A Parallel Approach to Faster VHDL Emulation Using Grid Processors

Muhammad Suleman, Siddharth Balwani
Department of Electrical and Computer Engineering
The University of Texas at Austin
Email: {suleman | balwani}@ece.utexas.edu

Abstract

This paper describes an architecture that is designed for the emulation of digital systems described in Very High Speed IC HDL (VHDL). We provide an alternate to the Field Programmable Gate Arrays (FPGA) widely used in industry for behavioral emulation of digital systems. The goal of the architecture is to reduce limitations imposed on emulation systems by FPGAs, e.g. restrictions on clock speed, and area. We use a grid processor approach to achieve this goal. This paper also includes an analysis of our design and how it compares with existing FPGA technology.

Introduction

The complexity of the designs that modern synthesis has to deal with follows Moore's law. Designs with several hundred million transistors are becoming common place. This makes functional verification a very hard task. Traditionally, simulation was used as a technique to verify the correctness of a design. However, the simulation speed is often not high enough. One way to speed up verification is by the use of emulators. Even though emulation speeds the process of verification by orders of magnitude, it still has its drawbacks. Many emulation systems are based on field programmable gate arrays (FPGAs). These limit the number of bits on a data path to a number far smaller than that demanded by the applications of today. For instance, due to area constraints it is extremely challenging, if not impossible to implement a 32-bit word length data path in FPGA technology [1].

The goal of our project is to efficiently solve the problem of fast and effective verification of circuits described in VHDL, while not being restricted by the same constraints that limit FPGAs. The system we propose is based on a Grid Processor Architecture (GPA). Some techniques from logic synthesis and technology mapping are proposed to make efficient use of hardware resources.

The outline of this paper is as follows. First we will discuss the semantics of the VHDL descriptive language and argue that VHDL is inherently parallel and not sequential. Later we will describe our hardware architecture and its characteristics. Next we will discuss the software aspects of the system. Finally we will analyze our system and compare it with FPGA based emulation systems.

VHDL Semantics and Identifying Parallelism

As mentioned by [2], VHDL is not inherently a sequential language but in fact is designed to allow parallel execution with deterministic results. [2] proposes use of parallel computers in order to simulate designs written in VHDL. [4] also describes the presence of multiple threads, that can be executed in parallel, in VHDL code.

VHDL code consists of entities and processes and a new thread can be instantiated by using a process statement. VHDL displays inter-process parallelism as well as intra-process parallelism, which is the parallelism that is the parallelism that exists within each process. [2]

These forms of parallelism displayed by VHDL are very similar to Thread level Parallelism (TLP) and Instruction Level Parallelism (ILP). If we use the abstraction of processes as being separate threads that can share data structures and the Register Transfer descriptions of processes as instructions, then the one-to-one mapping between the two concepts can be visualized. Figure 1 gives an example of a VHDL description and its corresponding matching to threads an assembly code.

VHDL Code	Assembly Code
Process A; A=B; C=A+8; Signal C; end . Process B; F=E; B=C+D; End	Thread 1: I1: MOV A, B; I2: ADD C, A, #8 I3: STC memC, C (Atomic store and signal) Thread 2: I4: MOV F, E I5: LD C, memC (Atomic load and signal)

Figure 1. Correspondence in Assembly and VHDL

Constraints and Design

This study of the common VHDL work load led us to clearly define our goals. Our objective was to use a processor architecture that could most efficiently exploit ILP and TLP while keeping the other emulation constraints in mind. Ideally a real-time reconfigurable hardware emulator should have multi-million ASIC gate logic capacity, identical system operation frequency to the final target system, and be able to seamlessly integrate multiple heterogeneous components. [5].

While keeping the above model in mind, we explored different parallel architectures that would allow us to satisfy our requirements.

We observed that the code within a VHDL process could be described in simple RT notation. This would imply that this code could be represented as a simple undirected graph G(V,E). We wanted to choose an architecture that would most closely replicate this structure. We decided on the abstraction of a graph H(V,E) that would connect processors using half-duplex interconnects. We wanted H(V,E) to be a super set of G(V,E). Ideally, any graph G(V,E) could be mapped on H(V,E) such that for every vertex v in G, there should be a node in H that has the same input and output characteristics. (We will also refer to G and H in later sections.)

In order to exploit the TLP, we need architecture capable of executing multiple threads with low overhead in the case of thread communication. The following sections will describe our proposed system that meets this specification.

Hardware Architecture

This section will give a conceptual description of the proposed architecture. We will discuss the specific implementation of this architecture for this project in the analysis sections.

The architecture we have chosen is the Grid Processor Architecture described in [7]. The grid processor has several characteristics that match our requirements. The processor grid can be easily visualized as the graph H(V,E) and any one process G can be mapped onto this grid. For simplicity let us assume that G can be always be mapped onto H. This implies that the ILP described in earlier sections can be exploited. The method we used to exploit TLP is discussed later in the paper.

Interconnect Network Topology

Our network is very similar to the one proposed in [6]. We have made some modifications keeping our constraints in mind to make the processor better fit our requirements. The processing elements (PEs) are connected to each other in the form of a 2-D mesh. Figure 2 (on next page) is a top level picture of the processor grid. Each arc between two nodes means a unidirectional 64-bit path between the nodes. Each node in the grid has an indegree of three and an outdegree of three. This does put some limitations on the fanout of gates in the graph G and a solution to this problem will be discussed in a later section.

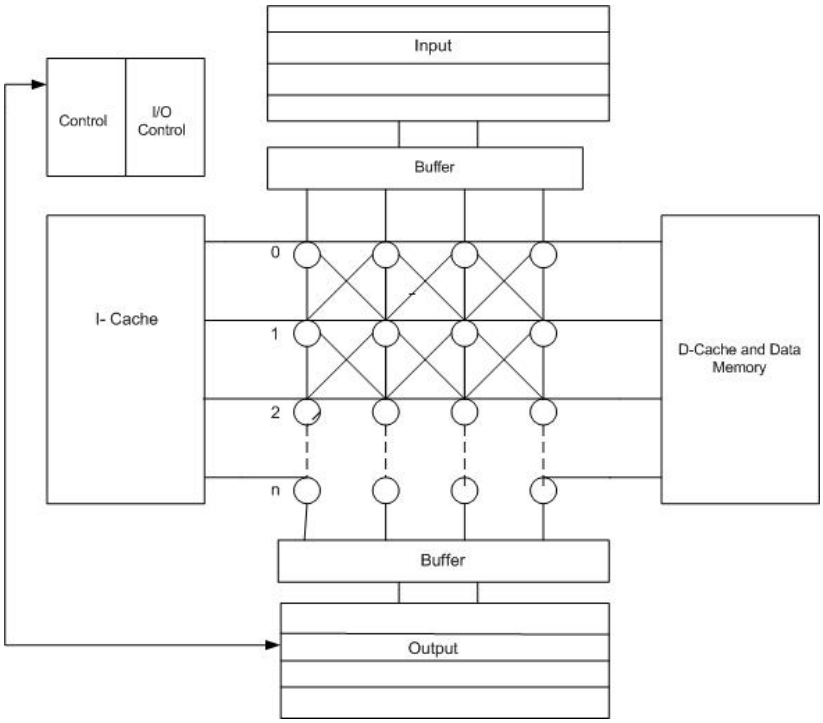

Figure 2: The 2-d mesh network

Communication over the network

The intra-process communication can be handled by the peer-to-peer network in the grid. The communication routes will be decided by the compiler and a Destination ID will be encoded in every instruction. Each node will do its function on the input as dictated by the instruction and will forward the result to the compiler specified destination. There are two types of communication over the network and we discuss them both in the following paragraphs.

There will be communication within each thread as long as the graph G can be mapped on graph H, this routing will just include forwarding the data to a node with the destination ID. If there does not exist a direct connection between the sending PE and destination PE, a *forward* instruction could be used at the intermediate nodes

The network also allows communication between two threads. There are two different ways of implementing this. The first is the case when there is a direct path to the destination node. In this case, the message can be sent over this path with a destination thread id padded to it. Otherwise if there does not exist a straight path to the destination node, the register file may be used to communicate the values. The compiler can find out data dependence at compile time and can create an arc between the two threads using the physical registers. In the rare circumstance that the system runs short of registers, the

compiler may use global shared memory for this communication. This, however, leads to a degradation in performance.

We will allow the processors on the perimeter of the grid to access memory, registers, inputs, and outputs from the chip. The bottom row will be expected to communicate with the Output devices while the top row gets the inputs from the Input pins. The I/O registers work exactly as PE and can send or accept data from to or the grid. The Instruction cache is directly connected to each element just as was proposed in [7]. These connections are shown in Figure 2.

Deadlock
The receive is non-blocking and data transfer is unidirectional and there is no request response protocol, hence there is no potential for deadlock in the system.

Network Node

Each network node consists of an ALU, a routing chip, an instruction scheduler, and on-chip data and instruction cache memory. Figure 3 shows the picture of a node. The various components of this node are described below:

Execution Core
The processing core is based on a simplified RISC processor. The functional units include an ALU that is capable of doing operations on one, two, or three data operands depending on the instructions. The operations supported by an ALU are common operations supported by a standard cell library. These include ADD, SUB, AND, OR, XOR, AOI32, etc. The ALU takes in the values at its input terminals and take an instruction word to be performed on the inputs. The functional units then forward their results to the router.

Instruction Scheduler
The instruction scheduler schedules the instructions to be executed by the ALU. The scheduler is essentially a set of reservation stations and an instruction is triggered as soon as all its operands are available. If two or more instructions become ready to execute at the same cycle one of them is randomly issued.

Router
The function of the router is to handle communication with the network. The router is capable of receiving up to three inputs in the same cycle. For each new input, it identifies the position where the data needs to be directed in the reservation station by looking at the Thread ID (TID) attached to the data.

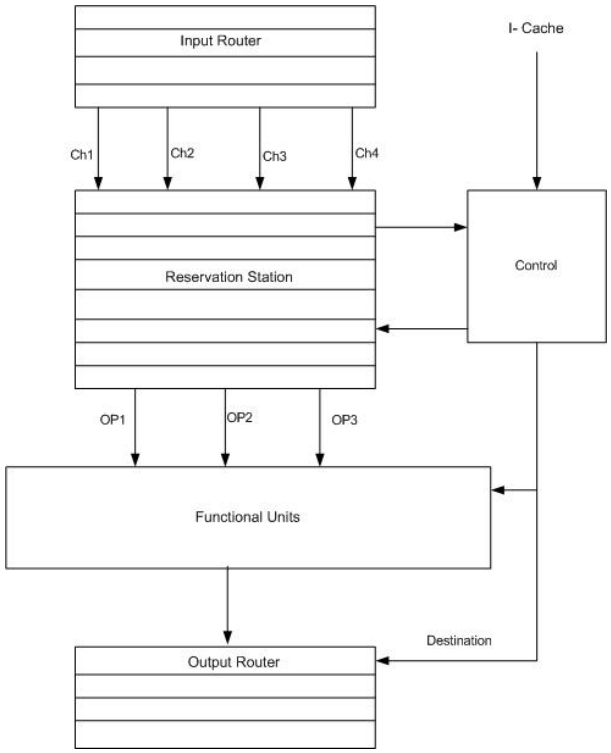

Figure 3: One network node

The router also outputs the data written to its buffer by the ALU. The destination PE is determined by the instruction itself. The frame number is also appended to the packet for the receiving router to correctly route the data in the reservations station.

Completion detection and latching

A completion is said to have reached when all the combinational netlists have been evaluated for the current clock cycle. Completion is detected when all the output slots have been filled and is report by the output register controller. When completion is detected the mapping engine again comes into play and maps the new set of instructions for the next clock cycle based on the state graph of the machine.

Instruction Fetch Mechanism

Instructions come in the form of maps produced by the compiler. The total execution can be thought of as a switch case statement in C. Whenever a block finishes execution the next block is fetched based on the current block and the inputs.

The next block is decided by doing a table look-up for the current state and the input states. An efficient method to implement this is to maintain a cache of next state calculations. The cache can be initialized to a specific value by the compiler and can be trained as we move along in time.

Next State Prediction

Next state prediction can be used as a technique to further optimize performance on the system. The next state can be predicted based on the recent history of the state and execution may be started. This will require the padding of stateID to the messages and use of stateIDs in the output processor. The prediction can simply be made void by resetting its entry in the output buffer and then assigning its frame ID to the correct block so the instructions get overwritten in the caches.

Memory accesses

The shared memory only needs to be accessed by the system if the number of global registers used for inter-thread communication does not suffice. Specific addresses in the memory network are then allocated for shared data by the compiler. In this case there are no likely conflicts except for RAW dependencies. These can be resolved by the compiler by allowing only one clock cycle to execute at any given time. During the clock cycle the threads are executed in a specific order determined by the mapper.

Input System

The system is basically an asynchronous state machine that communicates with the outside world and coveys the inputs to execution engine. In VHDL processes there is usually a sensitivity list where the output of a process is expected to be re-evaluated by the designer only if one of the sensitive inputs has been changed. This can be used to our advantage and the compiler can actually code the sensitivity information into a ROM in the input system. The input system will call the execution of the thread complete as soon as it is its turn to execute, if none of the sensitive inputs on the process have changed. Incase one of the sensitive inputs has changed, the input vector is fed into the grid and data is directed using the paths defined statically by the compiler. On receiving new inputs instructions are triggered in the execution grid and the frame is evaluated.

Output System

The Output system is also considered as a set of destinations PEs so data can be routed to them just like other PEs. An output state is considered ready when all the slots in the output register have been filled up. If all the output states are ready the output will be put in a buffer and will eventually become visible on the output pins. If the input system declares a block done, it communicates it to the output system and the system uses the same values as those in the previous entries in the buffer state for this entry as well.

Instruction Mapping

The process of mapping the RT description of a network onto our execution grid is called Instruction Mapping. This step is performed after compiling the behavioral VHDL code into RT notation and then performing High Level Synthesis. Once the HLS has been performed, the Instruction Mapping is very similar to technology mapping.

Instruction Mapping may be performed by taking our RT description and forming pattern graphs with our Execution Core as the basic standard cell library elements. Every ALU is considered as unit delay except when an ALU has to have a fan-out or fan-in of greater

than three. The cost the sub graph is then increased by a specific factor to reflect the penalty of using more than one ALU's to forward the result. Similarly when a process cannot fit in one map (when G cannot be wholly covered by H), we split the sub graph into two processes. This also requires the addition of extra penalty introduced due to overhead. Once all the sub graphs have been generated, we map each of our RT networks onto our processor grid. This is done using dynamic programming algorithm with the goal to minimize the delay and number of working functional units to minimize power and optimize frequency. The dynamic programming algorithm also performs combinational optimizations on the network to make it more suitable for use with our standard cell library. Each node is finally represented by an instruction that describes inputs and the pins the inputs will arrive on, the operation that needs to be performed on the inputs, and the destination node for the result of the instruction

Once every block has been mapped to the grid the next task is to assign order to the execution of these blocks. This is done using an algorithm proposed in [4]. The compiler tries to come up with a sorting of the processes such that non-blocking assignments which are clocked in parallel off the same clock must be sorted in order such that the effect of an earlier assignment does not influence the right hand side of any subsequent assignments. If the compiler is unable to come up with an efficient sequence for thread execution to eliminate coherency, it introduces intermediate variables. Once the ordering of the threads has been decided, threads are assigned thread IDs and are ready to be mapped onto the grid.

The branch statements in processor description pose a problem to our system. The nature of the VHDL language allows us to get some good performance benefits as the 'if and else' predicates may be evaluated either on the basis of inputs or the state variables. The predicates can hence be evaluated up front and appropriate instructions can be changed at the PEs. This would be done with the help of a controller in the system that will be used by the instruction fetch mechanism..

Another task of the compiler is to program the sensitivity list for the Input System ROM so that states that do not require evaluation can be ignored.

Analysis

In this section we will describe our implementation of the architecture. We will do an analysis of our design and derive the area, power, and delay characteristics of our implementation.

Since the 90-nm has very leakage current we will assume only static power for our analysis. We are also making a few assumptions as Switching Factor is concerned. We will use the SF for an ALU that approximates to be 0.5.

The equation
Power = (1/2)*(Cswitch density)*(chip area)*(SF)*(V^2)*(freq)

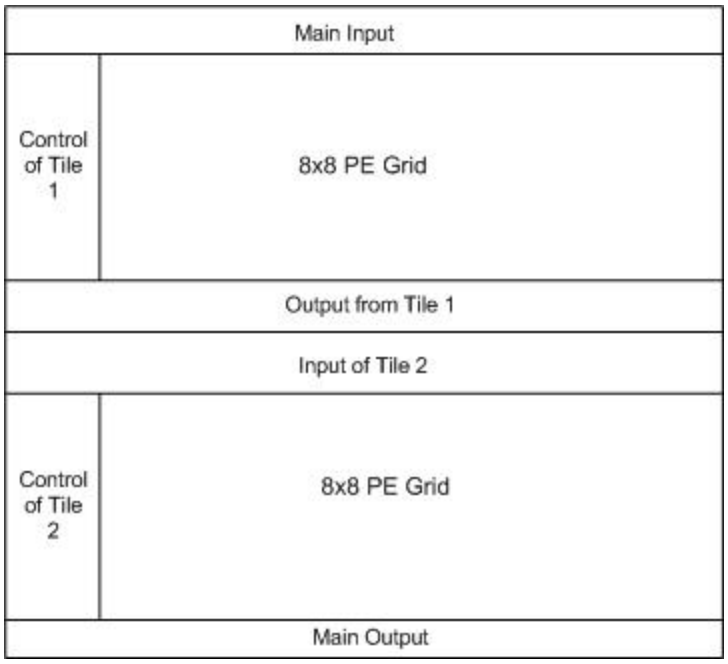

Figure 3: Floorplan of the Chip

Cswitch = 120 * (1.35 ^ 2) pF/mm^2

Area = 1 cm^2

Vdd=1V

Freq=10GHz

So the power calculated turns out to be, 21.87 Watts which is within our specs.

We estimate the area of the chip by using numbers from the Intel website for their 90-nm technology. A chip with 330 million transistors occupies an area of 109.08 mm^2 .By looking at table 1 , we obtained the following equation:

Area = 2 (0.665*64+34.20)= 153.59mm^2

A further optimization of the area would be to overlapping the area between the two chips. This would reduce the area and bring it to within our specifications.

TABLE 1: Summary of Area Calculations

Device	**Number of Transistors**	**Area / mm^2**	**Comments**
64 Entry Input Buffers	8192*6= 50000	.0166	100*64
64 Entry Output Buffers	8192*6=50000	.0166	100*64
Input Sensitivity List	32000*6= 1.5 Million	.499	32Kb
Input Control Logic	10000	3.3e-3	PLA
L2 Cache	100 Million	33.27	2 Mb
Output Control	100,000	.0333	
Instruction fetch	1 Million	.3327	
Next Block Prediction	100,000	.0332	
Total Number:	**102.81 Million**	34.207	
L1 Cache	384000	.12777	
Router inBuffer	115200	.0383	8 Kb
Router outBuffer	115200	.0383	
Router Buffer	50000	.0166	
ALU	5000	1.6e-3	
Control	1536000	.511	
Total	**2 Million**	.665	

Conclusion

In this project we designed a Grid Processor Architecture to emulate the parallel execution of VHDL. The main features of our processor as follows:

1. VHDL processes were mapped onto arrays of ALUs, permitting parallel execution of the code
2. The processor consists of 2 chips, each of which is a 64x64 array of simple processing elements, see figure 3.
3. The processing elements are connected to each other in the form of a 2D mesh
4. The processors use the register file for interthread communication. In the event the registers run out, shared memory is used.
5. The design is scalable, and the performance scales well with the number of transistors.

References

[1] G. Haug, U. Kebschull, W. Rosenstiel. A hardware platform for VLIW based emulation of digital designs.

[2] Sven Sköld and Rassul Ayani. Towards Parallel VHDL Simulation.

[3] R. Nagrajan, K. Sankaralingam, D. Burger, S. Keckler. A design space evaluation of grid processor architectures.

[4] Tension Technology, "A Verilog to C compiler."

[5] Chen Chang, Kimmo Kuusilinna1, Brian Richards, Robert W. Brodersen. Implementation of BEE: a Real-time Large-scale Hardware Emulation Engine

[6] K. Sankaralingam, R. Nagrajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen Keckler, Charles Morre. Exploiting ILP, TLP, and DLP with Polymorphous TRIPS Architecture