SNS COLLEGE OF ENGINEERING
(Autonomous)
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT III

EMBEDDED PROGRAMMING

COMPILATION TECHNIQUES AND

PROGRAM LEVEL PERFORMANCE ANALYSIS

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.

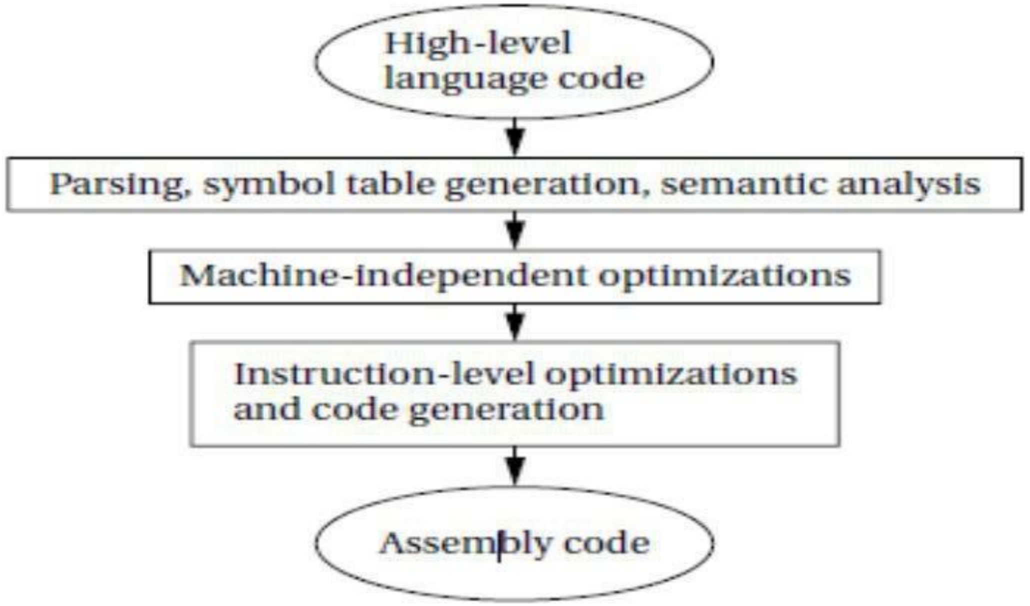
High-level language code
Parsing, symbol table generation, semantic analysis
Machine-independent optimizations
Instruction-level optimizations and code generation
Assembly code

Fig2.19The 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:

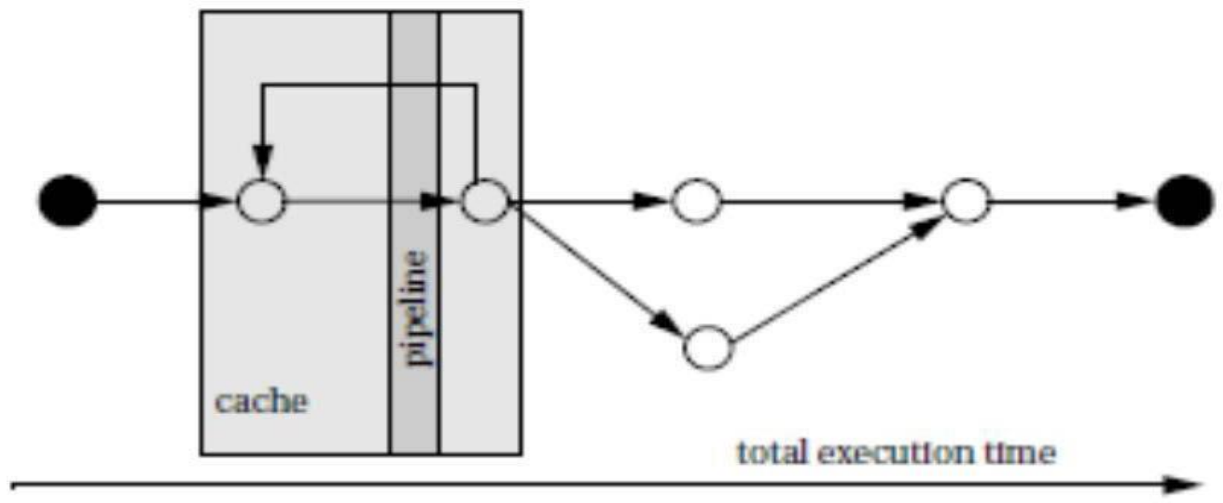

Fig 5.22 PROGRAM LEVEL PERFORMANCE ANALYSIS

- ❖ 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 instructionsaround 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.

Thank you