Atividades 3 e 4

Fluxo de projeto embarcados

Rafael Corsi
rafael.corsi@insper.edu.br

8 de fevereiro de 2017

Entregar na 3ª aula em formato PDF via github.

1. Criar softwares para microcontroladores utilizando suas especificidades (periféricos/ low power);
2. Avaliar e melhorar soluções embarcadas integrando hardware/software levando em conta adequação a uma aplicação;
3. Integrar em um protótipo hardware, software básico, sistema operacional de tempo real e módulos de interfaceamento com usuários, de comunicação e de alimentação;
4. Compreender as limitações de microcontroladores e seus periféricos;
5. Buscar e analisar documentação (datasheet) e extrair informações

1 Cross-compiler

- O que é cross compilação (cross-compiler) ?

2 Embarcados

(Mínimo: dois itens)

- O que é um RTOS, descreva uma utilizações.

- O que é desenvolvimento de projetos em V (Modelo V)?
- O que é um DSP ? O que difere de um microcontrolador ?

3 C

(Mínimo: um item)

1. Descreva a funcionalidade do:
   - Compilador C
   - Assembler
   - Linker
2. Qual a diferença entre C e C++ ?

4 Paralelismo vs Concorrência

Analise o texto a seguir extraído do livro : "Introduction to Embedded Systems - A Cyber-Physical Systems Approach (7.2.1)" e faça uma resenha sobre paralelismo e concorrência .

7.2.1 Parallelism vs. Concurrency

Concurrency is central to embedded systems. A computer program is said to be concurrent if different parts of the program *conceptually* execute simultaneously. A program is said to be **parallel** if different parts of the program *physically* execute simultaneously on distinct hardware (such as on multicore machines, servers in a server farm, or distinct microprocessors).

Non-concurrent programs specify a *sequence* of instructions to execute. A programming language that expresses a computation as a sequence of operations is called an **imperative** language. C is an imperative language. When using C to write concurrent programs, we must step outside the language itself, typically using a **thread library**. A thread library uses facilities provided not by C, but rather provided by the operating system and/or the hardware. Java is a mostly imperative language extended with constructs that directly support threads. Thus, one can write concurrent programs in Java without stepping outside the language.

Every (correct) execution of a program in an imperative language must behave as if the instructions were executed exactly in the specified sequence. It is often possible, however, to execute instructions in parallel or in an order different from that specified by the program and still get behavior that matches what would have happened had they been executed in sequence.

Example 7.5: Consider the following C statements:

```
double pi, piSquared, piCubed;
pi = 3.14159;
piSquared = pi * pi ;
piCubed = pi * pi * pi;
```

The last two assignment statements are independent, and hence can be executed in parallel or in reverse order without changing the behavior of the program. Had we written them as follows, however, they would no longer be independent:

```
double pi, piSquared, piCubed;
pi = 3.14159;
```

```
    piSquared = pi * pi ;
    piCubed = piSquared * pi;
```

In this case, the last statement depends on the third statement in the sense that the third statement must complete execution before the last statement starts.

A compiler may analyze the dependencies between operations in a program and produce parallel code, if the target machine supports it. This analysis is called **dataflow analysis**. Many microprocessors today support parallel execution, using multi-issue instruction streams or VLIW (very large instruction word) architectures. Processors with multi-issue instruction streams can execute independent instructions simultaneously. The hardware analyzes instructions on-the-fly for dependencies, and when there is no dependency, executes more than one instruction at a time. In the latter, VLIW machines have assembly-level instructions that specify multiple operations to be performed together. In this case, the compiler is usually required to produce the appropriate parallel instructions. In these cases, the dependency analysis is done at the level of assembly language or at the level of individual operations, not at the level of lines of C. A line of C may specify multiple operations, or even complex operations like procedure calls. In both cases (multi-issue and VLIW), an imperative program is analyzed for concurrency in order to enable parallel execution. The overall objective is to speed up execution of the program. The goal is improved **performance**, where the presumption is that finishing a task earlier is always better than finishing it later.

In the context of embedded systems, however, concurrency plays a part that is much more central than merely improving performance. Embedded programs interact with physical processes, and in the physical world, many activities progress at the same time. An embedded program often needs to monitor and react to multiple concurrent sources of stimulus, and simultaneously control multiple output devices that affect the physical world. Embedded programs are almost always concurrent programs, and concurrency is an intrinsic part of the logic of the programs. It is not just a way to get improved performance. Indeed, finishing a task earlier is not necessarily better than finishing it later. *Timeliness* matters, of course; actions performed in the physical world often need to be done at the *right time* (neither early nor late). Picture for example an engine controller for a gasoline engine. Firing the spark plugs earlier is most certainly not better than firing them later. They must be fired at the *right* time.

Just as imperative programs can be executed sequentially or in parallel, concurrent programs can be executed sequentially or in parallel. Sequential execution of a concurrent program is done typically today by a **multitasking operating system**, which interleaves the execution of multiple tasks in a single sequential stream of instructions. Of course, the hardware may parallelize that execution if the processor has a multi-issue or VLIW architecture. Hence, a concurrent program may be converted to a sequential stream by an operating system and back to concurrent program by the hardware, where the latter translation is done to improve performance. These multiple translations greatly complicate the problem of ensuring that things occur at the *right* time. This problem is addressed in Chapter 11.

Parallelism in the hardware, the main subject of this chapter, exists to improve performance for computation-intensive applications. From the programmer's perspective, concurrency arises as a consequence of the hardware designed to improve performance, not as a consequence of the application problem being solved. In other words, the application does not (necessarily) demand that multiple activities proceed simultaneously, it just demands that things be done very quickly. Of course, many interesting applications will combine both forms of concurrency, arising from parallelism and from application requirements.

The sorts of algorithms found in compute-intensive embedded programs has a profound affect on the design of the hardware. In this section, we focus on hardware approaches that deliver parallelism, namely pipelining, instruction-level parallelism, and multicore architectures. All have a strong influence on the programming models for embedded software. In Chapter 8, we give an overview of memory systems, which strongly influence how parallelism is handled.

7.2.2 Pipelining

Most modern processors are **pipelined**. A simple five-stage pipeline for a 32-bit machine is shown in Figure 7.2. In the figure, the shaded rectangles are latches, which are clocked at processor clock rate. On each edge of the clock, the value at the input is stored in the latch register. The output is then held constant until the next edge of the clock, allowing the circuits between the latches to settle. This diagram can be viewed as a synchronous-reactive model of the behavior of the processor.

5 Avaliação

Insatisfatório (I) :	• não apresentou o estudo • contém apenas 40% do total pedido • contém plágio (textos copiados sem referências)
Em Desenvolvimento (D) :	• apresentou o estudo com até 2 aulas de atraso • contém apenas 70% tópicos do exigido • texto incoerente
Essencial (C)	• apresentou o estudo com até 1 dia de atraso • entregou mais de 70% da atividade com respostas corretas • texto coerente
Proficiente (B)	• apresentou o estudo sem atraso • entregou mais de 70% da atividade com respostas corretas
Avançado (A)	• apresentou o estudo sem atraso • entregou mais de 90% da atividade com respostas corretas