EC8791 EMBEDDED AND REAL TIME SYSTEMS

UNIT III EMBEDDED PROGRAMMING 9

Components for embedded programs- Models of programs- Assembly, linking and loading - compilation techniques- Program level performance analysis - Software performance optimization - Program level energy and power analysis and optimization - Analysis and optimization of program size- Program validation and testing.

Unit-III 3-01

Embedded Programming

1. Components for embedded programs:

The components that are commonly used in embedded software are: the state machine, the circular buffer and the queue.

State machines are well suited to reactive systems; circular buffers and queues are useful in DSP.

i) State machines:

When inputs appear intermittenly rather than as periodic samples, it is often convenient to think of the system as reacting to those inputs.

The reaction of most systems can be characterized interms of the input received and the current state of the system. This leads naturally to a finite state machine style of describing the reactive system's behavior.

Finite state machines are usually first encountered in the context of hardware design.

ii) Circular buffers:

The circular buffer is a data structure that lets us handle streaming data in an efficient way.

At each point in time, the algorithm needs a subset of the data stream that forms a window into the stream. Because the size of the window does not change, we can use a fixed size buffer to hold the current data.

To avoid constantly copying data within the buffer we will move the head of the buffer in time.

Every time we add a sample, we automatically overwrite the oldest sample, which is the one that needs to be thrown out. When the pointer gets to the end of the buffer, it wraps around to the top.

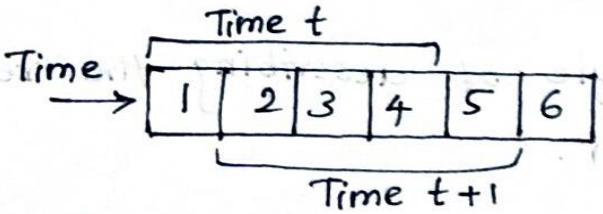


fig: Data stream

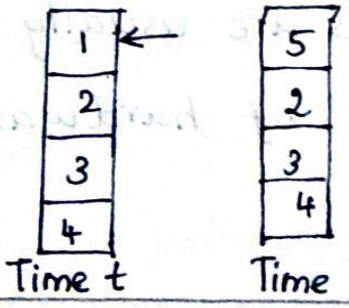


fig: Circular buffer

iii) Queues:

Queues are also used in signal processing and event processing. Queues are used whenever data may arrive and depart at somewhat unpredictable times or when variable amounts of data may arrive.

One way to build a queue is with a linked list. This approach allows the queue to grow to an arbitrary size.

Another way to design the queue is to use an array to hold all the data.

The queues in a producer/consumer may hold either uniform sized data elements or variable sized data elements.

In some cases, the consumer needs to know how many of a given type of data element are associated together.

The queue can be structured to hold a complex data type. Alternatively, the data structure can be stored as bytes or integers in the queue.

2. Models of programs:

Our fundamental model for programs is the control / data flow graph (CDFG). As the name implies, the CDFG has constructs that model both data operations and control operations.

Data flow graphs:

A data flow graph is a model of a program with no conditionals. In a high level programming language, a code segment with no conditionals.

```
w = a+b;
x = a-c;
y = x+d;
x = a+c;
z = y+e;
```

fig: A basic block in C.

There are two assignments to the variable x - it appears twice on the left side of an assignment. We need to rewrite the code in single assignment form, in which a variable appears only once on the left side.

```
w = a+b;
x1 = a-c;
y = x1+d;
x2 = a+c;
z = y+e;
```

fig: The basic block in single assignment form.

The single assignment form is important because it allows us to identify a unique location in the code where each named location is computed.

The value nodes may be either inputs to the basic block, such as a and b, or variables assigned to within the block, such as w and x1.

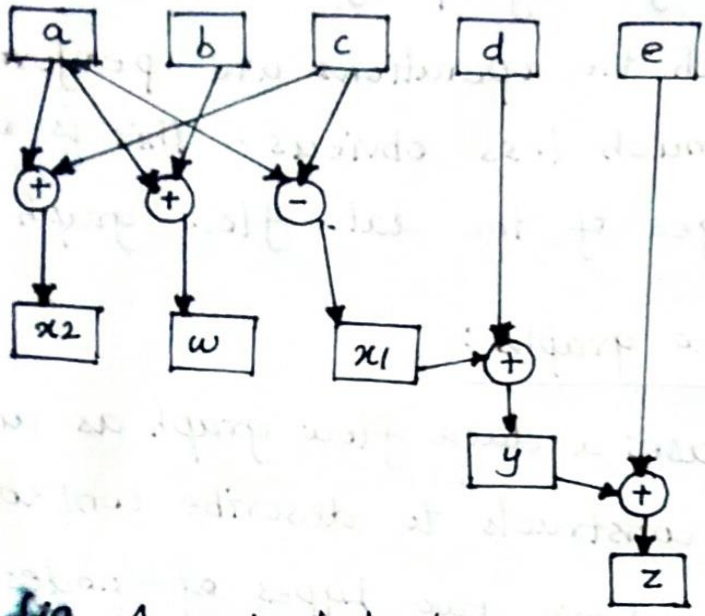


fig: An extended data flow graph for our sample basic block.

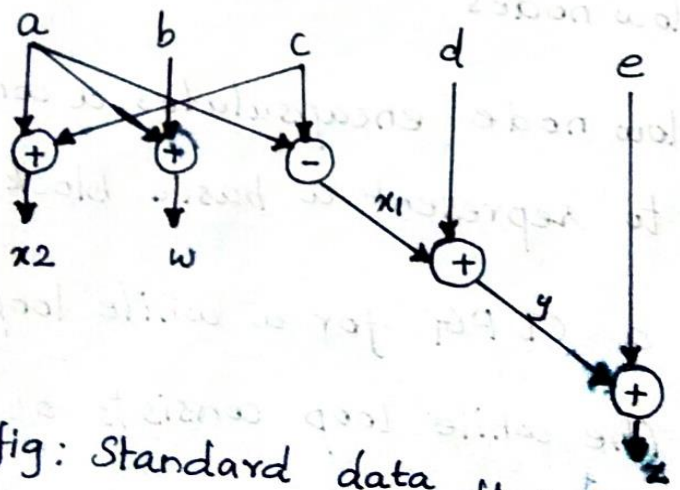


fig: Standard data flow graph for our sample basic block.

In standard flow graph, the variables are not explicitly represented by nodes: Instead, the edges are labeled with the variables they represent. As a result, a variable can be represented by more than one edge.

The data flow graph for the code makes the order in which the operations are performed in the C code much less obvious. This is one of the advantages of the data flow graph.

Control / data flow graphs:

A CDFG uses a data flow graph as an element, adding constructs to describe control. In a basic CDFG, we have two types of nodes:

- decision nodes
- data flow nodes

A data flow node encapsulates a complete data flow graph to represent a basic block.

Building a CDFG for a while loop is straight forward. The while loop consists of both a test and a loop body, each of which we know

how to represent a CDFG.

In C, for loop is defined interms of a while loop.

```
for (i=0; i<N; i++)
{   loop_body();
}
```

is equivalent to

```
i=0;
while(i<N)
{
  loop body();
   i++;
}
```

For a complete CDFG model, we can use a data flow graph to model each data flow node. Thus, CDFG is a hierarchical representation.

The execution model for a ~~CGH~~ CDFG is very much like the execution of the program it represents.

The CDFG does not require explicit declaration of variables but we assume that the implementation has sufficient memory for all the variables. We can define a state variable that represents a program counter in a CPU.

```
if (cond1)
    basic_block_1();
else
    basic_block_2();
basic_block_3();
switch (test1)
{
  case c1; basic_block_4 (); break;
  case c2; basic_block_5 (); break;
  case c3; basic_block_6 (); break;
```

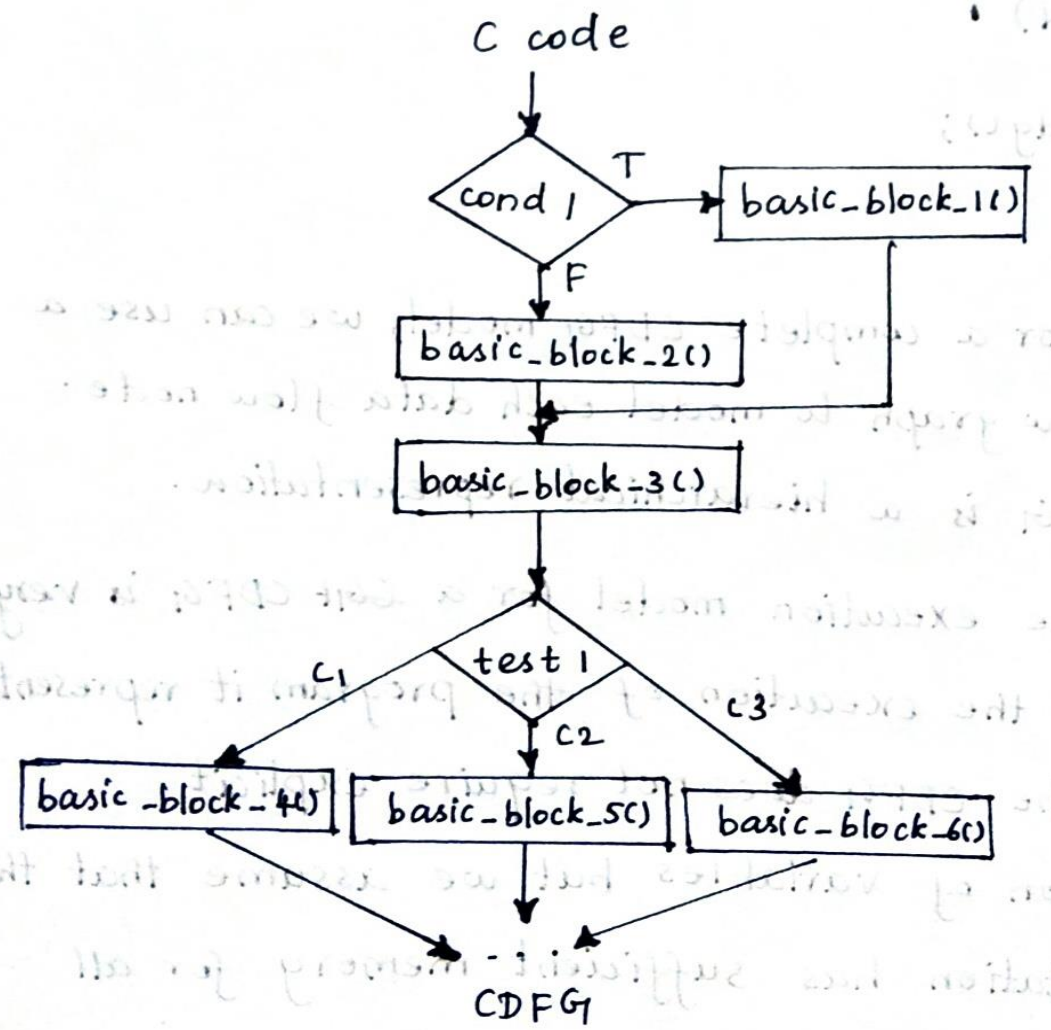
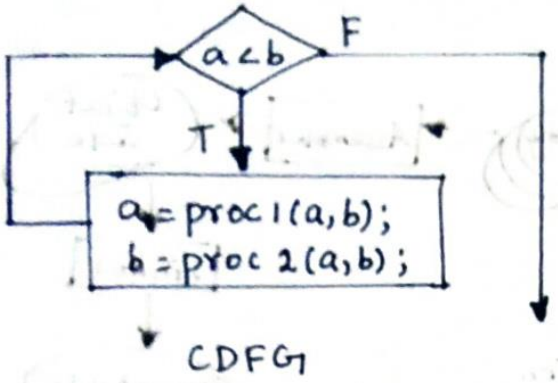


fig: C code and its CDFG.

```
while (a < b) {
    a = proc1(a,b);
    b = proc2(a,b);
}
```

C Code



CDFG

fig: A while loop & its CDFG.

The CDFG is not necessarily tied to high level language control structures. We can also build a CDFG for an assembly language program. A jump instruction corresponds to a nonlocal edge in the CDFG. Some architectures such as ARM and many VLIW processors, support predicated execution of instructions, which may be represented by special constructs in the CDFG.

3. Assembly, linking and loading: 3+0

Assembly and linking are the last steps in the compilation process. Loading actually puts the program in memory so that it can be executed.

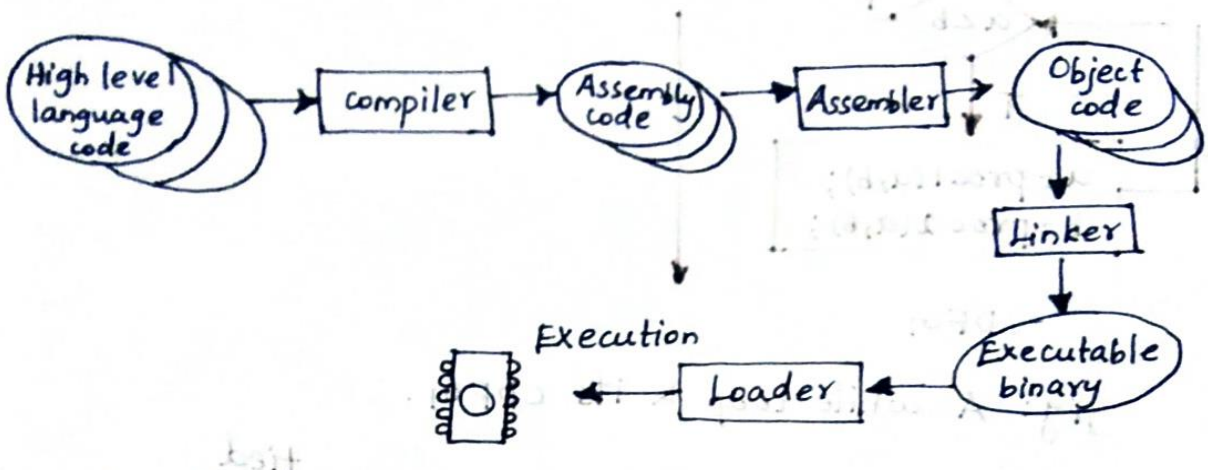


fig: Program generation from compilation through loading.

The compilation process is often hidden from us by compilation commands that do everything required to generate an executable programs.

The assembler's job is to translate symbolic assembly language statements into bit-level representation of instructions known as object code.

The assembler takes care of instruction formats and does part of the job of translating labels into addresses.

The program that brings the program into memory for execution is called a loader.

The simplest form of the assembler assumes that the starting address of the assembly languge program has been specified by the programmer. The addresses in such a program are known as absolute addresses.

Most assemblers allows us to use relative addresses by specifying at the start of the file that the origin of the assembly language module is to be computed later.

Assemblers:

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

Labels make the assembly process more complex, but they are the most important abstraction provided by the assembler.

Label processing requires making two passes through the assembly source code:

i) The first pass scans the code to determine the address of each label.

ii) The second pass assembles the instructions using the label values computed in the 1st pass.

```
        add r0,r1,r2
PLC →   xx add r3,r4,r5
        cmp r0,r3
        yy sub r5,r6,r7
```

Assembly code

xx	0x8
yy	0x10

Symbol table

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.

During scanning, the current location in memory is kept in a Program Location Counter (PLC).

The PLC is not used to execute the program only to assign memory locations to labels.

Assemblers allow labels to be added to the symbol table without occupying space in program memory.

Linking :

Many assembly language programs are written as several smaller pieces rather than as a single large file. Breaking a larger program into smaller files helps delineate program modularity.

A linker allows a program to be stitched together out of several smaller pieces. The linker operates on the object files created by the assembler and modifies the assembled code to make the necessary links between files.

Some labels will be both defined and used in the same file. Other labels will be defined in a single file but used elsewhere.

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 references based on available entry points.

```
label 1  LDR r0,[r1]
         ...
         ADR a
         ...
         B label 2
         ...
var1     %.1
```

```
label 2  ADR var1
         ...
         B label 3
         ...
x        %.1
y        %.1
a        %.10
```

External references	Entry points
a	label 1
label 2	var 1

File 1

External references	Entry points
var 1	label 2
label 3	x
	y
	a

File 2

fig: External references and entry points.

The linker proceeds in two phases.

Object code design:

We need to control the placement of several types of data:

- Interrupt vectors and other information for I/O devices must be placed in specific locations.
- Memory management tables must be set up.

• Global variables used for communication between processes must be put in locations that are accessible to all the users of that data.

Many programs should be designed to be reentrant. A program is reentrant if it can be interrupted by another call to the function without changing the results of either call. If the program changes the value of global variables, it may give a different answers when it is called recursively.

A program is relocatable if it can be executed when loaded into different parts of memory.

Relocabability requires some part of support from hardware that provides address calculation. But it is possible to write nonrelocatable code for nonrelocatable architectures.

In some cases, it may be necessary to use a nonrelocatable address such as when addressing an I/o device.

4. Compilation Techniques:

The Compilation process:

It is useful to understand how a high level language program is translated into instructions: interrupt handling instructions, placement of data and of instructions in memory etc.

We can summarize the compilation process with a formula:

> Compilation = translation + optimization

The high level language program is translated into the lowest level form of instruction: optimizations try to generate better instruction sequences that would be possible if the brute force technique of independantly translating source code statements we used.

Optimization techniques focus on more of the program to ensure that compilation decisions that appear to be good for one statement are not unnecessarily problematic for other parts of the program.

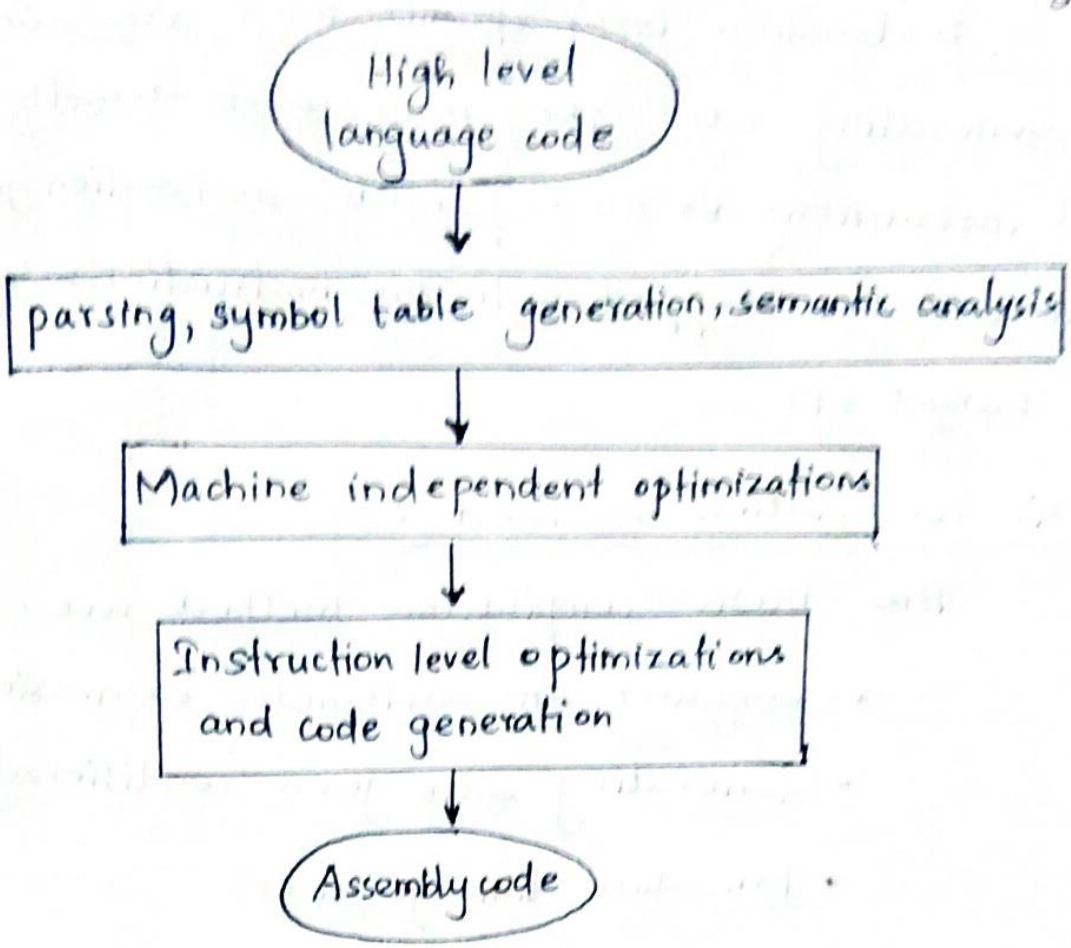


fig: The compilation process

Compilation begins with high level language code such as C or C++ and generally produces assembly code.

The high level language program is parsed to break it into statements and expressions. In addition a sympol table is generated which includes all the named objects in the program.

Simplifying arithmetic expressions is one example of a machine independent optimizations.

Instruction level optimizations are aimed at generating code. They may work directly on real instructions or on a pseudo-instruction format that is later mapped onto the instructions of the target CPU.

Basic Compilation methods:

The basic compilation methods are:

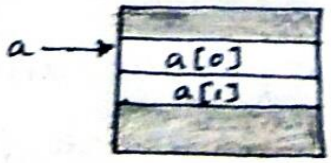
- Compiling an arithmetic expressions
- Generating code for a conditional
- Procedure linkage in C.

The compiler must also translate references to data structures into references to raw memories. In general, this requires address computations.

Arrays are interesting because the address of an array element must in general be computed at runtime, because, the array index may change.

e.g) a[i]

The layout is given by:



The zeroth element is stored as the first element of the array; we can create a pointer for the array that points to the array's head namely a[0]. If we call the pointer aptr, we can rewrite the reading of a[1] as

*(aptr+1).

Two dimensional arrays more challenging. There are multiple possible ways to layout a two dimensional array in memory.

Let us consider the row-major form. If the a[] array is of size N x M, then we can turn the two dimensional array access into a one-dimensional array access, thus, a[i,j] becomes a[i*M+j] where the maximum value for j is M-1.

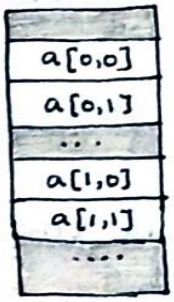


fig. Memory layout for two dimensional arrays.

The addition [*(aptr+4)] can usually done at compile time, requiring only the indirection itself to fetch the memory location during execution.

Compiler optimizations:

Basic compilation techniques can create inefficient code. Compilers use a wide range of algorithms to optimize the code they generate.

A simple but useful transformation is known as loop unrolling. It is important because it helps expose parallelism that can be used by later stages of the compiler.

Loop fusion combines two or more loops into a single loop. For this transformation, two conditions must be satisfied.

- The loops must iterate over the same values.
- The loop bodies must not have dependencies that would be violated if they are executed together.

Loop distribution is the opposite of loop fusion, that is decomposing a single loop into multiple loops.

Loop tiling breaks up a loop into a set of nested loops with each inner loop performing the operations on a subset of the data.

Array padding adds dummy data elements to a loop inorder to change the layout of the array in the cache.

Dead code is the code that can never be executed. Dead code can be generated by the programmers and compilers purposefully.

Deadcode can be identified by reachability analysis - finding the other statements or instructions from which it can be reached.

Deadcode elimination analyzes code for reachability and trims away dead code.

Register allocation is a very important compilation phase. If a section of code requires more registers than are available, we must spill some of the values out to memory temporarily.

We can solve register allocations problems by building a conflict graph and solving a graph coloring problem.

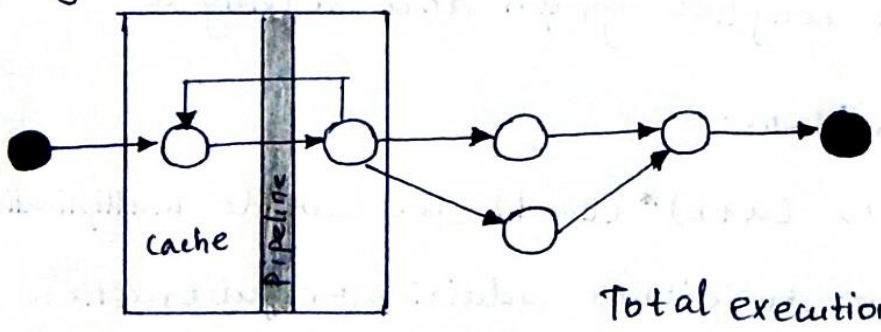
Consider, (a+b)*(c-d) we can do multiplication last but we can do either addition or subtraction first.

5. Program level performance analysis:

Because embedded systems must perform functions in real time, we often need to know how fast a program runs.

It is important to keep in mind that CPU performance is not judged in the same way as program performance.

- The execution time of a program often varies with the input data values because those values select different execution paths in the program. For example, loops may be executed a varying number of times and different branches may execute blocks of varying complexity.
- The cache has a major effect on program performance and once again, the cache's behavior depends in part on the data values input to the program.



• The execution time of an instruction in a pipeline depends not only on that instruction but on the instructions around it in the pipeline.

We can measure program performance in several ways:

• Some microprocessor manufacturers supply simulators for their CPUs. The simulators run on a workstation or PC, takes as input an executable for the microprocessor along with input data and simulates the execution of the program.

• A timer connected to the microprocessor bus can be used to measure performance of executing sections of code.

• A logic analyzer can be connected to the microprocessor bus to measure the start and stop times of a code segment.

There are three different types of performance measures on programs:

i) Average case execution time;

This is the typical execution time we would expect for typical data. Clearly, the first challenge is defining typical inputs.

ii) Worst case execution time:

The longest time that the program can spend on any input sequence is clearly important for systems that must meet deadlines.

iii) Best case execution time:

This measure can be important in multirate real time systems.

Elements of program performance:

The key to evaluating execution time is breaking the performance problem into parts. Program execution time [Sha89] can be seen as:

execution time = program path + instruction timing.

The path is the sequences of instructions executed by the program. The instruction timing is determined based on the sequence of instructions.

The simplest estimate of instruction timing is to assume that every instruction takes the same number of clock cycles ie) we need only to count the instructions and multiply by the per instruction execution time to obtain the program's total execution time.

- Not all the instructions take the same amount of time.
- Execution times of instructions are not independent.
- The execution time of an instruction may depend on operand values.

Measurement driven performance analysis:

The most direct way to determine the execution time of a program is by measuring it. The drawbacks are inorder to cause the program to execute its worst case execution path, we have to provide proper inputs to it.

Most methods of measuring program

performance combine the determination of the execution path and the timing of that path as the program executes, it chooses a path and we observe the execution time along that path and considering that program as the program trace.

The biggest problem in measuring program performance is figuring out a useful set of inputs to give the program. The other problem with input data is the software scaffolding.

Profiling is a simple method for analyzing software performance. A profiler does not measure execution time – instead it counts the number of times that procedures or basic blocks in the program are executed.

The alternative to physical measurement of execution time is simulation. A CPU sim

For purpose of performance analysis, the most important type of CPU simulator is the cycle-accurate simulator which performs a sufficiently

detailed simulation of the processor's internals that it can determine the exact number of clock cycles required for execution.

A simulator that functionally simulates instructions but does not provide timing information is known as an instruction level simulator.

6. Software Performance Optimization:

Loop Optimizations:

Loops are important targets for optimization because programs with loops tend to spend a lot of time executing those loops.

There are three important techniques in optimizing loops:

- code motion
- induction variable elimination
- strength reduction.

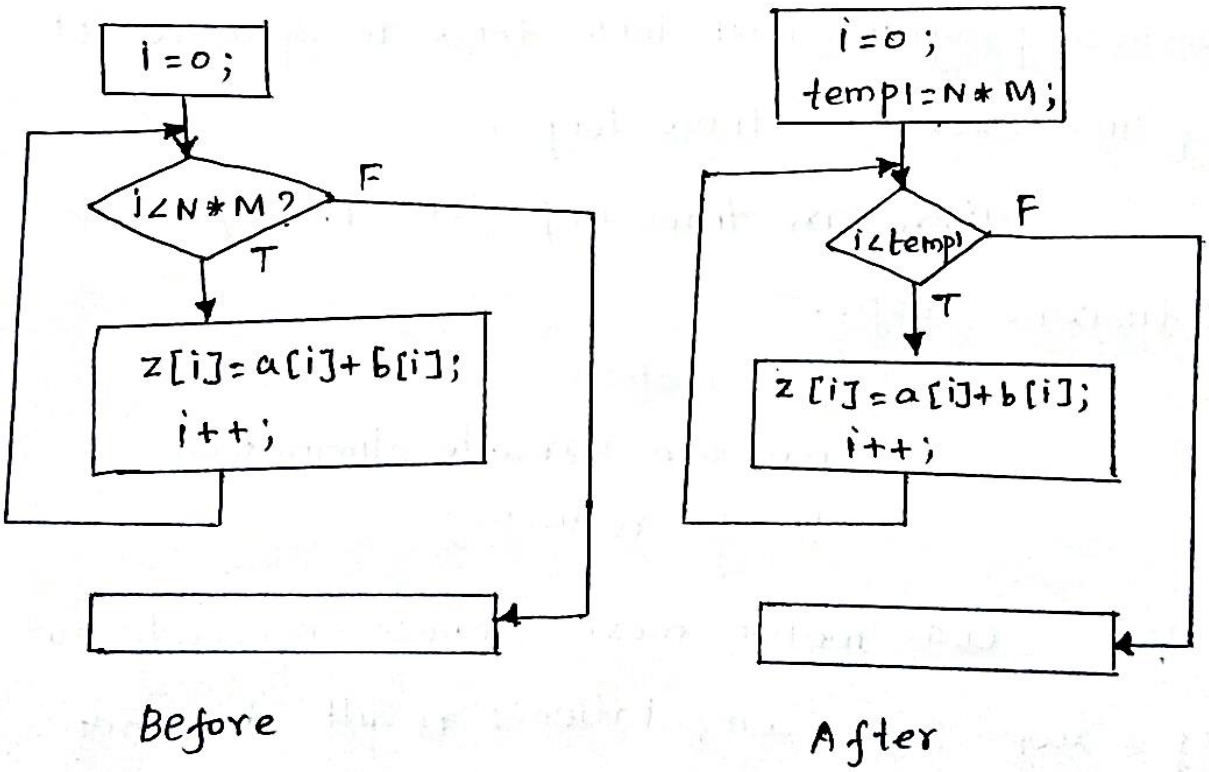
Code motion moves unnecessary code out of a loop. If a computation's result does not depend on operations performed in the loop body, then we can safely move it out of the loop.

e.g)
```
for (i=0; i<N*M; i++) {
    z[i] = a[i]+b[i];
}
```

An induction variable is a variable whose value is derived from the loop iteration variable's value. The compiler often introduces induction variables to help it implement the loop.

A nested loop is a good example:

```
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
        z[i][j] = b[i][j];
```



Before After

fig: Code motion in a loop.

```
for (i=0; i<N; i++)
    for (j=0; j<M; j++) {
        zbinduct = i * M + j;
        *(zptr + zbinduct) = *(bptr + zbinduct);
        }
```

In the above code zptr and bptr are pointers to the heads of the z and b arrays and zbinduct is the shared variable.

```
zbinduct = 0;
for (i=0; i<N; i++) {
    for (j=0; j<M; j++) {
        *(zptr + zbinduct) = *(bptr + zbinduct);
        zbinduct++;
        }
    }
```

This is a form of strength ~~indu~~ reduction because we have eliminated the multiplication from the induction variable computation.

Strength reduction helps us to reduce the cost of loop iteration.

e.g) y = z * 2;

Cache Optimizations:

A loop nest is a set of loops one inside the other. Loop nests occur when we process arrays.

A large body of techniques has been developed for optimizing loop nests. Rewriting a loop nests changes the order in which array elements are accessed.

Performance Optimization Strategies:

Few hints on program implementation are:

- Try to use registers efficiently.
- Make use of pagemode access in the memory system whenever possible.
- For instruction conflicts, try to rewrite the segment to make it as small as possible.
- For scalar data conflicts, move the data values to different locations to reduce conflicts.
- For array data conflicts, consider either moving the arrays or changing your array access patterns to reduce conflicts.

7. Program level energy and power analysis and Optimization:

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

The opportunities for saving the power are:

- We may able to replace the algorithms with others that do things in clever ways that consumes less power.
- Memory accesses are a major component of power consumption in many applications. By optimizing memory accesses we may be able to significantly reduce power.
- We may able to turn off parts of the system such as subsystems of the CPU, when we don't need them inorder to save power.

Several factors contribute to the energy consumption of the program are:

- Energy consumption varies somewhat.

from instruction to instruction.

- The sequence of instructions has some influence.
- The opcode and the location of the operands also matter.

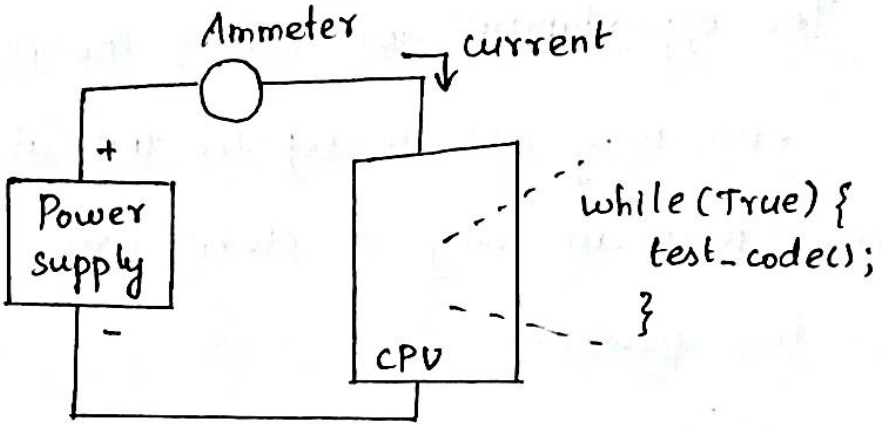


fig: Measuring power consumption for a piece of code

Choosing which instructions to use can make some differences in a program's energy consumption but concentrating on the instruction opcodes has limited payoffs in most CPUs.

In many applications, the biggest payoff in energy reduction for a given amount of designer effort comes from concentrating on the memory system.

Caches are an important factor in energy consumption. On the one hand, a cache hit saves a costly main memory access and on the other, the cache itself is relatively power hungry because It is built from SRAM not DRAM.

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 programs run slowly and the system consumes a lot of power due to the high cost of main memory accesses.

For improving energy consumption,

- Try to use registers efficiently.
- For scalar data conflicts, move the data values to different locations to reduce conflicts.
- For array data conflicts, consider either moving the arrays or changing your array access patterns to reduce conflicts.

8. Analysis and optimization of program size:

The memory footprint of a program is determined by the size of its data and instructions. Both must be considered to minimize program size.

Data provide an excellent opportunity for minimizing size because the data are most highly dependent on programming style.

A very low level technique for minimizing data is to reuse values. For instance, if several constants happen to have the same value, they can be mapped to the same location.

Data buffers can 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 transformation and careful instruction selection. Encapsulating functions in subroutines can reduce program size when done carefully.

When reducing the number of instructions in a program, one important technique is the proper use of subroutines. If the program performs identical operations repeatedly, these operations are natural candidates for subroutines.

Some microprocessor architectures support dense instruction sets, specially designed instruction sets that use shorter instruction formats to encode the instructions.

The ARM thumb instruction set and the MIPS-16 instruction set for the MIPS architecture are two examples of this type of instruction set.

Special compilation modes produce the program interms of the dense instruction set. Program size of course varies with the type of program, but programs using the dense instruction set are often 70-80% of the size of the standard instruction set equivalents.

9. Program Validation and Testing:

Complex systems need testing to ensure that they work as they are intended.

We can use various combinations of two major types of testing strategies.

i) Black-box methods generate tests without looking at the internal structure of the program.

ii) Clear-box also known as white box, methods generate tests based on the program structure.

Clear box testing:

The control / data flow graph extracted from a program's source code is an important tool in developing clear-box tests for the program.

Inorder to execute and evaluate these tests, we must be able to control variables in the program and observe the results of computations much as in manufacturing testing. In general we need to modify the program to make it more testable.

No matter what we are testing, we must accomplish the following three things in a list:

- Provide the program with inputs that exercise the test we are interested in.
- Execute the program to perform the test.
- Examine the outputs to determine whether the test was successful.

The next task is to determine the set of tests to be performed. We need to perform many different types of tests to be confident that we have identified a large fraction of the existing bugs.

Graph theory helps us get a quantitative handle on the different paths required. In an undirected graph, we can form any path through the graph from combinations of basic paths.

The graph is represented as an incidence matrix. Each row and column represents a node. We can use standard linear algebra techniques to identify the basis set of the graph.

A simple measure, cyclomatic complexity [McC76] allows us to measure the control complexity of the program.

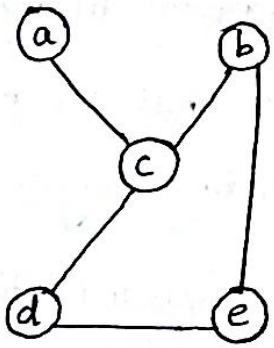
The cyclomatic complexity is given by,

$$M = e - n + 2p \quad \text{——①}$$

where, e → number of edges in the flow graph

n → number of nodes

p → number of components in the graph.



Graph

	a	b	c	d	e
a	0	0	1	0	0
b	0	0	1	0	1
c	1	1	0	1	0
d	0	0	1	0	1
e	0	1	0	1	0

Incidence matrix.

	a	b	c	d	e
a	1	0	0	0	0
b	0	1	0	0	0
c	0	0	1	0	0
d	0	0	0	1	0
e	0	0	0	0	1

Basis set

fig: Matrix representation of graph & basis set.

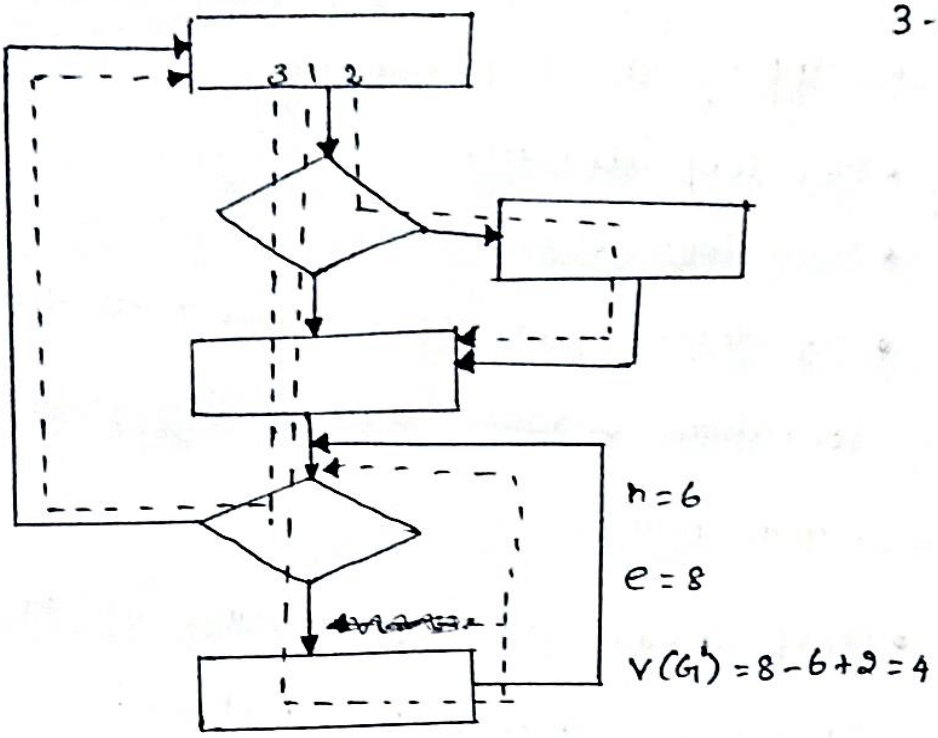


fig: Cyclomatic Complexity.

Another way of looking at control flow oriented testing is to analyze the conditions that control the conditional statements.

```
if ((a==b) || (c >=d) (...)
```

A simple condition testing strategy is known as branch testing.

Frankl and Weyuker (Fra88) have defined criteria for choosing which def-use pairs to exercise to satisfy a well behaved adequacy criterion.

There are several important cases that we should try at a minimum:

- Skipping the loop entirely.
- One loop iteration
- Two loop iterations.
- If there is an upper bound n on the number of loop iterations a value that is significantly below that maximum number of iterations.
- Tests near the upper bound on the number of loop iterations that is n-1, n, n+1.

Black box testing:

Black box tests are generated without knowledge of the code being tested.

Random tests form 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 inputs are applied. These tests are easy to generate.

Regression tests form an extremely important category of tests. When tests are created during earlier stages in the system design or for previous versions of the system, those tests saved to apply on later versions.