Journal of the Islamic University of Gaza, (Natural Sciences Series) Vol. 12, No.2, PP: 83-119, 2004

MODELING OF CONCURRENT EXECUTION OF PARALLEL PROGRAMS

Mohammad A. Mikki
mmikki@mail.iugaza.edu
Department of Electrical and Computer Engineering
Islamic University of Gaza, P.O. Box 108, Gaza, Palestine

تصميم نموذج للتنفيذ المتوازي للبرامج المتوازية

ملخص نقوم في هذا البحث بعرض طريقة جديدة لتصميم نموذج للتنفيذ المتوازي للبرامج يحاكي التوازي المتزامن في البرامج المتوازية. هذه الطريقة تستخدم لغة VHDL لتصميم النموذج. هذه الطريقة تنقسم إلى جزئيين: الجزء الأول هو عبارة عن برنامج إعراب لغة IF1. برنامج إعراب IF1 يقوم باستخدام رسم سريان البيانات الذي يمثل برنامج IF1 كمدخلات و يقوم بإنتاج كيانات جافا كمخرجات. الجزء الثاني هو مترجم IF1 إلى VHDL. هذا المترجم يأخذ كمدخلات كيانات جافا التي تم إنتاجها بواسطة برنامج إعراب لغة IF1 و يقوم بإنتاج برنامج VHDL كمخرجات.لإثبات طريقتنا نقوم بتطبيق برنامج محاكاة لغة VHDL لمحاكاة برنامج VHDL الناتج و الحصول على معايير قياس أداء نموذج للتنفيذ المتوازي للبرامج الذي قمنا بتصميمه. إن نتائج المحاكاة تبين أن طريقتنا دقيقة و قوية و أن النتائج تتطابق مع التوقعات. لغة VHDL تسمح للمبرمج أن يصف البرنامج عند مستويات مختلفة من التجريد و بطريقة هيكلية. هذا يجعلنا قادرين على وصف التوازي في البرامج على درجات مختلفة من النعومة و هذا يعني أن طريقتنا مناسبة لجميع أنواع الحاسبات المتوازية و كذلك التوازي المتزامن و غير المتزامن و لأجهزة الحاسوب ذات الذاكرة المشاركة أو التوزيعية، الخ. إن طريقتنا قادرة أيضا على توصيف توازي البيانات و كذلك توازي التحكم في البرامج المتوازية .

Abstract: A parallel-execution model (PEM) new approach that simulates concurrent parallelism in parallel programs is presented. The approach uses VHDL (VHSIC Hardware Description Language) as the modeling tool. The PEM approach consists of two components: The first component is an IF1 parser. The parser takes a dataflow graph represented by the IF1 (Intermediate Form 1) dataflow language as an input and produces Java objects as an output. The second component is an IF1-to-VHDL compiler. This compiler takes the Java objects produced by the IF1 parser as an input and produces the corresponing VHDL code that reprersents the graph as an output. To validate our approach we apply the VHDL simulator to simulate the resultant code and get the PEM performance metrics. Simulation results show that our model is accurate and powerful and the results coinside with the expectations.

VHDL allows the programmer to describe the program at different levels of abstraction in a hierarchical fashion. This enables us to model program parallelism at all levels of granularity, which means that our parallel program model is suitable for all parallel system platforms, including synchronous and asynchronous parallelsim, shared memory computer systems as well as message passing distributed memory systems, etc. Our model is also capable of modeling both data parallelism as well as control parallelism.

Keywords:
Data Dependency, Dataflow Graphs, IF1, Parallel Execution Model, VHDL

1. Introduction

Models of computation have several important uses relating to the design of computer systems. A model may serve as a pattern for structuring and analysing programs. Some models are intended as a basis for assessing performance of systems or applications [3]. Models that specify the behaviour of a computer system to the extents that it is relevant to correct execution of application programs are called program execution models because they explicitly describe the actions involved in the execution of a program by the specified computer system. A program execution model defines the semantics of the target representation for the compilers of high level languages and it provides a clear design goal for the computer system architect [3].

A parallel execution model for dataflow programs is a model in which parallel computations are executed in parallel. The model can concurrently exploit parallelism in dataflow programs. This model combines control parallelism and data parallelism. It also combines parallelism at all levels of granularity. In a coarse-grained dataflow execution model, a distributed application consists of a graph of modules. Each module may run on a different or the same phyisical processor and message passing between any two modules is through the edges[6].

The dataflow approach is an inherently parallel execution model represented by directed graphs. Nodes in the graph represent instructions and edges represent the data paths through which operands flow. The execution of a node occurs when all of its operands are available. The execution order of instructions is constrained only by the true data dependencies present in the program. This property of dataflow execution exposes maximum parallelism at all levels of granularity [6]. Dataflow approach can represent an abstract view of program execution or it can represent the actual dynamic execution of a program by a

microprocessor or computer system. The dataflow processing model is based on the notion of that, the computations take place when data are available. The central control unit is replaced by more distributed, data moving mechanism[6].
In this paper a parallel-execution model (PEM) approach that can simulate concurrent parallelism in dataflow programs is presented. The approach uses VHDL (VHSIC Hardware Description Language) as the modeling tool. We use VHDL in building the parallel execution model of the dataflow graphs because VHDL signal assignment statements are fundamentally parallel and their execution is triggered by event occuring on signals that the assignment statement is sensitive to. This is same as dataflow graph that represents programs where nodes (that represent computation) are fundamentally triggered by availability of events (data on their input). VHDL allows the programmer to describe the program at different levels of abstraction in a hierarchical fashion. This enables us to model program parallelism at all levels of granularity, which means that our parallel program model is suitable for all parallel system platforms, including synchronous and asynchronous parallelsim, shared memory computer systems as well as message passing distributed memory systems, etc. Our model is also capable of modeling both data parallelism as well as control parallelism. We use Java to code PEM approach. The PEM approach models parallel execution of pgograms written in IF1 datflow programming language. User applications written in different languages have to be converted into this common platform in order to use our apprpach. We use IF1 as the common platform because IF1 programs are basically dataflow graphs.
Our model has the following advantages:

- It can model both data parallel and control parallielism
- It can exploit maximum parallelism
- Computational nodes are executed asynchronously
- Computational nodes are scheduled dynamically
- It incorporates synchronization overhead and communication overhead
- It can model both message passing and shared variable communication models

Using VHDL in modeling dataflow graphs is behavioral modeling which can be divided into dataflow and algorithmic. Dataflow descriptions are nonprocedural

and use concurrent statements, whereas algorithmic descriptions are procedural and use sequential statements. Algorithmic modeling in VHDL is similar to conventional software programming in Pascal , C or Ada [4]. We did not use Java as the modeling language since it has the following limitations [9]:

1. Java concurrency level is too low level forcing programmers to work at a detailed level in dealing with thread creation, communication and synchronization
2. Java's synchronization model does not allow intra-object concurrency (i.e., an object is allowed to process incomming messages simultaneously.).

Verification tools assist in checking the correctness of the design. Timing analysis tools and simulators are common examples of verification tools. A simulator exercises a design over a period of time. Generating a set of outputs in response to a set of inputs. We will use VHDL-93 simulator.

VHDL plays an increasing part in digital systems design and modeling. VHDL description, also called a model, describes what a digital system does and how the system does it. Once written, a VHDL model can be executed by a software program called a simulator. A simulator runs a VHDL description, computes the outputs of the modeled digital system in response to a series of inputs applied over time. In other words, a simulator exercises the dynamic behavior of a VHDL model. VHDL is the newest standard language to address the rapidly growing complexity and sophistication of digital system design. The design of VHDL formally began in 1983 by the United States department of defense. And in 1987 was accepted as an IEEE standard. VHDL is rapidly gaining acceptance and is influencing advancements in design methodologies and design automation technology [4, 10, 12].

VHDL derives much of its syntax and semantics from Ada. A major difference in the languages comes from the notion of concurrent processes that permeates VHDL. Because the language is intended to model hardware components which are "always executing," VHDL is a highly concurrent language built upon a simulation cycle-based timing model. Interactions between VHDL processes occur over signals and include data and temporal aspects. VHDL is a Turing complete language and capable of representing very low level device models on up to system models by employing appropriate abstraction. Data encapsulation is part of VHDL [7]. Several past and current research and standardization

efforts focus on ways to extend the capabilities of VHDL to better support systems design and, in particular, to provide an integrated mechanism for modeling, simulation, and synthesis of software components [7].
VHDL allows the programmer to describe the program at different levels of abstraction in a hierarchical fashion. This enables us to model program parallelism at all levels of granularity, which means that our parallel program model is suitable for all parallel system platforms, including synchronous and asynchronous parallelsim, shared memory computer systems as well as message passing distributed memory systems, etc. Our model is also capable of modeling both data parallelism as well as control parallelism.
Using VHDL enables us to use a series of hierarchically related levels of abstraction. Modeling of parallel programs will be implemented at the various levels of abstraction including modeling of behavior and modeling of the structure. Behavior level model is the description of the program module by a procedure that defines the input/output behavior of the module over time, and structure level model is the description of a program module by means of a connection structure of more primitive modules. Model hierarchy implies the decomposition of the model entity into sub-components. Components in design entities provide powerful modeling capabilities. A hierarchical structural description is a powerful modeling construct in VHDL because it provides a mechanism to decompose the description of a large, complex digital system into smaller pieces.
A VHDL process represents the fundamental method by which concurrent activities are modeled. Processes execute in parallel. Within each process, code is executed sequentially. Change of the value of a variable in the sensitivity list results in the activation of the process.
We use VHDL because it has many important features which make it unique in representing dataflow programs. These characteristics include:

- Its support of top-down hierarchical design which enable accurate models to be developed and simplify the modeling process. This feature is used to model program parallelism at any level of granularity.
- VHDL has a simulator associated with it. This means that a model description written in VHDL can be used to validate the model.
- It has primitives that can model the temporal behavior of parallel programs.

- It supports process concurrency which is used to model parallel program concurrent tasks.
- The VHDL model has the advantages of extendibility, and reusability, where components may be used in modeling other entities and robustness.

Timing analysis tools and simulators are common examples of verification tools. Verification tools assist in checking the correctness of the design. An initial VHDL model is progressively expanded and refined by repeatedly simulating the model, examining the results, and modifying the model. The final model is sufficiently detailed to serve as both a specification for actual hardware implementation and as a documentation aid to communicate the design to other groups or organizations. Use of the 'generic' statement in VHDL is used to model parameterized design entities. Generics, like ports provide channels of communication with the entity. With generics, a design entity no longer represents one hardware unit. Rather, a parameterized design entity defines a template, representing a family of hardware units. A particular hardware unit is selected by setting the generics.
The rest of the paper is organized as follows: Section 2 presents some related work in the field of parallel execution modeling of programs. Section 3 prsents an overview of IF1 dataflow language. Section 4 presents the PEM approach developed in this research. Section 5 presents some simulation results. Finally, section 6 presents future work and concludes the paper.

2. Related Work

In this section we present some of the research work in the field of parallel execution modeling of programs.
[11] developed a system as part of the Cameron project, which compiles programs written in a single-assignment subset of C called SA-C into dataflow graphs and then into VHDL. The primary application domain is image processing. The goal of the Cameron project is to shift the programming paradigm from hardware-centered to software-centered. This is achieved by creating a software infrastructure that translates a high level algorithmic language into a hardware description language. In the Cameron project, data flow graphs are used as an intermediate representation between the algorithmic SA-C programming language and circuit-level FPGA configurations. Single

assignment semantics allow us to map SA-C variables to edges in a dataflow graph, while primitive operations in SA-C map to nodes. The dataflow graph makes data dependencies explicit, and is a convenient representation for many compiler optimizations. Dataflow graphs are then mapped to VHDL circuits by mapping edges onto wires and nodes onto VHDL components. The Cameron Project is a DARPA funded project, its function is to provide a simple and effective path between two otherwise separate areas. In this case hardware and software. It is designed to allow a programmer to write high level code that can be compiled straight into hardware. To do this, the project researchers developed a new language called SA-C, Single Assignment C. SA-C is a C-like language that is partially suited for compilation to hardware and has control structures to take advantage of parallelism. The SA-C program is compiled into host code, and a dataflow graph. The dataflow graph (DFG) is then translated into VHDL using a DFG - VHDL translator. The host code and VHDL are then put onto the hardware (an FPGA) and the process is complete.
[15] constructed a method for the partitioning of a single application specified in synchronous dataflow (SDF) [into multiple independently synthesizable, communicating VHDL hardware modules. Either synchronous or asynchronous communication between modules is allowed (global synchrony/asynchrony), and the clock timing and control are automatically generated. It has shown that this method guarantees the preservation of correct functional behavior as specified in the original SDF graph, and that many choices of partitioning into multiple hardware modules are possible. The ability to break up a larger application into smaller synthesizable hardware modules can lead to efficiencies in hardware synthesis, which is faster when performed on smaller hardware description language (HDL) specifications. It has demonstrated this method with some practical example applications that have been constructed in the Ptolemy system simulation and prototyping environment.
[5] prsents a parallel execution model called coarse-grained thread pipelining for exploiting speculative coarse-grained parallelism from general purpose application programs in shared-memory multiprocessor systems. This parallelization model which is based on fine-grained thread pipelining model proposed for the superthreaded architecture. Allows concurrent execution of loop iterations in a pipelined fashion with run-time data dependecy checking and control speculation.

[9] implemented Java4P, an extension of Java language, that offers a simpler concurrency model and overcomes Java limitations. In this model, threads are no longer associated with thread objects, allowing concurrency at any level of granularity. Thread creation is made implicit and synchronization is achieved through method guards. Synchronization specification is separated from the functional specification to provide a parallel programming model closer to sequentail programming.

[3] presents a set of principles for modular software construction and describes a program execution model on functional programming that satisfies the set of principles. The implications of these princiles for computer system architecture are discussed together with a sketch of the architecture of a multithread processing chip which promises to provide efficient execution of parallel computations while providing a sound base for modular software construction.

[14] explores the parallel execution of logic programs based on data flow analysis algorithm. A logic program is first applied by data dependency analysis] which can find all mode combinations possibly exist within a l ogic clause. This mode information is used to support a novel hyprid parallel execution model, which combines both top-down and bottom-up evaluation strategies. By adapting this model, various improvements can be achieved.

[1] presents a parallel execution model of logic programs. It can concurrently exploit AND and OR parallelism in logic programs. Their model employs a combination of techniques in an approach to executing logic programs in parallel, making trade-offs between number of processes, degree of parallelism, and communication bandwidth.

3. An Overview of IF1

A dataflow program is a graph, where nodes represent operations and edges represent data paths. Dataflow is a distributed model of computation: there is no single locus of control. It is asynchronous: execution ("firing") of a node starts when (matching) data is available at a node's input ports. In the original dataflow models, data tokens are consumed when the node fires. Some models were extended with "sticky" tokens: tokens that stay - much like a constant input - and match with tokens arriving on other inputs. Nodes can have varying granularity, from instructions to functions [2] .

IF1(Intermediate Form 1) is an acyclic graphical language developed as a target for SISAL (Streams and Iteration in a Single Assignment Language), a high

level functional programming language. IF1 is well suited for graphical display. The IF1 language consists of nodes, edges, types and graph boundaries. IF1 nodes are of two types: Simple and compound. Simple nodes represent elementary operations such as addition, subtraction, multiplication, equality and anding. Compound nodes represent complex constructs such as conditionals, loops and parallel construct ForAll. Edges represent data values, The literal edges describe constants. Graph boundaries describe functions, procedures and compound nodes by encapsulating sets of edges and nodes. As an example, a SISAL program, the corresponding IF1 source code and the corresponding IF1 graph are shown in Figures 1 (a) ,1 (b) and 1 (c) respectively.

In an IF1 file there are ten statements that describe any program. These statements are listed in Table 1

Detailed description of these statements is found in any IF1 manual such as in [13].

Advantages of using IF1 in our approach include:

1. Applications written in various high level programming languages can be mapped to a common single dataflow code
2. VHDL code executes as a dataflow program by the simulator
3. It has a “complex” node which can be used to implement the hierarchical nature of programs

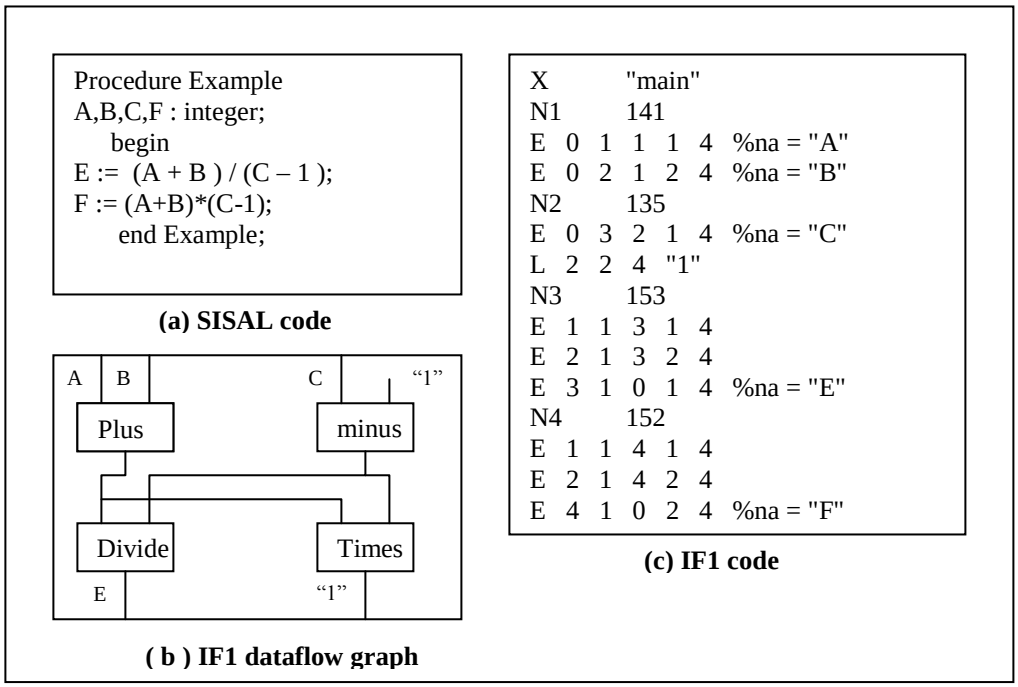


Figure 1: An example of IF1 SISAL, corresponding dataflow graph and IF1 code

4. F1 can be applied to fine-grained dataflow which requires the support of a programming language and machine architecture and coarse grained dataflow in which the basic execution unit is a program module [6].

Table 1: Statements of IF1

Key	Description	Key	Description
T	Type declaration	N	Simple node
C	Comment	{ , }	Compound node definition
I	Imported function	E	Edge definition
G	Subgraph of a compound node	L	Literal definition
X	Function graph		

4. Parallel Eexecution Modeling Approach

A parallel-execution model (PEM) approach that can simulate concurrent parallelism in dataflow programs is presented. The approach uses VHDL as the modeling tool. The procedure of PEM approach is shown in Figure 2. It consists of two components: The first component is an IF1 parser. The parser takes a dataflow graph represented by the IF1 dataflow language as an input and produces Java objects as an output. The IF1 dataflow graphs explicitly represent parallelism in parallel programs written in different languages. Emphasis in the design of the IF1 parser is on building the VHDL concurrent signal assignment statements. Applications written in different languages are converted into IF1 dataflow code in order to use our approach. This will enhance the use of dataflow graphs for representation of programs written in high level languages. IF1 code is inherently represented by a dataflow graph. The second component is an IF1-to-VHDL compiler. This compiler takes the Java objects produced by the IF1 parser as an input and produces the corresponding VHDL code that represents the graph as an output. The VHDL code is the parallel execution model of the dataflow program. This VHDL code represents the machine-independent simulation model of the application. It does not contain computation timing or communication costs. These are modeled using VHDL transport delay constructs. The VHDL model allows intra-object concurrently. To validate our approach we apply the VHDL simulator to simulate the resultant code and get the PEM performance metrics. These performance metrics include parallel execution time, communication overhead, synchronization overhead, speedup, efficiency, etc.

Precedence constraints in IF1 are preserved in VHDL by the VHDL simulator. Simulation is done by building performance test cases to prove that the model is correct. The simulation cycle defines how to mentally " play out" the execution of a VHDL model to understand its input/ output behaviour over time. In more formal terms, the simulation cycle defines VHDL dynamic semantics.

Dataflow graphs explicitly show data dependency between nodes and represent relationships among the computational tasks and explicitly present parallelism to user, and hence the PEM approach constructs appropriate process structures based on the dataflow graphs.

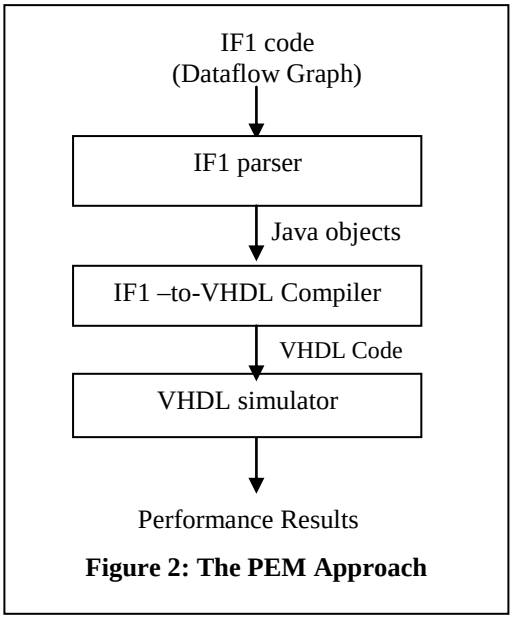


Figure 2: The PEM Approach

Our modeling approach is new, genuine and original. It uses VHDL in modeling the behavior of parallel programs. The use of VHDL in digital system design is recent, and its use in modeling of program parallelism is new. VHDL modeling exhibits the following criteria: It models computer programs accurately, it has different levels of abstraction, it includes a simulator which makes the validation process easier, VHDL as a modeling technique is the latest technology of digital system modeling compared to other modeling techniques such as the classical queuing models; transfer function models, procedural languages models etc,. Our approach makes it possible to compare the performance of different parallel programs.
In the following subsections we present the two main components in our approach, namely, the IF1 parser and the IF1-to-VHDL compiler.

4.1 The IF1 Parser

The first component in the PEM approach is the IF1 parser. The parser takes a dataflow graph represented by the IF1 dataflow language as an input and produces Java classes as an output. The IF1 parser defines some necessary classes that represent different IF1 constructs and features. It then converts the

IF1 code into Java objects needed for building the VHDL code in thew next component. The parser includes a cost assignment model for all nodes and edges of the IF1 file. The parser takes one input, the IF1 file and produces one output the Java objects. The IF1 objects are produced from IF1 program by scanning the IF1 file and parseing it and then constructing the objects representing the program. The Java classes and partial IF1 parser's code are listed in Figure 3.

In Figure 3, class 'Graph' defines a main, subgraph or call graph in an IF1 file,

```
// Java classes of IF1
Public class Node { ……  }
Public class Edge {  ……  }
Public class Graph {……   }
Public class Type {  ……  }
…..
// partial IF1 parser's code
open a new IF1 file
while end of opened IF1 file is not reached
  read a line
  parse the line
  switch (first character of the line) {
        case 'G':
        case 'I':
        case 'X':
              create a new java object of type Graph;
        case 'N': create a new java object of type Node (simple node);
        case '{': create a new java object of type Node (compound node);
        case 'E':
        case 'L':
              create a new java object of type Edge;
        case 'T': create a new java object of type Type;

        …….
  }
end while
```

Figure 3: IF1 classes and partial IF1 parser's code

class 'Node' defines a node in an IF1 file, class 'Edge' defines an edge in an IF1 program, and class 'Type' defines a type in an IF1 file. As listed in Figure 3, the IF1 parser reads an IF1 file and parses it line by line since IF1 is a line oriented programming language. Each line in IF1 represnts a completed

statement. For each line the parser builds a new Java object corresponding to the type of the line.
The IF1 parser's routines use recursive-descendent parsing technique. We model the IF1 program as follows:

$$prog = \{T,C,I,G\}$$

where T is the list of all types; C is the list of all comments; I is the list of external functions to be imported, and G is the prog represented in a graphical form. In order to construct the graph one only needs to construct G. where G is defined as follows:

$$G = \{G1,G2,....,Gn\}$$

where Gi for all I between 1 and n represents the main graph, a subgraph of a compound node, or a call graph. Gi in turn is defined as follows:

$$Gi = \{N,E\}$$

where N is the list of nodes in G, and E is the list of edges in G. If a node Ni in N is compound then:

$$Ni = \{G1,G2,...., Gc\}$$

where c is the total number of subgraphs in Ni. The parser returns a list of Type objects and another list of Graph objects that correspond to Gi's in G.

4.2 The IF1-TO-VHDL Compiler

The second component of the PEM approach is an IF1-to-VHDL compiler. This compiler takes the Java classes produced by the IF1 parser as an input and produces the corresponing VHDL code that reprersents the graph as an output. The VHDL code is the parallel execution model of the dataflow program. The parallel computations in the dataflow program are modeled as concurrent signal assignment statements in VHDL. Table 2 shows the mapping of IF1 constructs to the VHDL constructs (primitives used by the compiler).
As Table 2 shows, the main IF1 program is represented by a VHDL entity and architecture. The graph nodes are the concurrent signal assignment statements in the VHDL architecture. These statements are executed by the VHDL simulator when events occur on signals that the assignment statement is sensitive to. Modeling nodes in the dataflow graph by concurrent signal assignment statements makes these statements execute in parallel by the simulator. The tasks (nodes) can be fine grain (e.g., add, subtract, ..) or course grain (function calls, procedures, ..). This implies that the parallel execution model can model program parallelism at different levels of granularity whether

it is fine grained, medium grained, or coarse-grained. The signals that the assignment statement is sensitive to in the VHDL code are the incoming edges of the corresponding node in the IF1 graph and the output signal of the signal assignment statement in the VHDL code is the outgoing edge of the corresponding node in the IF1 graph. There is one signal assignment statement in the VHDL code for each output port (outgoing edge) of each node in the IF1 graph. The program input/output is modeled using the input and output ports of entities. The compiler models the node computational time as transport delay in VHDL model because it is the one which is used for high level modeling applications, such as architecture or performance modeling, that are not concerned with the low-level inertial aspects of hardware.

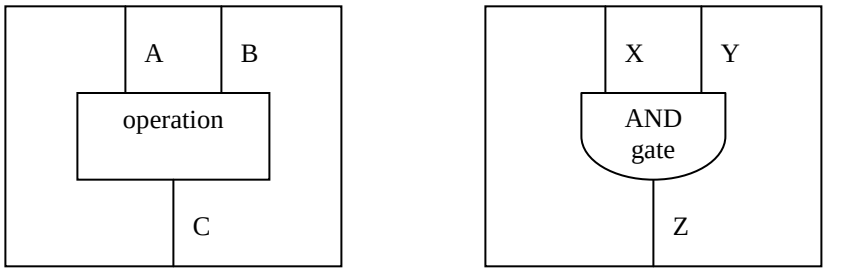
The approach uses behavioral architecture modeling. Behavioral architecture in VHDL uses concurrent signal assignment statements. What makes these statements different from assignment statements in typical programming languages is the fact that these statements execute in parallel (concurrently), not serially [10]. Inside the VHDL behavioral architecture there is no specified ordering of the assignment statements. The order of execution is solely specified by the events occurring on signals that the assignment statements are sensitive to [10]. In a behavioral model (architecture), there is no structure. There is no further hierarchy, and this architecture can be considered a leaf node in the hierarchy of the design [10]. A simulatable model is created by compiling the VHDL code that includes the entities and architectures.

Modeling all IF1 operations as 'AND' VHDL operations in combination of representing the availability of data at the input of a node as the signal's values becoming one in the corresponding VHDL code makes the output signal of the signal assignment statement becomes one only when all input signals (signals that the assignment statement is sensitive to) have the values of one (all become available) as shown in Figure 4. As 'operation' in Figure 4 (a) can not execute until both A and B inputs are available, Z will not become 1 until both X and Y become 1 in Figure 4 (b).

Table 2: Mapping IF1 constructs into VHDL primitives

IF1 feature	VHDL primitive
Graph	Entity and architecture
Graph input	Entity input port
graph output	Entity output port

Data type	Bit type
Concurrent nodes	Concurrent signal assignment statements
Sequential code	Single process
Node	Signal assignment statement
Literal	signal
Edge	signal
Arithmetic operation	AND logic operation
Computation time of node	Transport delay
Node's input edge	Signal to the right of the signal assignment operator
Node's output edge	Signal to the left of the signal assignment operator
Edge communication	Wait statement
Data availability at the node's input	event (signal value becomes one) occuring on signals that the assignment statement is sensitive to (signals to the right of the <= symbol
LoopA compound node	For loop statement



(a) IF1 node **(b) corresponding VHDL AND gate**

Figure 4: Modeling node operations in IF1 code

Partial code of the IF1-to-VHDL compiler is listed in Figure 5. As shwon in the Figure, there are three main classes: statement, entity and architecture. These are used to generate the VHDL code. There is a main entity which represents the main graph in IF1. There is also an entity for each subgraph in

IF1. This corresponds to the structural modeling in VHDL. A concurrent assignenet statemet is generated for each outport edge in each node.

```
// VHDL classes
class statement {
          private: int delay  ;
                   string output_signal;
                   string[] input_signal;
         public:          …..
};
class entity {String[] in_port,
out_port;  …}
class architecture {String[] signal;
…}

// Partial IF1-to-VHDL compiler's
// code
….
```

```
for the main IF1 graph
  create an entity and behavioral
architecture (model)
for each IF1 subgraph
  create a VHDL entity and architecture
for each IF1 node{
        create an AND gate
        for each IF1 inport  edge
        create a signal
        for each IF1 outport  edge
        create a concurrent signal
assignment statement
}  …..
```

Figure 5: IF1-to-VHDL compiler partial code

5. Experimental Simulation Results

In this section we present some experimental results to validate our model. We use our approach to compile some IF1 programs to the VHDL model, then we run the VHDL simulator to get the performance results (exceution time, communication overhead time, speedup, , etc.). Currently, only the parallel execution time is obtained. Note that the only limitation of executing statements in parallel in the model is the data dependency. The VHDL simulator used for the verification and validation of the approach is VHDL ModelSim SE Version 5.5 [8]. The simulator runs under Microsoft Windows 2000 environment. Simulation results show that our model is accurate and powerful and the results coinside with the expectations.

In order to show how IF1 could be compiled to VHDL, examples of VHDL source code that correspond to selected IF1 descriptions are shown below through the conduction of the following experiments.

The first experiment is shown in Figure 6. The experiment shows the computation of A:= (x + y)/z. The input IF1 file is listed in Figure 6 (a). Figure

6 (b) shows the corresponding dataflow graph. Figure 6 (c) shows the generated VHDL code. And finally, Figure 6 (d) shows the simulation of the PEM. As shown in Figure 6 (b) only one entity and one corresponding architecture are built. These correspond to the single main graph in IF1. The entity has 2 input ports and one output port. These correspond to the 2 inputs and one output of the IF1 graph respectively. Within the architecture there are 2 signals. These correspond to the output edge of node1 and first input edge of node2. The other edges are boundary edges and are converted to entity input and output ports. There are three signal assignment statements in the architecture. The first one corresponds to the execution of the first node in the IF1 file. The second one corresponds to the communication edge between node 1 and node 2. And the third statement corresponds to the execution of node 2. Note that there is a VHDL statement for each edge between nodes in the IF1 file and no VHDL statements for edges between the boundary of the IF1 graph and the nodes. Figure 6 (d) displays the visualization of the execution of the parallel dataflow program represented by IF1 code. This is obtained by simulating the VHDL code in Figure 6 (c) by the VHDL simulator. We view entity input ports that represent the input of the program and the entity output port which represents the output of the program. In addition we view the internal signals. When a signal's value becomes one in the simulation, then the data becomes available in IF1 code. Since the output port A becomes one at time of 210 ns, and input ports become one at the time of 100 ns. We conclude that the parallel time of the program is 110 ns. This corresponds to the summation of the execution times of nodes 1 and 2 in the IF1 graph. These nodes are executed sequentially. Note also that since the edge between node 1 and node 2 has no communication cost signals N1O_1 and N2I_1 become one at the same time i.e. at 150 ns which is the time needed to execute node 1 (after subtracting 100 ns which is the time when the input data becomes available). The simulation proves the correctness and validity of the model. And the simulation results show that our model is accurate and powerful and the results coinside with the expectations.

The second experiment is shown in Figure 7. The experiment shows the computation of A:= (x+y)/(x-y) ; B:= (x+y)*(x-y) with zero communication overhead. The input IF1 file is listed in Figure 7 (a). Figure 7 (b) shows the corresponding dataflow graph. Figure 7 (c) shows the generated VHDL code. And finally, Figure 7 (d) shows the simulation of the PEM. As shown in Figure 7 (b) only one entity and one corresponding architecture are built. These

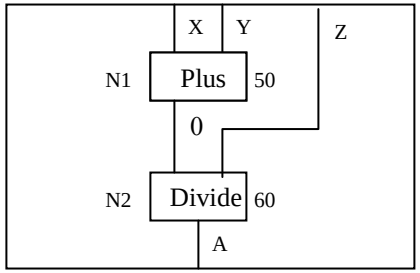
correspond to the single main graph in IF1. The entity has 2 input ports and 2 output ports. These correspond to the 2 inputs and 2 outputs of the IF1 graph respectively. Within the architecture there is a signal for each internal input and output edge of each node (an internal edge is an edge that is not connected to the IF1 graph boundary). There are four signal assignment statements in the architecture corresponding to the 4 nodes in the IF1 file. There is also a VHDL statement for each edge between nodes in the IF1 file. These represent the communication between nodes. Note that the edges are assigned zero communication costs. Figure 6 (d) displays the visualization of the execution of the parallel dataflow program represented by IF1 code. We view entity input ports that represent the input of the program and the entity output port which represents the output of the program. In addition we view the internal signals that correspond to the outputs of nodes 1 and 2. The output signal N1O_1 of node one becomes one at time 150 ns after 50 ns of the avaialbility of signals X and Y (which become available at time 100 ns). This is the execution time of node 1. And the output signal N2O_1 of node 2 becomes one at time 200 after 100 ns of the avaialbility of signals X and Y. Output A of node 3 becomes one after the 150 ns (execution time of node 3) of the avaialbility of both inputs of node 3 (the two inputs of node 3 become available at time 200 ns). This is shown in Figure 7 (d) by transition of signal A from 0 to 1 at time 350 ns. Output B of node 4 becomes one after the 200 ns (execution time of node 4) of the avaialbility of both inputs of node 4 (the two inputs of node 4 become available at time 200 ns). This is shown in Figure 7 (d) by transition of signal B from 0 to 1 at time 400 ns. The parallel execution time of the program is the time required for both outputs A and B of the program to become avaialable which is 300 ns. This is visually shown in Figure 7 (b). The sequential time of the program is 600 ns which is the summation of the execution times of the nodes in the IF1 program. The simulation proves the correctness and validity of the model. And the simulation results show that our model is accurate and powerful and the results coinside with the expectations.

The third experiment is shown in Figure 8. The experiment shows the computation of A:= (x + y)/(x-y) ; B:= (x + y)*(x - y) with non-zero communication overhead. The input IF1 file is listed in Figure 8 (a). Figure 8 (b) shows the corresponding dataflow graph. Figure 8 (c) shows the generated VHDL code. And finally, Figure 8 (d) shows the simulation of the PEM. This experiment is similar to the previous experiment except that in this experiment

internal edges of the IF1 file are assigned communication cost. The edge between the source node, node 1 and the destination nodes, node 3 and 4 is assigned 20 ns communication cost. And the edge between the source node, node 2 and the destination nodes, node 3 and 4 is assigned 10 ns communication cost. Now the total parallel execution time should include communication overhead cost. The only difference between Figures 7 (c) and 8 (c) is the values of the transport delay in the signal assignment statements that correspond to the four communication edges in the IF1 graph. The transport delay in Figure 7 (c) is zero and in Figure 8 (c) is non-zero. Now signals N3I_1 and N4I_1 do not become one instantly at the same time when N1O_1 becomes one. They are delayed by 20 ns, i.e., they become one at time 170 ns (i.e., after 20 ns of signal N1O_1 becomes one) . Also signals N3I_2 and N4I_2 do not become one instantly at the same time when N2O_1 becomes one.
They are delayed by 10 ns, i.e., they become one at time 210 ns (i.e., after 10 ns of signal N2O_1 becomes one). Output A of node 3 becomes one after the 150 ns (execution time of node 3) of the availability of both inputs of node 3 (the two inputs of node 3 become available at time 210 ns). This is shown in Figure 8 (d) by transition of signal A from 0 to 1 at time 360 ns. Output B of node 4 becomes one after the 200 ns (execution time of node 4) of the availability of both inputs of node 4 (the two inputs of node 4 become available at time 210 ns). This is shown in Figure 8 (d) by transition of signal B from 0 to 1 at time 410 ns. The parallel execution time of the program is the time required for both outputs A and B of the program to become available which is 310 ns. This includes both computational time and communication time. The parallel execution time is visually shown in Figure 8 (b).

```
X        "main"
N1       141
E  0  1   1   1   4  %na = "x"
E  0  2   1   2   4  %na = "y"
N2       135
E  0  3   2   2   4  %na = "z"
E  1  1   2   1   4  %cost = 0
E  2  1   0   1   4  %na = "a"
```

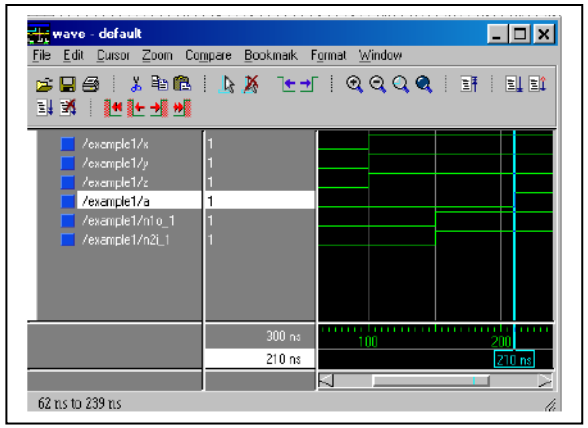
(a) IF1 code



(b) IF1 dataflow graph

```
Entity example1 is
   Port (x,y,z: IN bit;
         A : OUT BIT);
END Example1;
Architecture ex1_behavioral of Example1 is
signal N1O_1, N2I_1: bit;
begin
  N1O_1 <= transport ( X AND Y )  AFTER 50 ns;
  N2I_1 <= transport ( N1O_1 ) after 0 ns;
  A <= transport ( N2I_1AND Z ) after 60 ns;
End Ex1_behavioral;
```

(c) VHDL code



(d) Simulation

Figure 6: Computation of A := (x + y)/z

The sequential time of the program is still 600 ns which is the summation of the execution times of the nodes in the IF1 program. The communication overhead is not considered when calculating the sequential time because all nodes are executed by a single processor. The simulation proves the correctness and validity of the model. And the simulation results show that our model is accurate and powerful and the results coinside with the expectations.

The fourth experiment is shown in Figure 9. The experiment shows the multiplication of two 5x5 matrices. The 5x5 matrix multiplication program is shown in Figure 9(a). The corresponding IF1 file is listed in the appendix. Figure 9 (b) shows the corresponding dataflow graph. Figure 9 (c) shows the corresponding VHDL code. And finally, Figure 9 (d) shows the simulation of the PEM. As Figure 9 (d) shows, all elements of the resultant matrix C are computed in parallel. This is shown by them changing value simultaneously at time 50ns. This result coincides with the theoretically computed value since all elements of the input matrices A and B change value (become available) at time 20ns and it takes 30ns to do multiplication. This means that our model computes all c[i][j] for all i and j in parallel. This coincides with maximum possible parallelism in the matrix multiplication problem. The sequential time of the algorithm is $O(n^3)$ where n is the size of the matrix. $O(n^3)$ is equal to 5X5X5X30 ns which is the time required to compute all elements in matrix C sequentially. The parallel time of the algorithm is 5x30ns, which results in a speedup of n^2.

If we compare results of our approach to those of other existing approaches we find the following:

1. No research has been done in using VHDL to model parallelism in programs. Hence; our work is original, genuine and new.
2. Our approach gives more accurate results since it uses VHDL simulator. It uses deterministic approach. Other approaches use heuristic approaches to model parallelism. These approaches tend to partition the dataflow graphs into partitions where each partition is executed sequentially while there is inter-partition parallelism. Partitions are then scheduled onto different processors of a particular parallel computing system. Heuristics include minimizing some cost function such as communication overhead; scheduling overhead; total execution time; etc.

3. Our approach gives accurate results while other research gives approximate results.

Our approach shows maximum parallelism. The only constraint on parallelism is the data dependency between operations. Other approaches include other constraints on parallelism such as number of processors.

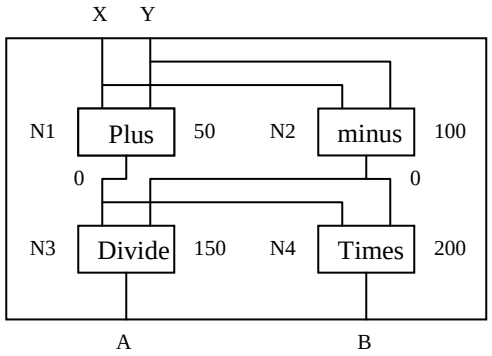
6. Future Work and Conclusion

A Parallel Execution Model (PEM) that can concurrently exploit parallelism in dataflow programs is presented in this paper. This model is based on VHDL and employs two compilation stages. The first stage takes an IF1 dataflow program and converts it into Java objects that represent the various components of the IF1 file. The second stage takes the Java objects produced by the first stage and generates VHDL code that represents the concurrent tasks of the IF1 dataflow program. The source code of the model is implemented using Java. To validate our model some simulation experiments were conducted. In these experiments we use our approach to compile some IF1 programs to the VHDL model, then we run the VHDL simulator to get the parallel execution time of the PEM. The VHDL simulator used for the verification and validation of the approach is VHDL ModelSim SE Version 5.5. The simulator runs under Microsoft Windows 2000 environment. The simulation results prove the correctness and validity of the model. In addition, simulation results show that our model is accurate and powerful and the results coincide with the expectations.

Work on the IF1-to-VHDL compiler is currently partially completed and is still under development. More work needs to be done to complete the compiler. Future work includes adding several optimization techniques in the translation of IF1 code into VHDL code. Another future work could be done in the topic of mapping the dataflow graph onto specific parallel platforms and getting the corresponding parallel performance metrics such as speedup, performance, and utilization.

```
X          "main"
N1         141
E  0  1  1  1  4      %na = "X"
E  0  2  1  2  4      %na = "Y"
N2         135
E  0  1  2  1  4      %na = "X"
E  0  2  2  2  4      %na = "Y"
N3         153
E  1  1  3  1  4      %cost = 0
E  2  1  3  2  4      %cost = 0
E  3  1  0  1  4      %na = "E"
N4         152
E  1  1  4  1  4      %cost = 0
E  2  1   4  2  4     %cost = 0
E  4  1   0  2  4     %na = "F"
```

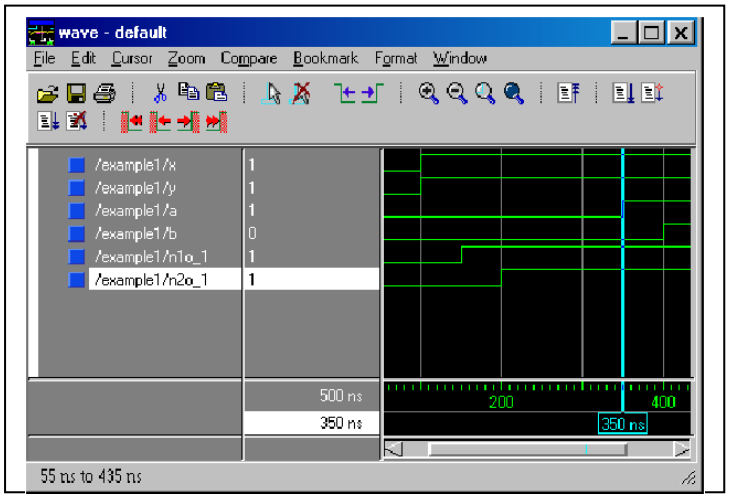
(a) IF1 code



(b) IF1 dataflow graph

```
Entity example1 is
  Port (x,y: IN bit;
        A,B : OUT BIT);
END Example1;
Architecture ex1_behavioral of Example1 is
signal N1O_1, N2O_1, N3I_1, N3I_2, N4I_1, N4I_2: bit;
begin
  N1O_1 <= transport ( x AND y )  AFTER 50 ns;
  N2O_1 <= transport ( x AND y )  AFTER 100 ns;
  N3I_1 <= transport ( N1O_1 ) after 0 ns;
  N3I_2 <= transport ( N2O_1 ) after 0 ns;
  N4I_1 <= transport ( N1O_1 ) after 0 ns;
  N4I_2 <= transport ( N2O_1 ) after 0 ns;
  A <= transport ( N3I_1 AND N3I_2 ) after 150 ns;
  B <= transport ( N4I_1 AND N4I_2 ) after 200 ns;
End Ex1_behavioral;
```

(c) VHDL code

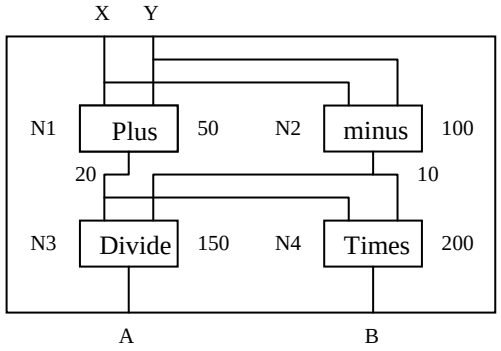


(d) Simulation

Figure 7: Computation of A := (x+y)/(x-y) ; B:= (x+y)*(x-y) with zero communication overhead

```
X          "main"
N1         141
E  0  1  1  1  4      %na = "X"
E  0  2  1  2  4      %na = "Y"
N2         135
E  0  1  2  1  4      %na = "X"
E  0  2  2  2  4      %na = "Y"
N3         153        `
E  1  1  3  1  4      %cost =20
E  2  1  3  2  4      %cost =10
E  3  1  0  1  4      %na = "E"
N4         152
E  1  1  4  1  4      %cost = 20
E  2  1   4  2  4     %cost =10
E  4  1   0  2  4     %na = "F"
```

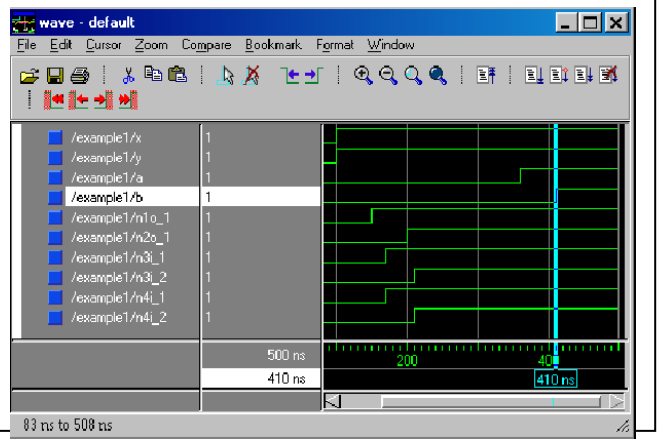
(a) IF1 code



(b) IF1 dataflow graph

```
Entity example1 is
  Port (x,y: IN bit;
        A,B : OUT BIT);
END Example1;
Architecture ex1_behavioral of Example1 is
signal N1O_1, N2O_1, N3I_1, N3I_2, N4I_1, N4I_2: bit;
begin
  N1O_1 <= transport ( x AND y )  AFTER 50 ns;
  N2O_1 <= transport ( x AND y )  AFTER 100 ns;
  N3I_1 <= transport ( N1O_1 ) after 20  ns;
  N3I_2 <= transport ( N2O_1 ) after 10  ns;
  N4I_1 <= transport ( N1O_1 ) after 20  ns;
  N4I_2 <= transport ( N2O_1 ) after 10  ns;
  A <= transport ( N3I_1 AND N3I_2 ) after 150 ns;
  B <= transport ( N4I_1 AND N4I_2 ) after 200 ns;
End Ex1_behavioral;
```

(c) VHDL code



(d) Simulation

Figure 8: Computation of A := (x+y)/(x-y) ; B:= (x+y)*(x-y) with non-zero communicaion overhead

```
Begin
    for i in 1..5 loop
        for j in 1..5 loop
            for k in 1..5 loop
                c(i,j) := c(i,j) + a(i,k)*b(k,j)
            end loop
        end loop
    end loop
end
```

Figure 9 (a) 5x5 matrix multiplication algorithm

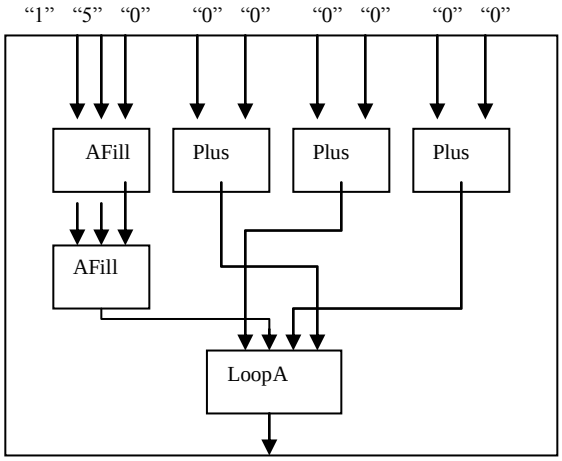


Figure 9 (b-1) Matrix multiplication IF1 Graph

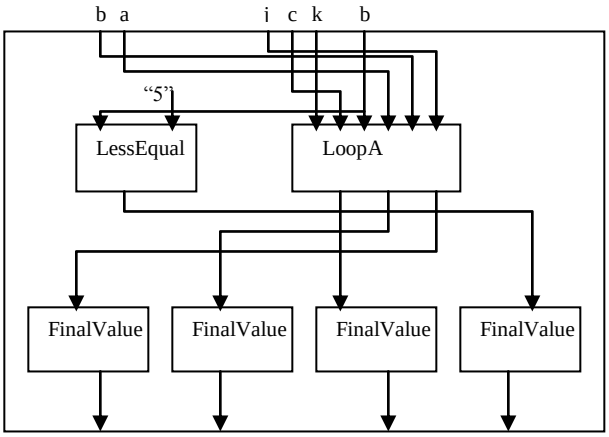


Figure 9 (b-2) Expansion of LoopA in Figure 9 (b-1)

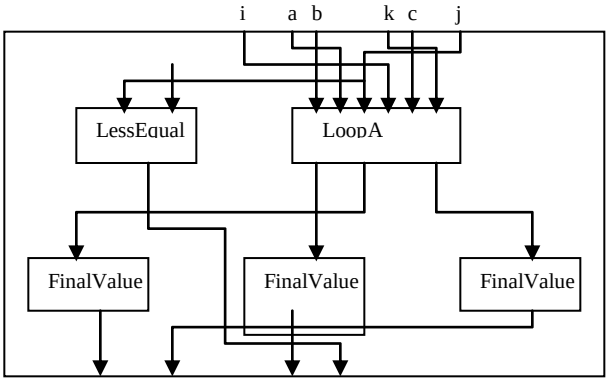


Figure 9 (b-3) Expansion of LoopA in Figure 9 (b-2)

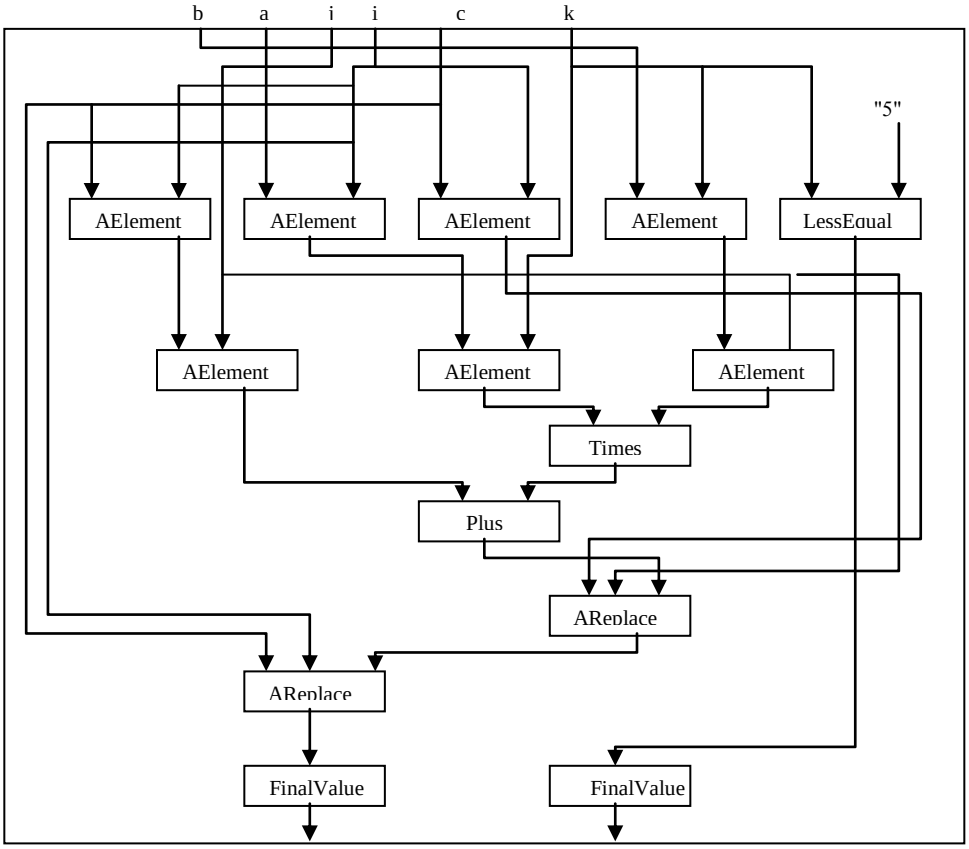


Figure 9 (b-4) Expansion of LoopA in Figure 9 (b-3)

Figure 9: 5x5 Matrix Multiplication (continued)

```
library ieee;
 use ieee.std_logic_1164.all;
package my_package is
        type Twod_array is array (0 to 4, 0 to 4) of bit ;
end my_package ;

library ieee ;
use ieee.std_logic_1164.all ;
use work.my_package.all ;
entity matrix_mult is
   port (a, b: in twod_array ;
        c : out twod_array)
end matrix_mult ;

architecture matrix_mult_behavioral of matrix_mult is
begin
  process(a,b) is
  begin
      for i in 0 to 4 loop
          for j in 0 to 4 loop
              for k in 0 to 5 loop
                  c(i,j) <= transport  c(i,j) OR  ( a(i,k) AND b(k,j)  )  after 30
ns ;
              end loop ;
          end loop ;
      end loop ;
  end process ;
end matrix_mult_behavioral ;

use work.my_package.all ;
entity tb is
end entity ;
```

Figure 9 (c) VHDL code

Figure 9: 5x5 Matrix Multiplication (continued)

```
Architecture tb_muliplier of tb is
        signal A,B,C: twod_array ;
        component matrix_mult is
                port (a, b: in twod_array ;
                        c : out twod_array ; )
        end component  ;
begin
        mm: matrix_mult port map(A,B,C       ; )
        process
        begin
           for i in 0 to 4 loop
                for j in 0 to 4 loop
                  A(i,j) <= '0'  after 10ns, '1' after 20 ns  ;
                  B(i,j) <= '0'  after 10ns, '1' after 20 ns  ;
                end loop  ;
           end loop  ;
                wait  ;
        end process ;
end tb_muliplier ;
```

Figure 9 (c) VHDL code (continued)

Figure 9: 5x5 Matrix Multiplication (continued)

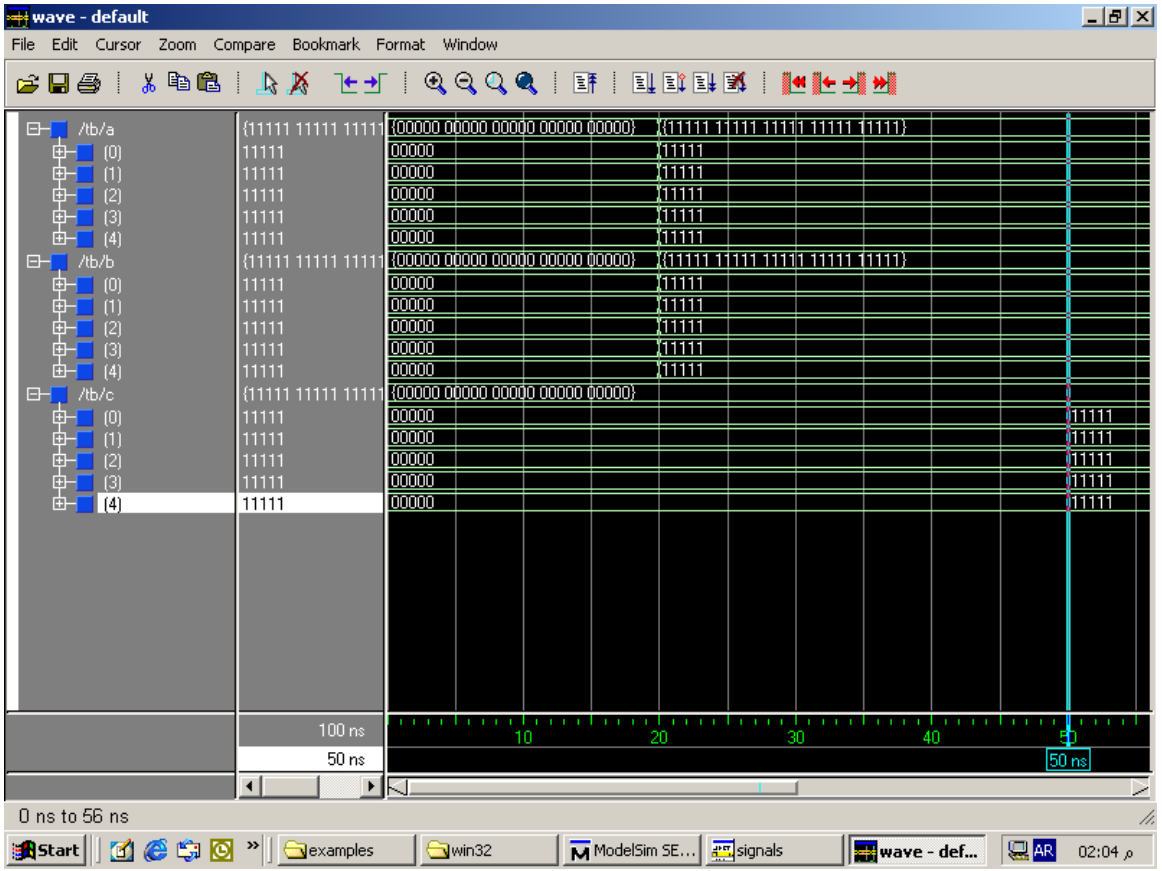


Figure 9 (d) Simulation

Figure 9: 5x5 Matrix Multiplication (continued)

Acknowledgement
The author would like to thank Eng. Husam Y. El-Zaq for his assistance in using VHDL ModelSim SE Version 5.5 in the validation stage of the develpoment stage of this paper.

Appendix: IF1 Code for Matix Multiplication Program

```
T       1       1       0               %na=Boolean
T       2       1       1               %na = Character
T       3       1       2               %na = Double
T       4       1       3               %na = Integer
T       5       1       4               %na = Null
T       6       1       5               %na = Real
T       7       1       6               %na = WildBasic
T       101     4       1               %na = Multiple(Boolean)
T       102     4       2               %na = Multiple(Character)
T       103     4       3               %na = Multiple(Double)
T       104     4       4               %na = Multiple(Integer)
T       105     4       5               %na = Multiple(Null)
T       106     4       6               %na = Multiple(Real)
T       107     4       7               %na = Multiple(WildBasic)
T       13      8       16      0       %na = b
T       12      8       16      13      %na = a
T       14      8       16      0       %na = c
T       11      3       12      14      %na = Procedure:matrix_mult
T       15      0       4               %na = array[OneDIM]
T       16      0       15              %na = array[TwoDIM]
T       116     4       16              %na = multiple{array[ThreeDIM]}

X 11 main  %sf=matrix2.a %fn =main
N 1   106
L       1       1       4       “1”
L       1       2       4       “5”
L       1       3       4       “0”
N 2     106
L       2       1       4       “1”
```

```
L       2       2       4       “5”
E       1       1       2       3       15      %na = c(2xD)
N 7     141                                     %op = 1
L       7       1       4       “0”             %mk = V
L       7       2       4       “0”             %mk = V
N 8     141             %op = 1
L       8       1       4       “0”             %mk = V
L       8       2       4       “0”             %mk = V
N 9     141     %op = 1
L       9       1       4       “0”             %mk = V
L       9       2       4       “0”             %mk = V
{ Compound      10      3
G       0       %fn = subgraph(init)
E       0       1       0       7       4       %na = j         %mk = V
E       0       4       0       8       16      %na = c(2XD)%mk = V
E       0       5       0       9       4       %na=k mk = V
L       0       10      4       “1”

G       0       %fn = subgraph(test)
N 1     132     %op=1
E       0       10      1       1       4       %na = i         %mk = V
L       1       2       4       “5”
E       1       1       0       1       1       %mk = V

G       0       %fn = subgraph (body)
{ Compound      1       3
G       0       %fn = subgraph(init)
E       0       1       0       7       4       %na = k         %mk = V
E       0       2       0       8       16      %na = c(2xD)    %mk = V
L       0       9       4       “1”

G       0       %fn = subgraph(test)
N 1     132     %op=1
E       0       9       1       1       4       %na = j         %mk = V
L       1       2       4       “5”
E       1       1       0       1       4       %mk = V
```

```
G       0       %fn = subgraph (body)
{ Compound  1       3
G       0       %fn = subgraph(init)
E       0       5       0       7       4       %na = c(2xD) %mk = V
L       0       8       4       “1”

G       0       %fn = subgraph(test)
N 1     132     %op=1
E       0       8       1       1       4       %na = k      %mk = V
L       1       2       4       “5”
E       1       1       0       1       1       %mk = V

G       0       %fn = subgraph (body)
N 1     105     %op=1
E       0       7       1       1       16      %na = c(2xD) %mk = V
E       0       4       1       2       4       %na = i      %mk = V
N 2     105     %op=1
E       1       1       2       1       15      %na = c(1xD) %mk = V
E       0       3       2       2       4       %na = j      %mk = V
N 3     105     %op=2
E       0       2       3       1       16      %na = a(2xD) %mk = V
E       0       4       3       2       4       %na = i      %mk = V
N 4     105     %op=2
E       3       1       4       1       15      %na = a(1xD) %mk = V
E       0       8       4       2       4       %na = k      %mk = V
N 5     105     %op=3
E       0       1       5       1       16      %na = b(2xD) %mk = V
E       0       8       5       2       4       %na = k      %mk = V
N 6     105     %op=3
E       5       1       6       1       15      %na = b(1xD) %mk = V
E       0       3       6       2       4       %na = j      %mk = V
N 7     152     %op=4
E       4       1       7       1       4       %mk = V
E       6       1       7       2       4       %mk = V
N 8     141     %op= 5
```

```
E       2       1       8       1       4       %mk = V
E       7       1       8       2       4       %mk = V
N 9     105     %op= 6
E       0       7       9       1       16      %na = c(2xD) %mk = V
E       0       4       9       2       4       %na = i      %mk = V
N 10    113     %op= 6
E       9       1       10      1       15      %na = c(1xD) %mk = V
E       0       3       10      2       4       %na = j      %mk = V
E       8       1       10      3       4       %mk = V
N 11    113     %op= 6
E       0       7       11      1       16      %na = c(2xD) %mk = V
E       0       4       11      2       4       %na = i              %mk = V
E       10      1       11      3       15      %mk = V
N 12    141     %op= 1
E       0       8       12      1       4       %na = k              %mk = V
L       12      2       4       "1"             %mk = V
E       11      1       0       7       16      %na = c(2xD) %mk = V
E       12      1       0       8       4       %na = k              %mk = V

G       0       %fn = subgraph (returns)
N 1     127     %op= 2
E       0       7       1       1       116     %na = c(2xD) %mk = V
E       1       1       0       1       16      %na = c(2xD) %mk = V
N 2     127     %op= 3
E       0       8       2       1       104     %na = k              %mk = V
E       2       1       0       2       4       %na = k              %mk = V
}       1       3       4       0       1       2       3    %op = 4
E       0       5       1       1       16      %na = b(2xD) %mk = V
E       0       4       1       2       16      %na = a(2xD) %mk = V
E       0       9       1       3       4       %na = j              %mk = V
E       0       3       1       4       4       %na = i              %mk = V
E       0       8       1       5       16      %na = c(2xD) %mk = V
E       0       7       1       6       4       %na = k              %mk = V
N 2     141     %op= 1
E       0       9       2       1       4       %na = j              %mk = V
L       2       2       4       "1"             %mk = V
```

```
E    1    2    0    7    4    %na = k              %mk = V
E    1    1    0    8    16   %na = c(2xD) %mk = V
E    2    1    0    9    4    %na = j              %mk = V

G    0    %fn = subgraph (returns)
N 1  127  %op= 2
E    0    7    1    1    104  %na = k              %mk = V
E    1    1    0    1    4    %na = k              %mk = V
N 2  127  %op= 3
E    0    8    2    1    116  %na = c(2xD) %mk = V
E    2    1    0    2    16   %na = c(2xD) %mk = V
N 3  127  %op= 4
E    0    9    3    1    104  %na = j              %mk = V
E    3    1    0    3    4    %na = j              %mk = V
}    1    3    4    0    1    2    3    %op = 5

E    0    9    1    1    4    %na = k              %mk = V
E    0    8    1    2    16   %na = c(2xD) %mk = V
E    0    10   1    3    4    %na = i              %mk = V
E    0    3    1    4    16   %na = a(2xD) %mk = V
E    0    2    1    5    16   %na = b(2xD) %mk = V
E    0    7    1    6    4    %na = j              %mk = V
N 2  141  %op= 1
E    0    10   2    1    4    %na = i              %mk = V
L    2    2    4    "1"       %mk = V
E    1    3    0    7    4    %na = j              %mk = V
E    1    2    0    8    16   %na = c(2xD) %mk = V
E    1    1    0    9    4    %na = k              %mk = V
E    2    1    0    10   4    %na = i              %mk = V

G    0    %fn = subgraph (returns)
N 1  127  %op= 2
E    0    7    1    1    104  %na = j              %mk = V
E    1    1    0    1    4    %na = j              %mk = V
N 2  127  %op= 3
E    0    8    2    1    116  %na = c(2xD) %mk = V
```

```
E    2    1    0    2    16   %na = c(2xD) %mk = V
N 3  127  %op= 4
E    0    9    3    1    104  %na = k      %mk = V
E    3    1    0    3    4    %na = k      %mk = V
N 4  127  %op= 5
E    0    10   4    1    104  %na = i      %mk = V
E    4    1    0    4    4    %na = i      %mk = V
}    10   3    4    0    1    2    3       %op = 6
E    8    1    10   1    4    %na = j      %mk = V
E    0    2    10   2    16   %na = b(2xD) %mk = V
E    0    1    10   3    16   %na = a(2xD) %mk = V
E    2    1    10   4    16   %na = c(2xD) %mk = V
E    9    1    10   5    4    %na = k      %mk = V
E    7    1    10   6    4    %na = j      %mk = V
E    10   2    0    1    4    %na = c      %mk = V
```

References

1- Chen A., and Wu C., January 1991 - *A Parallel Execution Model of Logic Programs,* IEEE Transactions on Parallel and Distributed Systems, Vol. 2, No. 1, P: 79-92

2- *Data Flow Computational Models: A brief overview and history*: www.cs.colostate.edu/ cameron/dataflow.html

3- Dennis J., Nov. 1997 - *A Parallel Program Execution Model Supporting Modular Software Construction*, Proceedings of the Third Working Conference on Massively Parallel Programming Models, 12-14, P: 50-60

4- Dewey A., **Analysis and design of digital systems**, International Thompson Publishing Inc., 1997

5- Kazi I., and Lilja D., Sep. 2001 - *Course-Grained Thread Pipelining: A Speculative Parallel Execution Model for Shared Memory Multiprocessors*, IEEE Computer, Vol. 12, No. 9, P: 952-966

6- Linghao S., and Takala J., June 4-7, 2001 - *Message Passing Integration for Large Scale Dataflow Computational Network*, Proceedings of International Conference on Telecommunications, Bucharest, Romania, Vol. 1, P: 428-432

7- Mills M., and Peterson G., November 8-12, 1998 - *Hardware/Software Co-design: VHDL and Ada 95 Code Migration and Integrated Analysis,* Air

Force Research Lab, Information Technology Division, Proceedings of the ACM SIGAda Annual International Conference on Ada Technology, Washington, DC, USA. ACM, P: 18-27

8- ModelSim SE– Version 5.5 VHDL Simulator: http://www.model.com/support/ documentation.asp

9- Lukito E. Nugroho L., and Sajeev A., June 23-25 1999 - *Java4P: Java with High-Level Concurrency Constructs*., Proceedings of the Fourth International Symposium on Parallel Architectures, Algorithms, and Networks, P: 328-333

10- Perry D., **VHDL**, Third Edition, McGraw Hill, 1998

11- Rinker R., Carter M., Patel A., Chawathe M., Ross C., Hammes J., Najjar W., and Bohm W., February 2001 - *An Automated Process for Compiling Dataflow Graphs into Reconfigurable Hardware*, IEEE Transactions on Very Large Scale Integration (VLSI) System, Vol. 9, No. 1

12- Skahill K., **VHDL for programmable logic**, Addison-Wesley Publishing Inc., 1996

13- Skedzielewski S., and Glauert J., July 1985 - *IF1 - An Intermediate Form of Applicative Languages*, Manual M-170, Lawrence Livermore National Laboratory, Livermore, CA, USA

14- Tsai J., and Li B.,October 26-29 1994 - *Improving Parallel Execution Performance for Logic Programs Using Mode Information*, Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing, P: 144-151

15- Williamson M., Spring 1998 - *Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications*, Ph.D. Dessertation, University of California, Berkeley, Berkeley, USA