Knowledge-Based VLSI Design Group
Memo KB-VLSI-81-9 (working paper)
Last Modified: June 22, 1981 8:45 AM
First Draft: May 18, 1981
Filed on [Phylum]<KB-VLSI>Memos>LMA.Press

Linked Module Abstraction:

A Methodology for Designing the Architectures of Digital Systems*

by
Mark Stefik (Xerox PARC)
Dan Bobrow (Xerox PARC)

Abstract. *Linked module abstraction* is a methodology for designing digital architecture in which the sequencing of computational events is given primary attention. The sequencing of events is specified in terms of modules that carry out instructions and links between them that determine the flow and control of information. The methodology provides a number of simple composition rules for rapidly composing systems without certain classes of bugs. The resulting system descriptions can be implemented as either synchronous or self-timed digital systems.

This paper develops the underlying concepts and proposes graphical and programming language notations for the methodology. Their viability for describing and exploring architectural alternatives is demonstrated by examples of familiar subsystems drawn from design practice. The paper also discusses how the methodology was deliberately engineered to factor the concerns of a digital systems designer.

*This research was done as part of the Knowledge-Based VLSI Design Project, in collaboration with Stanford University. The Stanford component of this research is funded by the Defense Advanced Research Projects Agency.

Table of Contents

1. Introduction

Digital systems and their specifications can be complicated. A designer of complex systems must be able to partition his concerns in order to focus on important things. This paper proposes a model of digital system design, *linked module abstraction*, in which the sequencing of computational events is given primary attention. The abstraction is explicit about the decomposition and sequencing of computations. It is uncommitted about the structures for achieving the timing and logical properties of devices.

This work is part of an effort to extend the design methodology of the Mead and Conway textbook [Mead80] to cover more aspects of integrated system design. The linked module abstraction is intended to be used in combination with other descriptions of digital systems and their design methodologies ([Bell81], [Tong81]). Each methodology provides a set of concepts and rules for composing digital systems, and emphasizes a limited set of design concerns. Table I summarizes a set of design methodologies that are being integrated in an experimental system for digital system design (Stefik and Brown [Stefik81a]):

Table I. The partitioning of design concerns

Description Level	Concerns	Composition Rules	Examples of Bugs
Layout	Physical dimensions	Lambda Rules	Separation Errors
CPS	Digital behavior	Composition of PUEs, PDEs, and pass transistors	Charge Sharing Switching Levels Threshold drops
CRL	Clocking	Stage composition	mixed clocks unclocked feedback
LMA	Event Sequencing	forking, joining, module composition	deadlock data not ready

The *layout* description level is concerned with the constraints imposed by the electrical properties of silicon and the fabrication process. Composition rules at this level are the familiar *lambda* design rules (e.g., Mead and Conway [Mead80], Lyon [Lyon81]). The *clocked primitive switches* level (CPS) [Bell81] describes circuits in terms of pull-up devices, pull-down devices, and pass transistors; it provides rules for composing these that preserve digital behavior. The *clocked register and logic* level (CRL) [Bell81] describes digital systems in terms of clocked registers and combinational logic; it is concerned with the correctness of clocking signals in two-phase systems. Like the CRL abstraction, linked module abstraction (LMA) describes systems in terms of registers and logic. In contrast, it is uncommitted about the implementation of sequencing: it indicates the order of computations using forking and joining constructions. The composition rules deal directly with the relationships between control and data, and with the prevention of deadlock in the use of shared modules. The LMA language describes the structure of digital systems in terms of modules and subsystems; it describes the paths that data can flow, the sequential and parallel activation of modules, and the placement of registers. Specifications at this level highlight critical architectural trade-offs such as communication versus redundant computation, copied structures versus shared structures, serial versus parallel computation.

One of the motivations for LMA descriptions is the observation that component (or *cell*) libraries at the *layout description level*, which are a mainstay of present CAD systems for integrated system design, are inherently tied to the lifetime of a technology. Seemingly minor changes (such as the addition of a metal layer or buried contacts) can effectively obsolete them. A library of LMA structures is more independent of implementation technology; it can be related to implementation technology through a series of intermediate descriptions and the knowledge for transforming the descriptions and making design trade-offs. As technology shifts, the *knowledge for transforming* the descriptions must be changed, but a library of LMA modules would be relatively stable. Given multiple sets of transformation rules, LMA descriptions could be transformed into multiple implementations (e.g., cMOS/SOS and nMOS).

This document is organized as follows: Section 2 presents the basic vocabulary of the linked module methodology. It defines the computational elements, their computational phases, and rules for composing them. Section 3 proposes an equivalent programming language for the graphical notation used in Section 2. The programming language uses familiar concepts from conventional software and gives them appropriate hardware interpretations. The language is intended to make it easy to *program* hardware according to linked module composition rules. Section 4 uses the notations to illustrate architectural alternatives for some interesting and well-known subsystems. Section 5 reconsiders the methodology and shows how it usefully partitions architectural design decisions. Section 5 also shows how the composition rules eliminate some classes of design errors. Section 6 acknowledges a debt to a diverse set of researchers who have thought about related problems.

Suggestions to the Reader

The main purpose of the linked module abstraction is to provide a language for describing digital architectures. Examples of these are given in Section 4, and the reader is encouraged to browse through these early. The sections on composition rules and language features preceding these examples are quite long and detailed. The reader is encouraged to skim these sections quickly, since the examples are understandable without all of the details of the language. The most important concepts are modules, forks and joins, calling buffers. For computer programmers, the only unfamiliar notations are the fork-join constructions.

2. Methodology and Graphical Notation

This section introduces the linked module methodology for describing digital systems. It describes the basic elements for computation and communication, and some composition rules for designing digital systems using them.

2.1. Modules

Modules in the linked module methodology are computational elements that perform *complete* instructions. By this we mean that a module, once started, completes what it is doing before it can be restarted. Each module has a number of directional lines: a **Go** line and a **Done** line for synchronizing communication with other modules, optional **Input** and **Output** data lines, and an optional **Interrupt** line. Each module also has a set of input buffers corresponding to the input data lines.

{Figure 1 showing GO, DONE,
Interrupt, & Data lines of Module.}

A module is controlled by the absorption and emission of *tokens* on the **Go** and **Done** lines, respectively. The operation of a module takes place in three phases as follows:

Waiting for Go. During this phase the module is idle. The input buffers hold their previous values. The output lines are stable and driven to their values from the previous cycle.

Running. The process of starting a module is to first place data on the input lines and then place a token on the **Go** line. This causes the module to load the input buffers from the input lines and to enter the running phase.

Sometimes the activating module drives only a subset of the input lines. Those input buffers whose corresponding input lines are not connected to the activating module retain their previous values.

During this phase the module performs its computation, which is some function of the values of the input buffers.

To complete the computation, the module may use other modules (including registers) to perform parts of the computation.

The output data lines are driven with the results of the computation during this phase. The *running* phase ends (after a finite delay) when the computation is complete.

Waiting for Done. In this phase the module places a token on the **Done** line. The phase ends when the token is absorbed by another module. After this phase, the module re-enters the *waiting for go* phase.

If a module does not have the optional **Interrupt** line, then it can not accept a token on the **Go** line unless it is in the first phase of the computational cycle. If, however, a module has an **Interrupt** line, it can be interrupted by placing a token on that line. The token is absorbed immediately and the module runs its interrupt computation to send constants to other modules.

Modules without explicit interrupt code go immediately to the *waiting for go* phase.

When the interrupt code is finished, the module goes to the *waiting for go* phase. A major application interrupt lines is for specifying initialization procedures in digital systems.

2.2. Functional Composition

An important advantage of specifying the computational cycle in terms of an abstract protocol is that the rules for linking modules together to form computational subsystems are quite simple. A simple example is the serial linking of modules to create composite functions. The following figure illustrates the linking of modules **F** and **G** to compute **F (G (x))**, that is, **F ∘ G** (x):

{Figure 2 of F (G (X)).}

In this figure, x is connected to the input of module G, and the output of G is connected to the input of F. Since G's **Done** line is connected to F's **Go** line, F can not begin its calculation until G has finished (which is exactly what we want). The figure shows a pipeline arrangement that allows G to begin another calculation as soon as F absorbs its token. If, however, we want to specify a composed module FG that does not accept another input until it has delivered its output, we can indicate this graphically as follows:

{Figure 3 of FG composite module.}

The module box indicates explicitly that FG coordinates its Go and Done tokens. In this case, G will not be offered another Go token (via FG) until after F has delivered its Done token to F An equivalent graphical notation for FG

{Figure 4 of FG Redrawn}

emphasizes that the coordination of F and G has nothing to do with their illustration in the interior of FG. That coordination is caused by the module box of FG interacting with the serial connection of the Go and Done lines of F and G.

2.3. Calling Buffers

We say that module F *calls* module G if module F activates module G to perform part of its computation. If F calls G, then necessarily G will finish before F. In the linked module abstraction, there are two constructions for calling: buffered and unbuffered. A buffered call passes the activating token through a *calling buffer* and an unbuffered call does not.

This section describes the communication element termed a *calling buffer* for communicating with *shared* submodules. Shared submodules are modules that are called by more than one module. Calling buffers are used to coordinate the use of shared submodules (e.g., to signal the calling module when submodules are finished and to allocate the use of submodules among calling modules without deadlock).

A calling buffer has a number of directional lines: a **Go** line and a **Done** line for synchronizing with the calling module, a **Call** and **Return** line for synchronizing with the called and shared submodule, and optional data lines. There are input data lines for connecting to the input buffer of the calling module, and output data lines for connecting with the output lines of the calling module; there are also *call data lines* and *return data lines* for data transmission to the called module. A data buffer is associated with each of the output data lines.

{Figure 5 of Calling Buffer}

Like a module, a calling buffer is controlled by the absorption and emission of tokens on its **Go** and **Done** lines, respectively.

In Section 2.5, we will see that there are two protocols for invoking shared modules. This section describes the *normal* or Start/Wait protocol. The other is the Wait/Start (or *reversed*) protocol.

The operation of a calling buffer takes place in four phases as follows:

Waiting for Go. During this phase the calling buffer is idle. The output buffers of the calling buffer hold their previous values. This phase ends when a token is placed on the **Go** line.

Calling. During this phase the calling buffer places its input data on the calling data lines, and emits a token on the **Call** line.

The input data lines of the calling buffer must be connected to either (1) the input buffer of the calling module, (2) the output lines of a module known to be idle, or (3) an output buffer of another calling buffer known to be idle at the time of the call. In the first case, the input buffer is stable since the calling module can only be activated during the *running* phase of the calling module. In the latter cases, the output buffer (or output lines) must be stable because the calling buffer (or module) is in a *waiting for go* state..

Waiting for Return. This phase is entered when the token is absorbed from the **Call** line. The calling buffer then absorbs the token from its **Go line.** During this phase, the shared submodule performs its computation. When the submodule is done, it returns the data on the return data lines and emits a token on the **Return** line. This causes the calling buffer to load the data into its output buffer and then to absorb the token from the **Return** line (releasing the shared submodule).

Waiting for Done. In this phase the calling buffer synchronizes with the calling module. It places the output data on the output data lines and emits a token on the Done line. This phase ends when the token is absorbed from the Done line. After this, the calling buffer re-enters the *waiting for go* phase.

The calling buffer is an important construction in our methodology. Section 2.5 shows that a calling buffer can be constructed using a *Start* element and a *Wait* element. Section 2.6 gives the composition rules for linking a calling buffer to the calling module and called submodule.

2.4. Linking Modules Together

A *token* in the linked module computational model is analogous to a *program counter* in a conventional computer. One can imagine representing the program counter as a token which sweeps through a program. The token activates an instruction when it reaches it, and moves on when the instruction is completed. In digital systems, unlike most computer programs, it is common for several modules to be active at the same time (i.e., for there to be more than one *program counter*). The linked module abstraction provides *forking* (i.e., fan-out) and *joining* (i.e., fan-in) constructions to control the sequencing in these systems. There are three major qualifiers called *all*, *any*, and *select* as explained in the text below:

The Synchronizing and non-Synchronizing All-Forks

The *non-synchronizing all-fork* delivers the output of one module to each of several receiving modules. It is indicated graphically as follows:

{Figure 6 of All-Fork}

When the input module emits a token on it **done** line, the all-fork emits a token to each of the receiving modules. Each of these modules absorbs the token when they are ready. When all of them have absorbed a token, the all-fork absorbs the token from the input module.

Since the data through an all-fork is always switched the same way that the tokens are, it is sometimes convenient to omit either the Go and Done lines or the data lines in the graphical notation. This convention is also followed in the notations for other kinds of forks and joins as appropriate.

A *synchronizing all-fork* starts all of the receiving modules simultaneously, when they are all ready. A synchronizing all-fork is indicated graphically by three horizontal bars (instead of two).

The Any-Fork

The *any-fork* is used to deliver the output of one module to any one of several other modules. It is indicated graphically as follows:

{Figure 7 of Any-Fork}

When the input module emits its token, the any-fork waits until one of the receiving modules is receptive. If more than one receiving module is ready, a precedence ordering is used.

The precedence can be indicated explicitly by numbering the lines to the receiving modules, or implicitly by left to right listing.

The any-fork emits a token to the chosen module and then absorbs the token from the input module.

The Select-Fork

The *select-fork* delivers the output of one module to one of several receiver modules. It is indicated graphically as follows:

{Figure 8 of Select-Fork}

When the input module emits the token, the select-fork reads a data line (labeled key) from the input module indicating which of the possible receiving modules has been selected, and then offers a token to it. When the receiving module has absorbed the token, the selector fork absorbs the token from the input module.

In Section 5, we will consider some ways to extend the coverage of the methodology by adding some more general (and complex) kinds of select-forks. For example, a simple generalization of the select-fork allows tokens to be sent to several receiving modules at once.

The All-Join

The *all-join* is used when a single module needs the data from several output modules. It is indicated graphically as follows:

{Figure 9 of All-Join}

Each of the output modules emits a token to the all-join when their data are ready. The all-join then offers a token to the receiving module. When the receiving module absorbs the token, the all-join absorbs all of the tokens from the output module.

The data lines from the output modules must go to distinct inputs of the receiving module to avoid conflicting data..

The Any-Join

The *any-join* is used when a single module can accept data from any of several modules. This is illustrated graphically as follows:

{Figure 10 of Any-Join goes here.}

The output modules emit tokens to the any-join when their data are ready. When the receiving module is ready, the any-join chooses one of the input modules using a precedence ordering.

As before, the precedence can be indicated explicitly by numbering the wires from the output modules. If no numbers are shown, then a left-to-right precedence is implicit.

The any-join then switches the *data-joins* to connect the data lines from the chosen output module to the data lines of the receiving module.

If some of the input wires of the receiving module are connected to only one of the output modules, then no data-join switch is required for them. These inputs will only be loaded when the activation is caused by the corresponding output module.

The any-join then offers a token to the receiving module. When the receiving module absorbs the token, the any-join absorbs the token from the activating output module. The any-join conserves tokens: tokens that are not absorbed during one cycle of operation, will be available for absorption during the next cycle according to precedence.

2.3. Examples

This section illustrates how to use the forks and joins for some common constructions.

Sequential Instructions

In the functional composition example in Figures 2 through 4, the data lines are interconnected so that the output of each module is the input of the next in the sequence. The following figure shows a more general example of sequential instructions:

{Figure 11 (a & b). Sequential Instructions}

F is a module which sequentially activates three other modules: A, B, and C.

The connections are exactly the same if the boxes labeled A, B, and C represent calling buffers for shared submodules.

Figure 11a shows a general way to connect **Go** and **Done** lines to sequence module activations. Module F can not finish until all of the instructions have been performed. Furthermore, the data on F's output lines must be valid until F is completed. Modules B and C compute output data for F. The all-join above F's **Done** ensures that B and C will hold their data (possibly in registers not shown) until F is done. The all-fork after B enables C to start without waiting for F to finish.

Some optimizations of the control wiring are possible in this example. Figure 11b suggests that it is enough to simply connect the control wires of the modules serially, leaving out the all-fork and all-join. To see why this works, consider that A, B, and C are in a module without loops, and A can only be started when the the module is. It follows that if B is running, then A must be finished; and if C is running, then B must be finished. Furthermore, by definition, the output lines of a module are stable once its computation has finished. Hence, it is not necessary to keep B in a *waiting for done* state, and the input terminal of the outer module's **Done** line can be connected to C (eliminating both the fork and the join). The more elaborate control wiring of Figure 11a is needed only there are loops going back to reactivate modules A or B.

Re-using Outputs

Sometimes it necessary to use the output of a module as input to more than one module. An example of this is the computation of F(G(H(x)), H(x)), where we want to use the *same* value of H(x) in computing both arguments of F. In conventional programming languages, the following sequence can be used to achieve this:

Temp ← H(x)

F (G(Temp), Temp)

This approach uses some intermediate storage (Temp) to avoid calling H(x) twice. The following figure shows how to specify the calculation using forks and joins, but without any additional registers:

{Figure 12 of F (G (H(x)), H(x)) }

The all-fork insures that H will hold its output lines until both G and F have accepted the data. The all-join insures that F will not begin its computation until both H and G have finished. Section 3 shows how the *wiring* of this construction can be expressed in a programming notation.

Starting and Waiting

Start and *Wait* are useful timing signals between modules. *Start* means to activate another module without waiting for it to finish. *Wait* means to wait for another module to finish, without necessarily initiating it. In the linked module abstraction, *Starts* and *Waits* usually come in pairs.

Start and *Wait* have been formalized as timing signals on *semaphore* variables in operating system design [Hansen73].

Figure 13 illustrates a way of generating a *start* signal in the linked module notation:

{Figure 13 of Start}

This construction uses a synchronizing all-fork to generate a token on the **Call** line to start the called module, a token on the **LinkOut** line to enable the associated *Wait*, and a token on the **ContinueOut** line to pass control to the next instruction. The optional input data lines can be used to pass data to the started module on the call data lines.

Although this implementation of *Start* does not wait for the module to finish, it does wait for it to be started.

Figure 14 shows how to construct a *Wait* in the linked module notation:

{Figure 14 of Wait}

This construction uses an all-join and a module (drawn sideways) to implement the *Wait*. The all-join accepts an enabling token on the **link-in** line from the associated *Start*. (A token arriving on this line indicates that the associated *Start* has fired, and the *Wait* can prepare to accept data from the started module.) When the awaited module finishes, it sends a token on the **Return** line. The optional return data lines, buffer, and output data lines are used to hold the output data from the awaited module. A token arriving on the **ContinueIn** line indicates that the intermediate instructions between the *Start* and *Wait* have been completed.

The reader may have noticed that the *calling buffer* can be implemented by a *Start* followed by a *Wait*. Figure 14.1 illustrates an interconnection that produces the 4-phase cycle of a *calling buffer*. (Because no intermediate instructions are used between the *Start* and the *Wait*, the **ContinueOut** and **ContinueIn** lines are not needed.)

Wait/Start Buffers

In a reversed protocol buffer the *Wait* comes before the *Start* as in the following figure:

{Figure 15. Wait-Start Buffer.}

This reversed-protocol is not suitable for subroutine-like calls, because the value returned does not depend on arguments sent in the current call. Like normal protocol, reversed protocol provides that the submodule be called once for every activation of the calling buffer. In Wait/Start buffers, however, the computation of the calling module potentially overlaps the computation in the called module.

Two uses for Wait/Start buffers are for communication with *data generators* and *data absorbers*. A data generator is a module which delivers data to the calling modules without using the parameters of the calling module. In ordinary computer programming, random number generators and assynchronous I/O would be examples. A data absorber would be a routine that accepts the data from the caller and processes it, without providing further feedback to the calling module. A possible example of this would be writing data to a disk. Modules that are called by a Wait/Start protocol need to be in a *waiting for done* phase after system initialization.

Token Generators and Absorbers

In all of the examples so far, modules are started, run for a period of time, and emit a single token. This kind of behavior is characteristic for most components of the digital systems that we have considered. Sometimes, however, it is convenient to have subsystems termed *generators* that emit information and tokens repeatedly without being restarted. Similarly, it is sometimes convenient to have subsystems termed *absorbers* that absorb information repeatedly, without emitting tokens.

Constructions for generators and absorbers is illustrated in the following figure:

{Figure 15.1. (a) Token Generator and (b) Token Absorber.}

Generators and absorbers both have an **initialize** line which must absorb a token to start them. (Such a line would be activated through a Start from another module.) Both systems are composed of two *null modules* (i.e., modules that perform no computation). The token generator works as follows.

(1) The generator receives a token on its **initialize** line.
(2) The first module runs and emits a token on its **Done** line.
(3) The second module absorbs the token, runs, and delivers a token to the all-fork.
(4) The all-fork sends one token to the receiver on its **Source** line, and another token back to the first module.

Once initialized, the token generator will emit fresh tokens on its **source** output whenever tokens are absorbed from that line. The token absorber works as follows:

(1) The absorber receives a token on its **initialize** line.
(2) The second module runs and emits a token on its **Done** line.
(3) The all-join receives one token.
(4) If a token appears on the absorber's **Sink** line, the all-join will emit a token to the first module.
(5) The first module will then run, and emit a token to the second module.
(6) As before, the second module will emit a token to the all-join.

Once initialized, the token absorber will absorb any tokens that appear on its **Sink** line.

2.6. Composition Rules

Composition rules for linked module abstraction govern the interconnection of modules, both externally and through calls. These rules are intended to make it easy to create subsystems and composite modules. Many of these composition rules will be incorporated into the grammar of the programming notation of Section 3.

Legal Connections Rule

The legal connections rule says that lines are oriented and that they only can connected terminals of the same type. This rule has been used implicitly in all of the examples so far. The following chart summarizes the legal connections:

Type of Line	Connects From	Connects To
Data Line	Output Data Terminals	Input Data Terminals
	Output Data Terminals	Select Key Terminals
	Output Data Terminals	Data Join Inputs
	Output Data Terminals	Return Data Terminals
	Call Data Terminals	Input Data Terminals
	Input Buffers	Input Data Terminals
	Data Join Inputs	Input Data Terminals
Control Line	Done Outputs	Go Inputs
	Done Outputs	Fork Inputs
	Done Outputs	Join Inputs
	Done Outputs	Return Inputs
	Go Outputs	Go Inputs
	Go Outputs	Call Inputs
	Fork Outputs	Go Inputs
	Split Outputs	Go Inputs
	Fork Outputs	Join Inputs
	Fork Outputs	Fork Inputs
	Join Outputs	Join Inputs
	Join Outputs	Join Inputs
	Return Outputs	Done Inputs

Single Connection Rule

Control wires and data wires connect an output terminal to an input terminal. No terminal can be connected to more than one wire.

Notation: Sometimes it is convenient to omit all-forks and all-joins in a figure by making them implicit when more than 1 wire is connected to a terminal.

Fan-out Rule

If the output data lines of a module (M) connect to more than one other module, then the Done line must be connected through a fork to the Go lines of all of the other modules.

More specifically,

If the data are intended to go to all of the other modules each time, then the Done line should be connected through an all-fork and the data can be connected directly.

If the data are intended to go arbitrarily to one of the other modules, then the Done line should be connected through an any-fork and the data must be connected through a corresponding any-data-fork.

If the data are intended to go selectively to one of the other modules, then the Done line should be connected through a select-fork and the data must be connected through a corresponding select-data-fork.

This rule ensures that module M will hold its output data (initiating no new computation) until the other modules have received it.

Fan-in Rule

If the input data lines of a module are connected to more than one other module, then its Go line must be connected through a join to the Done lines of all of the other modules.

This rule ensures that the input data for a module needs will be stable when the module begins.

This rule is superfluous in cases where it can be proven that the sequencing of modules necessarily implies that the input data must be stable. For example, serial modules without looping inside a calling module are necessarily activated sequentially, and whenever a module is activated the outputs of all previous modules in the series must necessarily be stable. In such cases, the data-join can be *optimized* away.

The Token Conservation Principle

Most of the linking methods exchange a single token in a transaction, with two exceptions: the all-fork emits more tokens than it absorbs, and the all-join absorbs more tokens than it emits. However, the operation of a module is based on the absorption and subsequent emission of exactly one token in each computation cycle. This leads to the following principle:

For all paths through a module, the emission and absorption of tokens inside must exactly balance.

The following figure shows some important fork-join constructions that balance their token transactions:

{Figure 15.2. Common Fork-Joins}

The combinations of forks and joins in the following figure do not conserve tokens:

{Figure 15.3. Fork-Joins that do not conserve tokens}

The all/any fork-join in Figure 15.3 acts as a token-multiplier; the any/all fork-join acts as a token divider. These constructions cannot be used by themselves in a module, since they do not conserve tokens.

Composite Module Rules

A module that calls other modules to perform parts of its computation is called a *composite* module. The submodules of a composite module may finish at different times. The general requirement for composite modules follows:

All of the outputs of a composite module must be computed before its Done token is emitted.

Submodules may be linked serially to perform sequential computations, or in parallel using forks and joins to perform parallel computations. In both cases it is essential to connect the **Done** lines of the submodules to the **Done** input terminal of the composite module in such a way that the composite module will emit an output token when the overall computation is finished.

A *single function composite module* is one that always computes the same function of its inputs. When it is done, it supplies output data on all of its output lines. Several ways to achieve this are listed below:

(1) By connecting the **Done** lines of all the modules that compute output data to an all-join. (This does not work if there are loops in the module.)

(2) By using forks and joins only in token-conserving fork-join combinations inside the module.

(3) By augmenting the constructions in rule (2) with loop structures as in Figure 15.4.

{Figure 15.4 Example of a Loop Construct.}

Figure 15.5 shows an example of an ill-formed composite module M. Submodules A, B, and C are activated in parallel, but M only waits for A; the tokens emitted by B and C are absorbed by the token absorber TA. The problem with M is that there is no guarantee that the data output by B and C will be valid when M indicates that it is done.

{Figure 15.5. Example of ill-formed Composite Module.}

Figure 11 shows an example of the interconnections for a single function module composed of sequential calls to submodules. Since submodules B and C compute output data, their **Done** lines are connected to the all-join at the bottom of the composite module.

A *multi-function composite module* is one that performs one of a set of functions according to a command. Typically, a different set of outputs is computed for each command. The following figure illustrates a multi-function module:

{Figure 16. Multi-function composite module.}

By convention, multi-function modules have a special data line designated *command*. This line is used as the key to a select-fork which activates a set of modules (called the *command code*) for each command. The code for a command is, in effect, similar to a single function module: it reads a subset of the composite module's input buffers, and drives a subset of the composite module's output lines. Commands may share input data and other computational structures; they are mutually exclusive in their times of execution. In Figure 16, submodules A1 and B1 are the command set for function 1, and module A2 is the command set for function 2.

Each command set for a composite module must follow the rules for a single-function module.

Shared Module Rule

Shared modules are modules that are called by more than one module. In this section, we will confine our attention to modules following the normal (or Start/Wait) protocol. A composition rule follows:

Calls to shared modules should be buffered.

This composition rule is motivated by the problem of deadlock, which sometimes occurs when parallel systems share resources. For a simple example of deadlock, consider two modules F and G, both of which use submodules A and B to some of their output data. When F receives a token on its **Go** line, it activates A and then B; when G receives a token on its **Go** line, it activates B and then A.. By the output holding rule, all submodules must hold their output data (staying in the *waiting for done* phase) until the calling module is finished. Suppose that F activates A just before G activates B. At this point, A waits for F to finish, F waits to activate B, B waits for G to finish, and G waits to activate A.. Without outside intervention, none of the modules will finish. This condition of circular waiting is called *deadlock.*

As described by Hansen [Hansen73] and others, the following conditions are necessary for the occurrence of a deadlock:

(1) *Mutual exclusion:* A resource can only be used by one process at a time.

(2) *Non-preemptive scheduling:* A resource can only be released by the process which has acquired it.

(3) *Partial allocation.* A process can acquire its resources piecemeal.

(4) *Circular waiting.* The previous conditions permit concurrent processes to acquire part of their resources and enter a state in which they wait indefinitely to acquire each other's resources.

Deadlocks are *prevented* by ensuring that one or more of the necessary conditions never hold. In the linked module abstraction, conditions (1) and (2) cannot be easily defeated. By linking calling buffers as in the following figure, we can defeat circular waiting:

{Figure 17 Sharing a Module}

In this configuration, the calling modules emit tokens on their **Call** lines whenever they are ready. When the shared module is ready, the any-join selects one of them, emits a token to the shared module, and absorbs a token from the chosen calling buffer. (This allows the calling buffer to enter its *waiting for return* phase, and starts the shared module performing its computation. When the shared module finishes, it emits a token on its **Done** line.

This argument depends on the assumption that the computation in the shared module will terminate. We will examine some conditions bearing on this in the following sections.

At this point, only one of the calling buffers is in a *waiting for return* phase. (The others are in a *waiting for call* phase). Hence the calling buffer that was chosen by the any-join is the only one able to absorb the returned token. This buffer loads the data into its output buffer and absorbs the token. This allows the any-fork to absorb the token from the shared module, so that the shared module can return to the *waiting for go* state. The shared module does not need to wait and hold data for the calling module because each calling buffer buffers the output data. Hence, circular waiting is prevented.

The proof of non-deadlock for *linked Start/Wait* constructions with intermediate instructions is essentially the same, and is left for the reader.

Non-Recursion Rule

The previous argument for preventing deadlock using calling buffers presumes that the shared module will finish in a finite time. This conditiȯn will not be met if the shared module tries to call itself or the calling module before finishing. This leads us to the following rule:

A module cannot call itself recursively (directly or indirectly).

The notion of module *calling* in the linked-module abstraction involves the sending of data to modules along data lines. This contrasts with the *stack* implementation of calls in software for conventional computers, and illustrates why recursion is disallowed: there is no place to store the calling arguments and return addresses. It is possible to implement a stack-oriented notion of *function calling* in hardware. Section 4 proposes several alternative implementations of stacks.

To determine whether modules are recursive, it is too strong to recursively list the modules that are called by a module. For example, it is legitimate for F to call G and G to call F if the calls are mutually exclusive by the logic of the modules.

No Double *Starts*

Calling buffers preclude the possibility of the calling module trying to start the shared submodule again, before receiving back the return token. This is because a *Wait* immediately follows the *Start* in a calling buffer. If calling buffers are not used exclusively, it is possible to cause a deadlock by invoking two *Starts* to the same submodule without intermediate *Waits.* As in the recursion example, such constructions lead to a deadlock, with the calling module waiting to start the shared module and the shared module waiting on the calling module to accept the first return token.

2.7. Subsystems

Sometimes it is convenient to consider collections of modules and linking constructions as a unit, even when the unit itself is not strictly a module (i.e., it does not have a single **Go** line and **Done** line). We call such constructions *subsystems* and delineate them graphically by dashed lines instead of solid module boxes. Communication with subsystems is ultimately directed to the modules that constitute them. Subsystems can perform several operations at the same time. A precise characterization of subsystems and notations for them are given in Section 3.11. Several examples of subsystems are presented in Section 4.

3. A Programming Language for Linked Modules

This section proposes a linear notation, called the *LMA programming language*, for describing linked modules. The linear notation is a kind of *hardware programming language*. It expresses the same concepts as the graphical notation in the previous section, but it uses many familiar notations and constructions from conventional programming languages. We believe that the linear notation makes it convenient to create linked module descriptions of the architecture and behavior of digital systems.

In addition to notations for linked modules that correspond to the graphical notations, some programming notations for the convenience of the digital designer are proposed. These are used to specify parameterized modules, conditional parts, and information for testing and debugging the modules in a system.

For pedagogical purposes, this section is organized around examples. It begins with the use of program variables to represent input buffers and output terminals. Subsequently, notations for submodules, registers, conditionals, case statements, concurrency, indexed set constructions are developed.

3.1. Inputs, Outputs, Submodules, and Assignment Statements

The following example illustrates the correspondence between the linear and graphical notations for a module M. M has two inputs (x and y) and two outputs (a and b). M uses two submodules, F and G. F takes x as its argument and computes a; G takes y as its argument and computes b. F is to be computed before G. The definitions of F and G are given externally:

{Figure 21. Example of two sequential submodule calls.}

```
Module M
inputs      [x,y:  Bits[16]]
outputs     [a,b:  Bits[16]]
```

```
components [F1: F, G1: G]
action [
   b ← F1(x);
   a ← G1(y); ]
end Module M
```

The definition of module M is partitioned into several kinds of specifications: the *inputs*, the *outputs*, the *components*, and the specification of the *action* of the module. In analogy with conventional software, the *inputs* and *outputs* are given variable names. The input variables correspond to the input buffers of the module and can only be read; the output variables correspond to the output lines of the module and can only be written. Function call notation (i.e., subroutine calls) are used to indicate calls to submodules.

In this example, the definitions of F and G are given externally to the definition of M, but M has its own unshared copies of them (F1 and G1, respectively). The notation is meant to suggest that F1 is the local name of a copy of F, which is externally defined.

For brevity in this text, we will sometimes use the global name (e.g., F) to refer to either the local copy or the external definition. It should be clear from context which is intended.

Assignment statements are used to indicate where the values of functions go. In this example, the assignment statements are interpreted as the *wiring* of module output lines. When a module returns multiple outputs, they can be combined in a single assignment statement as follows:

```
<x, y, z> ← M (arg1, arg2, arg3);
```

In conventional software, the intention of the following sequence of statements is clear.

```
a ← 1;
...
a ← 0;
...
```

In the LMA programming language, we want to draw on programming idioms whenever possible. In this case, if a is an output terminal, the apparent graphical notation would result in a violation of the fan-in composition rule. The output terminal cannot be driven both to 1 and to 0. In a Section 3.5 we will introduce the notion of *implied registers* as an alternate interpretation of the assignment statement, which admits such programming idioms.

In some cases, it will be possible to *optimize* the specification by reducing the apparent number of registers. We are open to suggestions on this point.

3.2. Sequence and Concurrency

The serial listing of program statements indicates that the statements are activated sequentially in their order of appearance. In the linked module abstraction, parallel execution is achieved using all-forks. The following figure illustrates the graphical and linear notations for a module M in which submodules B, C, and D are activated in parallel. In this example, submodule A is first activated using w as its argument. Then output of A is given to B, C, and D and their outputs are used to drive **x**, **y**, and **z**, respectively.

{Figure 22. Example of parallel activation of submodules.}

```
Module M
inputs       [w: Bits[16]]
outputs      [x,y,z: Bits[16]]
wires        [v]
components   [A1: A, B1: B, C1: C, D1: D]
action [
   v ← A1(w);
   <& x ← B1(v), y ← C1(v), z ← D1(v) &>; ]
end Module M
```

The notation <& ... &> indicates an all/all fork-join for parallel execution. The notations for fork-joins are summarized in the following table:

Notation	Meaning
<& ... &>	non-synchronizing all/all fork-join
<&! ... &>	synchronizing all/all fork-join
<| ... |>	any/any fork-join
<& ... |>	all/any fork-join*
<| ... &>	any/all fork-join*

* These fork-joins are not token-conserving.

Another point of interest is the *wires* specification for v, the output of A and the input of B, C, and D. The *wires* notation indicates unambiguously that B, C, and D all use the output of the *same call* to A..

The order of activation can also be indicated in the linear notation by the embedding of submodule calls. The next example shows the graphical notation and two equivalent linear notations for a module M which computes F (G (H (x))), that is, F ∘ G ∘ H (x):

{Figure 23. Composed Functions.}

```
Module M  Version 1
inputs      [x: Bits[16]]
outputs     [a: Bits[16]]
wires       [y,z]
components  [F1: F, G1: G, H1: H]
action [
   y ← H1(x);
   z ← G1(y);
   a ← F1(z); ]
end Module M

Module M  Version 2
inputs      [x: Bits[16]]
outputs     [a: Bits[16]]
components  [F1: F, G1: G, H1: H]
action [
   a ← F1 ( G1 (H1 (x))); ]
end Module M
```

The first version of M uses the wire notation to indicate data flow and lists the submodules H, G, and F in the order of activation. The second version of M uses embedded submodule calls to indicate both data flow and the order of activation.

3.3. Case Statements and Conditionals

The select/any fork-joins in the linked module notation indicate selective branching in control flow. This construction is important for any kind of conditional branching and for the multi-function composite modules of Section 2.6. In the linear notation, this concept is expressed most familarly in case statements and conditional statements.

The following example illustrates a multi-function composite module which accepts two commands: **increment** and **decrement**. If the *command* line carries a datum corresponding to **increment**, then submodule F is activated. If the *command* line carries a datum corresponding to **decrement**, then submodule G is activated. The figure gives the equivalent graphical and linear notations for this:

{Figure 24. Example of a Case Statement.}

```
Module M
inputs       [x: Bits[16], command: Action]
outputs      [a: Bits[16]]
components   [F1: F, G1: G]
action [
   [Case command
          [increment {a ← F1(x)}]
          [decrement {a ← G1(x)}]   ]
end Module M
```

We use braces { ... } to delimit a sequence of statements.

The decoding of the command variable is not made explicit in this notation. We have chosen a syntax to make it similar to conventional *Case* statements and also recognizably close to *messages* in an object-oriented language. At issues is whether the language should be explicit about the representation of selection keys. We are open to suggestions on this point.

In this example, both commands to module M use the same input data: x. In some cases, different commands use different data. For example, in the following module, the **new** command uses x, and the **prev** command no input data at all. (It returns data received from another module.)

{Figure 25. Data Steering in a multi-function module.}

```
Module M
inputs       [x: Bits[16], command: Action]
outputs      [a: Bits[16]]
components   [Backup: GenBackup[16] ]
constants    [save, retrieve: Action]
action [
   [Case command
         [new  {a ← x; Backup(save, x)}]
         [prev {a ← Backup(retrieve)}] ]
end Module M
```

GenBackup is a *parameterized* module; **Backup** is an instance or local copy of this in which **16** is the value of the parameter. Parameterized modules are discussed in a later section.

In this example, the input **command** steers the control flow in the select-fork as well as the data flow of x in the select-data-fork. Hence, the output of M (that is, a) is x if the **command** is **new**, and is the value of the call to backup if the **command** is **prev**.

In the linear notation, we have our first example of *constants* defined for use in the module. In this case, they are of type *Action* and are used to indicate commands to **Backup**. The examples above show the *Case* statement as the first statement in a module. These examples correspond to the multi-function composite modules of Section 2.6, in which the commands share data but are mutually exclusive in execution. *Case* statements can also be used to represent selective braching at other (deeper) parts of a module definition.

Another programming construction which maps directly onto selection is the conditional statement. The following example illustrates the graphical and linear notations for a module, which computes F(y) if x=4, and G(y) otherwise:

{Figure 26. Example of a Conditional Statement.}

```
Module M
inputs      [x,y: Bits[16]]
outputs     [a: Bits[16]]
components  [EQUALS: NUMEQUALS [16]]
action [
   If (EQUALS x 4)
   Then a ← F(y)
   Else a ← G(y)  ]
end Module M
```

It is an open issue whether the syntax of our notation should provide special syntax for commonly used basic modules like EQUALS (e.g., an equals sign). We are open to suggestions on this.

If more than one statement is desired for the *Then* clause or the *Else* clause, then braces should be used as follows:

```
...
If    x = 4
Then  {a ← F(y); b←(G (H(x));}
Else  {a ← G(y); b←H(x);}
```

LMA language extends the *If* syntax to provide *ElseIf* clauses when the conditions are intended to be mutually exclusive:

```
If x < 10 Then a ← G(x)
ElseIf x < 20 Then a ← F(x)
ElseIf x < 50 Then a ← H(x)
Else a ← J(x) ;
```

3.4. Caller Constraints

When a module has several commands, it is often the case that the different commands use different subsets of the input data. For example, the module M defined in Figure 25 uses different data for the new and **prev** commands: the new command uses the input variable x, and the **prev** command uses no input variables. Calls to module M would accordingly have different forms according to the command. For example, a call to M invoking the new command would specify the variable as follows:

```
M (new 16);
```

and a call to M invoking the **prev** command would not specify the variable:

```
Temp ← M(prev);  .
```

By convention, calls lacking particular arguments correspond to graphical notations in which *no wires go to the corresponding input buffers.* The semantics of this is that the unwired input arguments are *not changed* by the call.

In some cases, the correct operation of a module for certain commands depends on the non-initialization of particular input variables. It is important that calls with these commands do not specify extraneous arguments. This is indicated in the LMA programming language by extending the notation of the *Case* statement to specify which input variables are expected for each command. The following figure illustrates this for M:

```
Module M
inputs       [x: Bits[16], command: Action]
outputs      [a: Bits[16]]
components   [Backup: GenBackup[16] ]
constants    [save, retrieve: Action]
action [
   [Case command
         [new  (x)     {a ← x; Backup(save, x)}]
         [prev ()      {a ← Backup(retrieve)}]] ]
end Module M
```

The semantics of the notation in this example is that calls specifying new as a command must provide x, and calls specifying **prev** must *not* specify any arguments (except command).

This notation forces a convention that all calls to a particular command must use the same arguments. It does not allow for optional arguments for a command. We are open to suggestions on this point.

In analogy with the specification of input data for commands, it is sometimes appropriate to specify output data for commands. A call to a module that does not return the appropriate output data is probably in error. This is illustrated in the following example:

```
Module M1
inputs       [x: Bits[16], command: Action]
outputs      [a,b: Bits[16]]
components   [Backup: GenBackup[16] ]
constants    [save, retrieve: Action, zero: Integer[0,16]]
action [
   [Case command
          [new  (x) → (b)      {Backup(save, x), b ← 0}]
          [prev () → (a,b)     {b ← Backup(retrieve), a ← zero}]]]
end Module M1
```

In this example, calls with the **new** command must provide the input argument **x** and return the output argument **b**; calls with the **prev** command must provide no input arguments, and must return the output arguments **a** and **b**.

The reader will note that the specifications for input and output variables are distributed among several declarations. The types of all the arguments are specified in the *inputs* and *outputs* specifications. The correspondence of the arguments with the module commands is indicated in the extended *Case* statement.

3.5. Registers

In the examples so far, the storage of information in modules has been only in their input buffers. For many kinds of hardware, it is useful to have local registers, in which data can be stored for later retrieval. Information in registers persists between activations of modules. This section shows how

(1) Registers can be implemented as modules.

(2) The implementation of registers can be optimized to be essentially null modules.

(3) The declarations at the beginning of a module can be extended to yield simplified specifications for registers.

(4) The assignment statement and variable notation can be extended to represent storage and retrieval from registers.

(5) The need for registers can sometimes be implicit in the use of assignment statements in a module.

The following graphical and linear notations show one way to create a register:

{Figure 27. Module for a register.}

```
*Module* Register16
*inputs*    [dataIn: Bits[16], command: action]
*outputs*   [dataOut: Bits[16]]
*action* [
    [*Case* command
        [read () → (dataOut)
            {dataOut←dataIn}]
        [write (dataIn) → ()
            ]]  ]
*end Module* Register16
```

This definition of a register depends on the convention that calls to read the register do not have wires to dataIn, that is, they do not change dataIn. Given this observation, it is possible to simplify the definition of a register to be essentially a null module as follows:

{Figure 28. Simplified version of LM register.}

Registers are used so frequently that it is convenient to have simple linear notations for defining them. In the linked module language, registers can be defined as parameterized components. For example,

components [R1: Register [16], R2: Register [8], R3: Bit]

defines three registers R1, R2 and R3 with lengths 16 bits, 8 bits, and 1 bit respectively.

In addition to declarations for registers, the LMA programming language provides simple notations for reading and writing registers. For example, the following two versions of module M both use register R1. The first version uses submodule notation for accessing the register; the second version uses an extended interpretation of the assignment statement:

```
Module M
inputs       [x: Bits[16], command: Action]
outputs      [a,b: Bits[16]]
components   [R1: Register[16]]
constants    [read,write: Action]

action [
   [Case command
      [regular (x) → (a) {
         a ← F( R1(read), x);
         R1(write x)}]
      [special (x) → (b) {
         b ← R1(read);
end Module M
```

```
Module M
inputs       [x: Bits[16], command: Action]
outputs      [a,b: Bits[16]]
components   [R1: Register [16]]

action [
   [Case command
      [regular (x) → (a) {
         a ← F(R1,x);
         R1 ← x}]
      [special (x) → (b)
```

```
        {b ← R1}]
end Module M
```

In the second version of M, the interpretation of the symbol R1 is interpreted as either a **read** or **write** command to the register, depending on context.

The next example illustrates the notion of *implicit registers*:

```
Module M
inputs      [x: Bits[16], command: Action]


outputs     [a,b: Bits[16]]
components  [R1: Register [16]]
action [
   [Case command
      [regular (x) → (a,b) {
         a ← F(R1,x);
         b ← x}]
      [special (x) → (b)
         {b ← R1}]
end Module M
```

In this example, the output variable **b** is set differently in the **regular** and the **special** commands. The straightforward mapping of this onto the graphical notation would have two data lines driving the **b** terminal -- resulting in a violation of the fan-in composition rule. A different interpretation of the linear notation preserves the programmer's intent without violating the rule. This interpretation is that the symbol **b** represents an *implicit register*, whose output is connected to M's output terminal. In this interpretation, the assignment statements simply load that register (instead of driving M's output lines).

Another interpretation would be a select/select fork-join with a data-join for switching the wires to output terminal **b**. This interpretation avoids the nead for a register. The choice between these interpretations may be considered to be the choice of the *optimizing compiler*.

The same issue of *fan-in* applies to *wires*. In this case, the interpretation of such wires as register may not preserve programmer's intent. We are open to suggestions about this issue.

3.6. Parameterized Modules and Conditional Parts

Parameterization is a syntactic device for defining generalized structures. For example, the following is a parameterized definition of register, in which the size of the register is specified as a parameter:

```
Module Register
parameters  [size: Integer]
```

```
inputs      [dataIn: Bits[size], command: action]
outputs     [dataOut: Bits[size]]
action [
   [Case command
      [read () → (dataOut)
         {dataOut←dataIn}]
      [write (dataIn) → ()
         ]] ]
end Module Register
```

Declarations of instances of a parameterized module must provide the parameters. For example, the *components* declaration:

```
components  [R1: Register[16], R2: Register[8]]
```

creates two instances of Register and specifies their lengths. In this example, the type of the parameter size is integer. Parameters can be of any legal type. For example,

```
parameters     [ErrorHandler: F]
```

specifies an instance of module F as a parameter, and

```
parameters     [MyKind: Type]
```

specifies a type as a parameter.

One of the important uses for parameters is for defining modules that have certain components conditionally, that is, depending on parameters. Conditional parts are indicated in the LMA programming language by *When* statements. A *When* statement has a similar syntax to the *If* statement, but it has a different meaning. A *When* statement is evaluated when an instance of the module is created. It indicates that certain parts of the linked module specification should be conditionally included in the instance. The *When* statement does not itself correspond to hardware. In contrast, an *If* statement is *evaluated* only when the module is *run*; the *If* statement corresponds to a select-fork in the instance. Both statements are illustrated in the following example:

```
Module BLine
parameters   [RNbr: BLine, Size: Integer,
               EndFlg: Boolean, ErrorHandler: BLineFull]
components:  [HasData, LastDatum: Bit, R1: Register[Size]]
...
action [
   [Case command
         [push (in) → ()
            {{When EndFlg
```

```
              Then ErrorHandler(Full)
              Else {If HasData
                   then LastDatum ← 0;
                        RNbr(push R1)
                   else LastDatum ← 1;}}
                   R1 ← In]}]
...
end Module BLine
```

In this example, the call to **ErrorHandler** is included in an instance only if the EndFlg parameter is true. Otherwise, the *If* statement with the **Has-Data** predicate is included.

3.7. Sets of Components and Indexed Selections

Many hardware systems are constructed out of collections of identical entities. In programming notations, it is convenient to define such collections as indexed sets, so that individual elements can be referred to by number instead of by name. This is particularly useful for defining parameterized modules, where the size of the collection is determined by a parameter. The syntax for defining indexed sets of identical modules is illustrated in the following example:

```
Module BQUEUE
inputs       [command: action, DataIn: Item]
outputs      [DataOut: Item]
parameters   [depth: Integer, Item: Type]
components   [QC: Set[i, [1..depth], QueueCell]]
...
```

In this example, QC is defined as an indexed set of QueueCells. The syntax indicates that there are **depth** elements in the set, where **depth** is a parameter supplied when the module is instantiated. The use of the parameter **i** is ilustrated below:

```
components   [QC: Set[i, [1..depth],
                  QueueCell [Item: Item,
                             LeftNbr: QC[i-1],
                             RightNbr: QC[i+1]]]
```

This example establishes **LeftNbr** and **RightNbr** parameters for each element of the set QC: the right neighbor is defined as the next element in the set. By convention, all indexing in sets is computed *modulo* the length of the set. This makes QC circular in that the right neighbor of **QC[depth]** will be **QC[1]**. To avoid the circularity, we can use a *When* statement as follows:

```
components  [QC: Set[i, [1..depth],
                 QueueCell [Item: Item,
                            LeftNbr: When i>1 Then QC[i-1],
                            RightNbr: When i<depth Then QC[i+1]]]
...
```

This example establishes **LeftNBr** and **RightNbr** parameters for each element of the QC set; the *When* statement precludes the first (i.e., leftmost) element from having a left neighbor, and the last (i.e., rightmost) element from having a right neighbor.

The LMA programming language also augments the notations for forks and joins to facilitate using them with indexed sets. The notation:

<& *Set* [i, [1..depth], QC[i](MoveRight)] &>

represents an all/all fork-join which sends a **MoveRight** command simultaneously to every element of the **QC** set. Analogously, the notation:

<| *Set* [i, [1..depth], QC[i](MoveRight)] |>

represents an any/any fork-join which sends the command to one element of the set. Finally, a select/any fork join is indicated as follows:

<*Select* [key←currentPos, QC[key](MoveRight)] |> .

The semantics of this are essentially the same as the semantics of a *Case* statement. If several statements are to be executed for a given value of the key, then the statements should be surrounded by braces as in the following:

```
<Select [key←currentPos, {x ← QC[key](read);
                          QC[key](push y)}] |>
```

3.8. Starts and Waits

Start and *Wait* are key words in the LMA language and used as follows:

Start [M(arg1 arg2), label]

<x, y, z> ← *Wait* [M, label]

where label is a unique label used to identify the pairing of *Starts* and *Waits*. M is the name of the module started (and waited for), arg1 and arg2 are calling arguments to M, and x, y, and z are the values returned by M.

3.9. Interrupts

Interrupt code is used to initialize modules. The interrupt code for a module is indicated in the *interrupt* specification as follows:

```
Module M
inputs       [x: Bits[16], command: Action]
outputs      [a: Bits[16]]
components   [F1: F, G1: G, R1: Bit]
action [
   [Case command
           [increment {a ← F1(x); R1 ← 1;}]
           [decrement {a ← G1(x); R1 ← 0;}]   ]
interrupt [
   [R1 ← 0]
end Module M
```

An *interrupt* signal to a module is generated by an *interrupt*, annotated as in the following example:

```
Module M
inputs       [x: Bits[16], command: Action]
outputs      [a: Bits[16]]
components   [F1: F, G1: G, R1: Bit]
action [
   [Case command
           [initialize () → ()  {Interrupt [F1]}
           [increment (x) → (a) {a ← F1(x); R1 ← 1;}]
           [decrement (x) → (a) {a ← G1(x); R1 ← 0;}]   ]
interrupt
   [R1 ← 0]
end Module M
```

In this example, M sends an interrupt to F1 if it receives an initialize command.

3.10. Preconditions and Postconditions

Argument checking is an important debugging concept from programming practice. This section introduces the analogous concept for hardware: providing specialized and separable code for testing module interfaces during simulation.

Arguments are interfaces for subroutines. By argument checking we mean the systematic checking of the validity of the values of arguments passed to a subroutine during its execution, to determine whether they are consistent with its purpose and limitations. By checking arguments while exercising a system on test cases, a programmer can often spot cases where subroutines fail to work correctly together. This technique is useful for debugging interfaces when parts of a program are written by different people. It is also useful when the specifications of interfaces for a system are in flux.

Repeated checking of arguments whenever subroutines are run introduces an overhead on the calling process. One way to avoid this overhead is to treat the checking code as *conditionally compiled*: so that it can be excluded when the subroutine is believed to be debugged, and reintroduced later only if the system is changed or if further debugging is needed. In hardware, the *overhead* of repeated checking might be measured in terms of the *silicon area* of the logic for doing the test.

The LMA programming language provides *preconditions* and *postconditions* specifications for checking the arguments and returned values of modules. This code can be run in simulations of a digital system as needed, but is not part of the specifications to be implemented as hardware. An example of such code for a stack follows:

```
Module Stack
parameters ...
components [FullFlag: Bit ...]
...
action [
   [Case command
         [push (Item) → () ...]
         [pop ...]
         [full? ... ]
         [empty? ... ] ]
preconditions [
   [Case command
         [push {If FullFlag
                  Then ReportError(NoRoom)}
...
End Module Stack
```

In this example, the protocol for calling the stack involves first testing for available space using a full? command. If this protocol is violated and a push command is issued when the stack is full, then the *preconditions* will detect and report the error. Presumably, examples of incorrect protocol will be detected by adequate testing before the stack is implemented. When the stack is implemented, the *preconditions* code can be factored out from the rest of the specification.

3.11. Subsystems

The examples so far have dealt only with modules. This section generalizes the module concept to the *subsystem* concept. Subsystems share some of the same properties and LMA programming notations.

The following table summarizes some of the differences between modules and subsystems:

Modules	**Subsystems**
Finish processing one set of inputs before starting another set.	Pipelined processing of inputs.
There is one **Go** line and one **Done** line per module.	Subsystems can have several **Go** and **Done** lines.
Commands are mutually exclusive.	Some commands can be executed simultaneously.
Commands correspond to cases in a select/any fork.	Commands correspond to either named **Go** & **Done** lines, or to module commands.

Because of the similar ideas for subsystems, it is convenient to have similar LMA notations. The following example of a subsystem illustrates many of these conventions:

```
Subsystem  FIFO-Scrambler
parameters  [Item: Type, Size: Integer]
inputs      [DataIn: Item, SFactor: Type]
outputs     [DataOut: Item]
components  [FC: Set [i, [1..size], FSC[item: Item,
                                    LNbr: when i>1 then FSC[i-1],
                                    RNbr: when i<size then FSC[i+1]]
             CP1: Compactor[Size]]
commands    [
     [scramble (SFactor) → () {<& Set [i, [1..size], FC[i](Scramble SFactor)] &>}
     [push     (DataIn) → () {FC[1](push DataIn); CP1(foo);}]
```

```
        [reset () → () {CP1(init)}]
        [undo () → () {CP1(undo)}] ]
end Subsystem FIFO-Scrambler;
```

This example illustrates several points:

(1) LMA notation uses the same declarations for *inputs, outputs, parameters,* and *components* for subsystems and modules. (This makes it possible to have local names for subsystem structure, analogous to the module substructure.)

(2) Instead of an *actions* specification, the submodule has an analogous *commands* specification.

(3) Each command is associated with a **Go** line. For example, the **scramble** command is connected to an all/all fork-join, and the **push** command is connected to two sequential calling buffers.

(4) Whether the commands can be executed simulaneously must be inferred from the specification. For example, since the **reset** and **undo** commands are both calls to **CP1**, they must be mutually exclusive. Either command can overlap execution with a **scramble** command, and they may partially overlap a **push** command.

4. Architectural Examples

A major goal of the linked module abstraction is to provide suitable terms for describing digital architecture. The LMA language describes the structure of digital systems in terms of modules and subsystems; it describes the paths that data can flow, the sequential and parallel activation of modules, and the placement of registers. This section illustrates some alternative architectures for familiar subsystems.

4.1. Stacks (LIFO)

The first set of examples are architectures for stacks. It might seem surprising that there are interesting or fundamental choices in the design of a stack. After all, a stack is a simple thing:

(1) It has two basic commands: **push** and **pop.**

(2) It has a simple *last-in-first-out* (LIFO) behavior.

(3) It has two error conditions: **push** on a full stack, and **pop** on an empty stack.

However, there are several properties of stack architectures that are not determined by or considered in this simple characterization of stacks:

(1) When can a stack respond to a **pop** after a **push**? (Must all of the effects of the previous **pop** be settled out?)

(2) During a **pop**, what moves? (Data or pointers or ... ?)

(3) How many interconnections are necessary between the stack storage cells?

(4) How many storage cells must the stack controller be connected to?

(5) How is the number of elements stored in a stack represented? (i.e., how is the *top* of the stack represented?)

(6) How many registers are required per stack cell?

(7) Does the time to **push** an element into a stack constant depend on how full the stack is?

In the architectural examples that follow, we will see different ways to answer these questions. Five architectures for stacks are considered:

(1) **Pointer Stack.** This stack corresponds to the usual *software implementation* of stacks, using an index into an array of registers. The index points to the top of the stack. **Pop** and **push** operations read and write data according to the index.

(2) **Roving Marker Stack.** Like the pointer stack, this architecture uses an indicator of the top of the stack. In constrast, it uses a *mark bit* associated with each cell instead of an index register to indicate the top of the stack. In a **push** or **pop** instruction, all of the cells receive the command but only the one with the mark bit set performs the operation. The marker bits are essentially passive bits that are set and reset by the stack cells.

(3) **Ganged Marker Stack.** Like the previous architecture, this one uses marker bits to indicate the top of the stack. In this architecture, however, the marker bits are themselves active and pass their contents simultaneously in the appropriate direction.

(4) **Buffered Stages Stack.** In the previous implementations, only the indicators changed on **push** and **pop** instructions to indicate the top of the stack. In this implementation, the top of the stack is always the left-most stack cell and all the data in the stack move simultaneously as required. Intermediate buffer stages are required to allow all of the data to move simultaneously. The controller sends its commands to all of the stack cells.

(5) **Ripple Stack.** Like the buffered stages architecture, this one moves the data on **push** and **pop** instructions. In contrast, this architecture requires only local interconnections between the stack cells and no additional buffers between them. A **push** command from the controller starts at one end of the stack; the required movement ripples left to right through the stack.

4.1.1. The Pointer Stack

The pointer stack is the architecture most often implemented in software using random access memory. A register containing the address of the top of the stack is used to index into the memory. The following figure shows the basic interconnections and an LMA description of it.

{Figure 29. Pointer Stack.}

```
Module PointerStack
inputs [command: action, dataIn: item]
outputs    [dataOut: item]
parameters    [depth: Integer, item: Type]
components    [pointer: Register[addressWidth(depth)]
    --addressWidth(n) computes the number of bits necessary to address n items.
                    R: Set[i, depth, Register[item]]]
    --We use Register[item] as an abbreviation for Register[Size(item)]
action  [Case command
    [push  [if pointer<depth
              then {select[key←pointer | R[key]←dataIn];
              pointer←pointer+1}]]
    [pop      [if pointer>1
                 then {pointer←pointer-1;
                    <select key←pointer; dataOut←R[key] |>}]]]
interrupt  [{pointer←1}]]
end Module PointerStack
```

The **push** command loads the data, and then advances the pointer. The **pop** command decrements the pointer and then unloads the data. Because **PointerStack** is a module, the **pus** and **pop** commands are mutually exclusive. Addressing of memory registers is done with a select/any fork-join. The need for an adder (or at least a reversible counter) in the module is implicit in the "+" operation.

4.1.2. Roving Marker Stack

In this implementation, each stack cell has a marker bit. The cell containing the data at the top of the stack has its marker bit set to 1. On a **push** or **pop**, the marked storage cell requests its appropriate neighbor to set its marker bit. Since a marker bit is shifted to move the top of the stack, no adder is needed in this architecture. Commands are sent to all of the cells, but only the marked one needs to act. The following figure shows the basic connections in the graphical notation, and a detailed LMA program for the stack controller and for a cell with marker bit.

{Figure 30. Roving Marker Stack.}

```
Module RovingMarkerStack
inputs       [command: action, dataIn: item]
outputs      [dataOut: item]
parameters   [depth: Integer, item: Type]
constants    [push, pop, setBit, clearBit: Action]
components   [outReg: Register[item],
              MC: Set[i, depth,
                     MarkedCell[LNbr: when i>1 then MC[i-1],
                                RNbr: when i<depth then MC[i+1],
                                 out: outReg]]]
action  [Case command
    [push {<&! Set[i, depth, MC[i](push, dataIn) &>}]
    [pop        {<&! Set[i, depth, MC[i](pop)] &>;
                 dataOut←outReg}]
interrupt  [{<& Set[i, depth, MC[i](clearBit)]>;
                 MC[1](setBit)}]]
end Module RovingMarkerStack


Module  MarkedCell
inputs       [command:  action,  dataIn:  item]
parameters   [item:  Type,  LNbr:  MarkedCell,  RNbr:  MarkedCell,
              out:  Register[item]]
components   [myBit:  Bit,  dataReg:  Register[Item]]
action     [Case  command
    [push         {if myBit=1  then  {dataReg←dataIn;  MyBit←0;  RNbr(setBit)}}]
    [pop          {if myBit=1  then  {myBit←0;  LNbr(popValue)}}]
    [popValue     {myBit←1;  out←dataReg}]
    [setBit       {myBit←1}]
    [clearBit     {myBit←0}]]
end  Module  MarkedCell
```

In this architecture, the **push** and **pop** commands are sent from the controller to all of the MC's simultaneously using the synchronous all/all fork-join. The MC's are initialized so that only one of them has its **myBit** set to one, and this condition is preserved by the operations. The marker bit is set to 1 in the cell after the last datum. Items are passed between the marked cells and their controller is though the shared register **outReg**, which is passed as a parameter to the cells.

A variation on this architecture would avoid the input buffer dataIn in each marked cell by passing the information in a shared register specified as a parameter.

4.1.3. Ganged Marker Stack

This is another version of a moving marker stack, except that the all of the marker bit cells are moved, instead of just the top ones. They are shifted simultaneously by a command from the controller. The following figure shows a partial specification in the graphics notation and the LMA program language description for a ganged marker stack:

{Figure 31. Ganged Marker Stack.}

```
Module GangedMarkerStack
inputs       [command: action, dataIn: item]
outputs      [dataOut: item]
parameters   [depth: Integer, item: Type]
constants    [moveRight, moveLeft: Action]
components   [CS: CircularShifter[depth], outReg: Register[item],
                Data: Set[i, depth, Register[item]]]
action [Case command
    [push    {<& Set[i, depth, if CS(read i)=1 then Data[i]←dataIn]] &>;
             CS(moveRight)}]
    [pop     {CS(moveLeft);
             <& Set[i, depth, if CS(read i)=1 then dataOut←Data[i]] &> }]]
end Module GangedMarkerStack


Subsystem CircularShifter
parameters   [length: Integer]
components   [B: Set[i, length, Bit]]
commands [
    [moveRight () → ()         <& Set[i, length, B[i+1]←B[i] ] &>]
    [moveLeft () → ()          <& Set[i, length, B[i]←B[i+1] ] &>]
    [read (Index) → (dataOut)  <Select key←Index, dataOut ← B[key]|> ] ]
    [write (dataIn, Index) → () <Select key←Index, B[key] ← dataIn |>]
interrupt  [<& Set[i, length, B[i]←0 ] &>;
             B[1]←1]]
end Subsystem CircularShifter
```

The **push** and **pop** commands translate into all/all fork-joins containing select/any fork-joins for the conditional statements. Since all of the assignment statements in the **pop** case drive **dataOut**, it must be treated as an implicit register as in Section 3.5. In the **CircularShifter** subsystem, the "+" is a *compile-time* operation -- it does not map into adders, but into the relative wiring of the B[i]'s through calling buffers. Because of the buffering provided by the calling buffers themselves, it is not necessary to specify intermediate buffering betweeen the B[i] for the circular shifter.

4.1.4. Buffered Stages Stack

The organization of this stack differs from the previous stacks in the decision of what to move during stack operations. In the previous architectures, the *top* of the stack moves, but the data are stationary in the cells; in this architecture, the top of the stack is stationary but the data in the stack move. All of the cells pass data simultaneously, through *buffers* between the cells. The following figure shows a partial graphical description and the LMA programming descriptions of the stacks. The **LR** are the stack cells, the **BR** are the buffers, and the **BufferedStagesStack** is the controller:

{Figure 32. Buffered Stages Stack}

```
Module BufferedStagesStack
parameters  [depth: Integer, item: Type]
inputs [command: action, dataIn: item]
outputs   [dataOut: item]
constants [moveRight, moveLeft, getRight, getLeft: Action]
components  [dataReg: Register[item],
             LR: Set[i, depth,
                    LinkedRegister[item: item
                                   LNbr: when i>1 then BR[i-1]
                                                  else dataReg,
                                   RNbr: when i<depth then BR[i]]
             BR: Set[i, depth-1, Register[item]]
```

```
action [Case command
    [push  {dataReg←dataIn;
            <& Set[i, depth, LR[i](moveRight) &>;
            <& Set[i, depth, LR[i](getLeft)} &>}]
    [pop   {<& Set[i, depth, LR[i](moveLeft)] &>;
            <& Set[i, depth, LR[i](getRight)] &>;
            dataOut←dataReg}]
end Module BufferedStagesStack

Module LinkedRegister
inputs        [command: action]
components    [data: Register[item]]
parameters    [item: Type, LNbr: Register[item], RNbr: Register[item]]
action      [Case command
    [moveRight  () → ()    RNbr←data]
    [moveLeft   () → ()    LNbr←data]
    [getRight   () → ()    data←RNbr]
    [getLeft    () → ()    data←LNbr]]
end Module LinkedRegister
```

An interesting observation about simultaneous shifting of data in a series of modules is that some buffers other than the input buffers must be provided to hold the data. In this architecture, intermediate registers BR[i] have been used to buffer the data between the LR[i]. On a **push** command, the controller issues a sequence of two commands to the LR[i], telling them to first send data to the BR[i] on the right, and to then read the data from the BR[i-1] on the left. For proper operation, all of the **moveRight** operations must be complete before the **getLeft** operations are started. (On **pop** commands, the order is correspondingly reversed.) The wiring of the BR and LR is accomplished in the *components* specification of the controller.

We could have used calling buffers as implicit buffering between the LR[i]. In this program, the calling buffers provide no buffering because the actions in the LinkedRegister specify that no *inputs* are used for any of the commands.

4.1.5. Ripple Stack

This is another version of the stack that moves the data and uses the leftmost cell as the top of the stack. In contrast with the previous architecture, this one moves the data one at a time (rippling through the stack), and does not need intermediate buffers between the stack cells. It requires half as many registers as the previous version of the stack, but requires time proportional to the current depth of the stack.

```
Module RippleStack
inputs        [command: Action, dataIn: item]
outputs       [dataOut: item]
parameters    [depth: Integer, item: Type]
constants     [moveRight, startMoveLeft, pop, store: Action]
components    [ DC: Set[i, depth,
                    DataCell[item: item
                                  LNbr: when i>1 then DC[i-1],
                                  RNbr: when i<depth then DC[i+1]]]]
action [Case command
   [push  { DC[1](moveRight); DC[1](store, dataIn)}]
   [pop   {dataOut←DC[1](pop); DC[depth] (startMoveLeft)}]
interrupt  [<& Set[i, depth] {DC[i](store, 0); DC[i](pop)}] &>}]]
end Module RippleStack


Module DataCell
inputs       [command: action, dataIn: item]
outputs      [dataOut: item]
parameters   [item: Type, LNbr: DataCell[item], RNbr: DataCell[item]]
components   [Filled: Bit]
action       [Case command
   [moveRight      () → ()     {if Filled=1
                                then RNbr(moveRight);
                                RNbr(store, dataIn)}]
   [startMoveLeft  () → ()     {if Filled=0 then LNbr(startMoveLeft)
                                else {LNbr(moveLeft);Filled←0}}]
   [moveLeft       () → ()     {if Filled=1
                                then LNbr(moveLeft);LNbr(store, dataIn)}]
   [store          (dataIn) → ()    {-- no code needed, data stored in dataIn}]
   [pop            () → (dataOut)   {dataOut←dataIn; Filled←0}]]
end Module DataCell
```

This architecture has the interesting property that the controller is only connected to the first and last stack cells. The reader is invited to step through the operation of a push instruction to see how the control sequencing avoids the use of intermediate buffering.

4.2. Queue (FIFO)

4.3. Content Addressable Memory (CAM)

4.4. The TransQueue (XQ)

4.4.1. Implementation using a CAM

4.4.2. Implementation using Dual-Shift Comparators

4.5. Bidirectional Queue (2Q)

{To be written}

5. Engineering the Methodology

We define a *design level* (or *design space)* to be a set of terms describing objects at some level of detail and composition rules for putting them together. For digital systems, the terms describe digital components, which may be primitive switches or blocks of combinational logic depending on the level. Each level should deal a coherent set of related issues. Systems constructed according to the composition rules should be *correct* in some sense.

Table I summarizes our current experience with creating design levels for designing digital systems. As described in Section I, the layout level is concerned with the constraints imposed by the properties of silicon and the fabrication process. The composition rules are the *lambda* design rules. The CPS level describes circuits in terms of pull-ups, pull-downs, and pass transistors. It is concerned with preserving the *digital character* of the system [Bell81]. The CRL level is concerned with clocking.

Table I. The partitioning of design concerns

Description Level	Concerns	Composition Rules	Examples of Bugs
Layout	Physical dimensions	Lambda Rules	Separation Errors
CPS	Digital behavior	Composition of PUEs, PDEs, and pass transistors	Charge Sharing Switching Levels Threshold drops
CRL	Clocking	Stage composition	mixed clocks unclocked feedback
LMA	Event Sequencing	forking, joining, module composition	deadlock data not ready

A key observation is that these design spaces partition the concerns of a designer. They provide a systemmatic way of delaying the consideration of certain issues. When we started fashioning the LMA level, we knew that we wanted to focus on the behavioral and architectural aspects of digital systems. In the the LMA level,

(1) The chief behavioral notion is that behavior is described by a partially ordered set of computational events. (The LMA model describes these in terms of modules which define the events, and forks and joins which

determine the sequencing.)

(2) The chief structural notions are the explicit indication of storage (registers) and the explict connection of subsystems by data and control lines.

It is our belief that these notions usefully characterize *digital architecture.* Hence, in the Section 4 the five LMA descriptions of stacks are *not equivalent.* They differ precisely in their interconnection of cells, in the number of registers needed, and in their performance as determined by the predictable patterns of control.

(3) The chief notions of correctness in the LMA model are (a) the linking of data and control that insure that modules cannot begin until their data are available and (b) the avoidance of deadlock in the use of shared modules.

Thus, LMA descriptions composed according to the rules are, in this specific sense, correct. The notion of correctness is limited in each of the levels described in Table I. The layout level descriptions are correct in that they can be fabricated; the CPS descriptions are correct in that they have digital behavior; the CRL descriptions are correct in that they have no clocking errors; the LMA descriptions are correct in that they admit no deadlock and they unambiguously define which data are to be used in each computation.

Within a description level, tradeoffs must be made between the coverage and simplicity of the methodology. An example of this in the LMA level is the specification of a *select fork.* The following possible criteria were considered in the definition of the behavior of a select fork:

Selection Criteria
1. Outside selection by key
2. Self-selection by ready status
3. Priority or precedence rules

Synchronization Criteria
1. all at once
2. Individually when ready

Termination Criteria
1. At least N activated
2. No more than N activated

The criteria incorporated into the LMA model were:

Selection Criterion
1. Outside selection by key

Termination Criteria

1. At least 1 module activated.
2. No more than 1 module activated.

Other combinations offer increased coverage at the expense of the simplicity of the LMA composition rules. For example, if we allow more than one module to be selected by ready status, a select fork would yield an unpredictable or timing dependent multiplication of tokens. This would routinely require the use of token absorbers (see Section 2.3) to build modules satisfying the token conservation principle. Such modules are relatively baroque; our attitude is that such modules are possible, but they lie outside the coverage of the simpler designs, which we believe are substatially easier to understand and to create.

In the engineering of *notations*, our concern was with the verbosity of the specifications and with the applicability of familiar idioms. The graphical notation provided us with rich descriptions, but the graphical description of a real circuit (even the stack examples) yields a veritable spaghetti of connections. A linear notation based on the graphical notation, which names all of the connections between modules, suffers the same problem. Furthermore, such a linear notation has the interesting property that the lines can be sorted in any order without changing the meaning; the meaning is determined solely by the connectivity. We found such descriptions very hard to understand.

The proposed LMA programming language avoids these pitfalls as follows:

1. Most of the control lines can be elided using the composition rules about fan-out and the convention that sequential statements correspond to sequential executions.

We found that this convention works just as well for a *multiple program counter* language as it does for a conventional *single program counter* language.

2. The use of indexed set notation and parameterization made it possible to create generic modules.

3. By extending the meaning of the assignment statement, we were able to express a variety of constructions (wiring, connections to buffers, connections to registers).

These conventions made it possible to carry over a set of *programming idioms* from conventional programming languages -- lending a sense of familiarity to LMA programs.

In general, we feel that we have captured a coherent set of concepts and notations for describing digital architecture. Our next step is to experiment with these notations and ideas, to stress them by running them against real problems and examples. We are planning to do this in the context of a knowledge-based system containing the composition rules. Such a system will be both an experimental design aid, and a vehicle for testing design methodologies.

6. Relationships to Other Work

{This section to be expanded in later drafts.}

6.1. On the Linked Module Formalism

The work reported here draws on ideas from both computer science and electrical engineering. A wide variety of *hardware description languages* (See van Cleemput [vanCleemput79] for a survey) have provided terminology for describing and abstracting hardware. Our own work considers not only on how to describe hardware, but also how to synthesize descriptions that are free of certain classes of bugs.

Work on *models of computation* is also relevant. The work of Keller [Keller74] anticipates our notions of forks and joins, and of speed independent modules.

Seitz's work on *self-timed systems* [Seitz80a and Seitz80b] offers a state of the art analysis of such systems. Our LMA modules, however, are intended only as an abstraction for specifying intended behavior. Implementation of LMA descriptions may be in terms of synchronous systems.

The work on Petri nets [Peterson77] directly influenced our choice of tokens as a means of characterizing control flow. MacQueen [MacQueen79] provides a thorough review and bibliography of Petri nets and other computational models for distributed computation. Descriptions of hardware systems should probably be somewhat more specialized. For example, they need not admit the *creation* of processes (i.e., hardware does not usually create new hardware as it runs.)

Finally, work on the design of operating systems and the design of languages for multi-processing has been relevant, especially with regard to the phenomena of deadlock in the use of shared resources. A good tutorial for this material is Hansen [Hansen73].

6.2. Toward a Science of Design

We view the systematic creation of design methodologies and experimentation with them to be the core of our research. This memo is our second working paper describing tradeoffs in creating methodologies (See [Bell81]).

Research into the design process is a somewhat *soft science*. Design *as taught* is a branch of engineering, and design *as practised* is more akin to an art form than to a science. Attempts to characterize design as a science of *optimization* (e.g., dynamic programming methods) have had rather narrow success, and no detectable impact on the design of digital systems.

Simon [Simon81] has suggested ways of understanding design as a *satisficing* process, that is, a process using limited resources of finding solutions that are *good enough.* His observations about the role of hierarchical systems for coping with complexity have been influential in artificial intelligence research for years. Our own work complements this by seeking to understand how hierarchical systems can partition the concerns of a designer.

Our experimental approach to studying the knowledge of design using a knowledge-based computer system, has been strongly influenced by the work of Feigenbaum [Feigenbaum77] and his colleagues in the *Heuristic Programming Project* at Stanford University. A survey of the work on expert systems is available [Stefik81b].

7. Status of this Document

This memo reports on work that is still in progress. Suggestions and criticisms are solicited. Several sections of this memo are still to be written. An experimental system based on these ideas is under construction. It will be some time before we can evaluate these ideas in the light of experience with design of digital designs.

The ideas reported here have gone through several stages of refinement and revision. The proposed methodology and notations are the result of the combined efforts of both authors. The task of writing this memo has fallen mostly on Mark Stefik. The figures were debugged and prepared by Steve Kaufman.

Several colleagues have contributed thoughts along the way. Dick Lyon encouraged us to be precise about the description of the computational phases of modules. The incorporation of preconditions and postconditions was inspired by the suggestions of Nori Suzuki and Rod Burstall, and their use of *guardians* in their *Sakura* language. The characterization of *subsystems* was the result of Harry Barrow's persistent offering of examples that could not be reasonably captured in the formalism. The work has driven and shaped by collective thoughts of our collaborators at Stanford University on the KB-VLSI project. Thanks also to Lynn Conway for suggestions and encouragement, and for providing an environment in which this effort could flourish.

8. Bibliography

[Bell81]
Bell, A., Stefik, M., and Conway, L. *The deliberate engineering of methodologies for integrated system design*, Knowledge-Based VLSI Design Group Memo KB-VLSI-81-3 (working paper), April 1981.

[Feigenbaum77]
Feigenbaum, E. A., The art of artificial intelligence: themes and case studies of knowledge engineering, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, MIT, 1977., pp. 1014-1029.

[Hansen73]
Hansen, P. B., *Operating System Principles*, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1973.

[Keller74]
Keller, R. M., Towards a theory of universal speed-independent modules, *IEEE Transactions on Computers*, C-23:1, January 1974.

[Lyon81]
Lyon, R. F., Simplified design rules for VLSI layouts, *Lambda*, First Quarter 1981, pp. 54-59.

[MacQueen79]
MacQueen, D. B. *Models for distributed computing*, Institut de Recherche d'Informatique et d'Automatique, Rapport de Recherche N. 351, April 1979.

[Mead80]
Mead, C., and Conway, L. *Introduction to VLSI Systems*, Addison-Wesley Publishing Company, 1980.

[Peterson77]
Peterson, J. L., Petri Nets, *Computing Surveys*, 9:3, September 1977.

[Seitz80a]
Seitz, C. L., System Timing, in Mead, C. and Conway, L., *Introduction to VLSI Systems*, Reading, Mass.: Addison-Wesley Publishing Company, 1980, Chapter 7.

[Seitz80b]
Seitz, C. L., Ideas about arbiters, *Lambda*, First Quarter 1980, pp. 10-14.

[Simon81]
Simon, H. A., *The sciences of the artificial*, Cambridge, Massachusetts: The MIT Press, (second edition) 1981.

[Stefik81a]
Stefik, M. and Brown, H. *Toward a prototype of Palladio*, Knowledge-Based VLSI

Design Group Memo KB-VLSI-81-6 (working paper), May 1981.

[Stefik81b]
Stefik, M., Aikins, J., Balzer, B., Benoit, J., Birnbaum, L., Hayes-Roth, F., Sacerdoti, E. The architecture of expert systems: a guide to the organization of problem-solving programs, *to appear as Chapter 3 in* Hayes-Roth, F., Waterman, D., & Lenat, D.B. (Eds.), *Building Expert Systems*, 1982.

[Tong81]
Tong, C. and Stefik, M. Knowledge for transforming switch-level circuit representations to circuit layouts, Knowledge-Based VLSI Design Group Memo KB-VLSI-81-1 (working paper), March 1981.

[vanCleemput79]
vanCleemput, W. M., *Computer-aided design tools for digital systems*, IEEE Catalog No. EHO 132-1, 1979.

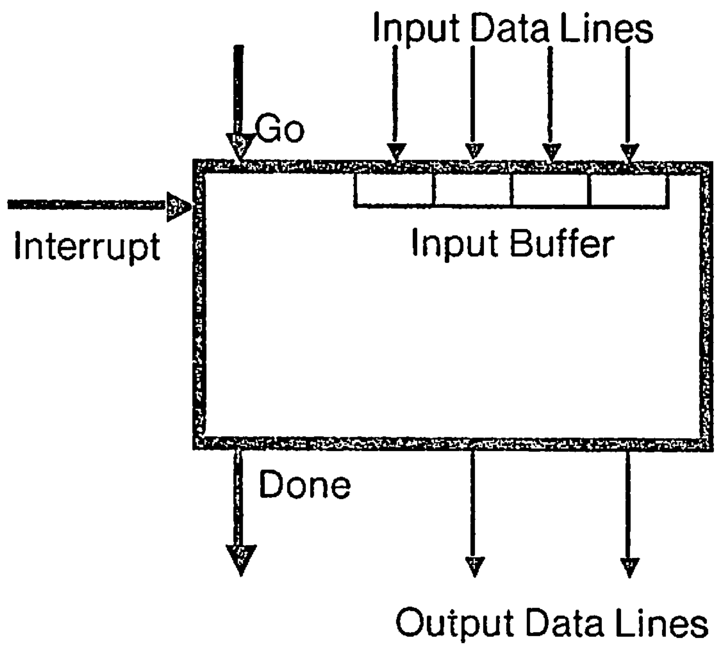


Figure 1: Basic Module

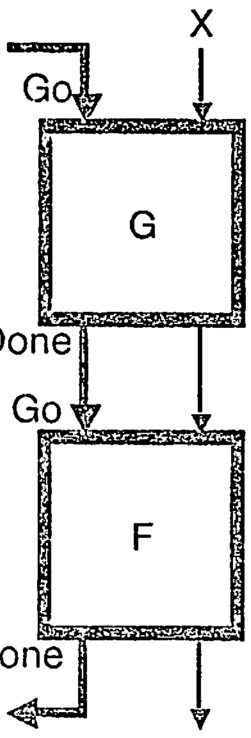


Figure 2:
Composed Functions

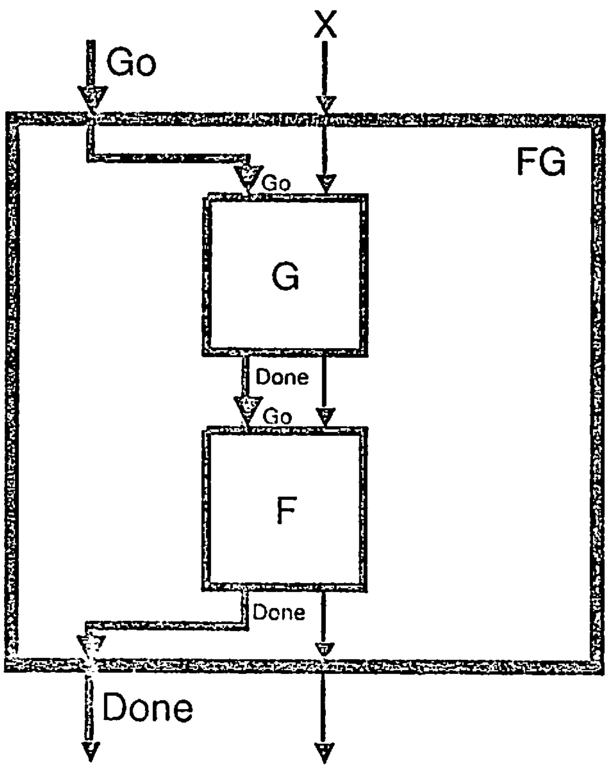


Figure 3: FG

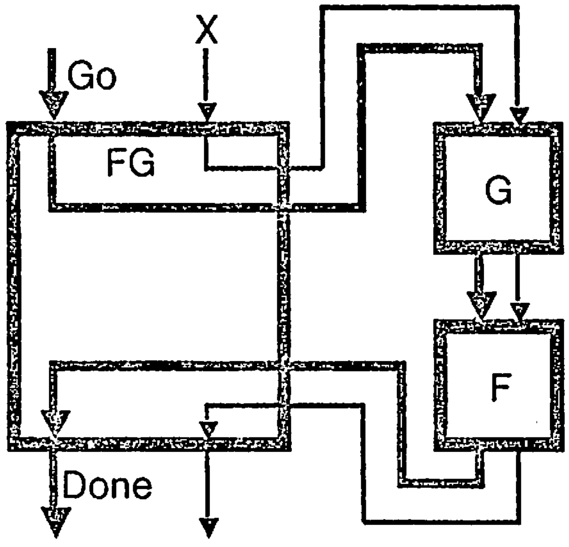


Figure 4: FG redrawn

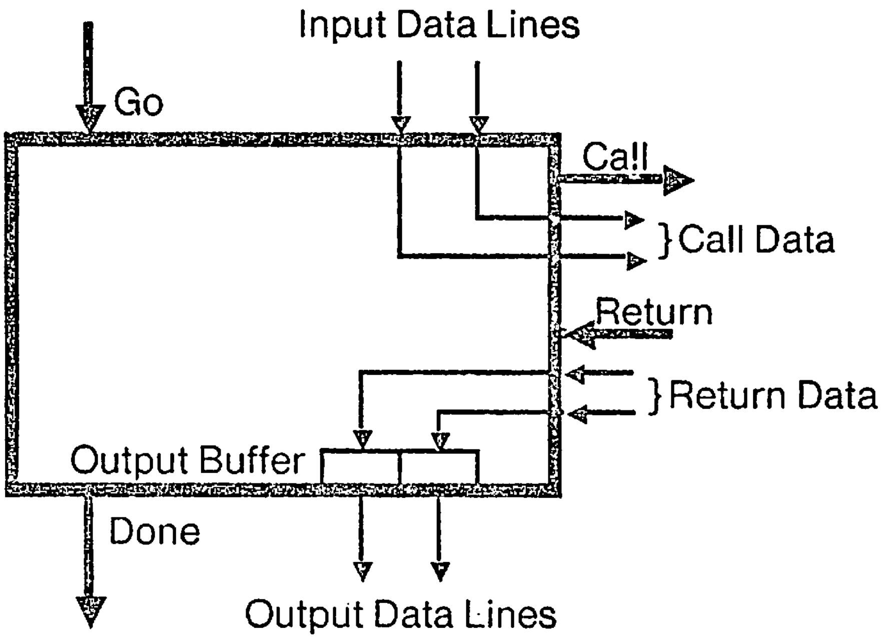


Figure 5: Calling Buffer

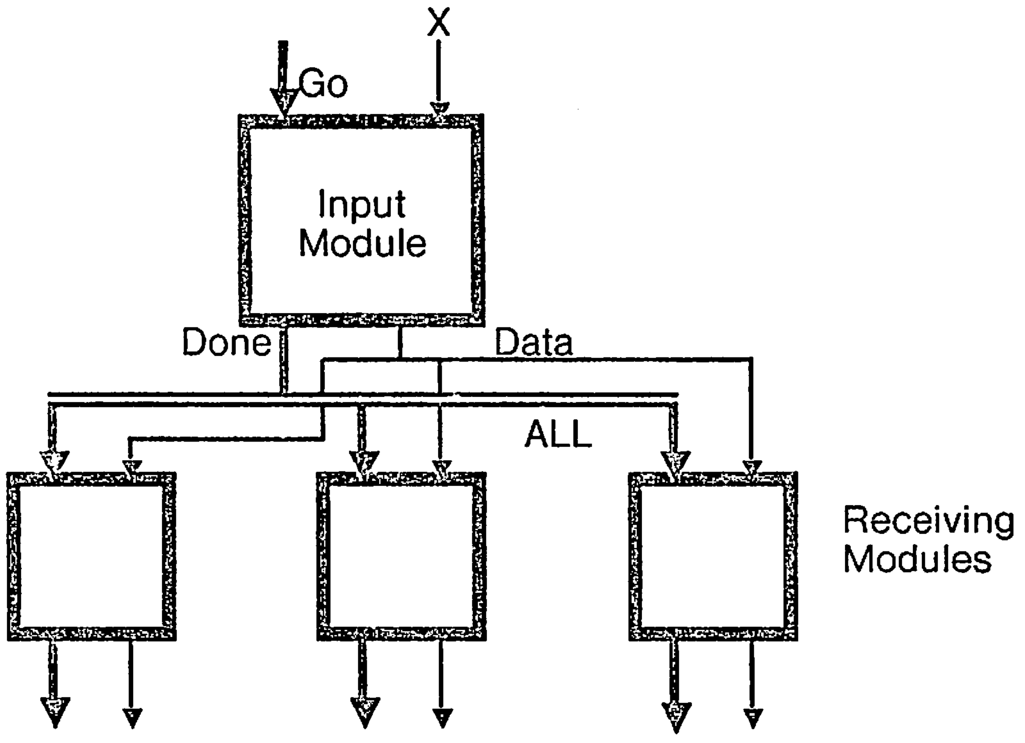


Figure 6: ALL-FORK

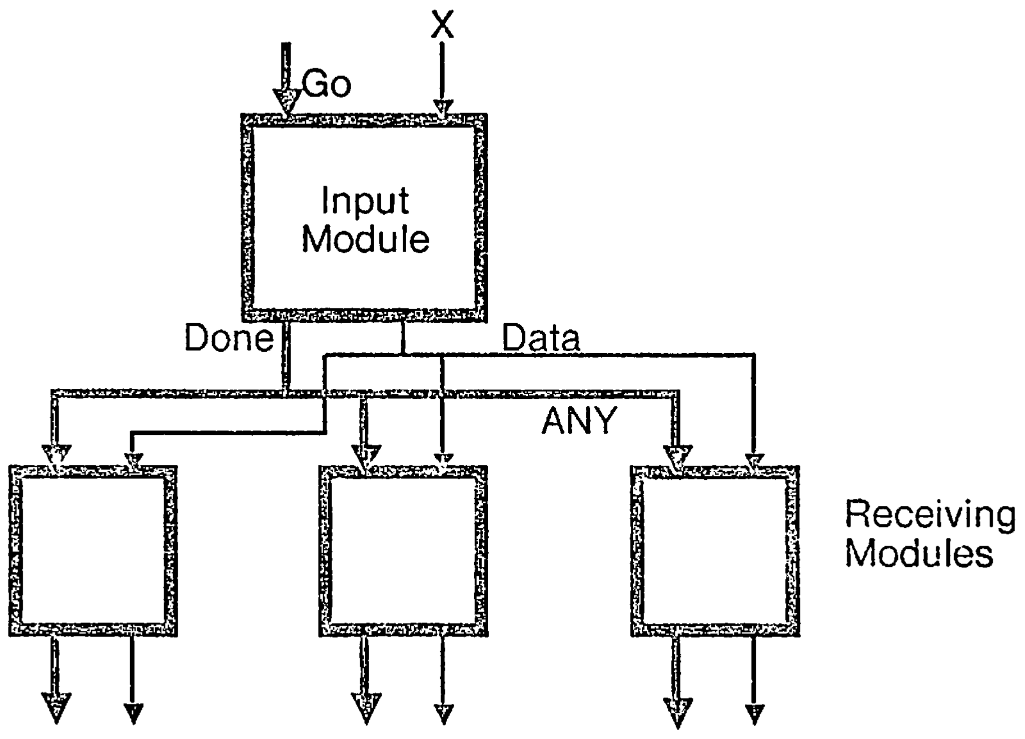


Figure 7: ANY-FORK

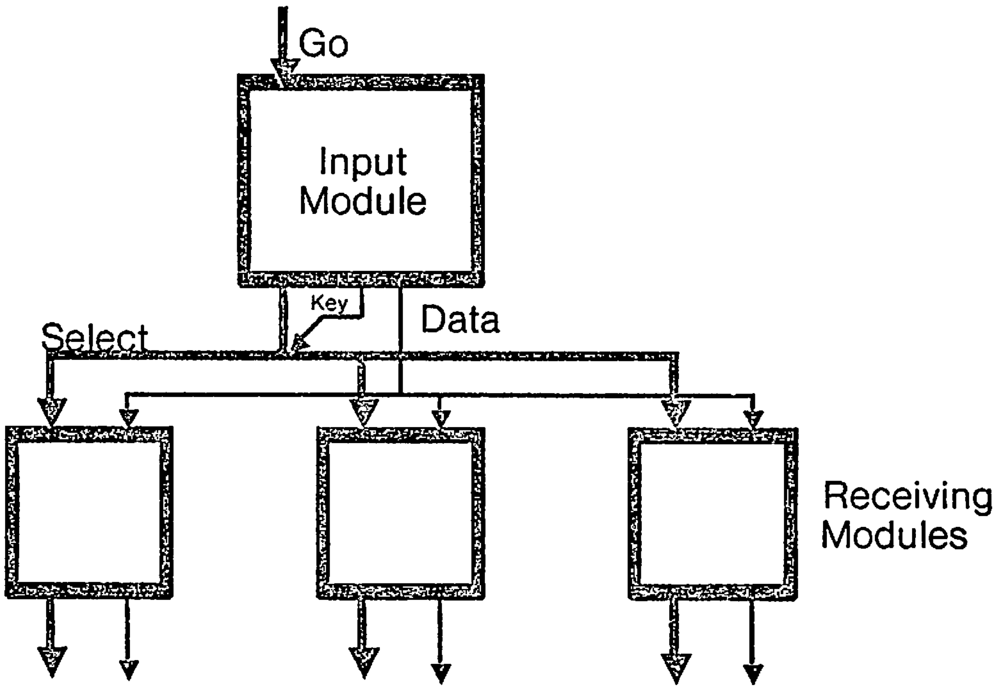


Figure 8: SELECT-FORK

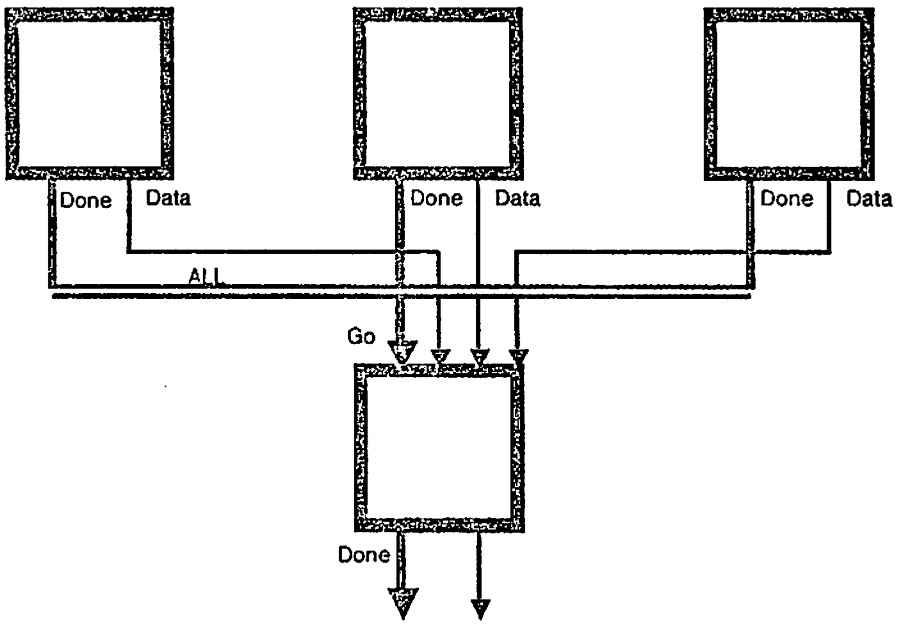


Figure 9: ALL·JOIN

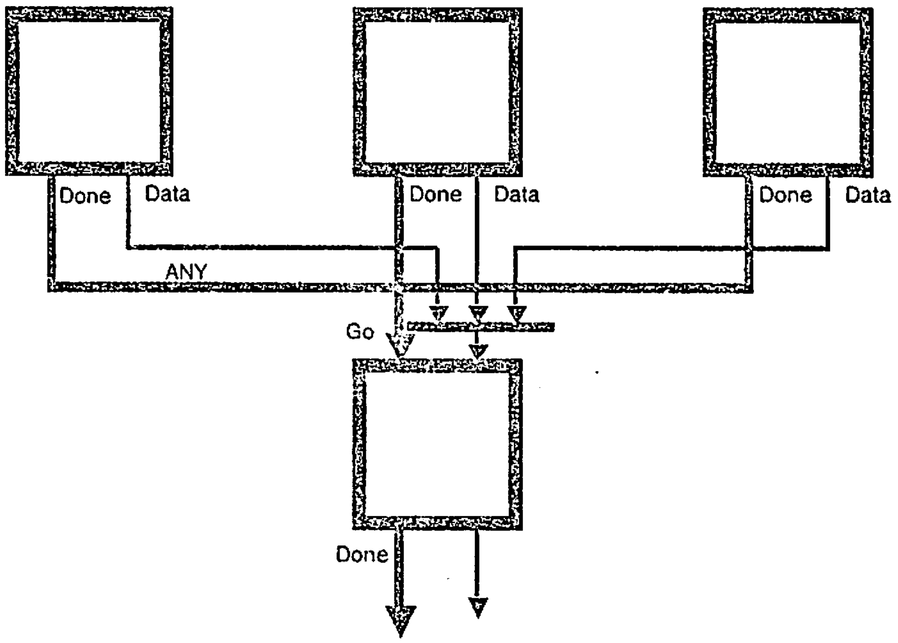
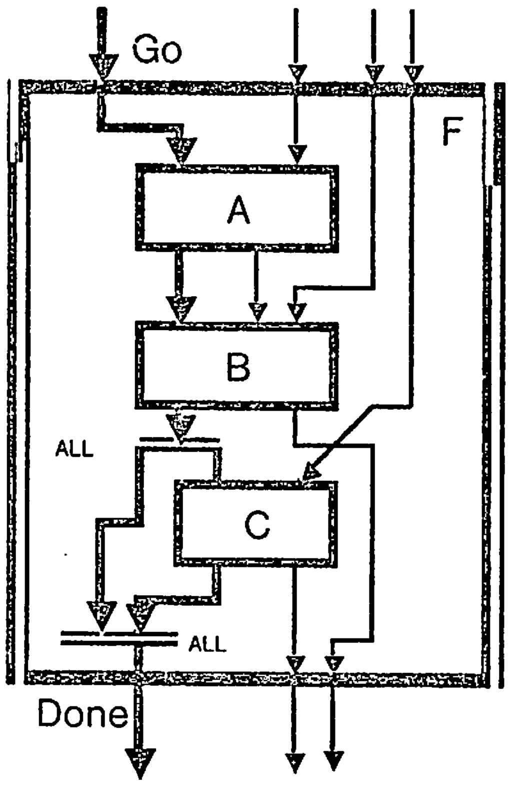
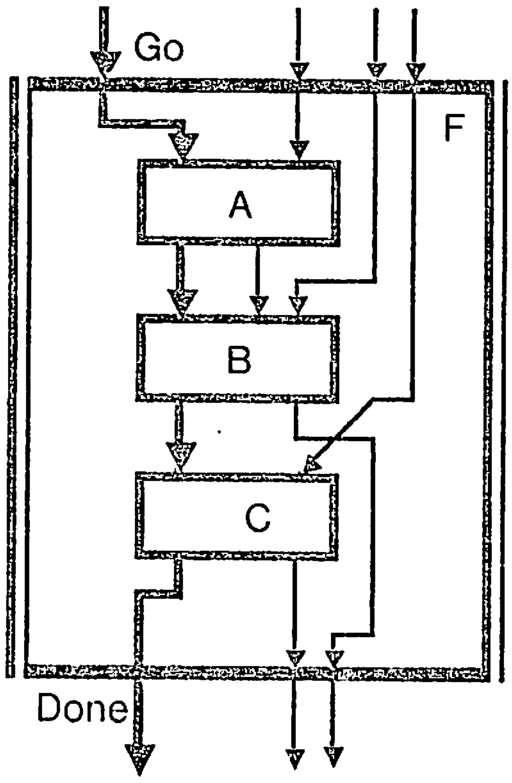


Figure 10: ANY·JOIN

a) General Form



b) Optimized Form



Figur e 11: Sequential Instructio1

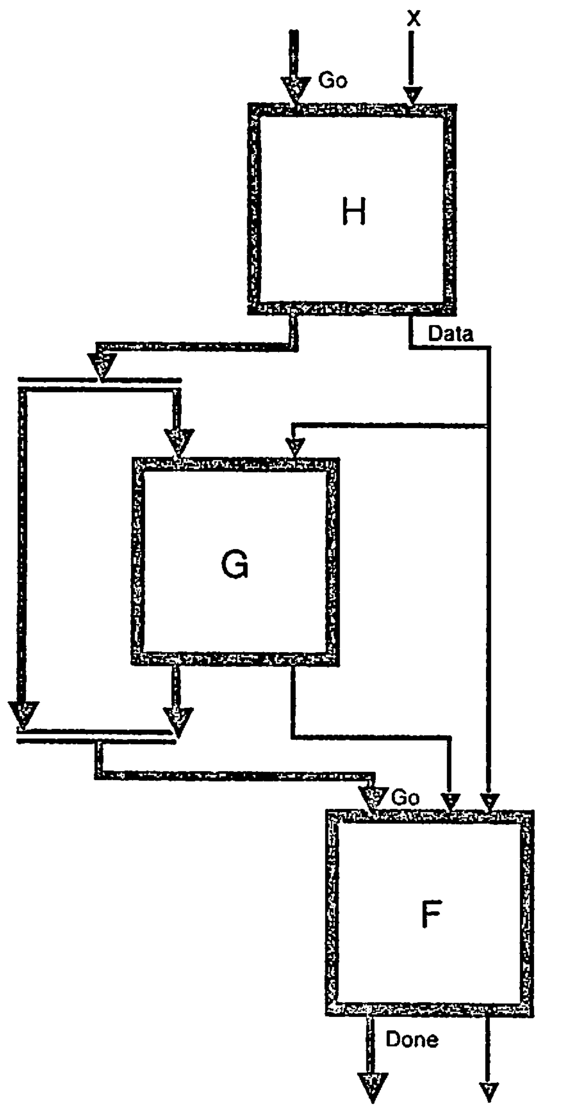


Figure 12: F(G(H(x)), H(x))

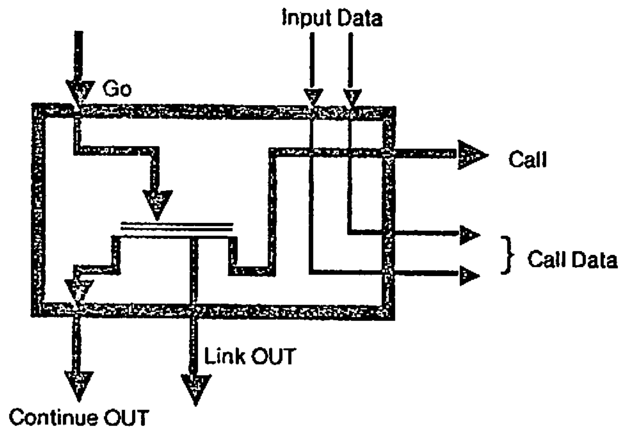


Figure 13: START Construction

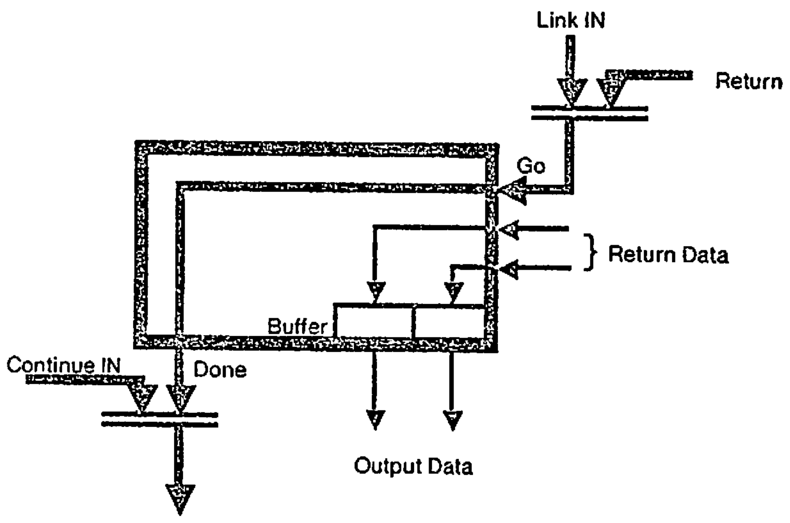


Figure 14: WAIT Construction

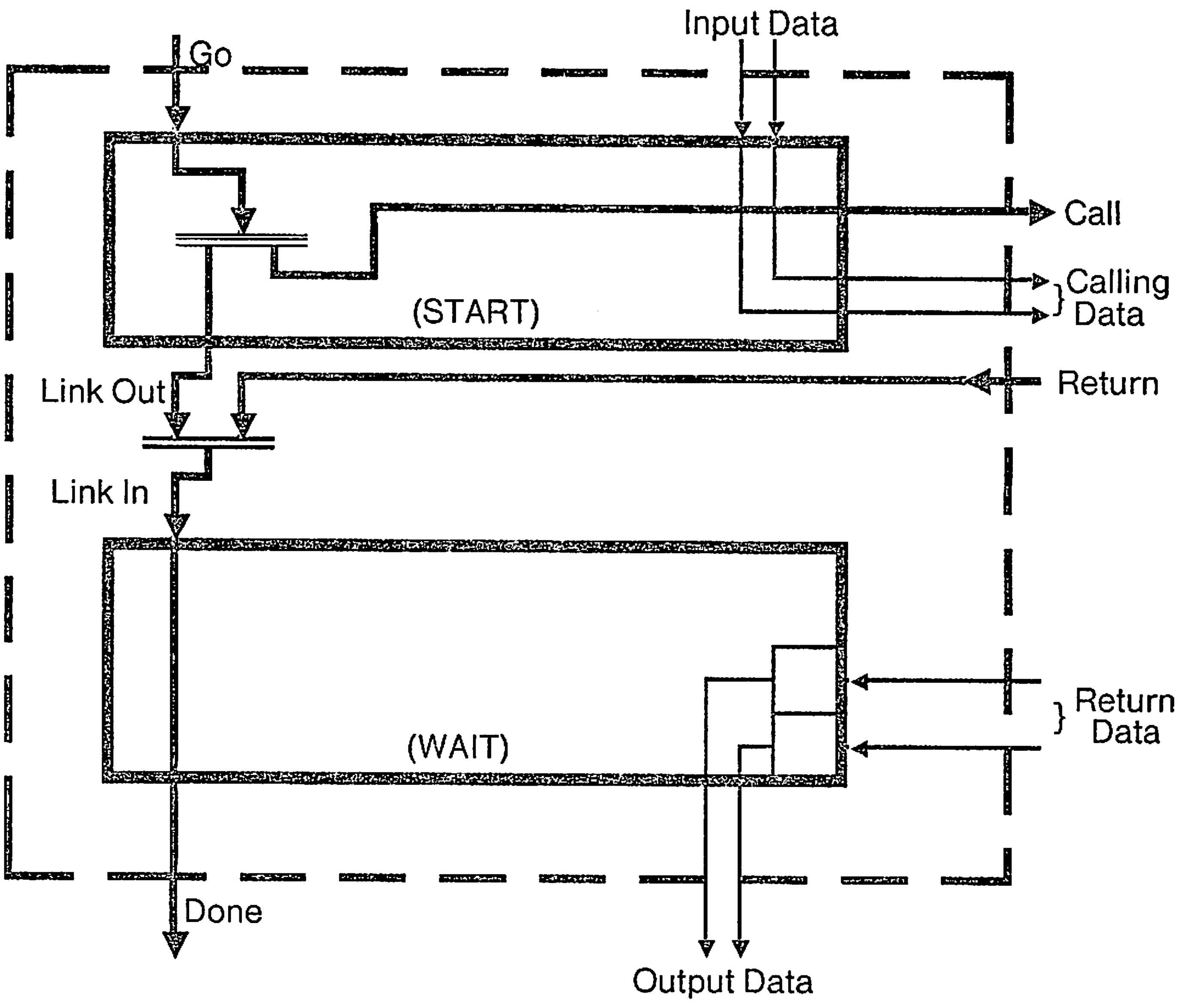


Figure 14.1: Calling Buffer
Constructed from a START and a WAIT

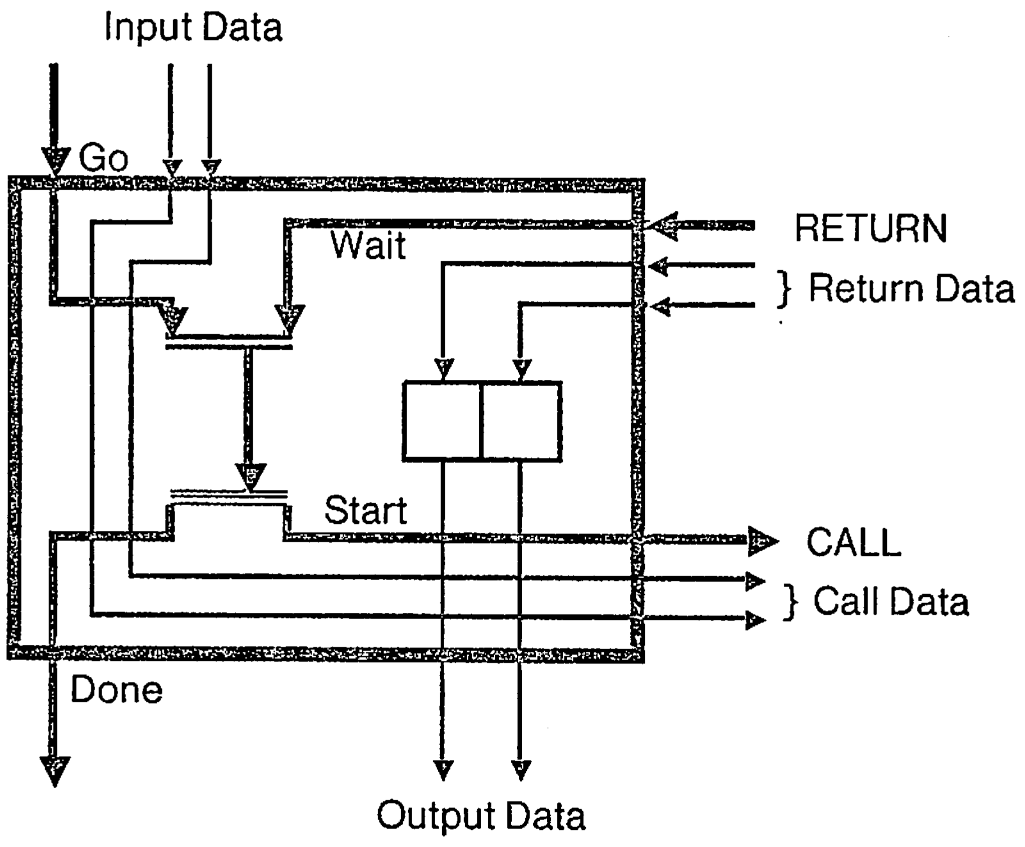


Figure 15: WAIT-START Buffer

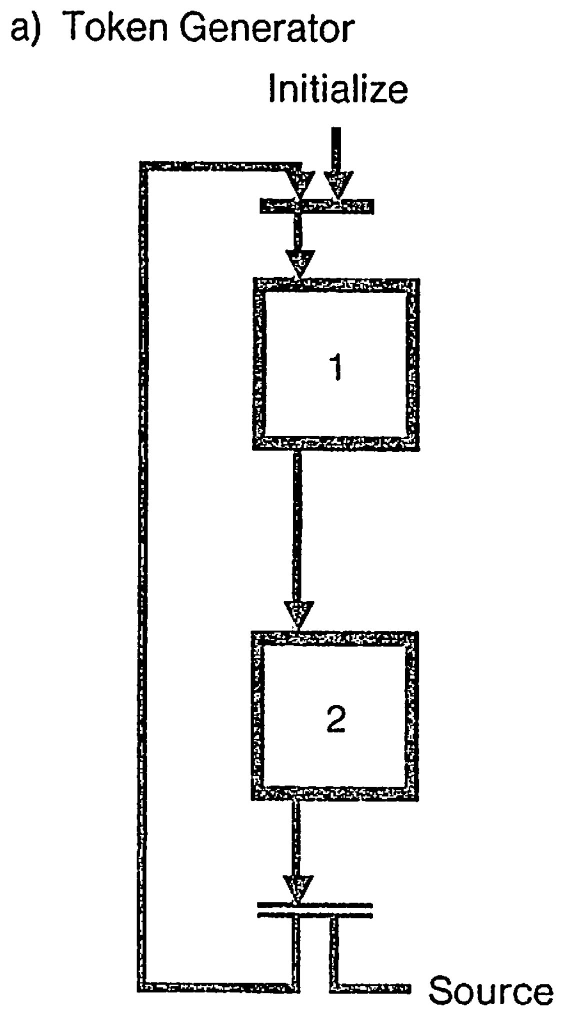


Figure 15.1 : Token Generator and Absorber

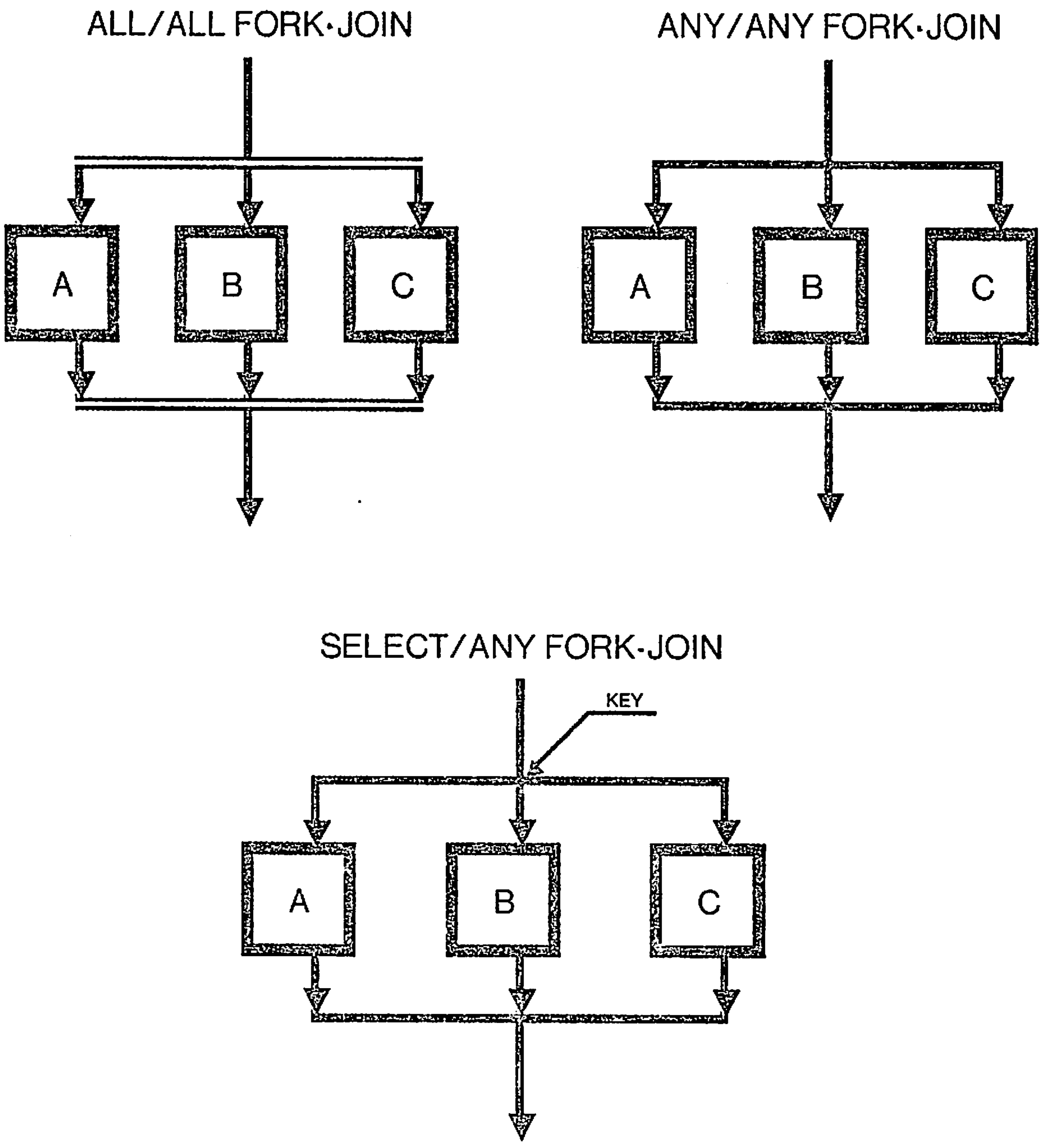
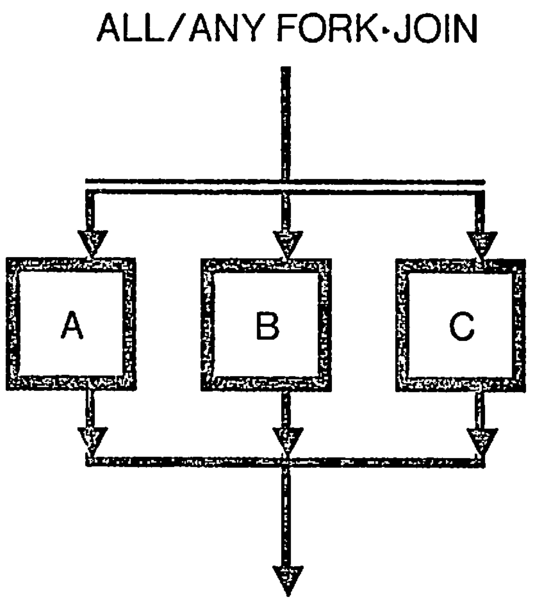


Figure 15.2 : Common FORK·JOIN Constructions



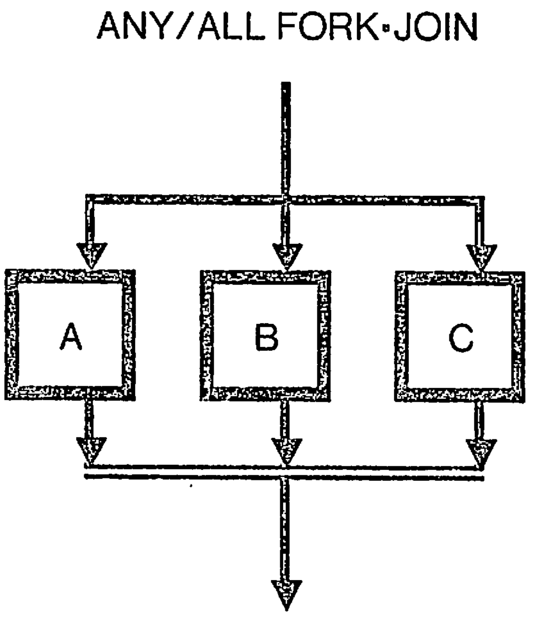


Figure15.3: FORK-JOINs Which Do Not Conserve Tokens

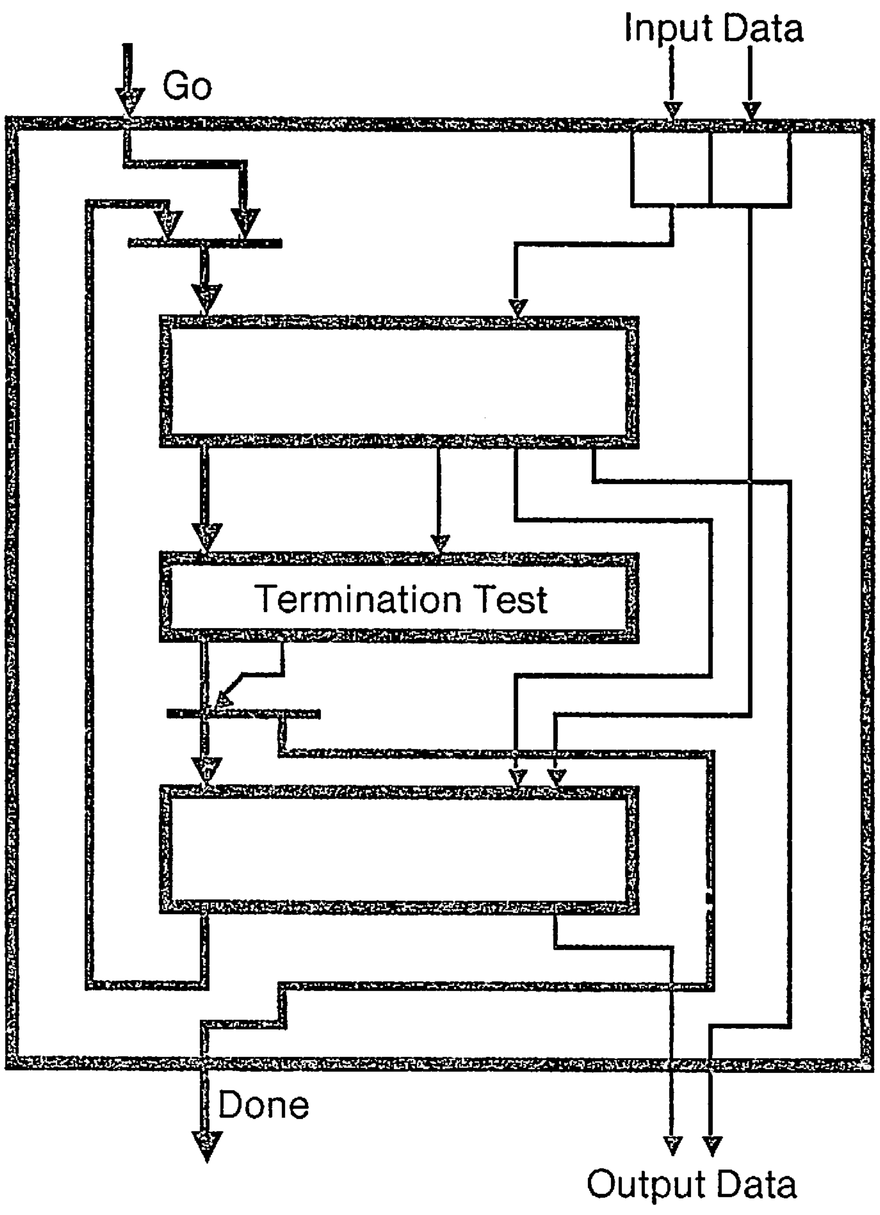


Figure 15.4: Example of a Loop Structure

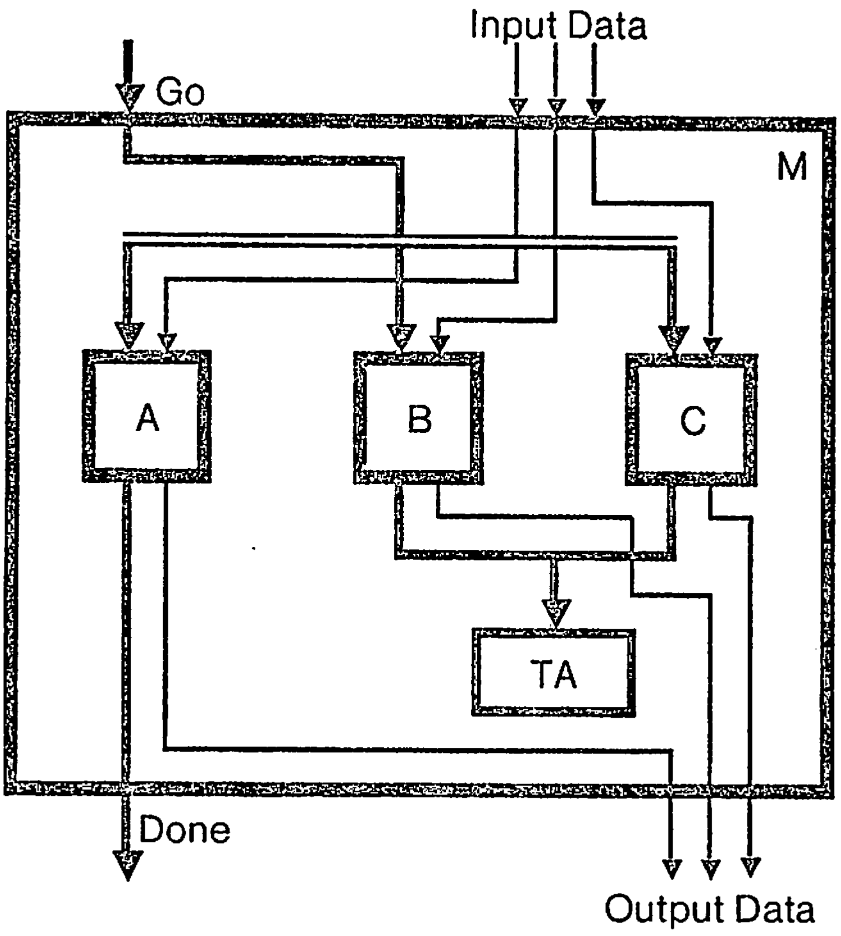


Figure 15.5: Example of an
Ill-Formed Composite Module

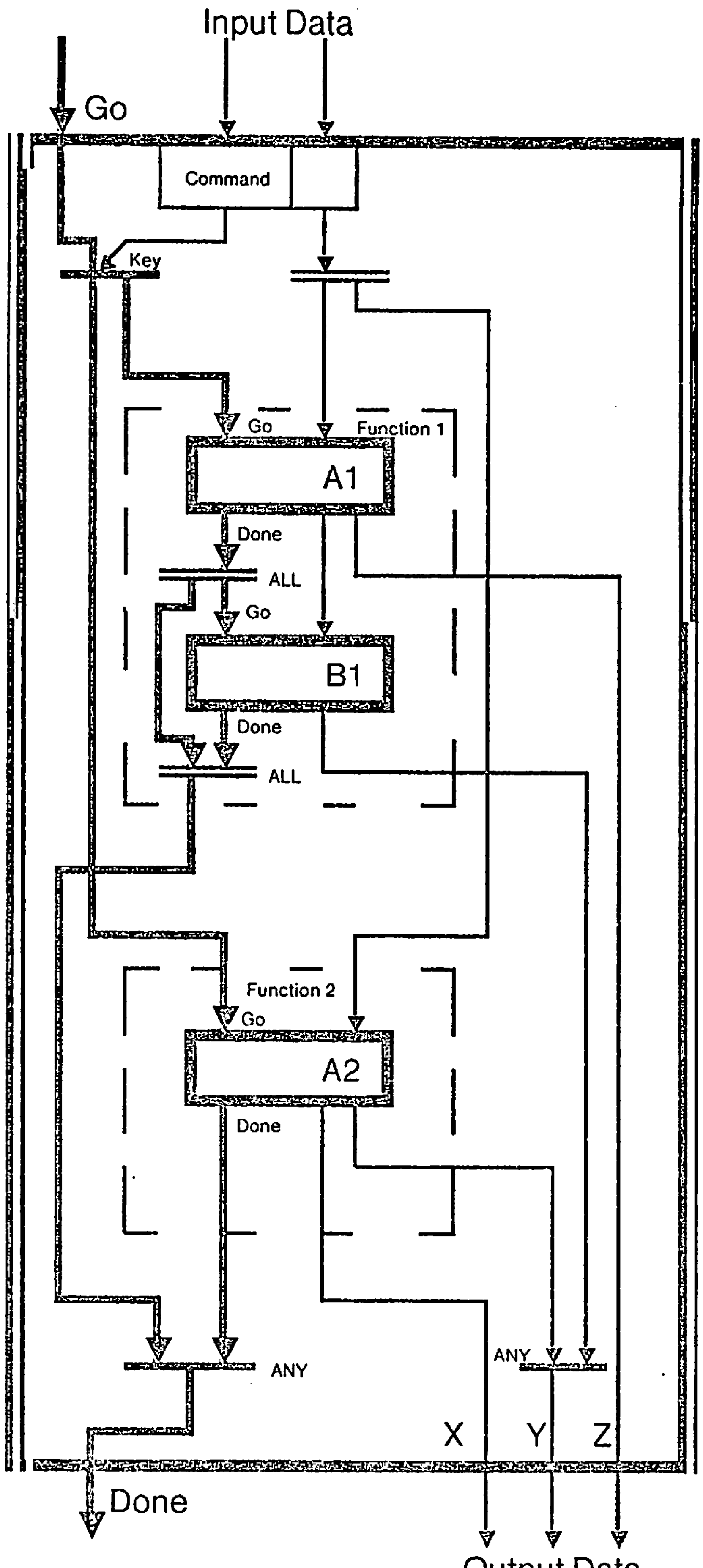


Figure 16: Example of a Multi-function Module

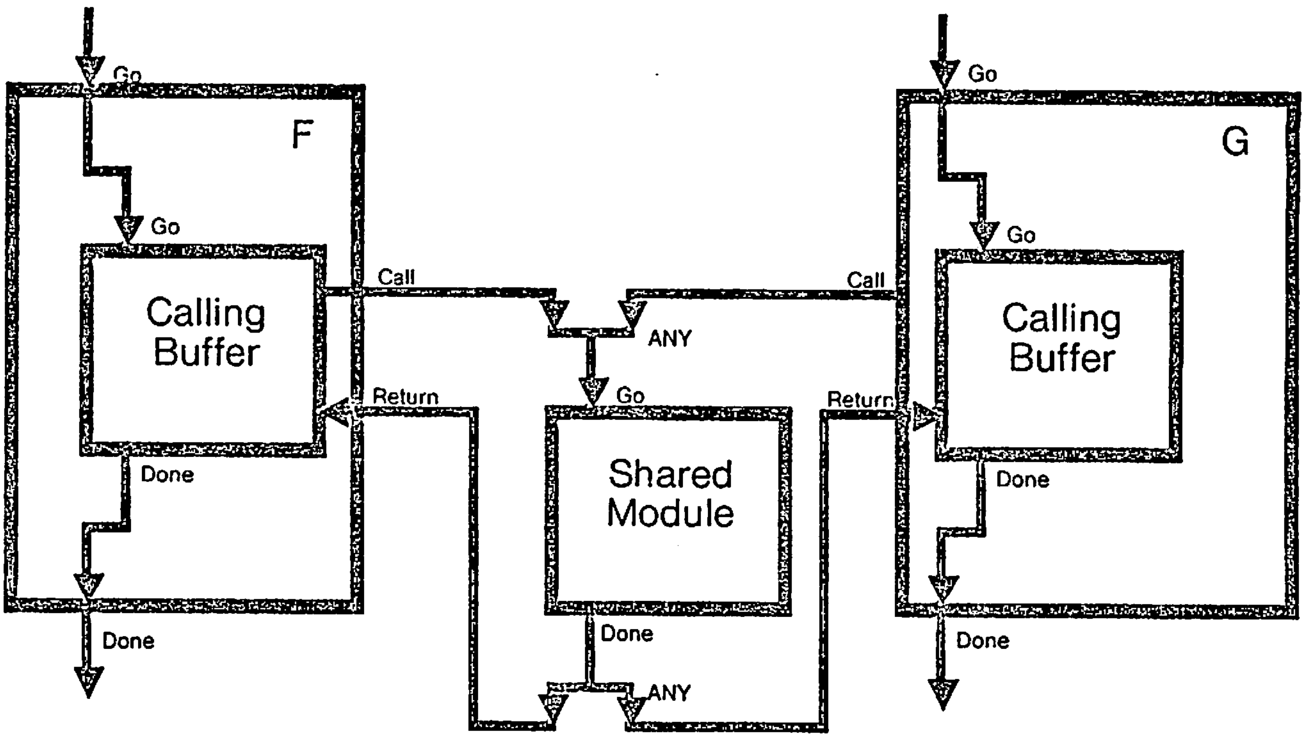


Figure 17: Sharing A Module

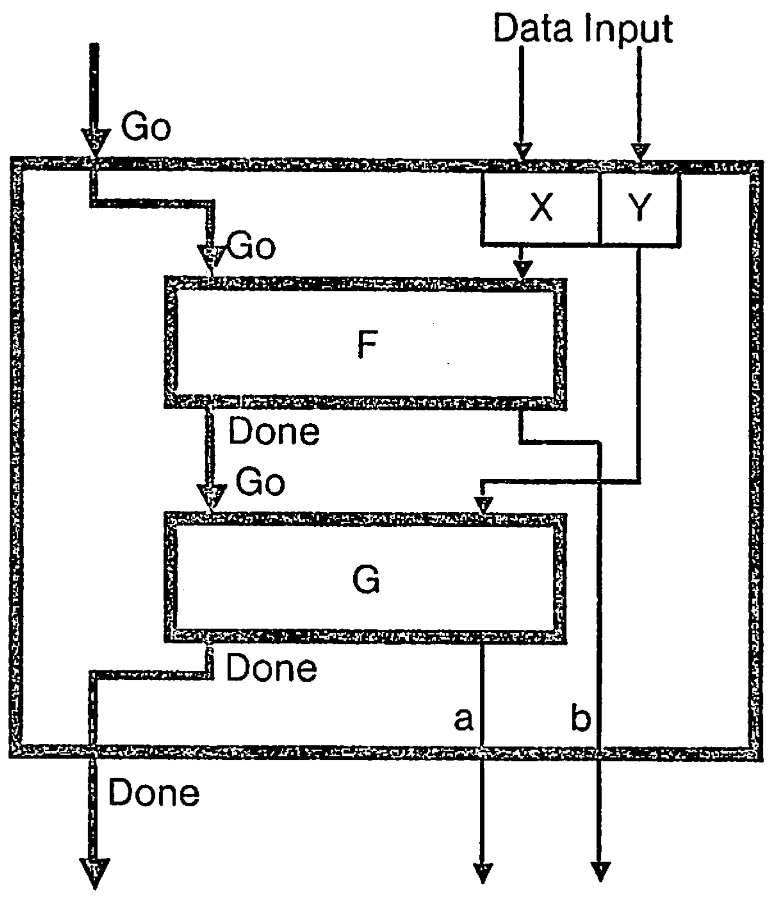


Figure 21:
Example of Two Sequential Submodule Calls

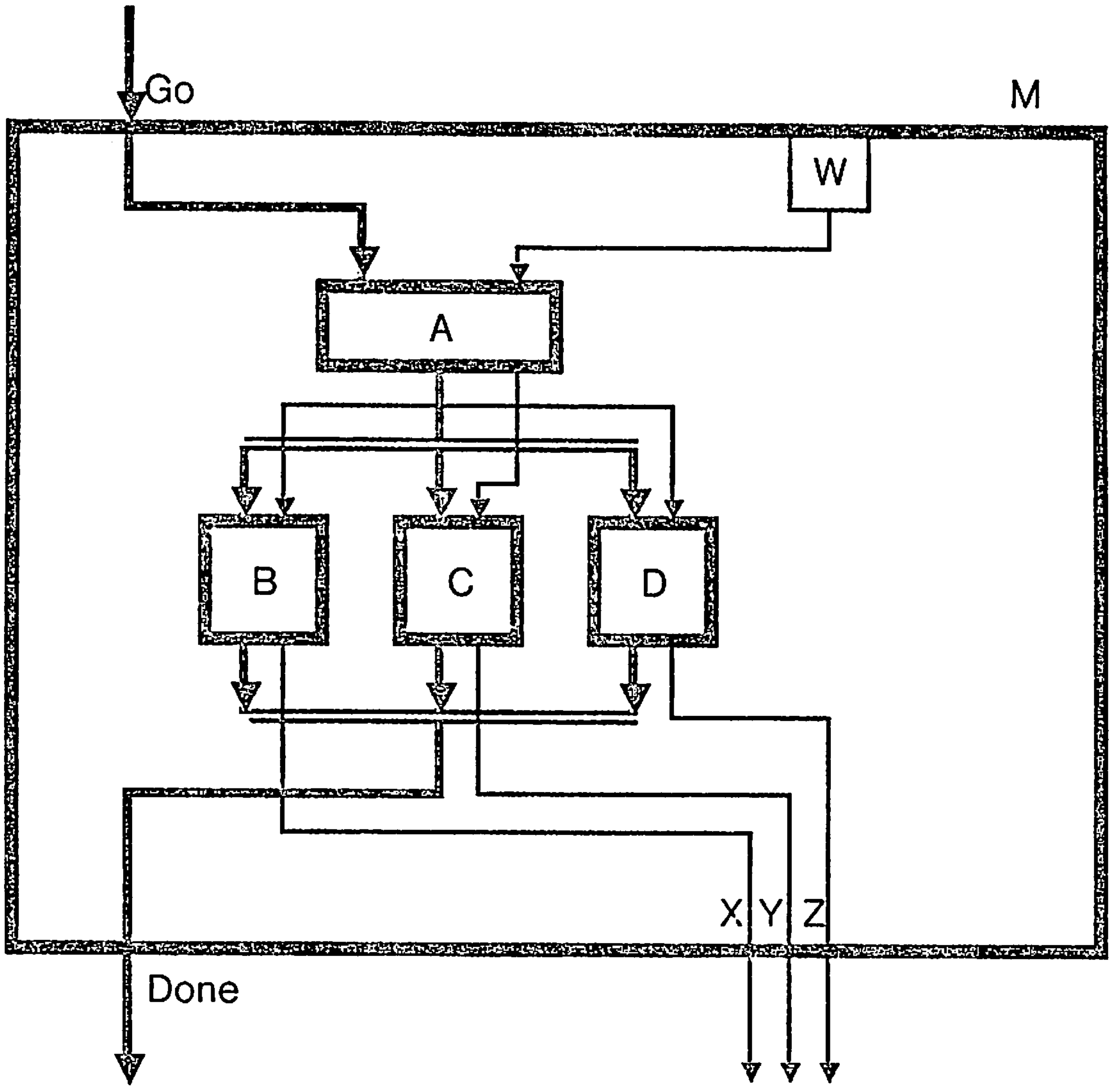


Figure 22: Example of
Parallel Activation of Submodules

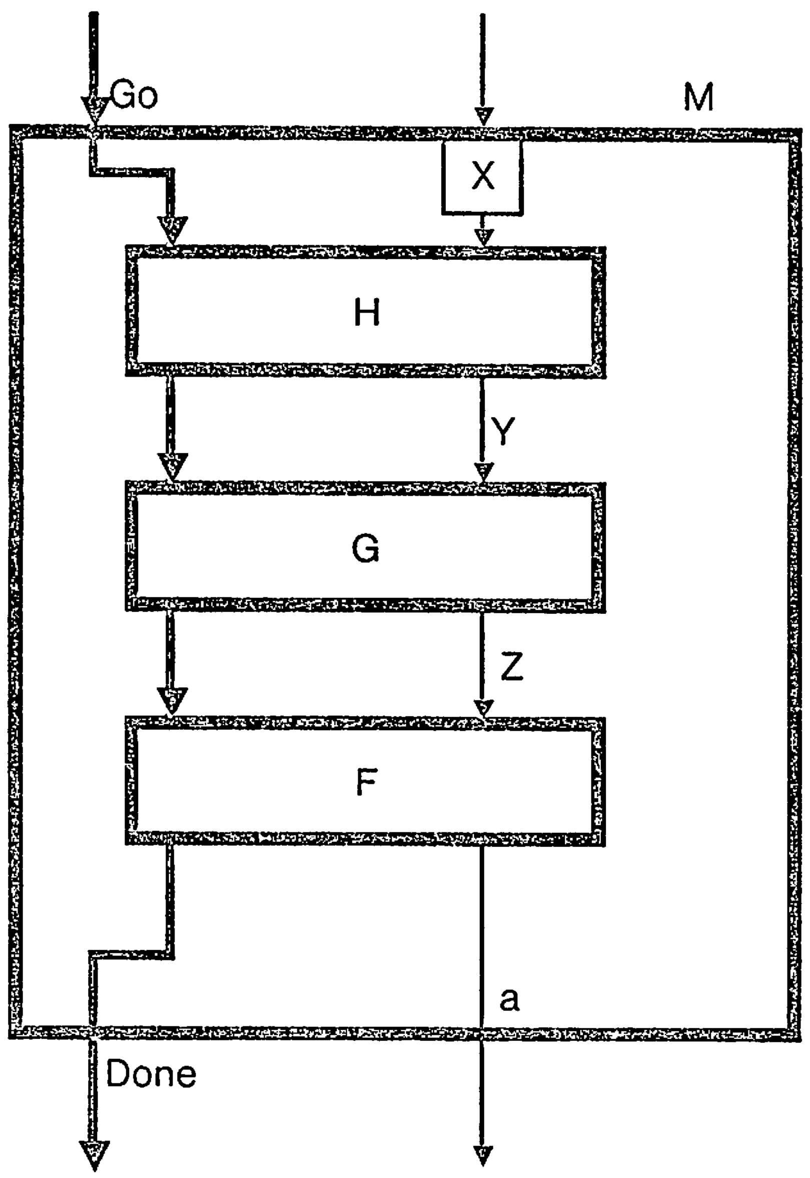


Figure 23: Composed Functions

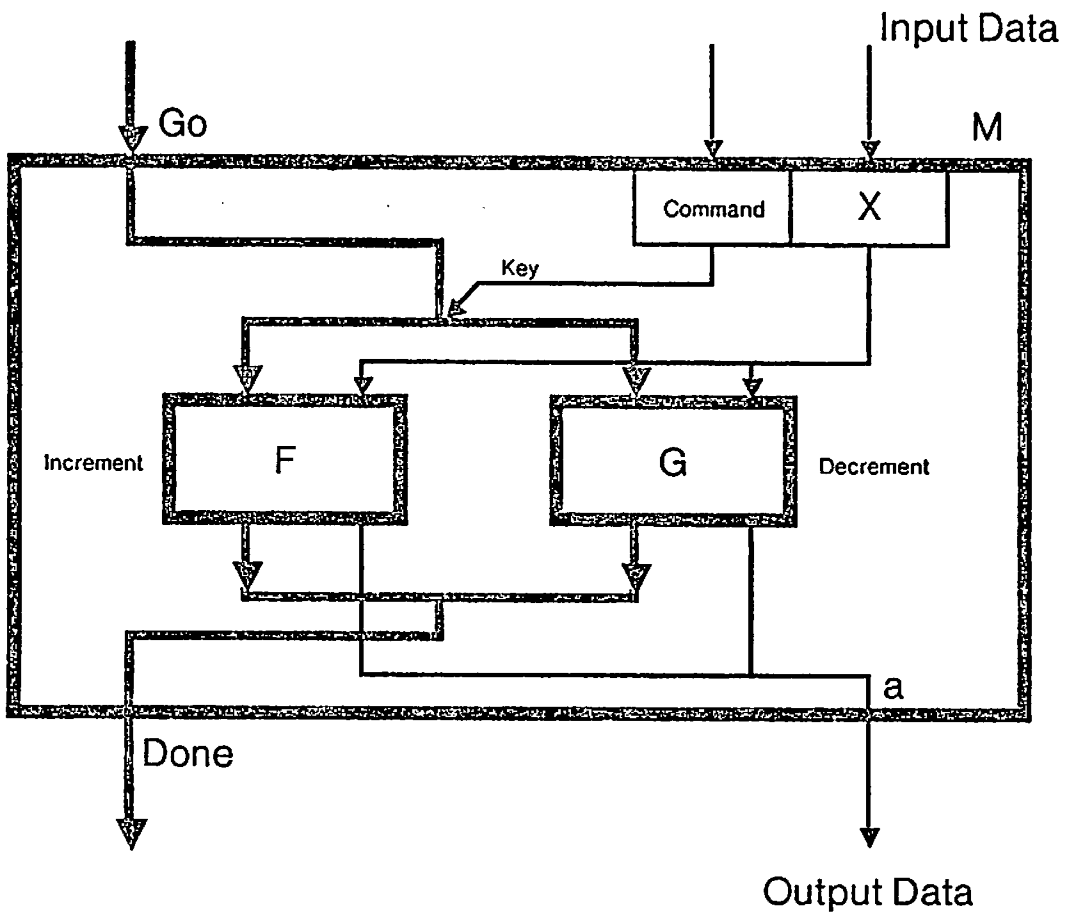


Figure 24: Example of a Case Statement

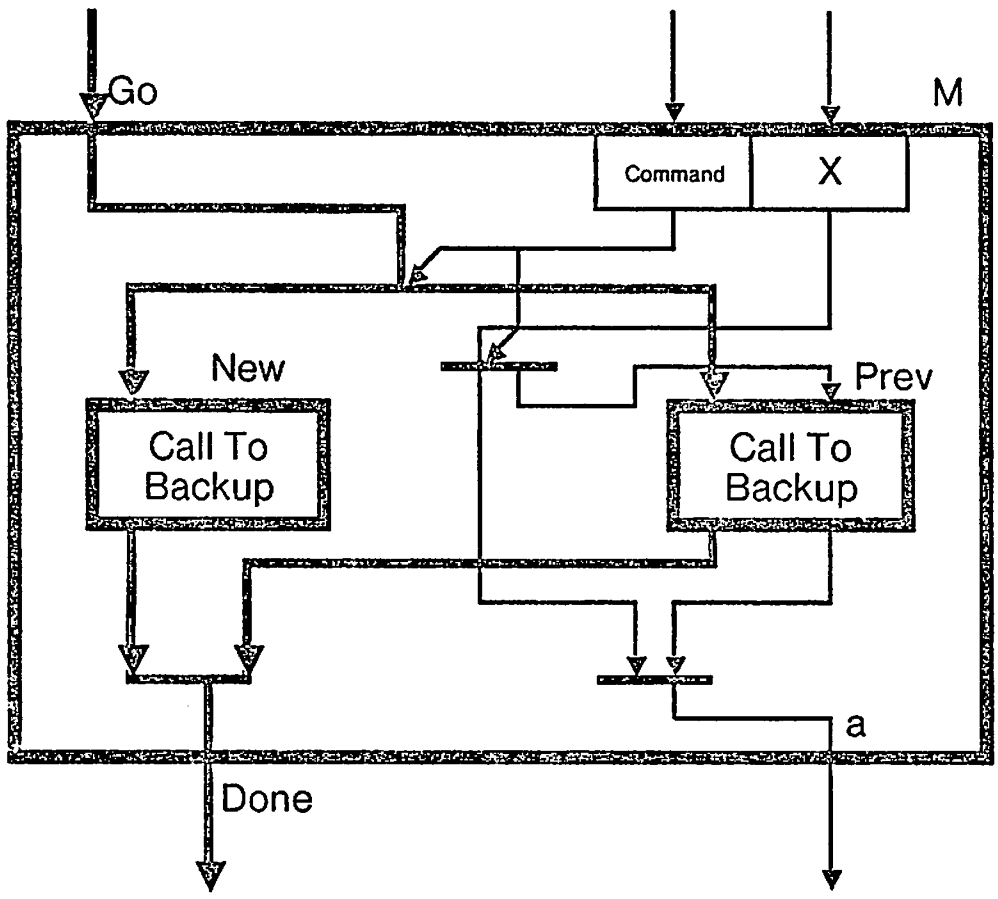


Figure 25: Data Steering
in a Multi-Function Module

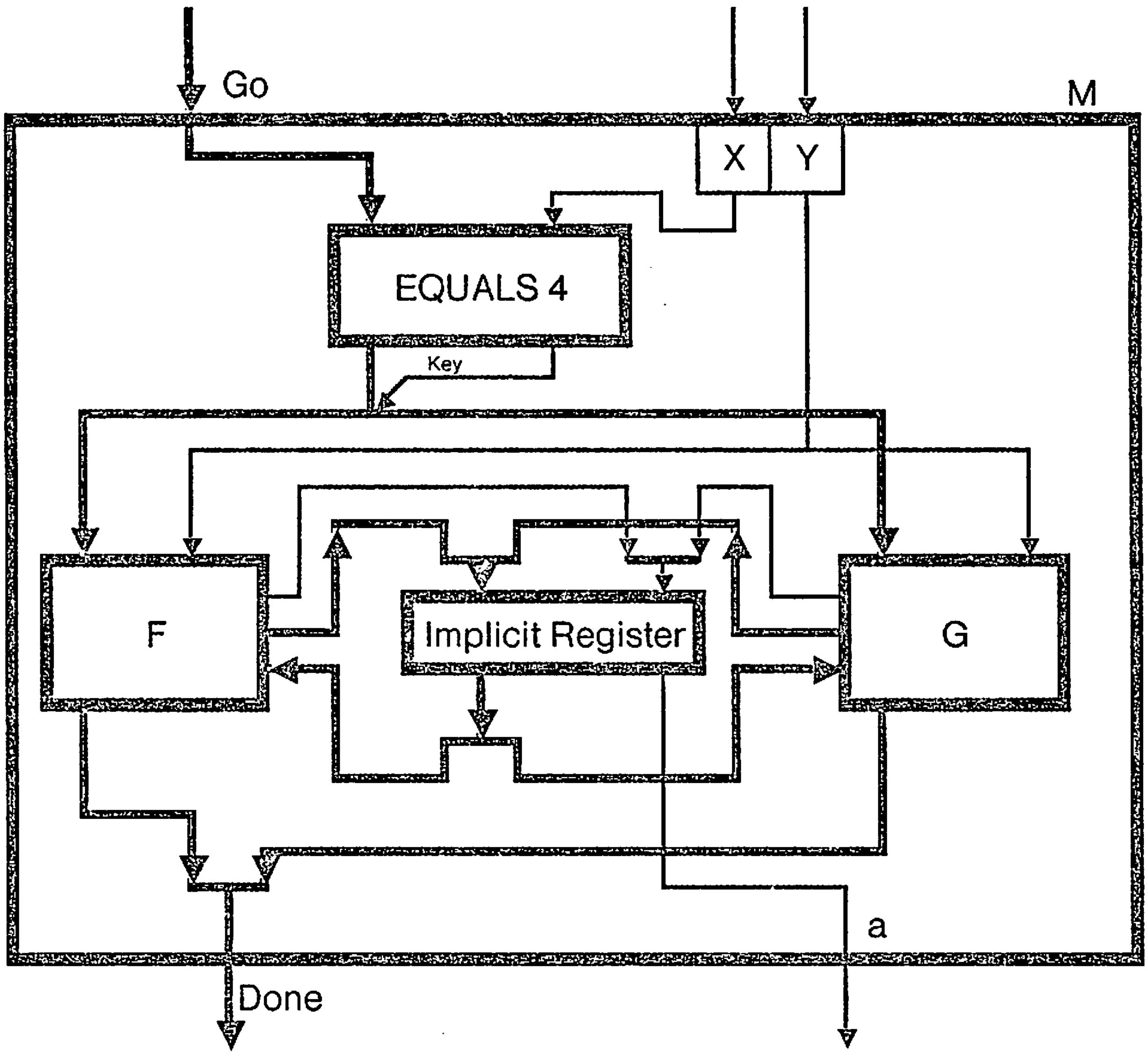


Figure 26: Example of Conditional Statement

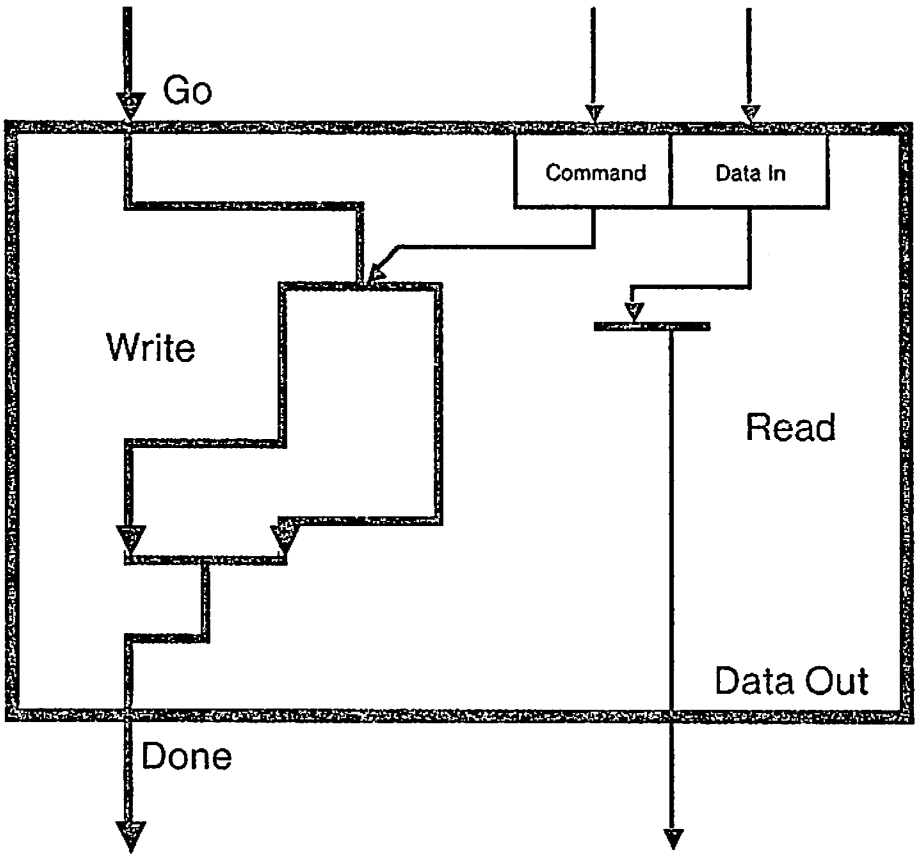


Figure 27: Module for a Register

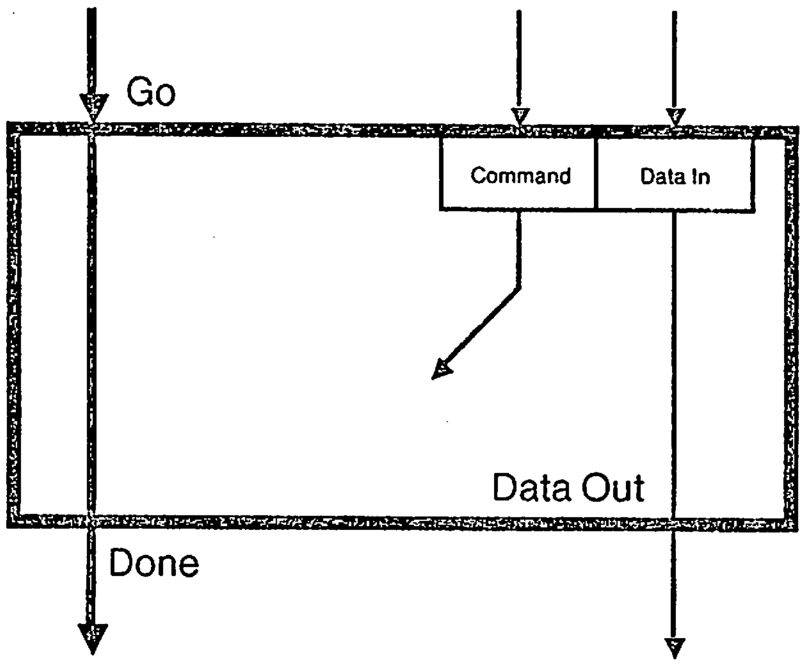


Figure 28: Simplified Version of LM Register

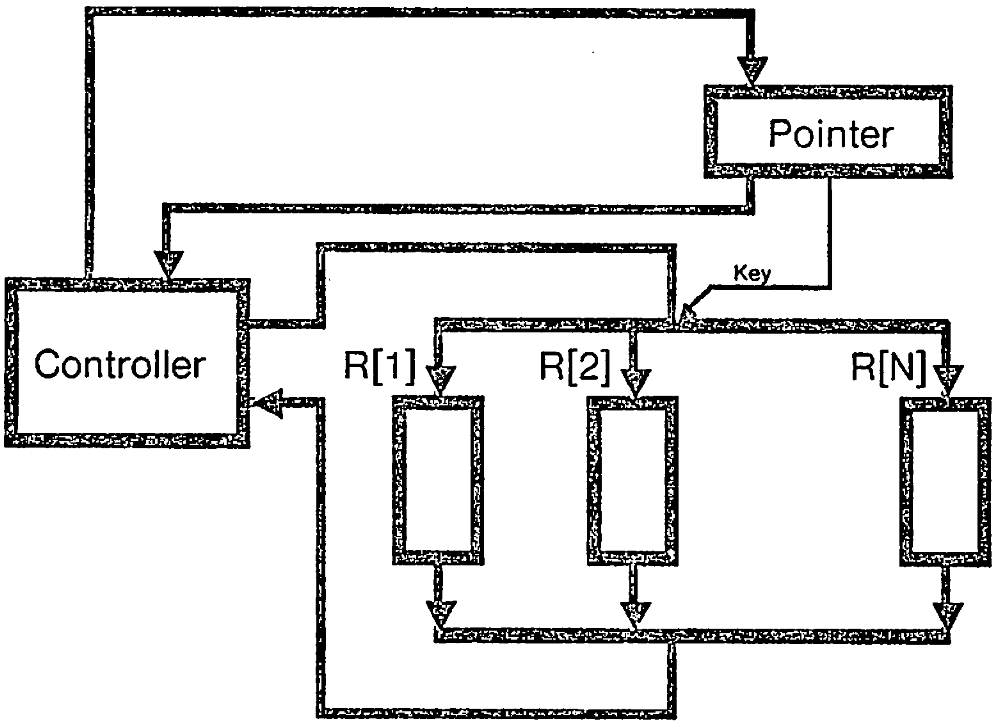


Figure 29: Pointer Stack

Figure 30: Roving Marker Stack

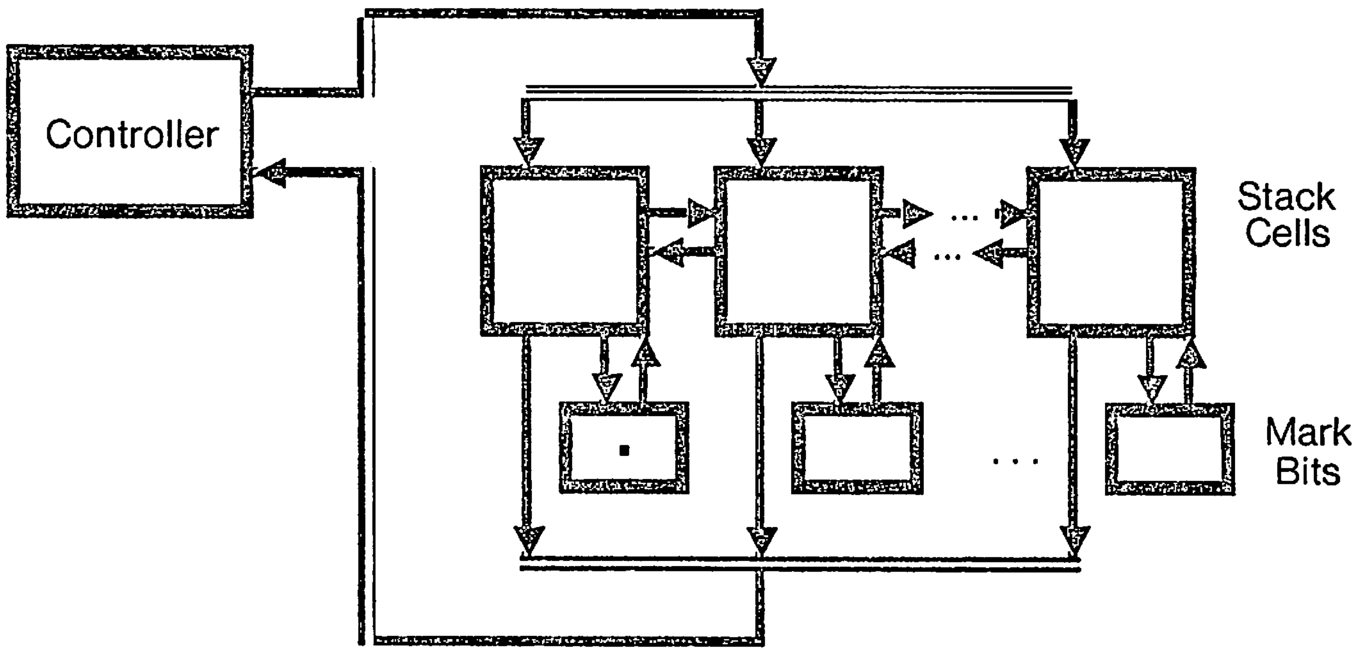
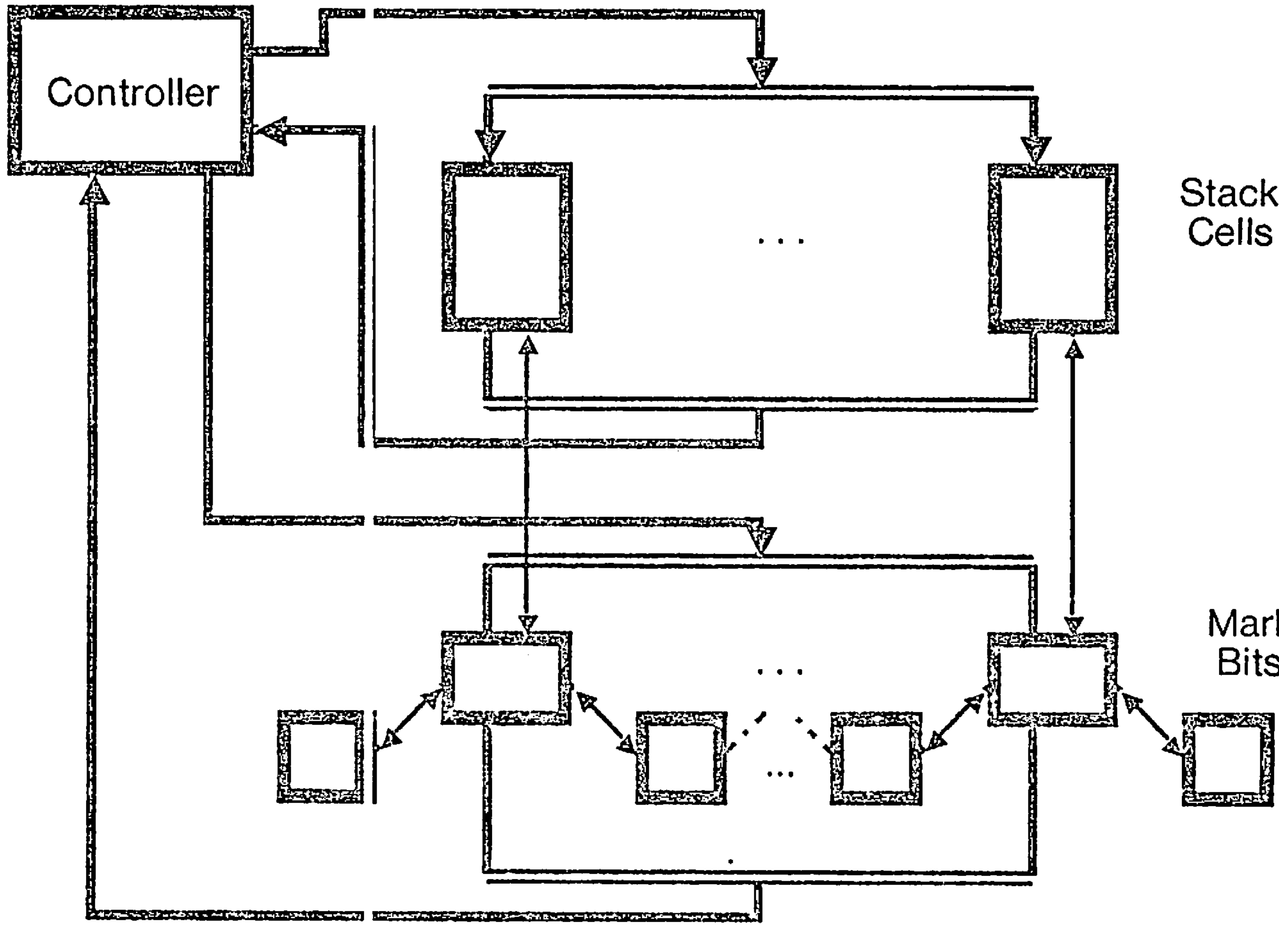


Figure 31: Ganged Marker Movement Stack


Controller
Stack Cells
Mark Bits