SILICON COMPILATION

Daniel D. Gajski
Department of Information and Computer Science
University of California
Irvine, California 92717

and

Nikil D. Dutt and Barry M. Pangrle
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Technical Report 87-07
March 1987

ABSTRACT

Silicon compilation is a term used for many different purposes. In this paper we define silicon compilation as a mapping from some higher level description into layout. We define the basic issues in structural and behavioral silicon compilation and some possible solutions to those issues. Finally, we define the concept of an intelligent silicon compiler in which the compiler evaluates the quality of the generated design and attempts to improve it if it is not satisfactory.

1. INTRODUCTION

The term silicon compilation was introduced by Dave Johannsen at Caltech [Joha79], where he used it to describe the concept of assembling parameterized layout modules. This term is now quite popular, and is used in a variety of different contexts throughout the IC CAD community.

In a narrow sense, silicon compilation is an extension of the standard cell approach where standard cells are replaced by parameterized cell compilers. These cell compilers allow users to customize the cell functional, electrical, and layout parameters. In the case of simple cells, such as NAND and NOR gates, the user specifies the number of inputs, chooses among several drive buffers, and selects the position of I/O ports around the boundary of the cell. More recently, compilers for microarchitectural components such as ROMs, RAMs, PLAs, and ALUs were added to the basic SSI set. Since the number of options increases with the complexity, special forms or menus are provided for the specification of compiler parameters.

In a much broader sense, silicon compilation is a translation process from a *higher-level description* into layout. Here, a *higher-level description* defines a description that hides details from the user, and is not just a textual equivalent of the layout. This translation process is broken down into several steps, with each step considered as a compiler for that level. Using this hierarchy, one can define a microarchitecture compiler that translates a higher level description into a set of registers, buses, and ALUs; or a silicon compiler that translates a structural description into layout.

The silicon compilation process is represented by the tripartite representation of design (Y-chart) shown in Figures 1(a) and 1(b) [GaKu83]. The three axes represent three different domains of description: behavioral, structural, and geometrical. Multiple levels of abstraction (or detail) are represented along each axis, and the information level becomes more abstract as one moves away from the vertex.

In the behavioral domain, the most important aspect of a design is its function and not its implementation. In this domain, the design is treated as a "black box" with a specified set of inputs, a set of outputs, and a set of functions describing the behavior of each output as a function of the inputs and time. For example, the boolean expression "x = a'b + ab' after 30 ns" (Figure 2(a)) gives only the functional specification of a cell whose output (x) becomes logical one 30ns after its inputs (a and b) become different logical values, and becomes logical zero 30ns after its inputs become the same logical value. This expression does not imply

anything about the implementation or the structure of the cell. In the above expression, the timing is modeled as a delay expressed in nanoseconds. At the more abstract levels of description, such as finite state machines and register transfer descriptions, time is modeled by the concept of state. At even higher levels of description, such as algorithmic descriptions, the concept of state is replaced by the concept of statement sequence. Statement sequences only prescribe the order of statement execution.

Several levels of abstraction are identifiable in the behavioral domain [WaTh85]. For example, differential equations are used at the circuit level and boolean expressions at the logic level. At the microarchitecture level, register transfer (or finite state machine) descriptions are used that specify the conditions to be tested, all register transfers to be executed, and the next control state for each state. An algorithmic description defines all the data structures and a sequence of transformations that manipulate them. At an algorithmic level, variables (or data structures) are left unbound to registers or memories, and operations are also unbound to functional units or control states.

The structural representation bridges the gap between the behavioral and geometric representations. It is a one–to–many mapping of a behavioral representation onto a set of components and connections under constraints like cost, time, and area. If, for example, the boolean expression from Figure 2(a) is mapped onto a set of components consisting of only two–input NAND gates with a maximum delay of 10 ns, then one of the structural representations will consist of four NAND gates as shown in Figure 2(b). The structural representation does not specify any physical parameters; such as the position of the four NAND gates on a printed circuit board or on a silicon chip.

Sometimes a structural representation, like a logic or circuit schematic, also serves as a behavioral description. For example, the behavioral description obtained from the structure in Figure 2(b) is x = ((a(ab)')'(b(ab)')')'. Such a derived behavioral description is conveniently used for design simulation and timing verification, but makes redesigning with new design rules almost impossible.

The most commonly used levels of structural representation are associated with the basic structural elements used. At the circuit level, the basic elements are transistors, resistors, and capacitors; while gates and flip–flops are at the logic level. ALUs, registers, RAMs, and ROMs are used to represent register–transfer as well as algorithmic structures. However, at the algorithmic level, microarchitectural components are grouped into datapaths, control units, and data storage. Also, more emphasis is placed on component synchronization and communication than on component implementation. Processors, memories, and switches are used at the system level.

The geometrical representation ignores, as much as possible, the function of the design and binds the structure in space (physical design) or to silicon (geometrical design). For example, if a gate array consists of two–input NAND gates arranged in cells of six gates each, then a possible binding of the structure from Figure 2(b) is shown in Figure 2(c). Each NAND gate and connection in the structure is assigned a physical location. The structure–to–geometry mapping is defined as a two step process. The first step, usually called symbolic or topological layout, determines relative or approximate positions for all structural elements. The absolute positions are determined in the second step after the substitution of layouts for symbols and layout compaction. Although symbolic layout can be viewed as an

independent design representation, it is included in the geometric representation in order to simplify the representation space. The most commonly used levels in geometric representation are: mask geometries, cell placement, and floor–planning with arbitrary size blocks. Note again that the floor–planning level may cover several distinguished levels in two other representational domains. At the system level, there is physical partitioning and placement on the board and cabinet levels; while on the geometrical level there is placement and floor–planning of blocks.

Finally, design is an iterative process where tradeoffs are performed in design cycles to meet the desired constraints (time, area, power, etc.). While humans normally perform this task, it is desirable to automate some of this decision making process. This is briefly described in a later section on future trends.

2. SILICON COMPILATION

The Y chart is used to represent different VLSI–design methodologies and to pictorially explain the differences between silicon–compiler based design systems.

Traditional methodologies require a designer to build a structure and define its function with basic components, such as gates; these are then used hierarchically to build higher level structures. Once the design is finished, it is flattened into a structure of basic components for simulation, placement, and routing (Figure 1(a)). This methodology doesn't efficiently exploit the hierarchical nature of the design, because the simulation, placement, and routing are performed at the lowest level of abstraction (where they are the most expensive to perform). Furthermore, the fixed functionality, as well as fixed electrical and layout properties, of the basic components leads to an inefficient layout.

The silicon compilation methodology tries to overcome these deficiencies by providing basic components that are fine–tuned to the user's specification at a higher level of abstraction, such as ALUs, PLAs, RAMs, datapaths, controllers, and core microcomputers. Furthermore, the methodology provides more abstract models for simulation, placement, and routing.

Silicon compilers are programs that generate layout descriptions together with higher–level models of that description such as functional, logic, timing, power and testability models to be used by verification, analysis and optimization tools such as functional and logic simulators, timing and power analyzers, and layout compactors. The functional, electrical, and layout requirements for each basic component are passed as parameters to the corresponding silicon compiler. A hierarchical methodology is supported by allowing silicon compilers to call other silicon compilers.

As with gate–array and standard–cell methodologies, silicon–compilation requires two experts: The tool maker and the tool user. A silicon–compiler writer (the tool maker) creates compilers for leaf cells and then uses these leaf cell compilers to construct module compilers, processor compilers, and other higher level compilers. These compilers are parametrized, with parameters for different design components stored in a table or a menu. Such a menu serves the role of a behavioral description for a design component. The task of a silicon–compiler writer is represented by the outward going spiral in Figure 1(b).

A system or application designer (the tool user) specifies the design using a behavioral or structural description. In the former case, five different tasks must be performed before the design is ready for fabrication:

1) The behavioral description is translated into a structural description *(synthesis)*.

2) The layout of each structural component is instantiated by a module compiler *(module compilation)*.

3) All structural components are placed on silicon and routed *(physical design)*.

4) The packaging is selected.

5) A test vector set is generated.

In the latter case, the translation task is not needed since the system designer specifies the design structure.

A typical IC design system based on the silicon–compilation methodology is shown in Figure 3. The design structure is specified by the user or is generated by the synthesizer from the behavioral description. In either case a Menu/Form package is used to capture the behavioral description of each component in the structure. The behavioral description is passed in the form of options or parameters to the corresponding silicon compilers. A technology file contains all of the process relevant design rules used for generating the layout. In an interactive environment, layout or schematic editors are used to alter compiler outputs. However, in this case, the "correct by construction" property of silicon compilers is lost. Similarly, timing, behavioral, or logic models are generated for each component in the structure. These models are linked together and passed to a timing analyzer or a simulator. Geometric models are linked together with placement and routing tools to form a chip composite. For interactive placement and routing a composition editor or a silicon assembler may be used [Trim84], and a package editor is used to provide packaging information to the foundry. Similarly, the simulator provides a test set for testing the assembled IC.

3. SILICON COMPILATION ISSUES

Commercial and research silicon compilers are classifiable according to several attributes.

3.1. User Expertise

Figure 1(b) shows that the field is roughly divided into compiler writers that design silicon compilers and system designers that use silicon compilers to design application specific integrated circuits (ASICs). Silicon compiler writers use IC languages to capture and parametrize layout, and other embedded languages to write simulation, timing, and power models for each silicon compiler. System designers specify systems as a structure of components, where each component is: (a) a predefined library component, (b) instantiated by a silicon compiler, or (c) defined from scratch by the designer. System designers are often associated with system companies or divisions that do not have their own fabrication facilities,

while silicon compiler writers are associated mostly with semiconductor manufacturers that provide custom design and foundry services.

This division in the market place was initially reflected in commercial silicon compilation systems from Silicon Design Labs (SDL), Silicon Compilers Inc. (SCI) and Seattle Silicon Technology (SST). Today, silicon compilation based systems offer both compilers for system design, and tools for developing custom compilers.

This crude division can be generalized by considering the design process as a generator of the final design description on one of the design levels using a design language such as the L–language [Buri85] for IC description or VHDL [Shah86] for logic description. In the future we will see compilers on different levels of design using different languages with the obvious tradeoff between control over the design and the designer's expertise.

3.2. Integrability

Silicon compiler systems come in two varieties:

1) They may be complete systems with their own set of proprietary tools such as simulators, timing verifiers, and routers for complete IC design.

2) They may be integrated into a standard CAE work station with most of the support tools provided by the host work station environment.

The advantage of the first approach is the enforcement of consistent design practices, a more efficient database, and better interfaces between different tools. The second approach offers a smooth transition from existing design methodologies, and the possibility of re–using already designed parts.

3.3. User Interface

The user interface specifies ways in which the designer may interact with the system during the design phase, in addition to specifying the initial input description. Some systems allow layout or schematic editing after it is generated by a silicon compiler. A layout editor is necessary during the compiler writing phase, since a cell layout is first created manually, then possibly converted into textual form and parametrized. Some systems allow preplacement and prerouting during the design composition.

Design documentation provides an auditing mechanism for the designer to keep track of the current state of the design. This includes simulation and timing analysis results at various levels, as well as several logic, schematic, and layout "views" that the silicon compiler systems produce. Some silicon compiler systems provide browsers that facilitate extensive user monitoring of the design. Systems like the one offered by SDA [LaMo85] provide frameworks for generating regular and semiregular structures.

3.4. Richness

Richness is characterized by the number and types of module generators available in the system. Module types are divided into basic modules and sequential modules. Basic modules, such as SSI logic, registers, ALUs, PLAs, RAMs, ROMs, and datapaths, take one control state for execution. Sequential modules, such as counters, finite state machines, controllers, and processors, require more than one control state for execution. Some companies offer several different types of controllers, such as Interrupt, Bus, DMA, and CRT controllers. Finite–state machine generators and other popular modules like the ones from the AMD 2901 family or patterns based on popular microprocessors are also available. As an example, a silicon compiler for a core microcomputer which is instruction set compatible with Intel's 8085 is offered by Silicon Design Labs.

3.5. Process Independence

Silicon compilers are not technology independent, that is, a silicon compiler written for a CMOS process can not easily be derived from one written for a Bipolar process. However, a compiler can be made process independent by specifying design rules in a separate technology file and specifying the layout in terms of symbols and constraints between those symbols. During the design instantiation, the symbols are replaced by their geometric representation and the layout is compacted according to the constraint values in the technology file. This method allows the compiler to cover a broad spectrum of processes, but its layout density depends on the sophistication of the compactor. The other strategy is to specify layout with generic lambda rules in which all constraints are a multiple of the basic lambda unit. The process independence is achieved by changing the value of lambda. The second method is simpler but it may produce less compact layouts at the end.

3.6. Quality Measures

Each design system based on silicon compilers can be evaluated through three quality measures:

1) Transistor density in sq. mills/transistor.

2) Compilation speed in transistors/hour.

3) Design time in transistors/person–hour.

The above quality measures depend heavily on the complexity and the regularity of the design. Until standard benchmarks are established, these measures are just indicative of silicon compilation capabilities and they should not be used to compare different systems. Recently, SST reported densities of .36 sq mills/transistors on a DSP chip slightly better than the standard 80386 microprocessor. This demonstrates the viability of silicon compilation as a design methodology.

In addition to design functionality, the transistor density also depends on the fabrication process. The density measure is similar to the computer performance measure of instructions/second which is not very accurate, but is a very popular approximation of performance. A better density measure uses the design functionality as a normalizing factor

instead of the transistor count, since the existence of each transistor may be difficult to justify.

The compilation speed is defined as the compiler run time without plotting, and should be normalized to the same machine. Reported compilation speeds vary from 5K - 25K transistors/hour for different chips and systems.

The design time depends heavily on the design complexity as well as the abstraction level of the compiler. Typically, design times range from 10–30 transistors/hour for various designs. Better design times (300–400 transistors/hour) are achieved by using high level compilers such as core compilers. These results indicate obvious tradeoffs between the design time, the compiler complexity, and the abstraction level of the behavioral specification.

3.7. Support tools

Tools for silicon compilation are divided into three groups: verification, analysis, and optimization tools.

The verification tools are used to verify the correctness of the input behavioral or structural description. Verification is accomplished by the designer specifying a set of input–output pairs and observing for each pair whether the description produces the correct corresponding output for each input. It must be noted that verification only proves that the design works for the given test set.

The most frequently used tools are behavioral, functional, logic, circuit, and fault simulators. Behavioral simulators are easy to write, particularly if the behavioral description language is embedded in another programming language. Functional and logic simulators are used in structural silicon compilation, where each module compiler generates a functional or logic module that is linked together with other modules as specified in the input structural description. A fault simulator allows a user to specify a set of test vectors, and determines the fault coverage of the given test set under an assumed fault model. Usually, a single stuck–at fault model is used. Circuit simulation is used mostly on leaf cells or when a foreign module is imported at the circuit level.

Analysis tools are used to determine the quality or "goodness" of the generated design. Very frequently, a timing analyzer is used that determines the delay from any input to any output, and the delay between elements. The maximum delay between storage elements determines the clock period. Thus, a timing analyzer can be used to predict the performance of a design. Similarly the testability analysis tools calculate the controllability and observability figures, that is, the relative measures of difficulty in controlling and observing signal values. Together, those figures define the testability of a design.

Optimization tools improve the quality of the design without performing any tradeoffs. Quality optimization tools make translating behavior to structure, or structure to geometry easier, since the translators do not have to produce an optimal design, but only a correct design. The most frequently used optimization tools at the layout level are programs for layout compaction and transistor sizing. At the topological level, many PLA folders minimize PLA area by sharing rows and columns between two or more input and output lines. At the logic level, synthesis programs, such as SOCRATES [GBGH86] and the Yorktown Silicon

Compiler [BBCD85], are capable of optimizing logic for a given input–output delay. Similarly, some behavioral level compilers are also capable of optimization for a given time delay. However, optimization at this level is a part of the synthesis and not a separate tool.

Presently, there are no existing commercial or research compilers that incorporate all these tools. Most compilers include a simulator for checking the input description, and a timing analyzer. Optimization tools are seldom available.

4. BEHAVIORAL SILICON COMPILATION

The following important issues are used to characterize behavioral silicon compilation: 1) architectural model, 2) input language, 3) bindings, 4) timing model, and 5) physical model. This framework is used to examine how some existing behavioral silicon compilers function. The following examples are meant to comprise a representative, but not an exhaustive, collection of the behavioral silicon compilers currently in existence in the commercial and research communities.

4.1. Architectural Model

Silicon compilers are generally designed for a class of applications using a specific architectural model. This architectural model is sometimes referred to as the target architecture. The architectural model describes the type and parameters of the hardware realizable from the input specification and can be very narrow, as in bit–serial silicon compilation, or very broad, as in a set of communicating finite state processes. Most silicon compilers fall into one of three architectural models: 1) Datapath compilers, 2) Control unit compilers, and 3) Datapath and control compilers. Within each model, further identification of the implementation style is made based upon the model's features.

4.1.1. Datapath

Compilers using the Datapath model primarily perform computations. These compilers do not perform control functions like data steering, or next state determination. The datapath style is categorized by the types of basic units used (adders, shifters, ALU's), the types of storage used (registers, register banks, memories, etc.), and the connection style (muxes, uni–directional buses, bidirectional buses, etc.). Pipelining may be used in the datapath as in [PaPa 86] to generate a fully pipelined datapath, or in the components that make up the datapath as in [PaGa 86].

Digital signal processing applications often require machines that perform repetitious calculations. These types of machines fit well in the Datapath model, because of the data flow nature of many of the DSP calculations. Several compilers have been developed for digital signal processing applications. FIRST [Berg83] and the GE bit–serial silicon compiler (BSSC) [JNHH85] are pipelined bit–serial datapath compilers. FIRST and BSSC use units like multipliers and adders, and use latches as storage elements for delays. The connection style is point–to–point between datapath elements. Figure 4 shows the organization of a typical digital signal processing application, whose architectural model mimics the depicted data flow

graph.

4.1.2. Finite State Machine Controller

The Finite State Machine (FSM) controller model is suitable for generating the control unit for a machine. The FSM concept consists of two basic elements as shown in Figure 5: the registers that store the state of the machine, and the combinational logic that performs the mapping of inputs and current state to a next state. In addition, the combinational logic component generates control signals for use by a datapath in that particular control state. The implemented FSM may be state–based, where the outputs are dependent on the current state only, or transition–based, where the outputs are a function of both the current state and the inputs.

In applications with little computation and only a few states, an FSM alone may suffice to implement the whole design, by implementing the corresponding datapath inside the FSM's combinational logic. In a more general environment, such as a microprogrammed architectural model, the architecture consists of a separate control unit and a datapath. Depending on the style of the control unit, the mapping logic may be implemented in a regular structure such as a PLA or a ROM, or directly as random logic. Several FSM synthesis systems have been developed that use PLA–based techniques as well as multi–level logic synthesis techniques [Sang85].

The Berkeley FSM synthesis system [RuSD85] has a transition–based model and synthesizes control units for custom–built datapaths. The implementation is in the form of a PLA targeted for CMOS technology. The design is synchronous, using a fixed, 2–phase clocking scheme.

The FSM synthesizer developed at AT&T Bell Labs [TPLM86] allows the user to describe either a state–based or a transition–based FSM. The synthesized FSM may be implemented in a variety of styles, including polycells, PLAs, and PALs.

4.1.3. Control Unit + Datapath

The control unit and datapath (CU+DP) model uses a datapath which performs the computations and a control unit that captures state information and produces control signals for the datapath. Communication between the control unit and the datapath is synchronous. Operations performed in the datapath in one state produce inputs to the control unit for determining the next state. Since the control unit is basically an FSM, the style of its implementation determines whether the inputs to the control unit are directly obtained from the datapath (transition–based) or are encoded into the current state of the machine (state–based). Pipelining can be implemented in the datapath and/or the control unit. Pipelining in the control can be as simple as the overlapping of instruction fetches and executions or it may be more complex.

Inherent parallelism in a design problem may be exploited by using multiple datapaths to perform computations on independent streams of data. Each datapath may be controlled by a separate control unit thus creating a group of single CU+DP processors. In this case, the architectural model is based on a structure of these processors running in parallel. Different

control, communication, and clocking strategies may be used within each processor, and between processors.

In MacPitts [Sout83] a processor is implemented as a microprogrammable datapath with a fixed clocking scheme. The datapath consists of registers and function units like adders and comparators. Figures 6(a) and 6(b) show the MacPitts architectural and datapath model.

The CMUDA system [THKR83] architectural model assumes a microprogrammable datapath controlled by a centralized transition–based FSM. Communication between processors is achieved synchronously by means of a fixed two phase clocking scheme. The datapath may be pipelined, but the control unit has limited pipelining that is determined by one of the three implementation styles described in [NaCP82]. The datapath consists of units like ALU's, comparators, etc. The storage elements consist of registers and memories, and the connection style is distributed with a limited number of buses. The ATT DAA [KGWF85] also has a similar architectural model.

Other systems that fit into the CU+DP architectural model include Silc [BlFR85], CADDY [Camp85], ELF [GiKN84], SYCO [Jerr86], YASC [JhSK85], and MIMOLA [Marw86].

The CU+DP model can be generalized to create structures of communicating processors, where each processor may be based on any one of the preceding basic architectural models.

4.2. Input Language

The input to a behavioral silicon compiler is a behavioral description that specifies the result(s) of a computation or a set of computations, but not the implementation. Every compiler uses a predefined style of implementation referred to as the architectural model. A compiler, therefore, must map or *bind* language variables, operations, and constructs to components within the architectural model. Usually this mapping is one–to–many, since there are several different ways of implementing each operation or construct. This implies that the compiler must make design tradeoffs that best satisfy the design goals. If the compiler does not make these design decisions, then the language must provide constructs that allow the user to make them. For example, if the architectural model consists of several communicating DP+CU processes, the language must have special constructs that allows the designer to partition the design into a set of communicating processes. If the compiler is capable of doing this automatically, a partitioner is used that automatically partitions the input description (perhaps optimally) into a set of processes.

Hence, the expressivity of a language reflects the generality of the architectural model used by the compiler, while language bindings specify how some constructs are mapped into the architectural model. This section discusses two basic issues relating to an input language: features that enhance the expressivity of the language, and bindings that the user may perform in the language. The output of the behavioral compiler is a structural design at the register transfer level which exhibits the desired behavior within the architectural model of the compiler.

4.2.1. Order of Execution

An input language with purely sequential constructs requires the compiler to extract parallelism from the input description. For example, the CMUDA system uses ISPS as its input description language. The ISPS description [Barb81] is compiled into a Value Trace (VT) representation [Snow78] (an intermediate graph form), from which parallelism is extracted.

Some input languages allow special constructs for expressing parallelism. Figure 7(a) shows the 'ser' and 'par' constructs in Silc used to swap values in variables a and b. Figure 7(b) shows how the same thing can be described in the SYCO compiler input: each statement is executed serially, and the user describes parallelism by including more operations in one statement. If the input description explicitly binds operations to states, no further parallelism is extracted. However, some compilers [JhSK85] [GiKn84] allow the specification of parallelism in the language using special constructs; more parallelism may be extracted at compile time during the state binding phase.

As an example, in the MacPitts system the user specifies the variables as being either registers or internal ports in the *def* section of the code. Variables declared as *registers* are assigned to registers and mark state (cycle) boundaries. Variables declared as *internal ports* represent data transfers between sequential operations performed in the same state. Given a sequence of statements where each variable is declared as an internal port, this sequence of statements is implemented combinationally and executes in only one state. However, if the variables are defined as registers, this computation chain is implemented as a series of states that must be executed sequentially to perform the computation.

4.2.2. Data Types

Data types identify the types (integer, boolean, etc.) of data in the input language. Data types influence the compilation process in two ways. First, it enables detection and possible correction of architectural model violations like combining datapaths of differing bit widths or differing types. Second, data types help guide the compilation process in specific ways. For example, in Silc, integer operations are mapped into the datapath, while boolean operations are mapped into the logic synthesis subsystem for control generation.

4.2.3. Macros

A macro may be viewed simply as an operation that is expanded into smaller, divisible operations. Macro–expansions serve two purposes. First, they provide a short–hand notation for the user to define pseudo–operations in the language. Second, they allow annotation of the input description to guide the compilation process in a specific direction. Annotating the input refers to the case where the user has already predetermined a specific structural module that is to be bound to the macro operator. Hence, these operators are deliberately pre–allocated by the user. The ELF system allows the definition of new hardware operators by means of the macro facility in ADA.

4.2.4. Subroutine Calls

Subroutines permit calling frequently used segments of a description, without having to define the segments more than once. The subroutine calls and returns are usually monitored in a stack which resides in the control unit.

In some systems, the input specification directly affects the time and area overhead for the compiled design. For example in SYCO, the control structure for a subroutine call gets compiled into a control slice at that level of the input specification's hierarchy. A deep nesting of calls results in a large communication overhead between control slices, but results in a savings in area. Conversely, flattening out the input description by using a shallow nesting of procedure calls increases the area but reduces the execution time.

4.2.5. Graph Representations

Before discussing how various bindings are performed by a compiler, it is instructive to examine the underlying graph representation (such as the CMUDA Value Trace) of the input descriptions. In general, the behavioral graph representation consists of nodes that correspond to operations in the input language and edges that correspond to the flow of data between the operations. The operation order specified in the input implies data and control dependencies on the graph. Data dependencies enforce a correct order of execution by allowing an operation to be scheduled only if its inputs are ready. Control dependencies arise from language constructs that involve conditional evaluation such as if, case, and loop constructs. In addition, the input description may have timing constraints specified between operations in the input or between labels in the input specification. These timing constraints are often represented as nodes that exist between the corresponding operations or edges in the graph.

As an example, Figure 8 shows a segment of code that is compiled into into a flowgraph in CADDY. The circles represent operations in the input description and the squares represent variables, constants, and intermediate values. The solid and dotted edges represent data and control dependencies respectively. The timing constraint in the input has been compiled into a control edge between the "CASE" node and the "OF" node. In this representation, control constructs such as the CASE statement above have been mapped into control flow, where each of the mutually exclusive branches "START = 0" and "START = 1" can proceed independently.

It is also possible to map these constructs into data flow, where each branch is simultaneously performed, and the result is chosen at the end of the CASE statement. Figure 9 shows how a similar segment of code (without the timing constraint) is compiled from an ISPS description into the Value Trace (VT) representation. Here the value of "A" is assigned to "A + APORT" if "START = 1", and is assigned to 0 if "START = 0", *after* both branches have been executed. Compiling conditional constructs into data flow usually results in a faster design at the expense of additional hardware (because the exclusive branches are executed simultaneously). ELF uses a control/data flow graph representation that is similar to CADDY's where control constructs are represented explicitly, while a flexible control/data flow graph is described in [OrGa86].

The amount of parallelism in the design is determined by the way in which bindings of operations (graph nodes) to states, and function units to operations are performed. Several

concepts are of relevance here.

Sequential operations must be executed in series. The sequential ordering may be enforced by control or data dependencies in the input description. If the clock cycle length is small, then sequential operations may be forced to execute in different states as shown in Figure 10(a). *Chaining* of operations involves using the output of one operation as the input to another in the same state. If the clock cycle length is long enough, it is possible to chain two sequential operations in the same state as shown in Figure 10(b). *Multicycle* operations are those that take more than one clock cycle to execute (Figure 10(c)). *Parallel operations have no direct dependencies between each other* and may execute independently in the same state or across several states as depicted in Figure 10(d).

4.3. Bindings

A behavioral compiler transforms the input description into the final design using a process of successive refinements and optimizations. This process *binds* structural and geometrical detail to the design within the architectural model of the silicon compiler. In all, the language to structure bindings may be classified into four groups:

(1) Operations to States,
(2) Operations to Units,
(3) Variables to Registers, and
(4) Connectivity.

State bindings represent the allocation of operations to specific control steps (or states) of the abstract machine. The amount of parallelism used in the machine is determined during state binding. Parallelism in this context means performing more operations (by using more structural components) per state for a given cycle length. The parallelism in the design is therefore limited by the number of available structural components and the state length (clock cycle period). The task of an operation–to–state binder, given a state length, is to minimize the number of states by maximizing structural component utilization.

Once an operation is assigned to a state, a structural component (eg. an ALU) that performs the required operation (eg. an add operation) must be assigned to that operation for that state. The bindings of operations to function units is typically a many to few assignment respectively. This is because multi–function components such as ALUs may be used for a different operation type during each state. Units can be used in many states but are allowed allocation only once per state. Hence, the number of units affects state bindings.

Data that is generated in one state and is used in another must be saved in registers. Register binding involves the allocation of registers to data carried across state transitions. Some systems have a one to one correspondence of the input language variables to the registers, while others optimize registers by sharing. Register sharing allows a register to store two or more variables that have disjoint lifetimes. By sharing variables in registers, it is possible to use fewer registers than if each variable is assigned to its own unique register.

Connectivity binding represents the allocation of connections between hardware components to create the necessary information paths. Connections are either point–to–point (muxes at each function unit and register input) or bussed (muxes at both the inputs and outputs of function units and registers). The bus model allows greater sharing of interconnection

by creating a larger number of paths for the same number of connections. This is equivalent to creating a switching network with one more level of switch boxes (muxes in this case) as compared to the point–to–point model.

The silicon compiler writer (Figure 1(b)) is free to make design tradeoff decisions while writing the compiler, and these tradeoffs are reflected in the architectural style, the physical model, and the timing model of the compiler. Some compilers allow the user to specify some design details in the input language (explicit binding), and the compiler binds the rest. *Advantages* of explicit bindings are that the user has tighter control over the design process and that it makes the compiler simpler. *Disadvantages* are that the compiler is restricted from performing optimizations and tradeoffs on the user's explicit bindings, and that the user has to have a detailed understanding of the design rules, and the architectural, physical, and timing models. If the compiler is restricted from performing tradeoffs, the user must rewrite the description in order to explore another design in the search space.

State, unit, register, and connection binding strategies vary from system to system. Different systems combine these binding problems together in varying degrees. The task of transforming a behavioral specification to an abstract architecture (eg. register transfer level) has been studied independently of silicon compilation for several years [Thom 77] [Snow 78]. Most systems take an algorithmic approach to these bindings. Due to the complexity of these binding problems, the algorithms are typically heuristic in nature and no system guarantees that its designs are optimal. As an example of this complexity, the problem of allocating language operators to datapath components can be transformed into a scheduling problem which is NP–complete [LDSM80].

The Elf system uses an urgency scheduling algorithm to perform operation–to–state bindings. Operation nodes in the graph are weighted and compared to timing constraints to determine their order of binding. Node weights are calculated by taking the minimum number of cycles to execute the node plus the maximum weight of the node's successors. Node urgencies are determined by taking a ratio of the node's weight and the number of cycles left until its time constraint. Nodes are allocated to states based on available components and their urgency. If a node (operation) is delayed it will approach a timing specification line and its urgency will increase. This raises the probability of the operation being allocated. If the timing specification cannot be met the user is notified.

The HAL [PaKn86] system uses a load balancing approach to conduct operation to state bindings. Load balancing attempts to distribute operations across states in such a way as to evenly distribute like operations while taking into account the speed requirements of the final design. This should help minimize the number of components necessary to perform any one type of operation. The operations are then grouped together to form units. Another pass is made to minimize the concurrency of operations using similar resources. At this stage the operations are bound to states and the remaining bindings (operation to unit binding, register binding, and connectivity binding) are performed in the last stage. The assignment (or binding) of operations to states is based on structural component locality in an attempt to minimize the length of interconnect lines.

EMUCS [HiTh83] performs operation–to–unit, register, and connectivity binding by creating cost tables and using a min–max criterion. Cost calculations are based on considerations such as the addition of another multiplexer or adding another connection. During each

iteration, the cost for binding each unbound graph element is computed. Next, the difference between the two least costly bindings is computed. The element with the largest difference in the two least costly bindings is bound using the lowest cost. EMUCS then iterates until all of the graph elements are bound.

Figure 11 shows an example of one iteration in the EMUCS algorithm. The graph has been sliced into two control steps and all of the registers have been bound. Under the graph is a cost table showing the cost of binding ALUs to operations in the graph. ALU_1 has been bound to operation_2 and so the row for operation_2 in the cost table is filled with X's. Since ALU_1 is bound to operation_2 in the first control step there is an X in the first column of the first row because an ALU may be used only once during a control step. All of the other ALUs are assumed to be unbound to any other hardware.

The costs are calculated by adding 10 units for each added connection and 5 units for each added multiplexer. The cost for binding ALU_2 to operation_3 is calculated by adding 10 units to connect each input, 5 units to add a multiplexer in front of register 3 (because ALU_1 already outputs to register 3), and another 10 units to connect ALU_2 to the multiplexer for a total of 35 units. If ALU_1 is bound to operation_3 all of the necessary connection exist and the cost is 0 units. The other costs are calculated in a similar manner.

A decision on which operation to bind is made by comparing the two least costly bindings for each operation (min-max criterion). The operation with the maximum difference between the two least costly bindings is chosen for binding using its least costly binding solution. In this example ALU_1 is bound to operation_3 during this iteration. In the next iteration the cost table is updated and the procedure repeated until all of the bindings have been made.

DAA [Kowa84] uses a two pass method based on a knowledge based expert system to perform bindings. The first pass uses rules to create and connect elementary components together and stores this information in an intermediate structure and control specification language. The second pass uses rules in a *clean-up* phase to merge components into more complex elements that yield the final design.

Table 1 summarizes some of the features of some current compilers with respect to their architectural model, the input language, and their bindings. Typically, these compilers use a standard set of available function unit types that are eventually mapped into a target architecture. The Yorktown Silicon Compiler (YSC) [BBCD85] uses an input description that is at a lower level (logical vs behavioral) of abstraction than the other compilers. A high-level front end for the YSC is currently under development.

4.4. Timing Model

The following aspects of the design process may be used to characterize the timing model: clocking, delay specification and communication protocol.

4.4.1. Clocking

Most compilers assume a synchronous model for the design. The clock may be mono-phase, 2-phase, or n-phase. Compilers like MacPitts and Silc assume a 2-phase clock with distinct read and write phases. This allows for straightforward compilation of parallel code in the language since read/write conflicts are avoided: all variables on the right-hand side of an assignment are read during phase 1 of the clock, while all variables on the left-hand side (stores) are written to in phase 2 of the clock. Thus the registers have master-slave semantics, allowing for both a read and a write to the same register in the same clock.

4.4.2. Delay Specification

Some input languages permit the specification of delay between statements in the behavioral description. The user may want to set maximum and minimum limits on these delay figures. If the delay spans a set of non-sequential statements (eg. loops), then the user is also specifying the maximum and minimum execution time for that section of the code. Maximum and minimum delays specified from a module's inputs to its outputs is often referred to as performance specification. The major issues here are how these delays are specified in the language, and how this information is used to guide the synthesis process.

The ELF system uses a subset of ADA [GiBK85] as the input language. Delay specification is supported in the language with the pair of constructs "TIMING.REFERENCE" and "TIMING.CONSTRAINT" to specify a maximum and minimum delay between the reference point and the constraint point in the algorithm. The construct "TIMING.PERIOD" may be applied to a loop, which simply specifies the maximum and minimum loop execution times. The timing constraint is compiled into a timing node that exists between two reference lines corresponding to the two reference points in the input description. Urgency weights are assigned to operations in the flow graph based on these timing constraints and operations are allocated to states based on their urgency weights.

Nestor and Thomas [NeTh86] show how timing constraints may be specified in the ISPS description which is used as input to the CMUDA system. Timing is specified by associating labels with operations in the ISPS description and maximum/minimum timing constraints between pairs of labels. The timing constraints guide scheduling of the Value Trace (VT) flowgraph into time steps using a list scheduling algorithm, which iteratively schedules VT nodes into states using a timing constrained priority function. In each iteration, the priority function indicates if scheduling a VT node in the current state violates a minimum timing constraint, or if postponing scheduling of the VT node to a later state violates a maximum time constraint.

4.4.3. Communication Protocol

If the design is viewed as a set of chips that communicate with each other, and each chip is composed of several processes that run in parallel, then different communication protocols may be used within the same process, between different processes, or between chips.

In synchronous processes, two or more tasks communicate after they have been executed. For example, a process (FSM) in Silc may contain several blocks of code which are to

execute in parallel. The statements inside a 'par' block are executed in one synchronous state, and communication between variables in these statements is achieved at the end of that state.

Within a chip, inter–process communication may be achieved synchronously or asynchronously. Silc allows for processes (FSM's) to run on different clocks. Asynchronous variables global to the communicating processes are declared in Silc's main body. The processes then communicate by handshakes on these global variables using the synchronization functions "data–ready" and "data–received" applied to the variable. As shown in Figure 12, FSM1 waits for "data–ready(shared–reg)" to become true before reading "shared–reg". On reading "shared–reg", the function "data–received" becomes true, and FSM2 may write another word into "shared–reg".

When communication between chips is desired, the user has to explicitly code the protocol in the input specification. For example, Figure 13 shows read and write protocols for an external memory chip in MacPitts. This protocol is synchronous, assuming that both the chips use the same clock. In the read protocol, the user has to ensure that the read signal and address are asserted for three clocks, while the input data is read in on the second clock. Likewise, for the write cycle, the user must specify that the address and data are asserted for three clocks, while the write signal is set high in the second clock.

Digital signal processing applications require different communication protocols based on the timing characteristics of the data to be transferred. The nature of the algorithms resembles a producer–consumer relationship, where the input signal is sampled at predefined constant intervals, and the computation is performed on the samples to produce the output. In Cathedral II [RDVG86], if large unordered blocks of data are transferred at a high sampling rate between processors and their timing is known at compile time, a synchronous protocol is used with a pair of interleaved RAMs as shown in Figure 14. If data is being transferred from processor 1 to processor 2, the protocol requires that during a time frame, processor 1 writes to one RAM while processor 2 reads from the other RAM. In successive time frames, the RAMs are switched. At less demanding data transfer rates, a single RAM with a bus connecting the two processors may suffice. In cases when the timing cannot be determined at compile time, a two way handshake may be employed for synchronization of the data transfer.

4.5. Physical Model

The physical model for a silicon compiler defines the topology of the chips it generates. This physical model varies from a completely prefixed footprint for some systems, to a completely random topology at the other end of the spectrum. In each case, the requirement for placement and routing differs.

Linear topology is based on a bit–slice structure where layout is generated by appending one bit cell slices. As a result, routing is reduced to a one–dimensional problem. Examples of systems that have a linear topology include MacPitts, SYCO, and GE's Bit–Serial Silicon Compiler (BSSC).

The MacPitts physical model consists of a control unit and a datapath as shown in Figure 6(a). The control part is implemented as a Weinberger array, while the datapath is

composed of bit slices of units (called organelles) drawn from a cell library. The horizontal bit slices in the datapath are separated by interconnection channels in which local buses run. MacPitts determines the placement of the units to optimally pack these local buses, attempting to minimize the channel width by making the buses share collinear tracks.

Figure 15(a) shows the physical model for Syco. Syco's control unit is composed of control "slices" which are stacked together, with the lowest level slice being adjacent to the datapath. The number of control slices is determined by the maximum nesting of procedures in the input description. Each control slice is defined by the the set of procedures at that level of hierarchy in the input description. The datapath for Syco (Apollon) is similar to that of MacPitt's except that it is organized around a two–bus scheme. The two buses are segmented into sub–datapaths which run in parallel, as shown in Figure 15(b). Apollon attempts to order the placement of these sub–datapaths based on a heuristic which uses the proximity of the registers in the sub–datapath. The problem of one–dimensional placement and routing is transformed into two subproblems: ordering (relative linear placement) of the sub–datapaths; and assignment of registers to the sub–datapaths.

FIRST uses a floor–plan consisting of two rows with a central routing channel. The user specifies blocks in the input language. FIRST sorts these blocks by height, and creates a partitioning into a tall set and a short set, where the division between tall and short is determined by minimizing the resulting area. The tall blocks are placed in the top row while the short blocks are placed in the bottom row. Within each row, blocks appear in the relative order specified in the language. Channel routing is then applied in the central channel. Figure 16 shows the FIRST topology.

LAGER's [PoRB86] chip–level physical model also consists of two rows of blocks, where each block is a bit–parallel processor that operates in parallel with the other processors on the chip. Within each row, the processors are optimally ordered and oriented so as to minimize the routing. Channel routing is used in the central channel between the two rows.

A completely random topology requires two–dimensional placement and routing of the basic cells. A system with a random topology allows for greater layout flexibility, but the compiler has to handle the complex problem of two dimensional placement and routing. Silc uses such a topological model where the blocks are first placed [MaFB84] and 2–dimensional routing in irregular channels is then performed [Cies84]. The Yorktown Silicon Compiler (YSC) goes one step further by allowing global floor–planning with flexible macrocells. YSC's target implementation uses macrocells built out of rows of complex cells. The global floor–planner determines the relative placement and the shape of each macrocell using a simulated annealing algorithm which takes into account critical timing and total wiring. The aspect ratio determined by the floor–planner forces the macrocell generator to tile complex cells in a requisite number of rows to meet the specified aspect ratio.

5. FUTURE TRENDS

The design process model consists of iterating over designs and exploring alternatives. Present day silicon compilers translate an input description into layout in a unique way. A designer working with a behavioral compiler must understand the translation process built

into the compiler and modify the input description so as to force the compiler to produce the desired results. When working with structural compilers, the designer must be able to evaluate a design and be able to choose a different style or a component. An intelligent silicon compiler incorporates knowledge about the design process and uses this knowledge to guide the transformations of the input specification through several design iterations until the specified set of constraints are met.

Intelligent compilation is broken down into four basic tasks: style selection, refinement, optimization, and strategy formulation.

The selection of styles depends on the goals assigned to a particular implementation. In order to automate the refinement, the process of translating overall design constraints into different design styles must be captured [BrGa86] [KnPa86].

Refinement (sometimes called partitioning or decomposition) is the process of translating a behavioral description into a structure of predefined components from the next lower level of design. As we mentioned before this translation is not unique; usually several different styles can be selected.

An optimization step improves utilization of allocated resources such as silicon area in PLA folding or layout compaction, the number of tracks in channel routing, or the number of functional units in a microarchitecture. Optimizations are performed after or together with the refinement. An optimization process is usually defined by an algorithm. Compaction, placement, and routing optimizations are typical, while clock speed / power sizing, and PLA folding are offered by some silicon compilers.

A strategy is a sequence of different style selection, refinement, and optimization steps. The type and the order of the steps in the sequence characterizes the strategy. Strategies are better understood at levels closer to physical design than at the higher, more abstract levels. It is natural that symbolic layout is followed by geometric layout, which is followed by compaction. However, the order of register, unit, and bus allocation, and their optimization on the microarchitecture level is not clear. Presently, all existing silicon compilers allow only one fixed strategy.

The whole design process can be thought of as a set of refinements and optimizations for each style at each level of design.

In order to formulate a proper design strategy, three more mechanisms must be added. The constraint–propagation process partitions constraints assigned to a design module into constraints for each of its components. The designer performs this task when using presently available compilers. The second mechanism deals with evaluating the final design of each component and estimating how well the constraints have been satisfied. The evaluation process is supported by providing time analysis, cost and power reports. For example, Silicon Compilers, Inc. provides clock reports, I/O timing reports, path delay reports, violation reports, and SCOAP reports. The third task deals with performing tradeoffs in case one or more constraints are not satisfied. The tradeoffs are made by choosing different design styles, such as choosing a ripple–carry adder over a carry–look–ahead adder if area goal is more important than the speed goal.

An intelligent silicon compiler model must be supported by two other mechanisms. First, it needs an input language that requires little explicit binding, so as to postpone the decision making process to the system at compile time. Second, it needs an internal representation which is flexible enough to allow tradeoff analysis and transformations based on user constraints. An example is shown in Figure 17, where the contents of two registers are incremented and exchanged. Depending on the constraints at compile time, the compiler may perform the exchange in one state if two adders are available (Figure 17(a)), in two states if only one adder and three buses are available (Figure 17(b)), or in three states if only one adder and two buses are available (Figure 17(c)).

Presently, goal definition, style selection, constraint propagation, design evaluation, and tradeoffs are made by the designer with the help of CAD tools such as simulators and timing verifiers.

6. CONCLUSIONS

Silicon compilation was presented as an evolving methodology that makes custom silicon affordable and removes the design time bottleneck. As this methodology evolves, the level of abstraction will rise from the circuit and logic level to the microarchitectural level. This evolutionary process should blend well with present CAD tools, so more silicon–compilation systems are expected to appear on standard workstations in the future.

Silicon compilers could be standard utility routines on future workstations, with the compiler layouts modifiable by the designer, and integrated with handcrafted custom and semi–custom parts. Furthermore, a designer should be able to add his or her own compilers to the set, create new compilers by using existing compilers, and add his or her own personal cells. Present day layout and schematic capture workstations are not well suited to providing this integrated mixed–mode methodology. Future workstations will require more powerful IC layout and design description languages, along with integrated design data bases supporting easy conversion between hierarchy abstraction levels. With such stations in place we will be able to design intelligent silicon compilers with knowledge–based technology to control the whole design process. Future trends include richer behavioral descriptions, automatic synthesis tools, and AI based tools for complete design–process automation.

7. REFERENCES

[Barb81] M. R. Barbacci, "Instruction Set Processor Specification (ISPS)," *IEEE Transactions on Computers,* vol. c–30, no. 1, January 1981.

[Berg83] N. Bergmann, "A Case Study of the F.I.R.S.T. Silicon Compiler," *Third Caltech Conference on VLSI,* 1983.

[BrGa86] F. Brewer and D. Gajski, "An Expert System Design Paradigm," *The 23nd Design Automation Conf.,* June 1986.

[BlFR85] T. Blackman, J. Fox, C. Rosebrugh, "The SILC Silicon Compiler: Languages and Features," *Proc. 22nd Design Automation Conf.,* June 1985.

[BBCD85] R. Brayton, N. L. Brenner, C. L. Chen, G. DeMicheli, C. T. McMullen, R. H. J. M. Otten, "The Yorktown silicon compiler," *Proc. ISCAS,* June 1985.

[Buri85] M. R. Burich, "Programming Language makes silicon compilation a tailored affair," *Electronic Design,* Dec. 12, 1985.

[Camp85] R. Camposano, "Synthesis Techniques for Digital System Design," *Proc. 22nd Design Automation Conf.,* June, 1985.

[Cies84] M. Ciesielski, "A New Approach to Routing in Irregular Channels for the Silc Silicon Compiler," *Proc. ICCAD,* Nov. 1984.

[GaKu83] D.D. Gajski, R. H. Kuhn, "New VLSI tools," *Computer,* Vol 16., No. 12, Dec. 1983.

[GiBK85] E. F. Girczyk, R. J. Buhr, and J. P. Knight, "Applicability of a Subset of Ada as an Algorithmic Hardware Description Language for Graph–Based Hardware Compilation," *IEEE Trans. on Computer–Aided Design,* Vol. CAD–4, No. 2, April 1985.

[GiKn84] E.F. Gircyzk, and J. P. Knight, "An Ada to Standard Cell Hardware Compiler Based on Graph Grammars and Scheduling," *Proc. ICCD,* Oct. 1984.

[GBGH86] D. Gregory, K Bartlett, A. deGeus, G. Hachtel, "SOCRATES: A system for automatically synthesizing and optimizing combinational logic," *23rd Design Automation Conference,* June, 1986.

[HiTh83] C.Y. Hitchcock and D.E. Thomas, "A Method of Automatic Data Path Synthesis," *Proc. 20th Design Automation Conf.,* June 1983.

[Jerr86] A.A. Jerraya, et al, "Principles of the SYCO Compiler," *The 23rd Design Automation Conf.,* June 1986.

[JhSK85] C. S. Jhon, G. E. Sobelman, D. E. Krekelberg, "Silicon compilation based on a data–flow paradigm," *IEEE Circuits and Devices Magazine,* May 1985.

[JNHH85] J.R. Jassica, S. Noujaim, R. Hartley, and M.J. Hartman, "A Bit Serial Silicon Compiler," *Proc. ICCD,* November 1985.

[Joha79] D. Johannsen, "Bristle blocks: a silicon compiler," *Proc. 16th Design Automation Conf.,* June 1979.

[Kowa84] T. J. Kowalski, "The VLSI Design Automation Assistant: A Knowledge–Based Expert System," *Ph.D. Dissertation, Carnegie Mellon University,* April 1984.

[KGWF85] T. J. Kowalski, D. J. Geiger, W. Wolf, W. Fichtner, "The VLSI design automation assistant: a birth in industry," *Proc. ISCAS,* June 1985.

[KnPa86] D.W. Knapp and A. Parker, "A Design Utility Manager," *The 23nd Design Automation Conf.,* June 1986.

[LaMo85] H–F. S. Law, J. D. Mosby, "An intelligent composition tool for regular and semi–regular VLSI structures," *Proc. ICCAD–85,* Nov. 1985.

[LDSM80] D. Landskov, S. Davidson, B. Shriver and P. Mallet, "Local Microcode Compaction Techniques," *Computing Surveys,* vol 12, No. 3, Sept. 1980.

[MaFB84] L.A. Markov, J.R. Fox and J.H. Blank, "Optimization Techniques for Two–dimensional Placement," *The 21st Design Automation Conf.,* June 1984.

[Marw86] P. Marwedel, "A New Synthesis Algorithm for the MIMOLA Software System," *The 23rd Design Automation Conf.,* June 1986.

[NaCP82] A.W. Nagle, R. Cloutier, and A.C. Parker, "Synthesis of Hardware for the Control of Digital Systems," *IEEE Trans. on Computer–Aided Design of Integrated Circuits and Systems,* Vol. CAD–1, No. 4, October 1982.

[NeTh86] J. A. Nestor and D. E. Thomas, "Behavioral Synthesis with Interfaces," *Proc. ICCAD,* Nov. 1986.

[OrGa86] A. Orailoglu and D. Gajski, "Flow Graph Representation," *The 23nd Design Automation Conf.,* June 1986.

[PaGa86] B.P. Pangrle and D.D. Gajski, "State Synthesis and Connectivity Binding for Microarchitecture Synthesis," *Proc. ICCAD,* Nov. 1986.

[PaKn86] P.G. Paulin and J.P. Knight, "HAL: A Multi–Paradigm Approach to Automatic Data Path Synthesis," *The 23nd Design Automation Conf.,* June 1986.

[PaPa86] N. Park and A. Parker, "Sehwa: A Program for Synthesis of Pipelines," *The 23nd Design Automation Conf.,* June 1986.

[PoRB86] S.P. Pope, L.M. Rabaey and R.W. Broderson, "Automatic Generation of Digital Signal Processing Circuits," *NATO Advanced Study Institute on Logic Synthesis and Silicon Compilation for VLSI Design,* L'Aquila, Italy, July 1986.

[RDVG86] J. Rabaey, H. De Man, J. Vanhoof, G. Goosens and F. Catthoor, "*Cathedral II : A Synthesis System for Multiprocessor DSP Systems,*" NATO Advanced Study Institute on Logic Synthesis and Silicon Compilation for VLSI Design, L'Aquila, Italy, July 1986.

[RuSD85] R. Rudell, A. L. Sangiovanni–Vincentelli, and G. DeMicheli "A Finite State Machine Synthesis System," *Proc. ISCAS,* June 1985.

[Sang85] A. L. Sangiovanni–Vincentelli, "An Overview of Synthesis Systems," *Proc. Custom Integrated Circuit Conf.,* May 1985.

[Shah86] M. Shahdad, "An overview of VHDL language and technology," *Proc. 23rd Design Automation Conference,* June, 1986.

[Sout83] J. R. Southard, "MacPitts: An Approach to Silicon Compilation," *Computer,* Vol 16., No. 12, Dec. 1983.

[Snow78] E. A. Snow, "Automation of Module Set Independent Register–Transfer Level Design," PhD Dissertation, Carnegie Mellon University, April 1978.

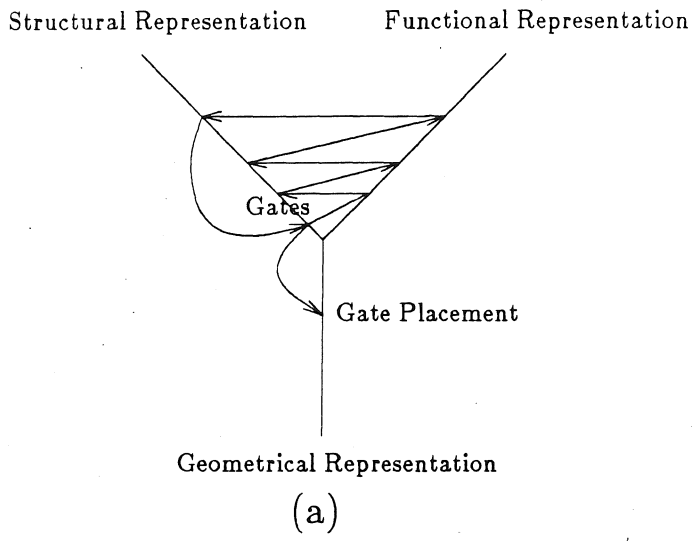
[THKR83] D. E. Thomas et al., "Automatic Data Path Synthesis," Computer, Vol 16., No. 12, Dec. 1983.

[TPLM86] C. Tzeng et al., "A Versatile Finite State Machine Synthesizer," *Proc. ICCAD,* Nov. 1986.

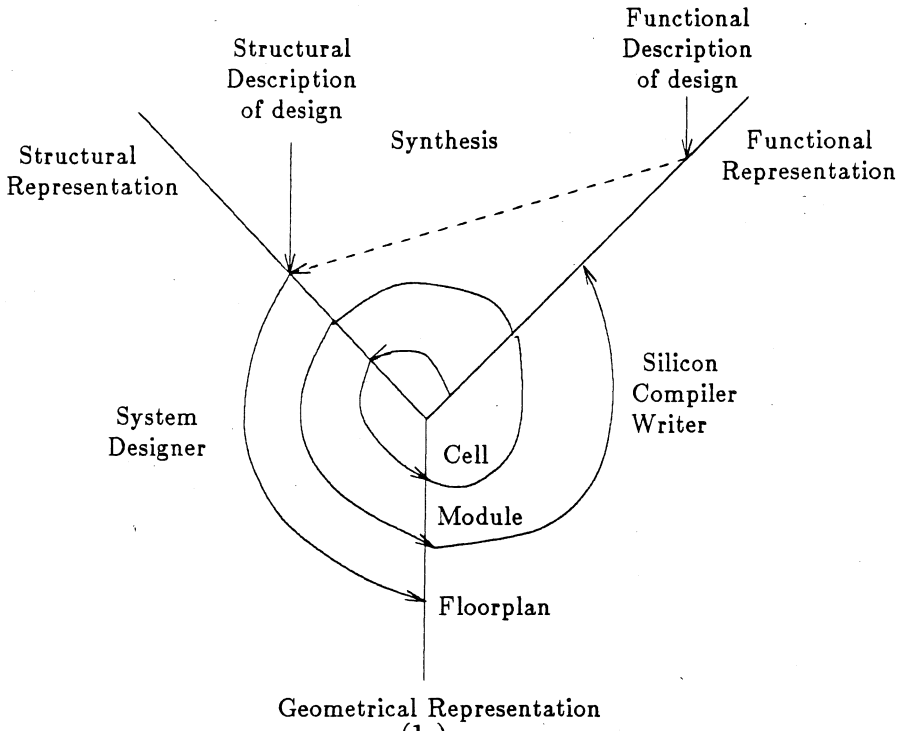
[Thom77] D. E. Thomas, "The Design and Analysis of an Automated Design Style Selector," *PhD Dissertation,* Carnegie Mellon University, April 1977.

[Trim84] S. Trimberger, "VTIcompose–A powerful graphical chip assembly tool," *Proc. ICCAD,* Nov. 1984.

[WaTh85] R. A. Walker and D. E. Thomas, "A Model for Design Representation and Synthesis," *Proc. 22nd Design Automation Conf.,* June 1985.



(a)

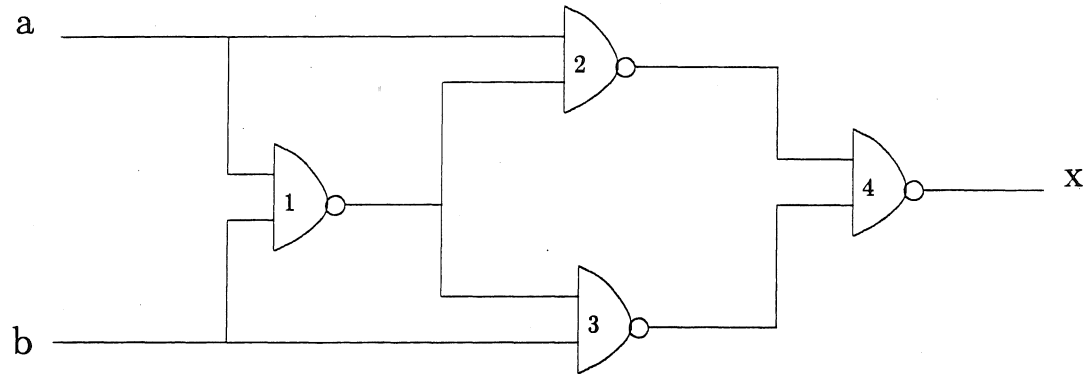


(b)

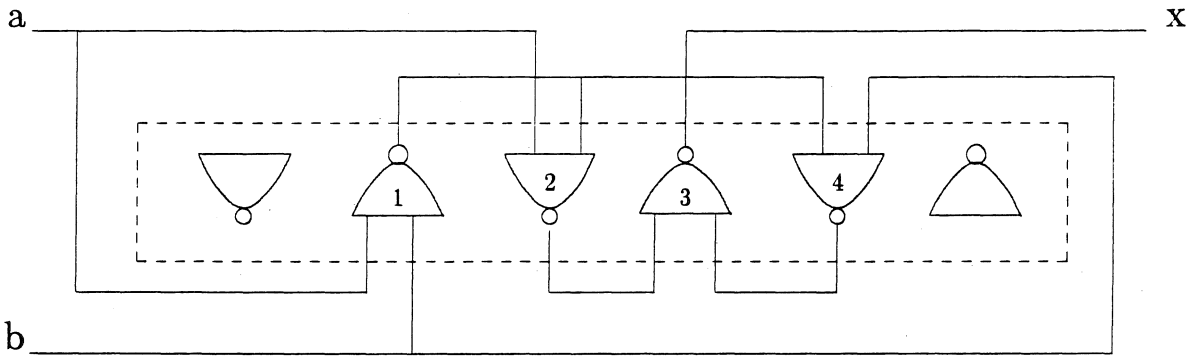
Figure 1. Design Methodologies:
(a) Gate–array
(b) Silicon Compilation

x = a'b + ab' after 30 ns

(a)



(b)



(c)

Figure 2. Representational Domains

(a) Functional
(b) Structural
(c) Geometrical

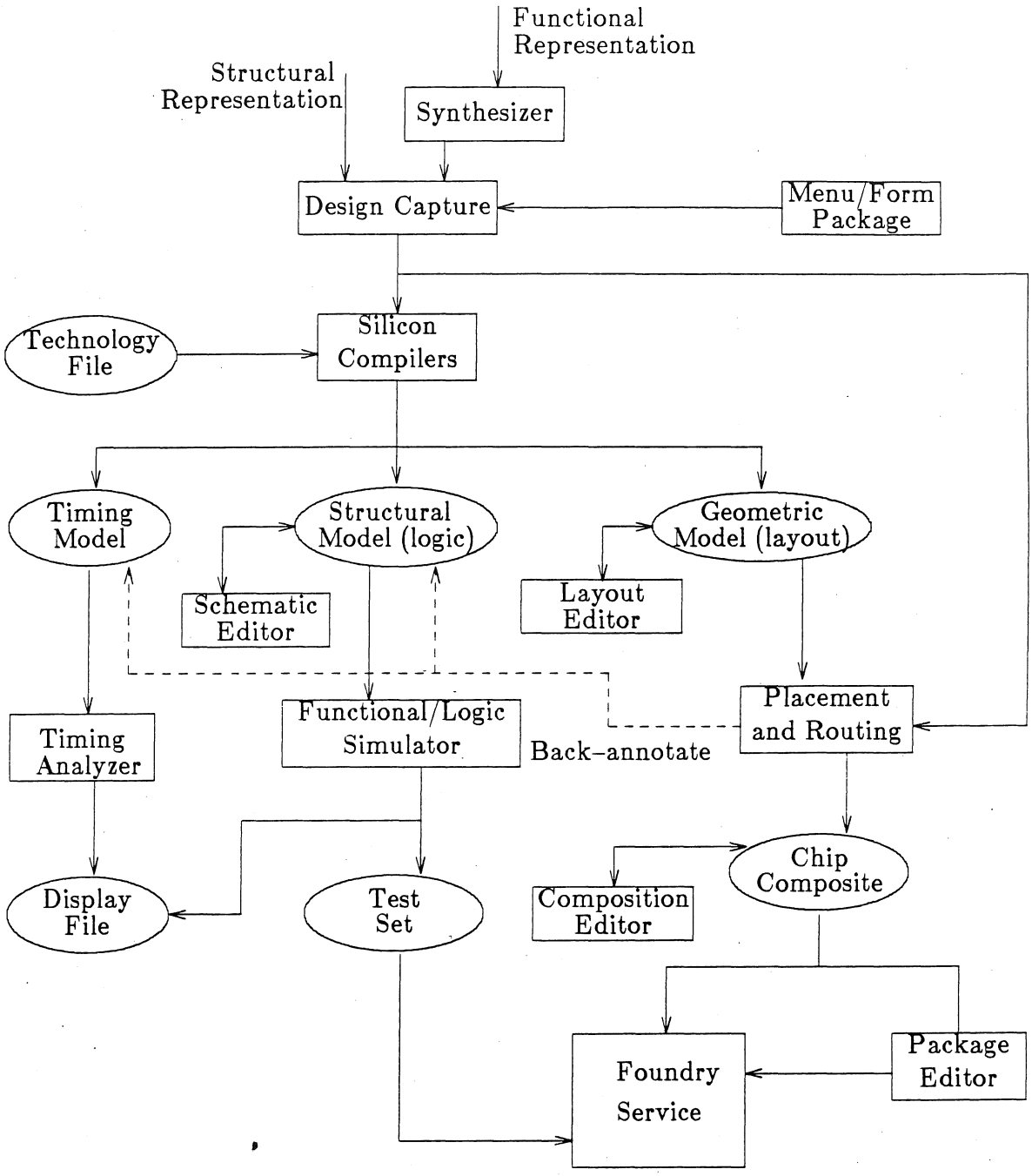


Figure 3. Silicon Compiler–based Design System

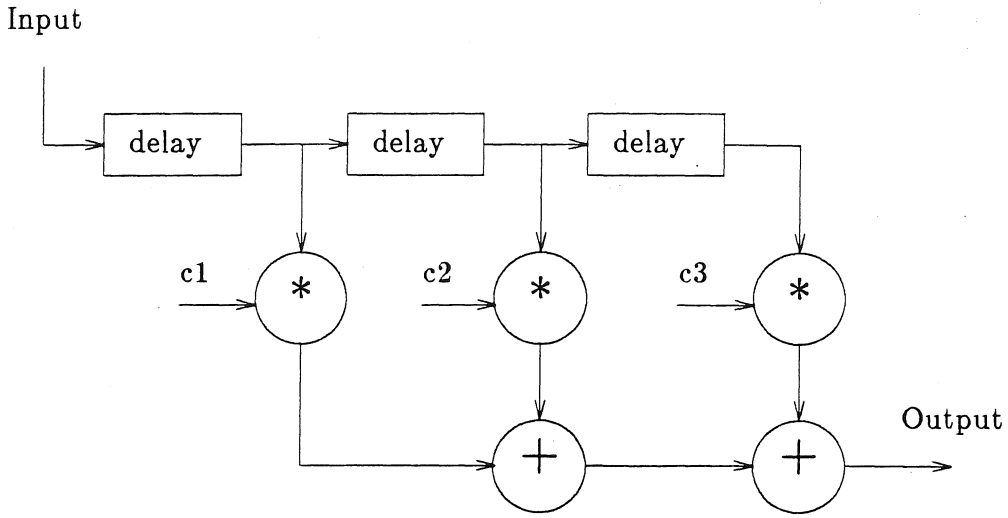


Figure 4. Digital Signal Processing Application

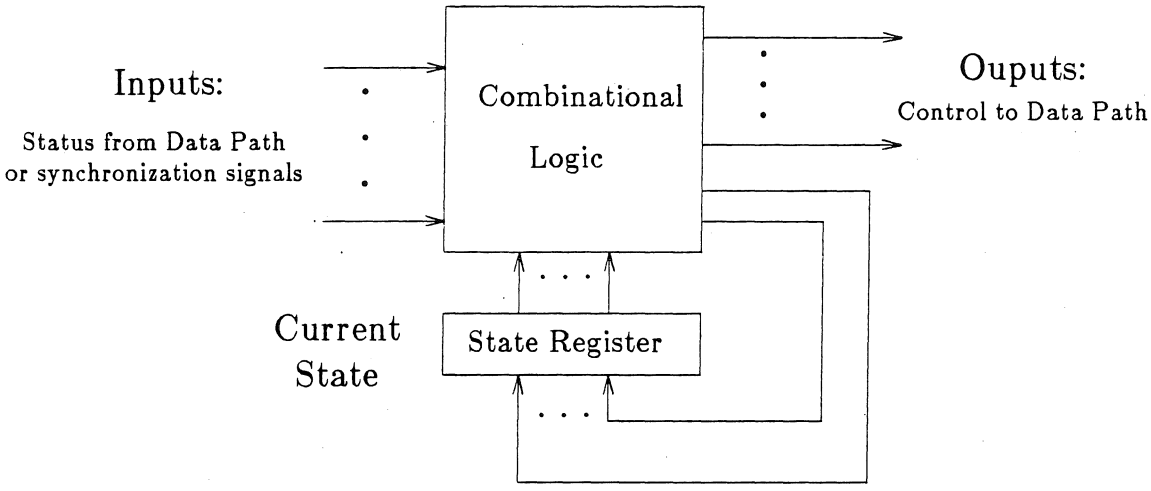


Figure 5. FSM–based Control Unit Model

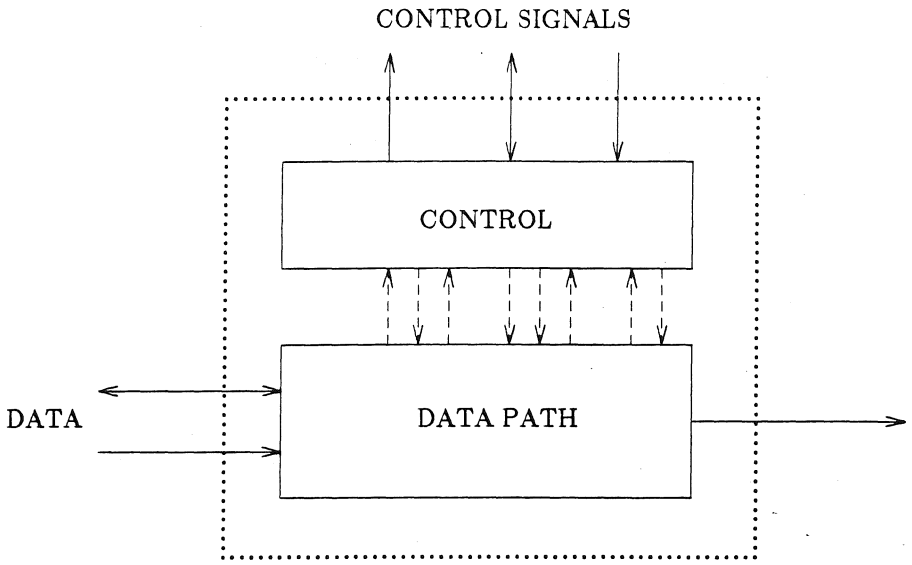


Figure 6(a). The MacPitts Architectural Model

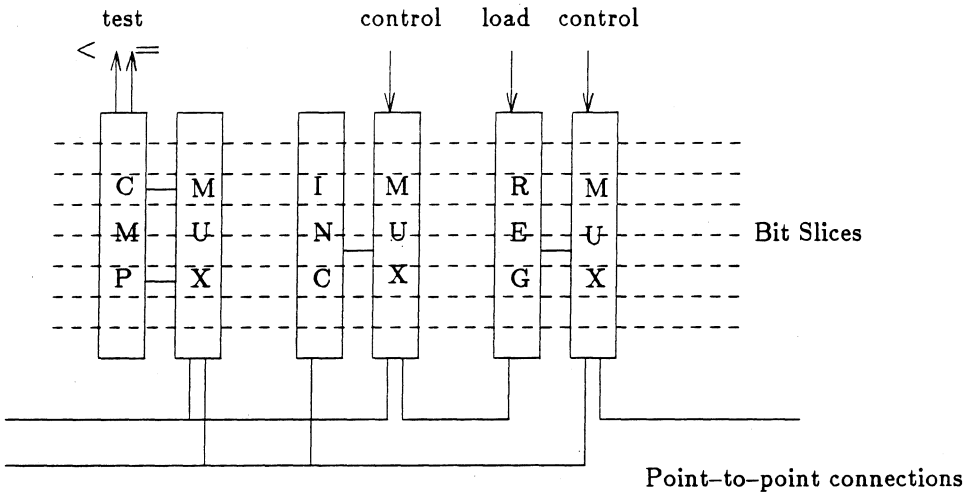


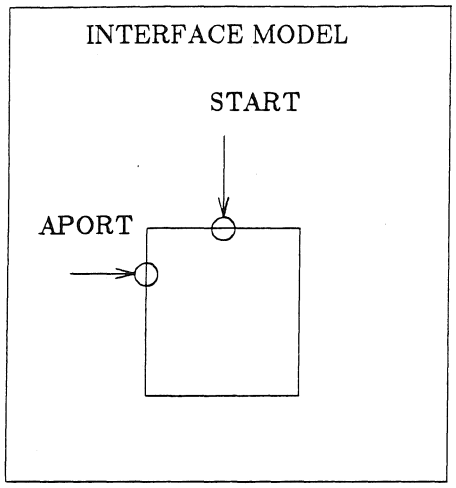
Figure 6(b). The MacPitts Data Path Model

```
(chip example
    (declare (word-length 8)
        ... )
    (fsm serial-swap
        (ser (set temp a)
            (set a b)
            (set b temp)))

    (fsm parallel-swap
        (par (set a  b)
            (set b a))))
```

Figure 7(a). Sequential/Parallel Constructs in Silc

```
program example( ... );
    procedure serialswap;
    begin
        temp <- a;
        a <- b;
        b <- temp;
    end
    procedure parallelswap;
    begin
        a <- b / b <- a;
    end
begin   %* MAIN *%
    call serialswap;
    call parallelswap;
end.
```

Figure 7(b). Sequential/Parallel Constructs in SYCO



INPUT DESCRIPTION

```
• • •
• • •

(CASE START OF
      '0':  A := '0';
      '1':  A := A + APORT;
            COUNT := COUNT + 1;
FO DELAY <= 200)
• • •
• • •
```

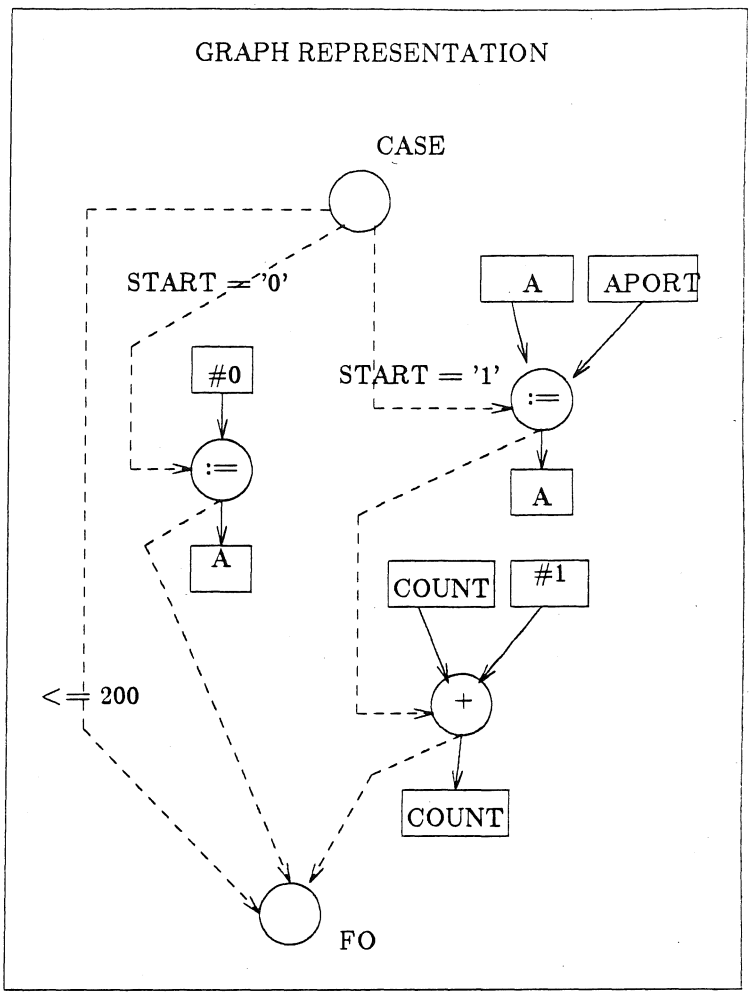


FIGURE 8. CADDY's FLOWGRAPH REPRESENTATION

INPUT DESCRIPTION

```
...
...

decode START
begin
    1 :=   A <- A + APORT next
           COUNT <- COUNT + 1;

    0 :=   A <- 0;
end
...
...
```

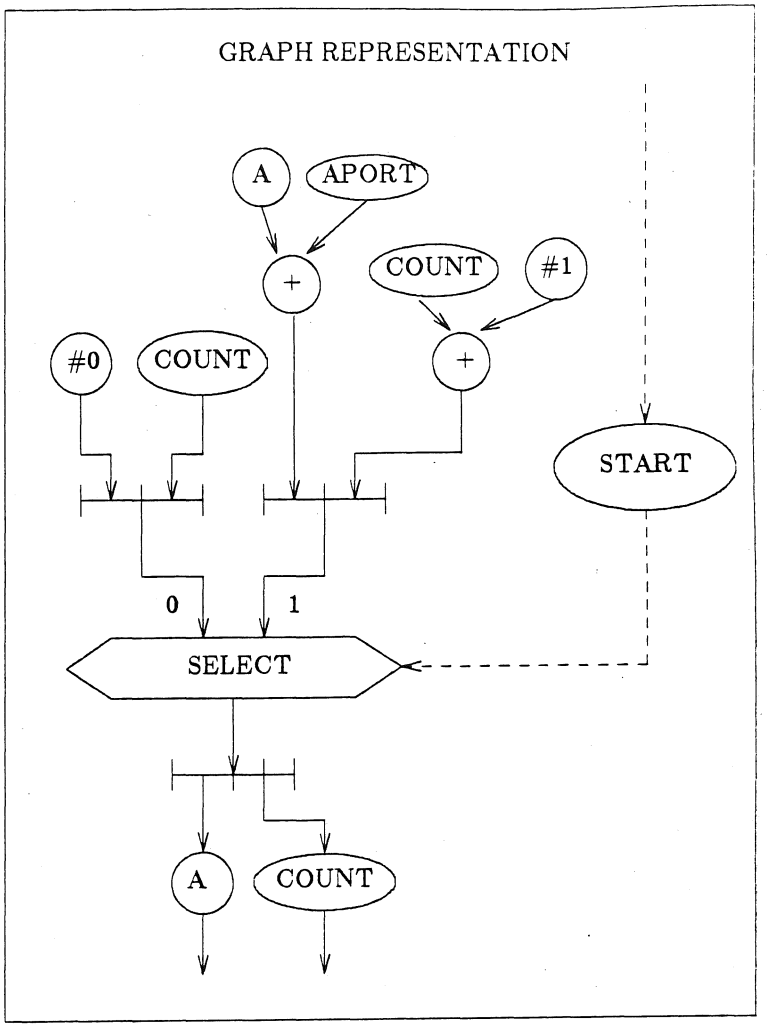


FIGURE 9. VT REPRESENTATION

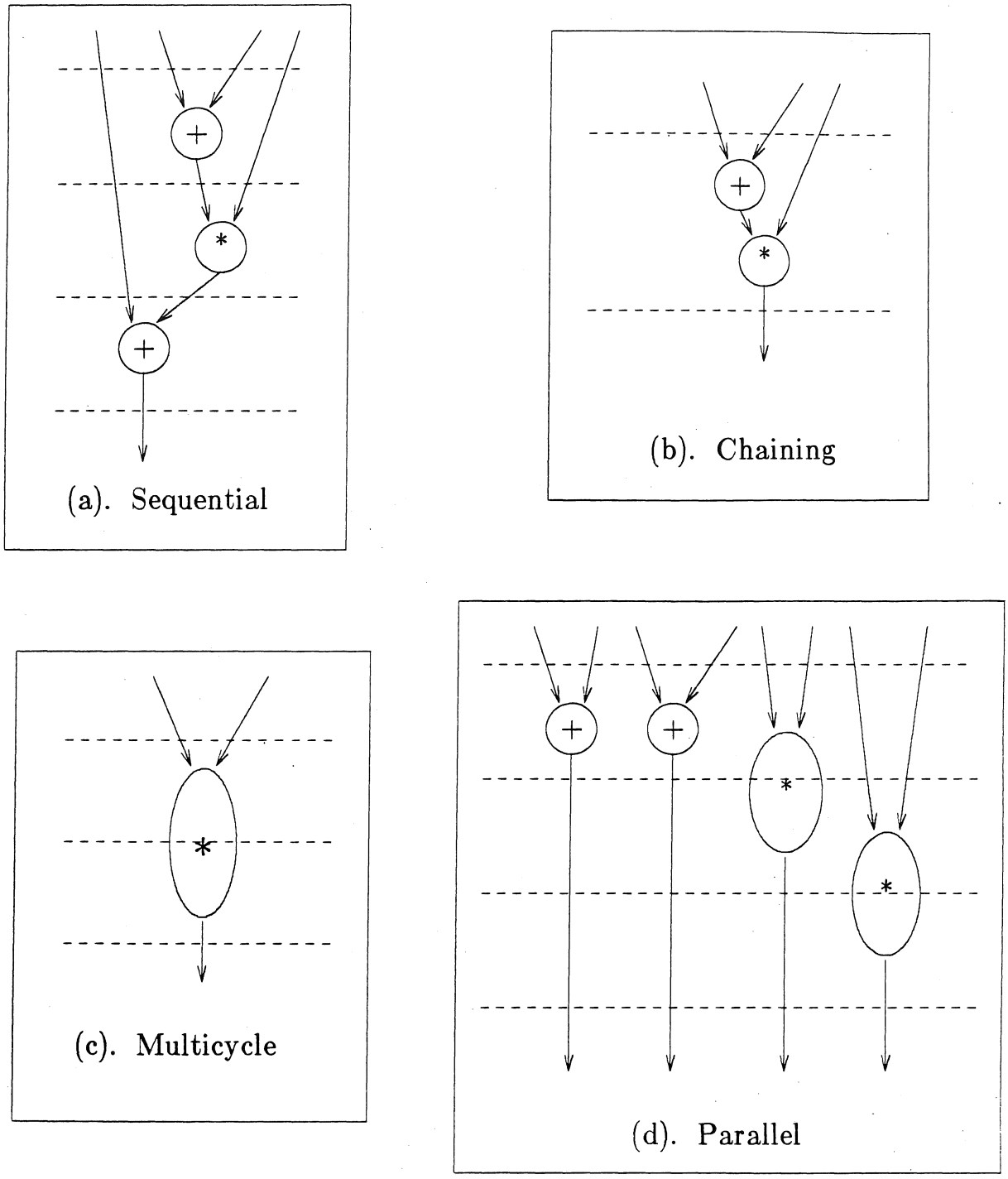
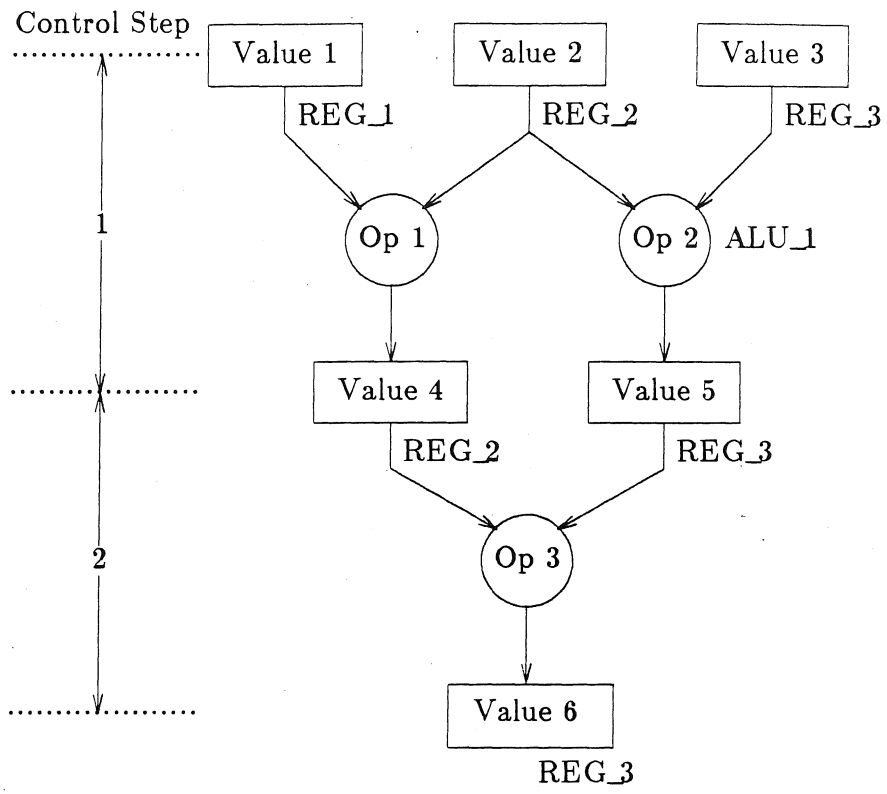


FIGURE 10. Execution of Graph Operations

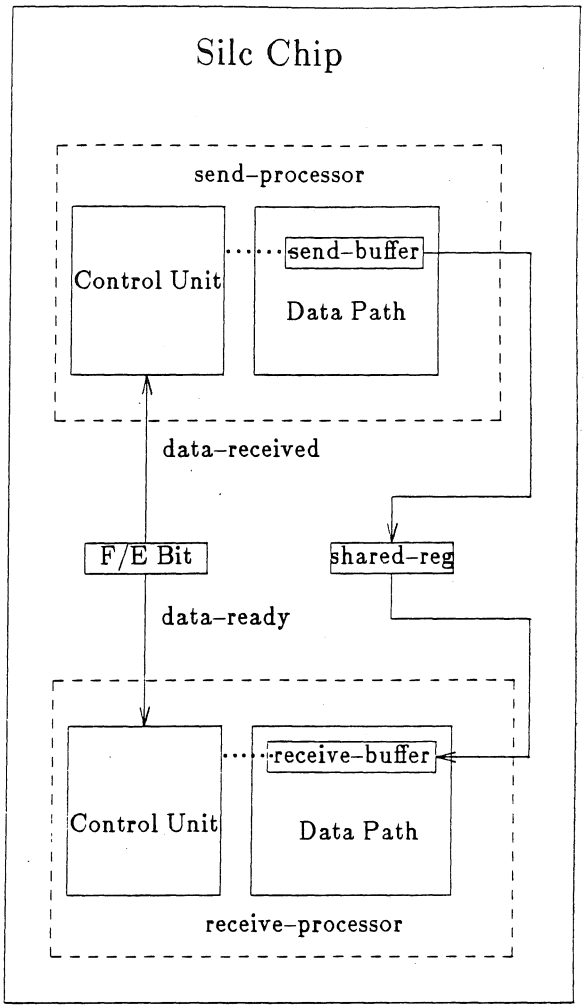


Op	ALU			Min–Max Cost
	1	2	3	
1	X	30	30	0
2	X	X	X	X
3	0	35	35	35

Figure 11. EMUCS Flowgraph and Cost Table

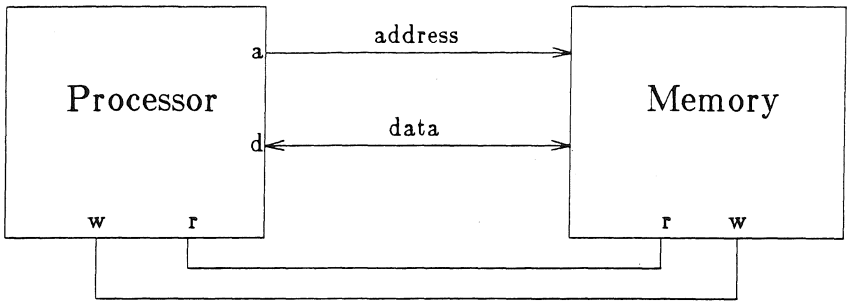
```
(chip async-protocol
        (declare (word-length 8)
                 :
        (clock1 pin 4)
        (clock2 pin 5)
                 :
        (variable shared-reg unstored asynchronous word))

    (fsm send-processor
        (declare (word-length 8)
                    :
                 (variable send-buffer stored word))
        (while t
                 :
           (while (not (data-ready shared-reg)))
           (set shared-reg send-buffer)
                 :
        ))

    (fsm receive-processor
        (declare (word-length 8)
                    :
                 (variable receive-buffer stored word))
        (while t
                 :
           (while (not (data-received stored-reg)))
           (set receive-buffer shared-reg)
                 :
        )))
```

(a) Silc Input Description



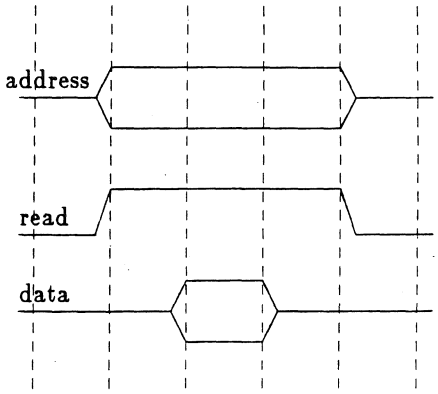
(b) Compiled Chip

Figure 12. Inter–FSM Communication in Silc



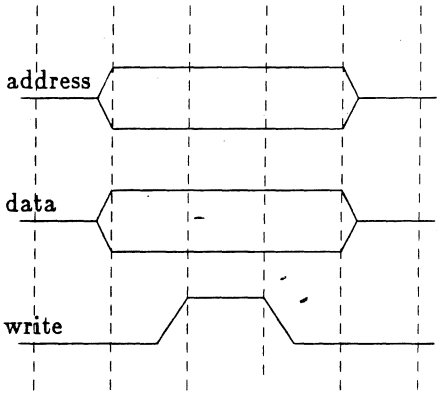
(a). Inter–chip communication

(par (setq address a) (setq read t))
(par (setq address a) (setq read t) (setq d data))
(par (setq address a) (setq read t))



(b). Read Protocol

(par (setq address a) (setq data d))
(par (setq address a) (setq data d) (setq write t))
(par (setq address a) (setq data d))



(c). Write Protocol

Figure 13. MacPitts External Protocol

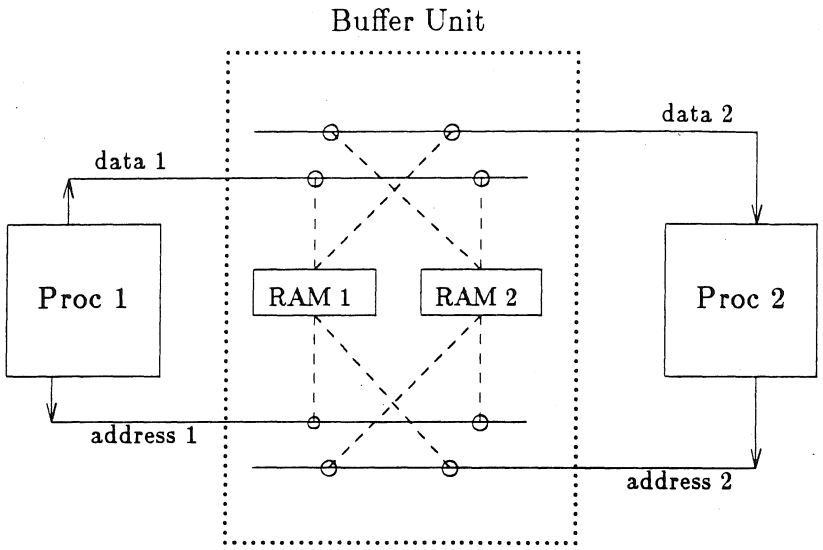


Figure 14. Synchronous Protocol using Interleaved RAMs

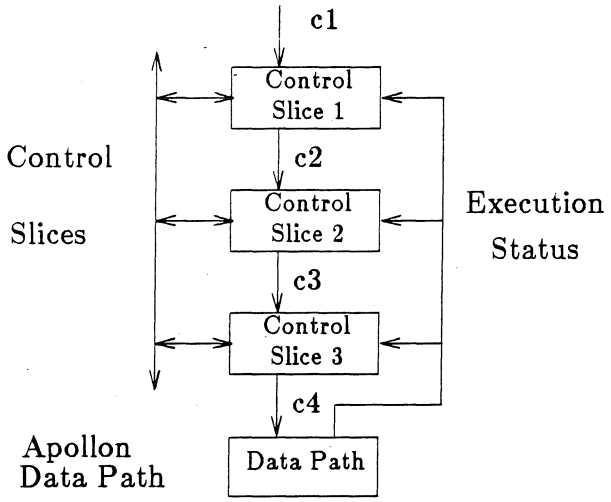


Figure 15(a). The SYCO Architectural Model

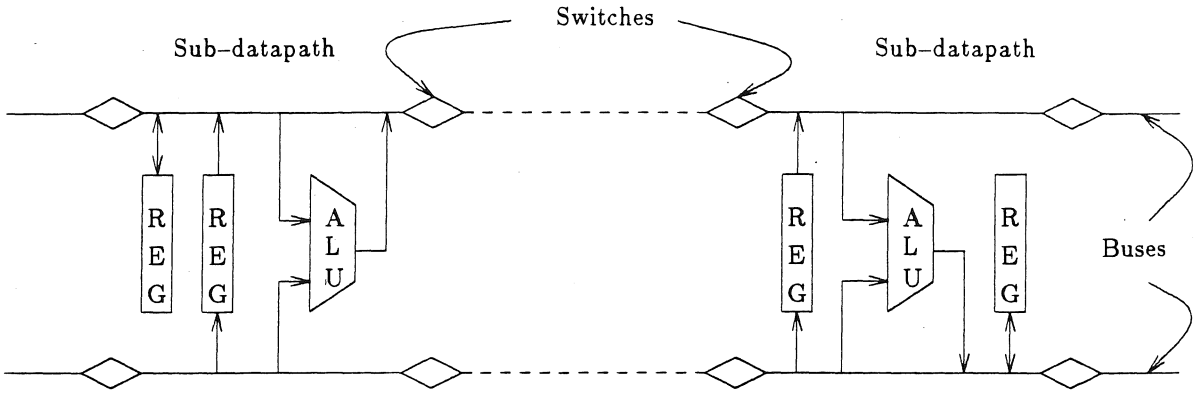


Figure 15(b). The Apollon Data Path

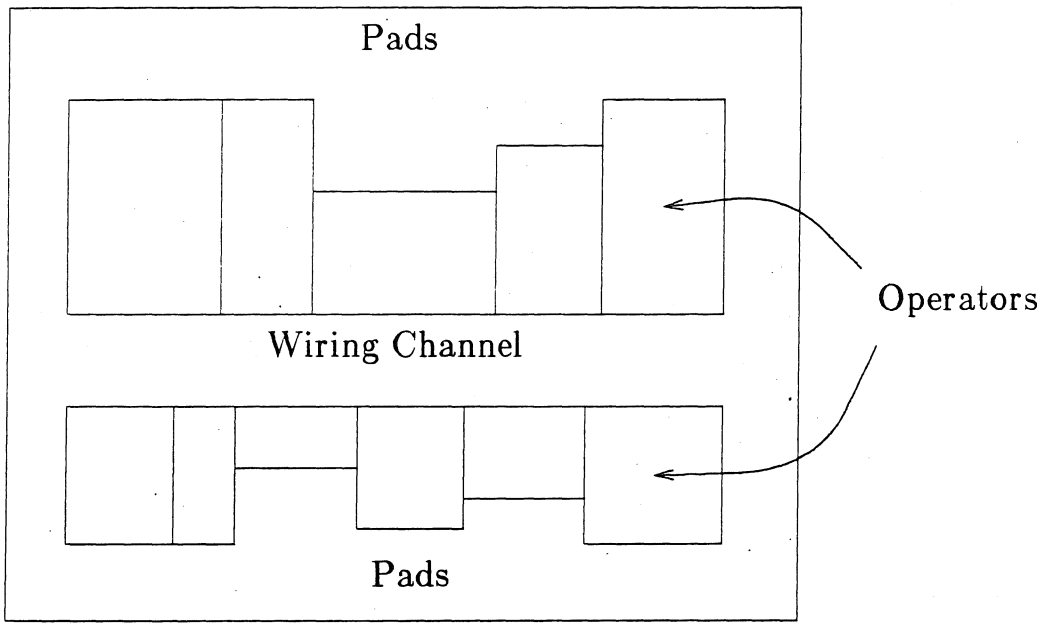
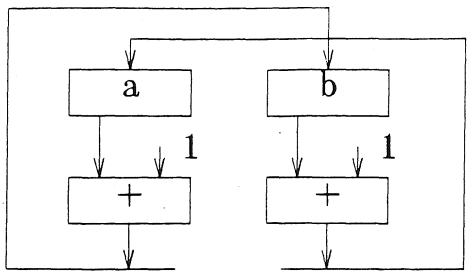


Figure 16. F.I.R.S.T. Topology

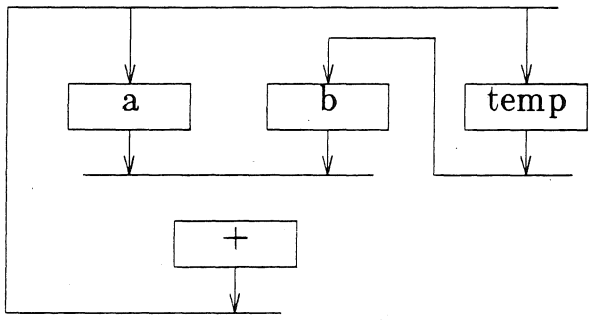
	Language	Intrmdiate Reprsntat'n	Conn'ctvty Model	Unit Models	Algorithms
Caddy/ DSL [Camp85]	Pascal–like	3 Graphs: data–flow control–flow constraints	Mux	Generated ALUs	Lifetime analysis Rule based reg and op assignment
Cathedral II [RDVG86]	Silage (applic–ative)	Data/Signal Flow–Graph w/control	Dedicated Busses	5 Execution Unit types	Integer Linear Programming
LAGER [PoRB86]	Silage (applic–ative)	Data/Signal Flow–Graph	Net List	Macrocells	Channel and River Routing
Silc [BlFR85]	LISP–Like Integer, Boolean	N/A	Mux	Generated ALUs with Chaining	Allocates Units Based on Conn'ctvty
SYCO [Jerr86]	Behav'rl Descr'ptn Level	N/A	2 Bus: MC68000 Based	ALU: Half Adder, Shifter, Carry, Flags	Minimize Dist. between Regs. and Units
Yorktown Silicon Compiler [BBCD85]	APL–like (Logic level)	YLE Logic Modules	Global Net List	Generated Combinat'l Logic	Sim. Annealing Logic Minimization

Table 1.



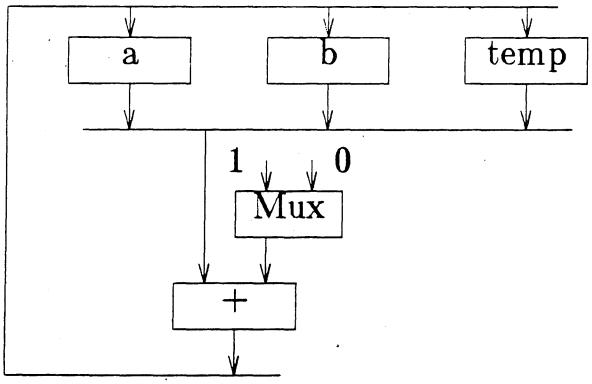
$a = b + 1, b = a + 1$

(a). Two Adders



$temp = a + 1$
$a = b + 1, b = temp$

(b). One Adder, Three Buses



$temp = a + 1$
$a = b + 1$
$b = temp$

(c). One Adder, Two Buses

Figure 17. Constraints and Tradeoffs