valuable early debugging help. The embedded code may need to be slightly modified to work with the testbench, but careful coding (such as using the #ifdef directive in C) can ensure that the changes can be undone easily and without introducing bugs.

4.6.2 Debugging Techniques

A good deal of software debugging can be done by compiling and executing the code on a PC or workstation. But at some point it inevitably becomes necessary to run code on the embedded hardware platform. Embedded systems are usually less friendly programming environments than PCs. Nonetheless, the resourceful designer has several options available for debugging the system.

The serial port found on most evaluation boards is one of the most important debugging tools. In fact, it is often a good idea to design a serial port into an embedded system even if it will not be used in the final product; the serial port can be used not only for development debugging but also for diagnosing problems in the field.

Another very important debugging tool is the ***breakpoint***. The simplest form of a breakpoint is for the user to specify an address at which the program's execution is to break. When the PC reaches that address, control is returned to the monitor program. From the monitor program, the user can examine and/or modify CPU registers, after which execution can be continued. Implementing breakpoints does

not require using exceptions or external devices. Programming Example 4.1 shows how to use instructions to create breakpoints.

Programming Example 4.1

Breakpoints

A breakpoint is a location in memory at which a program stops executing and returns to the debugging tool or monitor program. Implementing breakpoints is very simple—you simply replace the instruction at the breakpoint location with a subroutine call to the monitor. In the following code, to establish a breakpoint at location 0x40c in some ARM code, we've replaced the branch (B) instruction normally held at that location with a subroutine call (BL) to the breakpoint handling routine:

```
0 x 400 MUL r4,r4,r6          0 x 400 MUL r4,r4,r6
0 x 404 ADD r2,r2,r4    ⟶    0 x 404 ADD r2,r2,r4
0 x 408 ADD r0,r0,#1          0 x 408 ADD r0,r0,#1
0 x 40c B loop                0 x 40c BL bkpoint
```

When the breakpoint handler is called, it saves all the registers and can then display the CPU state to the user and take commands.

To continue execution, the original instruction must be replaced in the program. If the breakpoint can be erased, the original instruction can simply be replaced and control returned to that instruction. This will normally require fixing the subroutine return address, which will point to the instruction after the breakpoint. If the breakpoint is to remain, then the original instruction can be replaced and a new temporary breakpoint placed at the next instruction (taking jumps into account, of course). When the temporary breakpoint is reached, the monitor puts back the original breakpoint, removes the temporary one, and resumes execution.

The Unix *dbx* debugger shows the program being debugged in source code form, but that capability is too complex to fit into some embedded systems. Very simple monitors will require you to specify the breakpoint as an absolute address, which requires you to know how the program was linked. A more sophisticated monitor will read the symbol table and allow you to use labels in the assembly code to specify locations.

Never underestimate the importance of LEDs in debugging. As with serial ports, it is often a good idea to design a few to indicate the system state even if they will not normally be seen in use. LEDs can be used to show error conditions, when the code enters certain routines, or to show idle time activity. LEDs can be entertaining as well—a simple flashing LED can provide a great sense of accomplishment when it first starts to work.

When software tools are insufficient to debug the system, hardware aids can be deployed to give a clearer view of what is happening when the system is running. The ***microprocessor in-circuit emulator (ICE)*** is a specialized hardware tool that can help debug software in a working embedded system. At the heart of an

in-circuit emulator is a special version of the microprocessor that allows its internal registers to be read out when it is stopped. The in-circuit emulator surrounds this specialized microprocessor with additional logic that allows the user to specify breakpoints and examine and modify the CPU state. The CPU provides as much debugging functionality as a debugger within a monitor program, but does not take up any memory. The main drawback to in-circuit emulation is that the machine is specific to a particular microprocessor, even down to the pinout. If you use several microprocessors, maintaining a fleet of in-circuit emulators to match can be very expensive.

The ***logic analyzer*** [Ald73] is the other major piece of instrumentation in the embedded system designer's arsenal. Think of a logic analyzer as an array of inexpensive oscilloscopes—the analyzer can sample many different signals simultaneously (tens to hundreds) but can display only 0, 1, or changing values for each. All these logic analysis channels can be connected to the system to record the activity on many signals simultaneously. The logic analyzer records the values on the signals into an internal memory and then displays the results on a display once the memory is full or the run is aborted. The logic analyzer can capture thousands or even millions of samples of data on all of these channels, providing a much larger time window into the operation of the machine than is possible with a conventional oscilloscope.

A typical logic analyzer can acquire data in either of two modes that are typically called ***state*** and ***timing modes***. To understand why two modes are useful and the difference between them, it is important to remember that an oscilloscope trades reduced resolution on the signals for the longer time window. The measurement resolution on each signal is reduced in both voltage and time dimensions. The reduced voltage resolution is accomplished by measuring logic values (0, 1, x) rather than analog voltages. The reduction in timing resolution is accomplished by sampling the signal, rather than capturing a continuous waveform as in an analog oscilloscope.

State and timing mode represent different ways of sampling the values. Timing mode uses an internal clock that is fast enough to take several samples per clock period in a typical system. State mode, on the other hand, uses the system's own clock to control sampling, so it samples each signal only once per clock cycle. As a result, timing mode requires more memory to store a given number of system clock cycles. On the other hand, it provides greater resolution in the signal for detecting glitches. Timing mode is typically used for glitch-oriented debugging, while state mode is used for sequentially oriented problems.

The internal architecture of a logic analyzer is shown in Figure 4.27. The system's data signals are sampled at a latch within the logic analyzer; the latch is controlled by either the system clock or the internal logic analyzer sampling clock, depending on whether the analyzer is being used in state or timing mode. Each sample is copied into a vector memory under the control of a state machine. The latch, timing circuitry, sample memory, and controller must be designed to run at high speed

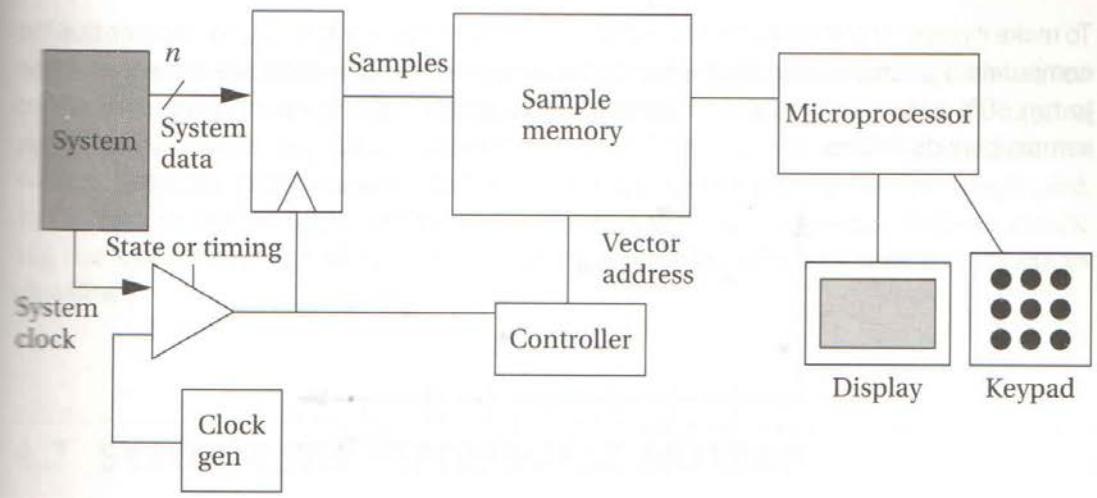


FIGURE 4.27
Architecture of a logic analyzer.

since several samples per system clock cycle may be required in timing mode. After the sampling is complete, an embedded microprocessor takes over to control the display of the data captured in the sample memory.

Logic analyzers typically provide a number of formats for viewing data. One format is a timing diagram format. Many logic analyzers allow not only customized displays, such as giving names to signals, but also more advanced display options. For example, an inverse assembler can be used to turn vector values into microprocessor instructions.

The logic analyzer does not provide access to the internal state of the components, but it does give a very good view of the externally visible signals. That information can be used for both functional and timing debugging.

4.6.3 Debugging Challenges

Logical errors in software can be hard to track down, but errors in real-time code can create problems that are even harder to diagnose. Real-time programs are required to finish their work within a certain amount of time; if they run too long, they can create very unexpected behavior. Example 4.2 demonstrates one of the problems that can arise.

g. Generate a frequency of 100 KHz on pin p2.3. Use Timer 1 in mode 1 assume XTAL of 22 MHz. (7×4)

Answer:

100 KHz Square wave

① $T = 1/f = \frac{1}{100} = .01\ ms = 10\ \mu s$

② 1/2 of it for high and low portions each = 5 µs

③ 5 µs / 0.5426 µs = 9 cycles.

④ 65,536 − 9 = FFF7H.

Code

```
        MOV TMOD, #10H      ; Timer 1, Mode 1
BACK:   MOV TL1, #0F7H      ; TL1 = F7H
        MOV TH1, #0FFH      ; TH1 = FFH
        SETB TR1            ; Start Timer 1
AGAIN:  JNB TF1, AGAIN      ; Wait for timer rollover.
        CLR TR1             ; Stop Timer 1
        CPL P2.3            ; Complement P2.3
        CLR TF1             ; Clear timer flag.
        SJMP BACK           ; reload timer.
```

Q.2 a. What are the criteria for selection of processor for use in an embedded system? (6)
Answer:

Sol. Q.2 (A) 15/10

The embedded system designer must select a µp for use in an embedded system. The choice of a processor depends on technical and non technical aspects. From technical perspective, one must choose a processor that can achieve that desired speed within certain power, size and cost constraints. Non technical aspects may include prior experties with a processor and its development environment, special licensing arrangment and so on.

Speed is a particularly difficult processor aspect to measure and compare. We could compare processor clock speed, but the number of instructions per second. but the clock cycle may differ greatly among processor We could instead compare instructions per second, but the complexity of each instruction may also differ greatly among processors. For example, one processor may require 100 instruction while another processor may require 300 instruction to perform the same computation.

Processor	Clock Speed	Peripherals	Bus Width	MIPS	Power	Price
Intel PIII	1GHz	2x16K L1, 256K L2, MMX	32	~900	97W	$900
Intel 8051	12 MHz	4K ROM, 128 RAM, 32 I/O Timer, UART	8	~1	0.2W	$7

One attempt to provide a mean for a faire comparision is the Dhrystone benchmark. A benchmark is a program intended to be run on different processors to compare their performance. It focuses on exercising a processor's integer arithmetic and string-handling capabilities. Its current version is written in C and is in the public domain. Because most processors can execute it in milli second, it is typically thousand of times, and thus a processor is said to be able to execute so many Dhrystones per second.

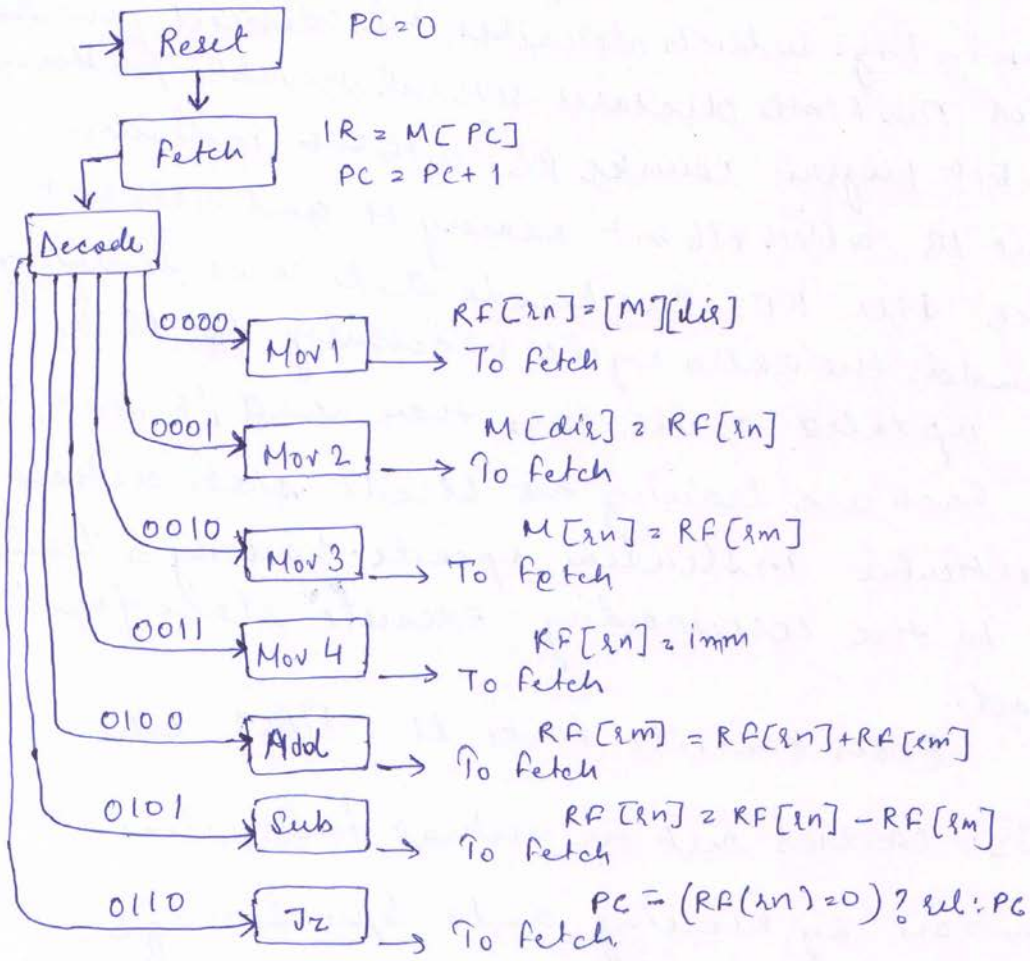
Another commonly used speed comparison unit, which happens to be based on the Dhrystone, is MIPS. One might think that MIPS simply means millions of instructions per second, but actually the common use of the term is based on a somewhat more complex notion.

The use and validity of benchmark data is a subject of great controversy. There is also a clear need for benchmarks that measure performance of embedded processor.

b. Design a Finite State Machine using a simple microprocessor. (6)

Answer:

2 b) Design a finite state m/c using a simple µP. A 15/10 Jan



A General Purpose processor is really just a single-processor whose purpose is to process instructions stored in a program memory. Therefore, we can design a general purpose processor using the single purpose processor design technique described in, while real µP intended for mass production are more commonly designed using custom method rather than the general technique of dissection, using the general technique here may prove a useful exercise that will illustrate the basic unity b/w single purpose & General Purpose processor.

Suppose we want to design a general purpose processor. We can begin by creating the FSMD shown in fig. which describes the desired processor behavior. The FSMD declares several variable for storage a 16-bit program counter PC, a 16 bit instruction register IR. a 64K x 16 bit memory M, and 16x16 bit register file RF. The decode state does nothing but adds the extra cycle necessary for IR to get updated so we can then read it on an arc. Each arc leaving the decode state detects a particular instruction opcode, causing a transition to the corresponding execute state for the opcode.

Each execute state, like MOV I add and JZ, carries out the actual instruction operation by moving data b/w storage devices, modifying data or updating PC.

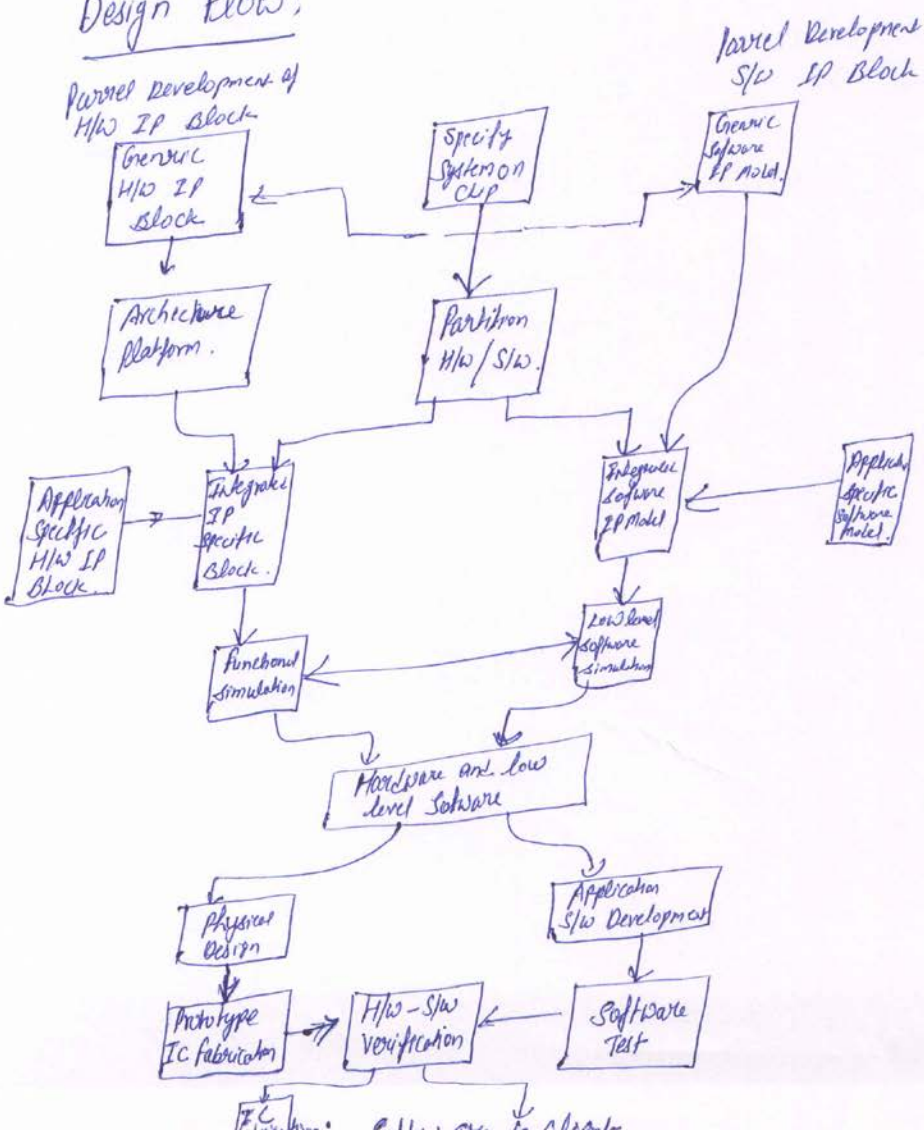
c. Give the features of SoC design. **(6)**

Answer:

Q.C feature of Soc Design:

Soc (System on chip) is an integrated circuit (IC) that integrates all components of a computer or other electronic system into a single chip. It may contain digital, analog, mixed-signal, and offe radio-frequency functions. all on a single chip substrate. Socs are very common in the mobile electronics market because of their low power consumption. A typical application is in the area of embedded system.

Design flow:



Q.3 a. What are the different types of ROM? Explain read/write mechanism of EEPROM. (6)

Answer:

(a) Types of Rom

Memories in the Rom family are distinguished by the methods used to write new data to them (usually called programming or burning) and the number of times they can be rewritten.

This classification reflects the evolution of Rom device from hard-wired to one-time programmable to erasable and programmable. The common feature across all these devices is their ability to retain data and programs forever, even when power is removed.

① The very first Roms were hard wired devices that contained a preprogrammed set of data or instructions. The contents of the Rom had to be specified before chip production, so the actual data could be used to arrange the transistors inside the chip. Hardwired memories are still used, though they are now called masked Roms to distinguish them from other type of ROM. The main advantage of a masked Rom is a low production cost.

The cost is low when hundreds of thousands of copies of the same Rom are required.

② Another types of Rom is the programmable ROM (PROM), which is purchased in an unprogrammed state. If you were to look at the contents of an unprogrammed Prom, you would see that all the bits are 1s. The process of writting your data to the Prom involves a special piece of equipment called a device programmer, which writtes data to the device by applying a higher-than normal voltage to special input pin of the chip. Once a Prom has been Programmed in this way its contents can never changed. If the code or data stored in the Prom must be changed, the chip must be discarded

and replaced with a new one. As a result, PROMs are also known as one-time programmable (OTP) devices.

(3) EPROM (erasable and Programmable ROM) is programmed in exactly the same manner as a PROM. However, EPROMs can be erased and reprogrammed repeatedly. To erase in EPROM, simply expose the device to a strong source of ultraviolet light. By doing this you essentially reset the entire chip to its initial-unprogrammed state.

The erasure time of an EPROM can be anything from 10 to 45 minutes which can make soft-ware debugging a slow proces.

Though more expensive than PROMs, their ability to be reprogrammed made EPROMs a common feature of the embedded software development and testing process many years. It is now relatively rare to see EPROMs used in embedded systems, as they have been supplanted by newer technologies

Read/Write Mechanism of EEPROM:

EEPROM is internally similar to an EPROM, but with the erase operation accomplished electrically. Additionaly, a single byte within an EEPROM can be erased and rewritten. One written, the new data will remain in the device forever - or at least until it is electrically erased. One tradeoff for this improved functionality is higher cost; another is that typically EEPROM is good for 10,000 to 100,000 write cycles.

EEPROMs are available in a standard parallel interface as well as a serial interface. In many design, the inter-IC (I^2C) or serial Peripheral Interface (SPI) buses are used to communicate with serial EEPROM devices.

b. Discuss common memory problem and possible solutions. (6)

Answer:

(b) Common memory problems and solutions. A15(W) Jan

A more common source of memory problems is the circuit board. Typical circuit board problems are:-

1. Problems with the wiring b/w the processor and memory devices.
2. Missing memory chip.
3. Improperly inserted memory chips.

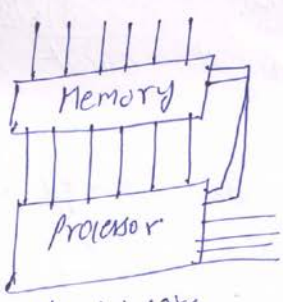
These are the problems that a good memory test algorithm should be able to detect. Such a test should also be able to detect catastrophic memory failures without specifically looking for them. So let's discuss circuit board problems in more detail.

Electrical Wiring problems:

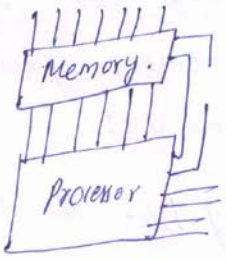
An electrical wiring problem could be caused by an error in design or production of the board or as the result of damage received after manufacture. Each of the wires that connect the memory device to the processor is one of three types.

- Address signal
- Data signal
- Control signal.

The address and data signals select the memory location and transfer the data respectively. The control signals tell the memory device whether the processor wants to read or write the location and precisely when the data will be transferred. Unfortunately one or more of these wire could be improperly routed or damaged in such a way that it is either shorted or open. shorting is often caused by a bit of solder splash, whereas an open wire could be caused by a broken trace. Both cases are illustrated in fig.



Shorted wire.



open wire.

Problems with the electrical connections to the processor will cause the memory device to behave incorrectly. Data might be corrupted when it's stored, stored at the wrong address, or not stored at all. Each of these symptoms can be explained by wiring problems on the data, address, and control signals, respectively.

If the problem is with a data signal, several data bits might appear to be "stuck together". Similarly, a data bit might be either "stuck high" (always 1) or "stuck low" (always 0). These problems can be detected by writing a sequence of data values designed to test that each data pin can be set to 0 and 1, independently of all the others.

In an address signal has a wiring problem, the contents of two memory locations might appear to overlap. In other words, data written to one address will instead overwrite the contents of another address. This happens because an address different from the one selected by the processor.

Another possibility is that on of the control signal is shorted or open. Although it is theoretically possible to develop specific tests for control signal problems, it is not possible to describe a general test that covers all platforms.

A 15/16

Missing memory chips

A missing memory chip is clearly a problem that should be detected. Unfortunately because of the capacitive nature of unconnected electrical wires, some memory tests will not detect this problem. for example, suppose you decided to use the following test algorithm: Write the value 1 to the first location in memory, verify the value by reading it back, write 2 to the second location, verify the value, write 3 to the third location, verify and so on. Because each read occurs immediately after the corresponding write, it is possible that the data read back represents nothing more than the voltage remaining on the data bus from the previous write. If the data is read back quickly, it will appear that the data bus from the previous write. If the data is read quickly, it will appear that the data has been correctly stored in memory, even though there is no memory chip at the other end of the bus!

To detect a missing memory chip, a better test must be used. Instead of performing the verification read immediately after the crossponding write, perform several consecutive writes followed by the same number of consecutive reads. for example Write the value 1 to the first location, write the value 2 to the second location, write the value 3 to the third location, and then verify the data at the first location, the second location, and so on. If the data values are unique, the missing chip will be detected: the first value read back will correspond to the last value written (3) rather than to the first (1)

Improperly inserted chips.

If a memory chip is present but improperly inserted, some pins on the memory chip will either not be connected to the circuit board at all or will be connected at the wrong place. These pins will be part of the data bus, address bus, or control wiring. The system will usually behave as though there is a wiring problem or a missing chip. So as long as you test for wiring problems and missing chips, any improperly inserted chips will be detected automatically.

Before going on, Lets quickly review the types of memory problems we must be able to detect. Memory chip only rarely have internal errors, but if they do, they are typically catastrophic in nature and should be detected by any test. A more common source of problems is the circuit board, where a wiring problem can occur or a memory chip might be missing or improperly inserted. other memory problems can occur, but the ones described here are the most common and also the simplest to test in a generic way.

c. Give the issues that need to be considered when upgrading software using flash memory. (6)

Answer:

(c) (4)

Issues that Need to be considered when upgrading software using flash memory.

Flash memory offers advantages over other types of memory. Systems with flash memory can be updated in the field to incorporate new features or bug fixes discovered after the product has been shipped. This can eliminate the need to ship the unit back to the manufacturer for software upgrades. There are several issues that need to be considered when upgrading software for units in the field.

Limit downtime :-

The timing of the upgrade should take place during downtime. Since The unit will Probably not be able to function at its full capacity during the upgrade, you need to make sure that the unit is not performing a critical task. The customer will have to dictate the most convenient time.

Power failure

How will the unit recover should power be removed while the upgrade is taking place ? If only a few bytes of the application image have been programmed into flash when the power is removed, you need a way to determine that an error occured and prevent that code from executing. A solution may be to include a loader that cannot be erased because it resides in protected flash sectors. One of the boot tasks for the loader is to check the flash memory for a valid application image. If a valid image is not present, the loader needs to know how to get a valid image onto the board, via serial port, network, or some other means.

Another solution for power failures may be to include a flash memory device that is large enough to store two application images. The current Image and the old Image. When new firmware

is available, the old image is overwritten with the new software; the current image is left alone. Only after the image has been programmed properly and verified does it become the current image. This technique ensures that the unit always has a valid application image to execute should something bad happen during the upgrade procedure.

Upgrade Code execution:-

From which memory chip will the software execute during the erase and programming of the new software? The software that downloads the image may be able to run from flash memory; however, the code to erase and reprogram a flash chip might need to be run from another memory device.

Device timing requirements:-

It is important to understand the timing requirements of the program and erase cycles for the particular flash device. It is best to make sure all data is present before starting the programming cycle. You wouldn't want to start programming the device and then be caught waiting for the rest of the new software to come in over a network connection. The device may have timing limits for program and erase cycles that cause the device to revert back to read mode if these limits are exceeded. The flash device driver would fail to write the data if this occurs.

Software image validity: It is important to validate the image that is written into the flash. This will ensure that the software is received into the unit correctly. The CRC algorithm presented earlier in this chapter may be sufficient to satisfy the validity of the upgrade software.

Security.

If security of the image is an issue, you may need to find an algorithm to digitally sign and/or encrypt the new software. The validation and decryption of the software would then be performed prior to programming.

Q.4 a. Explain communication basics for embedded system with a simple example of bus structure, read protocol and write protocol. (6)

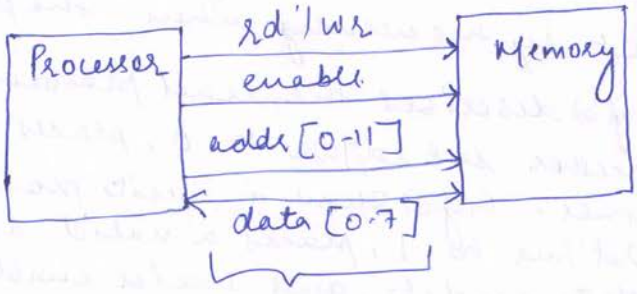
Answer:

(a) Ans Communication basics for embedded system with a simple Example

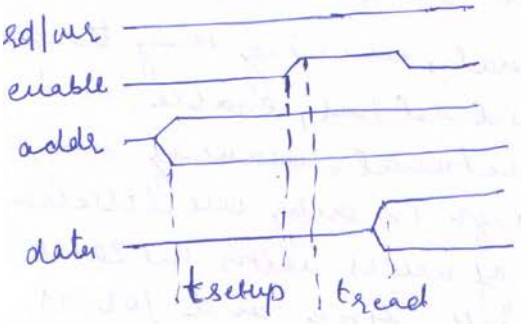
We begin by introducing a very basic communication example between a processor and a memory, shown in Fig. shows the bus structure, or the wires connecting the processor and the memory. The line rd/wr indicates whether the processor is reading or writing. The enable line is used by the processor whenever it wishes to carry out the read or write. Twelve address line addr. indicate the memory address that processor wishes to read or write. Twelve address lines addr. indicate the memory address that the processor wishes to read or write. Eight data lines data are set by the processor when writing or set by the memory when the processor is reading. Fig(b) describes the read protocol over these wires; the processor sets rd/wr to 0, places a valid data on the data lines. Fig(c) Shows a write protocol: the processor sets rd/wr to 1, places a valid address on addr, places data on data, and strobes enable, causing the memory to store the data. This very simple example brings up several points that we know describe.

Wires may be unidirectional, meaning they transmit in only one direction, as did rd/wr, enable, and addr. Or they may be bidirectional, meaning they transmit in two directions, though in only one direction at a time as did data. A set of wires with the same function is typically drawn as a thick line and for as a line with a small angled line drawn through it as was the case with addr and data.

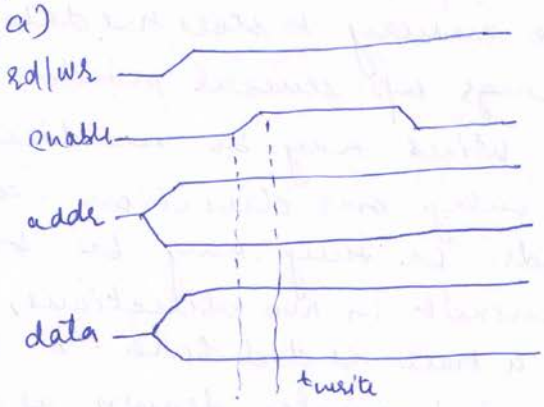
The term bus can refer to a set of wires with a single function within a communication. For example, we can refer to the "address bus" and the data bus in the above example. The term bus can also refer to the entire collection of wires used for the communication along with the comm. protocol over those wires. Both uses of the term are common and are often used in conjunction with one another. For example, we may say that the processor's bus consists of an address bus and a data bus. A protocol describes the rules for communicating over those wires. We deal primarily with low level hardware protocol in this chapter, while higher-level protocol, like IP (Internet Protocol) can be built on top of these protocols, using a layered approach.



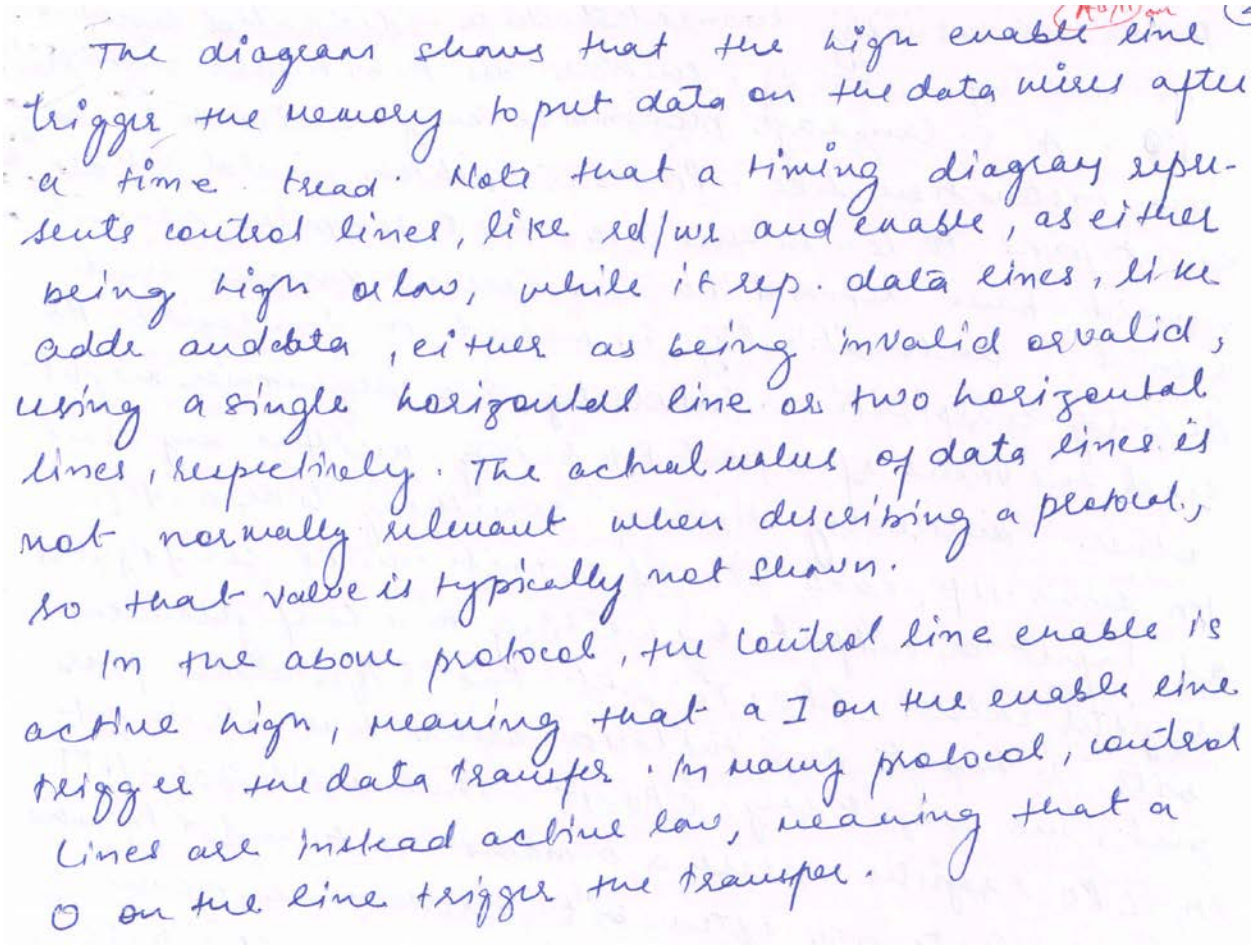
(a)



(b)



(c)



The diagram shows that the high enable line trigger the memory to put data on the data wires after a time tread. Note that a timing diagram represents control lines, like rd/wr and enable, as either being high or low, while it rep. data lines, like addr and data, either as being invalid or valid, using a single horizontal line or two horizontal lines, respectively. The actual value of data lines is not normally relevant when describing a protocol, so that value is typically not shown.

In the above protocol, the control line enable is active high, meaning that a 1 on the enable line trigger the data transfer. In many protocol, control lines are instead active low, meaning that a 0 on the line trigger the transfer.

b. Discuss embedded processor interfacing and explain port-based I/O and bus-based I/O. (6)

Answer:

Ans Microprocessor Interfacing : I/O Addressing
Port and Bus - Based I/O

A microprocessor may have ten or hundred of pins, many of which are control pins, such as a pin for clock input and another input pin for resetting the µp. Many of the other pins are used to communicate data to and from the µp. which we call processor I/O. There are two common methods for using to support I/O; port-based I/O and bus-based I/O

i) In port-based I/O; also known as parallel I/O a port can be directly read and written by processor instructions just like any other register in the µp; in fact, the

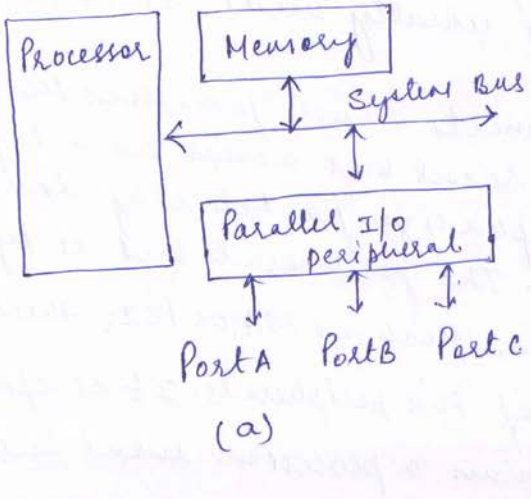
port is usually connected to a dedicated register. In this way, for eg, consider an 8-bit port named P0. A C-language programmer may write to P0 using an instruction like : P0 = 255, which would set all eight pins to 1s. In this case, the C compiler manual would have defined P0 as a special variable that would automatically be mapped to the register P0 during compilation. Conversely, the programmer might read the value of a port P1 being written by some other device by typing something like a = P1. In some µP, each bit of a port can be configured as input or output by writing to a configuration register called CP0. To set the high-order four bits to input and the lower-order four bits to output, we might say : CP0 = 15. This writes 00001111 to CP0 register, where a 0 means input and a 1 means output Ports are often bit-addressable, meaning that a programmer can read or write specific bits of the port. For example, one might say : x = P0.2 giving x the value of the number 2 pin of port P0.

(ii) In bus-based I/O, the µP has a set of address, data, and control ports corresponding to bus lines, and uses the buses to access memory as well as peripherals. The µP has the bus protocol built in to its hardware. Specifically, the software does not implement the protocol but merely executes a single instruction that in turn causes the hardware to write or read data over the bus. We normally consider the access to the peripheral as I/O, but

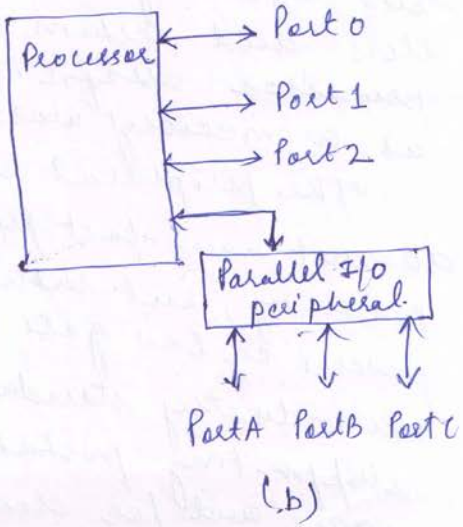
Conventionally consider the access to memory as I/O since the memory is considered more as a part of the µP.

A System may require parallel I/O (port-based I/O), but a µP may only support bus-based I/O. In this case, a parallel I/O peripheral may be used, as illustrated Fig. The peripheral is connected to the system bus on the other side, consisting just of a set of data line. The ports are connected to registers inside the peripheral, and the µP can read and write these registers in order to read and write the ports.

Even when a µP support port-based I/O, we may require more ports than are available. In this case, a parallel I/O peripheral can again be used, as illustrated in Fig. The µP has four ports in this ex. one of which is used to interface with a parallel I/O peripheral, which itself has three ports. Thus we have extended the number of available ports from four to six. Using a peripheral in this manner is often referred to as extended parallel I/O.



(a)



(b)

c. Draw and explain two-level bus architecture. (6)

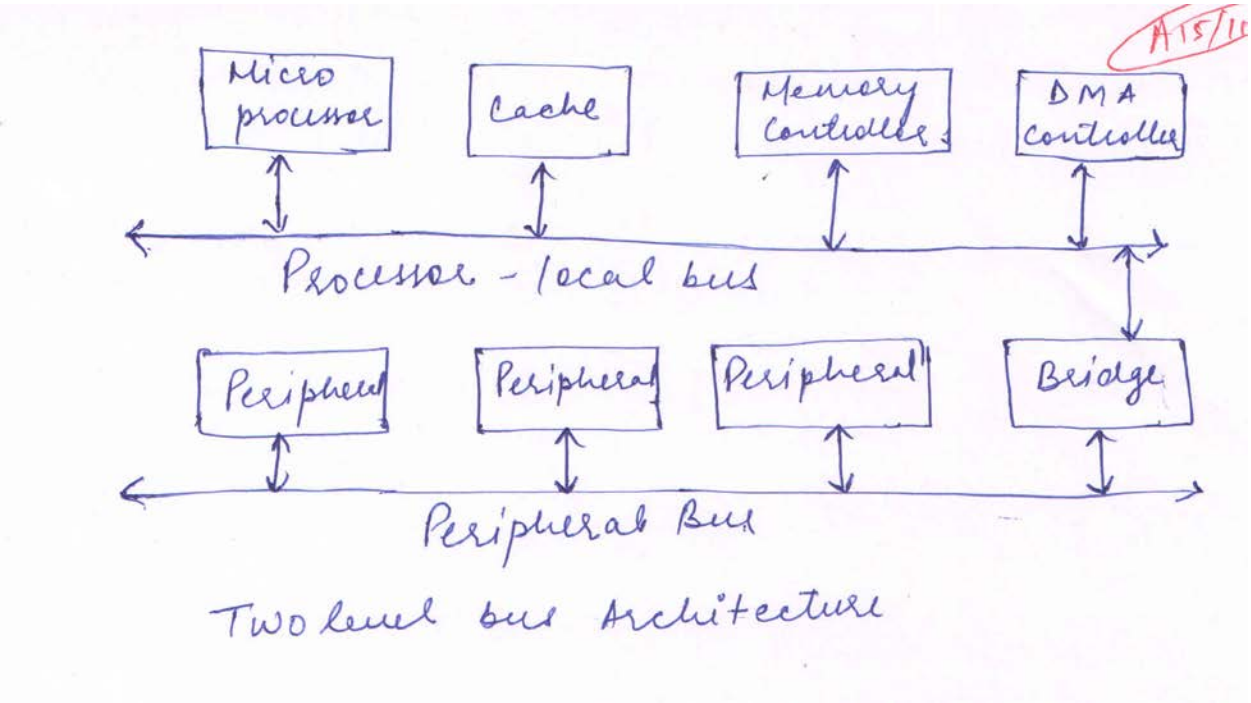
Answer:

TWO LEVEL BUS ARCHITECTURE

A microprocessor based embedded system will have numerous type of communication that must take place varying in their frequencies and speed requirement. The most frequent and high-speed communication will likely be between the µP. & its peripherals, like a UART. We could try to implement a single high speed bus for all the communications, but this approach has several disadvantages. First, it requires each peripheral to have a high-speed bus interface. Since a peripheral may not need such high-speed communication, having such an interface, may result in extra gates, power consumption and cost. Second, since a high speed bus will be very processor-specific, a peripheral with an interface to that bus may not be very portable. Third, having too many peripherals on the bus may result in slower bus.

Therefore, we often design system with two level of buses: a high speed processor local bus and a lower speed peripheral bus. The processor local bus typically connects the µP, cache, memory controllers and certain high-speed co-processors, and is processor specific. It is usually wide, as wide as a memory word.

The peripheral bus connects those processors that do not have fast processor local bus access as a top priority, but rather emphasize portability, low power, or low gate count. The peripheral bus is typically an industry standard bus, such as ISA or PCI, thus supporting portability of the peripherals. It is often narrow and/or slower than a processor local bus, thus requiring fewer pins, fewer gates and less power for interfacing.



Q.5 a. What is scheduler? Explain Priority based scheduling. (6)

Answer:

Now that we have a priority-based context switching mechanism, we have to determine an algorithm by which to assign priorities to processes. After assigning priorities, the OS takes care of the rest by choosing the highest-priority ready process. There are two major ways to assign priorities: ***static*** priorities that do not change during execution and ***dynamic*** priorities that do change. We will look at examples of each in this section.

6.3.1 Rate-Monotonic Scheduling

Rate-monotonic scheduling (RMS), introduced by Liu and Layland [Liu73], was one of the first scheduling policies developed for real-time systems and is still very widely used. RMS is a static scheduling policy. It turns out that these fixed priorities are sufficient to efficiently schedule the processes in many situations.

The theory underlying RMS is known as ***rate-monotonic analysis (RMA)***. This theory, as summarized below, uses a relatively simple model of the system.

- All processes run periodically on a single CPU.
- Context switching time is ignored.

- There are no data dependencies between processes.
- The execution time for a process is constant.
- All deadlines are at the ends of their periods.
- The highest-priority ready process is always selected for execution.

The major result of RMA is that a relatively simple scheduling policy is optimal under certain conditions. Priorities are assigned by rank order of period, with the process with the shortest period being assigned the highest priority. This fixed-priority scheduling policy is the optimum assignment of static priorities to processes, in that it provides the highest CPU utilization while ensuring that all processes meet their deadlines.

Example 6.3 illustrates RMS.

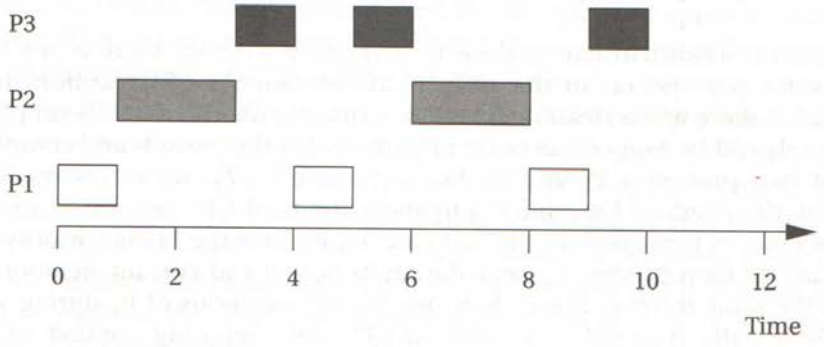
Example 6.3

Rate-monotonic scheduling

Here is a simple set of processes and their characteristics.

Process	Execution time	Period
P1	1	4
P2	2	6
P3	3	12

Applying the principles of RMA, we give P1 the highest priority, P2 the middle priority, and P3 the lowest priority. To understand all the interactions between the periods, we need to construct a time line equal in length to hyperperiod, which is 12 in this case.



All three periods start at time zero. P1's data arrive first. Since P1 is the highest-priority process, it can start to execute immediately. After one time unit, P1 finishes and goes out of the ready state until the start of its next period. At time 1, P2 starts executing as the

highest-priority ready process. At time 3, P2 finishes and P3 starts executing. P1's next iteration starts at time 4, at which point it interrupts P3. P3 gets one more time unit of execution between the second iterations of P1 and P2, but P3 does not get to finish until after the third iteration of P1.

Consider the following different set of execution times for these processes, keeping the same deadlines.

Process	Execution time	Period
P1	2	4
P2	3	6
P3	3	12

In this case, we can show that there is no feasible assignment of priorities that guarantees scheduling. Even though each process alone has an execution time significantly less than its period, combinations of processes can require more than 100% of the available CPU cycles. For example, during one 12 time-unit interval, we must execute P1 three times, requiring 6 units of CPU time; P2 twice, costing 6 units of CPU time; and P3 one time, requiring 3 units of CPU time. The total of 6 + 6 + 3 = 15 units of CPU time is more than the 12 time units available, clearly exceeding the available CPU capacity.

Liu and Layland [Liu73] proved that the RMA priority assignment is optimal using critical-instant analysis. We define the ***response time*** of a process as the time at which the process finishes. The ***critical*** instant for a process is defined as the instant during execution at which the task has the largest response time. It is easy to prove that the critical instant for any process P, under the RMA model, occurs when it is ready and all higher-priority processes are also ready—if we change any higher-priority process to waiting, then P's response time can only go down.

We can use critical-instant analysis to determine whether there is any feasible schedule for the system. In the case of the second set of execution times in Example 6.3, there was no feasible schedule. Critical-instant analysis also implies that priorities should be assigned in order of periods. Let the periods and computation times of two processes P_1 and P_2 be τ_1, τ_2 and T_1, T_2, with $\tau_1 < \tau_2$. We can generalize the result of Example 6.3 to show the total CPU requirements for the two processes in two cases. In the first case, let P_1 have the higher priority. In the worst case we then execute P_2 once during its period and as many iterations of P_1 as fit in the same interval. Since there are $\lfloor \tau_2/\tau_1 \rfloor$ iterations of P_1 during a single period of P_2, the required constraint on CPU time, ignoring context switching overhead, is

If, on the other hand, we give higher priority to P_2, then critical-instant analysis tells us that we must execute all of P_2 and all of P_1 in one of P_1's periods in the worst case:

$$T_1 + T_2 \leq \tau_1. \tag{6.5}$$

There are cases where the first relationship can be satisfied and the second cannot, but there are no cases where the second relationship can be satisfied and the first cannot. We can inductively show that the process with the shorter period should always be given higher priority for process sets of arbitrary size. It is also possible to prove that RMS always provides a feasible schedule if such a schedule exists.

The bad news is that, although RMS is the optimal static-priority schedule, it does not always allow the system to use 100% of the available CPU cycles. In the RMS framework, the total CPU utilization for a set of n tasks is

$$U = \sum_{i=1}^{n} \frac{T_i}{\tau_i}. \tag{6.6}$$

The fraction T_i/τ_i is the fraction of time that the CPU spends executing task i. It is possible to show that for a set of two tasks under RMS scheduling, the CPU utilization U will be no greater than $2(2^{1/2} - 1) \cong 0.83$. In other words, the CPU will be idle at least 17% of the time. This idle time is due to the fact that priorities are assigned statically; we see in the next section that more aggressive scheduling policies can improve CPU utilization. When there are m tasks with fixed priorities, the maximum processor utilization is

$$U = m(2^{1/m} - 1). \tag{6.7}$$

As m approaches infinity, the least upper bound to CPU utilization is $\ln 2 = 0.69$—the CPU will be idle 31% of the time. This does not mean that we can never use 100% of the CPU. If the periods of the tasks are arranged properly, then we can schedule tasks to make use of 100% of the CPU. But the least upper bound of 69% tells us that RMS can in some cases deliver utilizations significantly below 100%.

The implementation of RMS is very simple. Figure 6.12 shows C code for an RMS scheduler run at the OS's timer interrupt. The code merely scans through the list of processes in priority order and selects the highest-priority ready process to run. Because the priorities are static, the processes can be sorted by priority in advance before the system starts executing. As a result, this scheduler has an asymptotic complexity of $O(n)$, where n is the number of processes in the system. (This code assumes that processes are not created dynamically. If dynamic process creation is required, the array can be replaced by a linked list of processes, but the asymptotic complexity remains the same.) The RMS scheduler has both low asymptotic complexity and low actual execution time, which helps minimize the discrepancies between the zero-context-switch assumption of RMA and the actual execution of an RMS system.

```
/* processes[] is an array of process activation records,
   stored in order of priority, with processes[0] being
   the highest-priority process */
Activation_record processes[NPROCESSES];

void RMA(int current) { /* current = currently executing
process */
  int i;
  /* turn off current process (may be turned back on) */
  processes[current].state = READY_STATE;
  /* find process to start executing */
  for (i = 0; i < NPROCESSES; i++)
    if (processes[i].state == READY_STATE) {
        /* make this the running process */
        processes[i].state == EXECUTING_STATE;
        break;
    }
}
```

FIGURE 6.12

C code for rate-monotonic scheduling.

6.3.2 Earliest-Deadline-First Scheduling

Earliest deadline first (EDF) is another well-known scheduling policy that was also studied by Liu and Layland [Liu73]. It is a dynamic priority scheme—it changes process priorities during execution based on initiation times. As a result, it can achieve higher CPU utilizations than RMS.

The EDF policy is also very simple: It assigns priorities in order of deadline. The highest-priority process is the one whose deadline is nearest in time, and the lowest-priority process is the one whose deadline is farthest away. Clearly, priorities must be recalculated at every completion of a process. However, the final step of the OS during the scheduling procedure is the same as for RMS—the highest-priority ready process is chosen for execution.

Example 6.4 illustrates EDF scheduling in practice.

b. Discuss some of the important criteria used in making an RTOS selection. (6)

Answer:

Q.5

(b) Important Criteria used in making an RTOS selection:

Processor support:

The processor is typically the first choice in the hardware design on a project. Most RTOSes support the popular processors used in embedded systems. If the processor used on your project is not supported, you need to determine whether porting the RTOS to that processor is an option or if it is necessary to choose a different RTOS. Porting an RTOS is not always trivial.

Real time characteristics:

We have already covered the real-time characteristics of an RTOS, which include Interrupt latency, context switch time, and the execution time of each system call. These are technical criteria that are inherent to the system and cannot be changed.

Budget Constraints:-

RTOSes span the cost spectrum from open source and royalty free to tens of thousands of dollars per developer seat plus royalties for each unit shipped. You need to understand what your costs are in both cases. Open source might mean no upfront costs, but there might be costs associated with getting support when needed. You also need to understand the licensing details of the RTOS you choose.

Memory usage:

Clearly, in an embedded environment, memory constraints are a frequent concern. A few RTOSes can be scaled to fit the

smallest of of embedded systems. for example, by removing features to create a smaller footprint. others require a minimum set of resources comparable to a low-end PC. It is important to keep in mind the potential need to change an RTOS in the future, when memory is not a plentiful or costs need to be reduced.

Device drivers and software components:-

The device drivers included with an RTOS can aid in keeping the development on schedule. This reduces the amount of code you need to develop for particular peripherals. Many RTOSes support the common devices found in embedded system. If additional features are needed, such as networking support, graphics libraries, web interfaces, and filesystems, an RTOS might include these and have the code already integrated and tested. Some RTOS might require more fees for using these added features.

Technical support:

This may include a number of incidents or a period of phone support. Some RTOSes require you to pay an annual fee to maintain a service contract. for open source RTOSes, an open forum or mailing list might be provided. If more specialized support is needed, you'll have to search around to see what is available. Popular open source RTOSes have companies dedicated to providing support.

Tool Compatibility:-

Make sure the RTOS works with the assembler, compiler, linker, and debugger you have already obtained. If the RTOS does not support tools that you or your team are familiar with, the learning curve will take more time.

c. Explain interrupt handling in embedded system. (6)

Answer:

(C)

Ans Interrupt handling in embedded system:

There are several issues you need to be aware of when handling interrupts in embedded system that use an operating system including:

Interrupt priority:-

Interrupts have the highest priority in a system even higher than the highest operating system task. Interrupts are not scheduled the ISR executes outside of the operating system's scheduler.

Disabling interrupts:

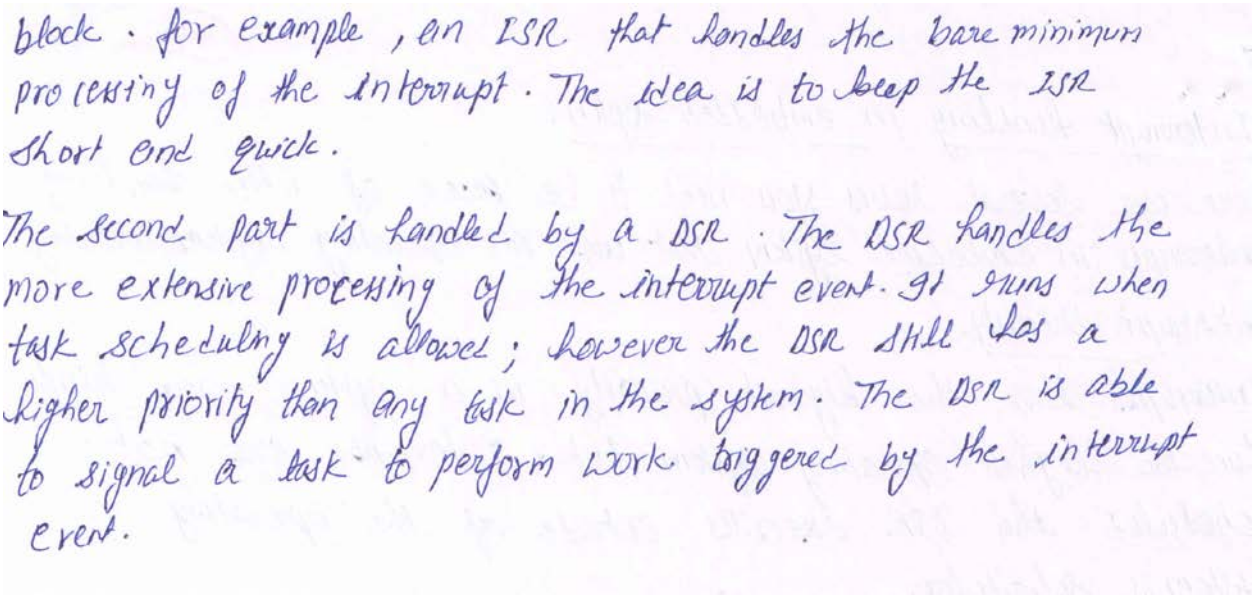
Because the operating system code must guarantee its data structures' integrity when they are accessed, the operating system disables interrupts during operations that alter internal operating system data, such as the ready list. this increase the interrupt latency. The responsiveness of the operating system comes at the price of longer interrupt latency. When a task disables interrupts, it prevents the scheduler from doing its job. Tasks should not disable interrupts on their own.

Interrupt stack

Some operating systems have a separate stack space for the execution of ISRs. This is important because, if interrupts are stored on the same stack as regular tasks, each task's stack must accommodate the worst-case interrupt nesting scenario. Such large stacks increase RAM requirements across all n tasks.

Signaling tasks:

Because ISRs execute outside of the scheduler, they are not allowed to make any operating system calls that can block. for


block. for example, an ISR that handles the bare minimum processing of the interrupt. The idea is to keep the ISR short and quick.

The second part is handled by a DSR. The DSR handles the more extensive processing of the interrupt event. It runs when task scheduling is allowed; however the DSR still has a higher priority than any task in the system. The DSR is able to signal a task to perform work triggered by the interrupt event.

Q.6 a. Discuss system synthesis and hardware/software co-design. (6)

Answer:

Ans(a) System Synthesis and Hardware/Software Codesign.

System synthesis converts multiple processes into multiple processors. The term system here refers to a collection of processors.

System synthesis involves several tasks. Transformation is the task of rewriting the process to be more amenable to synthesis. for example, a designer may have described some behavior using two processes. but analysis might show that these two processes are really exclusive to one another and thus could be merged into one process.

Like wise, a large process might actually consist of two independent operations that could be done concurrently, so that process could be divided into two processes. other common transformations include procedure inlining and loop unrolling.

Allocation is the task of selecting the number and types of processors to use to implement the processes. A designer might choose to use an 8-bit general-purpose processor along with a single-purpose processor. Alternatively, the designer might use a 32-bit general-purpose processor. And 8-bit general purpose processor, and multiple single-purpose processors. Allocation actually includes selecting processors, memories and buses. Allocation is essentially the design of the system architecture.

Partitioning is the task of mapping the processes to processors. One process can be implemented on multiple processors, and multiple processes can be implemented on a single processor. Likewise variables must be partitioned among memories and communications among buses.

Scheduling is the task of determining when each of the multiple processes on a single processor will have its chance to execute on the processor. Like wise, memory accesses and bus communications must be scheduled.

These tasks may be performed in a variety of orders, and iteration among the tasks is common.

System synthesis, like all forms of synthesis, is driven by constraints. A typical set of constraints dictates that certain performance requirements must be met at minimum cost. In such a situation, system synthesis might seek to allocate as much behavior as possible to a general-purpose processor, since a GPP may provide for low cost, flexible implementation. A minimum number of single purpose processors might be used to meet the performance requirements.

System synthesis for general-purpose processors only has been around for a few decades, but hasn't been called system synthesis. Name like multiprocessing, parallel processing, and real-time scheduling have been more common.

The maturation of behavioral synthesis in the 1990s has enabled the consideration of single purpose processor during the allocation and partitioning tasks of system synthesis.

Thus the term hardware/software codesign has been used extensively in the research community, to highlight research that focuses on the unique requirements of such simultaneous consideration of both hardware and software during synthesis.

b. Explain formal verification and simulation of hardware/software co-design. (6)

Answer:

(b)

Ans formal verification and simulation of hardware and software co-design

verification is the task of ensuring that a design is correct and complete. Correctness means that the design implements its specification accurately. Completeness means that the design's specification described appropriate output responses to all relevant input sequences.

The two main approaches to verification that analyzes a design to prove or disprove certain properties. We might seek to formally verify correctness of a particular design step, such as verifying that a particular structural description correctly implements a particular behavioral description, by proving the equivalence of the two descriptions for example, we might describe an ALU behaviorally and then create a structural implementation using gates. We can prove the correctness of the structure by deriving a Boolean equation for the outputs, creating a truth table for those equation, and showing that the truth table is identical to the table created from the original behavior.

Alternatively, we might seek to formally verify completeness of a behavioral description, be proving formally that certain situations always or never occur.

for example we might prove that for an elevator controller, the elevator door can never be open while the elevator is moving by deriving the conditions for the door being open and showing that these conditions conflict with those for the elevator moving

The more common approach to verification is simulation. formal verification is a very hard problem. and as such has been limited in practice to either smaller designs or to verifying only certain key properties. Instead by far the most common approach to verification in practice is simulation. Simulation is an approach in which we create a model of the design that can be executed on a computer. We provide sample input values to this model, and check that the output values generated by the model match our expectations, for example we can verify the correctness of an ALU by providing all possible input combinations. and checking the ALU output for correct results, which we of course have to compute using other means.

Compared with physical implementation Simulation has several advantage with respect to testing and debugging a system. The two most important advantages are excellent controllability and observability.

Controllability is the ability to control execution of the system. A designer can control time as well as data value.

- Simulation could take much time for system with complex external environments. A designer may spend more time modeling the external environment than the system itself.
- Simulation speed can be quite slow compared to execution of physical implementation.

c. Give the steps of development of process model. (6)

Answer:

(c)

Steps Design Process Models :-

A designer must proceed through several steps when designing a system. We can think of describing behavior as one design step, converting behavior to structure as another step, and mapping structure to a physical implementation as another step. Each step will obviously consist of numerous substeps. A design process model describes the order in which these step are taken.

The term process here should not be confused with the notion of a process in the concurrent process model, nor should it be confused with the IC manufacturing process. Here process refers to the manner in which the embedded system designer proceeds through design steps.

One process model is the waterfall model,

Water fall model

Suppose a designer has six month to build a system. In the waterfall model, the designer first exerts extensive effort, perhaps two months, describing the behavior completely. Once fully satisfied that the behavior is correct, after extensive behavioral simulation and debugging, the designer moves on to the next step of designing structure. Again, much effort is exerted, perhaps another two months, until the designer is satisfied the structure is correct. Finally, the physical implementation step is carried out, occupying

perhaps the last two months. The result is a final system implementation, hopefully a correct one. In the waterfall model, when we proceed to the Next step, we never come back to the earlier steps, much like water cascading down a mountain doesn't return to higher elevations.

Unfortunately, the waterfall model is not very realistic, for several reasons. First we will almost always find bugs in the later steps that should be fixed in an earlier step. For example when testing the structure, we may notice that we forget to handle a certain input combination in the behavior.

Second we often do not know the complete desired behavior of the system until we have a working prototype. For example, we may build a prototype device and show it to a customer, who then gets the idea of adding several features.

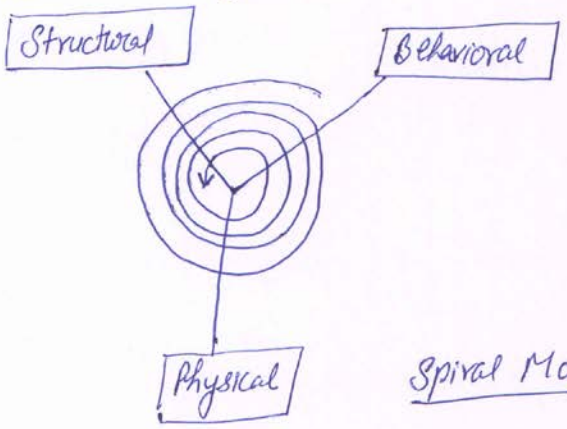
Third, system specifications commonly change unexpectedly. for example, we may be halfway done designing a system when our company decides that to be competitive, the product must be smaller and consume less power than originally expected, requiring several features to be dropped.

The accompanying unexpected iteration back through the three steps often result in missed deadlines, and hence in lost revenues or products that never make it to market.

An alternative process model is the spiral model.

Suppose again that the designer has six months to build the system. In the spiral model, the designer first exerts some effort to describe the basic behavior of the system, perhaps a few weeks. This description will be incomplete, but have

the basic functions, with many functions left to be filled in later. Next the designer move on to designing structure, again taking maybe a few weeks. Finally the designer create a physical prototype of the system. This prototype is used to test out the basic function, and to get a better idea of what function we should add to the system.



Spiral Model

With this experience, the designer proceeds to proceed through the three steps again expanding the original behavioral description or even starting with a new one, creating structure, and obtaining a physical implementation again. These step may be repeated several times until the desired system is obtained.

The spiral model has its drawback, too. The designer must come up with way to obtain structure and physical implementation quickly.

Q.7 a. Design a process control system and explain its different parts. (10)

Answer:

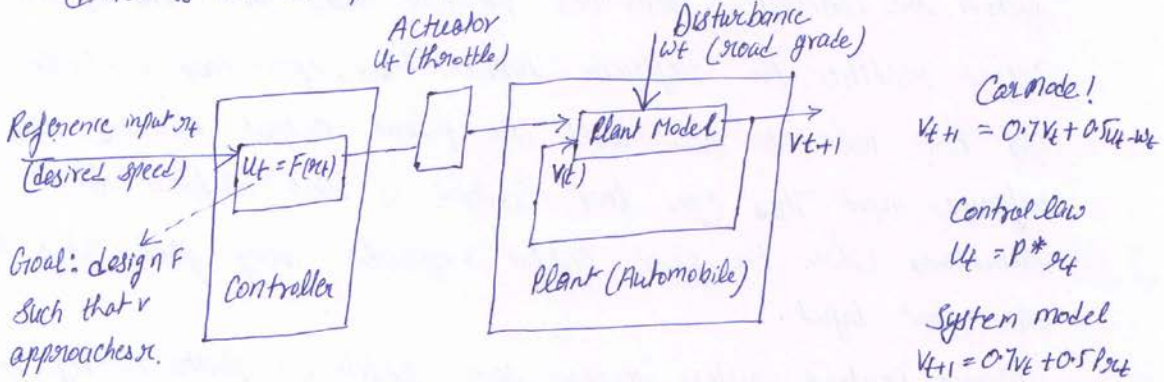
(A) Control Systems:-

Control systems minimally consist of several parts

(1) The plant, also known as the process, is the physical system to be controlled. An automobile is an example of a plant.

(2) The output is the particular physical system aspect that we are interested in controlling. The speed of an automobile is an example of an output.

(3) The reference input is the desired value that we want to see for the output. The desired speed set by an automobile's driver is an example of a reference input.



(a) Open-Loop Control

(4) The actuator is the device that we use to control the input to the plant. A stepper motor controlling a car's throttle position is an example of an actuator.

(5) The controller is the system that we use to compute the input to the plant such that we achieve the desired output from the plant.

6. A disturbance is an additional undesirable input to the plant imposed by the environment that may cause the plant output to differ from what we would have expected based on the plant input. Wind and road grade are examples of disturbances that can alter the speed of an automobile.

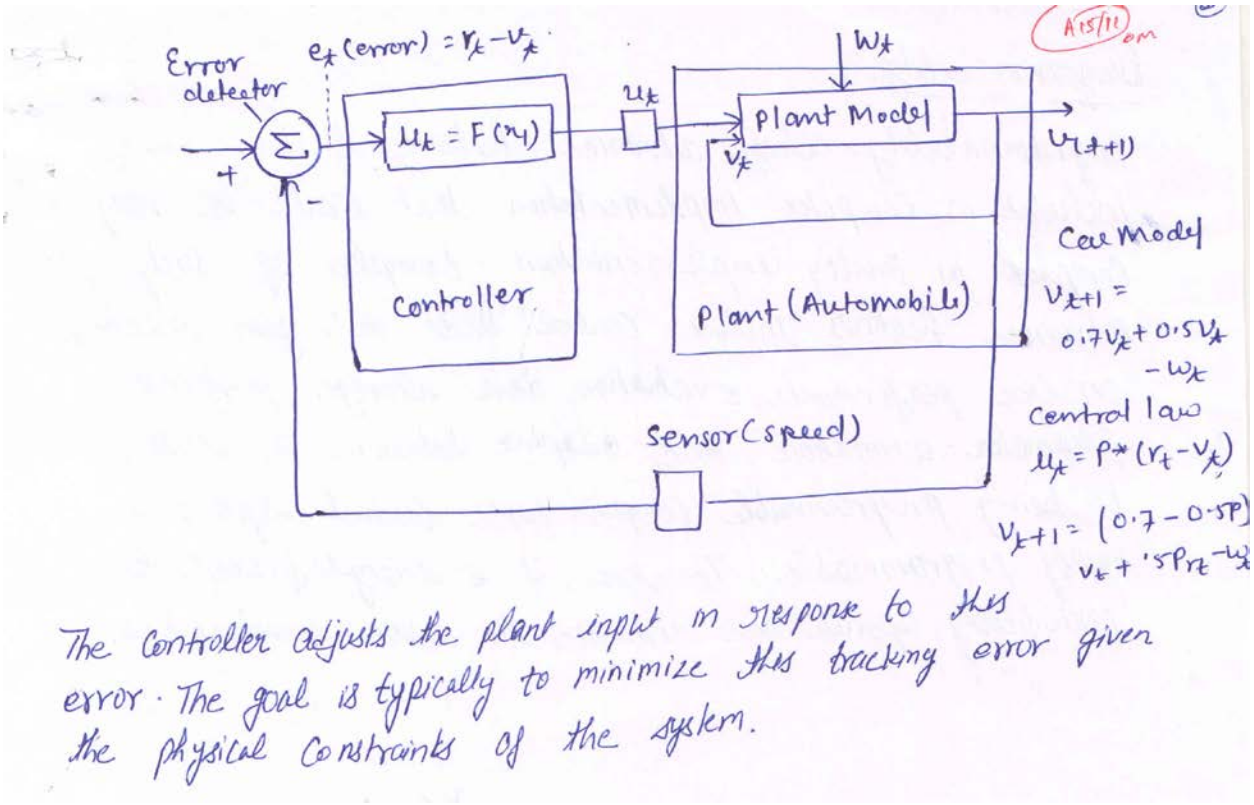
A control system with these components, configured in fig is refferred to as open-loop, or feed-forward, control system. The controller reads the reference input, and then computes a setting for the actuator. The actuator modifies the input to the plant, which along with any disturbance, results some time later in a change in the plant output. In an open-loop system, the controller does not measure how well the plant output matches the reference input. Thus, open-loop controller does not measure how well the plant output matches the reference input. Thus open loop control is best suited to situations where the plant output responds very predictably to the plant input.

Many control system possess som additional parts in fig (b)

(1) A sensor measures the plant output.

(2) An error detector determines the difference between the plant output and the refrence input.

A control system with these parts, configured as in figure (b) is known as a closed-loop or feedback, control system. A closed loop system monitors the error between the plant output and refrence input.



b. Discuss the benefits of computer-based control implementations. (8)

Answer:

(b)

Ans Benefits of Computer-Based Control Implementations.

Control systems can be implemented by either continuous time or digital time approaches. Since most processes that we are interested in controlling evolve as continuous variables in continuous time, and computer-based control approaches add additional complications such as quantization, overflow, aliasing and computation delay is an important to consider briefly the benefits obtained through embedded computer control.

Repeatability, Reproducability and Stability

The analog components in a control system are affected by aging temperature and manufacturing tolerance effects. Alternatively digital system are inherently repeatable. If two processor are loaded with the same program and data, they will compute identical results. They are also more stable than analog implementation in the presence of aging.

Programmability:

Programmability allows advanced features to be easily included in computer implementation that would be very complex in analog implementation. Examples of such advanced features include control mode and gain switching, on-line performance evaluation, data storage, performance parameter estimation, and adaptive behavior. In addition to being programmable computer based control system are easily programmable. Therefore, it is straightforward to periodically upgrade and enhance the system characteristics.

Text Book

Wayne Wolf, Computers as Components: Principle of Embedded Computing System Design, Second edition, Morgan Kaufmann Publishers, 2008.

Frank Vahid and Tony Givargis, Embedded Systems Design: A Unified Hardware/Software Introduction , John Wiley & Sons,2002.