1

INTRODUCTION

System-level design usually involves designing an application specified at a large granularity. A typical design objective is to minimize cost (in terms of area or power) while the performance constraints are usually throughput or latency requirements. The basic components of a system-level specification are called *tasks* (or nodes). The task-level description of a particular application (modem) is shown in Figure 1.1. The specification has two characteristics. First, tasks are at a

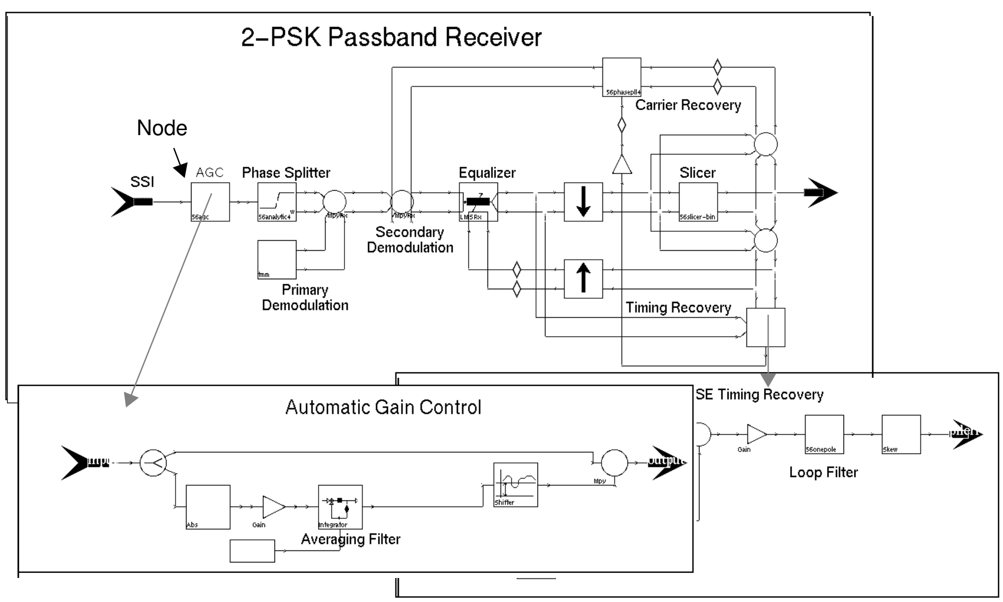


Figure 1.1. Task-level description of the receiver section of a modem.

higher level of abstraction than atomic operations or instructions. This allows for complex applications to be described easily and more naturally. Secondly, there is no commitment to how the system is implemented. Since the specification does not assume a particular architecture, it is possible to generate either a hardware, or a software, or a mixed implementation. This is especially important for the synthesis of complex applications whose cost and performance constraints often demand a mixed hardware-software implementation.

System-level design is very broad problem. In this thesis, we restrict our attention to the design of **embedded systems with real-time signal processing components**. Examples of such systems include modems for both tethered and wireless communication, cordless phones, disk drive controllers, printers, digital audio systems, data compression systems, etc. These systems are characterized by stringent performance and cost constraints.

Such applications often tend to have mixed hardware and software implementations. For instance, full-software implementations (program running on a programmable processor) often cannot meet the performance requirements, while custom-hardware solutions (custom ASIC) may increase design and product costs. It is important, therefore, not to commit each node in the application to a particular mapping (hardware or software) or implementation (design style within hardware or software) when specifying the application. The appropriate mapping and implementation for each node can be selected by a global optimization procedure after the specification stage. The task level of abstraction allows this flexibility. Also, such embedded systems typically have a product cycle time of 18 months or so [Keutzer94] which imposes a severe time-to-market constraint. Manual development of a lower-level design specification (such as at the RTL level) is quite intensive due to the sheer complexity of the applications. As a result, it is desirable to specify the application at the task level and allow a design tool to generate the lower levels of implementation from it.

Such a system-level design approach is now viable due to the maturity of lower-level design tools and semiconductor technology. Computer-aided design tools that operate at lower levels of abstraction (as shown later in Figure 1.5) are quite robust. The next step is to use these CAD tools for system-level design. Also, advances in semiconductor manufacturing have made it possible to fabricate a "system on a chip". For instance, core-based ASICs that contain cores of programmable processors along with custom hardware on the same die are now a commercial reality.

In the next section, we discuss a few key issues in system-level design. Our approach to solving these problems is outlined in Section 1.2.

1.1 Issues in System-level Design

Figure 1.2 summarizes the key issues in system-level design. These are *partitioning*, *synthesis*, *simulation,* and *design-space exploration*. In this thesis, we address these issues in order. Several hardware and software implementation options are usually available for each node in the task-level description. The *partitioning* process determines an appropriate mapping (hardware or software) and an implementation for each node. A partitioned application has to be *synthesized* and *simulated* within a unified framework that involves the hardware and software components as well as the generated interfaces. The system-level design space is quite large. Typically, the designer needs to explore the possible options, tools, and architectures, choosing either automated tools or manually selecting his/her choices. The *design-space exploration* framework attempts to ease this process.

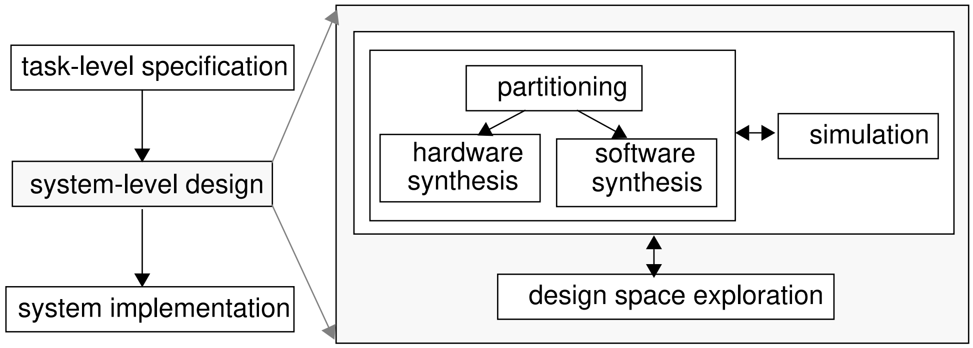


Figure 1.2. System-level design

We next discuss each of these issues in more detail.

1.1.1 Partitioning

Perhaps the most important aspect of system-level design is the multiplicity of design options available for every node in the task-level specification. Each node can be implemented in several ways in both hardware and software mappings. The *partitioning* problem is to select an appropriate combination of mapping and implementation for each node.

For instance, a given task can be implemented in hardware using design options at several levels.

1. *Algorithm level*: Several algorithms can be used to describe the same task. For instance, a finite impulse response filter can be implemented either as an inner product or using the FFT in a shift-and-add algorithm. As a trivial example, a biquad can be described using the direct form or the transpose form (Figure 1.3).

2. *Transformation level*: For a particular algorithm, several transformations [Potkonjak94] can be applied on the original task description. Figure 1.3-b shows two such transformations. In the first transformation, multiplication by a constant (filter coefficients) is replaced by equivalent shift and add

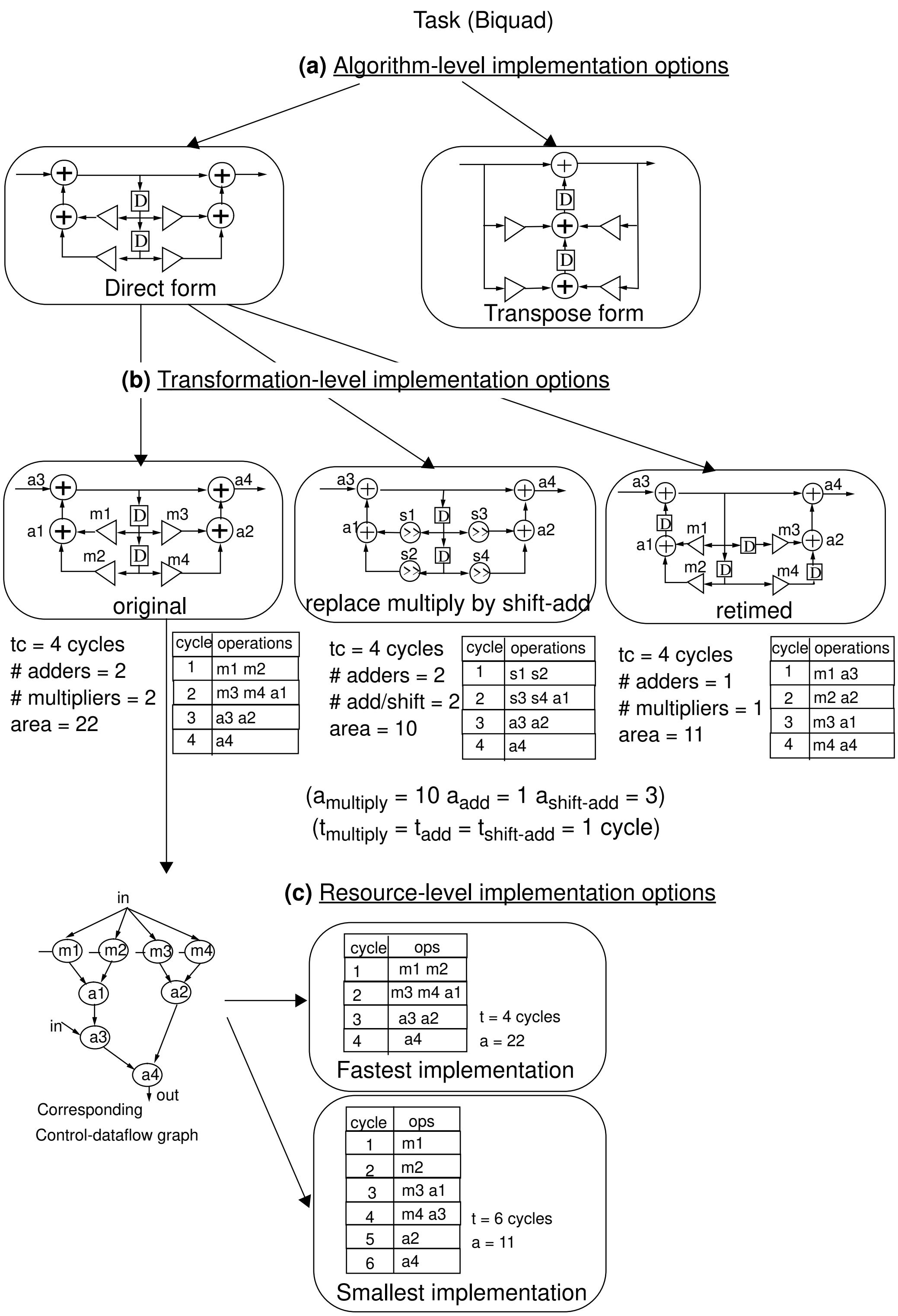


Figure 1.3. Hardware design options for a “node” at the algorithmic, transformational, and resource levels.

operations. Since the area of a shift-add operator is smaller than that of a multiplier, the transformed design has a smaller area for the same execution time. The second transformation shows the retimed design. By changing relative positions of delay elements, the operations can be distributed over the various clock cycles to get a better resource utilization and consequently a smaller area for the same execution time.

3. *Resource level*: A task, for a specified algorithm and transformation set, can be implemented using varying numbers of resource units. Typical resource units are adders and multipliers. Varying the resource units changes the values of the implementation metrics (area and execution time). Figure 1.3-c shows two resource-level implementation options for the direct form biquad. It shows the schedule and area for the fastest (critical path of 4 cycles) and smallest (only one resource of each kind) designs.

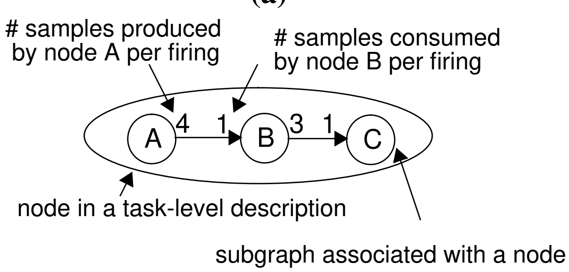
Similarly, different software synthesis strategies can be used to implement a given node in software. For instance, inlined code is faster than code using subroutine calls, but has a larger code size. Thus, there is a trade-off between code size and execution time. Figure 1.4-a shows a 3 node subgraph associated with a node. The schedule describes the sequence in which nodes have to be executed such that at the end of one pass through the graph, the number of samples on all the arcs is restored to the number at start (i.e., there is no accumulation of data). A possible schedule, along with the corresponding generated code, is shown in Figure 1.4-b. Figure 1.4-c shows three possible software implementations differing in size and execution time.

Thus each node in the task-level description can be implemented in several ways in either hardware or software. Current design tools can generate multiple implementations for every task. In system-level design, there are a number of such tasks and the *overall* design is to be optimized. Clearly, it is not enough to optimize each task independently. For example, if each task in the task-level specification were fed to a highlevel hardware synthesis tool that is optimized for speed (i.e., generates the fastest implementation), then the overall area of the system might be too large. *Hardware-software partitioning* is the problem of determining an implementation for each node so that the overall design is optimized. There are two parts to this problem: (1) *Binary partitioning* is the problem of determining, for each node, a hardware or a software mapping and a schedule. (2) *Extended partitioning* is the problem of selecting an appropriate implementation, over and above binary partitioning.

1.1.2 System Synthesis

Once the appropriate implementation for each task has been determined,

(a)



(b)

a possible schedule	corresponding code
A(4BCCC)	see code below

```
code for node A;
for(j=1;j<=4; j++)
{
    code for node B;
    code for node C;
    code for node C;
    code for node C;
}
```

(c)

possible schedules	code size	data size	execution time
ABBBBCCCCCCCCCCCC	17c	16b	17t
A(4BCCC)	5c	7b	17t + 4y
A(4B(3C))	3c	7b	17t + 16y

Assumptions
code size of one node = c words
code size for loop constructs = 0
execution time of one node = t cycles
execution overhead for loop constructs = y cycles
buffer size per sample communicated = b words

Figure 1.4. Software design options for a node

the hardware-software *synthesis* problem is that of synthesizing the implementations. Implementations for nodes mapped to hardware or software can be generated by feeding the nodal descriptions to synthesis tools. Figure 1.5 shows both the hardware and the software design paths followed by existing synthesis tools. The hardware design path consists of highlevel hardware synthesis [McFarland90a], followed by logic synthesis [Brayton90], and layout synthesis [DeMicheli86]. The software design path comprises highlevel software synthesis [Pino95a], followed by compilation and assembly.

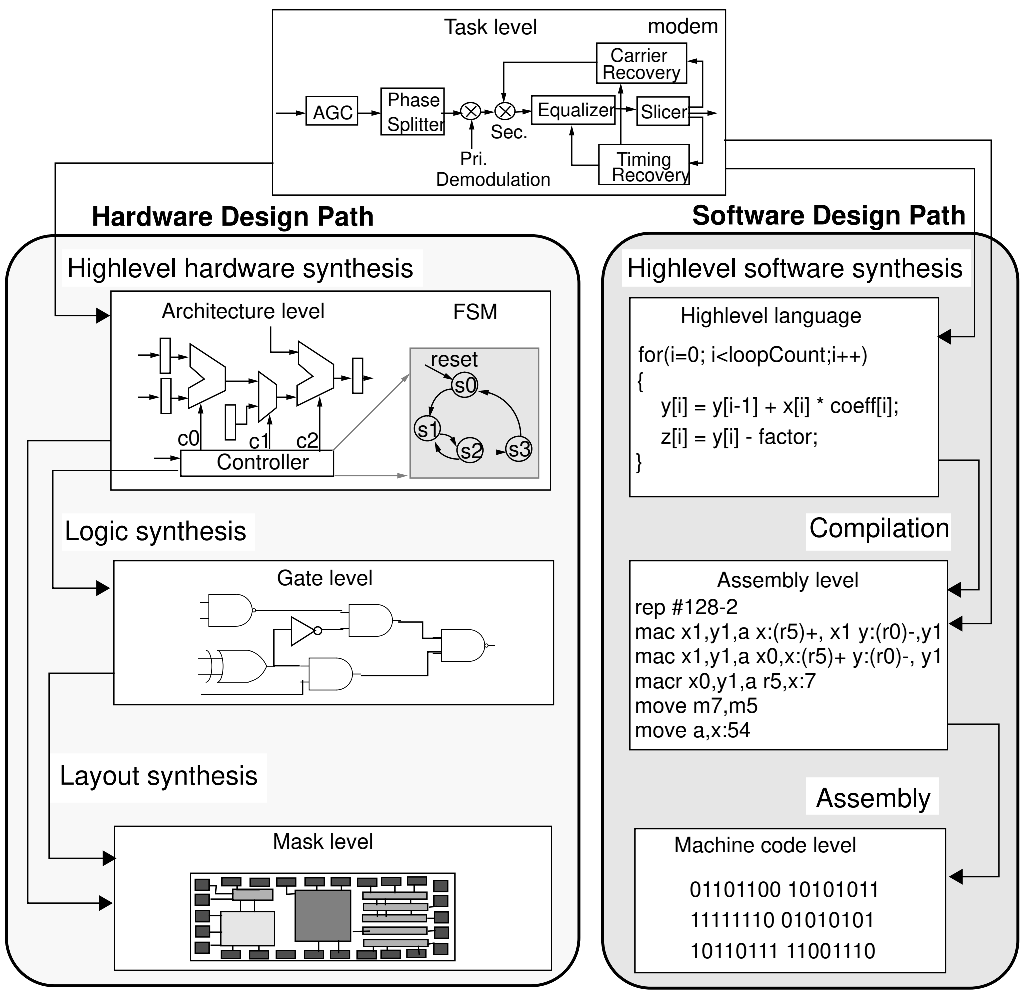


Figure 1.5. Design specification hierarchy and translation across levels of abstraction.

1.1.3 System Simulation

Simulation plays an important role in the system-level design process. At the specification level, it is possible that the design may be specified using a combination of one or more semantic models. Tools that allow the simulation of such heterogeneous specifications are required. At the implementation level, simulation tools that support mixed hardware-software systems are needed.

Throughout the design process, it should be possible to simulate systems where the components are described at different levels of abstraction. As parts of the design evolve from specification to their final implementation, functional models can be replaced by more lower-level structures. A simulation environment should be capable of supporting these aspects.

1.1.4 Design-Space Exploration

System-level design is not a black-box process, but relies considerably on user creativity. For instance, the user might want to experiment with the design parameters, the tools used, or the sequence in which these tools are applied. As such, there is no hardwired design methodology. The design process could get quite unwieldy as the user experiments with the design methodology. As a result, an infrastructure that supports *design-space exploration* is also a key aspect of the system-level design process.

Tools for design-space exploration fall under two categories: estimation and management. Estimation tools are primarily used for *what-if* analysis, i.e., they give quick predictions on the outcome of applying certain synthesis or transformation tools. Management tools orchestrate the design process, i.e., for systematic control of the design data, tools, and flow.

Design methodology management includes tools for visualizing the design process and managing the invocation of tools. Since tools involved in the system-level design process are often computationally intensive, it is important to avoid unnecessary invocation of tools. This requires that the design flow be specified in a modular way so that only desired tools may be invoked. Further, the designer should not have to manually keep track of the tools that have been run. Thus, in order to support efficient exploration of the design space, a mechanism to manage the design flow, tools, and data is required.

1.2 The Codesign Philosophy for System-level Design

Designing systems containing both hardware and software components is not a new problem. The traditional design approach has been somewhat *hardware-first* in that the software components are designed after the hardware has been designed and prototyped. This leaves little flexibility in evaluating different design options and hardware/software mappings. With isolated hardware and software design paths, it also becomes difficult to optimize the design as a whole. Such a design approach is especially inadequate when designing systems requiring strict performance and a small design cycle time. Our approach to designing such systems is to adopt the *codesign*[1] philosophy.

The key tenet in codesign is to avoid isolation between hardware and software designs. The strategy allows the hardware and software designs to proceed in parallel, with feedback and interaction between the two as the design progresses. This is accomplished by developing tools and methodologies that support the

1. One of the early definitions of codesign was given by Franke and Purvis [Franke91] as "The system design process that combines the hardware and software perspectives from the earliest stages to exploit design flexibility and efficient allocation of function". Wolf [Wolf94] emphasizes that "the hardware and software must be designed together to make sure that the implementation not only functions properly but also meets performance, cost, and reliability goals".

tightly coupled design of the hardware and software through a unified framework. The goal of codesigning the hardware and software components of a system is to achieve high-quality designs with a reduced design time.

This thesis presents a systematic approach to the system-level design of embedded signal processing systems, based on the codesign philosophy.