HARDWARE/SOFTWARE CO-DESIGN: APPLICATION DOMAINS AND DESIGN TECHNOLOGIES

GIOVANNI DE MICHELI
Stanford University
Stanford, CA 94305

1. Introduction

The design of electronic systems is complex because of the interplay of different components, which may be heterogeneous in nature. Moreover, systems consist of a very large number of elementary constituents (e.g., electronic transistors, machine-level instructions), which yield numerous design options. System-level design benefits from the use of *system-level computer-aided design* (CAD) tools to support modeling, validation and synthesis. Whereas computer-aided design tools have reached maturity in the domain of synthesizing and validating *very large scale integration* (VLSI) circuits, few tools and structured design methodologies exist yet for electronic systems, due to their heterogeneous nature, where different components may require different design techniques and where the interactions among heterogeneous components has to be dealt with.

It is often the case that most functions (e.g., computation, control) in a system are performed by its digital component. There is an ongoing trend in migrating analog-signal processing to digital-signal processing. While it is obvious that the electronic industry needs computer-aided design tools to support all facets of system design, we concentrate here on the digital parts and refer to them as (digital) systems for brevity.

The majority of digital systems are programmable, and thus consist of hardware and software components. The value of a system can be measured by some objectives that are specific to its application domain (e.g., performance, design and manufacturing cost, ease of programmability) and it depends on both the hardware and the software components. *Hardware/software co-design* means meeting system-level objectives by exploiting the synergism of hardware and software through their concurrent design. Since digital systems have different architectures and applications, there are

G. De Micheli and M. Sami (eds.), Hardware/Software Co-Design, 1–28.

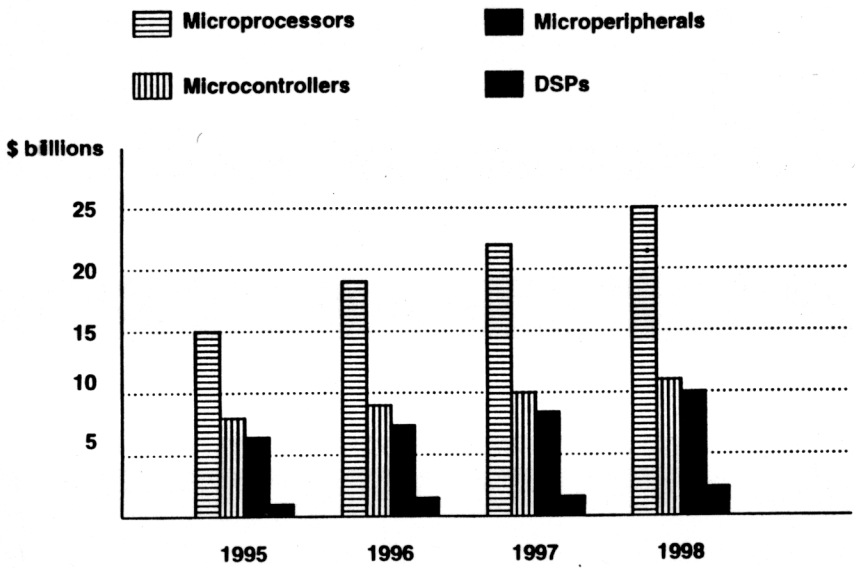

Figure 1. Forecast of the growth of electronic components. Source: Dataquest

several co-design problems of interest. Such problems have been tackled by skillful designers for many years, but detailed-level design performed by humans is often a time-consuming and error-prone task. Moreover, the large amount of information involved in co-design problems makes it unlikely that human designers can optimize all objectives, thus leading to products whose value is lower than the potential one.

The recent raise in interest in hardware/software co-design is due to the introduction of CAD tools for co-design (e.g., commercial simulators) and to the expectation that solutions to other co-design problems will be supported by tools, thus raising the potential quality and shortening the development time of electronic products. Due to the extreme competitiveness in the market-place, co-design tools are likely to play a key strategic role. This factor motivates researchers to dig into new scientific problems and/or to develop novel methods to solve classic problems (e.g., retargetable compilation) in this perspective. CAD tool providers are eager to explore the business opportunities in the growing co-design tool market. The forecast of the worldwide revenues of integrated circuits sales (Figure 1), and in particular for those used in dedicated applications (Figure 2), explains the high demand of electronic system-level design tools, whose volume of sales is expected to grow at a compound annual rate of 34% in the 1993-1998 time-frame, according to Dataquest.

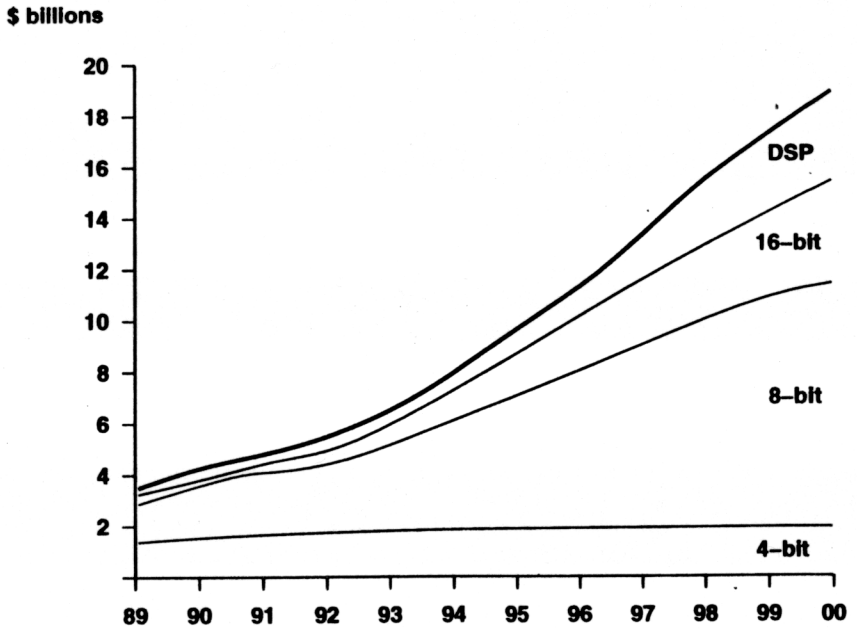

Figure 2. Worldwide revenues for the sales of microcontrollers and DSP integrated circuits. The emboldened line shows the total growth. Source: ICE

The evolution of integrated circuit technology is also motivating new approaches to digital circuit design. The trend toward smaller mask-level geometries leads to higher integration and higher cost of fabrication, hence to the need of amortizing hardware design over large production volumes. This suggests the idea of using software as a means of differentiating products based on the same hardware platform. Due to the complexity of hardware and software, their *re-use* is often key to commercial profitability. Thus, complex macro-cells implementing *instruction-set processors* (ISPs) and *digital-signal processors* (DSPs) are now made available and called processors *cores* (Figure 3 [19]). Standardizing on the use of cores or of specific processors means leveraging available software layers, ranging form operating systems to embedded software for user-oriented applications. Thus software re-use is also a driving force in the market. For example, the standardization of personal computers and workstations on few processor architectures has been an on-going trend in the recent years.

The recent introduction of *field-programmable gate array* (FPGA) technologies has blurred the distinction between hardware and software. Traditionally a hardware circuit used to be configured at manufacturing time.

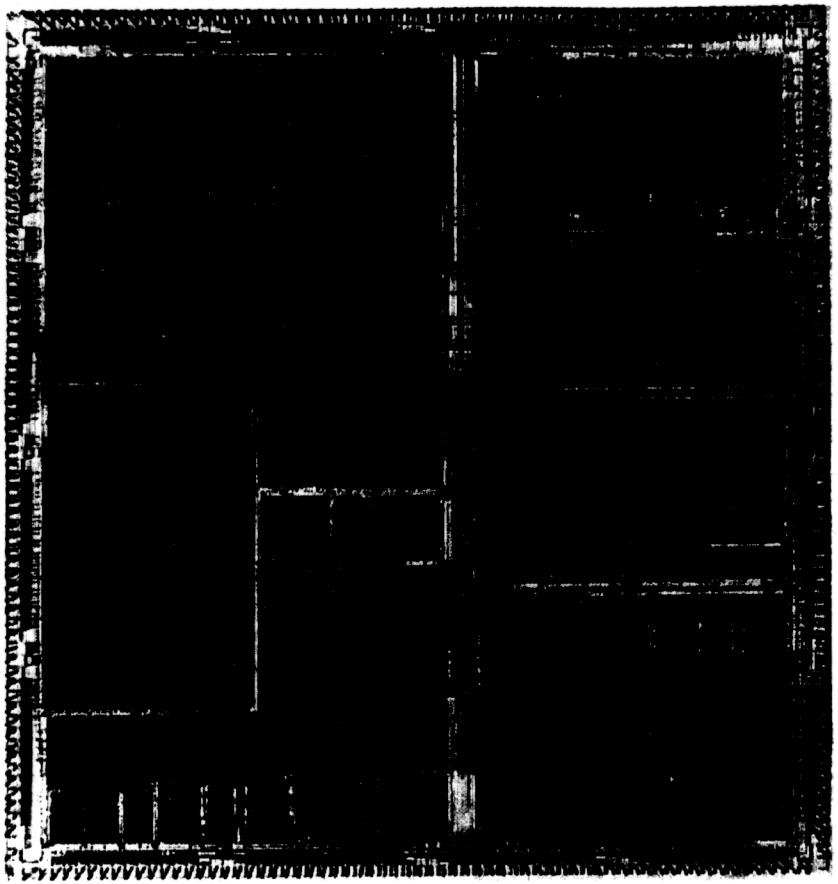

Figure 3. Example of an integrated circuit with programmable cores. The VCP chip has two processors: the VC core, which is based on the MIPS-X processor, is placed in the top right corner, while the VP+ DSP processor occupies the top left part of chip, above a memory array. Courtesy of Integrated Information Technology.

The functions of a hardware circuit could be chosen by the execution of a program. Whereas the program could be modified even at run-time, the structure of the hardware was invariant. With field-programmable technology it is possible to configure the gate-level interconnection of hardware circuits after manufacturing. This flexibility opens new applications of digital circuits, and new hardware/software co-design problems arise. For example, a hardware board can be programmed to implement a specific software function with better performances than a micro-processor.

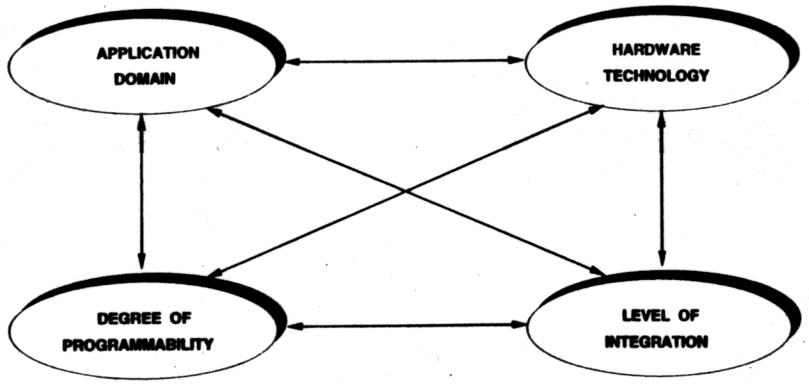

Figure 4. Distinguishing features of electronic systems

This chapter provides a general introduction to hardware/software co-design, and points the reader to specific co-design issues which are dealt with in the following chapters in more detail. Due to the broad nature of the topic, we describe first distinguishing features of electronic systems that are useful to classify co-design problems. We consider next system-level co-design issues for different kinds of systems and finally we review the state of the art of computer-aided co-design methods and tools.

2. Distinguishing features of electronic systems

We associate co-design problems with the classes of digital systems they arise from. We attempt to characterize these systems using general criteria (Figure 4). In particular, we use notions of *application domain*, degree of *programmability*, hardware *implementation technology*, and *level of integration* of the system. These factors are interrelated. For example, the level of integration depends on the implementation technology. In turn, the integration level may affect the degree of programmability of a system. Some applications require specific technologies and specific programmability features.

2.1. APPLICATION DOMAINS

When considering digital systems, we can distinguish among: *general-purpose* computing systems, *dedicated computing and control* systems, and *emulation and prototyping* systems. The first class includes traditional computers. Computers can be different in size, ranging from laptops to supercomputers, but they are characterized by the fact that the end-user can program

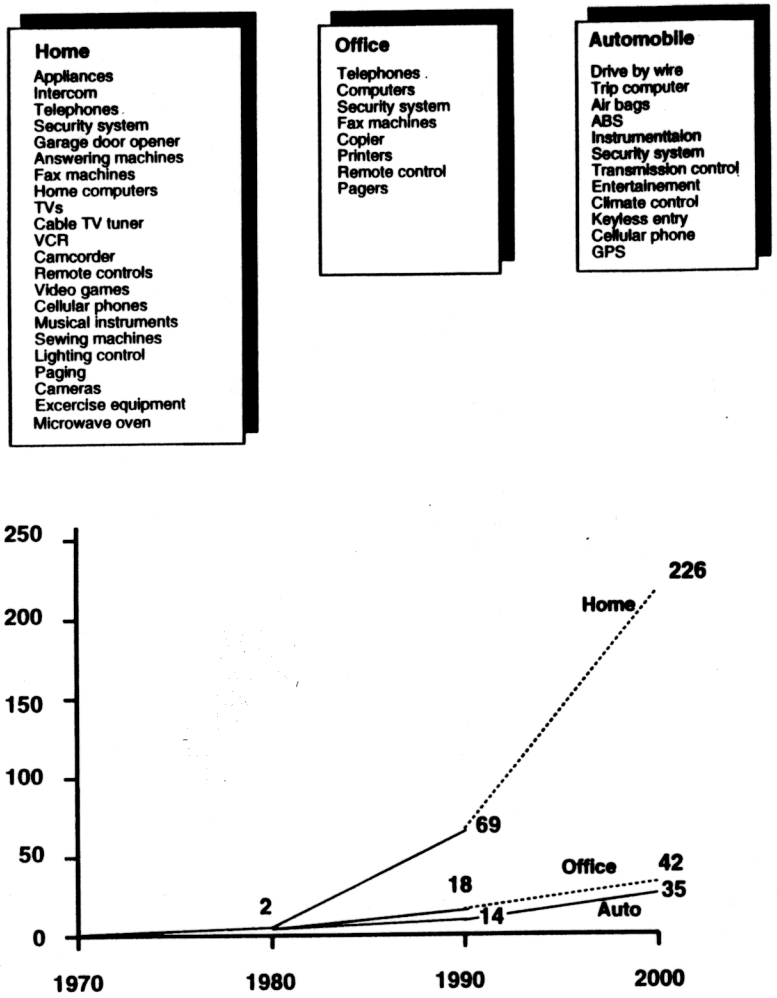

Figure 5. Examples of some applications of microcontrollers and their growth forecast. Source: Electronic News.

the system. Even though many users of computers just run specific applications (e.g., EXCEL) and do not even have a compiler or know its use, programming-language compilers are always available to the interested user. In other words, general purpose computers can support applications of different kinds, determined by the software developed or acquired by the user.

In contrast, dedicated computing (and/or) control systems are conceived with specific target applications. The user has limited access to programming the system, which is already provided by dedicated software programs. An example is the *fly by wire* control system of an aircraft. These systems are commonly called *embedded systems.* Embedded systems may be dedicated to control functions (e.g., embedded controllers) or to data communication and processing (e.g., telephone networks). In some cases, embedded systems perform both data processing and control, as in the case of radio-navigation systems coupled with steering devices. Embedded system design has been growing steadily over the last few years. Applications are ubiquitous in the manufacturing industry (e.g., plant and robot control), in consumer products (e.g., intelligent home devices), in vehicles (e.g., control and maintenance of cars, planes, ships), in telecommunication applications, and in systems for the defense of the territory and of the environment.

Emulation and prototyping systems make a class of their own. They are usually based on programmable hardware technology: hardware is configured using *hardware compilers* also called *synthesis systems* [15]. Specialized users program these systems. Prototypes are often use to validate a design concept. Thus they are used as intermediate points of the design and manufacturing flow of a product.

2.2. DEGREE OF PROGRAMMABILITY

There is a wide range in the degree of programmability of various digital systems. Systems can be programmed at the *application, instruction* or *hardware* levels. The most restrictive approach is the application level, where the system is running dedicated software programs that allow the user to specify desired functionality options using a specialized language. Examples range from programming a video-cassette recorder (VCR) to setting navigation instructions in an automated steering controller of a ship.

Most programmers of personal computers and workstations are familiar with writing, compiling and executing application programs, while a smaller group is used to writing and maintaining operating-system routines. In these cases, the digital system is programmed at the instruction level, because the end result of compilation is the generation of instruction sequences for the architecture being considered. The degree of programmability is defined by the processor architecture itself.

Hardware-level programming means configuring the hardware (after manufacturing) in a desired way. A simple and well-known example is microprogramming, i.e., determining the behavior of the control unit by a microprogram, which can be and stored in binary form in a memory. Emulation of other architectures can be achieved by altering the microprogram.

Today microprogramming is common for DSPs, but not for ISPs using RISC architectures [28].

Extreme examples of hardware-level programmability are given by systems where both the control unit and the data path can be reconfigured at run time using field-programmable connections available with FPGA technology. Thus, a specific hardware engine can be configured to run a particular application.

Overall, programmability increases the applicability of a digital system, but it does not increase its performances except on tailored applications. For general-purpose computing, top performance is achieved today by super-scalar RISC architectures [28], which are programmed at the instruction level. For dedicated applications, hard-wired (non-programmable) *application-specific integrated circuits* (ASICs) achieve often the best performance and the lowest power consumption. ASICs require a specific hardware design, and to bear its correspondent design and *non-recurrent engineering* (NRE) costs. Thus ASIC parts my be expensive if not produced in large volume. Moreover, such ASICs are not reusable for applications other than those for which they were conceived. Intermediate solutions are possible such as ASICs with programmable features and/or embedded cores. (See Section 2.4.)

In some application domains, such as data processing for telecommunications, it has been shown practical to replace standard processors by *application-specific instruction set processors* (ASIPs), which are instruction-level programmable processors with an architecture tuned to a specific application. The motivation for designing ASIPs stems from the fact that systems containing ASIPs are dedicated and thus run programs with a specific instruction mix. The instruction set and hardware structure are chosen to support efficiently such instruction mix, usually by having specific register sets and specific interconnections between the registers, busses and other hardware resources.

It is possible to envision ASIPs as intermediate solutions between ISPs and ASICs. ASIPs are more flexible than ASICs, but less than instruction-set processors. Nevertheless, they can be used for a family of applications in a specific domain. The performance of an ASIP on specific tasks can be higher than an instruction-set processor (because of the tuning of the architecture to the instruction mix) but it is usually lower than an ASIC. Opposite considerations apply to power consumption. The ASIP design-time and non-recurring engineering costs can be amortized over a larger volume than an ASIC, when the ASIP has multiple applications. Moreover, engineering changes and product updates can be handled graciously in ASIPs by reprogramming the software and avoiding hardware redesign. Unfortunately, being an ASIP a specific architecture, it requires a compiler development which adds to the non-recurring engineering costs. Such a compiler must also produce high-quality machine code to make the ASIP solution competitive.

Figure 6. Example of a chip with an ASIP core: the core and its microcode memory are visible in the top right corner. (STi1100 VideoPhone CODEC chip. Courtesy of SGS-Thomson Microelectronics.)

2.3. TECHNOLOGY

Digital systems rely on VLSI circuit technology. A system may have components with different scale of integration (e.g., discrete and integrated component) and different fabrication technologies (e.g., bipolar and CMOS). The choice of hardware technology for the system components affects the overall performance and cost, and therefore is of primary importance. We do not want to report here on the quantitative merit of different technologies. We

comment briefly instead on the qualitative impact on hardware/software co-design of technologies that allow field programmability of digital systems.

System-level field programmability can be achieved by storing programs in read/write memories and/or by exploiting programmable interconnections. In the former case the software component is programmed, while in the latter the hardware is configured. When referring to programmable interconnection technology, we need to distinguish between devices that can be programmed once only (such as anti-fuse based FPGAs [21]) and those that can be re-programmed (such as those where on-chip memory entries determine the connectivity [54]).

Re-programmable FPGAs support multiple hardware reconfigurations on the field, i.e., after manufacturing, and open the door to novel system-level solutions and to very interesting co-design problems (described in Section 3.) Therefore we consider in the sequel only re-programmable FPGAs, and when referring to programmable interconnection technology, we mean memory-based FPGAs [54].

Despite the growing importance of FPGA technology, it is important to remember that only a small fraction of digital systems exploit this feature today for a variety of reasons. First, performance of FPGAs is usually one order of magnitude less than in the corresponding non-programmable implementation in a technology with similar mask dimensions. These relegates programmable hardware parts to non-performance-critical portions of digital systems. Second, FPGA chips are more expensive as compared to dedicated hardware components in standard non-programmable technologies when produced in high volumes.

2.4. LEVEL OF INTEGRATION

Digital systems can be lumped or distributed, but we consider here only the former class. The level of integration is a key factor in the system cost. It is usually convenient to reduce the number of parts of a system, by integrating as many functions as possible on a single chip. Thus, chips have been manufactured that integrate digital processing, memory storage, analog functions and transducers.

When considering co-design problems, we can distinguish between systems consisting of components (like ASICs, ASIPs, processors and memories) mounted on a board or chip carrier and single-chip systems consisting of an ASIC with one or more processor cores and/or memories. The programmable core is usually an ISP, ASIP or DSP provided as a macro-cell in the ASIC implementation technology. (Figure 3 [19] and Figure 6 [27]).

Whereas a core may provide the same functionality as the corresponding standard part, cost and performance considerations may bias the choice of integration level. The advantages of higher integration are usually higher reliability, lower power budget and increased performance. The last two factors come from the lack of I/O circuitry in the core and its direct connection to the application-specific logic. The disadvantages are larger chip sizes and higher complexity in debugging the system.

3. System-level co-design issues

We consider now system-level design problems as originating from specific application-domain requirements and from broad realization choices.

3.1. GENERAL-PURPOSE COMPUTING SYSTEMS

The primary goal of instruction-set processors is to be the engine of general-purpose computing systems, even though today ISPs are used also in dedicated computing systems. Co-design of ISPs is peculiar because performance requirements are at the extreme edge allowed by the current semiconductor technology. High performance is achieved by a combination of hardware and software choices.

A co-design task in ISP development is providing architectural support for operating systems, because the operating system is the software layer that has most interactions with the underlying hardware. Note that some dedicated computing/control system use either a specialized operating system (e.g., real-time kernel [49]), or a simplified operating system, or no operating system at all [13]. Nevertheless, since the primary use of ISPs is in general-purpose computing, architectures are always thought of with particular target programming languages and operating systems in mind.

Compiler development should start as early as architectural definition. Indeed, compilers are needed for the evaluation of the the choices of instruction set and overall processor organization, to verify whether the overall performance goals are met. Whereas retargetable compilers are useful in the architectural development phase, optimizing compilers are key to achieve fast-running code on the final products.

The organization of modern processors exploits deep pipelines, concurrency and memory hierarchies. Hardware/software trade-off is possible in pipeline-control and cache-management mechanisms. For example, an important co-design issue is determining the cache-memory size and selecting a cache-management algorithm, which matches the cache size and speed. Cache management may be implemented using hardware or software schemes, or a combination of both. Cache-management algorithms are especially important for the design of multi-processor computing systems, where cache coherence has to be maintained [30].

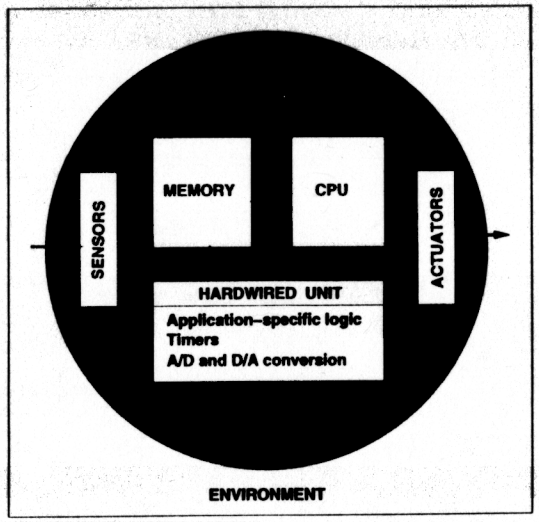

Figure 7. Scheme of the essential parts of an embedded control system

The selection of an instruction set is usually guided by performance and compatibility goals. This task is generally based on experience and on the evaluation of software simulation runs, even though recent efforts have aimed at developing tools for computer-assisted instruction set selection [31]. ISP co-design and performance evaluation issues are presented in Chapters 2 and 3 respectively.

3.2. DEDICATED COMPUTING AND CONTROL SYSTEMS

Both general-purpose DSPs and ASIPs are commonly used for dedicated (embedded) data-processing systems. Differently from ISPs and general-purpose DSPs, ASIPs may be designed to support fairly different instruction sets, because compatibility requirements are less important and supporting specific instruction mixes is a desired goal [2]. Unfortunately the price of the flexibility in choosing instruction sets is the need of developing application-specific compilers. Despite the use of *retargetable-compiler* technology [38], the computer-aided development of compilers that produce high-quality code for specific architectures is a difficult problem and solved only in part to date, namely for fixed-point arithmetic operations. Thus, many designers still use assembly-code programming for embedded applications, which is a long, tedious and error-prone task. Today, the overall design time is higher for the software portion than for the hardware portion

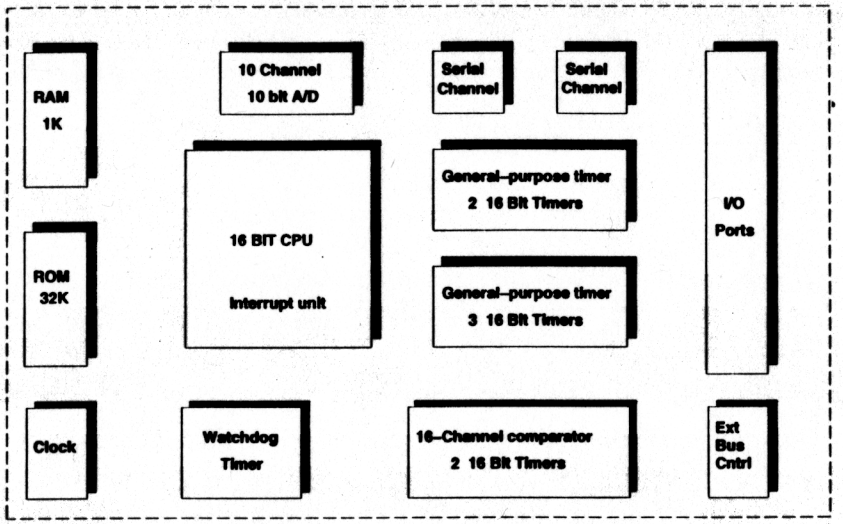

Figure 8. Block diagram of the ST10 microcontroller

of design for embedded applications (e.g., telecom [43].) Chapters 4, 5 and 13 are devoted to co-design issues in telecommunication applications.

Embedded control systems are dedicated systems that usually control mechanical components via *actuators*. Input data are provided by *sensors*, that are sometimes integrated on the same chip. (Figure 7). Some inputs/outputs can be analog signals, whose conversion to and from the digital representation may be done on chip. Often embedded control systems have a data-processing component.

Control systems are *reactive* systems, because they are meant to react to stimuli provided by the the environment. *Real-time* control systems [49] implement functions that must execute within predefined time windows, i.e., satisfy some *hard* or *soft* timing constraint [57, 58]. Timers are thus important components of real-time controllers.

The required functions and size of embedded controllers may vary widely. Standard programmable micro-controllers and micro-controller cores provide usually low-cost and flexible solutions to a wide range of problems (Figure 8). Nevertheless, control systems that are either complex (e.g. avionic controls) or that require high-throughput data processing (e.g., radio-navigation) need specific designs that leverage components or cores such as ISPs, DSPs, and/or ASIPs.

Whereas performance is the most important design criterion for information processing systems, *safety* and *reliability* are extremely important

for control systems. Safety means that the system performs the desired functions under any possible environmental condition. Reliability measures the probability of correct control operation even in presence of failures of some component. Since control functions can be implemented both in hardware and in software, specific design disciplines must be used to insure safety and reliability. Some formal verification techniques for embedded controllers are nowadays available to insure design correctness by comparing different representation levels [13] or assessing system properties. System-level testing techniques must be used to check the correctness of operation of the physical system implementation. Functional redundancy may be used to enhance reliability.

Specific co-design problems for dedicated computing and control systems include modeling, validation and synthesis. These tasks may be complex because the system function may be performed by different components of heterogeneous nature, and because the implementation that optimizes the design objectives may require a specific hardware/software partition. These tasks and the corresponding co-design tools will be described in more detail in Section 4. In addition, embedded control systems are described in Chapters 9 and 10, while Chapter 12 presents specific issues in automotive electronics.

3.3. EMULATION AND PROTOTYPING SYSTEMS

We consider first how co-design techniques can accelerate software execution. There are often bottlenecks in software programs that limit their performance (e.g., executing transcendental floating-point operations, inner loops where sequences of operations are iterated). ASIC co-processors reduce the software execution time, when they are dedicated to support specific operations (e.g., floating-point or graphic co-processors) or when they implement the critical loops in hardware while exploiting the local parallelism.

Whereas ASIC co-processors accelerate specific functions, co-processors based on programmable hardware can be applied to the speed-up of arbitrary software programs with some distinctive characteristics.

One of the first examples of programmable co-processors is provided by the *Programmable Active Memories* (PAMs) [6], which consist of a board of FPGAs and local memory interfaced to a host computer. Two models of PAMs, named *PeRLe-0* and *PeRLe-1*, were built. They differ in the number and type of FPGA used, as well as operating frequency. The hardware board for *PeRLe-1* is shown in Figure 9 [55].

To accelerate the execution of a program with a PAM, the critical portion of the program is first extracted and compiled into the patterns that

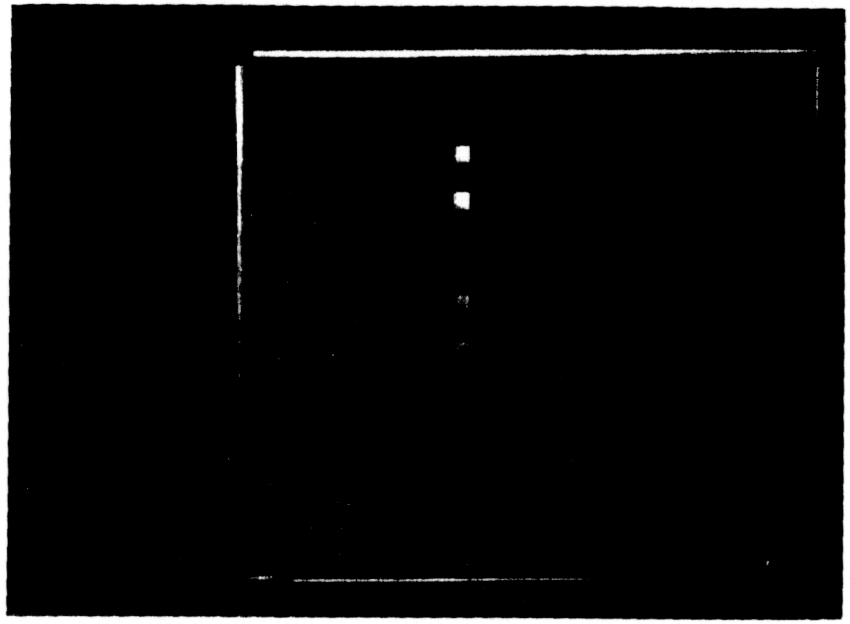

Figure 9. The *PeRLe-1* implementation. Courtesy of P.Bertin

configure the programmable board. Then, the non-critical portion of the program is executed on the host, while the critical portions are emulated in hardware. Experimental results show a speed-up of one to two orders of magnitude, as compared to the execution time on the host [6].

The major hardware/software co-design problems consist of identifying the critical segments of the software programs and compiling them efficiently to run on the programmable hardware. The former task is not yet automated for PAMs and is achieved by successive refinement, under constraints of communication bandwidth and load balancing between the host and the programmable hardware. The latter task is based on hardware synthesis algorithms, and it benefits from performance optimization techniques for hardware circuits [6, 15].

Another example of the use of programmable hardware is the acceleration of system-level simulation by hardware emulation [42], as described in Chapter 14. In this case, a co-design methodology is used to improve upon simulation tools, which in turn are used for the co-design of other systems.

Computer-aided prototyping can be done by configuring a programmable hardware board so that it emulates the functionality of the target system. This technique is very useful to validate a hardware circuit before man-

ufacturing it, and thus reducing the likelyhood of an expensive redesign. Prototypes provide design engineers with more realistic data on correctness and performance than system-level simulation [40].

Prototyping of complex digital systems including multiple hardware components and software programs is appealing to designers, because it allows to test the execution of software programs on hardware, while retaining the ability of changing the hardware (and software) implementation concurrently. Once the hardware configuration has been finalized, it can be mapped onto a 'hard' silicon implementation using synthesis systems [15], that accept as inputs hardware models compatible with those used by the emulation systems (e.g., *VHDL* and *Verilog HDL* models).

4. Computer-aided co-design methods and tools

A design team who is conceiving a digital system has to to produce an implementation that satisfies the system-level specifications in a short time. Moreover, it must maximize the system value while reducing the cost. These two objectives are often articulate: the value is related to the performance (often measured on an application), to the power consumption, and to the size of the potential market, i.e. to the degree to which the product provides solutions to various users. The cost is often related to the number of hardware parts that need to be designed, manufactured or acquired, to the size of the silicon dies (i.e., to yield and packaging costs) and to the cost of software development (e.g., compilers). The value and cost are also related to the ease of debugging, testing and extending the systems with new features, as well as to reliability, maintenance and servicing issues.

Computer-aided co-design tools can be instrumental in achieving high-valued system design with limited cost for several reasons. CAD tools require formal system-level specifications, by means of hardware/software languages or charts, which leads to a structured design methodology. Language-based specifications facilitate hardware and software re-use, because hardware and software components can be modeled as programs in the specifications languages and shared among different projects. Moreover, coherent design specifications support analysis and validation tools (e.g., system-level simulation), which can reduce design time by pointing out potential design errors. Computer-aided synthesis tools can explore rapidly different design alternatives, in the search for those implementation choices that maximize the system value and minimize the cost, as well as generating automatically detailed low-level descriptions of the implementation and thus reducing design time.

Today only some co-design problems are efficiently solved by CAD tools and a comprehensive computer-aided co-design methodology is still in an

embryonic stage. The overall objective in co-design tool research and development is providing integrated environments for concurrent specification, validation and synthesis of both hardware and software. We describe next the state of the art of modeling, validation and synthesis.

4.1. MODELING

Modeling digital systems is often complicated by the heterogeneity of its components. Thus, various modeling styles are used today and each one fits best a specific application. We need to distinguish first between the models of computation and the hardware/software languages. Computation models have an underlying mathematical structure (e.g., Petri nets, finite-state machines). Some hardware/software languages are means of expressing mathematical models, while others serve the purpose of describing systems and/or computation without a formal semantics. Computational models and description styles are described in Chapters 8, 9 and 16. We consider here only some approaches used in practice for system design.

Functional modeling of digital systems is often done using programming languages, such as *C* or *C++*. These functional models are used to check generic properties of the system and to derive some measures of its performance and cost. Such models are applicable to hardware and software components with no distinction, but the unfortunate drawback is that programming-language models of digital hardware are too general and do not capture all necessary specifications of the hardware component.

Specification models are used to describe the required behavior. There are different flavors of models characterized by abstraction levels and semantics. While we do not want to propose here a complete list of specification paradigms, we mention some that are representative of specific styles and find some use in design of systems which are manufactured.

Some systems can be broadly described using the *finite-state machine* (FSM) model, or interconnection of FSMs. Such systems can be represented graphically by *Statecharts* [26] or by its derivatives [20, 41]. Alternatively, they can be modeled by textual languages such as *SDL* [48] (which has also a graphical representation) or *Esterel* [5].

Hardware components that can be modeled as partial orders of tasks find in *VHDL* [45] and *Verilog HDL* [52] their most natural description. Synchronous data-flows (as in DSP circuits) can be described by applicative languages like *Silage* [29] or *DFL* [59].

The search for a perfect language [17] has always fascinated researchers. As in other domains, the idealistic goal of developing a single language (textual or graphical) that can express all desired features has not been achieved, because of the heterogeneity of the system components, conflict-

ing interests in communities of CAD tool providers and users, and the lack of an expressive formalism that can capture well all features of interest.

As far as co-design is concerned, specification and design *frameworks* that support multiple and extensible design front-ends seem to be the foreseeable solution. Such frameworks should support then a variety of tools for *validation* [35] and *synthesis* [34, 50] of the hardware and software components. It is also obvious that there exists some application domains where the components of interest are homogeneous enough that a single specification language suffice.

4.2. VALIDATION

System-level validation (called also co-validation) means gaining a reasonable certainty that the design is free from errors and therefore is ready for manufacturing. As circuits and systems become increasingly complex, validation becomes more important as well as more difficult. Validation can be done by means of formal verification, simulation (or co-simulation) or emulation. Verification tools check the congruency of design representations and/or try to prove specific properties (e.g., the existence of no deadlock conditions). Today verification is practically applicable in some restricted domains, where the modeling style is formal and homogeneous. An example is provided by finite-state models of systems, where a variety of techniques developed for verifying hardware FSM implementations can be used [7, 8, 13]. Another example is given by methods based on theorem-proving, as described in Chapter 17. Unfortunately, the state-of-the-art is far from the stage where complex hardware/software systems can be fully verified, because they require often heterogeneous modeling paradigms and because the properties to be verified may be of different nature.

Simulation is the traditional way of validating hardware correctness, by examining a set of output responses to input stimuli. Whereas simulation tools are widely available (even though only few support mixed hardware/software specifications), their use cannot insure the correctness of the design because only a small percentage of input/output patterns can be analyzed in large-scale designs. Nevertheless, simulation tools are very useful for co-design.

Let us consider first how heterogeneity applies to different application domains. Lumped, general-purpose computing systems consist of one (or more) hardware processors, some dedicated hardware circuits and a set of software layers. These systems can be considered as weakly heterogeneous, because they consist essentially of hardware components, that can be modeled in a *hardware description language* (HDL) such as *VHDL* or *Verilog HDL*, in addition to software components that can be expressed in a programming language. Similar considerations apply to simple embedded control systems, i.e., not requiring analog/digital conversion and data processing. Conversely, data processing and communication systems, possibly coupled with controllers, as well as distributed systems, require heterogeneous modeling. Examples are cellular telephones (having software, digital, analog and radio-frequency components), networks for computing or communication (requiring to model the communication medium), and avionic control systems (integrating radio-navigation functions with data-processing and control functions). Whereas weakly heterogeneous systems can be simulated by extending HDL simulators, strongly heterogeneous systems require specialized simulation environments, that will be described later.

We consider next different strategies for simulation of weakly heterogeneous systems. A simulator should provide the following desirable features: adequate timing accuracy, fast execution and visibility of the internal registers for debugging purposes [47]. Unfortunately, designers often face the following problems. First, when validating hardware/software systems, it is often necessary to simulate large software programs to get meaningful results. Since the simulation of each software instruction corresponds to many events in the hardware, then detailed hardware models require very long simulation run times. Second, some simulation paradigms require hardware models in HDLs. Availability of hardware models at the desired abstraction level, through development or acquisition, may be a decisive factor for reaching the simulation objectives. Finally, the availability of models in *VHDL* and *Verilog HDL* and of fast simulators for these languages restrict often the modeling language choices.

The first simulation strategy that we consider is to use an HDL simulator, with a model for the processor and for application-specific circuits. If a precise timing model is used, accurate timing and complete functionality of the system can be obtained at the expense of long simulation runs. If zero-delay models are used, the correct transitions at the clock edges can be monitored, thus providing a way to assess the correctness of synchronous systems with shorter run times. Alternatively, and instruction-set model of the processor can be used, that captures the processor behavior in response to the instructions, while insuring correct register and memory values. Even faster simulation can be obtained, at the expenses of timing accuracy.

The second simulation strategy is to avoid processor models. A limiting assumption is now that the system to be simulated has a hardware/software communication protocol such that communication delay has no effect on the system functionality. Then the software component can be compiled and the object code linked to the HDL simulator, which simulates the model of the application-specific hardware. With this strategy, hardware/software

communication is replaced by handshake, that can be implemented via different mechanisms (e.g. interprocess communication [4]). In this case, the speed of the simulation is limited by the HDL simulator speed, but the internal register information is not observable.

A third simulation strategy is to emulate the hardware component by mapping it onto programmable hardware. The emulation system may have different configurations. For example, a board of FPGAs can be coupled to a processor and/or to a user-board where standard components and memories can be placed and connected. Olukotun described a simulator that performs a preprocessing step in which the system model (to be simulated) is automatically partitioned and scheduled to be executed on the hardware platform so that run time is minimized [42]. (See also Chapter 14). In general, the speed of emulation is roughly one order of magnitude less than the execution on the target hardware. Debugging may be slow, because of the limited visibility of internal states [47].

Simulation of strongly heterogeneous and distributed systems can be performed by specialized simulators, such as PTOLEMY [35]. PTOLEMY has an extensible, object-oriented kernel and supports several computation models. The models are not implemented in the kernel, but in domains that can interact without knowing their semantics. Thus, multiple models of specification and simulation can be used for the same system. PTOLEMY provides some domains (e.g., data-flow and discrete-event models), while others can be developed by the user. PTOLEMY has been successfully used in the field of signal processing and communication-system design. (See also Chapter 8).

4.3. SYNTHESIS

Synthesis of digital systems consists of deriving a detailed representation of its implementation, starting from system-level specifications. It is also called co-synthesis, because it involves both hardware and software synthesis.

The principal component of general-purpose computing systems is the processor. Hardware synthesis techniques may be applied to behavioral models of processors. Nevertheless, many high-level design decisions are not automated, such as pipeline organization and data-path design, because designers like hand-crafted solutions, sometimes *ad hoc*, that yield the best performance with the available technology. The rationale is that few novel ISPs are designed and implemented (as compared to ASICs) and that performance is so sensitive to such design choices that manual design and tuning is justified. Moreover, the high-volume production projected for processor design permits to recover high design costs. On the other hand, control functions are often synthesized from behavioral models in state of the art processors.

The design of a software compiler bears strict relation to the organization of a processor architecture, and thus compilers are co-designed with processors. Only recently, automated tools have been proposed to co-synthesize the pipeline control unit and the compiler back-end [32, 33], in the search of the best compromise to avoid pipeline hazards and optimize performance.

Synthesis of lumped embedded systems is the natural evolution of existing hardware high-level synthesis methods [15]. A working hypothesis is that the overall system can be modeled consistently and be partitioned, either in the original specification, or manually, or automatically, into a hardware and a software component. The hardware component can be implemented by application-specific circuits using existing hardware synthesis tools; the software component can be generated automatically to implement the function to which the processor (or core) is dedicated. Synthesis must also provide a means for interfacing and synchronizing the functions implemented in the hardware and software components. Synthesis of application-specific hardware is described in Chapter 15. We concentrate here on the other tasks.

4.3.1. *System-level partitioning*

The overall system cost and performance are affected by its partition into hardware and software components. At one end of the spectrum, hardware solutions may provide higher performance by supporting parallel execution of operations, at the expense of designing and fabricating one (or more) ASICs. At the other end of the spectrum, software solutions may run on high-performing processors available at low cost due to high-volume production. Nevertheless, operation serialization and lack of specific support for some tasks may result in loss of performance. Thus a system design for a given market may find its cost-effective implementation by splitting its functions between hardware and software.

System-level partitioning into hardware and software components has been investigated by several researchers [3, 18, 23, 24, 51, 53]. We shall mention two approaches: the synthesis of dedicated co-processors for software execution acceleration [18] and the migration of non-critical functions to software [24]. The two problems have complementary objectives: the former attempts to maximize performance while the latter tries to minimize the system cost, subject to performance constraints.

The COSYMA synthesis tool suite [18] partitions a system specification to accelerate software execution by using a dedicated hardware co-processor (to be synthesized). The original system model is described as a software program in the C^* language which is an extension of the C programming

language to support performance constraints. Thus a software implementation of the initial model is readily available (by compilation) but it may be delivering performance inferior to the expectations. The system model is compiled into a control/data-flow graph and a partitioning algorithm identifies the computational bottlenecks and then migrates the corresponding functions to application-specific hardware. As an example of an application, the *chromakey* algorithm for high-definition television (HDTV) running on a SPARC processor has been expedited by a factor of three by identifying a critical loop that takes 90 % of the software run time and fabricating an ASIC with 17000 equivalent gates.

The VULCAN synthesis tool suite [22, 23, 24] uses instead a hardware model of the system and attempts to reduce the cost of its implementation by transferring non-critical operations to a standard processor or processor core, such as an *I8086* or an *R3000*. In particular, systems are modeled in hardware description language *HardwareC*, that has a *C*-like syntax but a hardware semantics. Performance requirements, in terms of *latency* and *data-rate* constraints, are specified in the system model. Whereas a hardware implementation can be derived from the *HardwareC* model (by hardware synthesis [16]) the co-synthesis approach is as follows. The system model is compiled into a control/data-flow graph, which is partitioned yielding: i) a set of software threads to be compiled and executed on the standard processor and ii) the specification of the remaining hardware circuits, that can be synthesized as a netlist of logic gates. Hardware/software synchronization units for interfacing the processor to the application-specific logic are also automatically generated [22, 24, 25].

Despite the appeal of automatic synthesis techniques, some obstacles need to be overcome before computer-aided system partitioning becomes practical. First of all, the quality of a partition depends on performance/cost estimators based on an abstract system representation (control/data-flow graph). Such estimators need to be fast, because they are invoked frequently by the partitioning algorithm. Developing precise fast estimators from abstract models is a difficult problem, still under investigation [58]. (See Chapter 3). Second, designers may be more interested in a coarse-grained partition of a system, i.e., in finding the hardware/software allocation of a set of modules in which the system-level specification is partitioned. This approach contrasts those described before, in that we loose the degrees of freedom for optimizing system value and cost using a fine-grained operation-level representation. On the other hand, designers' expertise can be more easily exploited in achieving a desired partition which privileges some macroscopic choices.

4.3.2. *Software synthesis and retargetable compilation*

Software synthesis is a generic term describing the production of machine-level code for embedded applications. The major task in software synthesis is compilation, which may take different flavors according to the target hardware architecture. Compilation may be preceded by the automatic generation of the software program to be compiled, in the case that either the software component is specified in a formalism that is not directly compilable (e.g., *SAO* [46], FSM state diagrams) or in the case that the software component consists of fragments obtained by automatic partitioning tools [24].

Software compilation for embedded applications differs from compilation for general-purpose computing systems in three main respects. First, embedded systems run dedicated code that needs to be compiled once. Hence compilation time is not important and specific code-optimization techniques can be used without worrying about long compile times. Second, embedded systems have often to satisfy real-time constraints. Thus the compiled code has not only to be compact but also to satisfy specific timing requirements. Tight requirements on code quality lead often designers to use assembly-code programming. It is one of the purposes of software synthesis research to provide designers with good-quality compilation tools for high-level languages such as *C* and *C++*. Third, embedded systems use often ASIPs, which have specific instruction sets and irregular structures. For example, some ASIPs have instructions with operand and result locations that are instruction-specific, non-homogeneous register sets and specific interconnections among the computing and storage resources. Thus, compilers have to deal with several peculiarities.

It is obvious that designing separate compilers for each ASIP architecture is unreasonable, due to the large number of ASIP designs and to the effort needed to develop a compiler. Thus, techniques for porting compilers to different architectures have been proposed and such compilers are called *retargetable compilers* [36, 43]. To be more precise, Marwedel suggested to classify these compilers according to the time needed to retarget them to a specific architecture [38]. A *portable* compiler is a compiler that can be adapted to a specific architecture by rewriting a significant portion and therefore whose portability requires time and effort. A *compiler-compiler* is a program that can generate a compiler from a description of the target architecture. Thus generating a new compiler is essentially describing the target architecture and executing the compiler-compiler. Finally, a *machine independent* compiler is one that contains a full suite of pattern matching routines for different architectures.

Usually compilers have a source-code-dependent front-end, an intermediate code generation/optimization stage, and a code generation back-end.

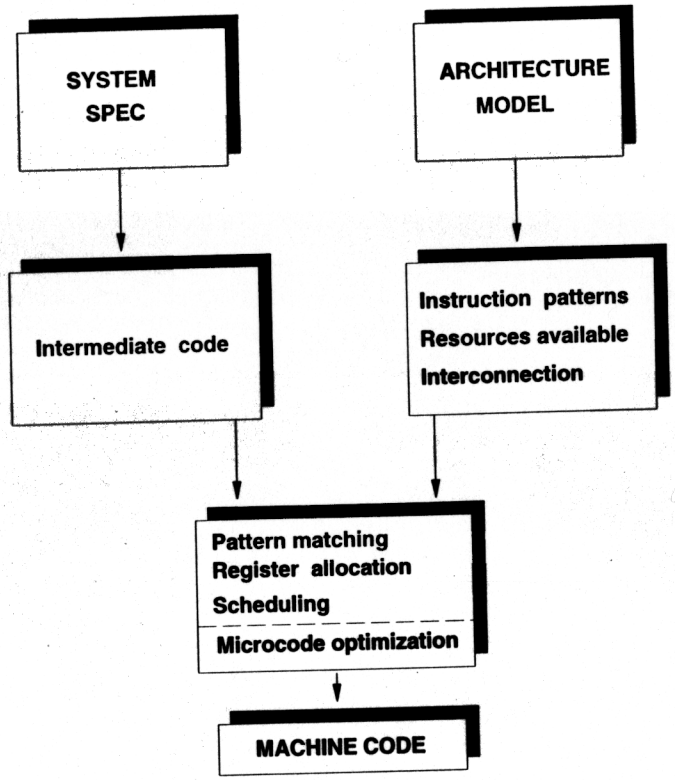

Figure 10. Major tasks in a retargetable compilation

Retargetable compilation differs from standard compilation techniques in the code generation back-end. The major tasks of code generation are *instruction selection*, *register allocation* and *instruction scheduling*. These three tasks interact strongly with each other. Instruction selection means finding the appropriate instructions in the target architecture to execute the intermediate code. There are two subproblems: matching the intermediate-code to the target patterns [39] and selecting the best ensemble of instructions to implement a program fragment. Often dynamic programming techniques are used for the selection step [43]. Register allocation means assigning operands and results to the available registers, while satisfying the data routing paths available in the architecture. A major goal of register assignment is to avoid *spills* to memory, which usually involves some delay. Scheduling means finding the appropriate sequence of operations. Sometimes a schedule must satisfy real-time constraints, thus constraining the

relative spacing between instruction pairs. Exact and heuristic algorithms have been proposed for this task [11, 12, 25, 36].

Some ASIPs support a few parallel instruction streams and their control unit is microprogrammed. In this case, an important problem is compacting the microcode, i.e., determining the instruction-level parallelism and the encoding of each word. Several classical [1] and newer [43] algorithms can be used for this step. Approaches and algorithms for retargetable compilation are described in Chapters 6 and 7.

4.3.3. *Interface synthesis*

Several problems fit under the umbrella of interface synthesis. In general, interface synthesis in system-level design consists of generating the software routines, and/or hardware circuits, that interface processors or application-specific components to a communication channel operating under a given protocol. Interface synthesis is motivated by the fact that several standards are used in digital design (e.g., *PCI, VME*) and that system-level models should avoid specifying the details of the communication mechanism. For example, Wenban et al. [56] described how to model communication protocols with language *Promela* and to derive from the *Promela* model *C++* routines for processors (or cores) as well as interface circuits (described at the gate level) for the application-specific component of a system.

Specific interfacing problems arise when system-level specifications are partitioned automatically into hardware and software components that exchange data. Then, a communication and synchronization mechanism must be provided to interface the hardware and software components. This mechanism can be embodied by a hardware circuit and/or a software routine [22, 24].

Another specific problem is interfacing processors to to peripheral devices, such as sensors and actuators, in embedded system design. Computer aided design tools such as CHINOOK [10] provide a means of interfacing automatically peripheral devices to a processor. This entails allocating the processor ports to the devices and deciding whether device drivers should be implemented by software running on the processor or by dedicated hardware. Another important problem is scheduling the processor communication [11, 12], which may be complicated by the presence of real-time constraints as well as of tasks with data-dependent delays [12]. Details are presented in Chapter 10.

5. Summary

The progress in electrical system design will depend, among other factors, on the level of support provided by computer-aided design tools. In par-

ticular, the design of the digital component of systems will benefit from performing concurrent hardware/software design and thus from exploiting the synergism of hardware and software in the search for solutions that use at best the current manufacturing technology and the availability of hardware components and software programs.

At present, the overall CAD support for hardware/software co-design is still weak, but growing at a rapid pace because the potential payoffs make it an attractive area for research as well as an exciting business opportunity. Whereas simulation tools have reached a level of maturity such that some co-simulators are commercially available, algorithms and techniques for system-level synthesis are still in the research stage. Nevertheless, product-level use has been reported of both high-level synthesis of hardware components and retargetable compilation.

Overall, hardware/software co-design is a wide field of research, because of the diversity of applications, design styles and implementation technologies. It is likely that the impact of CAD tools on system-level design will be more profound than the impact of CAD tools on integrated circuit design.

6. Acknowledgments

This survey has been sponsored by NSF under grant # INT 9123222 and by ARPA under contract # DABT63-95-C-0049.

References

1. T.Agerwala, "Microprogramming Optimization: A Survey", *IEEE Transactions on Computers*, Vol C-25, No. 10, pp. 962-973, October 1976.
2. A.Alomary, T.Nakata, Y.Honma, M.Imai and N.Hikichi, "An ASIP Instruction set Optimization Algorithm with Functional Module Sharing Constraints", *Proceedings of ICCAD*, pp. 526-532, 1993.
3. E.Barros, W.Rosenstiel and X.Xiong, "A Method for Partitioning UNITY Language in Hardware and Software", *Proc. of EURODAC*, pp. 220-225, 1994.
4. D.Becker, R.Singh and S.Tell, "An Engineering Environment for Hardware-Software Co-simulation", *Proceedings of DAC*, pp. 129-134, 1992.
5. A.Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems", *IEEE Proceedings*, Vol. 79, No.9., pp.1270-1282, September 1991.
6. P. Bertin and D. Roncin and J. Vuillemin, "Introduction to Programmable Active Memories", in J.McCanny, J.McWhirter, E.Schwartzlander (Editors), *Systolic Array Processors*, Prentice Hall, 1989.
7. J.R. Burch and E. M. Clarke and K.L. McMillan and D.L. Dill and L.J.Hwang, "Symbolic Model Checking: 10^{20} States and Beyond", *Information and Computation*, Vol. 98, No. 2, pp. 142-170, June 1992.
8. Jerry R. Burch and Edmund M. Clarke and David E. Long and Kenneth L. McMillan and David L. Dill, "Symbolic Model Checking for Sequential Circuit Verification", *IEEE Transactions on CAD/ICAS*, Vol. 13, No. 4, pp. 401-424, April 1994.
9. D.Bursky, "Microcontroller Design Exploits Reusable Cores", *Electronic Design*, Vol. 42, No. 6, pp. 53-68, March 1994.

10. P.Chou, E.Walkup and G.Borriello, "Scheduling Strategies in the Co-Synthesis of Reactive Real-Time Systems", *IEEE Micro*, Vol. 14, No. 4, pp. 37-47, August 1994.
11. P.Chou and G.Borriello, "Software Scheduling in Co-synthesis of Reactive Real-Time Systems", *Proceedings of DAC*, pp. 1-4, 1994.
12. P.Chou and G.Borriello, "Interval Scheduling: Fine-Grained Code Scheduling for Embedded Systems", *Proceedings of DAC*, 1995.
13. M.Chiodo, P. Giusto, A. Jurecska, H. Hsieh, L.Lavagno and A. Sangiovanni, "A Formal Methodology for Hardware/Software Co-design of Embedded Systems", *IEEE Micro*, Vol. 14, No. 4, pp. 26-36, August 1994.
14. G. De Micheli, Computer-Aided Hardware/Software Co-design, *IEEE Micro*, Vol. 14, No. 4, pp. 10-16, August 1994.
15. G. De Micheli, *Synthesis and Optimization of Digital Circuits*, Mc Graw-Hill, 1994.
16. G. De Micheli, D. Ku, F. Mailhot, and T. Truong, "The Olympus Synthesis System for Digital Design," *IEEE Design & Test* , pp. 37-53, October 1990.
17. U.Eco, *La Ricerca della Lingua Perfetta*, Laterza, 1993.
18. R.Ernst, J.Henkel and T. Benner, "Hardware-Software Co-synthesis for microcontrollers", *IEEE Design & Test*, pp. 64-75, December 1993.
19. C.Feigel, "Processors Aim at Desktop Video", *Microprocessor Report*. Vol. 8, No. 2, February 1994.
20. D. Gajski, F.Vahid, S. Narayan, F.Vahid and J.Gong, *Specification and Design of Embedded Systems*, Prentice-Hall, 1994.
21. J.Green, E. Hamdy and S.Beal, "Antifuse Field Programmable Gate Arrays", *IEEE Proceedings*, Vol. 81, No. 7, pp. 1041-1056, July 1993.
22. R. Gupta, *Co-synthesis of Hardware and Software for Digital Embedded Systems*, Kluwer, 1995.
23. R. Gupta, Claudionor Coelho and G. De Micheli, "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components", *Proceedings of DAC*, pp. 225-230, 1992.
24. R. Gupta and G. De Micheli, "System Co-synthesis for Digital Systems" *IEEE Design & Test*, Vol. 10, No. 3, pp. 29-41, September 1993.
25. R. Gupta and G. De Micheli, " A Co-synthesis Approach to Embedded System Design Automation" *Embedded System Journal*, (to appear).
26. D. Harel, A.Pneuli, J. Schmidt and R.Sherman, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, No. 8, pp. 231-274, 1987.
27. M. Harrand et al, "A Single Chip Videophone Video Encoder/Decoder", *Proc. of IEEE International Solid-State Circuits Conference*, pp. 292-293, February 1995.
28. J. Hennessy and D.Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1990.
29. P.Hilfinger, "A High-Level language and Silicon Compiler for Digital Signal Processing", *CICC, Proceedings of the Custom Integrated Circuit Conference*, pp. 213-216, 1985.
30. M.Horowitz and K.Keutzer, "Hardware-Software Co-Design", *Proceedings of SASIMI*, Nara, pp. 5-14, 1993.
31. B.Holmer, "A Tools for Processor Instruction Set Design", *Proceedings of DAC*, pp. 150-155, 1994.
32. I. Huang and A. Despain, "High-level Synthesis of Pipelined Instruction Set Processors and back-end Compilers", *Proceedings of DAC*, pp. 135-140, 1992.
33. I. Huang and A. Despain, "Hardware/Software Resolution of Pipeline Hazards in Pipeline Synthesis of Instruction Set Processors", *Proceedings of DAC*, pp. 594-599, 1993.
34. T.Ismail and A.Jerraya, "Synthesis Steps and Design Models for Codesign", *IEEE Computer*, No.2, pp. 44-52, February 1995.
35. A. Kalavade and E. Lee "A Hardware-Software Co-design methodology for DSP Applications", *IEEE Design & Test*, Vol. 10, No. 3, pp. 16-28, September 1993.
36. D.Lanneer, J. Van Praet, A. Kifli, K.Schoofs, W.Guerts, F. Thoen and G.Goossens,

"CHESS: Retargetable Code Generation for Embedded DSP Processors", in P.Marwedel, G.Goossens, Editors, *Code Generators for Embedded Processors*, Kluwer, 1995.

37. C.Liem, T. May and P. Paulin, "Instruction-Set Matching and Selection for DSP and ASIP Code Generation", *Proceedings of the European Design and Test Conference*, pp. 31-37, 1994.
38. P.Marwedel, G.Goossens, Editors, *Code Generators for Embedded Processors*, Kluwer, 1995.
39. P.Marwedel, "Tree-based Mapping of Algorithms to Predefined Structures", *Proceedings of ICCAD*, pp. 586-593, 1993.
40. L. Maliniak, " Logic Emulator Meets the Demands of CPU Designers", *Electronic Design*, April 1993.
41. S. Narayan, F.Vahid and D.Gajski, "Translating System Specifications to VHDL", *Proceedings of EDAC*, pp. 390-394, 1991.
42. K.Olukotun, R.Helahel, J.Levitt and R. Ramirez, "A Software/Hardware Cosynthesis approach to Digital System Simulation", *IEEE Micro*, Vol. 14, No. 4, pp. 48-58, August 1994.
43. P.Paulin, C.Liem, T.May, and S.Sutarwala, "Flexware: A Flexible Firmware Development Environment for Embedded Systems", in P.Marwedel, G.Goossens, Editors, *Code Generators for Embedded Processors*, Kluwer, 1995.
44. P.Paulin, C.Liem, T.May, and S.Sutarwala, "DSP Design Tool Requirements for Embedded Systems: a Telecommunications Industrial Perspective", *Journal of VLSI Signal Processing*, No. 9, pp. 23-47, 1995.
45. D.Perry, *VHDL*, McGraw-Hill, 1991.
46. M.Rohmdani, A. Jerraya, P. de Chazelles and A. Jeoffry, "Composing Activity Charts/StateCharts, SDL and SAO specifications for Codesign in Avionics", *Proceedings EURODAC*, 1995
47. J.Rowson, "Hardware-Software Co-Simulation", *Proceedings of the Design Automation Conference*, pp. 439-440, 1994.
48. R.Saracco and P.Tilanus, "CCITT SDL: Overview of the Language and its Applications", *Computer Networks and ISDN Systems*, Vol. 13, No.2., pp.65-74, 1987.
49. K.Shin and P.Ramanathan, "Real-Time Computing: A New Discipline of Computer Science and Engineering", *IEEE Proceedings*, Vol. 82, No. 1, pp. 6-24, January 1994.
50. M. B. Srivastava and R. W. Broderson, "Rapid-Prototyping of Hardware and Software in a Unified Framework", *Proceedings of ICCAD*, pp. 152-155, 1991.
51. M.Theissinger, P.Stravers and H.Veit, "CASTLE: a design Environment for codesign", *Proceedings of the International Workshop on Hardware/Software Co-design*, pp.203-209, Grenoble, September 1994.
52. D.Thomas and P. Moorby, *The Verilog Hardware Description Language*, Kluwer, 1991.
53. D.Thomas, J.Adams and H.Schmitt, "A Model and Methodology for Hardware-Software Co-design", *IEEE Design & Test*, Vol. 10, No. 3, pp. 6-15, September 1993.
54. S.Trimberger, "A Reprogrammable Gate Array and Applications", *IEEE Proceedings*, Vol. 81, No. 7, pp. 1030-1041, July 1993.
55. J. Vuillemin, P. Bertin, D. Roncin, M.Shand, H.Touati and P.Boucard "Programmable Active Memories: Reconfigurable Systems Come of Age", *IEEE Transactions on VLSI* (to appear).
56. A.Wenban, J. O'Leary and G.Brown, "Codesign of Communication Protocols" *IEEE Computers*, No. 12, pp. 46-52, December 1993.
57. W.Wolf and E.Frey, "Tutorial on Embedded System Design", *Proceedings of of ICCD*, pp. 18-21, 1992.
58. W.Wolf "Hardware-Software Co-Design of Embedded Systems", *Proceedings of IEEE*, Vol. 82, No.7, pp. 967-989, July 1994.
59. P.Willekens et al. "Algorithm Specifications in DSP Station using Data Flow Language", *DSP Applications*, Vol.3, No. 1, pp. 8-16, January 1994.