Chapter 2

Background and Related Work

Designers have always used abstraction as an aid in designing and building complex systems. These abstractions hide details of a lower level of the system that are not relevant at the higher level. For example, rather than design a complex circuit directly from transistors, a circuit designer uses logic gates which are collections of transistors that perform some useful function. These gates may be combined into larger subcircuits at the next level of abstraction. At a still higher level of abstraction, the behavior of a system may be described as a program and translated automatically into a circuit. A system that transforms a system description at one level into a description at another, more detailed, level is known generically as a *silicon compiler* [28]. There is a large and active research community investigating various forms of silicon compilation.

Advocates of silicon compilation are generally interested in synthesizing traditional synchronous circuits. An alternative approach is to use asynchronous circuit techniques instead. Even though asynchronous systems are not widely used, there is a body of work investigating these types of systems that goes back to the 1950's. The idea of designing circuits by writing program descriptions goes back to the late 1970's, but it is only recently that the idea of using program notation to describe asynchronous circuits has started to gain popularity.

2.1 Silicon Compilers

The term "silicon compiler" was introduced by Johannsen [44] in 1979 to describe the concept of automatically assembling parameterized pieces of layout into a larger system. This term has gained popularity and has now been applied to a huge variety of different approaches ranging from the automatic generation of PLA structures from Boolean expressions to synthesizing entire systems based on the description of an algorithm. One view of a silicon compiler is of a program that translates a system description directly into a

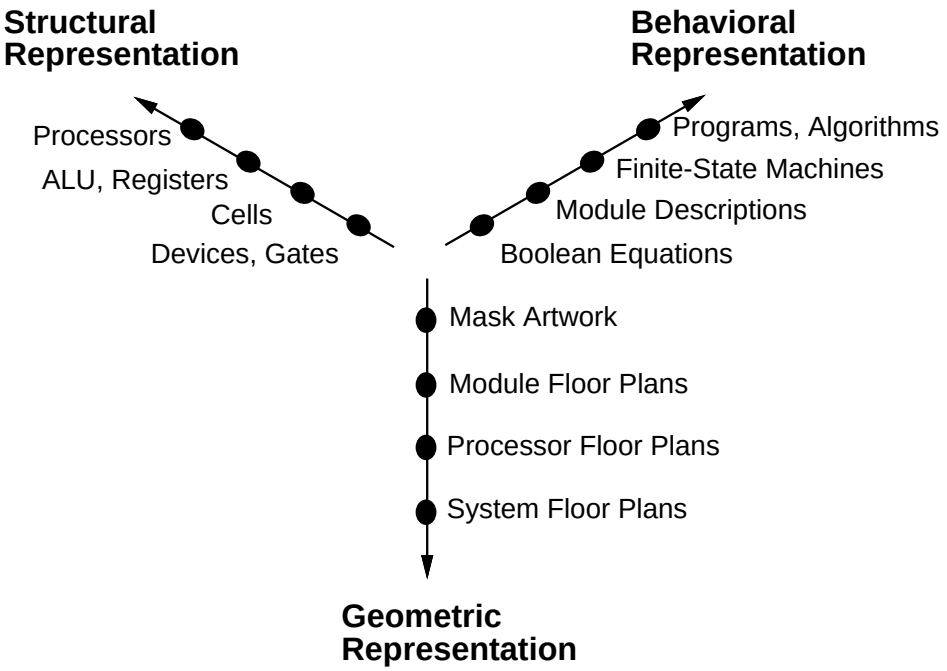

Figure 2.1: Representation of the Levels of Design Description

VLSI mask layout suitable for fabrication. Another view holds that translation between any of these levels could be called a silicon compiler, with a series of these more specialized compilers are used to compile from the source description to the desired level of detail.

The different approaches to silicon compilation can be described with the help of a *Y-chart* as shown in Figure 2.1 [28]. The axes in this chart represent three different domains of description: behavioral, structural, and geometric. Along each axis are different levels of the domain of description. The farther away from the center of the diagram, the more abstract the level of the description. Different approaches to silicon compilation can be represented as arcs along a domain's axis or between axes. In the following sections I give a brief background of some of the different approaches to silicon compilation.

2.1.1 Structural Silicon Compilers

One approach to designing systems using a silicon compiler is to impose restrictions on the target circuit and allow the designer to specify the structure of the desired system in terms of the restricted circuit types. This corresponds to moving inward along the structural arm in Figure 2.1. The attributes of the system described in the higher level description might include connectivity, absolute placement, relative placement, or versions of basic cells parameterized for word size.

For example, a description of an ALU from one level of the hierarchy could be compiled into an implementation as a set of standard cells. The descriptions of those cells might be compiled further into the mask artwork used to fabricate the cells on an integrated circuit. In both cases, it is the structure and connectivity of the circuit that is described at each level.

The lowest level structural silicon compilers are known as *cell compilers*. These programs translate cell descriptions containing only a few tens of transistors into mask layout. The resulting layout is usually constrained in terms of cell input and output format, device sizing, and the geometry of the mask layers. An early attempt to stylize the layout implementation of multilevel combinatorial logic is called a Weinberger array [105]. This structures limits gates to simple NOR gates and places the gates in a one-dimensional array. A technique for generating more complex gates is based on finding an Euler path in the graph derived from the circuit schematic [101]. In this layout method, parallel n-type and p-type diffusion regions are interrupted with polysilicon gates to form the complex gate. Gate matrix methods [52] extend this idea aligning gates as vertical rows of polysilicon with diffusion regions occupying different rows.

The goal of a cell compiler is to translate some textual description to the geometry of VLSI layout. Languages used to describe the various restricted layout styles range from simple descriptions of the absolute placement of rectangles that make up the layout on a fixed grid [106], to relative placement of rectangles on an abstract grid [51]. Rule-based expert systems have also been used to translate textual descriptions into layout [46].

A silicon compiler at a slightly higher level of abstraction is known as a *module compiler*. Modules are loosely defined to be microarchitectural entities that perform a small number of specific functions and consist of one or more arrays of cells or tiles of a specific type. For example, a Programmable Logic Array (PLA) can be used to implement arbitrary sum-of-products Boolean equations [61], or an extension of a PLA such as Path Programmable Logic (PPL) can implement boolean equations and storage elements in the same array [30]. These structures may be described as a generic template and a set of cells that occupy that template. The design of a module compiler consists of describing the template, tiles, interfaces, template personalization, and models used by other tools. The template may be described using a procedural language, or using a layout editor to describe geometric relationships [50, 60].

At a still higher level, *standard-cell* based systems describe the connectivity of modules. Johannsen's Bristle Blocks [44], for example, synthesized circuits consisting of a core of data path modules controlled by one central clocked controller. There are a large number of silicon compilers that echo this approach, most using some method for describing the clocked finite state controller and the placement and/or connection of the data path elements being controlled [2, 49].

2.1.2 Behavioral Silicon Compilers

Rather than use a language to describe the *structure* of the desired system, a different approach to silicon compilation is to use a language to describe the *behavior* of the system. This approach corresponds to moving along the behavioral arm of Figure 2.1 or moving from the behavioral arm to the structural arm. For example, rather than describing the placement of cells that make up a PLA, a designer may simply specify the Boolean function that the PLA is to implement. The silicon compiler can then generate the appropriate personalization of the PLA template. Moving up to higher levels of abstraction, early work in describing the behavior of circuits at a higher level suggested a similarity between optimizing compilers for programming languages and the problem of control step assignment in hardware systems [86].

MacPitts [87] is an early example of a silicon compiler that describes the behavior of a circuit as an algorithm rather than as a structure. The input language describes each control step separately and generates a simple floor plan consisting of a linear data path assembled from standard cells, and a Weinberger array control structure. A variety of silicon compilers have been developed that follow the same basic approach of defining the behavior of a system in terms of some special input language and generating a circuit description in terms of a set of modules [99, 98].

Once a silicon compiler can generate circuits from language descriptions, these circuits can be improved by allocating the available circuit resources in different ways. Investigation of the dataflow graph of the process and partitioning of that graph can result in circuit resources being shared by different parts of the process. The Design Automation Assistant (DAA) [47, 48] developed at Carnegie Mellon University, for example, used a rule base consisting of design knowledge gathered from expert designers to drive the compilation and resource allocation process. The DAA took system descriptions, translated them into dataflow representations similar to conventional software compilers, and then used the rule base to drive the assembly and optimization of the system. Buses were generated to communicate information, and opportunities to share hardware were discovered by manipulating the dataflow graph.

Silicon compilers have also been developed for special purpose applications such as signal processing [85], or language recognition [27], and special purpose hardware description languages have been developed that allow both structural and behavioral descriptions of circuits to be expressed in the same language [84]. More recently, general purpose computer languages like ADA have been used as hardware description languages for synchronous circuit implementation [69, 5].

The common theme in these varied approaches to behavioral silicon compilation is that describing the behavior of a system rather than its structure is a powerful way to design systems. The structural details of the design are abstracted away to allow the designer to concentrate on the function of the system. Complex systems can be described succinctly

with the implementation details relegated to the silicon compiler.

2.2 Asynchronous Circuits

One broad classification of digital systems is based on whether the system uses a global synchronizing clock. Circuits whose operations are synchronized to such a clock are known as *synchronous* and those whose operations are not subject to a periodic clock signal are known as *asynchronous*. Although most modern commercial digital systems are synchronous, asynchronous systems have always been of interest to researchers. Early work in asynchronous circuits by Huffman [40], Muller [66, 65] and Unger [102] was concerned with state machine synthesis. Using these formalisms state machines are implemented as a combinational network with delayed feedback rather than a clock signal. More recent work by Hayes [32], and by Molnar [64] has continued to focus on implementing asynchronous state machines in this way.

The complexity involved in implementing these purely asynchronous state machines is such that the techniques are applicable only for very small examples or that they lead to inefficient implementations. Perhaps the largest self-timed system built to date was Davis' Data Driven Machine [21] which used a collection of this type of state machines. Asynchronous state machine implementations using a slightly different approach based on a "one-hot" state encoding have been explored by Hollaar [38], and by Carter [15].

Asynchronous systems can also be specified using Petri nets [73]. Patil [72] has suggested an asynchronous logic array as a systematic way of directly implementing Petri nets, although the implementation is not delay-insensitive. Molnar [64] uses a form of Petri net called an Interface State Graph (ISG) to specify the behavior of control circuits but his synthesis method does not try to implement the ISG directly. Chu [17] has continued this graph-based approach to specifying asynchronous circuit in his work.

Macromodules, developed at Washington University around 1970 [19, 70], demonstrated the usefulness of flexible system building blocks that use distributed asynchronous control. Modules from a set of around 15 different types were used to design and implement parallel computing structures of diverse kinds. Our target language of data and control modules is similar to the macromodule approach in many ways. A small number of other cell sets for asynchronous design similar to the macromodules approach have also been explored [18, 74]. Speed-independent modules on a smaller scale than macromodules were proposed by Keller [45] and, more recently, by Sutherland and Sproull [94, 95].

2.2.1 Silicon Compilers for Asynchronous Circuits

Recently there has been interest in using concurrent programs as specifications for asynchronous circuits. Martin and his colleagues at Caltech [13, 57, 12] have had good success

compiling a language based on CSP [37] into gate-level circuits. Initial programs are first decomposed into many smaller processes initiated by signals on new communication channels. These simpler processes are expanded to include details of the four-phase handshake used for control signals and then mapped into a set of production rules that define when the handshake signals are set and reset. These productions are strengthened to enforce sequencing, and then mapped to a library of gates to construct the circuit. A variety of circuits ranging from simple first-in first-out (FIFO) buffers to a small 16-bit microprocessor have been compiled and implemented using their techniques [58, 55, 54].

A group at Philips Labs [68, 103] has also compiled a similar CSP based language into circuits by translating the initial program into an intermediate representation that includes details of the four-phase control signals, and then implementing the circuit with a library of simple circuits. A group at Berkeley [63, 62] has used a description language based on Chu's graph-based description method to automatically compile specifications into asynchronous circuits in the specialized domain of signal processing.

2.2.2 Verification of Asynchronous Circuits

In addition to research in the synthesis of asynchronous circuits, there has been some work done in the verification of the circuits that are implemented. One important issue is the exact meaning of verification itself. In general, it is only meaningful to consider correctness relative to some specification of the system. Another issue is to consider what types of properties can be expressed in the specification and verified in the circuit. In concurrent program verification, a distinction is usually made between safety properties and liveness properties. A safety property asserts that nothing bad will happen when the system operates. A liveness property asserts that something good will happen when the system operates. For example, a constraint that no more than a single button will be pressed at any time is an example of a safety property. A constraint that once a button is pressed, a light will eventually go on is an example of a liveness property. Safety properties are generally easier to prove than liveness.

Trace theory, first proposed by Hoare [36], and later adopted for speed-independent VLSI by Rem, Snepsheut, and Udding [76, 104, 100] has been used to reason about the correctness of circuits in terms of safety properties. Traces are sequences of events at the interfaces of modules and are used to characterize the behavior of that module. Ebergen [26, 25] has developed formal models for CSP-like languages based on traces that are used to describe individual modules that can be combined to make larger circuits. Dill [24, 23] has adapted previous results using trace theory to implement automatic hierarchical verification of safety properties of speed-independent circuits and extended his model to deal with liveness properties as well. Work on the verification of asynchronous circuits allows the semantics-preserving properties of a translation system to be checked.