UNIT-III Embedded Programming

Components for Embedded Programs.

⇒ Embedded software uses three components.

1. State machine
2. Circular buffer
3. Queue

⇒ State machines are well suited to reactive system such as user interfaces, Circular buffers and queues are useful in digital signal processing.

State Machine:

* A state machine is any object that behaves different based on its history and current inputs.
* Many embedded systems consists of a collection of state at various levels of the electronics or software.
* In general, a state machine is any device that stores the status of something at a given time and can operate on input

to change the status and/or cause an action or output to take place for any given change.

In practice, however, state machines are used to develop and describe specific device or program interactions.

⇒ To summarize it, a state machine can be described as:

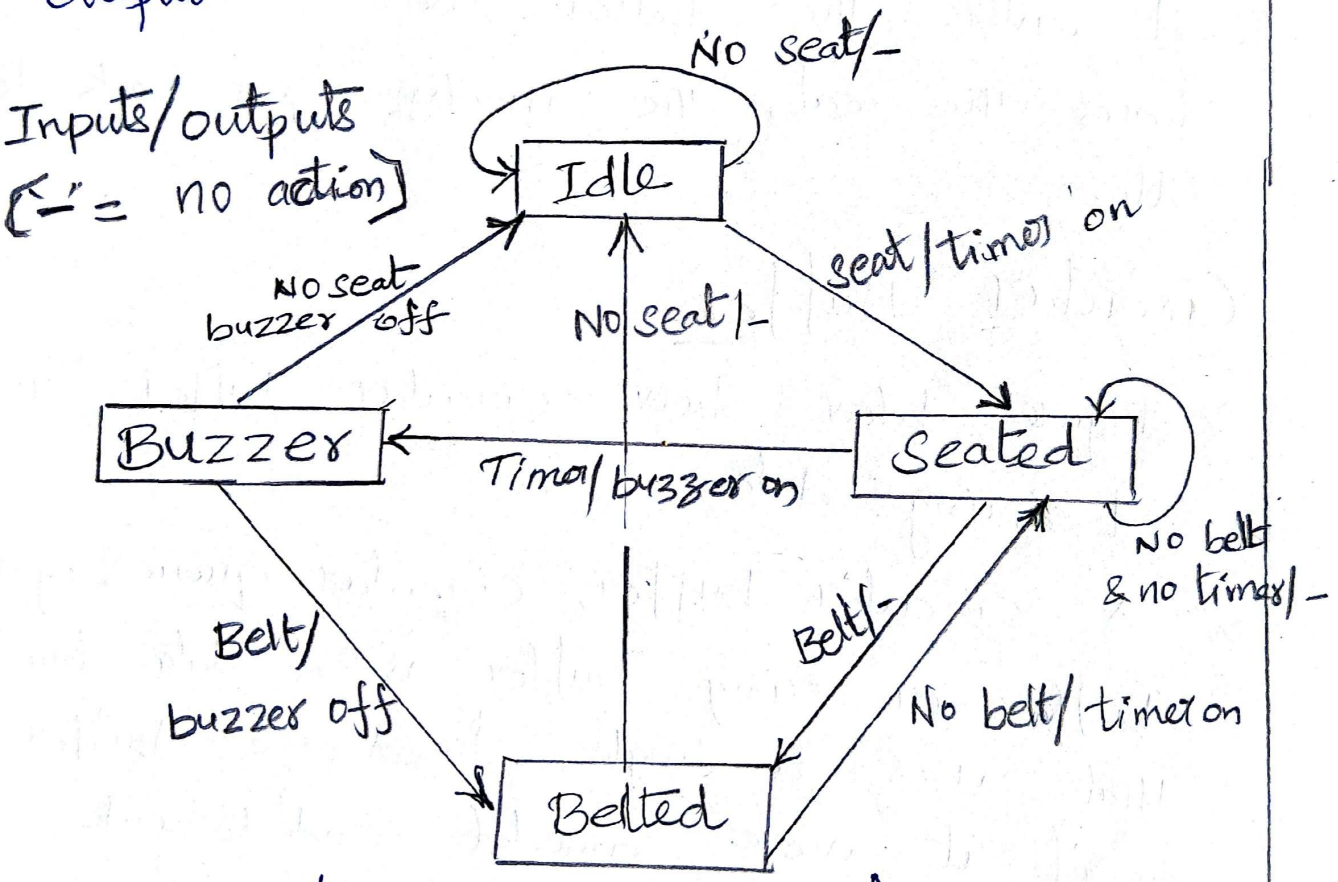
1. A set of states
2. An initial state or record of something stored somewhere
3. A set of input events.
4. A set of output events.
5. A set of actions or output events that maps the states and input to output.
6. A set of actions or outputs events that maps the states and inputs to states

Finite state Automation (FSA), Finite state Machine (FSM) or state Transition Diagram (STD) is a formal method used in the specification and design of wide range of embedded and real time systems.

State Machine for seat belt controller.

⇒ Controllers job is to turn on a buzzer if a person sits in a seat and does not fasten the seat belt within a fixed amount of time.

⇒ This system has three inputs and one output.



⇒ The inputs are a sensor for the seat to know when a person has sat down, a seat belt sensor that tells when the belt is fastened and a timer that goes off when the required time interval has elapsed. The output is the buzzer.

⇨ The idle state is in force when there is no person in the seat. When the person sits down, the machine goes into the seated state and turns on the timer.

⇨ If the timer goes off before the seat is fastend, the machine goes into the buzzer state. If the seat goes on first, it enters the belted state. When person leaves the seat, the machine goes back to idle.

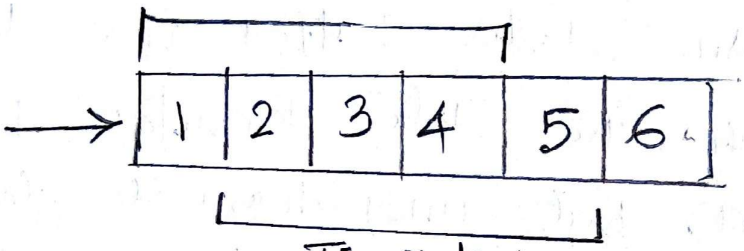
Circular Buffers

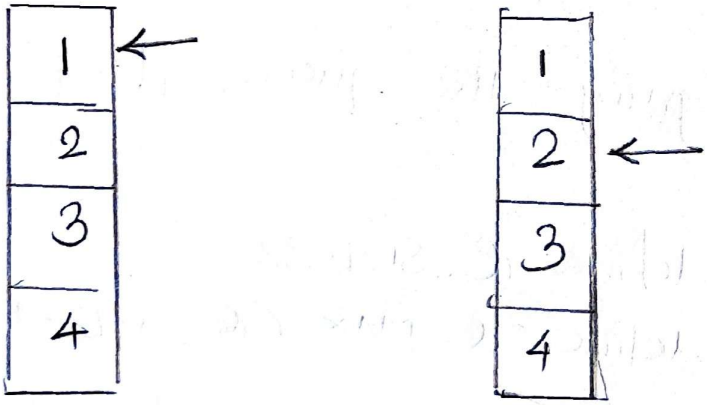
⇨ Figure below shows circular buffer for streaming data.

⇨ A circular buffer, circular queue, cyclic buffer or ring buffer is a data structure that uses a single, fixed-size buffer as if it were connected end-to-end. This structure lends itself easily to buffering data streams.

⇨ The circular buffer behaviour is ideal for emplementing any data structure that is statically allocated and behaves like FIFO

⇒ Circular buffer are a special type of buffer where the data is circulated around a buffer. In this way they are similar to a single buffer that moves the next data pointer to the start of the buffer to access the next data. In this way the address pointer circulates around the addresses.





Circular buffer

⇒ When a buffer underruns, it indicate that there is no more data in the buffer and that further processing should be stopped. This may indicate an error if the system is designed so that it would never run out of data

⇒ If it can happen in normal operation then the data underrun signal indicates a state and not an error. In both cases, a single signal is needed to recognise this point.

Queue

⇒ Queues are also used in signal processing and event processing.

⇒ Queues are used whenever data may arrive and depart at somewhat unpredictable times.

⇒ Queue is also referred to as an elastic buffer. An elastic buffer is a device that helps smooth the data transfer between two similar, but unsynchronized clock domains.

⇒ Linked link is used for building queue.

For designing the queue, it is declares as follows:

```
# define Q_SIZE 32
# define Q_MAX (Q_SIZE-1)
int q[Q_SIZE];      /* array for queue */
int head, tail      /* position of head and
                       tail in the queue */
```

Models of Programs

⇨ The basic concept is to enable the execution of an instruction whenever its required operands become available. Programs for data driven computations can be represented by data flow graphs.

⇨ Each instruction in a data flow computer is implemented as a template, which consists of the operator, operand receivers and result destinations.

⇨ Operands are marked on the incoming arcs and results are on outgoing arcs.

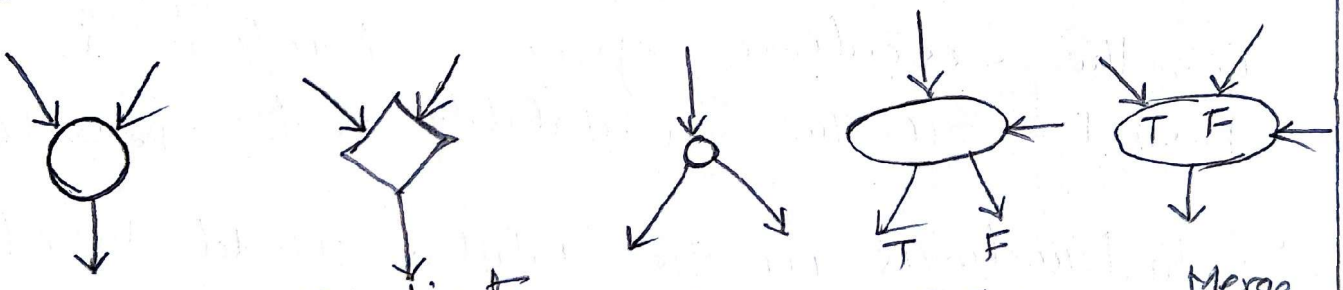
⇨ Dataflow model of execution is asynchronous, i.e., the execution of an instruction is based on the availability of its operands.

⇨ Instructions in the dataflow model do not impose any constraints on sequencing except the data dependencies in the program.

⇨ The dataflow model incurs more overhead in the execution of an instruction cycle compared to its control-flow counterpart due to its fine-grained approach to parallelism.

⇨ In dataflow machines each instruction is considered to be a seperate process. To facilitate data-driven execution each instruction that produces a value contains pointers to all its consumers. Since an instruction in such a dataflow program contains only references to other instructions. it can be viewed as a node in a graph.

⇨ Data flow program is represented as a directed graph. G = G (N, A), where nodes in 'N' represent instructions and arcs in A represent data dependencies between the nodes. The operands are conveyed from one node to another in data packets called tokens.



Basic primitives of the dataflow graph.

In dataflow computers, the machine level language is represented by dataflow graphs.

The above figures shows basic primitives of the dataflow graph.

- In dataflow machines each instruction is considered to be an instruction cycle compared to its "control flow" counterpart due to its fine-grained approach to parallelism.

- In dataflow machines each instruction is treated to be a seperate process. To facilitate data driven execution each instruction that produces a value contains pointer to all its consumers.

- Dataflow program is represented as a directed graph, $G = \mathbb{G}[N, A]$, where nodes in N represent instructions and arcs in 'A' represent data dependencies between the nodes. The operands are conveyed from one node to another in data packets called tokens.

- Dataflow graph exhibits two kinds of parallelism in instruction execution.

a) **Spatial parallelism**: Any two nodes can be potentially executed concurrently if there is no data dependence between them.

b) **Temporal parallelism**: This type of parallelism results from pipelining independent waves of computation through the graph.

- The dataflow graph is similar to a dependence graph used in intermediate representation of compilers.

Static Model

⇨ The static model allows at most one instance of a node to enabled for firing A data flow actor can be executed only when all of the tokens are available on its input arcs and no tokens exist on any of its output arcs.

Limitation of Static Model:

1. Consecutive iterations of a loop can only be pipelined.
2. Due to acknowledgement tokens, the token traffic is doubled.
3. lack of support for programming constructs that are essential to modern progeamming language.

Dynamic Model

It permits activation of several instances of a node at the same time during run-time. To distinguish between different instances of a node, a tage is associated with each token that identifies the context in which a particular token was generated.

- Advantage: Better performance as it allows multiple tokens on each thereby unfolding more parallelism
- Associative memory would be ideal.

Assembly linking and loading:

⇨ Assembly and linking are the last steps in the compilation process. they turn a list of instructions into an image of the program's bits in memory.

⇨ A Compiler is a computer program that translates a program written in a high-level language to the machine language of a computer.

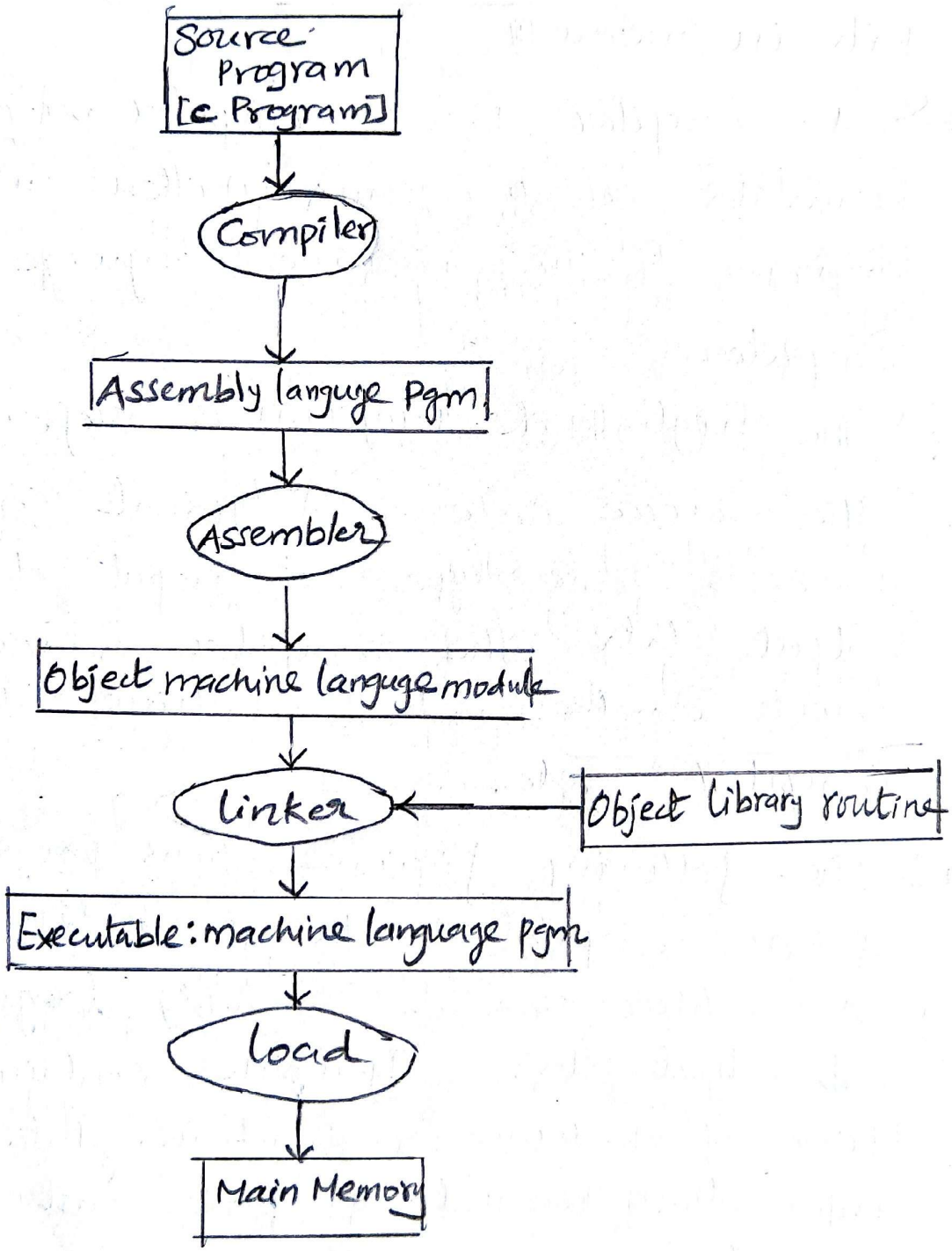
⇨ The high-level program is referred to as "the source code". A typical computer program processes some type of input data to produce output data. The compiler is used to translate source code into machine code or Compiled Code.

⇨ The following figure shows program generation from compilation through loading.

⇨ Assembler converts assembly language programs into object files. Object files contain a combination of machine instructions, data and information needed to place instructions properly in memory.

⇨ Linker merges the objects files produced by seperate compilation or assembly and creates an executable file-

Loader: Part of the OS that brings an executable file residing on disk into memory and starts it running



Program generation from compilation through loading.

Assembler:

⇨ The assembler is responsible for translating the assembly language program into machine code. When the source code is essentially a simbolic representation for a numerical machine language is called an assembly language.

⇨ A pure assembly language is a language in which each statement produces exactly one machine instruction.

⇨ When translating assembly code into object code, the assembler must translate opcodes and format the bits in each instruction, and labels into addresses.

label: It is an identifier and optional field. Labels are used extensively in programs to reduce reliance upon programmers when data or code is located. The maximum length of label differs between assemblers. Some accept upto 32 characters long, other only four characters.

⇨ The name of each symbol and its address is stored in a symbol table that is built during the first pass. The symbol table is built by scanning from the first instruction to the last.

⇨ Symbol table contains name and address field. Symbol table is built by the analysis phase. It also contains flag to indicate errors.

⇨ During scanning the current location in memory is kept in a Program location counter (PLC).

⇨ Memory allocation mean the fixing the address of the assembly language statement. Suppose we want to fix the memory address of M, then also fix the address of remaining instructions.

⇨ location counter is data structure used to implement the memory allocation. Location counter [LC] is always made to contain the address of the next memory word in the target program.

⇨ SYMTAB includes the name and value for each label in the source program, togethe with flags to indicate error conditions. It also contains information about the data area or instruction labeled.

⇨ During Pass 01: labels are entered into SYMTAB as they are encountered in the source program along with their assigned address.

⇨ During Pass 02, symbols used as operand as a hash table for efficiency of insertion and retrieval. Entries are rarely deleted from this table.

⇨ It is possible for both passes of the assembler to read the original source program as input. Information such as location counter values and error flag statement can be communicated between the two passes.

⇨ For this reason, Pass I usually writes an intermediate file that contains each source statement together with its assigned address, error indications etc. This file is used as the input to Pass 2.

Linking

⇨ This program might make use of other programs, or libraries of programs that are linked together into a single program and the interconnecting references are resolved.

⇨ Linker is a program which combines the target program with the code of other programs and library routines.

⇨ During the process of linking the object module is created. This object module contains the target code and information about other programs and library routines that are required to call during the program execution.

⇨ The output of translator is a program called object module. The linker processes these object modules binds with necessary library routines and prepares a ready to execute program. such a program is called binary program.

⇨ The binary program also contains some necessary information about allocation and relocation. The loader then loads this program into memory for execution purpose.

⇨ The place in the file where a label is defined is known as an entry point. The place in the file where the label is used is called an external reference. The main job of the loader is to resolve external reference based on available entry points.

Compilation Techniques

- Compilation combines translation and optimization. The high-level language program is translated into the lower-level form of instructions; optimization try to generate better instruction sequences than would be possible if the brute force technique of independently translating source than would be possible. if the brute force technique of independently translating source code statements were used.

1. Lexical analysis: The lexical analysis is also called scanning. It is the phase of compilation in which the complete source code is scanned and your source program is broken up into group of string called token.

2. Syntax analysis: The syntax analysis is also called parsing. In this phase the tokens generated by the lexical analyser are grouped together to form a hierarchical structure. The syntax analysis determines the structure of the source string by grouping the token together.

3. Semantic analysis: Once the syntax is checked in the syntax analyser phase the next phase i.e. the semantic analysis determines the meaning of the source string. For example meaning of source string means matching of parenthesis in the expression, or matching of if .... else statement or performing arithmatic operations of the expressions that that are type compatible, or checking the scope of operation.

4 Intermediate code generation:
The intermediate code is a kind of code which is easy to generate and this code can be easily converted to target code. This code is in variety of forms such as three address code, quardruple, triple, posix.

5. Code optimization: The code optimization phase attempts to improve the intermediate code. This is necessary to have a faster executing code or less consumption of memory.

6. Code generation: In code generation phase the target code gets generated. The intermediate code instructions are translated into sequence of machine instruction.

Program Level Performance Analysis:

⇨ Execution time of a program often varies with the input data values because those values select different execution paths in the program.

⇨ Cache has a major effect on program performance.

⇨ Execution times may vary even at the instruction level.

⇨ Floating-point operations are the most sensitive to data values, but the normal integer execution pipeline can also introduce data-dependent variations.

- Program performance is measured in following ways:

1. Simulator of CPU supplied by manufacture.
2. Timer connected to the microprocessor bus can be used to measure performance of executing sections of code.
3. A logic analyzer can be connected to the microprocessor bus to measure the start and stop times of a code segment.

Elements of Program Performance.

- Program execution time is given as

Execution time = Program path + Instruction timing

- The path is the sequence of instructions executed by the program. This instruction timing is determined based on the sequence of instructions traced by the program. path, which takes into account data dependencies.
- Program execution times depend on several factors:

1. **Input data values:** Different values, different execution paths.
2. **Cache behavior:** Also dependent on input values.
3. **Instruction level:** Floating-point operations and pipelining effects.

⇒ Program paths offer insight into a program's dynamic behavior that is difficult to achieve any other way. Unlike simpler measures such as program profiles, which aggregate information to reduce the cost of collecting or storing data, paths capture some of the usually invisible dynamic sequencing of statements.

⇒ Examination of programs path has unveiled a striking degree of path locality, which the Computer/compiler communicates have potentially exploited to increase program performance.

Software Performance Optimization

loop Optimizations.

- Optimizing loops particularly important in compilation, since loop account for much of the executions times of many programs.
- The code optimization can be significantly done in loop of the program. Specially inner loop is a place where program spends large amount of time.
- Hence if number of instructions are less inner loop then the running time of the program will get decreased to a large extent. Hence loop optimization is a technique in which code optimization performed on inner loops.
- Three methods are used for loop optimizations: **code motion, instruction variable elimination, and strength reduction.**

1. code motion: It is a technique which moves the code outside the loop. Hence is the name. If there is a technique which moves the code. If there lies some expression in the loop whose result remains unchanged even after executing the loop for several times, then such an expression should be placed just before the loop

- Here before the loop means at the entry of the loop.

```
while (i <= Max-1)
{
  sum = sum + a[i];
}
```

Before optimization

```
n = Max-1
while (i<=n)
{
  sum = sum+a[i];
}
```

After optimization.

2. Instruction Variables

* An induction variable is a variable in a loop, whose value is a function of the loop iteration number $V = f(i)$
* A variable x is called an induction variable of loop L if the value of variable gets changed every time. It is either decremented or incremented by some constant.

For example:

```
B1
i := i + 1
t1 := 4j
t2 := a[t1]
if t2 < 10 goto B1
```

* In above code the values of i & t_1 are in locked state. That is when value of i gets incremented by 1 then t_1 gets incremented by 4. Hence i and t_4 are induction variables.

3. Strength reduction:

The strengths of certain operators is higher than others. For instance strength of * is higher than +. In strength reduction technique the higher strength the higher strength operators can be replaced by lower strength operations.

Program Level Energy and Power Analysis and optimization :

* Power consumption is a particularly important design metric for battery-powered systems because the battery has a very limited lifetime.

* Power consumed by the CPU is a major part of the total power consumption of a computer system and thus has been the main target of power consumption analysis.

* How long the device needs to run and whether the batteries can be recharged, need to be thought out head of time. In some systems, replacing a battery in a device can be a big expense.

* There are several methods to conserve power in an embedded system, including clock control, power-sensitive processors, low-voltage ICs and circuit shutdown.

* By measuring the current drawn by the processor as it repeatedly executes distinct instructions or distinct instruction sequences, it is possible to obtain most of the information that is required to evaluate the power consumption.

* Power is modeled as a base cost for each instruction plus the inter-instruction overheads that depend on neighboring instructions.

* The base cost of an instruction can be considered as the cost associated with the basic processing needed to execute the instruction

* However, when sequences of instructions are considered, certain inter-instruction effects come into play, which are not reflected in the cost computed ~~of~~ solely from base cost.

* **1. Circuit state**: Switching activity depends on the current inputs and previous circuit state.

2. Resource constraints: Resource constraints in the CPU can lead to stalls e.g. pipeline stalls and write buffer stalls.

3 Cache misses: Another inter-instruction effects is the effect of cache misses.

- As the instruction cache size increases, the energy cost of the software on the CPU declines, but the instruction cache comes to dominate the energy consumption.

* If the cache is too small, the program runs slowly and the system consumes a lot of power due to the high cost of main memory accesses.

* If the cache is too large, the power consumption is high without a corresponding payoff in performance. At intermediate values the execution time and power consumption are both good.

Methods for improving energy consumption:

1. Try to use registers efficiently
2. Analyze cache behaviour to find major cache conflicts.
3. Make use of page mode accesses in the memory system whenever possible.

* Some additional observations about energy optimization as follows:

1. Moderate loop unrolling eliminates some loop control overhead.
2. Software pipelining reduces pipeline stalls, thereby reducing the average energy per instruction.
3. Eliminating recursive procedure calls where possible saves power by getting rid of function call overhead.

Analysis and Optimization of program Size:

- Data provide an excellent opportunity for minimizing size because the data are most highly dependent on programming style.
- In data dominated applications, such as image or speech signal processing applications. summing up the sizes of all the arrays is the most straightforward way to get an upper bound of the memory requirement.
- In the data dependency relations in the code are used to find the number of array elements produced or consumed by each assignment, from which a memory trace of upper and lower bounding rectangle as a function of time is found.
- Care should be taken while designing buffer size. Data can sometimes be packed. Such as by storing several flags in a single word and extracting them by using bit-level operations.

* A very low-level technique for minimizing data is to reuse values. Data buffers can often be reused at several different points in the program.
* Minimizing the size of the instruction text of a program requires a mix of high-level program transformations and careful instruction selection.
* Encapsulating functions in subroutines. Can reduce program size when done carefully.

Program Validation and Testing:

* Testing is an organized process to verify the behavior, performance, and reliability of a device or system against designed specifications.
* Debugging is the process of removing defects (bugs) in the design phase to ensure that the synthesized design, when manufactured will behave as expected. Testing is a manufacturing step to ensure that the manufactured device is defect free.

⊗ Embedded software development uses specialized compilers and development software that offer means for debugging. Developers build application software on more powerful computers and eventually test the application in the target processing environment.

- Testing methods are of two types.

1. **Black-box testing**: This method generates tests without looking at the internal structure of the program.
2. **White box testing**: This method generate tests looking based on the program structure. This method also called as **clear-box testing**.

Black Box Testing: It is also called functional testing. It is testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.

- With black box testing, the software tester does not have access to the source code itself. The code is considered to be a "big black box" to the tester who can't see inside the box.
- Black-box is based on requirements and functionality, not code.
- Random tests from one category of black-box test. Random values are generated with a given distribution.
- The expected values are computed independently of the system, and then the test inputs are applied. A large number of tests must be applied for the result to be statistically significant, but the tests are easy to generate.
- Using black box testing techniques, testers examine the high-level design and the customer requirements specification to plan the test cases to ensure the code does what it is intended to do.
- Functional testing involves ensuring that the functionality specified in the requirement specification works.

⊗ System testing involves putting the new program in many different environments to ensure the program works in typical customer environments with various versions and types of operating systems and/or applications.

Advantages:

1. Tests the final behavior of the software.
2. Can be written independent of software design.
3. Can be used to test different implementation with minimal changes.

Disadvantages:

1. Doesn't necessarily know the boundary cases.
2. Can be difficult to cover all portions of software implementation.

White Box Testing:

⇨ Often called "**structural**" testing.

⇨ Knowing the internal working of a product, test that all internal operations are performed according to specifications and all internal components have been exercised.

⇨ It involves tests that concentrate on close examination of procedural detail. Logical paths through the software are tested.

⇨ White box testing focuses on the internal structure of the software code. The white box tester knows what the code looks like and writes test causes by executing methods with certain parameters.

⇨ Test cases exercise specific sets of conditions and loops

⇨ A white-box testing technique that focusses exclusively on the validity of loop constructs. Four different classes of loops exist: Simple loops, nested loops, concatenated loops and unstructured loops.

Advantages:

1. Usually helps getting good coverage.
2. Good for ensuring boundary cases and special cases get tested.

Disadvantages: 1. Tests based on design might miss bigger picture system problems.
2. Tests need to be changed if implementation/algorithm changes.
3. Hard to test code that isn't there with white box testing.