UNIT III
EMBEDDED PROGRAMMING

Components for Embedded Program

In this section, we consider code for three structures or components that are commonly used in embedded software: the state machine, the circular buffer, and the queue. State machines are well suited to **reactive systems** such as user interfaces; circular buffers and queues are useful in digital signal processing.

State Machines

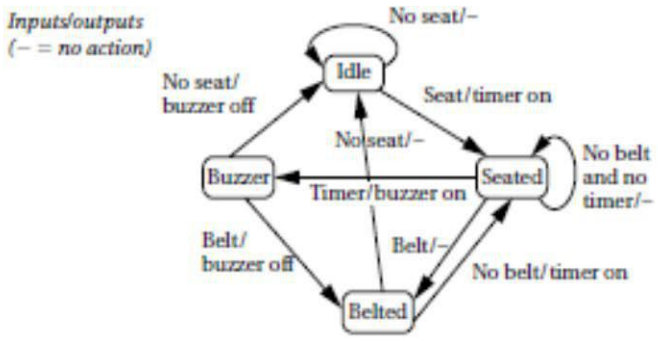
When inputs appear intermittently rather than as periodic samples, it is often convenient to think of the system as reacting to those inputs.

The reaction of most systems can be characterized in terms 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 behaviour.

Moreover, if the behaviour is specified in that way, it is natural to write the program implementing that behaviour in a state machine style.

The state machine style of programming is also an efficient implementation of such computations. Finite-state machines are usually first encountered in the context of hardware design.

A software state machine



Stream-Oriented Programming and Circular Buffers

The data stream style makes sense for data that comes in regularly and must be processed on the fly. For each sample, the filter must emit one output that depends on the values of the last n inputs. In a typical workstation application, we would process the samples over a given interval by reading them all in from a file and

then computing the results all at once in a batch process. In an embedded system we must not only emit outputs in real time, but we must also do so using a minimum amount of memory.

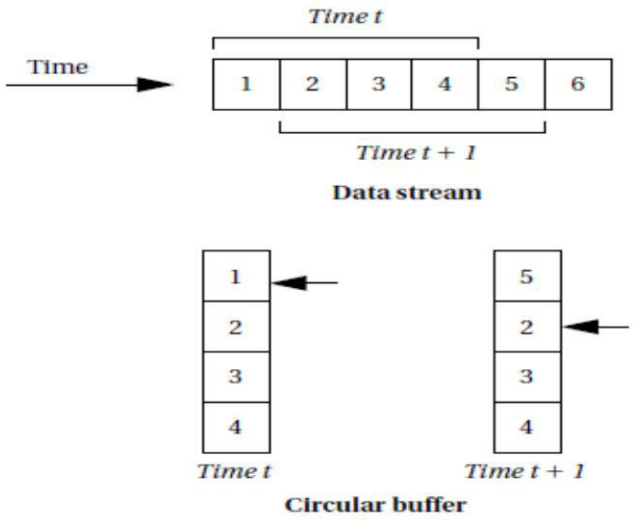


Fig. A circular buffer for streaming data

The circular buffer is a data structure that lets us handle streaming data in an efficient way. Figure illustrates how a circular buffer stores a subset of the data stream. At each point in time, the algorithm needs a subset of the data stream that forms a window into the stream. The window slides with time as we throw out old values no longer needed and add new values. Since 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. The buffer points to the location at which the next sample will be placed; 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.

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. A queue is often referred to as an ***elastic buffer***. One way to build a queue is with a linked list. This approach allows the queue to grow to an arbitrary size. But in many applications we are unwilling to pay the price of dynamically allocating memory. Another way to design the queue is to use

Models of programs

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—more precisely,

with only one entry and exit point is known as a basic block. Figure 2.14 shows a simple basic block. As the C code is executed, we would enter this basic block at the beginning and execute all the statements.

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

A basic block in C.

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

The basic block in single-assignment form

Before we are able to draw the data flow graph for this code we need to modify it slightly. 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.
Since our specification is C code, we assume that the statements are executed sequentially, so that any use of a variable refers to its latest assigned value. In this case, x is not reused in this block (presumably it is used elsewhere), so we just have to eliminate the multiple assignment to x. The result is shown in Figure 2.14, where we have used the names x1 and x2 to distinguish the separate uses of x.
The single-assignment form is important because it allows us to identify a unique location in the code where each named location is computed. As an introduction to the data flow graph, we use two types of nodes in the graph round nodes denote operators and square nodes represent values.
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.

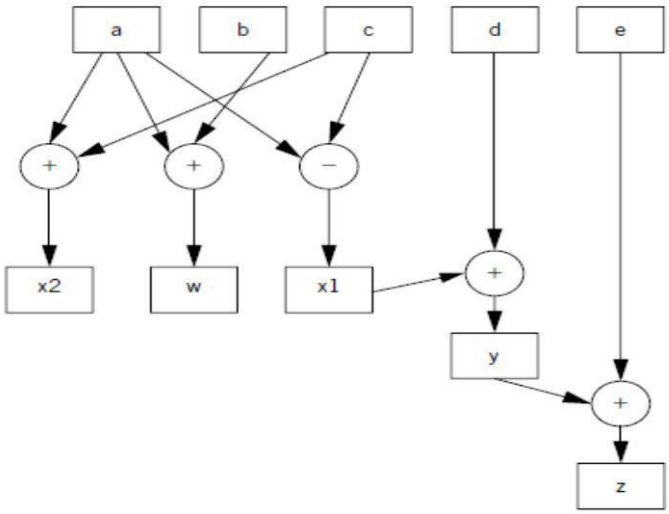


Fig.An extended data flow graph for our sample basic block

The data flow graph for our single-assignment code is shown in Figure 2.15.The single-assignment form means that the data flow graph is acyclic—if we assigned to x multiple times, then the second assignment would form a cycle in the graph including x and the operators used to compute x.

Control/Data Flow Graphs

A CDFG uses a data flflow graph as an element,adding constructs to describe control.

In a basic CDFG, we have two types of nodes: ***decision nodes*** and ***data flflow nodes***. A data flflow node encapsulates a complete data flflow graph to represent a basic block.We can use one type of decision node to describe all the types of control

in a sequential program. (The jump/branch is, after all, the way we implement all those high-level control constructs.)

Figure 5.6 shows a bit of C code with control constructs and the CDFG con structed from it. The rectangular nodes in the graph represent the basic blocks. The basic blocks in the C code have been represented by function calls for simplic ity. The diamond-shaped nodes represent the conditionals. The node's condition is given by the label, and the edges are labeled with the possible outcomes of evaluating the condition.

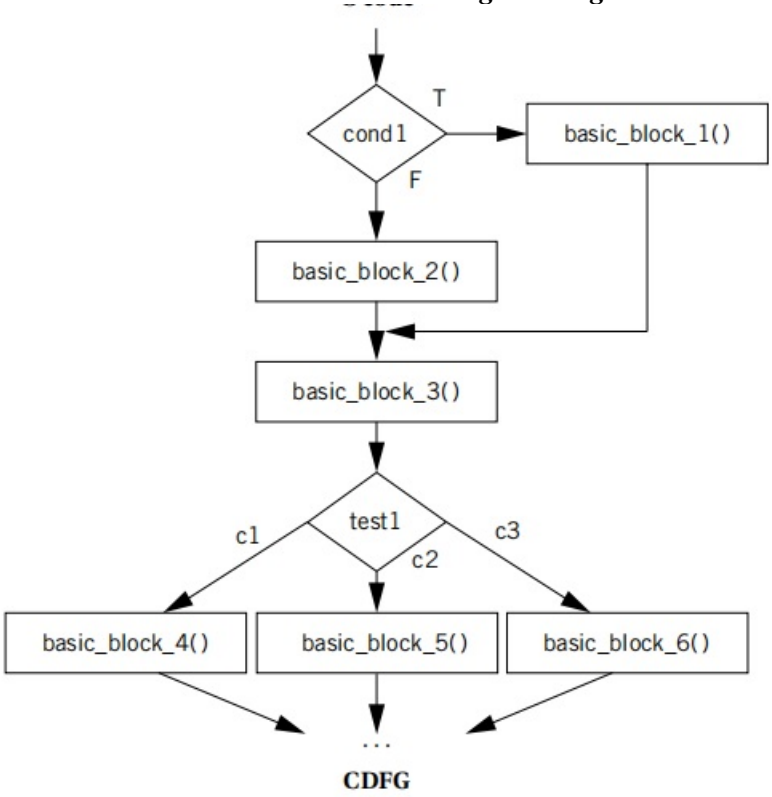
Building a CDFG for a while loop is straightforward, as shown in Figure 5.7. The while loop consists of both a test and a loop body, each of which we know how to represent in a CDFG. We can represent for loops by remembering that, in C, a for loop is defifined in terms of a while loop. The following for loop

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

is equivalent to

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

```
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;
}
```



CDFG

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. Loading actually puts the program in memory so that it can be executed. In this section, we survey the basic techniques required for assembly linking to help us understand the complete compilation and loading process.

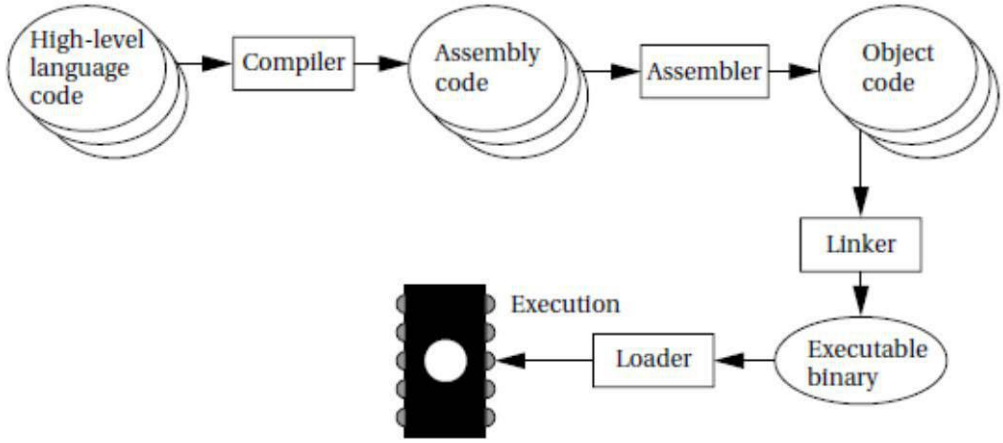


Fig. Program generation from compilation through loading.

Figure highlights the role of assemblers and linkers in the compilation process. This process is often hidden from us by compilation commands that do everything required to generate an executable program. As the figure shows, most compilers do not directly generate machine code, but instead create the instruction- level program in the form of human-readable assembly language. Generating assembly

language rather than binary instructions frees the compiler writer from details extraneous to the compilation process, which includes the instruction format as well as the exact addresses of instructions and data.
The assembler's job is to translate symbolic assembly language statements into bit-level representations of instructions known as **object code**. The assembler takes care of instruction formats and does part of the job of translating labels into addresses. However, since the program may be built from many files, the final steps in determining the addresses of instructions and data are performed by the linker, which produces an **executable binary** file. That file may not necessarily be located in the CPU's memory, however, unless the linker happens to create the executable directly in RAM. 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 language program has been specified by the programmer. The addresses in such a program are known as **absolute addresses**.

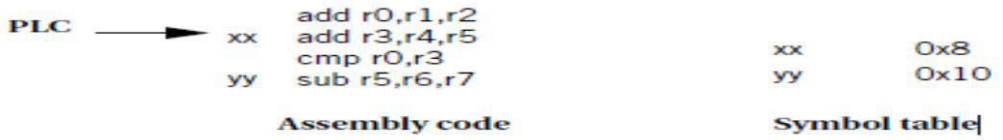
Assemblers
When translating assembly code into object code, the assembler must translate opcodes and format the bits in each instruction, and translate labels into addresses. In this section, we review the translation of assembly language into binary.

Labels make the assembly process more complex, but they are the most important abstraction provided by the assembler. Labels let the programmer (a human programmer or a compiler generating assembly code) avoid worrying about the locations of instructions and data. Label processing requires making two passes through the assembly source code as follows:

1. The first pass scans the code to determine the address of each label.
2. The second pass assembles the instructions using the label values computed in the first pass.
As shown in Figure 2.17, 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)**. Despite the similarity in name to a program counter, the PLC is not used to execute the program, only to assign memory locations to labels. For example, the PLC always makes exactly one pass through the program, whereas the program counter makes many passes over code in a loop. Thus, at the start of the first pass,

the PLC is set to the program's starting address and the assembler looks at the first line. After examining the line, the assembler updates the PLC to the next location (since ARM instructions are four bytes long, the PLC would be incremented by four) and looks at the next instruction. If the instruction begins with a label, a new entry is made in the symbol table, which includes the label name and its value. The

value of the label is equal to the current value of the PLC. At the end of the first pass, the assembler rewinds to the beginning of the assembly language file to make the second pass. During the second pass, when a label name is found, the label is looked up in the symbol table and its value substituted into the appropriate place in the instruction.



But how do we know the starting value of the PLC? The simplest case is absolute addressing. In this case, one of the first statements in the assembly language program is a pseudo-op that specifies the **origin** of the program, that is, the location of the first address in the program. A common name for this pseudo-op (e.g., the one used for the ARM) is the ORG statement.

ORG 2000

Which puts the start of the program at location 2000. This pseudo-op accomplishes this by setting the PLC's value to its argument's value, 2000 in this case. Assemblers generally allow a program to have many ORG statements in case instructions or data must be spread around various spots in memory.

Linking:

Many assembly language programs are written as several smaller pieces rather than as a single large file. Breaking a large program into smaller files helps delineate program modularity. If the program uses library routines, those will already be preassembled, and assembly language source code for the libraries may not be available for purchase.

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 as illustrated in Figure 2.18. 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. As a result of the need to know how definitions and references connect, the assembler passes to the linker not only the object file but also the symbol table.

Even if the entire symbol table is not kept for later debugging purposes, it must at least pass the entry points. External references are identified in the object code by their relative symbol identifiers.

```
label1   LDR r0,[r1]
         ...
         ADR a
         ...
         B label2
         ...
var1     % 1
```

External references	Entry points
a	label1
label2	var1

File 1

```
label2   ADR var1
         ...
         B label3
         ...
x        % 1
y        % 1
a        % 10
```

External references	Entry points
var1	label2
label3	x
	y
	a

File 2

Fig. External references and entry points

The linker proceeds in two phases.

First, it determines the address of the start of each object file. The order in which object files are to be loaded is given by the user, either by specifying parameters when the loader is run or by creating a **load map** file that gives the order in which files are to be placed in memory. Given the order in which files are to be placed in memory and the length of each object file, it is easy to compute the starting address of each file.

At the start of the second phase, the loader merges all symbol tables from the object files into a single, large table. It then edits the object files to change relative addresses into addresses. This is typically performed by having the assembler write extra bits into the object file to identify the instructions and fields that refer to labels. If a label cannot be found in the merged symbol table, it is undefined and an error message is sent to the user.

COMPILATION TECHNIQUES.

It is useful to understand how a high-level language program is translated into instructions. Since implementing an embedded computing system often requires controlling the instruction sequences used to handle interrupts, placement of data and instructions in memory, and so forth, understanding how the compiler works can help you know when you cannot rely on the compiler.

Next, because many applications are also performance sensitive, understanding how code is generated can help you meet your performance goals, either by writing high-level code that gets compiled into the instructions you want or by recognizing when you must write your own assembly code.

The compilation process is summarized in Figure 2.19. Compilation begins with high-level language code such as C and generally produces assembly code. (Directly producing object code simply duplicates the functions of an assembler which is a very desirable stand-alone program to have.)

The high-level language program is parsed to break it into statements and expressions. In addition, a symbol table is generated, which includes all the named objects in the program. Some compilers may then perform higher-level optimizations that can be viewed as modifying the high-level language program input without reference to instructions.

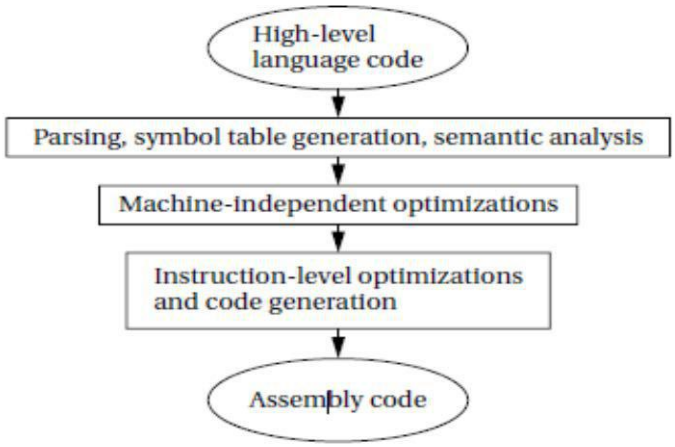


Fig.The compilation process

Simplifying arithmetic expressions is one example of a machine- independent optimization. Not all compilers do such optimizations, and compilers can vary widely regarding which combinations of machine-independent optimizations they do perform.

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. This level of optimization also helps

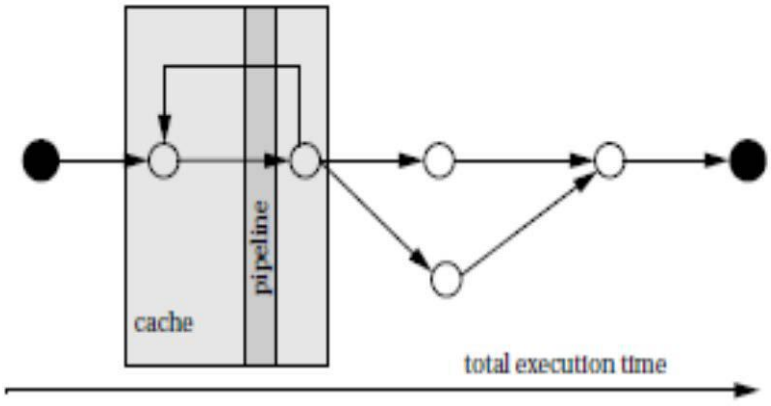
modularize the compiler by allowing code generation to create simpler code that is later optimized. For example, consider the following array access code:

x[i] = c*x[i];

A simple code generator would generate the address for x[i] twice, once for each appearance in the statement. The later optimization phases can recognize this as an example of common expressions that need not be duplicated. While in this simple case it would be possible to create a code generator that never generated the redundant expression, taking into account every such optimization at code generation time is very difficult. We get better code and more reliable compilers by generating simple code first and then optimizing it.

PROGRAM LEVEL PERFORMANCE ANALYSIS

Because embedded systems must perform functions in real time we often need to know how fast a program runs. The techniques we use to analyze program execution time are also helpful in analyzing properties such as power consumption. In this section, we study how to analyze programs to estimate their run times. We also examine how to optimize programs to improve their execution times; of course, optimization relies on analysis. It is important to keep in mind that CPU performance is not judged in the same way as program performance. Certainly, CPU clock rate is a very unreliable metric for program performance. But more importantly, the fact that the CPU executes part of our program quickly does not mean that it will execute the entire program at the rate we desire. As illustrated in Figure 5.22, the CPU pipeline and cache act as windows into our program. In order to understand the total execution time of our program, we must look at execution paths, which in general are far longer than the pipeline and cache windows. The pipeline and cache influence execution time ,but execution time is a global property of the program. While we might hope that the execution time of programs could be precisely determined, this is in fact difficult to do in practice:



■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.

■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. In general, the execution time of an instruction in a pipeline depends not only on that instruction but on the instructions around it in the pipeline.

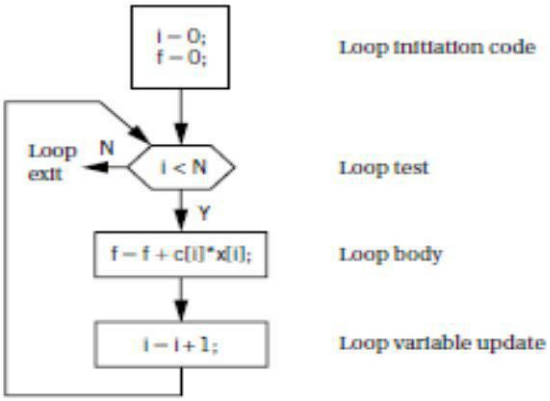
We can measure program performance in several ways:

■Some microprocessor manufacturers supply simulators for their CPUs: The simulator runs on a workstation or PC, takes as input an executable for the microprocessor along with input data, and simulates the execution of that program. Some of these simulators go beyond functional simulation to measure the execution time of the program. Simulation is clearly slower than executing the program on the actual microprocessor, but it also provides much greater visibility during execution. Be careful—some microprocessor performance simulators are not 100% accurate, and simulation of I/O-intensive code may be difficult.
■A timer connected to the microprocessor bus can be used to measure performance of executing sections of code. The code to be measured would reset and start the timer at its start and stop the timer at the end of execution. The length of the program that can be measured is limited by the accuracy of the timer.
■A logic analyzer can be connected to the microprocessor bus to measure the start and stop times of a code segment. This technique relies on the code being able to produce identifiable events on the bus to identify the start and stop of execution. The length of code that can be measured is limited by the size of the logic analyzers buffer. We are interested in the following three different types of performance measures on programs:
■**Average-case execution time** This is the typical execution time we would expect for typical data. Clearly, the first challenge is defining typical inputs.
■**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. In some cases, the input set that causes the worst-case execution time is obvious, but in many cases it is not.
■**Best-case execution time** This measure can be important in Multirate real-time systems First, we look at the fundamentals of program performance in more detail. We then consider trace-driven performance based on executing the program and observing its behavior.

Elements of Program Performance:

The path is the sequence of instructions executed by the program (or its equivalent in the high-level language representation of the program). The instruction timing is determined based on the sequence of instructions traced by the program path, which takes into account data dependencies, pipeline behaviour, and caching. Luckily, these two problems can be solved relatively independently.

Although we can trace the execution path of a program through its high-level language specification, it is hard to get accurate estimates of total execution time from a high-level language program. The number of memory locations and variables must be estimated, and results may be either saved for reuse or recomputed on the fly, among other effects. These problems become more challenging as the compiler puts more and more effort into optimizing the program. However, some aspects of program performance can be estimated by looking directly at the C program. For example, if a program contains a loop with a large, fixed iteration bound or if one branch of a conditional is much longer than another, we can get at least a rough idea that these are more time-consuming.



Segments of the program. Of course ,a precise estimate of performance also relies on the instructions to be executed, since different instructions take different amounts of time. (In addition, to make life even more difficult, the execution time of one instruction can depend on the instructions executed before and after it

To measure the longest path length ,we must find the longest path through the optimized CDFG since the compiler may change the structure of the control and data flow to optimize the program's implementation. It is important to keep in mind that choosing the longest path through a CDFG as measured by the number of nodes or edges touched may not correspond to the longest execution time. Since the execution time of a node in the CDFG will vary greatly depending on the instructions represented by that node, we must keep in mind that the longest path through the CDFG depends on the execution times of the nodes. In general, it is good policy to choose several of what we estimate are the longest paths through the program and measure the lengths of all of them in sufficient detail to be sure

that we have in fact captured the longest path. Once we know the execution path of the program, we have to measure the execution time of the instructions executed along that path. The simplest estimate is to assume that every instruction takes the same number of clock cycles, whichmeans we need only count the instructions and multiply by the per-instruction execution time to obtain n the program's total execution time. However, even ignoring cache effects, this technique is simplistic for the reasons summarized below.

■Not all instructions take the same amount of time. Although RISC architectures tend to provide uniform instruction execution times in order to keep the CPU's pipeline full, even many RISC architectures take different amounts of time to execute certain instructions. Multiple load-store instructions are examples of longer-executing instructions in the ARM architecture. Floating point
instructions show especially wide variations in execution time—while basic multiply and add operations are fast, some transcendental functions can take thousands of cycles to execute.

■Execution times of instructions are not independent. The execution time of one instruction depends on the instructions around it. For example ,many CPUs use register bypassing to speed up instruction sequences when the result of one instruction is used in the next instruction. As a result, the execution
time of an instruction may depend on whether its destination register is used as a source for the next operation (or vice versa).

■The execution time of an instruction may depend on operand values. This is clearly true of floating-point instructions in which a different number of iterations may be required to calculate the result. Other specialized instructions can, for example, perform a data-dependent number of integer operations.

Measurement-Driven Performance Analysis:

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. We refer to the record of the execution path of a program as a **program trace** (or more succinctly ,a **trace**) . Traces can be valuable for other purposes, such as analyzing the cache behavior of the program. Perhaps the biggest problem in measuring program performance is figuring out a useful set of inputs to provide to the program. This problem has two aspects. First, we have to determine the actual input values. We may be able to use benchmark data sets or data captured from a running system to help us generate typical values. For simple programs, we may be able to analyze the algorithm to determine the inputs that cause the worst-case execution time.
The other problem with input data is the **software scaffolding** that we may need to feed data into the program and get data out. When we are designing a large system,

it may be difficult to extract out part of the software and test it independently of the other parts of the system. We may need to add new testing modules to the system software to help us introduce testing values and to observe testing outputs.

We can measure program performance either directly on the hardware or by using a simulator. Each method has its advantages and disadvantages. Physical measurement requires some sort of hardware instrumentation. The most direct method of measuring the performance of a program would be to watch the program counter's value: start a timer when the PC reaches the program's start, stop the timer when it reaches the program's end. Unfortunately, it generally isn't possible to directly observe the program counter. However, it is possible in many cases to modify the program so that it starts a timer at the beginning of execution and stops the timer at the end. While this doesn't give us direct information about the program trace, it does give us execution time. If we have several timers available, we can use them to measure the execution time of different parts of the program.

A logic analyzer or an oscilloscope can be used to watch for signals that mark various points in the execution of the program. However, because logic analyzers have a limited amount of memory, this approach doesn't work well for programs with extremely long execution times. Some CPUs have hardware facilities for automatically generating trace information. For example, the Pentium family microprocessors generate a special bus cycle, a branch trace message, that shows the source and/or destination address of a branch [Col97]. If we record only traces, we can reconstruct the instructions executed within the basic blocks while greatly reducing the amount of memory required to hold the trace. The alternative to physical measurement of execution time is simulation. A CPU simulator is a program that takes as input a memory image for a CPU and performs the operations on that memory image that the actual CPU would perform, leaving the results in the modified memory image.

For purposes 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 so that it can determine the exact number of clock cycles required for execution. A cycle-accurate simulator is built with detailed knowledge of how the processor works, so that it can take into account all the possible behaviours of the micro architecture that may affect execution time. Cycle-accurate simulators are slower than the processor itself, but a variety of techniques can be used to make them surprisingly fast, running only hundreds of times slower than the hardware itself. A cycle-accurate simulator has a complete model of the processor, including the cache. It can therefore provide valuable information about why the program runs too slowly. The next example discusses a simulator that can be used to model many different processors.

SOFTWARE PERFORMANCE OPTIMIZATION.

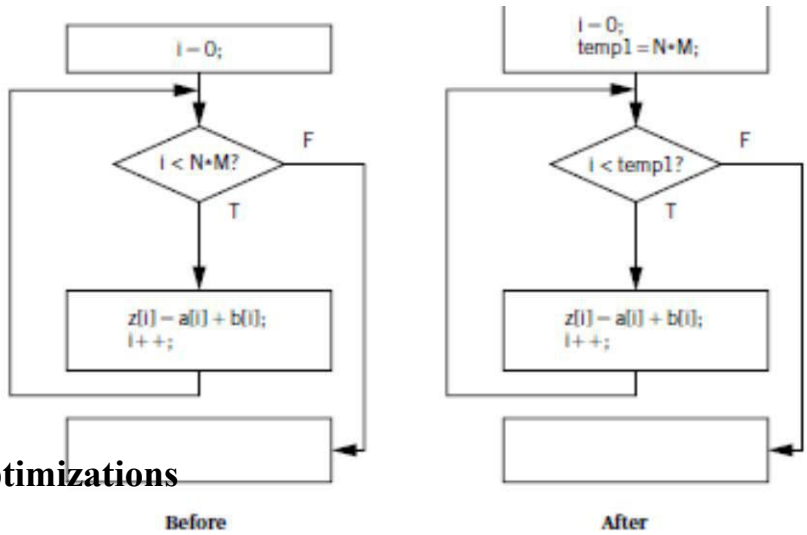
Loop Optimizations:

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,** and **strength reduction**. Code motion lets us move 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. Code motion opportunities can arise because programmers may find some computations clearer and more concise when put in the loop body, even though they are not strictly dependent on the loop iterations. A simple example of code motion is also common

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

The code motion opportunity becomes more obvious when we draw the loop's CDFG as shown in Figure 5.23.The loop bound computation is performed on every iteration during the loop test, even though the result never changes. We can avoid N *_M _*1 unnecessary executions of this statement by moving it before the loop, as shown in the figure. 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. Properly transformed, we may be able to eliminate some variables and apply strength reduction to others. A nested loop is a good example of the use of induction variables. Here is a simple nested loop

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



Before After

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 nest changes the order in which array elements are accessed. This can expose new parallelism opportunities that can be exploited by later stages of the compiler, and it can also improve cache performance

ANALYSIS AND OPTIMIZATION OF EXECUTION TIME, POWER, ENERGY, 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. Because inefficient programs often keep several copies of data, identifying and eliminating duplications can lead to significant memory savings usually with little performance penalty.

- Buffers should be sized carefully rather than defining a data array to a large size that the program will never attain, determine the actual maximum amount of data held in the buffer and allocate the array accordingly. 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. For instance, if several constants happen to have the same value, they can be mapped to the same location. Data buffers can often be reused at several different points in the program. This technique must be used with extreme caution, however, since subsequent versions of the program may not use the same values for the constants.

- A more generally applicable technique is to generate data on the fly rather than store it. Of course, the code required to generate the data takes up space in the program, but when complex data structures are involved there may be some net space savings from using code to generate data.
- 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. Because subroutines have overhead for parameter passing that is not obvious from the high-level language code, there is a minimum-size function body for which a subroutine makes sense.
- Architectures that have variable-size instruction lengths are particularly good candidates for careful coding to minimize program size, which may require assembly language coding of key program segments. There may also be cases in which one or a sequence of instructions is much smaller than

alternative implementations for example, a multiply-accumulate instruction may be both smaller and faster than separate arithmetic operations.

- 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.
- Even if the operations vary somewhat, you may be able to construct a properly parameterized subroutine that saves space. Of course, when considering the code size savings, the subroutine

PROGRAM VALIDATION AND TESTING.

Complex systems need testing to ensure that they work as they are intended. But bugs can be subtle, particularly in embedded systems, where specialized hardware and real-time responsiveness make programming more challenging. Fortunately, there are many available techniques for software testing that can help us generate a comprehensive set of tests to ensure that our system works properly.

. In this section, we concentrate on nuts-and-bolts techniques for creating a good set of tests for a given program. The first question we must ask ourselves is how much testing is enough. Clearly, we cannot test the program for every possible combination of inputs. Because we cannot implement an infinite number of tests, we naturally ask ourselves what a reasonable standard of thoroughness is. One of the major contributions of software testing is to provide us with standards of thoroughness that make sense. Following these standards does not guarantee that we will find all bugs. But by breaking the testing problem into sub problems and analyzing each sub problem.

we can identify testing methods that provide reasonable amounts of testing while keeping the testing time within reasonable bounds. The two major types of testing strategies:

■**Black-box** methods generate tests without looking at the internal structure of the program.

■**Clear-box** (also known as **white-box**) methods generate tests based on the program structure.

In this section we cover both types of tests, which complement each other by exercising programs in very different ways

Clear-Box Testing

The control/data flowgraph extracted from a program's source code is an important tool in developing clear-box tests for the program. To adequately test the program, we must exercise both its control and data operations. In order 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 may need to modify the program to make it more testable. By adding new inputs and outputs, we can usually substantially reduce the effort required to find and execute the test. Example 5.11 illustrates the importance of observability

and controllability in software testing. No matter what we are testing,we must accomplish the following three things in a test:

■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..

Being able to perform this process for a large number of tests entails some amount of drudgery, but that drudgery can be alleviated with good program design that simplifies controllability and observability. 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.

Black-Box Testing

Black-box tests are generated without knowledge of the code being tested. When used alone black-box tests have a low probability of finding all the bugs in a program. But when used in conjunction with clear-box tests they help provide a well-rounded test set, since black-box tests are likely to uncover errors that are unlikely to be found by tests extracted from the code structure. Black-box tests can really work. For instance, when asked to test an instrument whose front panel was run by a microcontroller, one acquaintance of the author used his hand to depress all the buttons simultaneously. The front panel immediately locked up. This situation could occur in practice if the instrument were placed face-down on a table, but discovery of this bug would be very unlikely via clear-box tests. One important technique is to take tests directly from the specification for the code under design.

The specification should state which outputs are expected for certain inputs. Tests should be created that provide specified outputs and evaluate whether the results also satisfy the inputs. We can't test every possible input combination, but some rules of thumb help us select reasonable sets of inputs. When an input can range across a set of values, it is a very good idea to test at the ends of the range. For example, if an input must be between 1 and 10, 0, 1, 10, and 11 are all important values to test. We should be sure to consider tests both within and outside the range, such as, testing values within the range and outside the range. We may want to consider tests well outside the valid range as well as boundary-condition tests.

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 test inputs are applied. A large number of tests must be applied for the results to be statistically significant, but the tests are easy to generate. Another scenario is to test certain types of data values. For example, integer valued inputs can be generated at interesting values such as 0,1,and values near the maximum end of the data range. Illegal values can be tested as well.

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 should be saved to apply to the later versions of the system.

Clearly, unless the system specification changed, the new system should be able to pass old tests. In some cases old bugs can creep back into systems, such as when an old version of a software module is inadvertently installed. In other cases regression tests simply exercise the code in different ways than would be done for the current version of the code and therefore possibly exercise different bugs. Some embedded systems, particularly digital signal processing systems, lend themselves to numerical analysis. Signal processing algorithms are frequently implemented with limited-range arithmetic to save hardware costs. Aggressive data sets can be generated to stress the numerical accuracy of the system. These tests can often be generated from the original formulas without reference to the source code

Evaluating Function Tests

How much testing is enough? Horgan and Mathur [Hor96] evaluated the coverage of two well- known programs, TeX and awk. They used functional tests for these programs that had been developed over several years of extensive testing. Upon applying those functional tests to the programs, they obtained the code coverage statistics shown in Figure 5.30.The columns refer to various types of test coverage: block refers to basic blocks, decision to conditionals, p-use to a use of a variable in a predicate (decision), and c-use to variable use in a non predicate computation. These results are at least suggestive that functional testing does not fully exercise the code and that techniques that explicitly generate tests for various pieces of code are necessary to obtain adequate levels of code coverage. Methodological techniques are important for understanding the quality of your tests. For example, if you keep track of the number of bugs tested each day, the data you collect over time should show you some trends on the number of errors per page of code to expect on the average, how many bugs are caught by certain kinds of tests, and so on. We address methodological approaches to quality control in more detail. One interesting method for analyzing the coverage of your tests is **error injection**.

First, take your existing code and add bugs to it, keeping track of where the bugs were added. Then run your existing tests on the modified program. By counting the number of added bugs your tests found, you can get an idea of how effective.

	Block	Decision	P-use	C-use
TeX	85%	72%	53%	48%
awk	70%	59%	48%	55%

the tests are in uncovering the bugs you haven't yet found. This method assumes that you can deliberately inject bugs that are of similar varieties to those created naturally by programming errors.

If the bugs are too easy or too difficult to find or simply require different types of tests, then bug injection's results will not be relevant. Of course, it is essential that you finally use the correct code, not the code with added bugs.

Platform level performance Analysis

Bus based systems add another layer of complication to performance analysis platform level performance involve much more than the CPU we often focus on the CPU because it processes instructions but any part of the system can affect total system performance. More precisely the CPU provides an upper bound on performance but any other part of the system can slow down the CPU merely counting instruction execution times is not enough

Consider the simple system we want to move data from memory to the CPU to process it to get the data from memory to the CPU we must

- Read from the memory
- Transfer over the bus to the cache
- Transfer from the cache to the CPU

The time required to transfer from the cache to the CPU is included in the instruction execution time , but the other two times are not .

Bandwidth as performance

The most basic measure of performance we are interested in is bandwidth the rate at which we can move data ultimately if we are interested in real time performance we are interested in real time performance measured in seconds but often the simplest way to measure performance is in units of clock cycles however different parts of the system will run at different clock rates. We have to make sure that we apply the right clock rate to each part of the performance estimate when we convert from clock cycles to seconds

Bus bandwidth

Bandwidth questions often come up when we are transferring large blocks of data for simplicity let's start by considering the bandwidth provided by only one system component the bus consider and image of 320 pixels with each pixel composed of 3 bytes of data this gives a grand total of 230 400 bytes of data if these images are video frames, we want to check if we can push one frame through the system within the 1/30 sec that we have to process a frame before the next one arrives.

Let us assume that we can transfer one byte of data every microsecond which implies a bus speed of 1 Mhz. in this case we would require 230400 us =0.23 sec to transfer one frame that is more than the 0.033 sec allotted to the data transfer we would have to increase the transfer rate by 7xto satisfy our performance requirement

We can increase bandwidth in two ways we can increase the clock rate of the bus or we can increase the amount of data transferred per clock cycle for example if we increased the bus to carry four bytes or 32 bits per transfer we would reduce the transfer time to 0.058 sec if we could also increase the bus clock rate to 2 Mhz. then we would reduce the transfer time to 0.029sec , which is within our time budget for the transfer

Part – A

1. What is the bus protocols especially, the four-cycle handshake? April 2014

Protocols are the set of rules and conditions for the data communication. The basic building block of most bus protocols is the four-cycle handshake.

Handshake ensures that when two devices want to communicate. One is ready to transmit and other is ready to receive.

The handshake uses a pair of wires dedicated to the handshake; such as enq(meaning enquiry) and ack (meaning acknowledge). Extra wires are used for the data transmitted during handshake.

2. What is a data flow graph? [CO2-L1-April 2014]

A data flow graph is a model of a program with no conditions. In a high level programming language, a code segment with no conditions and one entry point and exit point.

3. What are CPU buses? [CO2-L1-Nov/Dec 2013 & May/June 2013]

Data bus
Address bus
Control bus
System bus.

4. List out the various compilation techniques. [Nov/Dec 2013]

There are three types of compilation techniques:

Analysis and optimization of execution time.
Power energy and program size
Program validation and testing.

5. State the basic principles of basic compilation techniques. [May/June 2013]

Compilation combines translation and optimization.
The high level language program is translated in to lower level form of instructions; optimizations try to generate better instruction sequences.

Compilation = Translation + optimization

6. Name any two techniques used to optimize execution time of program. [Nov/Dec 2012]

Instruction level optimization
Machine independent optimization.

7. What does a linker do? [Nov/Dec 2012]

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 assemble code to make the necessary links between files.

8. What are the four types of data transfer in USB? [May/June 2012]

Control transfer
Interrupt transfer
Bulk transfer
Isochronous transfer (sequence of data)

9. Give the limitation of polling techniques. [May/June 2012]

It is wasteful of the processors time, as it needlessly checks the status of all devices all the time.

It is inherently slow, as it checks the status of all input/output devices before it comes back to check any given one again.

Priority of the device cannot be determined frequently.

10. Define BUS.

A bus is a connection of wires. The bus defines a protocol by which the CPU communicates with memory and I/O devices.

11. Define memory mapped I/O?

Memory-Mapped I/O (MMIO) and Port-Mapped I/O (PMIO) (which is also called isolated i/o) are two complementary methods of performing input/output between the cpu and peripheral devices in a computer. Memory-mapped I/O, uses the same address bus to address both memory and i/o devices – the memory and registers of the i/o devices are mapped to (associated with) address values. So when an address is accessed by the cpu, it may refer to a portion of physical ram, but it can also refer to memory of the i/o device.

12. Define RAM?

RAM is an acronym for random access memory, a type of computer memory that can be accessed randomly; that is, any byte of memory can be accessed without touching the preceding bytes. RAM is the most common type of memory found in computers and other devices, such as printers.

13. What is dynamic RAM?

Dynamic Random-Access Memory (DRAM) is a type of random-access memory that stores each bit of data in a separate capacitor within an integrated circuit. The capacitor can be either charged or discharged; these two states are taken to represent the two values of a bit, conventionally called 0 and 1.

14. What is ROM?

Read-Only memory (ROM) is a class of storage medium used in computers and other electronic devices. Data stored in ROM can only be modified slowly, with difficulty, or not at all, so it is mainly used to distribute firmware (software that is very closely tied to specific hardware, and unlikely to need frequent updates).

15. What are the 4 types of data transfer used in USB? [CO2-L1]

Control Transfer, Isochronous Transfer, Interrupt Transfer, Bulk Transfer

19.

Give the limitations of polling technique.

1) it is wasteful of the processors time, as it needlessly checks the status of all devices all the time,2) it is inherently slow, as it checks the status of all I/O devices before it comes back to check any given one again,3) when fast devices are connected to a system, polling may simply not be fast enough to satisfy the minimum service

requirements, 4) priority of the device is determined by the order in the polling loop, but it is possible to change it via software.

16. What is USB? Where is it used? [CO2-L1]

It is an external bus standard that supports data transfer rates of 12 Mbps. A single USB port can be used to connect up to 127 peripheral devices, such as mice, modems, and keyboards.