A CUSTOM ARCHITECTURE FOR DIGITAL LOGIC SIMULATION

by

Jiyong Ahn

B.S in E. E., Chung-Ang University, Seoul, Korea, 1985

M.S in E. E., Chung-Ang University, Seoul, Korea, 1987

M.S. in E. E., University of Pittsburgh, 1994

Submitted to the Graduate Faculty
of the School of Engineering
in partial fulfillment of
the requirements for the degree of
Doctor
of
Philosophy

University of Pittsburgh
2002

COMMITTEE SIGNATURE PAGE

This dissertation was presented

by

Jiyong Ahn

It was defended on

January 30, 2002

and approved by

(Signature)_____
Committee Chairperson
Raymond R. Hoare, Assistant Professor, Department of Electrical Engineering

(Signature)_____
Committee Member
Marlin H. Mickle, Professor, Department of Electrical Engineering

(Signature)_____
Committee Member
James T. Cain, Professor, Department of Electrical Engineering

(Signature)_____
Committee Member
Ronald G. Hoelzeman, Associate Professor, Dept. of Electrical Engineering

(Signature)_____
Committee Member
Mary E. Besterfield-Sacre, Assistant Professor, Dept. of Industrial Engineering

ACKNOWLEDGMENTS

I would like to express my thanks to my advisor, Prof. Ray Hoare for his guidance and friendship. I would also like to thank to all of my committee members Prof. Mickle, Prof. Cain, Prof. Hoelzeman and Prof. Sacre for their insights.

I would like to express my appreciation to my friends, Yee-Wing, Tim, Majd, and Jose, Michael Grumbine and Sandy for their long term friendship. And I also thank to my office mates. Special thanks to Dave Reed and Sung-Hwan Kim who provided me valuable help and encouragement. I also owe thanks to my colleagues over at Pittsburgh Simulation Corporation, Jess, Mike, Dave, Harry and Gary. I would like to thank to Mr. and Mrs. Paul and Colleen Carnaggio for their support and understanding.

Lastly, I would like to express my most sincere appreciation and affection to my family. My parents and my sisters had to endure for a very long time. I especially thank to my wife Okhwan for her patience and encouragement. They are my inspiration to reach the destination of this long and frustrating journey.

ABSTRACT

Signature_____
Raymond R. Hoare

A CUSTOM ARCHITECTURE FOR DIGITAL LOGIC SIMULATION

Jiyong Ahn, Ph. D.

University of Pittsburgh

As VLSI technology advances, designers can pack larger circuits into a single chip. According to the International Technology Roadmap for Semiconductors, in the year 2005, VLSI circuit technology will produce chips with 200 million transistors in total, 40 million logic gates, 2 to 3.5 GHz clock rates, and 160 watts of power-consumption. Recently, Intel announced that they will produce a billion-transistor processor before 2010. However, current design methodologies can only handle tens of millions of transistors in a single design.

In this thesis, we focus on the problem of simulating large digital devices at the gate level. While many software solutions to gate-level simulation exist, their performance is limited by the underlying general-purpose workstation architecture. This research defines an architecture that is specifically designed for gate-level logic simulation that is at least an order of magnitude faster than software running on a workstation.

We present a custom processor and memory architecture design that can simulate a gate level design orders of magnitude faster than the software simulation, while maintaining 4-levels of signal strength. New primitives are presented and shown to significantly reduce the complexity of simulation. Unlike most simulators, which only use zero or unit time delay models, this research provides a mechanism to handle more complex full-timing delay model at pico-second accuracy. Experimental results and a working prototype will also be presented.

DESCRIPTORS

Behavioral Modeling

Discrete Event Simulation

Hardware Logic Emulator

Hardware Logic Simulator

I/O-path and State Dependent Delay

Multi-level Signal Strength

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

Table No. Page

1.0 INTRODUCTION

As VLSI technology advances, designers can pack larger circuits into a single chip. According to the International Technology Roadmap for Semiconductors[(1)*], VLSI circuit technology in the year 2005 will produce chips with 200 million transistors in total, 40 million logic gates, 2 to 3.5 GHz clock rates and 160 watts of power-consumption. At this rate, a one billion-transistor chip may be less than ten years away; however, current design methodologies can only handle tens of millions of transistors in a single design.

The final output of the Electronic Design Automation (EDA) software is a synthesized circuit layout that can be fabricated. Figure 1 shows the design automation domains. The design automation process starts with a high-level design specification that is transformed into a physical design that can be fabricated. The upper right branch of the design automation *Y Chart*[(2)] is the behavioral domain. In this domain, the circuit design and fabrication technologies are described. Synthesis turns these descriptions into components in the *Structural Domain*, shown as the left branch of the *Y Chart*. Structural components can be transformed into the physical domain for fabrication. This research focuses at chip simulation in the structural domain at the gate and flip-flop level.

* Parenthetical references placed superior to the line of text refer to the bibliography.

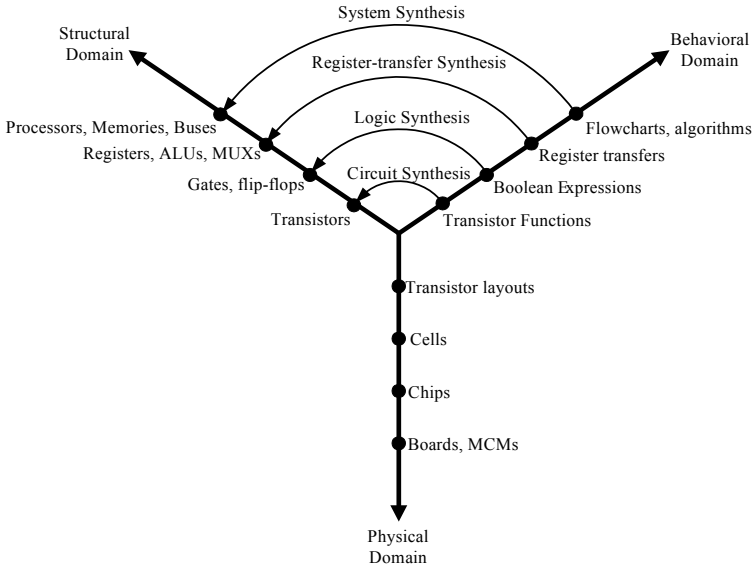


Figure 1 Y-Chart[2]

This research is motivated by the desire to create large devices that encapsulate entire systems on a single chip. These "System on a Chip" (SoC) designs typically incorporate a processor, memory, a bus, and peripheral devices. Due to the large design task, portions of the SoC may be purchased as Intellectual Property (IP) blocks and are typically described at the behavior level and at the gate (or mask) level. Once incorporated into the design, the entire design must be verified for correctness in functionality and in timing. Gate-level simulation is required for design verification at the pico-second level. Power consumption and thermal topology analysis are also required for modern high-speed IC design. Such simulation can require many hours to many days to complete. This adversely affects the design cycle time and thus, the time to market.

1.1 System on a Chip

Due to the complexity of a *System on a Chip* (*SoC*), there can be several million logic gates in a single design. Table 1 summarizes the size of current top of the line processors manufactured by AMD[3] and INTEL[4]. These growing number of transistors and gates in a single design will severely impact every aspect of the design automation process, simply because the size of data that the design automation tools have to handle becomes prohibitively large. This is because EDA tools normally rely on generic workstations for their platform. Therefore, even the highly efficient EDA algorithms are limited to the performance and the capacity of the workstation on which they are running.

Table 1 CPU Comparison[3,4]

Core	**K7**	**K75**	**Thunderbird**	**P-III Katmai**	**P-III Coppermine**
Clock Speed	500-700MHz	750-1100MHz	750-1100MHz	450-600MHz	500-1133MHz
L1 Cache	128KB	128KB	128KB	32KB	32KB
L2 Cache	512KB	512KB	256KB	512KB	256KB
L2 Cache speed	1/2 core	2/5 or 1/3 core	core	1/2 core	core
Process Tech	0.25 micron	0.18 micron	0.18 micron	0.25 micron	0.18 micron
Die Size	184 mm^2	102mm^2	120mm^2	128mm^2	106mm^2
TR count	22 million	22 million	37 million	9.5 million	28 million

1.2 Design Verification through Simulation

Logic simulation is one of the fields in EDA that the hardware designers depend on for the design verification and gate-level timing analysis. As designs get complex, designers rely on the performance of the logic simulation to verify the design's correctness at the various levels of abstraction. Logic level is the preferred level for the

designers to test their design because levels higher than the logic level (*i.e.*, Register-transfer level and above) are not accurate enough to extract the performance of the design and the level below the logic level (*i.e.*, transistor level and below) requires too much computing time. Designers can simulate their design before they synthesize the design (*i.e.*, pre-synthesis simulation), after the design has been synthesized (*i.e.*, post-synthesis simulation), and/or after the gates have been placed in particular locations of the chip (*i.e.*, post place-and-route).

Pre-synthesis simulation typically uses a hardware description language (HDL) (*e.g.* VHDL, Verilog) to describe the circuit. Simulation at this level uses a *delta-delay* model that assumes that the gate delays are a delta-time that is so small as to not be noticeable except in the ordering of events. Wire and gates delays are ignored. These assumptions greatly increase the speed of the simulation but they do not give the design accurate timing information. This level of simulation is used to verify accuracy of the high-level design and control mechanisms.

As shown in Figure 2, synthesis transforms the HDL into a gate-level description of the circuit. This design step can be lengthy because a single line of HDL can be synthesized into hundreds of gates (*e.g.* arithmetic operations). However, at the gate-level there is a one-to-one relationship between each gate and its standard-cell transistor layout. Each gate or flip-flop represents from two to fifty transistors, but the layout of these groupings of transistors is known. Thus, once a design is at the gate level, a technology can be specified and accurate timing information can be achieved.

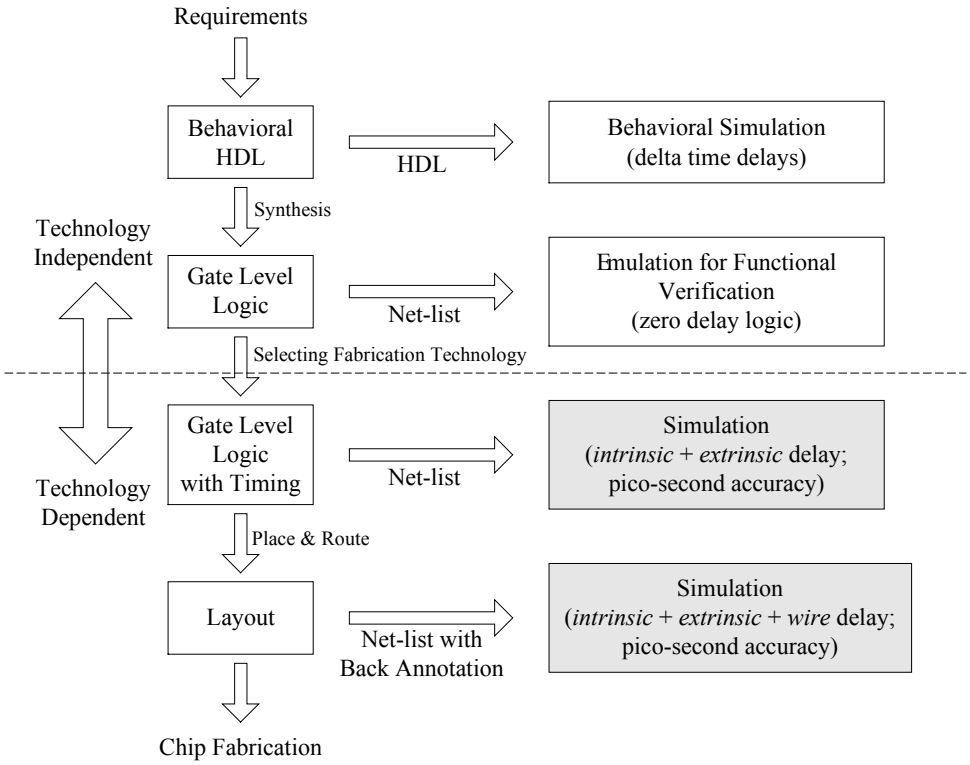


Figure 2 Design Flow

Before a technology is chosen, the circuit's functional behavior can be emulated. In this design phase, the gate-level design is mapped to a reconfigurable architecture that emulates the circuit's behavior. Emulation can be used to verify the functional behavior of a circuit, but does not accurately represent the actual timing of the circuit, because emulation is technology independent.

Gate-level simulation can also be used to determine the functional behavior of a circuit, but is slower than emulation, because simulation incorporates technology-specific gate delay to determine a circuit's behavior. The advantage of gate-level simulation is its accuracy. This level of simulation is useful in determining the technology that is required for each level of circuit performance.

After the circuit's functional behavior is verified and the technology is chosen, the location of each gate within the circuit is determined. This phase is called place-and-route because each gate's VLSI implementation is placed within the chip area and wires are routed among the different gates to implement the specified circuit. After this phase is performed, the wire delay between the gates can be incorporated into the simulation. At this point, the timing of the circuit can be estimated in pico-seconds (10^{-12} seconds). The problem with simulating gates at this level of accuracy is performance. This research focuses on increasing the speed of this level of simulation.

1.3 Intellectual Property Blocks

Some circuit elements are now made as a package and are being sold separately in the form of Intellectual Property (IP) blocks. There are two types of IP blocks. One is called *Hard IP*, which is in the form of mask layout, and the other is *Soft IP*, which is in the form of gate-level description called a net-list. In either case, the designers need to test/simulate these IP blocks along with their own design to verify its functionality and timing. Therefore, the speed of the logic simulation becomes more critical in design automation when the size of the design grows larger and IP blocks are incorporated.

One of the hurdles in using IP blocks from another company is verification. When IP is purchased, the customer typically gets a behavioral level description of the block that describes the timing and functionality of the circuit. However, this description typically can't be synthesized to ensure the designer's work is protected. The customer

also gets a gate-level description of the circuit that can be incorporated into their design. However, the gate-level design is typically technology independent, and therefore the timing information that describes the IP block is not accurate. Using technology-specific gate-level simulation, this timing information can be obtained.

1.4 Time to Market

In modern digital system design, reducing the "time to market" is critical to achieve success. Therefore, reducing the design cycle time is also critical. As previously described, each level of the design cycle increases the accuracy of the results but also increase the amount of time to achieve these results. If the design is not fully verified at each level of the design cycle, errors will propagate to the next level and the design cycle time will increase. Modern designers cannot tolerate this type of design cycle roll back, because it wastes the designer's time and increases the time-to-market.

One of the critical steps in the design phase is the verification of the entire chip after it has been completed and is ready to be fabricated. If a timing glitch is not found before fabrication, then months of design cycle is wasted, and the chip will still need to be simulated to find the error. Thus, it is critical to improve the performance of post-place-and-route simulation to decrease the time-to-market.

1.5 Test Coverage and Fault Modeling

In addition to the logic verification, designers need to know whether the set of simulation input vectors cover the entire circuit testing. Without this information, it is difficult to know if the design is fully tested with the given set of input vectors. If the circuit design has a large number of inputs, then the total number of input vector combination becomes astronomically large. In such cases, the designer wants to test the design with only a subset of input vectors. Therefore, some mechanism to check if the input vector set has covered the entire data path of the design must exist.

During circuit fabrication, a wire can be shorted to V_{dd} or to *Ground*. These *stuck-at faults* cause circuits to behave in an unpredictable manner if the circuit is not designed for such situations. To determine if these faults have occurred, test vectors need to be developed. Verification of these test vectors is time consuming because the circuit must be simulated thousands of times as these faults can occur for every wire network. Thus, hardware acceleration for logic simulation with stuck-at faults would reduce the development time for fault detection.

1.6 Power Consumption Computation

Power consumption in modern VLSI design is an important issue. Portable and/or hand-held electronic equipments are continuously emerging in the market. The power consumption of these devices will not only determine the battery life, but also decide the heat characteristics of a chip. As mentioned earlier, modern digital circuits are

getting smaller in size and faster in speed. The clock frequency of the chip plays a major role in power dissipation. When a chip consumes a lot of power, it inevitably becomes hot. Due to the thermal characteristics of a silicon device, over-heated chips will not function correctly. Modern high-speed processors all require a solid cooling mechanism to operate properly.

If the thermal topology can be pre-determined (before the chip is fabricated), the layout generation process can reference the thermal characteristics so that "hot spots" can be evenly distributed over the chip area. Such chips will run much cooler and less erratic when used in extreme conditions. Designers are now facing another problem of this heat issue. Due to the "time to market" constraint, thermal topology analysis cannot be done after the chip is fabricated. Therefore, it is important to extract the thermal characteristics of the design at both pre- and post-place-and-route steps.

Power consumption of a chip is described as the sum of *static dissipation* and *dynamic dissipation*. Static dissipation is due to the leakage current that is caused by the reverse bias between the diffusion region and the substrate. Static dissipation is small in value, and can be treated as a constant if the target technology is known.

Dynamic dissipation is due to the load capacitance and the transition activity of the output transistor. When a gate changes its output state (either from '0' to '1' or '1' to '0'), both p- and n-transistors are on for a short period of time. This results in a path from V_{dd} to *GND* such that power and ground is electrically shorted for a brief period of time, and power is consumed through this path. Current flow is also needed to charge and discharge the output capacitive load.

Dynamic dissipation is formulated[5] as:

$$P_d = C_L V_{dd}^2 f_p$$

Where C_L is the load capacitance, V_{dd} is supply voltage and f_p is the switching frequency.

Assuming supply voltage is constant, the dynamic power dissipation depends on the number of output changes of a gate and its capacitive load. Therefore, accurate simulation and recording of a circuit's switching behavior would provide critical insight into a circuit's thermal and power characteristics.

2.0 LOGIC SIMULATION

The current design process relies on a software-based logic simulator running on high performance workstations. Advanced processor and system architectures with generous amounts of memory can increase the performance of logic simulation to a degree, but eventually hit a performance barrier due to their memory access bottleneck and due to their general purpose design. Software benchmarks will demonstrate this point in Section 2.3 .

Logic simulators, especially software-based logic simulators[35], have been around for decades. Logic simulators are widely used tools to analyze the behavior of digital circuits, to verify logical correctness, and to verify timing of the logic circuits. Logic simulators are also used for fault analysis when a test engineer wants to determine the information about faults that are detected by a proposed test sequence[6].

Unlike circuit simulators (*e.g.* SPICE), which compute continuous time characteristics of the transistor-level devices, logic simulators rely on abstract models of digital systems that can be described using Boolean algebra. Logic simulators model a gate as a switching element with an intrinsic time delay that remains in steady-state when its inputs remain constant. They yield discrete output values as opposed to analog output (*e.g.*, SPICE simulation)[6]. However, current gate models have timing characteristics that specify timing down to 10^{-12} seconds.

For gate-level simulation, circuits are described in terms of primitive logic gates and their connectivity information. Such gate-level circuit descriptions are called net-

lists because they describe a network of interconnected gates. The primitive logic gates are typically evaluated by table look-up or by calling a software function.

2.1 Logic Simulation Algorithms

There are two main categories of algorithms in logic simulation. They are the *compiled approach* and the *discrete event-driven approach*.

2.1.1 Compiled Approach

To determine the logic behavior of a circuit, the *compiled approach* transforms the net-list into a series of executable machine-level instructions. Since the "Arithmetic and Logic Unit" (ALU) of a general-purpose processor is usually equipped with logical computation functionality, a net-list can be mapped directly into the machine code to perform logic simulation. The problem of this approach is that all the gates in the circuit are evaluated regardless of the input change. In addition, the compiled approach can only handle a zero- or unit-delay model, with a limited number of fan-in and fan-out due to the width of the instruction set.

2.1.2 Discrete Event Driven Approach

Instead of evaluating every gate as in the compiled approach, event-driven simulation considers a change in an input signal as an event. Gates are only evaluated when an event occurs.

Figure 3 illustrates how the algorithm works. Gates are labeled G1 through G10, and events are labeled E1 through E3. Consider a change in the input signal *c* from '0' to '1'. This *event* triggers the evaluation of G1, which generates an output change from '0' to '1'. The output change of G1 becomes a new event E1, which triggers the evaluation of G4 and G6. The evaluation of G6 generates an output change that will generate another new event E2. Event E2 triggers the evaluation of G8 that in turn generates the output change and a new event E3. Notice that the new event in G4 is evaluated but does not generate any new event, because the input *i* is '0', which forces G4 to hold the output value unchanged.

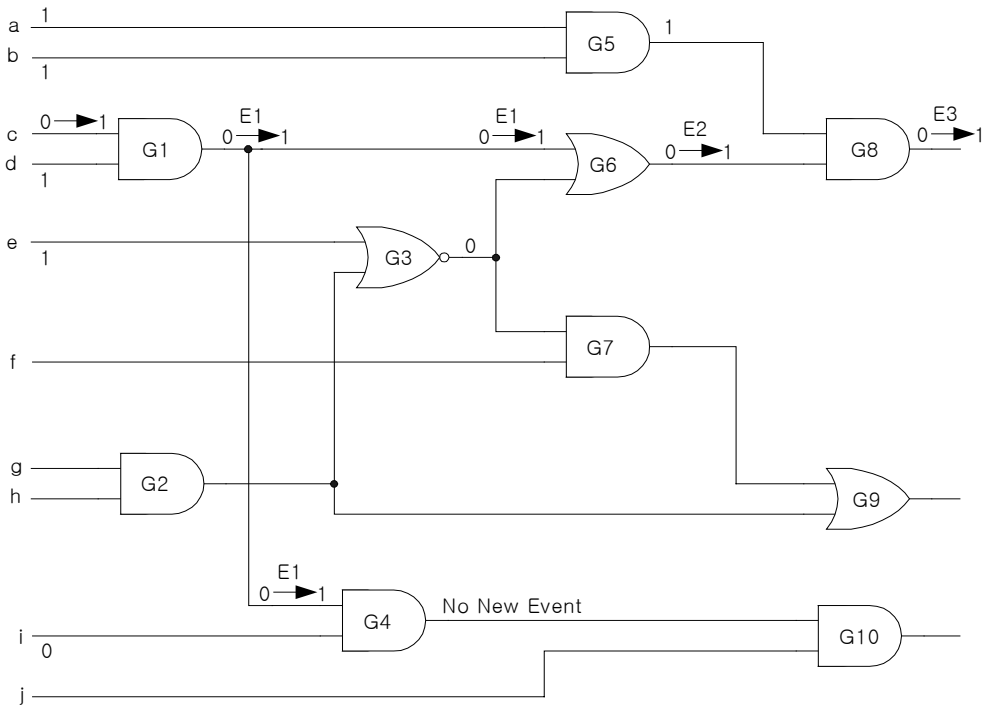


Figure 3 The Discrete Event Logic Simulation

A change in the output signal of a gate at time *t* will generate a *future event* that will occur at time $t+\delta$, where δ = *intrinsic_delay* + *extrinsic_delay* + *wire_delay*. *Intrinsic_delay* is based on the type of gate being implemented. For example, an inverter has a smaller *intrinsic_delay* than an exclusive-or gate because it can be implemented with fewer transistors. *Extrinsic_delay* is the delay due to the capacitive load that must be overcome to change the logic level. A gate with a high fan out will have a higher *extrinsic_delay* than a gate with a smaller fan out. The *wire_delay* is due to the capacitive load placed on the circuit due to the output wire length.

These future events are usually stored in a separate data structure to keep track of the correct time ordering of events. Thus, the simulation algorithm can safely access the events without executing them out of order[6, 7]. If a gate *G* is simulated due to an event

E, and it is determined that the output has changed its state, then all the gates that are driven by this output signal have to be simulated at the future time instance $t + \delta$, as described above. Logic gates usually have more than one fan-out, and thus, multiple future events can be generated as a result of evaluating one gate. These future events have to be managed/scheduled according to their timing information so that all the events can be evaluated in correct time order.

2.2 Related Work

There are several research projects that speed up discrete event logic simulation. Depending on the approach, we can classify them in two distinct groups. One is approaching from the parallel computing environment and the other is using the a custom hardware accelerator.

The use of a parallel computer can also be classified as compiled approach and discrete-event approach. There are parallel-compiled approach and parallel discrete event simulations. The parallel compiled approach still maintains its weakness as in the single processor case, that is, they can only handle unit- or zero-delay models[6]. Such delay models do not provide enough information about the circuit being simulated, therefore the compiled approach will not be covered in detail.

2.2.1 Parallel Discrete Event Logic Simulation Algorithms

In a parallel computing environment, each processor is called a *Processing Element* (PE). Each PE may have its own memory (distributed memory model) or may share one big memory (shared memory model). Both models require that PE's communicate with each other to ensure the correctness of the task they are processing (*e.g.* data dependency).

A major difference in parallel discrete event simulation is in the mechanism of managing the *simulated time*. In a parallel computing environment, the input net-list is partitioned and mapped into each of the Processing Elements (PE's). In such a case, the new events generated as a result of logic evaluation in one PE can affect the event execution order in other PE's. For example, if PE_0 generates a future event E_0 with time stamp t_0, that has to be sent to PE_1 (because the gate that this event is connected to is stored in PE_1), and if PE_1 is currently simulating an event E_5 with time stamp t_5, then it is called the *violation of causality constraint*, because E_0 should execute before E_5. This violation occurs when the events are executed out of order. In such case, it is possible that all of the work that has been done until *current simulation time* becomes void. Simulation time has to be rolled back to consider the propagated past event and the circuit has to be re-simulated.

Since each PE has no way of knowing when the new events will be propagated from other PE's, PE's cannot perform the simulation tasks independently from each other. The simulation time has to be controlled globally to ensure the correctness of the

simulation task. To control the simulation time across all PE's, the concept of *Global Virtual Time* (GVT)[12] is used for global synchronization. GVT is a notion of simulation time that all PE's must follow. If one PE is lagging behind the GVT due to the load, all other PE's must wait to be synchronized.

The parallel discrete event logic simulation can be classified in two categories: synchronous and asynchronous. They are both based on the mechanism to control the simulation time so that the causality constraint can be satisfied.

2.2.2 Synchronous Algorithm.

The synchronous approach follows the sequential simulation algorithm for each PE. It performs updates in parallel and evaluation in parallel[8]. Global synchronization is needed for each time instance. In other words, GVT can advance only when all the PE's agree on it. Soule *et. al.*[8] have implemented the synchronous parallel algorithms on an Encore Multimax shared memory multiprocessor using a centralized event queue for all PE's and reported a speedup of 2 on eight processors. When they used distributed event queues, they were able to achieve a speedup up to 5 on eight processors. Banerjee[6] states that synchronous parallel logic simulation algorithms typically achieve speedups of 5 or 6 on eight processors even though the inherent concurrency is quite high. This is because the load balancing of the parallel system and the synchronization overhead.

2.2.3 Asynchronous Algorithm: Conservative and Optimistic Approaches

Unlike the synchronous approach, the asynchronous approach allows the simulation time to be advanced for each PE locally without global synchronization. Since the simulation time is advanced independently, the algorithm is prone to a deadlock situation[9]. As an example, suppose 3 processors, A, B and C, are sending and receiving messages to and from each other and from the external world. If A is processing a message from C with time stamp 12, and B is processing a message from A with time stamp 8, and C is processing a message from B with time stamp 10, and each processors received a message with higher time stamp (*e.g.* 20, 30, 40 each) from the external world, then each processor cannot determine whether it is safe to process the external messages due to the gap in time stamp, and they block themselves waiting for messages from each other to fill the time stamp gap.

Based on the methods used to handle this deadlock situation, asynchronous approaches fall into two categories: *conservative* and *optimistic*. The conservative approach strictly avoids the possibility of violating the causality constraint by relying on a mechanism to determine when it is safe to process an event. If a processor contains an unprocessed event *E* with time stamp *T* and no other event with a smaller time stamp, and that processor can determine that it is impossible for it to receive another event with a time stamp smaller than *T*, then that processor can safely process *E* because it can guarantee that doing so will not later cause a violation of the causality constraint[6]. This requires a lot of inter-processor communication for querying each other's state. This can

potentially cause a deadlock. Some conservative algorithms use a NULL message to avoid deadlock situations[10, 11]. And are classified as deadlock avoidance algorithms. Speed up of 5.8 to 8.5 were reported by Soule *et. al.* using Encore Multimax Multiprocessor with 14 PE's[8].

The optimistic approach allows causality errors to occur, and then detects and recovers from these errors by using a rollback mechanism. Time Warp, proposed by Jefferson[12], is the most well known optimistic algorithm. In Time Warp, a causality error is detected when an event message is received that contains a time stamp smaller than that of the current event being processed. A recovery process is accomplished by undoing the effects of all events that have been processed prematurely by the processor. An event might have done two things that have to be rolled back. It might have changed the output of a logic gate, and/or it might have sent an event to the other processors.

The optimistic approach has better CPU utilization as compared to the conservative approach, but when there are rollbacks, some of the CPU cycles previously used for computation are wasted. A speed up from 6.5 to 20 on 32 processors was achieved by using this approach[13].

2.2.4 Scheduling Algorithm for Discrete Event Logic Simulation

As mentioned in the previous section, the future events have to be managed according to their time stamp. One way of scheduling these events in software is to use a list of lists data structure called an *event wheel*[7]. In this data structure, a fixed number of

time slots are assigned in a circular structure to store the list of events with the same time stamp, as shown in Figure 4. After the new events are generated, they are inserted into the proper time slot in the event wheel. The size of the event wheel is typically 64 and a special mechanism is used to monitor the overflow of the scheduler[6].

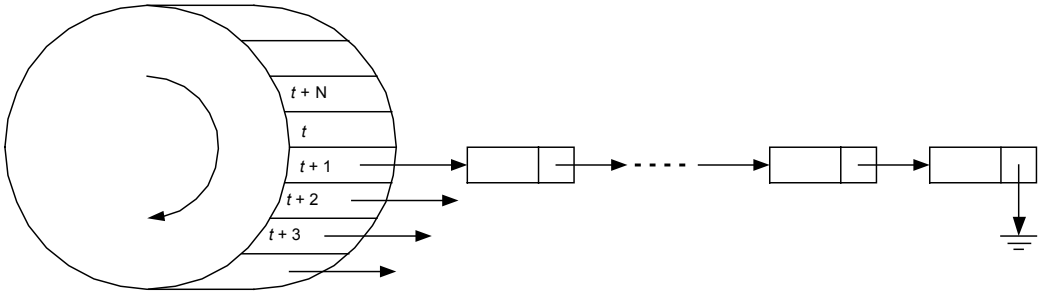


Figure 4 Event Wheel for Event Scheduling

As was show in Table 1 in Section 1.1 , the clock frequency of modern digital systems is already reaching over 1 GHz (sub-nano-second). If the timing grain ranges from pico seconds for high-speed designs to micro seconds for slow-speed designs, the event wheel described above will not be able to handle the situation. If the size of the event wheel has to be increased, then it becomes inefficient because most of the event wheel time slots will be empty.

2.2.5 Hardware Accelerators

Several researchers have investigated the use of dedicated hardware accelerators for logic simulation. Hardware accelerators can be classified into two categories. One is when the actual simulation is performed by custom hardware, and the other is functional emulation. A hardware simulator runs the simulation algorithm on a dedicated hardware,

and provides fast and accurate simulation results. Emulation only replicates a circuit's gate-level functionality and does not provide any mechanism for timing-based simulation of individual gates. The focus of this thesis is on hardware simulation accelerator and not on hardware emulator.

Recently, commercial vendors such as Quickturn and IKOS introduced the hardware logic emulator. A hardware logic emulator usually utilizes an array of Programmable Logic Devices (PLDs), especially Field Programmable Gate Arrays (FPGAs) as a platform, and programs the entire net-list into the array of PLDs. The Quickturn RPM emulation system[21] and IKOS Virtual Logic Emulator[22] both use a large number of FPGAs on a printed circuit board. In the Quickturn RPM board, each FPGA is connected to all its nearest neighbors in a regular array of signal routing channels. Several such boards are connected together in a system.

In general, emulators are more powerful than simulation engines in terms of speed since the logic elements inside of the PLD literally execute the logic function given by the input net-list. But as the name implies, hardware emulators can only emulate, not simulate. They lack the functionality of simulating the circuit's characteristics correctly given by the designer's intention and/or the target technology. In other words, the hardware emulators can only be used to perform the circuit's functional verification (*i.e.* logical correctness). This is a natural phenomenon because the design is mapped into the FPGA and actually run on the FPGA system, the actual circuit behavior on the target technology cannot be modeled. Therefore losing all the delay-timing information of the design.

IBM has done the most significant work on a compiler driven logic simulator. IBM has three generations of logic simulation parallel machines, all three of which are using the parallel compiled approach[6]. They are the Logic Simulation Machine (LSM), Yorktown Simulation Engine (YSE) and Engineering Verification Engine (EVE)[14, 15, 16]. All machines use same basic architecture, consisting of 64 to 256 processors connected by cross-bar switch for inter-processor communication.

LSM is IBM's first generation of custom designed simulation machine. It can handle 5 inputs with 3 logic signal levels and has a 63K gate capacity. YSE is the second generation of IBM's effort. It can handle 4 different signal levels (0, 1, undefined, high-impedance) and up to 4 inputs, with a 64K gate capacity. YSE is distinguished from its predecessor by its simulation mode, general-purpose function unit, a more powerful switch communication mechanism, and an alternate host attachment. YSE hardware consists of identical logic processors, each running pre-partitioned piece of the net-list. Each logic processor can accomplish a complete function evaluation in every 80 nano-second period (12.5 million gates per second)[17]. EVE is the final enhancement of YSE, it uses more than 200 processors. EVE handles 4 signal levels and 4 inputs, with 2M gate capacity with peak performance of 2.2 billion gates per second[6]. All three of IBM's simulation engines can only handle zero- or unit-delay model, which is only suitable for verification of logical correctness.

Another commercial accelerator for logic simulation is the Logic Evaluator LE-series offered by ZyCAD Corporation[18]. It uses a synchronous approach and a bus-based multiprocessor architecture with up to 16 processors, that implements scheduling

and evaluation in hardware. It exhibits a peak performance of 3.75 million gate evaluations per second on each processor, and 60 million gate evaluations per second on 16-processor model[6, 18].

The MARS hardware accelerator exploits function parallelism by partitioning the simulation procedure into pipelined stages[20]. The MARS partitions the logic simulation task through functional decomposition, such as signal update phase and gate evaluation phase. Both phases are further divided into 15 sub-task blocks, such as input and output signal management unit, fan-out management unit, signal scheduler, and housekeeper unit, etc. They employ exhaustive truth table as their gate evaluation primitives (up to 256 primitives with 4 inputs maximum). MARS is designed and built as an add-on board to the workstation. It can process 650 thousand gate evaluations per second at 10 MHz.

A commercial vendor, IKOS, builds the hardware logic simulation engine named "NSIM", which is currently the top of the line in the market. They claim that they can provide the simulation performance approximately 100 times faster than that of software simulation[19]. IKOS NSIM is a true full-timing simulator. But it requires that users to use its own primitives, and forces the designer to model their design in terms of IKOS primitives. This is a big limiting factor of IKOS. When a library cell vendor creates a new type of cells, the designer has to find a way to model this new cell using IKOS primitives. It also adds more loads to the simulation engine because each library cell is modeled using multiple IKOS primitives and those primitives have to be evaluated by the simulation engine.

2.3 Performance Analysis of the ISCAS'85 Benchmark Circuits

In order to obtain the performance bottleneck of software, a simple C program for logic simulation was made and tested on the benchmark circuits. ISCAS'85 benchmark circuits[23] were initially designed for fault simulation, but have been widely used by the logic simulation community. This is because there are no benchmarks specifically made for logic simulation. The size of this benchmark set is relatively small, and various researchers have noted the need for the standardized logic simulation benchmark circuits in various sizes. Unfortunately, the new benchmark circuit is not available yet.

Table 2 ISCAS'85 Benchmark Circuits[23]

	Function	Total Gates	Input Lines	Output Lines
C7552	ALU and Control	3,512	207	108
C6288	16-bit Multiplier	2,416	32	32
C5315	ALU and Selector	2,307	178	123
C3540	ALU and Control	1,669	50	22
C2670	ALU and Control	1,193	233	140
C1908	ECAT	880	33	25
C1355	ECAT	546	41	32
C880	ALU and Control	383	60	26

```
FOR each elements with time stamp t
   WHILE (elements left for evaluation with t) DO
      EVALUATE element
      IF (change on output) then
         UPDATE input & output values in memory
         SCHEDULE connected elements
      ELSE
         UPDATE input values in memory
      END IF
   END WHILE
   Advance time t
END FOR
```

Figure 5 Algorithm for Discrete Event Logic Simulation

The algorithm shown in Figure 5 can be divided into 3 phases. They are *evaluate, update* and *schedule*. The *evaluation* phase can be carried out by a simple table lookup of each Boolean primitive. The lookup table normally contains predefined sets of input/output signals. The *update* phase handles the output value change. After the evaluation, if the output signal changes due to the input signal change, the output value stored in the memory has to be modified accordingly. The *schedule* phase deals with the execution ordering of the events. Since the algorithm deals with a non-unit-delay model of simulation, the newly generated events have different time stamps, depending on the type of the gate. These new events should be placed in the execution schedule according to its time stamp. Otherwise, the simulation will violate the causality constraint and produce incorrect simulation results.

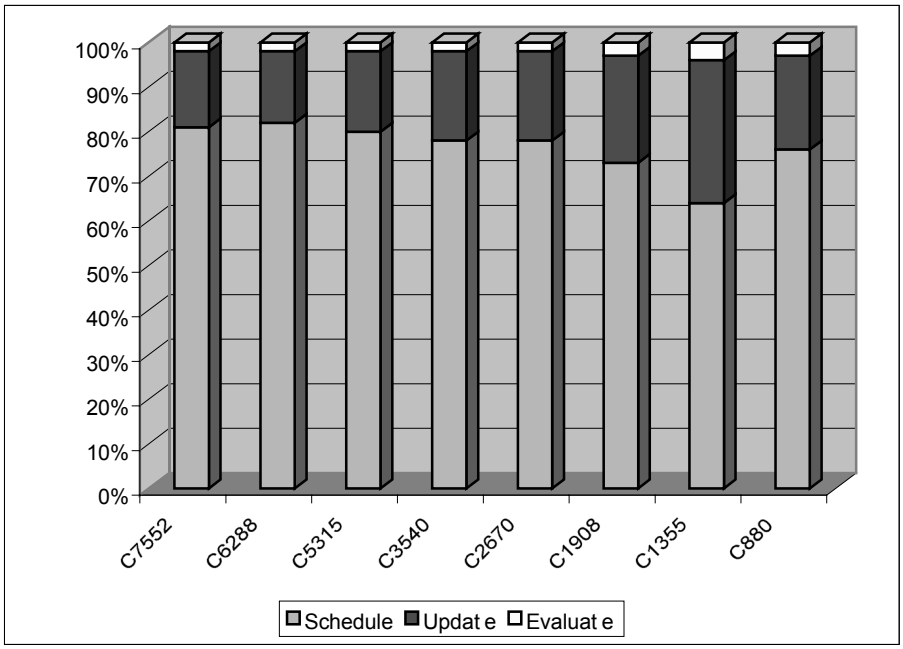


Figure 6 Run Time Profile of Various Benchmark Circuits (ISCAS'85)[23]

Figure 6 and Table 3 show the run time profile of the ISCAS'85 benchmark circuits. To extract the run time profile of software based logic simulation performance, we have implemented a simple C program with a generic synchronous algorithm and measured the CPU cycle of each subtasks. We have found that *evaluate* phase only spent 2% to 4% of the total run time, *update* used 16% to 32% and *schedule* phase, especially execution schedule management task that runs a "quick-sort" routine, used up most of run time (64% to 82%).

Table 3 Run Time Profile of Various Benchmark Circuits (ISCAS'85)[23]

Circuit	Schedule	Update	Evaluate
C7552	81%	17%	2%
C6288	82%	16%	2%
C5315	80%	18%	2%
C3540	78%	20%	2%
C2670	78%	20%	2%
C1908	73%	24%	3%
C1355	64%	32%	4%
C880	76%	21%	3%

To ensure the temporal correctness of the simulation, events, that are stored in scheduler, have to be ordered in time according to the time stamp. Whenever the new events are generated due to the *evaluate* phase, the scheduler sorts the events according to the time stamp. The schedule sorting involves major memory movement. Like many other application, memory bandwidth is the major bottleneck of the logic simulation algorithm. To handle finer timing resolution (discussed in Section 2.2.4), the event wheel algorithm was discarded, and sorting directly on the schedule was applied.

2.3.1 Analysis of Peak Software Performance

Not only logic simulation, but most Electronic Design Automation (EDA) problems are extremely memory intensive tasks. EDA problems usually do not benefit from cache memory due to the enormous memory space requirement and random memory access behavior. For example, most of the engineering problems generally work on numbers that are usually represented as an array/matrix of numbers. When the problem or application (*e.g.* MATLAB) requires a large memory space, they usually

exhibit temporal and spatial locality fairly well. In such cases, the speed and the amount of the cache memory will greatly improve the speed of computation.

In the case of EDA problems, the software performs operations on a group of data primitives that represents a circuit element. The circuit elements are represented as a record within a data structure, and the record usually contains multiple numbers and characters grouped as one record per circuit element. The record also contains some number of pointers to store the connectivity information of each circuit element (fan-in and fan-out). The size of a record is usually much larger than the system memory bus width, and EDA algorithms are often forced to perform a multiple memory access to retrieve the information of a single circuit element.

Since every circuit design is unique in its contents, the circuit's connection information varies from design to design. Therefore, logic simulation exhibits random memory access patterns, especially for highly complex circuit designs. Figure 7 shows the data structure for a logic gate and Figure 8 shows the data structure for the event queue of the logic simulation software. One circuit element takes up six 32-bit integers for a single record. To read the information about one circuit element, the 32-bit processor has to initiate six memory references.

```
struct  ram_struct {
   unsigned int function_type:5;
   unsigned int num_fanin:2;
   unsigned int input_val1:4;
   unsigned int input_val2:4;
   unsigned int input_val3:4;
   unsigned int input_val4:4;
   unsigned int current_output:4;
   unsigned int next_output:4; // 31 bits → 32-bit integer

   unsigned int output_change_count:8;
   unsigned int delay:20;
   unsigned int num_fanout:4; //32 bits → 32-bit integer

   unsigned int dest1:24;
   unsigned int dpid1:2; // 26 bits → 32-bit integer

   unsigned int dest2:24;
   unsigned int dpid2:2; // 26 bits → 32-bit integer

   unsigned int dest3:24;
   unsigned int dpid3:2; // 26 bits → 32-bit integer

   unsigned int dest4:24;
   unsigned int dpid4:2; // 26 bits → 32-bit integer
};
```

Figure 7 Data Structure Used for Circuit Elements in Software Simulation

```
struct _q_struct {
   unsigned time_stamp;
   unsigned gate_id:24;
   unsigned pin_id:2;
   unsigned val:4;
};
```

Figure 8 Data Structure for Event Queue

After the net-list for a design has been parsed into the memory, it is more likely that the next pointer will point to the non-adjacent memory location (possibly to a memory location very far away). In such cases, cache memory will exhibit more *cache misses* than *cache hits*. The operating system then has to spend more cycles for cache

management. Since the sequential software algorithm is run on a generic workstation, the performance of the algorithm is inevitably bound to the workstation's internal architecture. Most workstations are based on a fixed-width memory bus architecture, which severely limits the performance of the memory access, especially for the applications like logic simulation. Also, the processors in the work-station contain extra circuits such as floating point ALUs and pipelines that are not needed for logic simulation. It is obvious that in order to get a better performance on logic simulation task, we need to get a better memory performance than that of a workstation.

Table 4 Read-Modify-Write Memory Performance of Pentium-III 450MHz

Amount of Memory in Bytes	Time in nano second
240	761
480	762
2,496	765
5,016	763
25,152	780
50,328	793
251,640	807
1,258,272	924
2,516,568	955
12,582,912	1,027
25,165,824	1,049
50,331,648	1,075
75,497,472	1,097
100,663,296	1,107
125,829,120	1,255,746

To simulate the memory access behavior of logic simulation, a simple C program was written to extract the memory performance. From the viewpoint of memory access, the logic simulation task is equivalent to the series of memory *read-modify-writeback* operations. To imitate this *read-modify-writeback* behavior, we used the identical data

structure that is used in logic simulation software, and created a block of memory with this data structure. Then using a random number generator, we accessed a memory location for read and modified the contents and then wrote back the data into the same memory location. From Table 4, we found that the memory access speed for this task takes about 1000 nano-second on average.

2.4 Limitations of the Von Neuman Architecture

As discussed in this chapter, the software solution running even on the highest performance workstations will not be able to provide the performance needed by large circuit logic simulation. This is because the modern workstations are based on the "Von Neuman" architecture. The characteristics of Von Neuman architecture are:

- Load/Store
- Fixed Width Instruction/Data
- General Purpose
- Single Memory (Virtual memory concept)
- Cache Hierarchy

Having a single narrow memory architecture forces the simulation task to perform multiple memory accesses to obtain related information for a single gate. Cache memory does not usually help for large circuit simulations because cache misses causes overhead. Therefore, a traditional Von Neuman architecture cannot provide the performance

required for logic simulation task. A new custom architecture will be described in the next chapter.

3.0 HARDWARE SIMULATION ENGINE ARCHITECTURE

As was shown in previous chapter, the bottleneck of software-based simulation performance is caused by the poor performance of memory. Logic simulation can be divided into three tasks: Evaluate, Update and Schedule. Evalutate and Update are normally carried out as a single task and modeled as *read-modify-writeback* because each event affects a particular gate that must be *read* from memory, the gate's inputs and possibly its outputs are *modified*, and the gate's information must be *written back* to memory. As shown earlier, a *read-modify-writeback* operation in software requires between 1200 and 1400ns. The Schedule task determines the order in which events occur. This is essentially a sorting problem and again is memory intensive.

By dividing the entire logic simulation task into Evaluation, Update and Schedule subtasks, a performance profile of a benchmark circuit with such division can be graphed, as shown in Figure 9. Based on the results of the performance analysis of a software simulation algorithm, we designed a custom architecture to perform an event driven logic simulation algorithm in hardware. The goal is to overcome the performance bottleneck found in software simulation by implementing the simulation in hardware.

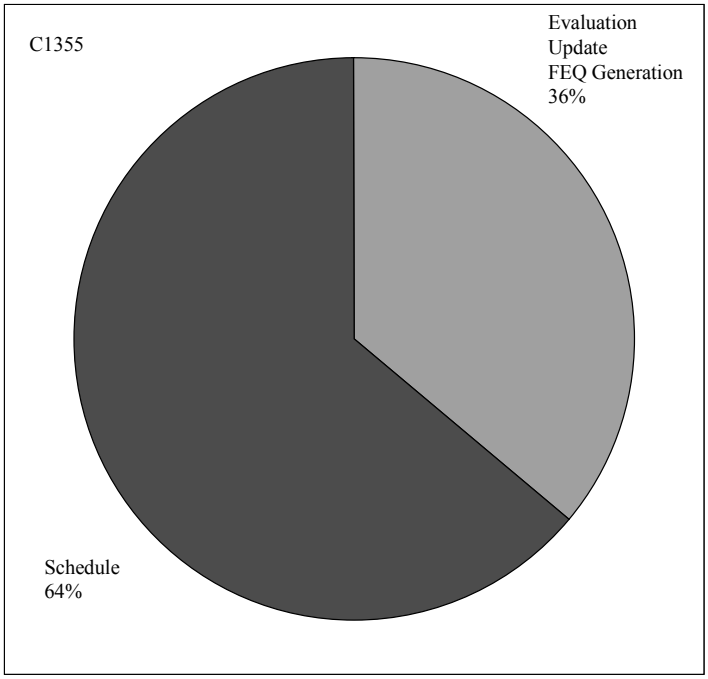


Figure 9 Run-Time Profile of Benchmark Circuit C1355

3.1 Statement of the Problem

The problem that the hardware logic simulation architecture should solve can be summarized as following three categories:

1. Accuracy: A Full-Timing, Gate-Level Simulation at the pico-second accuracy is the most important feature for the modern digital logic simulators. Without the full-timing information, timing problems cannot be seen.
2. Capacity: Due to modern CMOS technology, the size of the VLSI chips reaches to millions of gates. Therefore, the capacity of the simulation hardware should be large enough to process these large designs. This work targets design sizes from

100,000 to millions of gates. Common in EDA area, a large size design is computationally complex due to its large memory space requirement.

3. Speed: The software simulators running on a high speed workstation consumes weeks to months of simulation run time when processing a large design. Hardware logic simulation needs to be an order of magnitude faster than the software to shorten the design cycle time and speed time-to-market.

3.2 Overview

Figure 10 illustrates the task flow of the hardware accelerated logic simulation system. The logic simulation task on a hardware accelerated simulation system is carried out in four phases. First, pre-processing software is run on a workstation to construct the data structure for the simulation hardware using following inputs: circuit's gate level description, circuit's gate-level timing information and the cell function of the selected technology. Second, the constructed data structure is downloaded into the simulation hardware. Third, the actual simulation is carried out in the hardware. Fourth, the results from the hardware are uploaded into the workstation for the user to examine the simulation results. The focus of this thesis is to design the logic simulation hardware to provide the simulation results faster than is possible on a Von Neuman workstation.

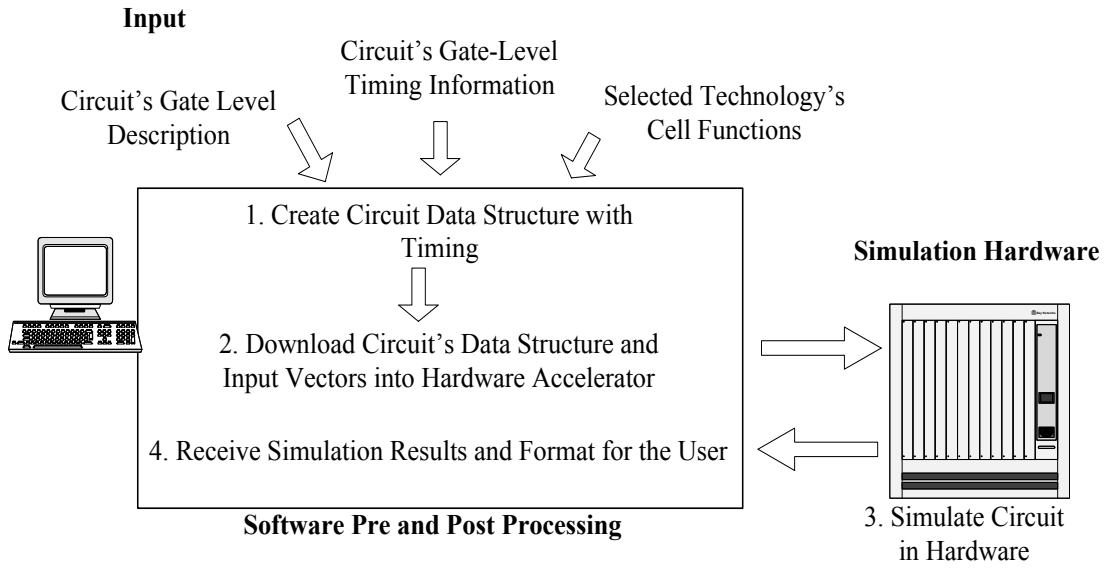


Figure 10 Hardware Accelerated Simulation

Our simulation hardware is divided into three task blocks: the Logic Engine, Future Event Generator, and the Scheduler, as shown in Figure 11. The Logic Engine performs logic evaluation due to an input-change event received from the Scheduler and computes the output using the information stored in the Net-list and Configuration Memory. The Logic Engine computes the output of the gate without considering the delay of the gate. If the output value is changed, the Future Event Generator performs the delay computation using the data in delay memory and generates new event called a *future event*. A future event is an event that is to occur some time in the future. The future event is passed to the scheduler through the future event queue. This new event is passed back to the Logic Engine at the proper future simulation time. If the output does not change, no future events are generated and logic computation terminates. When the logic computation completes, the Logic Engine writes the new input values and possible new output values back into the net-list and configuration memory. The Scheduler

manages new incoming events that are generated as a result of logic evaluation. All future events are stored in the event memory. The Scheduler determines when to increment the simulation time and examines its list of future events. A *pending event* is a former *future event* whose execution time is equal to the simulation time and should be executed. The Scheduler retrieves pending events from the Event Memory and forwards them to the Logic Engine for evaluation. This process is continues until the pre-determined simulation time.

Each of the main function blocks in our hardware (*i.e.*, the Logic Engine, Future Event Generator and the Scheduler) can be viewed as a processor with one instruction, which only performs logic simulation task. This removes extra overhead caused by fetch-decode-execute cycles of a Von Neuman architecture, since our architecture does not rely on any general purpose instructions. The remainder of this chapter discusses each block in more detail and outlines this work.

Our design also distinguishes itself from the others in that new features such as power consumption measure are built into the simulation hardware. This architecture will not only provide the logical verification and performance, but will be able to guide the place-and-route process to evenly distribute the thermal "hot spots." Our design also targets a wider range of delay timing resolution (finer timing grain) compared to existing simulators so that it can be used in a co-simulation environment, for example, logic gates mixed with software running on a built in processor such as System on Chip (SOC) environment. For such applications, the timing grain spans from a few pico-seconds to several micro-seconds, and this makes the traditional event wheel approach inefficient.

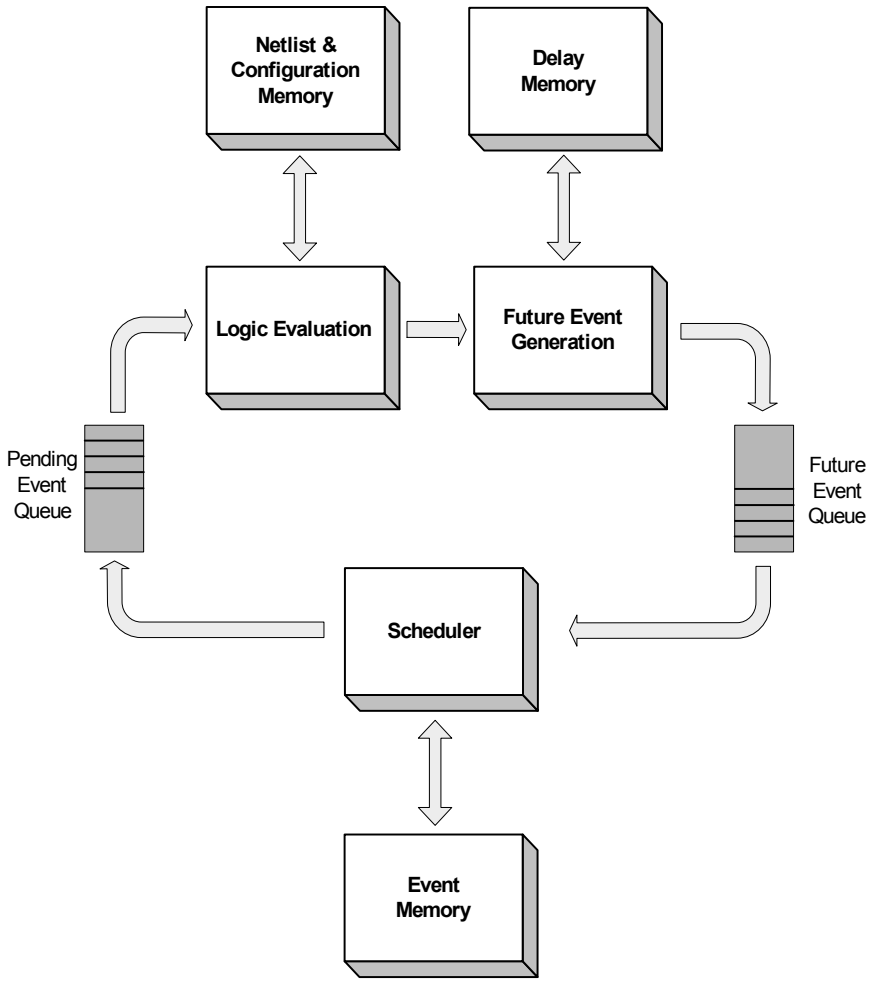


Figure 11 Overview of the Architecture

3.3 Logic Engine

In Chapter 4, we demonstrate the computation of various logic primitives with minimal hardware resources. We then introduce a new concept and primitives to perform this logic computation more efficiently. One of the issues is that modern digital logic simulation requires multi-level signal strengths such as logic-Low (‘0’), logic-High (‘1’), High-Impedance (‘Z’), and Unknown (‘X’). Having these multi-level signals, not just

Boolean 1's and 0's, makes the logic simulation task more complex. The software simulation uses a lookup table to compute these multi-level signal strengths through multiple table lookup activities. Using a large lookup table, hardware requires only one lookup to achieve the result. To gain this performance, the amount of hardware resource grows exponentially with the gate input size. We illustrate the concept of behavioral modeling by introducing *Any* and *All* primitives to reduce the size of the lookup table and perform the logic simulation task faster and more efficiently than a single lookup table. We then introduce a "Universal Gate" to perform four-level logic evaluation for numerous Boolean logic gates. In addition to the Boolean logic gates, we present a group of frequently used macro cells as single primitives such as a Multiplexer, Full Adder and Flip-Flop. A detailed description of the Logic Engine can be found in Chapter 4.

3.3.1 Mapping into Hardware Memory

The pre-processing software generates the memory map according to the input circuit design. Figure 12 shows an example circuit. The pre-processing software reads the circuit description written and cross-links the gates according to the circuit's fan-out information. It then generates the net-list and configuration memory map as shown in Table 5. All of the input and output values are initialized as Unknown ('X') state. The delay memory mapping will be discussed in the next section.

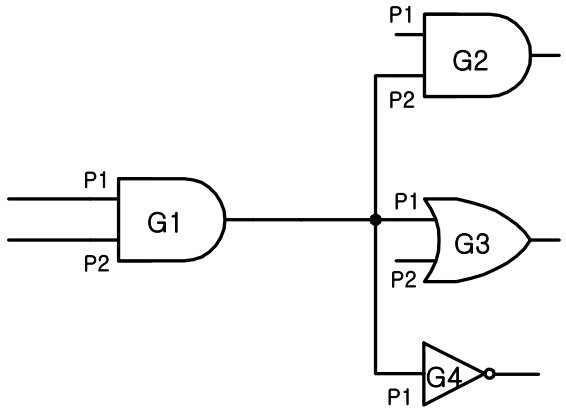


Figure 12 Mapping Circuit Net-list into Logic Engine Memory

The Net-list and Configuration Memory stores all information about a gate that is necessary for it to be evaluated and stores all information about the state of the gate. Each location in memory corresponds to a particular gate, and thus, the gate's identification number is a physical memory address. This drastically increases performance by removing virtual memory.

Table 5 Net-list and Configuration Memory Map

Address	Delay Base Address	Power Count	Input Values	Output Values	Destination Gate ID	Destination Pin ID	Gate Type
4	7	0	X	X	-	-	INVERT
3	5	0	X X	X	-	-	OR
2	3	0	X X	X	-	-	AND
1	1	0	X X	X	2, 3, 4	2, 1, 1	AND

The first column in Table 5 stores the memory address which holds the delay values of each gate. The second column stores the power count and the third and fourth column store the input and output states of the gate, respectively. These values are initialized as Unknown (‘X’). The Fifth and the sixth columns store the fan-out

information for each gate and the last column stores the gate type information for configuration.

3.3.2 Test Coverage and Stuck-at Fault Simulation

In addition to the accurate simulation of a logic circuit design, the architecture can be applied to test coverage, false path detection and stuck-at fault simulation. To do this, an output change count (called *power count*) mechanism is built into the architecture design. A *power count* is associated with each gate and is incremented each time the output transitions. Test engineers can simulate the circuit under test with a partial set of input vectors, and can determine whether the input vector has exercised the data paths by examining each gate's output change count. For example, if the gate's output change count stays unchanged after a series of input vector applications, the test engineer can conclude that either the design contains a false path, or the input vector set applied is not enough to exercise all of the data paths. This example is illustrated in Figure 13 in Gate G6.

Another application of the logic engine design is stuck-at fault detection. Stuck-at fault simulation can be performed by replacing the gate model with a faulty gate model in the evaluation block, which is described in the previous section. Test engineers can run the simulation and compare the results for the case with a faulty gate model to the normal gate model. This is shown in Figure 13 with Gate G8.

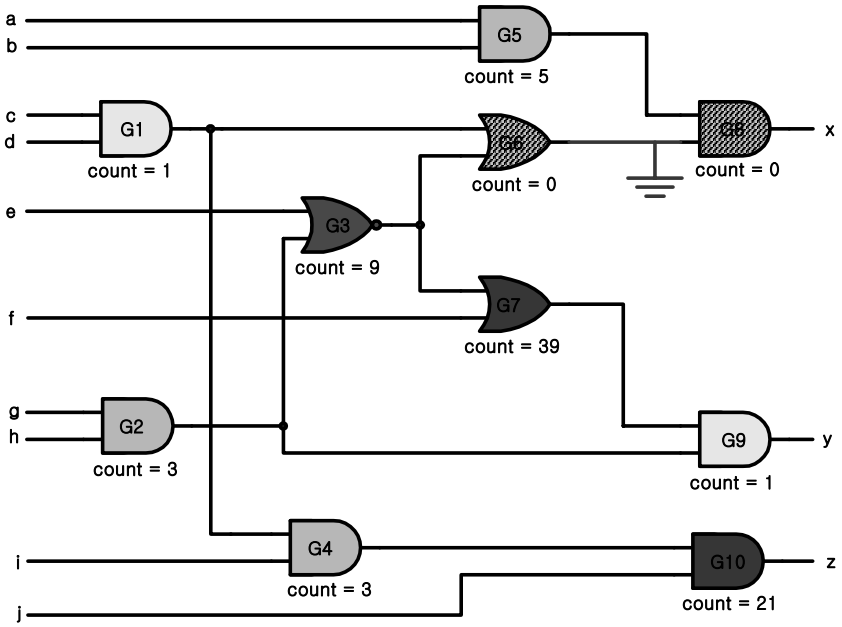


Figure 13 Use of Output Change Count

3.3.3 Power Consumption Estimation

The output change count can also be used to estimate power consumption. As was discussed in Section 1.6, dynamic power dissipation is a function of output change frequency. By counting the output state changes of the logic gates, we can extract the estimated power consumption for each gate. As gates consuming more power naturally generate more heat, this power count can also be used as a measure for thermal topology. If the design being simulated is post-synthesis, we can determine which part of the design will consume more power and therefore run hotter. If the design is pre-layout, we can predict which area will be hot, which can then be utilized in the place and route process. The place and route tool can take this information and distribute the thermal "hot spots" more evenly. As an example, the gates G7 and G10 shown in Figure 13 have the highest

output change counts among other gates in the circuit. Therefore, designers can examine these values and conclude that gates G7 and G10 will consume the most power and these two gates are not recommended to be placed close together for the thermal distribution.

3.4 Future Event Generator

After the logic evaluation is performed and the new output is acquired, the Logic Engine compares the new output value and the current output value. If the output value has changed, then future events will be generated. To generate these future events, the proper delay value associated with the cell currently being simulated has to be added to the simulation time so that the new event can be scheduled, and passed back to the logic engine for evaluation in the appropriate future simulation time.

Delays of logic cells can be classified as *intrinsic* and *extrinsic* delays. An *Intrinsic* delay refers to the cell's own delay when the cell does not drive any load. *Extrinsic* delay refers to the external capacitive load caused by the interconnecting wire and other cell's input gate capacitance. Furthermore, depending on the cell type, there are 3 different delay types: *fixed delay*, *path dependent delay*, and *state dependent delay*. In Chapter 5, we illustrate the differences between these delay models. These delay types require an increasing and variability amount of storage for the delay values. Therefore, storing and addressing these delay value becomes a problem. For example, in the fixed delay model, the only variable is whether the output changed to high or low. The path dependent delay model adds input-to-output path information to the equation. The state

dependent delay model adds the state of the input value to the path dependent model. We will address gates with such delay characteristics, and explore different delay memory architectures and mechanisms to handle these delay models, and discuss the pros and cons of each.

The pre-processing software also reads the delay information provided by SDF file and creates the delay memory map for the circuit. Table 6 shows the generated delay memory map for this example.

Table 6 Delay Memory Map

Delay Address	Delay
1	Rise Time = 8
2	Fall Time = 6
3	Rise Time = 7
4	Fall Time = 5
5	Rise Time = 9
6	Fall Time = 8
7	Rise Time = 4
8	Fall Time = 3

3.5 Scheduler

As was shown in Chapter 2, scheduling events consumes a major portion of logic simulation. In Chapter 6, we discuss the performance of various sorting algorithms and identify the characteristics of the schedule task. We also discuss the problems related with co-simulation and discuss why the current event-wheel based scheduling will not be applicable to a scheduling problem with a wide variety of timing resolution. We will illustrate how a parallel sub-memory scanning mechanism can be used to avoid the high cost for a linear search and yet be able to provide efficient memory usage and improved

speed for scheduling. The design space will be explored with event memory sizes, and their performance will be computed and discussed. The design space to be explored includes the following:

- A Single memory with one linear search algorithm.
- Multiple memories and linear search sub-blocks in parallel. (2-level)
- Multiple memories and a linear search with combinational global search.

A comparison of these three approaches and the software approach will be made.

3.6 Experimental Results and Scalability

To demonstrate that our design is feasible, a proof-of-concept implementation of our architecture has been created. Its performance was measured and is reported in detail in Chapter 7. Through out this thesis, the logic element of FPGA has been used for unit of measure for size comparison.

We also discuss net-list pre-processing software and its data structure. The pre-processing software plays a crucial role in generating the correct memory image to be loaded into our hardware design. We then demonstrate our design with a simple test circuit, and discuss the performance related issues.

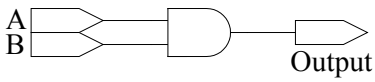
Scalability issues will also be discussed with a 100,000 gate capacity design, and show how each field of the data structure has to be scaled. Our prototype also demonstrates extra features we have implemented, such as power count (as discussed in Section 3.2 and 3.3). Finally, the performance and feature comparison between our

design and existing hardware and software simulators will be presented and discussed. In order to make a fair comparison between our architecture and others, we have quantified our experimental results in terms of FPGA Logic Elements. These Logic Elements (LE's) are created using a 4-input lookup table, a flip-flop, and a number of other AND and OR gates that are used to interconnect the LE with other LE's. As an approximation, a single FPGA LE can be implemented in an ASIC using 1-10 standard cell gates.

4.0 LOGIC EVALUATION ARCHITECTURE

Logic evaluation of a gate can be performed in two ways. One is to use Boolean primitives to compute the logic value. The other is to rely on a table lookup that lists every possible input and output combination. The problem with using Boolean logic primitives is that it only works with two-level signals such as Logic Low ('0') and Logic High ('1'). When the input contains four or more level signals such as Hi-Impedance ('Z') or Unknown ('X'), the input signal has to be encoded into 2 or more bits, which makes it difficult to compute with generic Boolean logic primitives.

Evaluating a two-input AND gate is seemingly simple in its Boolean equation form, as shown in Figure 14, but using this Boolean equation is not enough to extract its behavior, since it only deals with logic '0' and logic '1' for computation. Simulators for modern digital systems must handle multi-level signals such as 'Z' and 'X' in addition to the logic '0' and logic '1' in order to provide accurate simulation results at pico-second precision.



Output = A * B

Figure 14 Two-Input AND Gate

IEEE standard logic 1164[24] defines 9 different signal strengths and describes how the Boolean logic should be evaluated for various input signal strengths for VHDL. IEEE standard 1364[31] for Verilog defines 4 signal strengths as 0, 1, Z (High Impedance)

and X (Unknown). This research is focused on the Verilog standard, and shows how logic gates can be efficiently simulated with 4 signal strengths.

A two-bit encoding scheme is required to express four signal strengths. While this two-bit representation is good for saving the storage space, it is difficult to manipulate within the hardware. Instead, a "one-hot encoded" (4-bit) notation is used to represent these four level signal strengths. Table 7 shows the encoded signals that our design will use to represent different signal strengths. Logic Low ('0') will be represented as "0001" (decimal 1), Logic High ('1') will be represented as "0010" (decimal 2), Hi-Impedance ('Z') will be noted as "0100" (decimal 4), and Unknown ('X') will be represented as "1000" (decimal 8).

Table 7 One Hot Encoded Signals

Signal	Meaning	One Hot Encoded	Decimal Number
0 or L	Logic Low	"0001"	1
1 or H	Logic High	"0010"	2
Z	High Impedance	"0100"	4
X	Unknown	"1000"	8

One mechanism that can handle multi-level signal strength ('0', '1', 'Z', and 'X') is a standard lookup table. The problem with a lookup table is that the size of the table grows exponentially with number of inputs, and quickly becomes unmanageable. As shown in Table 8, a simple 2-input AND gate can have up to 16 entries in lookup table. In general, a gate with N input signals will have up to 4^N entries in its lookup table representation. Table 9 and Figure 15 show this exponential-size problem. Therefore, evaluating a logic gate using the lookup table often requires a large memory.

Table 8 Lookup Table for 2-Input AND Gate

A	B	Output
0	0	0
0	1	0
0	Z	0
0	X	0
1	0	0
1	1	1
1	Z	X
1	X	X
Z	0	0
Z	1	X
Z	Z	X
Z	X	X
X	0	0
X	1	X
X	Z	X
X	X	X

Table 9 Lookup Table Size Computation

Number of input	Truth Table entries
1	4
2	16
3	64
4	256
5	1,024
6	4,096
7	16,384
8	65,536
9	262,144
10	1,048,576

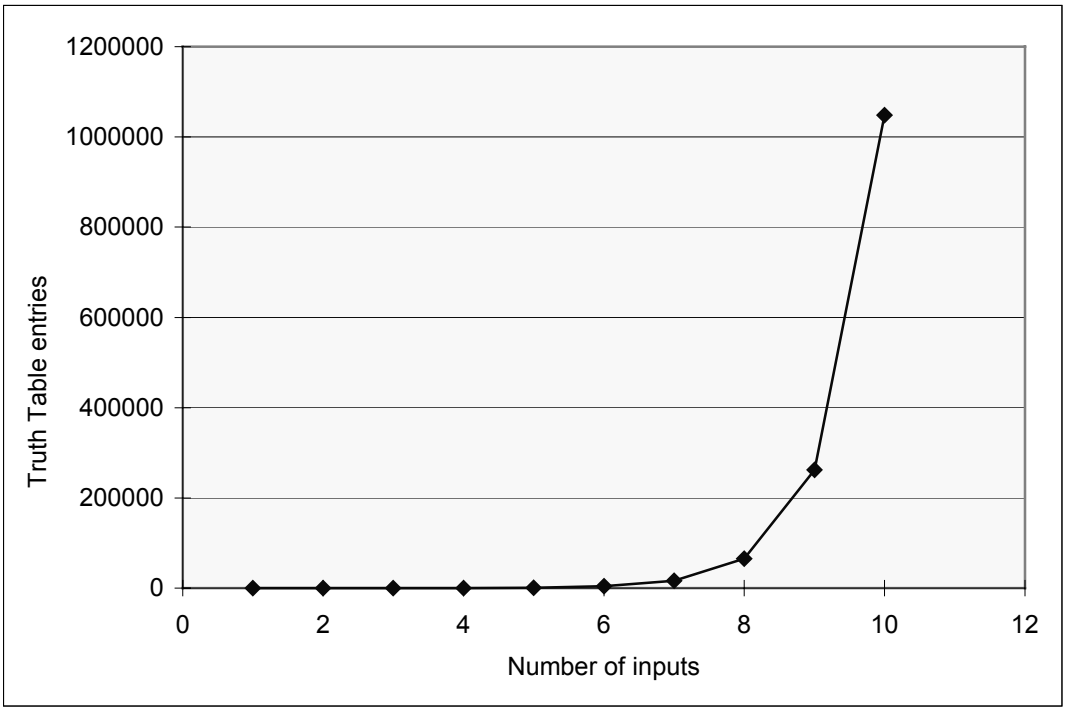


Figure 15 Lookup Table Size Growth

ASIC libraries tend to grow in its size and items so that it becomes difficult for these growing libraries to be simulated efficiently with a fixed logic simulation engine design. Therefore, flexibility becomes a major issue. To make simulation engines flexible and yet simple, logic gates are grouped based on their functionality and behavior. Table 10 lists the function group along with the number of gates for a particular vendor cell library. The library contains many gates with same functionality, but with different driving capabilities.

Table 10 Function Group and Number of Gates for Each Group

Name	Quantity
INVERTER	62
BUFFER	56
AND/NAND	42
OR/NOR	42
AND-OR/AND-OR-INVERT/OR-AND/OR-AND-INVERT	320
XOR/XNOR	18
MUX	34
FULL ADDER	16
FLIP FLOP	66

When we examine the behavior of the AND gate shown in Table 8, we can simplify this 16-entry table into the 3-entry table shown in Table 11 by creating two functions Any( ) and All( )[36, 37]. Furthermore, when we model the behavior, the table size remains unchanged regardless of the number of inputs. Regardless of the number of inputs, the behavior of the AND gate does not change.

Table 11 Behavioral Modeling of 2-Input AND Gate

Inputs	Output
Any 0	0
All 1	1
ELSE	X

The above discussion motivates the need for behavioral modeling of logic gates and the need for new hardware primitives. From Section 4.1 through Section 4.4 , we will discuss the behavioral modeling of the inverter/buffer, AND/OR, XOR, and AO/OA cells.

There are 3 function groups that have to be modeled individually because their input pins have special meaning and they have a unique behavior. For example, a clock input pin for the flip-flop can only be connected to the clock signals. And a select input

pin of a multiplexer (MUX) can only be connected to the proper signals. Contrary to the generic AND and OR gates, in which we can interchange the inputs without causing functional change, certain gates contain these special I/O pins that have to be connected to proper signals to guarantee the functionality. For example, if a clock input and a data input of a flip-flop are interchanged, then that flip-flop will not function as the designer intended.

Multiplexers are frequently used items in the digital system design. Although, they can be expressed as a collection of Boolean primitives, they require multiple layers of AND and OR gates to express a complex multiplexer (MUX). Therefore, its behavioral modeling is motivated and defined as a MUX primitive in Section 4.6 . A Full adder gate is also defined as its own primitive and discussed in Section 4.7 . Finally, a D Flip-Flop's behavior is modeled and defined as primitives in Section 4.8 . Scalability is discussed in Section 4.9 .

4.1 Inverter and Buffer Cells

Table 12 shows the lookup table of Inverter and Buffer cells. Table 13 illustrates the functional behavior of Inverter and Buffer cells based on the lookup table shown in Table 12. An inverter will invert the input value when the input is ether Logic-High ('1') or Logic-Low ('0'), but will generate Unknown ('X') as output when the inputs are High-Impedance ('Z') or Unknown ('X') according to the definition given in IEEE standard 1164[24]. Similarly, the Buffer cell will pass the input to its output when the input is

either Logic-High or Logic-Low, but will generate Unknown as its output value when the input value is either High-Impedance or Unknown.

Table 12 Standard Lookup Table for Inverter/Buffer Gates

A	Inverter	Buffer
0	1	0
1	0	1
Z	X	X
X	X	X

Table 13 Priority Lookup Table for Inverter/Buffer Gates

Input Pattern	Inverter	Buffer
0	1	0
1	0	1
ELSE	X	X

The inverter can be attached to other gate evaluation logic to form the negated gate design. For example, the AND gate evaluation engine design actually contains both normal output and negated output for NAND design. The OR gate evaluation design also has the inversion logic of the out to obtain NOR functionality. Notice that we are using "ELSE" clause in the table. The "ELSE" clause means that our lookup table contains a "priority". Using priority can introduce performance penalty for a large lookup table, but as was shown in Table 13, the size of our lookup table is only 3.

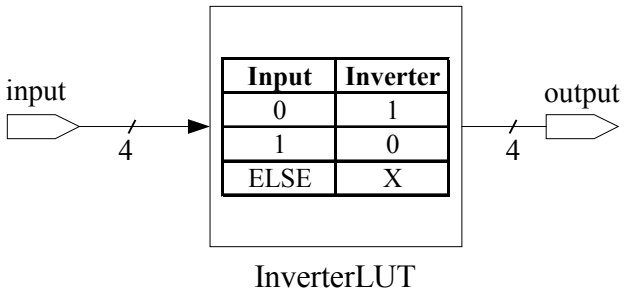


Figure 16 Inverter Design

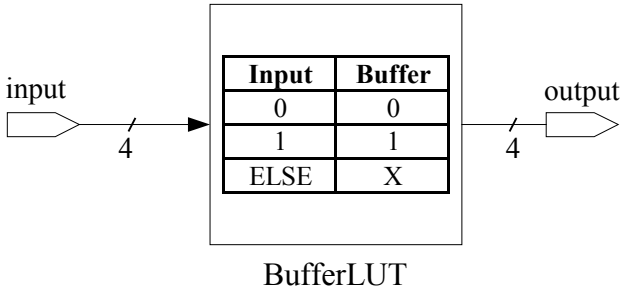


Figure 17 Buffer Design

Our design of the inverter and buffer can handle four level signal strengths whereas normal Boolean logic can handle only two. By using the "ELSE" to the lookup table, we were able to optimize the size of the lookup table by 25% (by grouping 'Z' and 'X' inputs, therefore 4 down to 3). Optimization results are a function of the number of inputs and therefore, larger cells will show better improvements.

4.2 AND/NAND and OR/NOR Cells

Table 14 describes the behavior of a 2-input AND gate and a 2-input NAND gate in lookup table form. From this table, we can see that the output of the 2-input AND gate becomes Logic-Low ('0') when **any** one of the input value is Logic-Low ('0'). Also the output of the 2-input AND gate will be Logic-High ('1') when **all** of the input values are Logic-High ('1'). If any one of the inputs becomes High-Impedance ('Z') or Unknown ('X'), then the output of the AND gate will generate the Unknown ('X') value.

Table 14 Lookup table for 2-Input AND/NAND Gates

A	B	AND	NAND
0	0	0	1
0	1	0	1
0	Z	0	1
0	X	0	1
1	0	0	1
1	1	1	0
1	Z	X	X
1	X	X	X
Z	0	0	1
Z	1	X	X
Z	Z	X	X
Z	X	X	X
X	0	0	1
X	1	X	X
X	Z	X	X
X	X	X	X

This behavior is modeled in Table 15. The size of lookup table has been reduced from 16 (= 4^2) down to 3. This size reduction can be generalized to the AND/NAND gates with any number of inputs.

Table 15 Priority Lookup table for AND/NAND Gates

Input Pattern	AND	NAND
Any('0')	0	1
All('1')	1	0
Else	X	X

Therefore, if we have the Any and All primitives as readily available functions, then AND/NAND gate evaluation with any number of inputs becomes extremely simple, as illustrated in Figure 18. This is not the case for the standard lookup table. As shown in Figure 15, the table size grows exponentially. The size of LUT in Figure 18 and Figure 19 do not change for any number of inputs. Scalability of Any( ) and All( ) is discussed in later section.

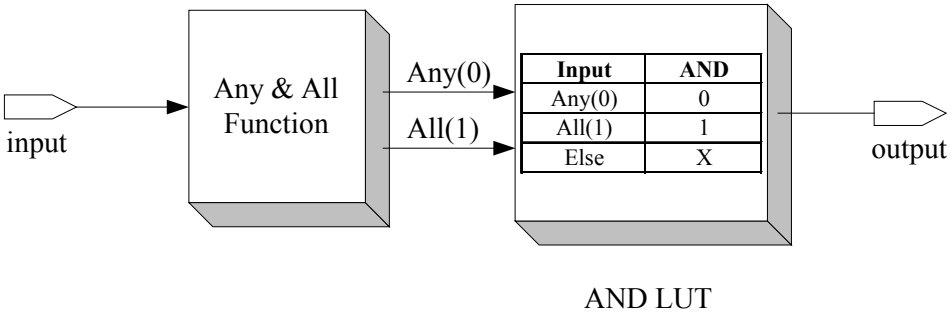


Figure 18 AND Gate Evaluation Design Using Any and All Primitives

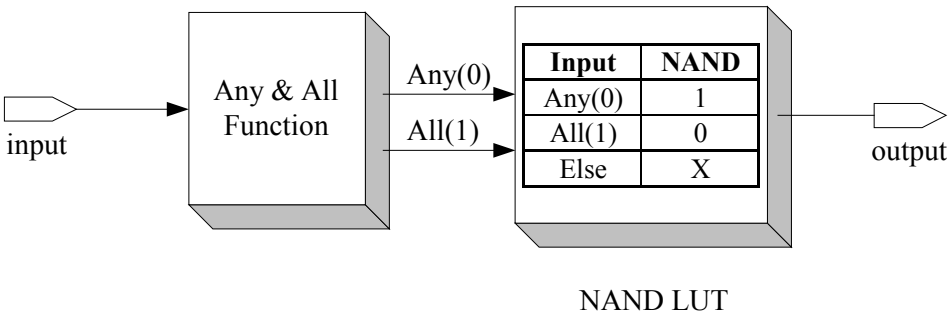


Figure 19 NAND Gate Evaluation Design Using Any and All Primitives

Again, notice that our lookup table contains a priority. Any('0') has higher priority than All('1'), and All('1') has higher priority than the "ELSE" part of the lookup table. To illustrate this point, assume that we have 4-input AND gate with the values shown in Table 16 (a). Input B will cause the Any('0') function to be TRUE, and it will also cause All('1') to be FALSE. When we evaluate this result, Any('0') has higher priority and is already TRUE, the output of the AND gate becomes '0'. But in the case shown in Table 16 (b), the Any('0') function will be FALSE because none of the inputs are '0'. And the input D will cause the All('1') function to be FALSE. Therefore, our priority lookup table will match the "ELSE" part of the table, and the result of the AND gate evaluation will be 'X'.

Table 16 Any/All Function for a 4-Input AND Gate

(a) Any('0')=True and
All('1')=False

Inputs	Value
A	1
B	0
C	1
D	X

(b) Any('0')=False and
All('1')=False

Inputs	Value
A	1
B	1
C	1
D	X

Table 17 Lookup Table for 2-Input OR/NOR Gates

A	B	OR	NOR
0	0	0	1
0	1	1	0
0	Z	X	X
0	X	X	X
1	0	1	0
1	1	1	0
1	Z	1	0
1	X	1	0
Z	0	X	X
Z	1	1	0
Z	Z	X	X
Z	X	X	X
X	0	X	X
X	1	1	0
X	Z	X	X
X	X	X	X

Table 17 shows the standard lookup table for 2-input OR/NOR gates. The output of 2-input OR gate will produce Logic-High (‘1’) when **any** one of the input value is Logic-High (‘1’). The output of the 2-input OR gate will be Logic-Low (‘0’) when **all** of the input values are Logic-Low (‘0’). If any one of the inputs becomes High-Impedance (‘Z’) or Unknown (‘X’), then the output of the OR gate will generate Unknown (‘X’) value.

Table 18 Priority Lookup Table for OR/NOR Gates

Input Pattern	OR	NOR
Any(‘1’)	1	0
All(‘0’)	0	1
ELSE	X	X

This behavior is modeled in Table 18. Again, we reduced the size of lookup table down to 3 with the Any( ) and All( ) function pair with any number of inputs. Figure 20 and Figure 21 shows the OR/NOR logic evaluation design.

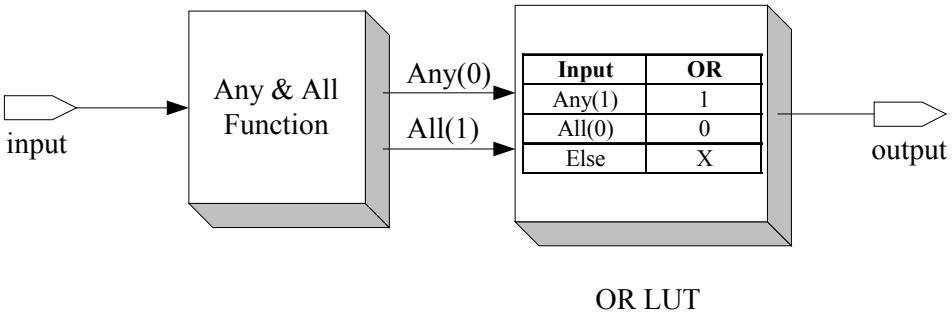
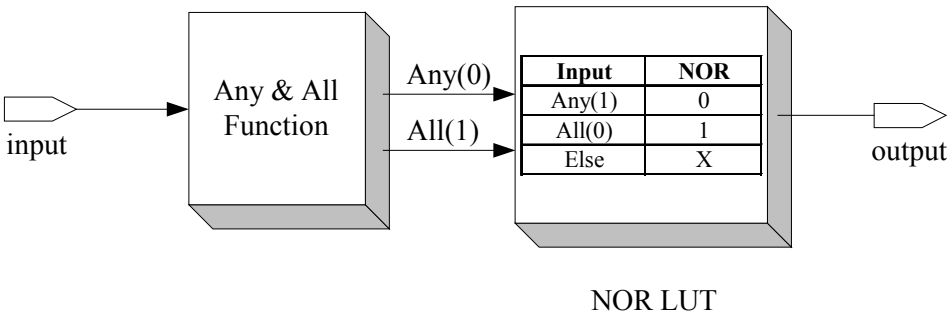


Figure 20 OR Gate Evaluation Design Using Any and All Primitives



NOR LUT

Figure 21 NOR Gate Evaluation Design Using Any and All Primitives

In summary, we have shown that with Any( ) and All( ) primitives, we can achieve the AND/NAND/OR/NOR gate evaluation design. This is a significant improvement in the size of lookup table because we were able to reduce the size down to a 3-entry priority lookup table for any number of inputs. As for standard lookup table approach, the size of the table grows exponentially as the number of input grows. Table 19 shows the size comparison for a various number of inputs. The "Number of Primitive Functions" column shows how many primitives are being using. Since we are using Any('0')/All('1') for the AND gate or Any('1')/All('0') for the OR gate, the value for the column is 2. The "Function Width" column shows how many primitive functions are needed to implement the design. Since our Any( ) and All( ) primitives work in pairs and should be applied on each input values, it is a function of total number of inputs. As a conclusion, we can see that as the number of inputs for AND/OR gates grows, our approach can reduce the size of table. It should be noted that the function width grows linearly with input size and will consume resources proportional to the number of inputs.

For an n input AND/NAND/OR/NOR operation, the width of Any/All function will be ($n \times$ size of Any/All function) because the Any/All function is applied to each

input signals. Section 4.5 will describe this in detail. The size of our priority lookup table is a constant with a value of 3.

Table 19 Lookup Table Size Comparison for AND/NAND/OR/NOR Gates

Number of Inputs for AND/NAND/OR/NOR	Lookup Table Size	Using Any/All			LUT Reduction Factor
		Number of Primitive Functions	Function Width	Priority LUT Size	
2	16	2	2	3	5
3	64	2	3	3	21
4	256	2	4	3	85
5	1,024	2	5	3	341
6	4,096	2	6	3	1,365
7	16,384	2	7	3	5,461
8	65,536	2	8	3	21,845
9	262,144	2	9	3	87,381
10	1,048,576	2	10	3	349,525
N	*4^N*	2	*N*	3	*(4^N)* / 3

4.3 XOR/XNOR Cells

Table 20 shows the lookup table for a 2-input XOR/XNOR gates. The output of the 2-input XOR gate will be Unknown ('X') when **any** of the input values is High-Impedance ('Z') or **any** of the input values is Unknown ('X'). Otherwise, the XOR gate will follow the normal Boolean logic function for XOR gate behavior. In other words, XOR will output Logic-High ('1') when the number of Logic-High ('1') inputs is ODD number and output Logic-Low ('0') when the input has an EVEN number of Logic-High ('1') values.

Table 20 Lookup Table for 2-Input XOR/XNOR Gates

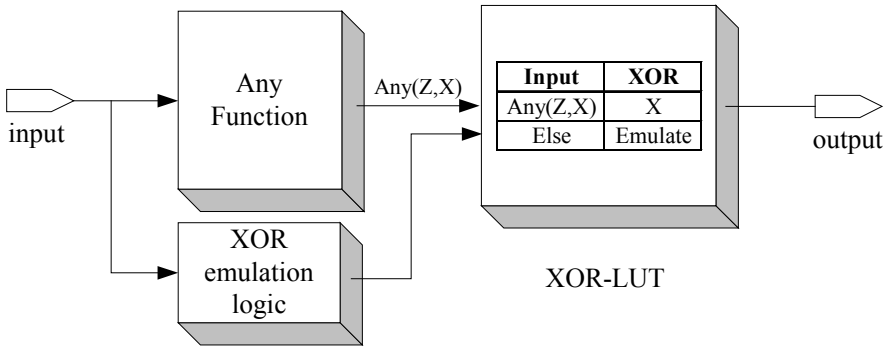
A	B	XOR	XNOR
0	0	0	1
0	1	1	0
0	Z	X	X
0	X	X	X
1	0	1	0
1	1	0	1
1	Z	X	X
1	X	X	X
Z	0	X	X
Z	1	X	X
Z	Z	X	X
Z	X	X	X
X	0	X	X
X	1	X	X
X	Z	X	X
X	X	X	X

The XNOR gate will behave the same as XOR for any High-Impedance and Unknown input case. For Logic-Low (‘0’) and Logic-High (‘1’) case, XNOR will output Logic-High (‘1’) when the input has an EVEN number of Logic-High (‘1’) values and will output Logic-Low (‘0’) otherwise. When the input values only contain Logic-High and Logic-Low values, the behavior of the XOR/XNOR can be *emulated* using a programmable logic device as shown in Table 21, which captures the behavior of XOR/XNOR gates. The lookup table size is also reduced down to 3, but XOR/XNOR requires extra emulation hardware.

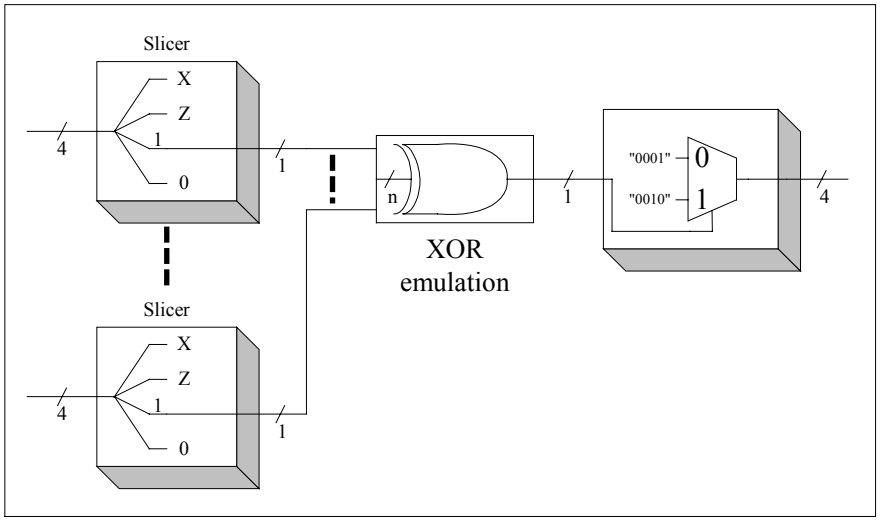
Table 21 Priority Lookup Table for XOR/XNOR Gates

Input	XOR	XNOR
Any(Z,X)	X	X
ELSE	Emulate	Emulate

Figure 22 (a) illustrates XOR gate evaluation design using an *Any*( ) function and Figure 22 (b) shows the actual hardware XOR gate operation. As with AND/OR gates, the lookup table size for XOR evaluation does not change for any number of inputs. Figure 23 (a) and (b) show XNOR gate evaluation and how the emulation circuit is used.



(a)



(b)

Figure 22 XOR Gate (a) Evaluation Design Using Any Primitives, (b) Emulation Logic

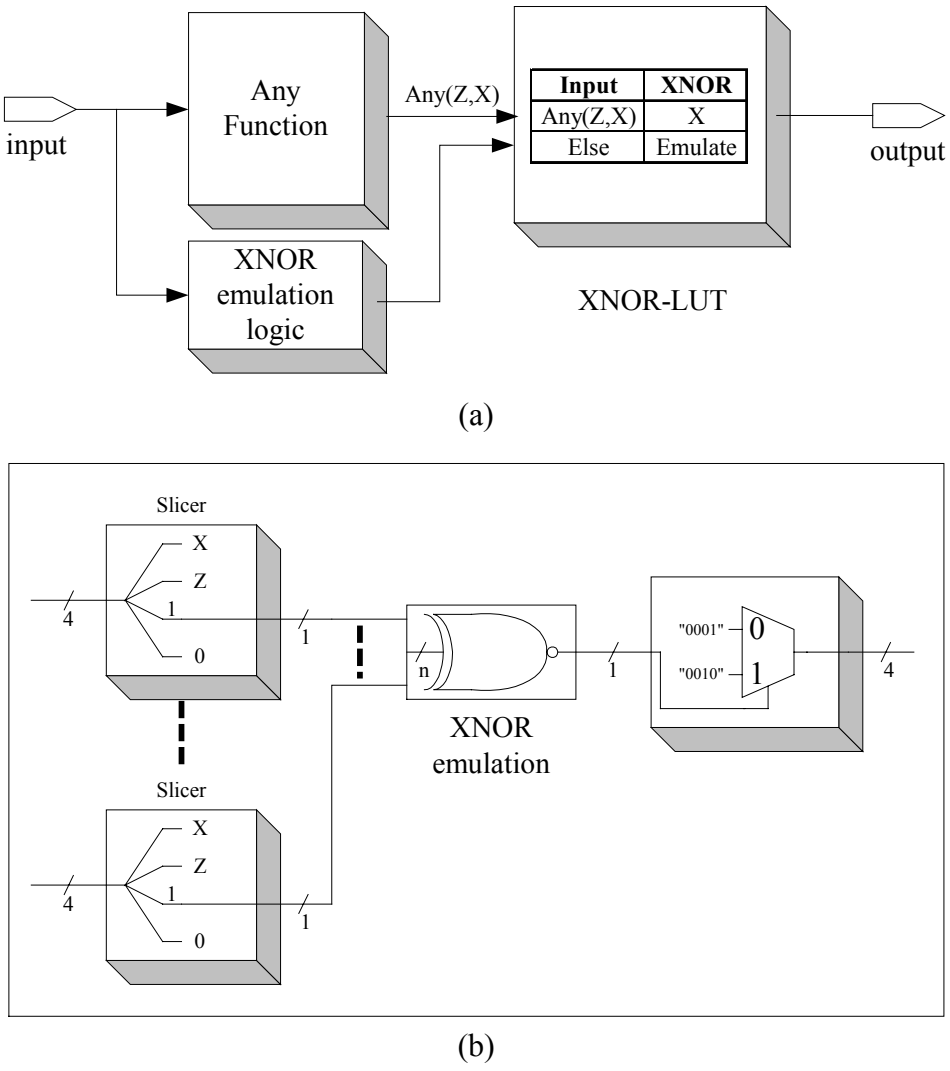


Figure 23 XNOR Gate (a) Evaluation Design Using Any Primitives, (b) Emulation Logic

Table 22 shows the size of the lookup table for both a conventional lookup table and our priority lookup table. The second column indicates the size of standard truth table. For 4-level signal strength with N inputs, the table size is 4^N. The third column states how many Any/All primitives are used to make our priority lookup table. As shown in Figure 22 (a), we are relying on only one primitive Any('Z', 'X'). The fourth column indicates how many of the primitives being used to implement the design and it

follows the number of input. As we can see from the table, if we have an Any( ) function which can perform multiple match, *e.g.* Any('Z', 'X'), then we can reduce the size of our lookup table to a 2 entry table.

Table 22 Lookup Table Size Comparison for XOR/XNOR Gates

Number of Inputs for XOR/XNOR	Lookup Table size	Using Any/All			LUT Reduction Factor
		Number of Primitive Functions	Function Width	Priority LUT Size	
2	16	1	2	2	8
3	64	1	3	2	32
4	256	1	4	2	128
5	1024	1	5	2	512
6	4096	1	6	2	2048
7	16384	1	7	2	8192
8	65536	1	8	2	32768
9	262144	1	9	2	131072
10	1048576	1	10	2	524288
N	4^*N*	1	*N*	2	(4^*N*)/2

In summary, we have introduced the concept of multiple-match *Any*( ) function. We have shown how to incorporate Boolean emulation when input signals are (0 and 1). Experimental results including the XOR gate size and the function width are given in Section 4.9 .

4.4 AO/AOI and OA/OAI Cells

AO gate is a 2-level logic macro cell. The first level gates are composed of multiple AND gates and the second level logic is implemented as a single OR gate. Likewise, OA is a 2-level logic macro cell with multiple OR gates in the first level logic and a single AND gate as second level logic which takes first level OR gate's outputs as its input signals. Figure 24 shows a simple AO22 and gate.

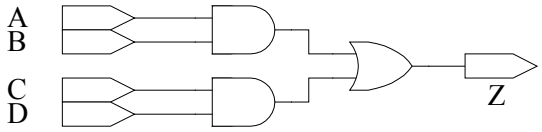


Figure 24 AO22 Gate

There are numerous combinations of AO and OA gates depend on how many first level gates are used and how many inputs per first level gates have, we can build different type of AO and OA gates. AO22 is a particular vendor's cell naming convention. A general form of naming is AO*abcd*, where the variables *abcd* indicates the number of inputs in the first level logic gates. For example, when we have AO432 gate, it means that there are 3 AND gates with 4-inputs, 3-inputs, and 2 inputs, respectively, in the first level logic and their output signals are connected to 3-input OR gate. Therefore, AO22 stands for 2-level AND-OR gates with two 2-input AND gates in the first level and 2-input OR gate in the second level.

AO/OA gate evaluation logic can be implemented using AND/OR evaluation logic as basic building blocks, as discussed in Section 4.2 . The design of AO22 is

illustrated in Figure 25. Notice that the first level gates are AND and the second level logic is the OR gate.

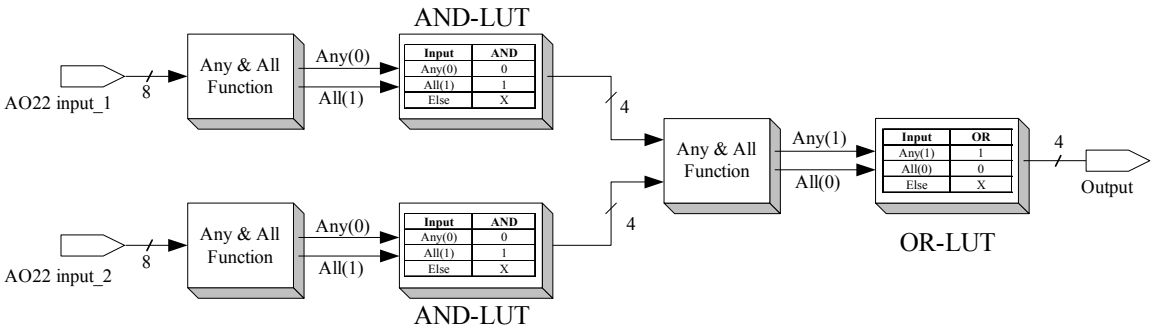


Figure 25 Implementation of AO22 Using AND/OR Evaluation Logic

Since AO/OA cells are built with AND/OR cells, the lookup table size depends on the number of AND/OR gates used to build the AO/OA cells. Table 23 and Table 24 summarize the comparison between standard lookup table and our approach. In Table 23, the first column indicates the type of AO/OA cells. The first item in the first column is for AO22, which contains two 2-input AND gates in the first level. The last entry is for the general case with AO*kkk…k*. It contains *n* AND gates in the first level logic and each AND gate contains *k* inputs.

Table 23 Lookup Table Size for AO Gate

Name Encoding	Number of AND/OR Gates	Number of Inputs	Lookup Table Size	Using one Lookup Table per Gate
22	3	4	256	48
222	4	6	4096	64
333	4	9	262144	256
3333	5	12	16777216	320
4444	5	16	4294967296	1,280
55555	6	25	1.1259E+15	6,144
88888888	9	64	3.40282E+38	589,824
kkk…k	*n+1*	*k * n*	4^(*k*n*)	*(n+1) * (4^k)*

The priority lookup table size for a single AND/OR gate is 3, as was discussed in Section 4.2. There are n gates in the first level logic and 1 extra gate in the second level logic, for a total of (n+1) gates. Therefore, our lookup table size for general AO cell case is $3\times(n+1)$. The standard lookup table still requires exponential size. Since AO cells contain multiple gates, the total number of inputs is quite large ($k\times n$). Thus the size of the lookup table is $4^{(k\times n)}$ for 4 signal strengths. As a result, we were able to achieve linear size growth of the lookup table with $3\times(n+1)$.

Table 24 Priority Lookup Table Size for AO Gate

Name encoding	Using Any/All			LUT Reduction Factor
	Number of Functions	Function Width	LUT Size	
22	2	6	9	5.33
222	2	9	12	5.33
333	2	12	12	21.33
3333	2	16	15	21.33
4444	2	20	15	85.33
55555	2	30	18	341.33
88888888	2	72	27	21,845.33
kkk…k	2	*k*(n+1)*	*3*(n + 1)*	*(n+1)*(4^k) / (3*(n + 1))*

By using Any/All function, we have made a large improvement in the size of the lookup table. Since we are combining AND/OR cells as primitives to construct more complex cells, our priority lookup table size grows linearly as a function of the number of first level gates. But it still is much smaller than the standard lookup table size.

4.5 Universal Gate

As discussed in previous sections, the Any/All functions constitute the basis of our design, and can reduce the size of the lookup table considerably. In this section, we will discuss the implementation of the Any/All functions using a "one-hot encoding" scheme. A simple example will be presented for illustration. By combining the Any/All functions and reduced lookup table, we will construct a "Universal Gate" design.

The properties of Any/All function are given below:

- Any(i): TRUE if any input has a value i.
- Any($i, j,$): TRUE if any input has the value either i or j.
- All(i): TRUE if all inputs are of value i.
- All($i, j,$): TRUE if all inputs are of value i or j.

4.5.1 Any/All Simulation Primitives

Previous sections motivate the need for the Any/All detection circuit. The idea is to mask off the input patterns that we don't want to see and only pass the pattern that we want. We can implement this with simple circuitry and a 4-bit one-hot encoding scheme. In Any/All circuit design, all of the input signals are expanded to 4-bit one-hot encoding as shown in Table 7 and then masked with appropriate values that we wish to look for.

For example, an 'X' is encoded as "1000". The Any('X') function is defined as "Does any input match 1000?". Similarly, All('X') function can be rewritten as "Do all

inputs match 1000?" Figure 26 shows a logic circuit that implements Any( ) and All( ) by bit-wise ANDing a mask (e.g. "1000") with the encoded input and then ORing all of the AND results. Labeled as "match", the output of the OR gate indicates that the input does or does not match the mask. For the Any( ) function, only one match has to be TRUE for Any( ) to be TRUE. Thus the match signals are OR'ed together. For the All( ) function, all inputs must match and the match signals are AND'ed together.

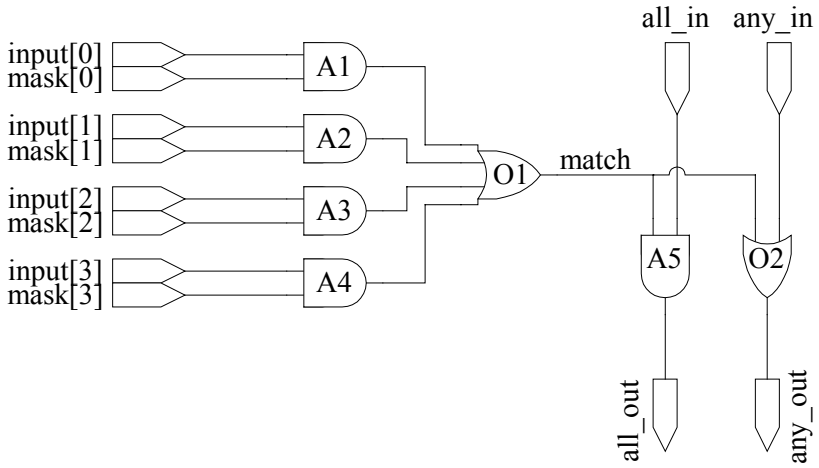


Figure 26 Circuit for Any and All Functions for a Single Signal

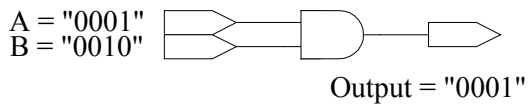
Since we employed a one-hot encoding scheme for our signal representation, one of the properties of our Any/All function is that we can mix the multiple patterns that we are interested in into one mask. For example, to examine Z and X matches from the input signal simultaneously, we can mix Z-mask ("0100") and X-mask ("1000") by applying bit-wise OR operation to those two masks resulting a ZX-mask ("1100"). Therefore, one implementation of Any/All circuit can perform Any('Z', 'X') and All('Z', 'X') functions simultaneously.

As a usage example, assume that we are evaluating a two-input AND gate with the first input (input_1) value set to logic '0' and the second input (input_2) set to logic '1' as shown in Figure 27 (a). The input_1 (value set as 2-bit encoded to "00") will be expanded as "0001" and the input_2 (2-bit encoded as "01") will be expanded to one-hot-encoded value of "0010". Then the appropriate mask value is applied to remove the unwanted input patterns as shown in Figure 27 (c). For AND gate evaluation, we are interested in Any('0') and All('1') patterns. To perform Any('0') function, the zero-mask ("0001") is applied to the mask input port of the circuit shown in Figure 28 (a). The input_1[0] and mask[0] both contain '1', therefore the top AND gate (A_{11}) will produce a '1' output, that drives the OR gate (O_{11}), generating a '1' output. It then passes the value to final gates (A_{15} and O_{12}). The output of the AND gate (A_{15}) will produce a '1' and it is interpreted as All('0') is TRUE. The output of the last OR gate (O2) will also generate a '1' which is interpreted as Any('0') is TRUE. These values will be passed to the input_2 evaluation phase.

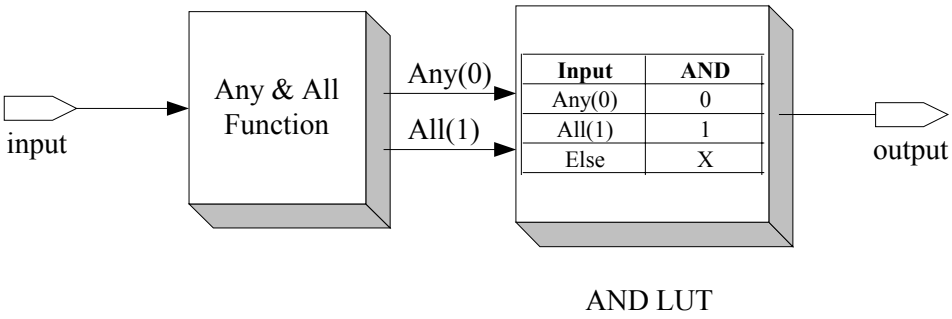
When we apply the zero-mask ("0001") to the input_2 ("0010"), the bit-wise AND will set all the AND gates (A_{21} to A_{24}) to '0', and the output of the OR gate (O_{21}) will also be '0'. This will be AND'ed with previous All('0') result (which was '1') in gate A_{25}, resulting All('0') as FALSE, and OR'ed with previous Any('0') result in gate O_{22}, setting Any('0') as TRUE. Therefore, we conclude that the 2-input AND gate we are evaluating contains at least one zero in its inputs.

At the same time, the one-mask ("0010") will be applied to the Any/All circuit as in Figure 28 (b). For input_1, the bit-wise AND operation performed by gates A_{31} to A_{34}

will generate all ‘0’s as their outputs. Therefore, the output of the OR gate (O_{31}) will produce ‘0’ as output. This in turn will set the All(‘1’) output to FALSE and Any(‘1’) output to FALSE. For input_2, the second AND gate (A_{42}) will generate ‘1’ as output while the others will be set to ‘0’. The output of the OR gate (O_{41}) will therefore be set to 1 and AND’ed with the previous All(‘1’) output, which is FALSE. Therefore the final value of All(‘1’) for this example will be FALSE. The output of the OR gate (O_{41}) will be OR’ed with previous Any(‘1’) value (FALSE) in the last OR gate (O_{42}) and the result will be set to TRUE, which is the final result for Any(‘1’). Therefore, this example contains at least one Logic High (‘1’) as its inputs. Figure 27 (c) shows the inputs and mask values for this example. Our Any/All design has successfully detected All(‘0’) is FALSE, Any(‘0’) is TRUE, All(‘1’) as FALSE, and Any(‘1’)” as TRUE for the given 2-input AND gate. With these results and the priority lookup table shown in Figure 27 (b), the final result for the given 2-input AND gate will be Logic Low (‘0’) because Any(‘0’) was TRUE and All(‘1’) was FALSE.



(a)



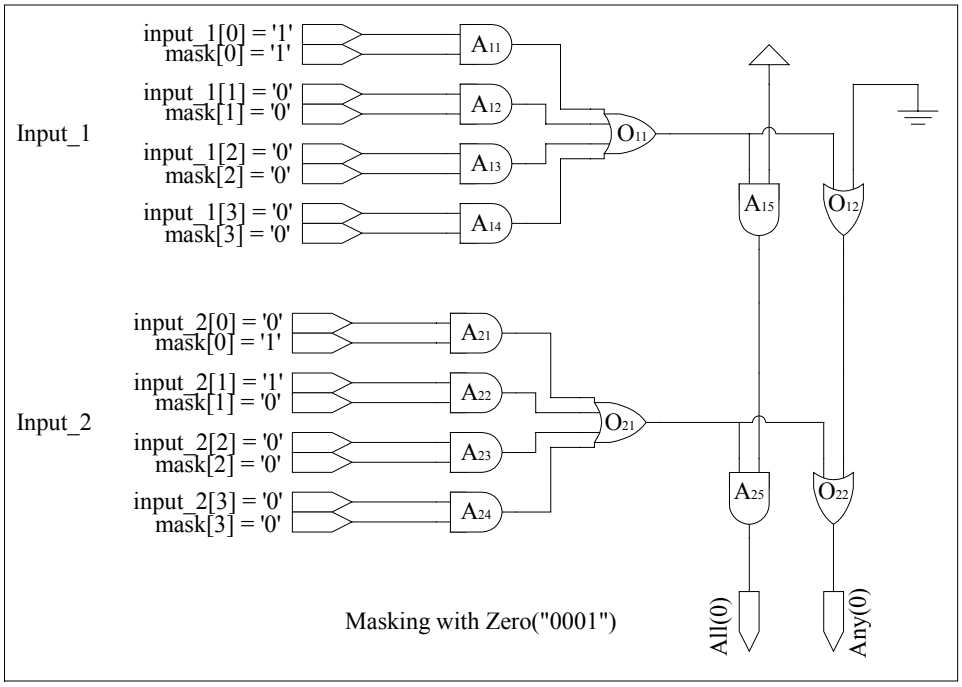
(b)

Any(0)/All(0)				
Input 1	0	0	0	1
Mask 1	0	0	0	1
Input 2	0	0	1	0
Mask 1	0	0	0	1

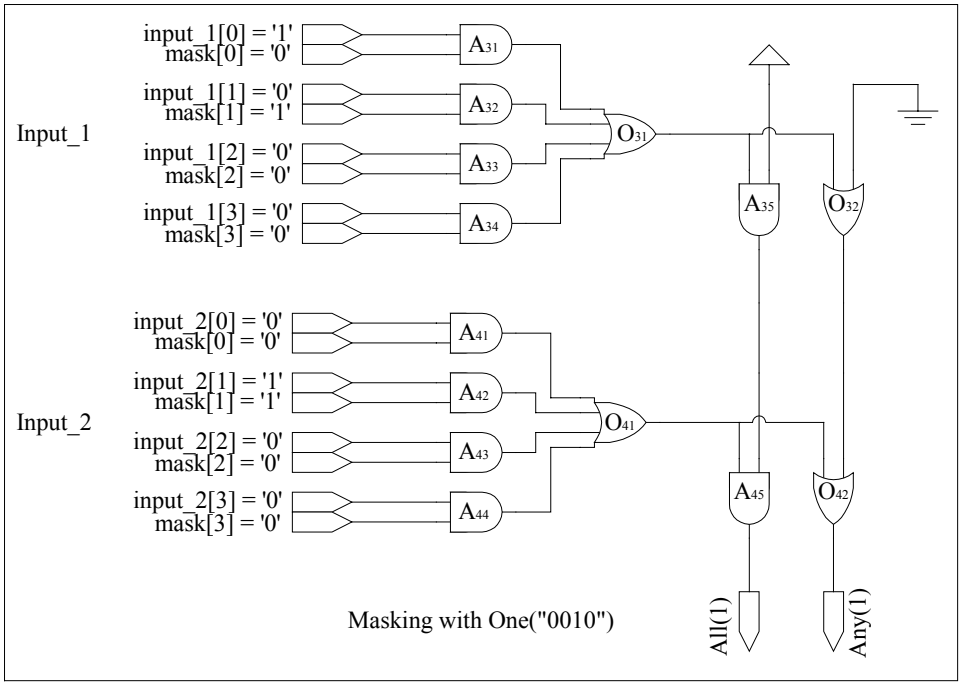
Any(1)/All(1)				
Input 1	0	0	0	1
Mask 2	0	0	1	0
Input 2	0	0	1	0
Mask 2	0	0	1	0

(c)

Figure 27 Any and All Based 2-Input AND Gate Evaluation Example



(a)



(b)

Figure 28 Any and All Primitives for 2-Input AND Gate Example

4.5.2 Universal AND/NAND/OR/NOR

The Universal gates work in pairs for performing Any( ) and All( ) function. AND/OR gates require a universal gate with zero and one masks, XOR gates require a universal gate with Z mask and X mask.

Most vendor libraries have fan-in limits. At the time of this writing, a typical maximum fan-in for a particular library we are using allows up to 5-inputs for logic primitives. For future expansion, we will allow up to 8-inputs for one level logic primitives such as AND/OR gates. The circuit for an 8-input Any/All primitives is shown in Figure 29.

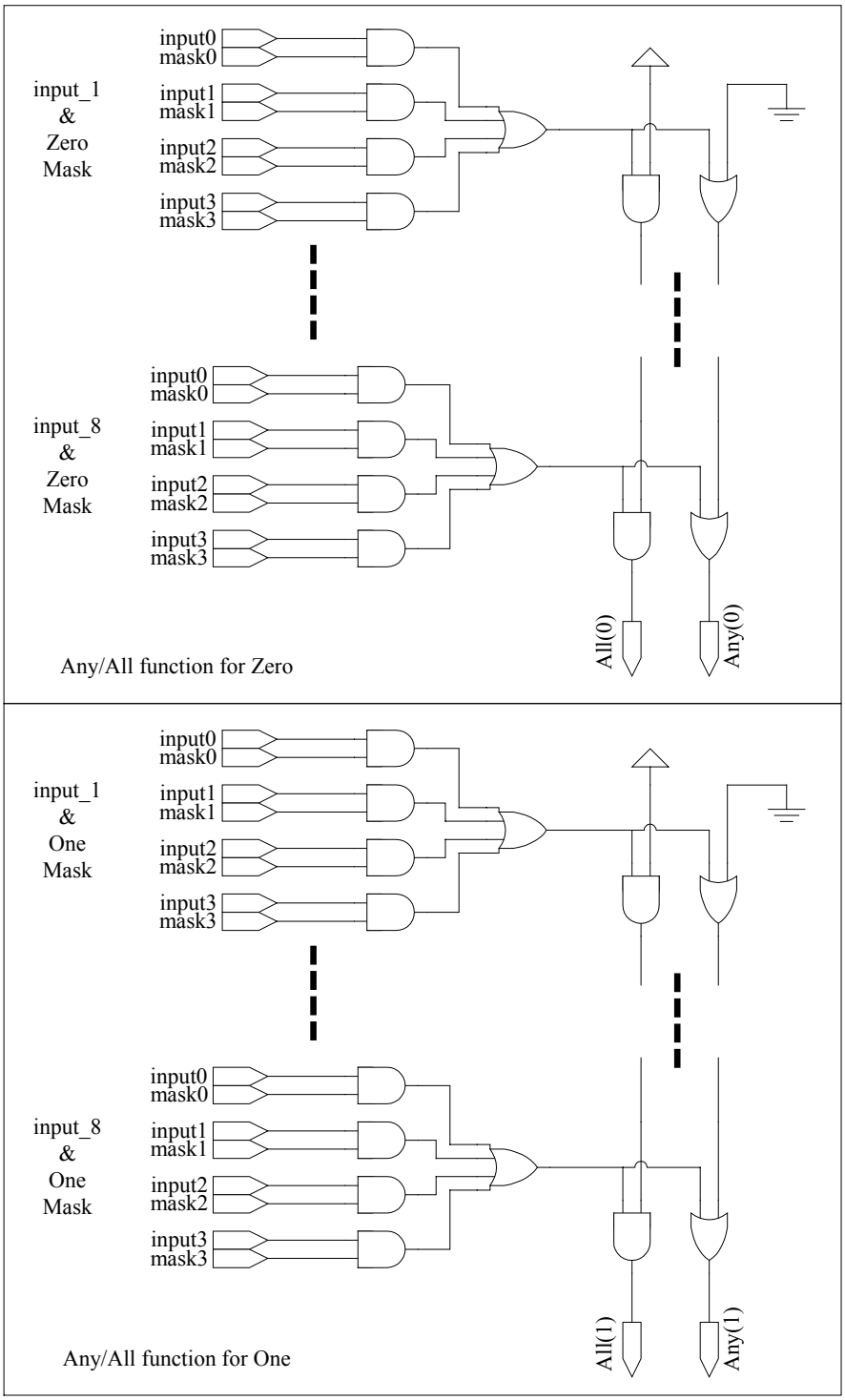


Figure 29 An 8-Input Any/All Design

Figure 30 and Figure 31 show the implementation of 8-input AND gate and 8-input OR gate evaluation engine cores, respectively. The universal gate will generate the Any( ) and All( ) outputs for the given input and then corresponding evaluation logic will determine the final output value based on the priority lookup table described in previous sections. Note that each input has 32-bits as input because each of the 8 inputs are 4-bit wide, due to the one-hot-encoding.

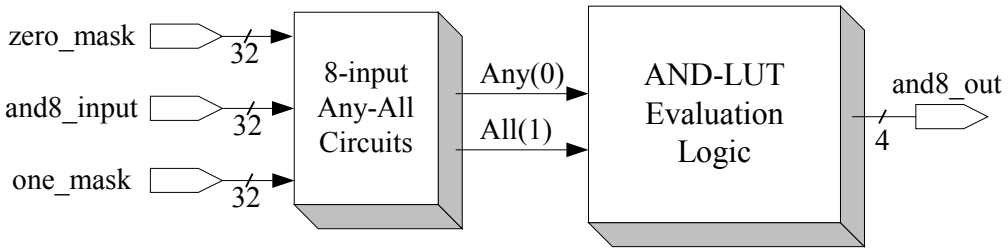


Figure 30 An 8-Input AND Gate Simulation Engine Core

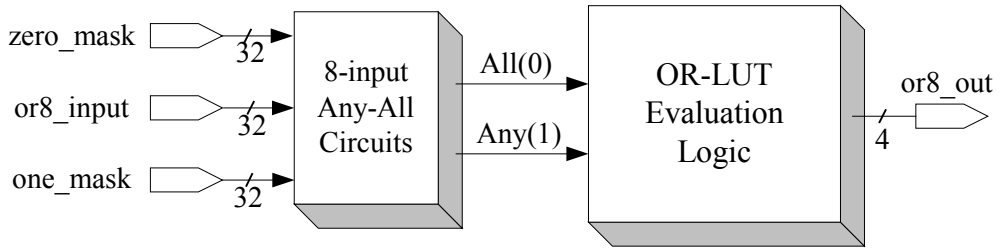


Figure 31 An 8-Input OR Gate Simulation Engine Core

To make these AND and OR evaluation logics more versatile, inversion logic, which was described in Section 4.1 , is added to the input and output ports. The output inversion will allow us to handle NAND evaluation based on AND logic, and NOR evaluation with OR logic circuits. It will also allow us to deal with more diverse forms of

Boolean logic gate evaluation, such as logic gates with partially inverted inputs as shown in Figure 32.

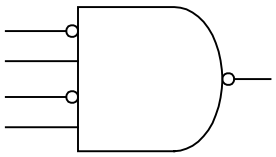


Figure 32 NAND Gate with Some Inputs Inverted

Figure 33 shows our implementation of a universal AND/NAND evaluation logic circuit. Figure 34 describes the universal implementation for OR/NOR evaluation logic circuitry.

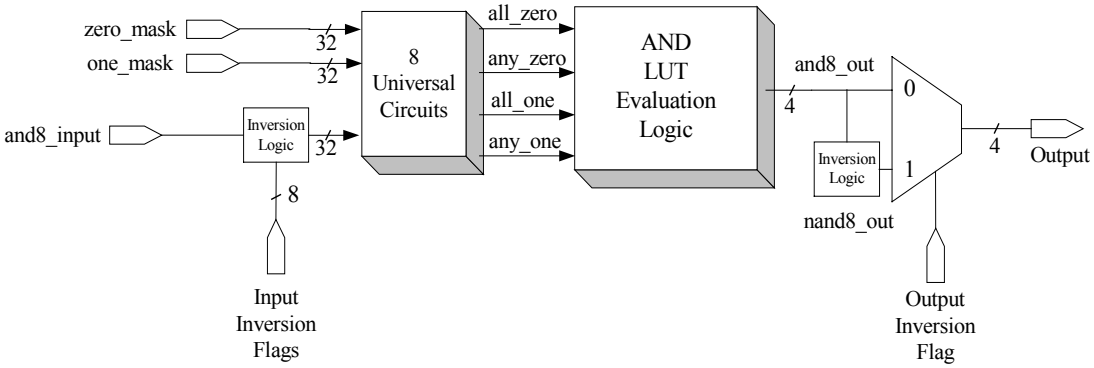


Figure 33 Implementation of 8-Input AND/NAND Gates

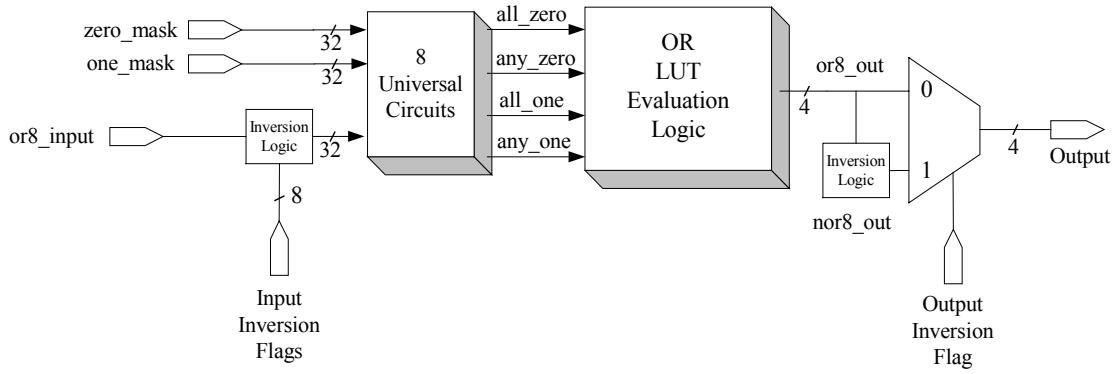


Figure 34 Implementation of 8-Input OR/NOR gates

The evaluation logic for AND/NAND and OR/NOR circuit can be merged into one to form a universal AND/NAND/OR/NOR evaluation logic. Figure 35 is the final form of our universal logic evaluation circuit for 8-input AND/NAND/OR/NOR gates.

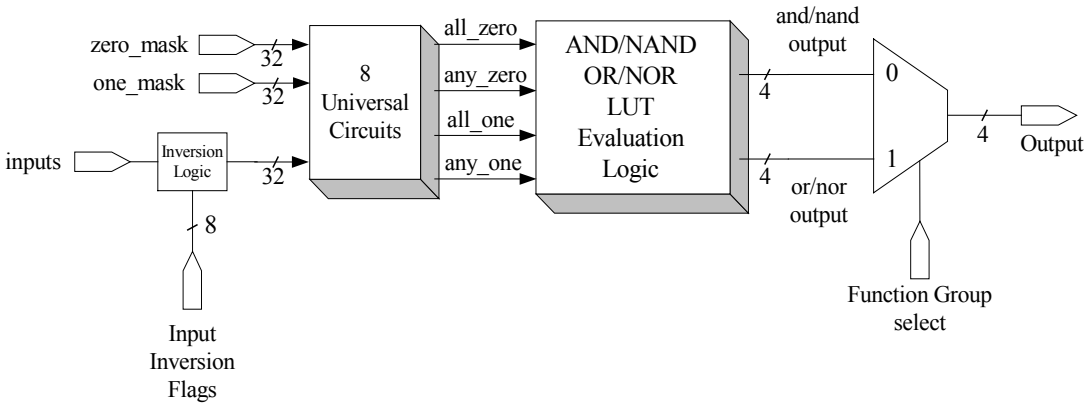


Figure 35 A Universal 8-Input AND/NAND/OR/NOR Evaluation Logic

4.5.3 Universal XOR/XNOR

The XOR/XNOR gate can be evaluated as shown in Figure 36. We use our Universal circuit to detect Any('Z') and Any('X') for the given input vector set and when

either one of them becomes TRUE, we choose Unknown (X) as the XOR gate's output. Otherwise, we can use the built-in XOR logic circuit (emulation) from our hardware platform, because any input vector combination that generates both Any('Z') = FALSE and Any('X') = FALSE means that all the inputs are either 0 or 1. Using XOR emulation will avoid implementing EVEN and ODD detection circuits.

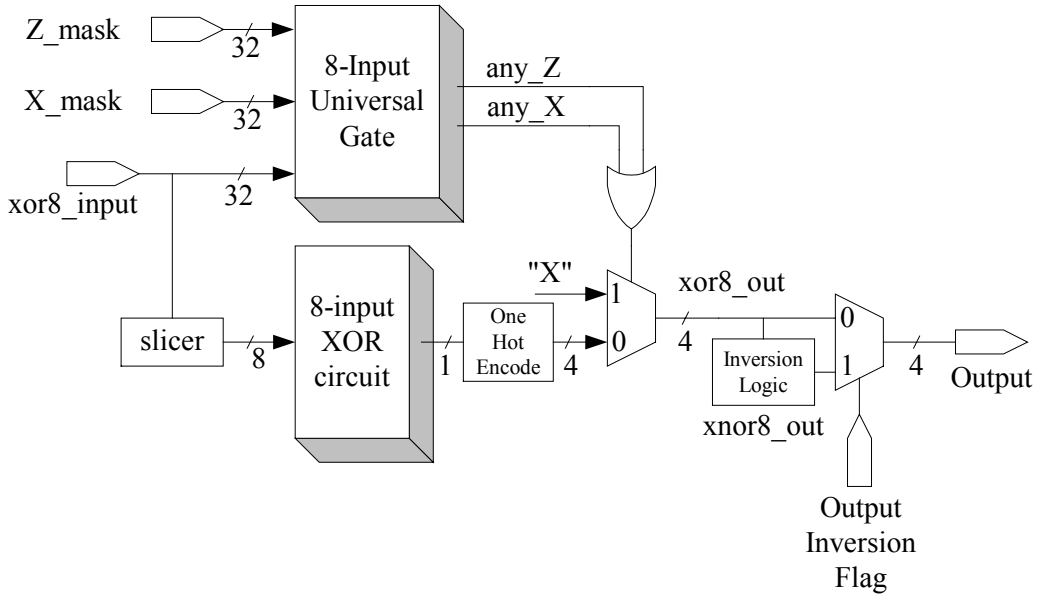


Figure 36 Implementation of 8-Input XOR/XNOR Gates

4.5.4 Universal AO/AOI/OA/OAI

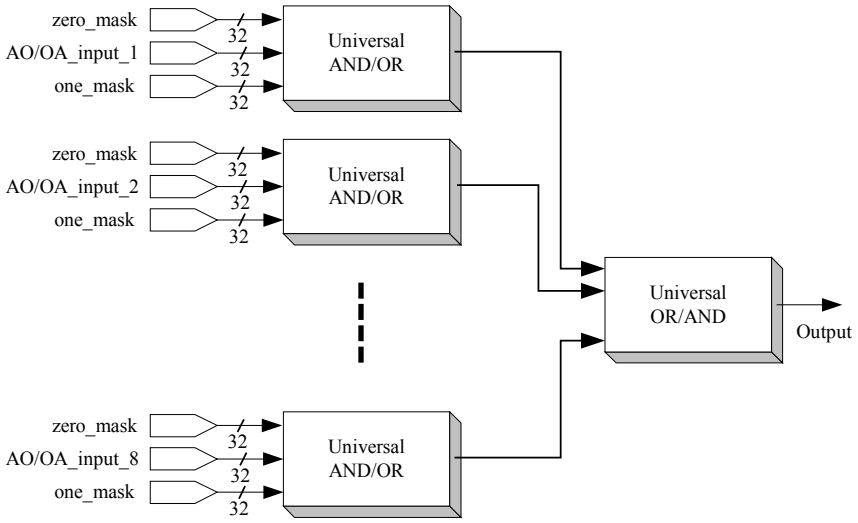


Figure 37 A Universal Implementation of AO/AOI/OA/OAI Evaluation Logic

When we use the universal AND/NAND/OR/NOR logic (shown in Figure 35) for a basic building block, we can implement a universal form of AO/AOI/OA/OAI evaluation logic. Figure 37 illustrates this universal AO/AOI/OA/OAI evaluation logic. The second level evaluation circuit is arranged in such a way that if the first level performs the AND function evaluation, then the second level logic will evaluate the OR function. Likewise, if the first level circuit is evaluating the OR function, then the second level circuit will perform the AND function evaluation. The "Inversion flag" and "Function Group Select" inputs were omitted in the figure for brevity.

4.6 Multiplexer Primitive

A Boolean equation of 2-to-1 Multiplexer (MUX) can be simply written as: *OUTPUT = D0 * SD' + D1 * SD*.

The lookup table for the 2-to-1 MUX is shown in Table 25 using 4-level logic. If we are only dealing with simple Boolean logic levels (1's and 0's), then the implementation of any MUX becomes simple. However, the Boolean equation shown above does not handle multi-level signal strengths such as Unknown ('X') and Hi-Impedance ('Z'). Furthermore, if we model a Multiplexer with a higher number of inputs (4-to-1 or 8-to-1 MUX, *e.g.*) with a lookup table, then the number of entries in the lookup table becomes large. Specifically, a *N*-to-1 MUX has $N + \log_2 N$ inputs and the lookup table has $4^{(N+\log_2 N)}$ entries for 4-level signal strength.

Table 25 shows the lookup table for a 2-to-1 MUX containing 64 entries. This is surprisingly large for a 2-to-1 MUX. Since it has 3 inputs, the possible combinations of signal strength is 4^3 (= 64). But if we observe the Table 25, the behavior model of 2-to 1 MUX can be summarized as following.

- When SD = '0', output is D0
- When SD = '1', output is D1
- When SD = 'Z' or 'X', and D0 = D1, output is D0
- When SD = 'Z' or 'X', and D0 $\neq$ D1, output is 'X'

Table 25 The Lookup Table for 2-to-1 MUX

Input			Output	Input			Output
D0	**D1**	**SD**	**Z**	**D0**	**D1**	**SD**	**Z**
0	0	0	D0	Z	0	0	X
0	0	1	D1	Z	0	1	D1
0	0	Z	X	Z	0	Z	X
0	0	X	X	Z	0	X	X
0	1	0	D0	Z	1	0	X
0	1	1	D1	Z	1	1	D1
0	1	Z	X	Z	1	Z	X
0	1	X	X	Z	1	X	X
0	Z	0	X	Z	Z	0	X
0	Z	1	D1	Z	Z	1	X
0	Z	Z	X	Z	Z	Z	X
0	Z	X	X	Z	Z	X	X
0	X	0	X	Z	X	0	X
0	X	1	X	Z	X	1	X
0	X	Z	X	Z	X	Z	X
0	X	X	X	Z	X	X	X
1	0	0	D0	X	0	0	X
1	0	1	D1	X	0	1	D1
1	0	Z	X	X	0	Z	X
1	0	X	X	X	0	X	X
1	1	0	D0	X	1	0	X
1	1	1	D1	X	1	1	D1
1	1	Z	D0	X	1	Z	D1
1	1	X	D0	X	1	X	X
1	Z	0	D0	X	Z	0	X
1	Z	1	X	X	Z	1	X
1	Z	Z	X	X	Z	Z	X
1	Z	X	X	X	Z	X	X
1	X	0	D0	X	X	0	X
1	X	1	X	X	X	1	X
1	X	Z	X	X	X	Z	X
1	X	X	X	X	X	X	X

The assumption of the above observation is that Hi-Impedance ('Z') input signals are treated as Unknown ('X'). Table 26 shows this summary of a 2-to-1 Multiplexer's behavior. The third item of Table 26 requires a circuit for checking equivalence. If D0

and D1 are the same, then 2-to-1 MUX will choose D0. Otherwise our MUX will output "X" if SD = 'X'. Figure 38 shows this equivalence checking circuit design.

Table 26 Priority Lookup Table for 2-to-1 MUX Primitive (d = don't care)

Input		Output
D0, D1	**SD**	**Z**
d	0	D0
d	1	D1
D0 = D1	Any(Z,X)	D0
ELSE		X

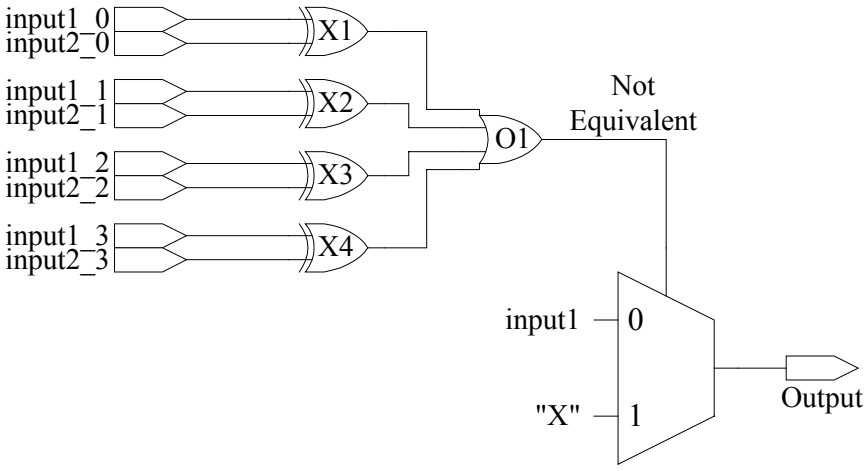


Figure 38 Equivalence Checker for 2-to-1 MUX

Figure 39 shows the implementation of our 2-to-1 MUX design. The circuit contains the actual 2-to-1 MUX and our equivalence checker design. Based on the result of Any('Z') and Any('X'), the proper circuit's output will be chosen by the final 2-to-1 MUX. Notice that a buffer (as was shown in Figure 17) was inserted to filter out the "Z" input and transform it into "X".

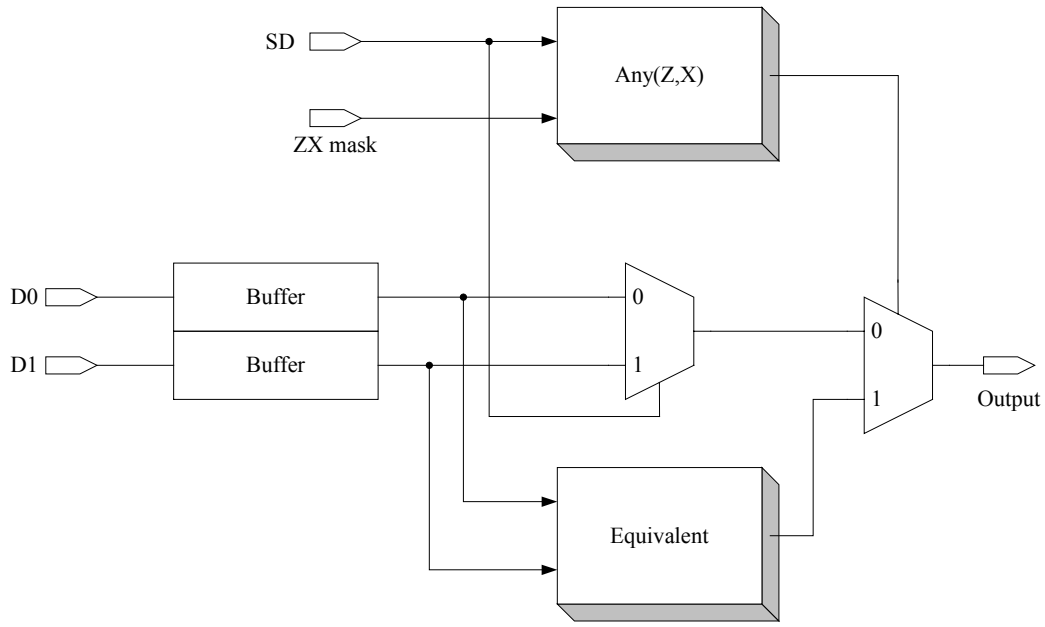


Figure 39 A 2-to-1 MUX Design

A 2-to-1 MUX is the basic building block of all the MUX primitives. A 4-to-1 MUX uses three 2-to-1 MUXes as its components. Table 27 shows the behavior of a 4-to-1 MUX, which is built using three 2-to-1 MUX'es.

Table 27 Priority Lookup Table for 4-to-1 MUX Primitive (d = don't care)

Inputs						Output
D0	**D1**	**D2**	**D3**	**SD1**	**SD2**	**Z**
d	d	d	d	0	0	D0
d	d	d	d	1	0	D1
d	d	d	d	0	1	D2
d	d	d	d	1	1	D3
D0=D2	d	D0=D2	d	0	Z/X	D0
d	D1=D3	d	D1=D3	1	Z/X	D1
D0=D1	D0=D1	d	d	Z/X	0	D0
d	d	D2=D3	D2=D3	Z/X	1	D2
D0 = D1 = D2 = D3				Z/X	Z/X	D0
ELSE						X

Figure 40 shows the implementation of a 4-to-1 MUX design. It uses three 2-to-1 MUXes designed previously as building blocks. An 8-to-1 MUX can easily be designed using two 4-to-1 MUXes and one 2-to-1 MUX as its components.

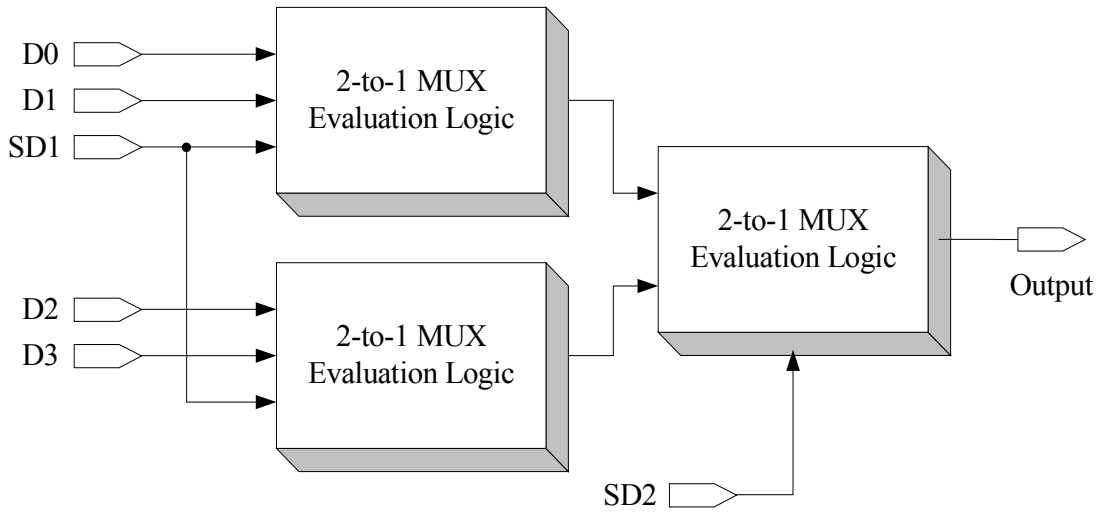


Figure 40 A 4-to-1 MUX Design

In summary, the lookup table for a Multiplexer design can grow very large. But we have reduced the size of the lookup table by behavior modeling and designed a 2-to-1 MUX primitive. This primitive can be used to construct larger Multiplexers. Table 28 shows the size comparison between the standard lookup table and our lookup table. As we can see from the second column, the standard lookup table's size grows exponentially, while our approach grows linearly.

Table 28 Lookup Table Size Comparison for MUX

Number of Inputs	Using Standard Lookup Table	Using Any/All			LUT Reduction Factor
		# of Functions	Function Width	LUT Size	
2	16	4	2	2	8
4	256	12	6	6	43
8	65536	28	14	14	4681
16	4294967296	60	30	30	143165577
N	*4^N*	*4 * (N-1)*	*2 * (N-1)*	*2 * (N-1)*	*(4^N) / (2 * (N-1))*

4.7 Full Adder

A Full Adder has two outputs. They are Sum, and CarryOut. Both can be expressed in the combinational Boolean equations.

$Sum = A \oplus B \oplus C$

$Cout = (A \bullet B) + (B \bullet C) + (C \bullet A)$

Where, *A* and *B* are the inputs for the Adder, *C* is Carry-in, *Cout* is Carry-out.

Table 29 shows the exhaustive list of the full adder cell in lookup table form. The size of standard lookup table is 4^3. A single Full Adder gate has 64 entries due to its 3 input lines. But from the equations shown above, we can observe that *Sum* is simply a three input XOR gate and *Cout* is also a simple AO gate, which we have already modeled in previous sections. Therefore the lookup table calculations from Table 21 and Table 23 can be used to compute lookup table for our design.

Table 29 Lookup Table for Full Adder

A	B	C	Sum
L	L	L	L
L	L	H	H
L	L	Z	X
L	L	X	X
L	H	L	H
L	H	H	L
L	H	Z	X
L	H	X	X
L	Z	L	X
L	Z	H	X
L	Z	Z	X
L	Z	X	X
L	X	L	X
L	X	H	X
L	X	Z	X
L	X	X	X
H	L	L	H
H	L	H	L
H	L	Z	X
H	L	X	X
H	H	L	L
H	H	H	H
H	H	Z	X
H	H	X	X
H	Z	L	X
H	Z	H	X
H	Z	Z	X
H	Z	X	X
H	X	L	X
H	X	H	X
H	X	Z	X
H	X	X	X

A	B	C	Sum
Z	L	L	X
Z	L	H	X
Z	L	Z	X
Z	L	X	X
Z	H	L	X
Z	H	H	X
Z	H	Z	X
Z	H	X	X
Z	Z	L	X
Z	Z	H	X
Z	Z	Z	X
Z	Z	X	X
Z	X	L	X
Z	X	H	X
Z	X	Z	X
Z	X	X	X
X	L	L	X
X	L	H	X
X	L	Z	X
X	L	X	X
X	H	L	X
X	H	H	X
X	H	Z	X
X	H	X	X
X	Z	L	X
X	Z	H	X
X	Z	Z	X
X	Z	X	X
X	X	L	X
X	X	H	X
X	X	Z	X
X	X	X	X

A	B	C	Cout
L	L	L	L
L	L	H	L
L	L	Z	L
L	L	X	L
L	H	L	L
L	H	H	H
L	H	Z	X
L	H	X	X
L	Z	L	L
L	Z	H	X
L	Z	Z	X
L	Z	X	X
L	X	L	L
L	X	H	X
L	X	Z	X
L	X	X	X
H	L	L	L
H	L	H	H
H	L	Z	X
H	L	X	X
H	H	L	H
H	H	H	H
H	H	Z	H
H	H	X	H
H	Z	L	X
H	Z	H	H
H	Z	Z	X
H	Z	X	X
H	X	L	X
H	X	H	H
H	X	Z	X
H	X	X	X

A	B	C	Cout
Z	L	L	L
Z	L	H	X
Z	L	Z	X
Z	L	X	X
Z	H	L	X
Z	H	H	H
Z	H	Z	X
Z	H	X	X
Z	Z	L	X
Z	Z	H	X
Z	Z	Z	X
Z	Z	X	X
Z	X	L	X
Z	X	H	X
Z	X	Z	X
Z	X	X	X
X	L	L	L
X	L	H	X
X	L	Z	X
X	L	X	X
X	H	L	X
X	H	H	H
X	H	Z	X
X	H	X	X
X	Z	L	X
X	Z	H	X
X	Z	Z	X
X	Z	X	X
X	X	L	X
X	X	H	X
X	X	Z	X
X	X	X	X

Figure 41 illustrates the design of the Full Adder gate. A universal XOR and a universal AO cells are reused to implement this design.

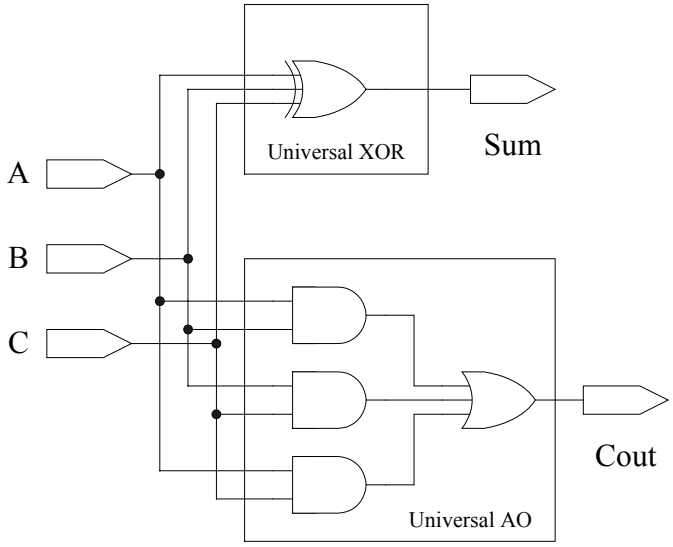


Figure 41 Full Adder Design

From the Table 22 and Table 24, the priority lookup table size for XOR in Figure 41 is 2 and the priority lookup table size of Universal AO222 is 12. Therefore our lookup table size for Full Adder implementation is 13. In summary, for our Full Adder primitive design, we were able to achieve a lookup table size reduction of 64/13 = 4.9 times smaller without creating additional primitives.

4.8 Flip-Flop Evaluation

Flip-Flops contain special inputs such as “clock”, “clear”, and “preset”. The “clear” and “preset” inputs are asynchronous inputs and will be discussed in the end of

this section. The "Clock" input is a special signal that triggers the action on its signal level transitions (on its edges).

The cells discussed in previous sections rely only on its signal events, but the clock signal requires edge-event. The clock edge event in essence is also a signal event, except that the signal needs to be compared with its previous value. Otherwise, this clock edge event will increase the number of event types and complicates our signal-encoding scheme (*i.e.*in addition to 0, 1, Z, X signals, we need to encode a rising event, a falling event, etc.). If we include this clock value comparison mechanism inside of the Flip-Flop evaluation hardware, our current encoding scheme can still be used. This will reduce the scheduler's load because the scheduler does not have to process different event types. Table 30 lists the possible clock signal transitions for the Flip-Flop shown in Figure 42.

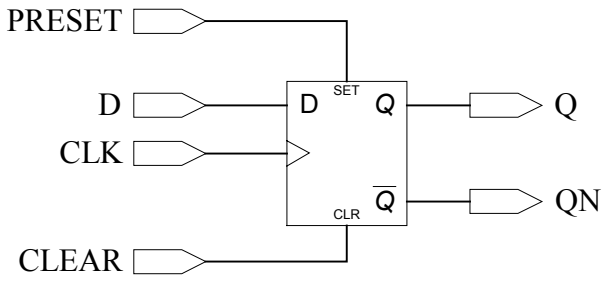


Figure 42 D-type Flip-Flop

Table 30 illustrates the behavior of a positive edge triggered D Flip-Flop on a various clock signal transitions. The behavior of the Flip-Flop falls into two groups. When the current clock value Logic-High ('1') and the previous clock value was not Logic-High ('0', 'Z', 'X'), then the data input ('D') will be latched in. Anything else will not cause the Flip-Flop to react. All we need is to detect the clock signal has been risen.

To detect this clock rising event, we can use Any/All primitive that we developed in previous section. If Any('1') for current clock is TRUE and if Any(0, Z, X) of previous clock is TRUE then clock rising event is true. Figure 43 shows this design.

Table 30 Behavior of Positive-Edge Triggered D Flip-Flop

Previous CLK	Current CLK	Transition Notation	CLK Event Type
0	0	(00)	No change
0	1	(01)	Trigger
0	Z	(0Z)	No change
0	X	(0X)	No change
1	0	(10)	No change
1	1	(11)	No change
1	Z	(1Z)	No change
1	X	(1X)	No change
Z	0	(Z0)	No change
Z	1	(Z1)	Trigger
Z	Z	(ZZ)	No change
Z	X	(ZX)	No change
X	0	(X0)	No change
X	1	(X1)	Trigger
X	Z	(XZ)	No change
X	X	(XX)	No change

As shown in Table 30, there are 16 different clock transitions. If we implement this in a lookup table, all the possible clock transitions have to be enumerated as well. To do this, the clock signal needs to be encoded in four bits rather than two bits (one for current clock and the other for previous clock) and it will make the table size even bigger. Therefore, the Flip-Flop shown above figure has 4-inputs total, but it appears as 5-inputs due to the clock signal encoding. The lookup table size of D Flip-Flop will be $4^5 = 1024$.

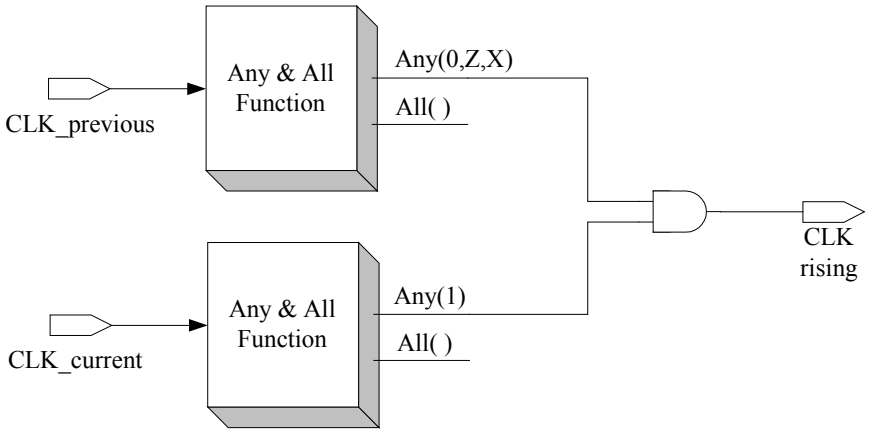


Figure 43 Clock Event Detection Design

When we examine the behavior of the Flip-Flop with data input ('D'), we can characterize the behavior in lookup table format as in Table 31. Notice that the table contains the priority. The first item in the table takes highest priority over other items. By using the equivalence checker circuit described in Figure 38, the data input value and the Q_OLD value are checked first. The rest of the items in the table are considered when the equivalence check becomes FALSE.

Table 31 Priority Lookup Table for D Flip-Flop

Condition	Output
CLK rising	D
ELSE	Q_OLD

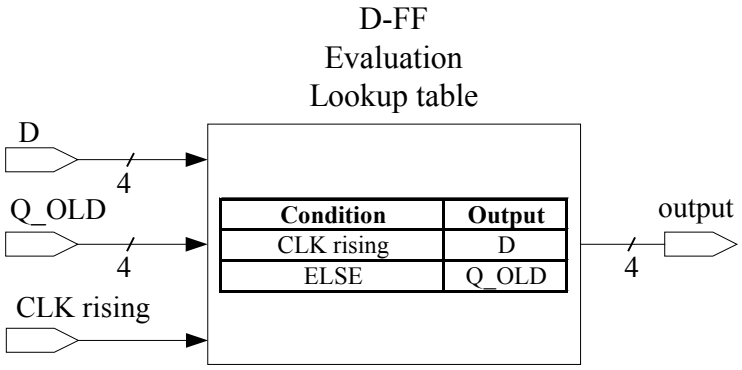


Figure 44 D Flip-Flop Evaluation Core Design

The Clear and Preset inputs are asynchronous inputs. They take the highest priority over any other input signals, so the Flip-Flop evaluation algorithm should always check these values first. Table 32 lists the "Clear" and "Preset" behavior. Notice that "Clear" and "Preset" should never be enabled together.

Table 32 Behavior Model of Clear and Preset

Clear	Preset	Async. Info
0	0	Normal Op
0	1	Preset
0	Z	No change
0	X	No change
1	0	Clear
1	1	Illegal
1	Z	Clear
1	X	Clear
Z	0	No change
Z	1	Preset
Z	Z	No change
Z	X	No change
X	0	No change
X	1	Preset
X	Z	No change
X	X	No change

If they are enabled together, then the Flip-Flop falls into an illegal state and the output value becomes the Unknown (‘X’) state. When “Clear” or “Preset” is either Hi-Impedance (‘Z’) or Unknown (‘X’), then they are not considered strong enough to cause the action to happen. Again, using the Any/All primitives, we can implement this asynchronous behavior of the Flip-Flop. Figure 45 shows this behavior. Each Any/All function will detect Any(‘1’) for each input. They are then merged into a 2-bit signal (Asynchronous information), which selects desired output.

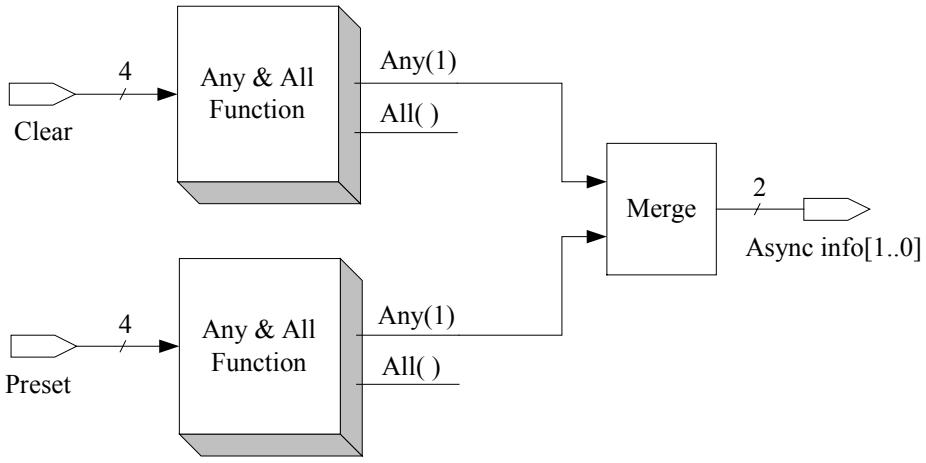


Figure 45 Design for Checking Clear and Preset

Combining all the components together with a 4-to-1 MUX, we can model the D-type Flip-Flop with asynchronous “Clear” and “Preset”. Figure 46 shows the design implementation. When Any(‘1’) for “Clear” and Any(‘1’) for “Preset” are both TRUE then the 4-to-1 MUX will select ‘X’ as its output. When Any(‘1’) for “Clear” is TRUE and Any(‘1’) of “Preset” is FALSE, then the MUX will select ‘L’ as its output and vice versa.

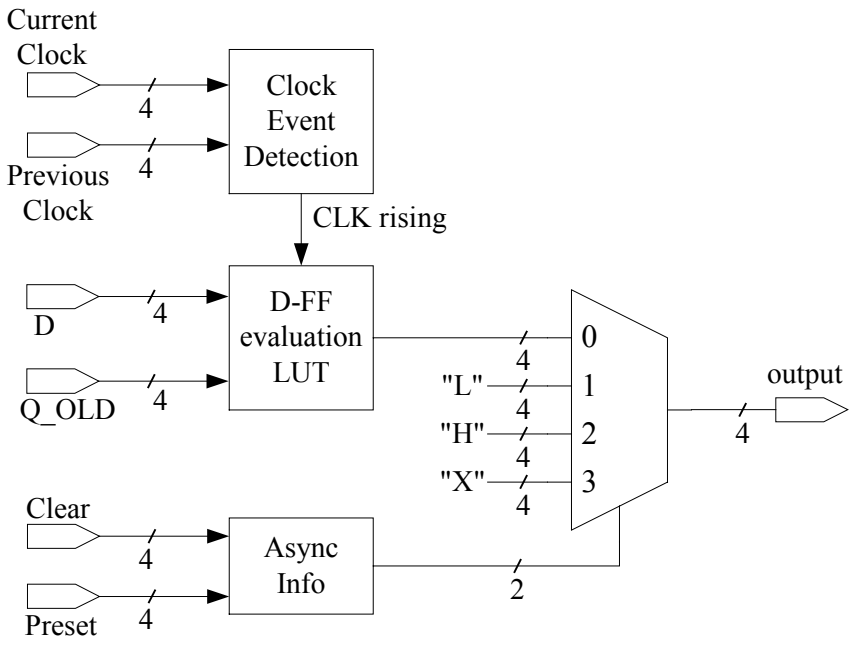


Figure 46 Implementation of D Flip-Flop with Asynchronous Clear and Preset

For D Flip-Flop modeling, we have used 4 Any/All primitives and a 4-to-1 MUX to implement the evaluation logic. This is approximately equivalent to the size of 4-input AND gate evaluation design.

4.9 Scalability of Primitives and Experimental Results

This section presents the size comparison for the lookup table based implementation of evaluation and our approach, which is based on Any/All primitives and a priority lookup table. Altera's EP20K200EFC672-1X was used to synthesize both implementations. The compilation report for resource usage is summarized in Table 33 in terms of Logic Elements (LEs) used by the FPGA manufacturer. The concept of Altera's Logic Element (LE) will be briefly described in the following section. As we

can see from the table, our design grows linearly as the number of inputs grows. On the other hand, the size of the standard lookup table approach grows exponentially, as we expected from previous discussions, while our priority lookup table size grows nearly negligible. Figure 47 shows this behavior. Notice that the standard lookup table approach for 8-input AND gate has failed to synthesize for the target platform.

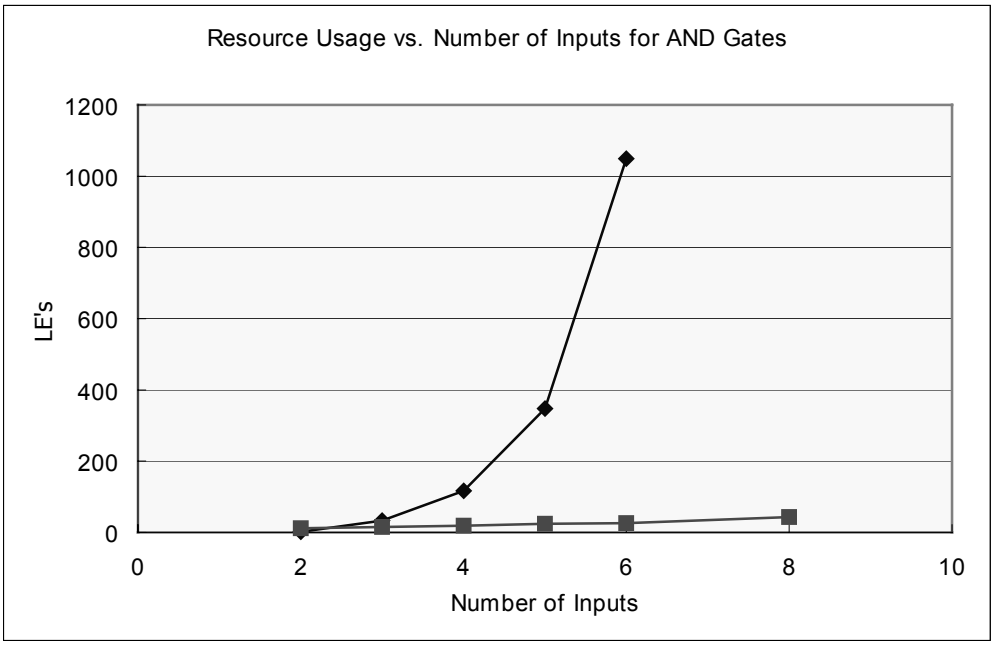


Figure 47 Growth Rate of Resource Usage for Lookup Table

Table 33 Resource Usage Comparison

	LUT	PLUT
AND2	2	12
AND3	33	15
AND4	117	19
AND5	348	24
AND6	1049	26
AND8	FAILED	43

Table 34 lists the resource usage for Any/All primitives with different number of inputs. Table 35 shows the size of logic evaluation primitives. The logic primitives are pre-scaled to handle up to maximum allowed inputs. As a conclusion, the universal gate and other primitives with priority lookup table approach have optimized the hardware resource usage within a linear growth to the number of inputs.

Table 34 Resource Usage for Any/All Primitives

Number of Input	LE
1	4
2	8
3	10
4	14
5	17
6	20
7	24
8	27

Table 35 Resource Usage for Logic Evaluation Primitives

Altera's EP20K200EFC672-1X	
Primitive	**LE**
Univ_AndOr8	145
Univ_AoOa8x8	686
Univ_XOR8	143
MUX41	95
FA	48
D-FF	21
Priority LUT with Size 3	2
INVERTER	3

4.10 Altera's Logic Element

A Logic Element (LE) is a basic functional building block of Altera's FPGA chip [30]. Within each LE is a four input lookup table (LUT), a D-type flip-flop with

programmable control logic, cascade chain interconnects, and routing multiplexers. The usage of the LUT and cascade chain allows LUTs from different LEs to be connected in such a fashion as to allow large Boolean functions or special purpose logic to be implemented. Further discussion about FPGA is out of scope of this work, and will not be discussed any further. Figure 48 shows the Logic Element Architecture provided by Altera [30].

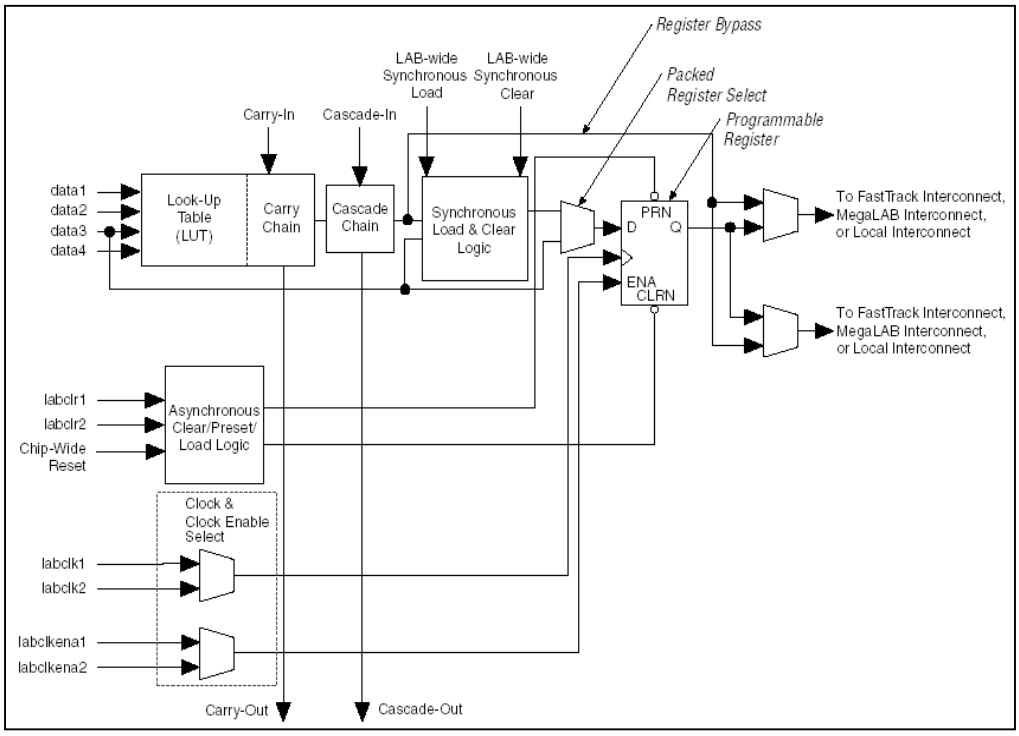


Figure 48 A Logic Element (LE) Architecture

5.0 GATE DELAY AND FUTURE EVENT GENERATION ARCHITECTURE

Circuit designs start at a high-level of abstraction and go through a series of transformations until the final mask layout is obtained. One of the synthesis steps is "technology mapping", which decides the CMOS technology to be applied to the design (*e.g.*, 0.25 μ-process, 0.16 μ-process, *etc.*). Technology mapping determines switching characteristics, such as rise and fall time delay of the cells, capacitive load of wire, *etc*., which are defined by the cell library.

To simulate the logic design accurately, a full-timing simulation must be performed. Full-timing simulation requires that accurate gate delay models be incorporated into the logic simulation. The problem with full-timing simulation is that it requires extra operations to the simulation procedure, and the logic evaluation process slows down.

There are two categories of delays to consider in the logic design. One is *intrinsic delay*, caused by the gate itself. And the other is the *extrinsic delay* due to the fan-out gate load and wire load. Also, there are three types of delays to be considered depending on the gate's operating condition. They are *fixed delay*, *path dependent delay*, and *state dependent path delay*. In this chapter, we discuss these different delay types and describe the architecture required to utilize each type of delay. The major issue with this portion of the design is efficiently storing the delay information such that it can be accessed by dedicated hardware in the shortest time possible.

5.1 Delay Types

There are three types of gate delays defined by the IEEE Standard Delay Format (SDF) [32]. They are (1) *fixed delay*, (2) *input-to-output path delay* (*I/O-path delay*), and (3) *state dependent input-to-output path delay*. The SDF standard also defines the delay format in the form of 3-tuple as (minimum:typical:maximum). These three numbers are based on the fabrication condition and operating condition.

Fixed delay is given as a rise/fall time pair for an entire gate depending on the output change. Most simple logic primitives are defined using this fixed format. Fixed delays require only 2 values per gate, as the gate delay is modeled as a function of output rise and fall.

Path dependent delay is modeled as a function of input-to-output path and the output rise/fall. Based on how the ASIC cell vendors implement their cells (*i.e.* transistor size, and layout), the delay value can vary based on which input caused the output change (I/O-path). An *N* input gate with *path dependent delay* will have *N* pairs of rise/fall time delay information.

Furthermore, in the *state dependent delay* model, path dependent delays can be further differentiated depending on the state of the other input signals. *State dependent delay* allows more accurate delay modeling according to the input-to-output path and the state of the input combination. For example, on a XOR gate, a delay path formed by an input to the output will be determined by other input signal's value at that moment. The state dependent delay is illustrated in Figure 49. Figure 49 (a) shows the delay path from

input *A* to the output port when the other input *B* is '0'. Input *B* will force the AND gate (A_1) to stay at '0', which will not affect the OR gate (O_1), therefore the output change through this AND gate (A_1) will not affect any further signal propagation. The only signal path that will affect the output signal change is from the input signal *A* to the AND gate (A_2) and through the OR gate (O_1). Likewise, as shown in Figure 49 (b), when *B* is '1', the AND gate (A_2) will remain at '0' and will not affect the output of OR gate (O_1). The only signal path is from the input *A* to the inverter (I_1) through the AND gate (A_1), and then through the OR gate (O_1).

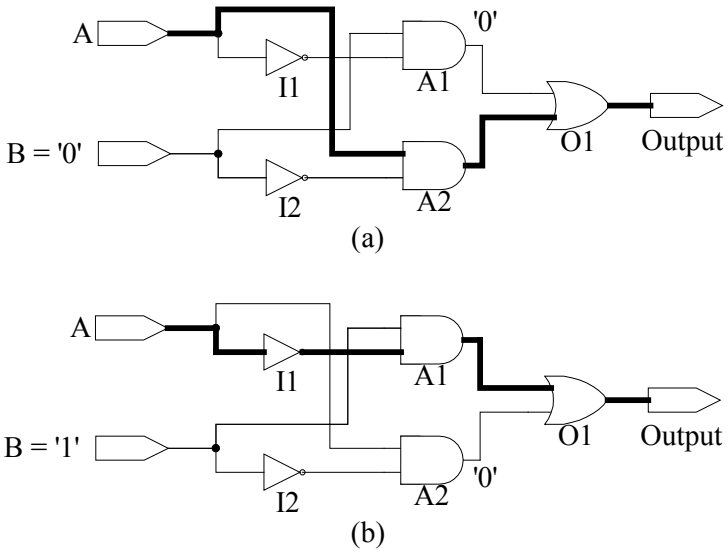


Figure 49 Path Dependent Delay of 2-Input XOR Gate (a) When B = '0'; (b) When B = '1'

In general, an *N* input gate with *state dependent delay* can have $N \times 2^{(N-1)}$ state dependent delay values, because for each of *N* inputs, there are (*N-1*) other inputs whose binary combination value will determine the state. For example, a 4-input XOR contains 4×2^3 = 32 different states; each state has set of rise and fall time delay information. Therefore, a 4-input XOR gate contains a total of 64 different delay values when

accounting for the rise/fall time delays. Thus, the total number of delays for an *N* input gate with *state dependent delay* is $N \times 2^N$.

In addition to the gate delays, in the full-timing delay model, the delay value is divided into *intrinsic* and *extrinsic* delays. The *intrinsic* delay value is the logic gate's own delay caused by internal transistors when the gate is not driving anything. The *extrinsic* delay is caused by the wire that connects from the output to the other logic gate's input and the capacitive load of the other gates being driven by the logic gate.

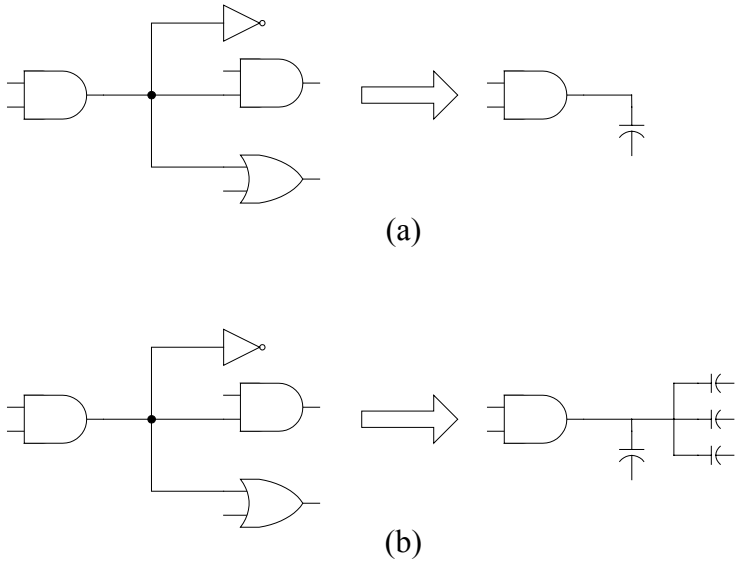


Figure 50 Delay Models (a) Lumped Delay; (b) Distributed Delay

The extrinsic delay can be modeled as "*lumped delay model*", where the wire and input gate load is lumped into one large capacitive load, or can be modeled as "*distributed delay model*" where the output wire is segmented into regions and different capacitance values are assigned to the wire segments. Figure 50 illustrates these delay models. This work is focused on the "*lumped delay model*". Distributed delay can be modeled with the lumped delay model by using "zero delay buffers" to model each wire.

Zero delay buffers have no *intrinsic* delay but can have *extrinsic* delay. Figure 51 shows how to model "distributed delay" using "lumped delay modeling."

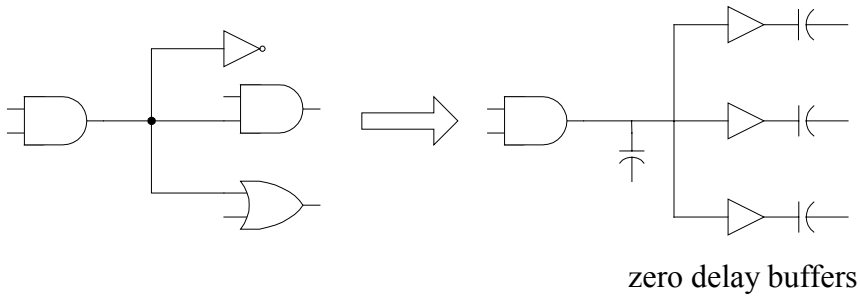


Figure 51 Distributed Delay Modeling Using Lumped Delay Model

As discussed in previous chapter, most of the existing logic simulation only handle the zero-delay or the unit-delay model, where the delay through each gate is treated as either zero or a single abstract time unit. These zero-delay and unit-delay models can only be used for pre-technology mapping to check the design correctness (logical correctness) and cannot be used for post place and route (post-technology mapping) where the actual delay can cause serious timing malfunction.

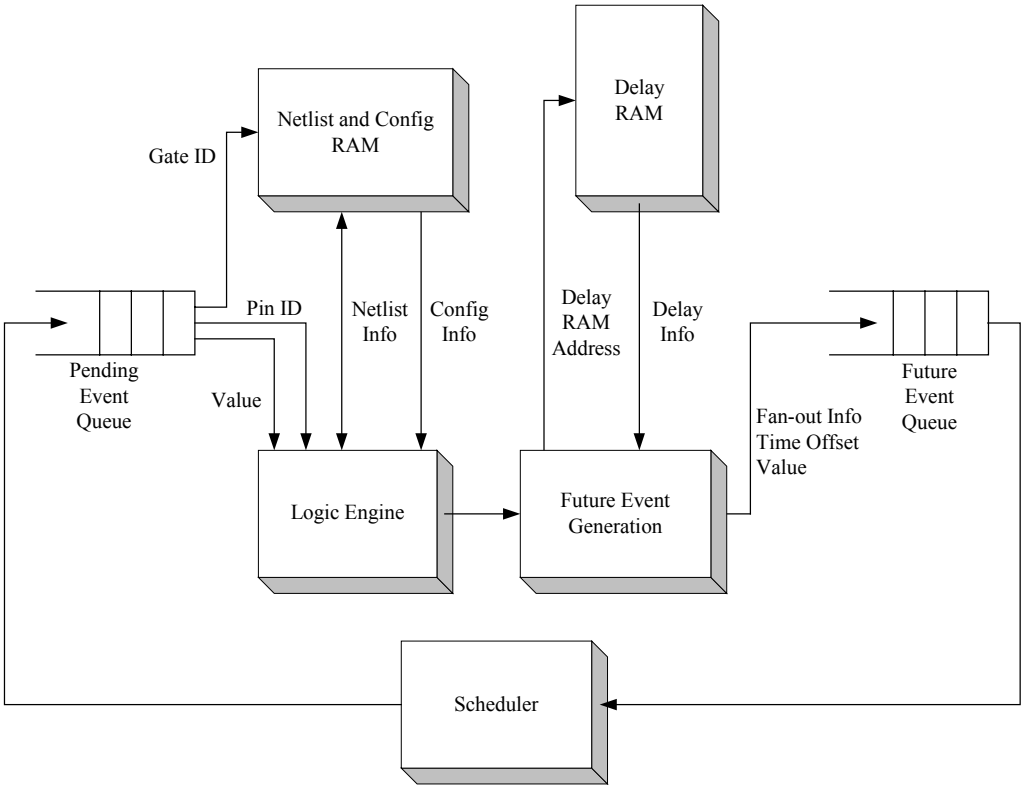


Figure 52 The Delay Architecture

As shown in Figure 52, the scheduler determines when the events are to be processed and in what order. The events to be processed in the future are labeled "Future Events" and are stored within the scheduler. Also previously discussed, the current simulation time is called Global Virtual Time (GVT). The events whose execution time stamp is equal to current GVT are labeled as *Pending Events* and are sent to the Logic Engine for immediate processing. The Logic Engine evaluates each *Pending Event* on the specified gate, and, if the output of the gate changes, a *Future Event* is generated. While the Scheduler (described in Chapter 6) performs an important role in the ordering of events, this chapter is on efficiently determining the delay value that should be associated with each gate.

5.2 Net-list Update and Future Event Generation

The simulation engine updates the net-list memory whenever there is an evaluation action triggered by the pending event queue (PEQ). The update is performed whether the output of the gate being simulated is changed or not, because the net-list has to maintain the current input value. Therefore, when the output value has been changed due to the event evaluation of the gate, the logic engine updates both the input and output of the gate in the net-list. If an output change does not occur, the engine still performs the net-list update with the current input change only.

When the result of a functional evaluation indicates that there has been a change in the output value, then the logic engine will generate a future event and pass it to the scheduler so that it can be arranged in the proper future time. The contents of the future event queue (FEQ) passed to the scheduler, are the fan-out Gate ID, fan-out Pin ID, time offset from the current GVT, and the output value. These items are assembled into one piece of data and passed to the scheduler. The scheduler then assigns this future event data in the proper time slot and sends back to the engine in the form of Gate ID, Pin ID, and Value when GVT rolls into the appropriate future time.

5.3 Delay Simulation Architecture

The delay values of a *Future Event* are given by an SDF file for each gates used in a design. Pre-processing software will parse the delay information and organize it into

the Delay Memory. The problem is to determine the index of the delay memory for a given gate, if and only if the output value has changed.

The *base address* of the Delay Memory is known statically at the pre-processing software run time. However, there are two pieces of dynamic information that cannot be pre-determined until the logic evaluation is performed. The first is the output change information, which can only be known after the evaluation task is performed. The other is delay information, which is a function of (1) the output change, (2) the input that triggered the output to change, and (3) the values of the other inputs. The output change will determine the rise/fall time, the information on "which input" will determine the path, and the value of other inputs will determine the state. All of these pieces of information will be used for delay evaluation based on the delay type to be applied to the gate. The following sections will describe different delay memory architectures and discuss their strengths and weaknesses.

5.4 Fixed Delay Memory Architecture

A fixed delay can be expressed as a function: DELAY(*BaseAddress*, *Rise/Fall*). The Base Address can be determined at the pre-processing time for each gate, but the rise/fall information cannot be determined until the actual evaluation takes place and the output value is obtained and compared with its previous value. Therefore, the rise/fall is dynamic information. We can arrange this delay information on a linear memory space (1-dimensional memory) as shown in Figure 53. This linear array scheme can utilize the

memory with 100% efficiently, without any fragmentation, because the delay values are stacked up on top of each other and every gate is dependent on the output rise/fall.

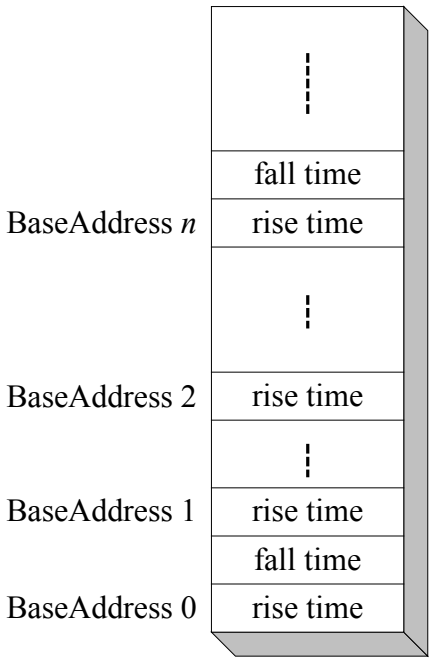


Figure 53 A Linear Array of Delay Memory

When the gate has either path dependent or state dependent delay, then computing the delay address becomes more costly because the dynamic information has to be added to the Base Address. On a linear array of memory, the delay values are stored as a stack of rise-time and fall-time pairs. Since the circuit design normally contains all three gate delay types, the delay address computation has to handle all three situations. As mentioned earlier, the base address is pre-determined by the pre-processing software; the problem now is to compute the offset amount from the base address.

A fixed delay is a function of base address and rise/fall time information. Path dependent delay is a function of address, rise/fall and input-to-output path information.

State dependent delay is a function of base address, rise/fall time, I/O-path, and the state of other output values.

1. Fixed Delay Address = (Base Address + rise/fall)
2. Path dependent Delay Address = (Base Address + path + rise/fall)
3. State dependent Daly Address = (Base Address + path + state + rise/fall)

We can generalize the delay computation with case 3 (state dependent) given above by setting the state to 0 for path dependent delay and both state and path to 0 for fixed delay. Figure 54 shows our generalized delay address computation scheme.

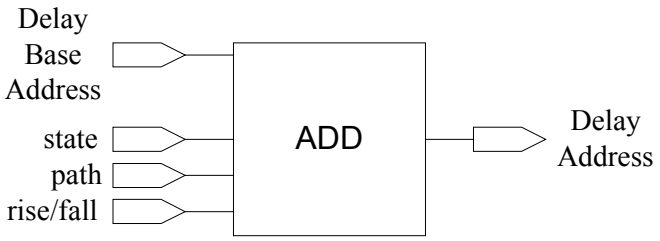


Figure 54 Delay Address Computation by Adding

The problem of this method is that it requires the *ADD* operation to determine the delay address. Performing these *ADD* operations can impact the performance of the system.

5.5 Path Dependent Delay Memory Architecture

A path dependent delay is expressed as a function is: DELAY(*BaseAddress*, *I/O-path*, *Rise/Fall*). I/O-path and rise/fall information is not known until the gate evaluation is finished. If we organize the memory into a 2-dimensional array using I/O-path and the

rise/fall on each axis, and treat that 2-D array as a page of memory (each page is identical in size), then the delay lookup becomes simple. Figure 55 shows this scheme. A fixed delay is shown in the bottom layer (with Base Address 0) of Figure 55. Base Address 1 and Base Address *n-1* in the figure denote the path dependent delays.

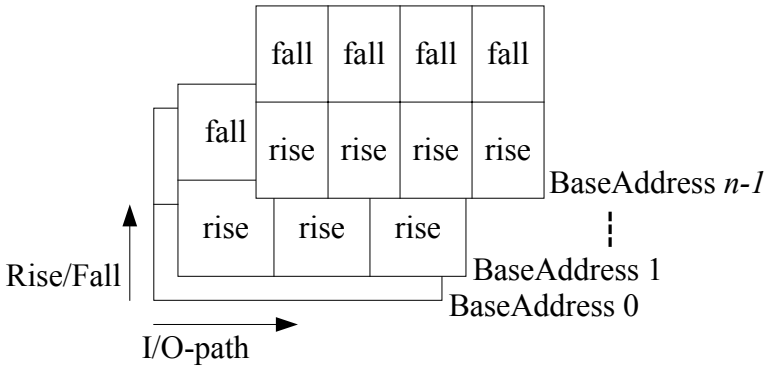


Figure 55 A 2-D Array Delay Memory

The problem with the 2-D array is that it can become fragmented and less efficient, because the page size of the 2-D array is fixed therefore the memory is not fully utilized.

In addition to that, if the gate has a state dependent delay property, the memory configuration has to be either multiple of 2-D arrays or use a large page size of single 2-D array. Either configuration forces the system to perform extra operations to obtain the address of the delay information.

5.6 State Dependent Delay Memory Architecture

A state dependent delay expressed as a function is: DELAY(*BaseAddress*, *I/O-path*, *State*, *Rise/Fall*). The dynamic information is therefore, I/O-path, state, and rise/fall information. The easiest memory organization would be to use a 3-dimemsional memory for each Base Address by using I/O-path, state, and rise/fall information on each axis. However, this 3-dimensional memory configuration will waste even more memory, especially when all of three delay types are mixed in the design, the delay memory will be severely fragmented. Memory is wasted when we are dealing with fixed delay cells or path dependent cells, the state and path axes are not used for the fixed delay and the state axis is not used for the path dependent delay case.

5.7 Generic Delay Memory Architecture

To overcome the extra operation overhead discussed earlier, the pre-processing software can organize the widths of base address, state and path information so that the delay address can be obtained by simply combining the pre-arranged information.

If we allow the width of the base address as a *variable*, then we can assemble the delay address without having to perform the *ADD* operation. Since the information for base address, the width of I/O-path, and the number of states can be pre-processed, software can perform this width computation for the base address, path, and state information fields.

The path information encoding requires *ceiling(log_2(N))* bits, and state information encoding requires *(N-1)* bits, where *N* is the number of inputs. Then, we can segment the linear memory into a formulated size of *ceiling(log_2(N))+(N-1)*, when we are dealing with the state dependent delay gates. Then the software arranges the base address to point to each segments and clears the lower bits so that base address can be computed with the rest of the information. Now, rather than using adders, we can compute the delay memory address with a simple bit-wise OR operation, as shown in Figure 56, for state dependent delay model.

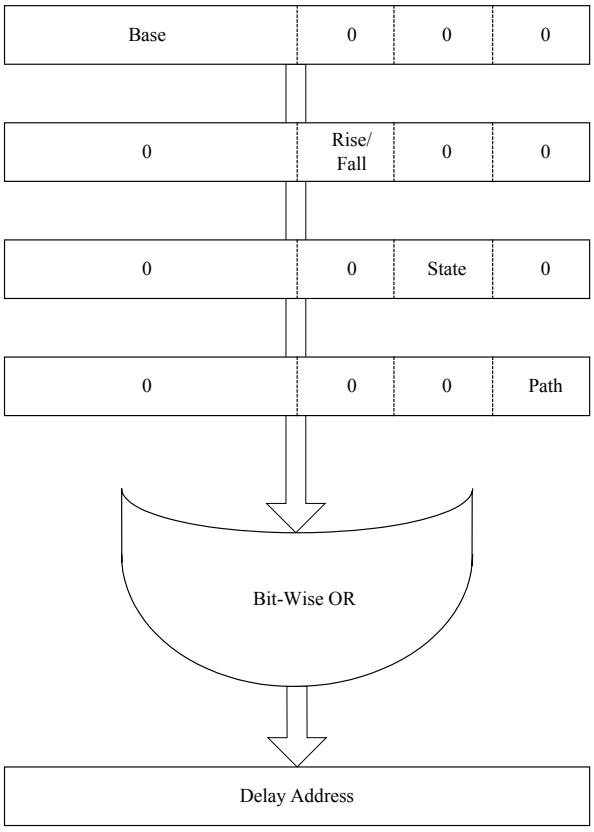


Figure 56 Bit-Wise OR to Compute Delay Address for State Dependent Delay

I/O-path dependent delay can be arranged the same way as state dependent delay except the state information is not used. Fixed delay only require one extra bit for rise/fall information.

As mentioned previously, the total number of delay values is given by $N \times 2^N$, where N is the total number of inputs (equivalent to number of I/O-paths) and the second term is the total number of states per I/O-path. It is obvious that the second term 2^N will be much larger than the number of inputs N. The number of states is always given as a power of 2, but total number of inputs are not so. This indicates that if we order the path variable in front of the state variable in the address field, we will have much larger fragmentation. Therefore, we are placing the path variable behind the state variable. Figure 57 (b) and (c) show the location of the "Path" variable in the delay address map.

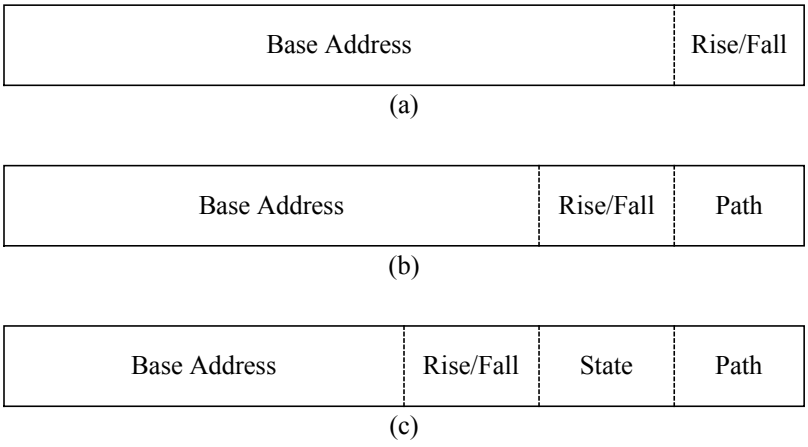


Figure 57 Delay Address Map (a) Fixed Delay, (b) Path Dependent, (c) State Dependent

As an example, assume that we are given a 4-input gate and the delay address space is 1M (20 bits). If the gate is an AND gate with a *fixed delay*, then the software generates a 19-bit *base address* with the *rise/fall* bit cleared. When the rise/fall

information becomes available after the evaluation of the gate, the base address and the rise/fall information are bit-wise OR'ed to form a delay address.

If the given gate is XOR with a *state dependent delay*, the software generates 14-bit *base address* with rise/fall bit, *path* bits and *state* bits cleared. The *path* variable can be determined based on input PinID and *state* can be determined based on the value of other inputs. Again, when the rise/fall information is available, all four variables are bit-wise OR'ed to generate delay address.

If the gate contains *path dependent delay*, software generates 17-bit base address with *rise/fall* bit and *path* bits cleared. After the gate's output is computed, the base address, path and rise/fall variables are bit-wise OR'ed to generate delay address. Table 36 summarizes the bit widths for each delay case.

Table 36 4-Input Gate with Various Delay Types

	size (bit)	Fixed delay	Path dep.	State dep.
Base Address	14 to 19	19	17	14
State	0 or 3	0	0	3
Path	0 or 2	0	2	2
rise/fall	1	1	1	1

Also, if we place the delay information in any order (any mix of fixed, path and state dependent delay) then the memory can be further fragmented. This is because we require each base address to start at a power of 2 for path and state dependent delays.

To avoid this fragmentation, the pre-processing software will start placing fixed delay information in the bottom of the memory. Then the path dependent and the state dependent delay information will be placed on top of the fixed delay information. Figure

58 shows this arrangement. The splitting line between the fixed-delay area and path and state dependent area is the nearest address to the power of 2 after the fixed delay information area.

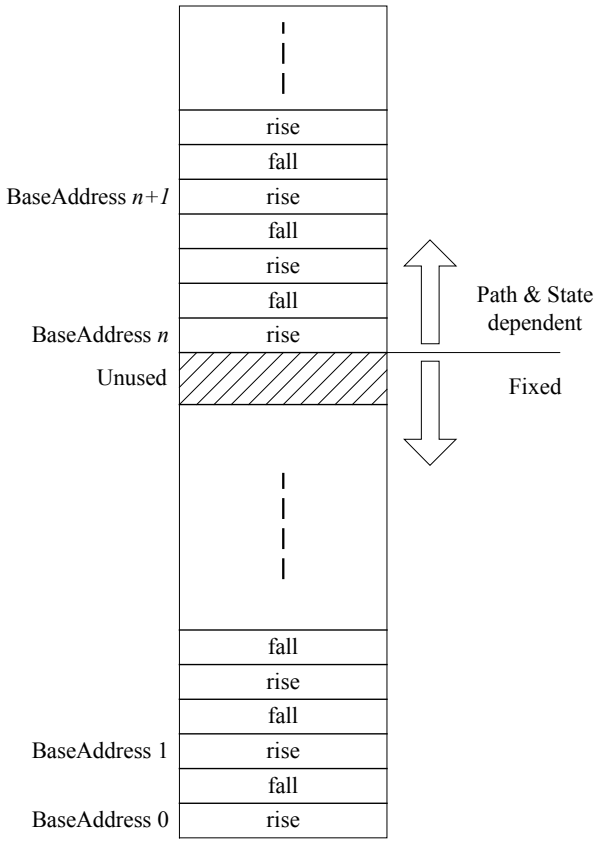


Figure 58 Delay Memory Map

5.8 Delay Architecture Conclusion

Wasted memory cannot be tolerated because it reduces the capacity of the system. Therefore, the only viable solution is to rely on a linear array and to design a memory layout with the least amount of wasted memory space. Delay lookup is not just a simple

memory lookup because there are 3 different types of delays stored in a single memory. Due to this mixture of different delay types, computing the delay address requires some operations. State dependent delay will limit the capacity of delay memory, which in turn will limit the capacity of the entire architecture. Therefore, we have decided to choose a normal linear array with the *ADD* operation. Although the segmented linear memory idea seems appealing, it still fragments the memory. The trade off here is the performance versus space. Space is chosen over performance to accommodate the growing size of modern digital design. The performance penalty of using a four input adder is expected to be minimal and could be pipelined to further minimize the performance cost.

6.0 SCHEDULER ARCHITECTURE

When new events (future events) are generated as a result of logic evaluation activity, the *scheduler* manages these new events according to their time stamp. This is to ensure that the execution order does not violate the *causality constraint* discussed in Chapter 2. The problem of the scheduler task is that new events have to be ordered according to their time stamp, and this ordering of events consumes a large number of cycles. As was shown in Figure 6, the *scheduler* takes up a major portion of run time for the logic simulation software. This is because the nature of scheduling involves a large amount of memory activity (*i.e.* searching for the events with the smallest simulation time stamp), and this memory activity causes a bottleneck in the scheduling task.

Researchers were able to solve this problem by employing the event wheel [7] algorithm, which removes the need for sorting. It was successful in zero-delay or unit-delay model simulations, where a relatively small event wheel can handle the scheduling task. However, in hardware/software co-simulation environment, where the granularity of delay timing changes from pico-seconds to milli-seconds, the size of the event wheel has to be increased to handle large amount of timing grains. In such cases, the event wheel cannot perform efficiently, as was discussed in Section 2.2.4 , because most hardware simulation events use fine timing grains such as pico- to nano-seconds but software simulation events will use coarse grains such as micro- to milli-seconds.

Therefore, the event wheel becomes sparsely populated. Three architectures for the scheduler task, which can be used in co-simulation environment, will be explored and

discussed in this chapter. Section 6.1, discusses a plain linear scanning architecture. In Section 6.2, we explore parallel linear scanning by dividing memory space into sub-sections. In section 6.3, we examine parallel linear scanning with a binary tree by replacing a global minimum search circuit with a combinational binary tree configuration.

6.1 Linear Scanning

One alternative is to store the future event in linear memory and perform a sort operation. This approach can handle large time grains, but requires a large number of CPU cycles as a trade off, as was shown in Chapter 2 by our quick sort software scheduler benchmark results. The performance of the sorting algorithm on a workstation is well defined. The bubble sort and insertion sort algorithms run with $O(N^2)$ [33], quick sort and merge sort requires $O(N \times log(N))$ [33] for the same task. Each time the scheduler increments the simulation time (GVT), the scheduler must perform a sort operation with both new and existing events. Performing a sort operation simply requires too many CPU cycles. Even after the sort operation, only the events with the smallest time stamp are passed to the logic evaluation block, and when the GVT increments, remaining events must be sorted again.

Instead of sorting inefficiently, the scheduler can perform a search (scan) operation, continuously searching for events with minimum time stamp in the memory. The scan operation requires $O(N)$ cycles, given N elements in memory. When we

perform the scan operation, regardless of GVT, the event with minimum GVT is searched and passed to evaluation block. We can formulate the performance of scanning as: $C \times O(N) \approx O(N)$, where C is the number of events with the same minimum time stamp.

Therefore, in comparison, the sort algorithm can produce multiple events with the same time stamp in one sort operation with $O(N \times log(N))$ cycles, but scanning has to run $O(N)$ cycles to find the first event with smallest time stamp and then another $O(N)$ to find all other events with the same time stamp. Thus memory scanning is faster than memory sorting, since $O(N) < O(N \times log(N))$.

A dedicated hardware can be designed to scan the memory at peak speed. Figure 59 shows a linear scanning scheme.

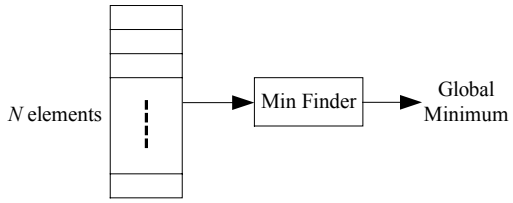


Figure 59 A Linear Memory Scanning

6.2 Parallel Linear Scanning

To speed up the scheduling task and maintain large timing grain, the new architecture with parallel linear memory sub-scanning has been considered to utilize the concurrency in memory space search. Figure 60 illustrates this architecture. We divide the memory of size N into p segments, each with a depth of k (= N/p) and attach a *minimum finder* circuit to each memory segment. Each segment will require k cycles to

find a local minimum in *parallel*, and all of the resulting p local minimums will be searched again for a global minimum, consuming another p cycles. The run time for the architecture will be $O(k)+O(p)$. Our hardware uses p pieces of k $(=N/p)$ deep memory, one p:1 multiplexer, and (p+1) minimum finder circuits. The size of the design can be computed as:

Size = (p+1) × sizeof(min finder) + sizeof(p:1 Multiplexer).

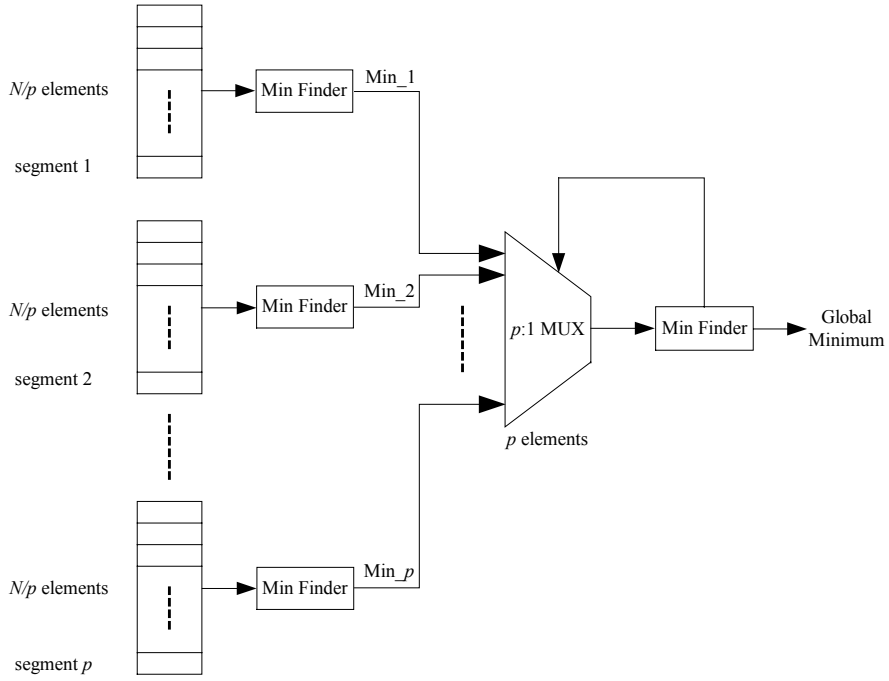


Figure 60 An Architecture for Parallel Linear SubScanning

Table 37 shows the resource usage of different sized multiplexers as a size measure. Synthesis result by Quartus-II reports that our min finder design uses 301 (out of 8,320 possible resources) Logic Elements (LEs).

Table 37 Resource Usage for Multiplexer Component Using 16-Bit Words

	Logic Elements
2:1 MUX	16
4:1 MUX	32
8:1 MUX	87
16:1 MUX	153
32:1 MUX	340
64:1 MUX	666
100:1 MUX	1,040
128:1 MUX	1,332
256:1 MUX	2,663
512:1 MUX	5,327
1024:1 MUX	10,653

Unlike software sorting or scanning, which is limited by the memory architecture (single, narrow, sequential), our design can perform the task concurrently using multiple small segments of memory. The design is composed of a simple minimum search circuit (controller and comparator) attached to the memory. And this simple design is repeated (p + 1) times (p for local min, 1 for global min).

6.3 Parallel Linear Scanning with Binary Tree

We can further improve the previous design by replacing the global minimum finder with a group of comparators and multiplexers in a binary tree configuration as shown in Figure 61. In this design, the local minimum searching is identical to the previous design, but we use (p–1) comparators and multiplexers in the form of binary tree, thereby removing the last stage (the global scanning mechanism). Instead, the global minimum searching is performed by a purely combinational circuit, which can perform its minimum searching task in a few cycles, yielding the performance of O(k),

where $k = N/p$. Each circle in the Figure 61 represents a comparator and a 2-to-1 multiplexer, as shown in Figure 62.

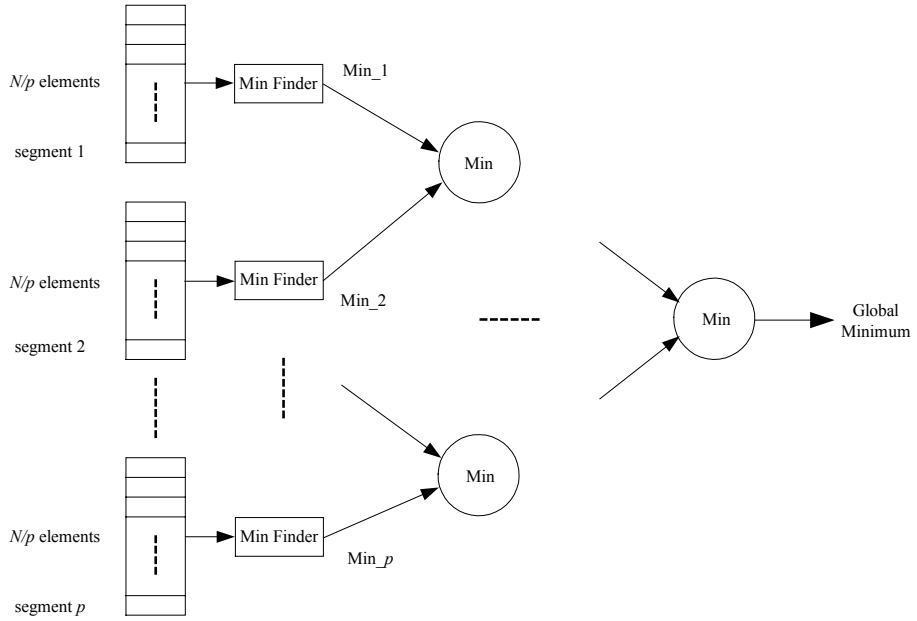


Figure 61 Comparator and Multiplexer in a Binary Tree

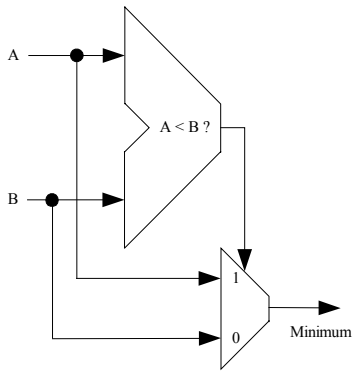


Figure 62 Comparator and Multiplexer for Finding Minimum

The problem with this design is its resource usage. The number of resources required grows exponentially as the depth of binary tree grows. When the binary tree

depth reaches 9, it becomes larger than our target device. Therefore, the design shown in Figure 61 will perform faster, but, as a trade off, it becomes impractical for a large sized binary tree. Table 38 and Figure 63 illustrate this problem. In Table 38, Altera's EP20KE200 device (total resource is 8,320) was chosen as a target platform. When the number of input reaches to 256, the binary tree approach nearly depletes the resource. If the number of input is 512 or more, the design does not fit into a single chip.

Table 38 Quartus-II Synthesis Report for Resource Usage of Binary Tree

Number of Inputs	Number of Resources	Resource Usage Altera EP20KE200
2	32	0.38%
4	96	1.15%
8	224	2.69%
16	480	5.77%
32	992	11.92%
64	2016	24.23%
128	4064	48.85%
256	**8160**	**98.08%**
512	16352	196.54%
1024	32736	393.46%

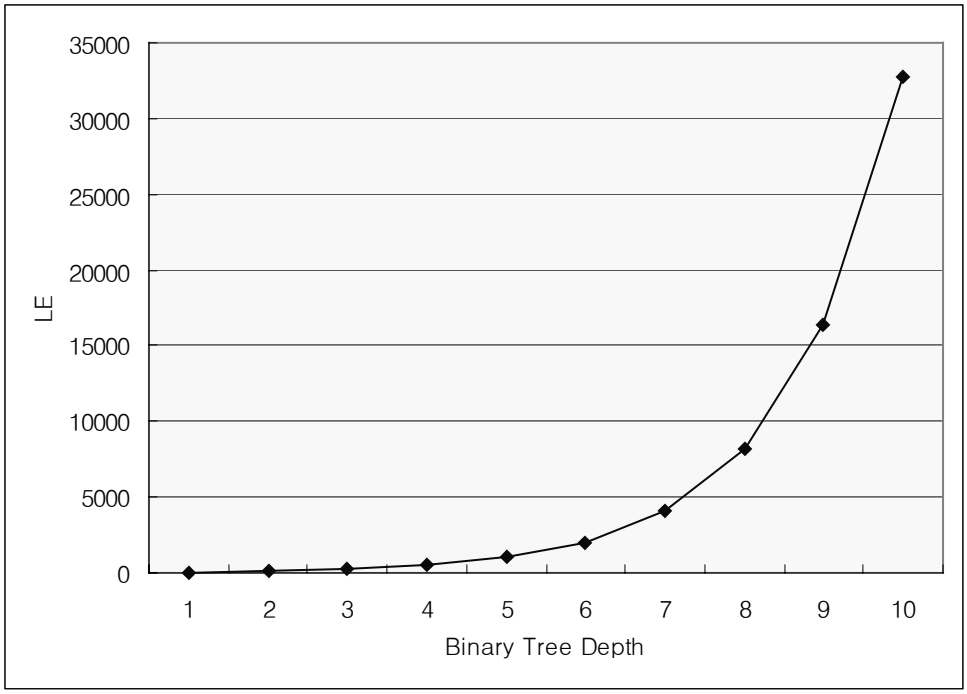


Figure 63 Resource Growth Rate for Binary Tree

6.4 Summary

We choose the normal parallel linear scanning scheme (shown in Figure 60) for our scheduler, since the binary tree approach will require a nearly impossible amount of resources for a large sized event queue. Our design will still out-perform the software approach by utilizing concurrency in the minimum value search algorithm. Table 39 summarizes the architectures we have explored in this chapter. A simple linear scanning consumes the least amount of resources while it requires a large number of cycles to find a minimum value in the memory. Parallel linear scanning with binary tree will perform

faster than any scheme given in the table, but it consumes too much resource. Normal parallel linear scanning optimizes the performance and cost.

Table 39 Size and Performance Comparison between Scheduler Algorithm

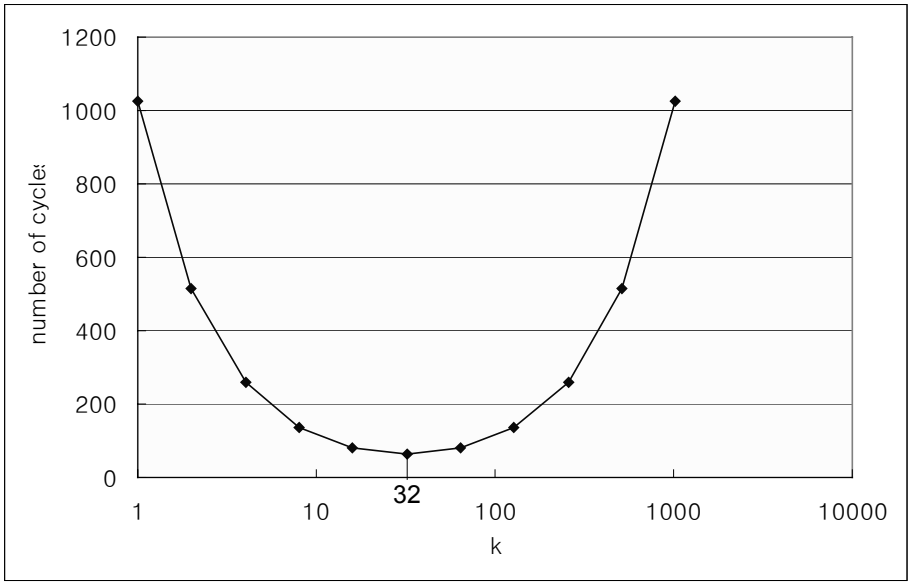
	Linear Scanning	Parallel Linear Scanning	Parallel Linear Scanning with Binary Tree
Size	sizeof(min finder)	(p+1)*sizeof(min finder) + sizeof(p:1 MUX)	p*sizeof(min finder) + sizeof((p-1) binary tree cells)
Speed	*O(N)*	*O(k) + O(p)*	*O(k)*

The performance of our design depends on the size of the segment, k and the total number of segments p. If $k >> p$, then the performance depends on $O(k)$, *i.e.* the local minimum search will dominate the overall run time. In general, if $k >> p$, then it can be written as $O(k) + O(p) \approx O(k)$. The software scanning algorithm performs $O(N)$, therefore, we have a speedup of $O(N)/O(k) = O(N)/O(N/p) \approx p$

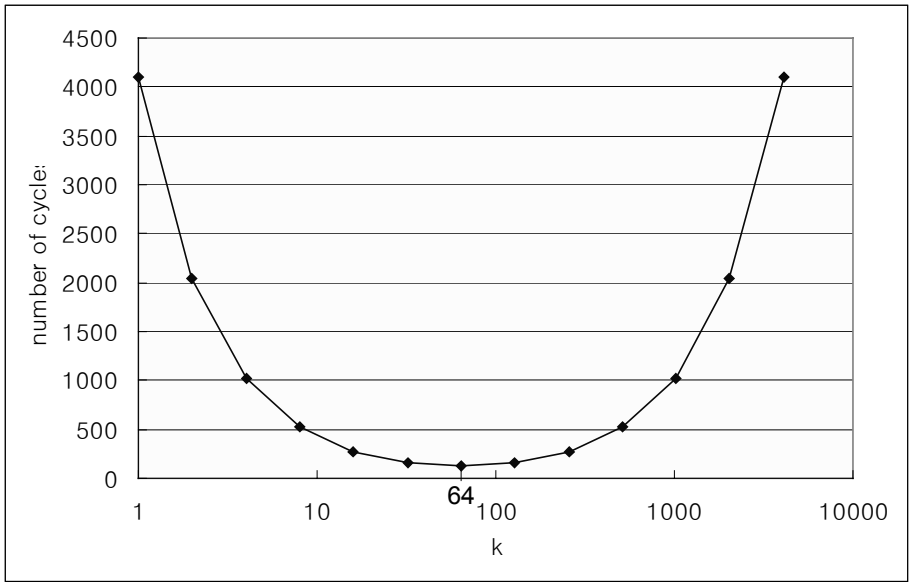
On the other hand, if $k << p$, then $O(p)$ dominates the overall performance (global minimum search will dictate the run time). The overall performance can be expressed as $O(k) + O(p) \approx O(p)$. And the speedup will be:

$$O(N)/O(p) = O(N)/O(N/k) \approx k$$

If $p = k$, then the performance will be $2 \times O(k) \approx O(k)$; this will be the optimal performance for our linear scanning scheduler. Figure 64 (a) and (b) shows the performance graph in terms of k and number of CPU cycles when $N = 1024$ and $N = 4096$, respectively. The peak performance point is when $p = k$, with $k = \sqrt{N}$, because, $k = N/p$ and $k^2 = N$.



(a)



(b)

Figure 64 Performance Graph (a) N = 1024; (b) N = 4096

In practice, since our target capacity is 100,000 gate design, a scheduler would have approximately 10,000 events pending if 10% of the circuitry is active at any given

time. Thus, if we compute optimal p and k for N = 10,000 then, $p = k = \sqrt{N} = 100$. From Table 39, the size of the design is computed as (p+1) × sizeof(min_finder) + sizeof(p:1 MUX). Since p = 100 and size of minimum finder circuit is 301 (given in Section 6.2), we have (100+1) × 301 + 1,040 = 31,441 Logic Elements. The speed of our scheduler would be approximately 200 cycles ($O(k) + O(p)$).

7.0 EXPERIMENTAL RESULTS AND PROTOTYPE

In this chapter, we will describe how all the materials we have discussed in previous chapters fit together as a logic simulation system. A simple test circuit is presented in Section 7.1 as a proof-of-concept to demonstrate that our design performs as we expected.

In Section 7.2 implements this circuit and shows all of the functional blocks described from Chapter 4 to Chapter 6 (logic evaluation, delay architecture, and the scheduler blocks) are put together to form a logic simulation hardware system. The test circuit is parsed through the pre-processing software, which creates the memory image of the net-list and the delay values. These values are then loaded into the design. Initial input events are loaded into the scheduler. As the simulator is started, the logic evaluation block and the scheduler block process the data according to the event order.

Section 7.3 discusses scalability. The proof-of-concept prototype has a very limited gate capacity due to the limitation of the FPGA used. When we expand our design to a 100,000 gate capacity, the memory depth will be significantly increased to accommodate the capacity growth and require off-chip memory. However, the width of memory address signal grows using a log scale, and most of the items such as "list of input" and "list of output" remain unchanged. Since our design references the memory in a single access, this growth of width will not affect the performance. The performance of the prototype is measured and extrapolated for a 100,000 gate capcity design.

The performance and feature of our design are then compared with the existing hardware and software logic simulators in Section 7.4. We will show that our design will out-perform existing logic simulators of comparable timing accuracy. Additionally, there are features in our design that are not found in other simulators.

In order to make a fair comparison between our architecture and others, we have quantified our experimental results in terms of FPGA Logic Elements. These Logic Elements (LE's) are created using a 4-input lookup table, a flip-flop, and a number of other AND and OR gates that are used to interconnect the LE with other LE's. As an approximation, a single FPGA LE can be implemented in an ASIC using 1-10 standard cell gates.

7.1 Prototype

To prove the design concept, we made a small circuit to test various types of universal gate components. Figure 65 shows this circuit, a parity checker and a D-type Flip-Flop. One of the *XOR* gates was intentionally "unwrapped" into *AND*, *OR*, and *INVERTER* cells to demonstrate our universal gate, and cause enough event propagation throughout the circuit to exercise our design. The circuit was kept small so that the entire hardware simulator could be tested inside a single FPGA so that the event execution order could be visually verified. Scalability to a 100,000 gate design is also addressed in this chapter.

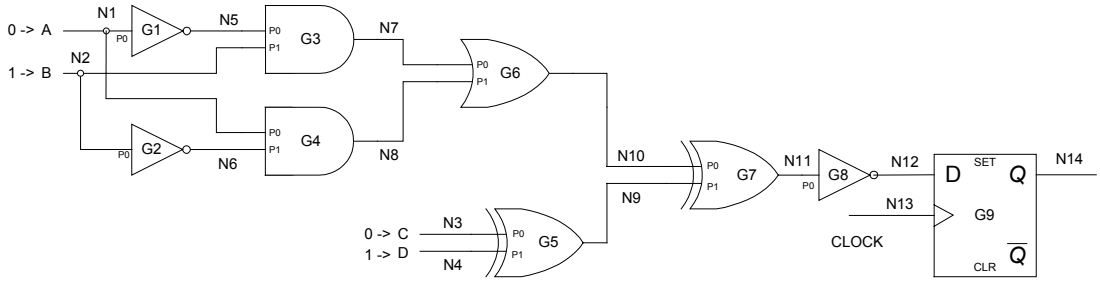


Figure 65 Parity Checker Test Circuit

In the prototype shown in Figure 65, we assume that all of the initial input events are started at the same simulation time (GVT). *G1* through *G9* represent gates, *N1* to *N14* represent wire, P_0 and P_1 represent input pins. And *A*, *B*, *C*, *D* are primary inputs to the circuit. Since the input signal *A* is connected to *G1* and *G3*, input signal *B* is connected to *G2* and *G4*, and the inputs *C* and *D* are driving *G5*, there are total of six initial events, as listed in Table 40.

Table 40 Initial Events

Event Number	Gate ID	Pin ID	Value	GVT
1	1	0	0	1
2	4	0	0	1
3	2	0	1	1
4	3	1	1	1
5	5	0	0	1
6	5	1	1	1

Table 41 shows the order of the simulation event flow when our design simulates the test circuit. All of the initial events are assumed to be started at the same time, with a simulation time (GVT) of 1. All of the inputs and outputs of the gates in the test circuit are initialized to the unknown ('X') state. Notice that in events 4, 5, 7, 8, 10, the output does not change, and therefore a future event is not generated. The event number 15

represents when the clock input signal became a '1', therefore our flip-flop design has detected a rising edge and latched the data input stored at event 14.

Table 41 Simulation Event Flow of Prototype Running Test Circuit Simulation

Event Number	Function	Current GVT	GATE Number	PIN Number	Input Value	Current Output	New Output	Destination Gate ID	Destination Pin ID	Delay	Future GVT
1	INV	1	1	0	0	X	1	3	0	10	11
2	AND	1	4	0	0	X	0	6	1	9	10
3	INV	1	2	0	1	X	0	4	1	7	8
4	AND	1	3	1	1	X	X	3	1	-	-
5	XOR	1	5	0	0	X	X	7	1	-	-
6	XOR	1	5	1	1	X	1	7	1	10	11
7	AND	8	4	1	0	0	0	6	1	-	-
8	OR	10	6	1	0	X	X	7	0	-	-
9	AND	11	3	0	1	X	1	6	0	12	23
10	XOR	11	7	1	1	X	X	8	0	-	-
11	OR	23	6	0	1	X	1	7	0	8	31
12	XOR	31	7	0	1	X	0	8	0	9	40
13	INV	40	8	0	0	X	1	9	0	9	49
14	FF	49	9	0	1	0	0	-	-	-	-
15	FF	100	9	1	1	0	1	-	-	11	-

7.2 Overall Architecture

All of the design units described in previous chapters are put together as a system as shown in Figure 66. The design contains a *logic evaluation block*, *input assembler*, *delay address computation block*, *output comparator*, *future event generator*, *scheduler*, *pending event queue*, *future event queue*, *net-list and configuration memory*, and *delay memory*. The functionality of each blocks are:

- Logic evaluation block: computes the output of a gate with a given input vector.
- Input assembler: generates the input vector of a gate before evaluation.

- Output compare block: compares newly computed output and existing output. If they are different, raises output change flag and computes whether the new output has risen or fallen.
- Delay address computation block: computes the address with rise/fall value and gate's delay type given by function group variable.
- Future event generator: creates the future event based on delay value acquired and the fan-out information of a gate.
- Scheduler: orders incoming future events according to the simulation time (GVT).
- Pending event queue: stores events to be processed by the logic evaluation block.
- Pending Event: each pending event is comprised of Gate ID, Pin ID, and input value.
- Future event queue: stores new events generated due to the logic evaluation.
- Future Event: future event is comprised of destination Gate ID, destination Pin ID, delay value, and the signal value.
- Net-list and configuration memory: stores net-list information such as list of input values, list of output values, fan-out information. Also stores the configuration information such as mask values, input and output inversion flag, and function group variable, *etc*.
- Delay memory: stores rise and fall time values for each gates in the design.

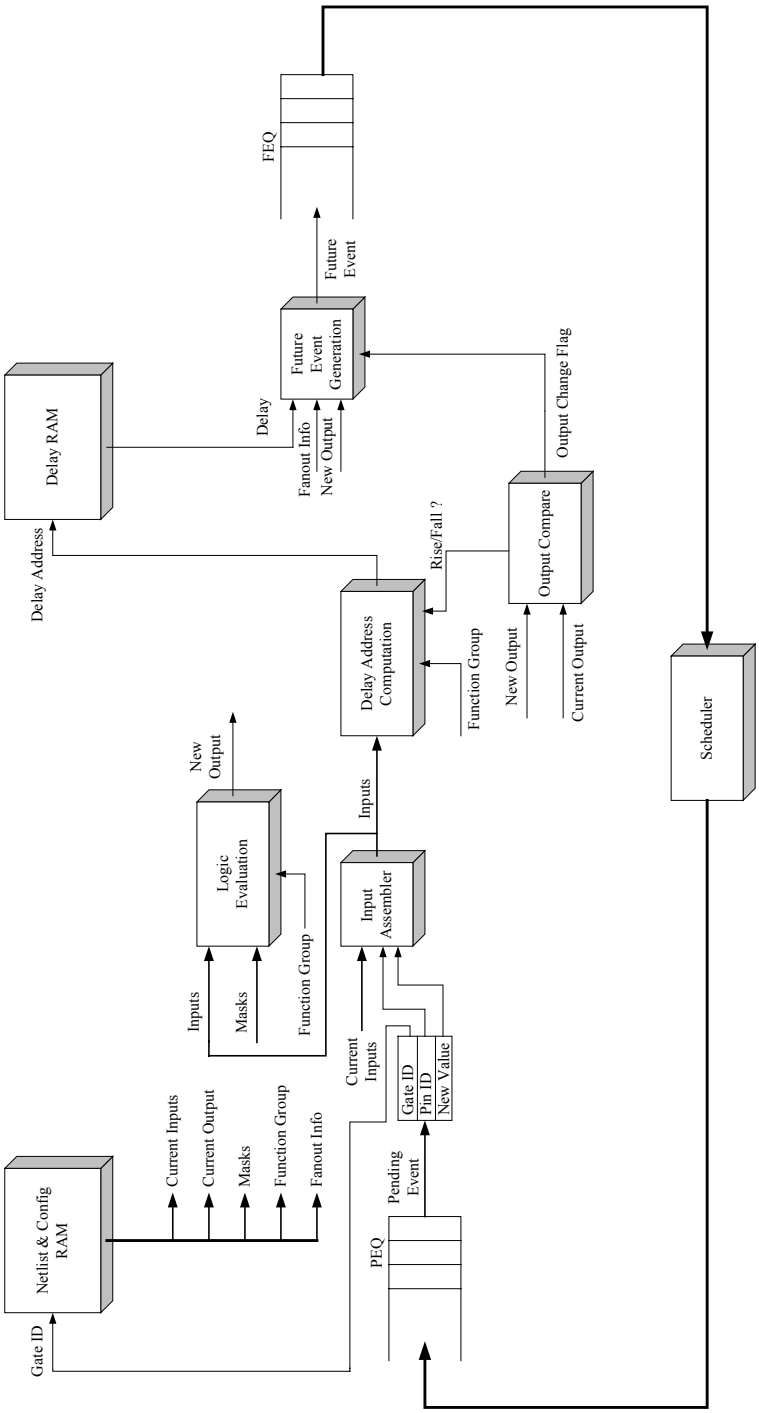


Figure 66 System Architecture for Logic Simulation Engine

The data flow of the logic simulation algorithm is summarized below:

1. A pending event is read from pending event queue.
2. Reads the net-list and configuration memory using the Gate ID, given by pending event, as an address.
3. Input vector is assembled using the value and Pin ID (from pending event) and the current input (from net-list memory).
4. Logic evaluation is performed and new output value is computed.
5. New out and current out is compared. If they are different, output change flag is set and rise/fall information is acquired.
6. If output change did not occur, go to step 1.
7. If output change has occurred, delay memory address is computed and delay memory is referenced.
8. Future event is generated with new output value, fan-out information, delay value.
9. Future event is sent to future event queue
10. Scheduler is continuously searching for the events with minimum time stamp.
11. When all of the pending events with current simulation time stamp (GVT) have been processed, the scheduler advances GVT and sends new set of pending events to pending event queue.

7.2.1 Net-list and Configuration Memory and Delay Memory

The width of the net-list and configuration memory is shown in Table 42. For the prototype, we limited the maximum number of inputs to four and the maximum number of outputs and fan-outs to two.

Table 42 Data Structure of Net-list and Configuration Memory

Data Item	Bits	Comments
Delay Base Address	4	16 delay location
Power Count	4	counts upto 16
Number of Inputs	2	4 input maximum
List of Input Value	8	2 bits/input, 4 inputs
L1 Input Invert	4	1 bit per input
Number of Output	1	2 outputs
List of Output Value	4	2 bits/output, 2 outputs
Output Invert	2	1 bit per output
Mask1	2	mask for 0, 1, Z, X
Mask2	2	mask for 0, 1, Z, X
Function Group	3	8 different functions
Fan-out Information	12	(4 bit GateID + 2 bit PinID) * 2 fan-outs
TOTAL	48	

The delay value is stored in a simple linear memory, as was discussed in Chapter 5. For simplicity, only 8 bits were used for prototype design.

7.2.2 Logic Evaluation Block

The logic evaluation block, shown in Figure 67, contains all of the universal gate primitives described in Chapter 4. The input signals are organized by the input assembler block, and fed into the logic evaluation block. All of the primitives receives this input signal and produces the output. The output of each primitive is connected to the

multiplexer, and the proper output value is selected by the "Function Group" parameter associated with that gate. This function group parameter is stored in the net-list and configuration memory. The result of logic evaluation will determine the output change, delay address computation, and future event generation tasks.

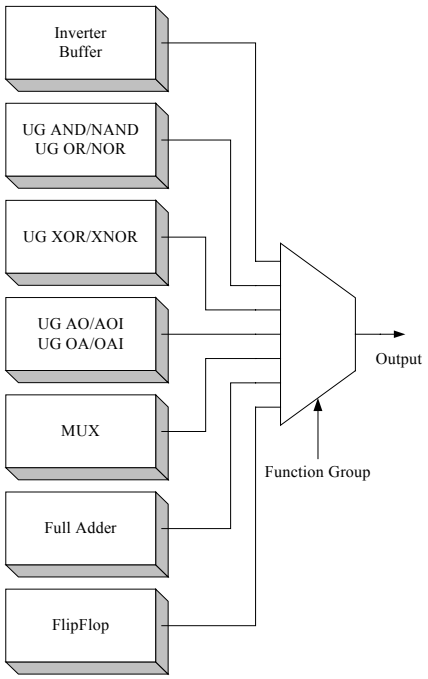


Figure 67 Logic Evaluation Block

7.2.3 Pending Event Queue and Future Event Queue

The logic evaluation block and the scheduler communicate through the queue structure. The scheduler sends a pending event, which contains a Gate ID, Pin ID, and Value. The evaluation block then sends a future event, which consists of Delay, Destination Gate ID, Destination Gate's Pin ID, and the Value.

Table 43 illustrates the data structure of the Pending Event Queue used for the prototype. The Pending Event is sent from the scheduler to the logic evaluation engine, which conveys the message "which pin of what gate has the change of value event" at current simulation time (GVT).

Table 43 Pending Event Queue Structure

Item	Bits	Comments
Gate ID	4	16 gates capacity
Pin ID	2	4 input pins per gate
Value	2	4-level signal strength value

Table 44 shows the data structure of the Future Event Queue for the prototype. The Future Event is sent from the evaluation engine to the scheduler with "which pin of what gate *will* have a value change event at Time Offset from the current GVT".

Table 44 Future Event Queue Structure

Item	Bits	Comments
Delay Time Offset	4	gate delay limited to 16 delay units
Destination Gate ID	4	16 gates capacity
Destination Gate's Pin ID	2	4 input pins per gate
Value	2	4-level signal strength value

7.2.4 Delay Address Computation Block

When the scheduler sends a Pending Event to the Logic Engine, the net-list information is read in from the memory location specified by its Gate ID. The input signals are assembled and fed into the evaluation block, and the new output is computed. The new output and current output are then compared to acquire the information about whether or not they are different. If they are different, the output change flag goes high.

Also, if the output has changed, the output rise/fall information is obtained at the same time. If the output has not changed (output change flag becomes low), then no future event is generated and the input and output values are updated in the net-list memory. Detailed description about delay memory address computation was presented in Chapter 5.

7.2.5 Performance of Prototype

Our prototype was implemented on Altera's EP20K200EFC672-1X FPGA. The compilation report states that the design runs at 39.8 MHz, consuming 1054 Logical Elements and 952 Embedded System Blocks. This represents 12% of the FPGA logic elements and 1% of the FPGA internal RAM. shows the simulation results of the prototype design. The "Gate ID" is highlighted to compare with Table 41. We can see that the gates in the design are being processed in exactly the same order as the table.

Evaluation of a pending event takes 8 cycles before a future event is generated. The scheduler can provide new event in every 36 cycles in the worst case. The worst case happens when the scheduler memory is empty and just received a new future event. The scheduler will go through all of the empty location to find a minimum time stamp and pick up the event just received. Therefore the worst-case performance of our design is 44 cycles per event.

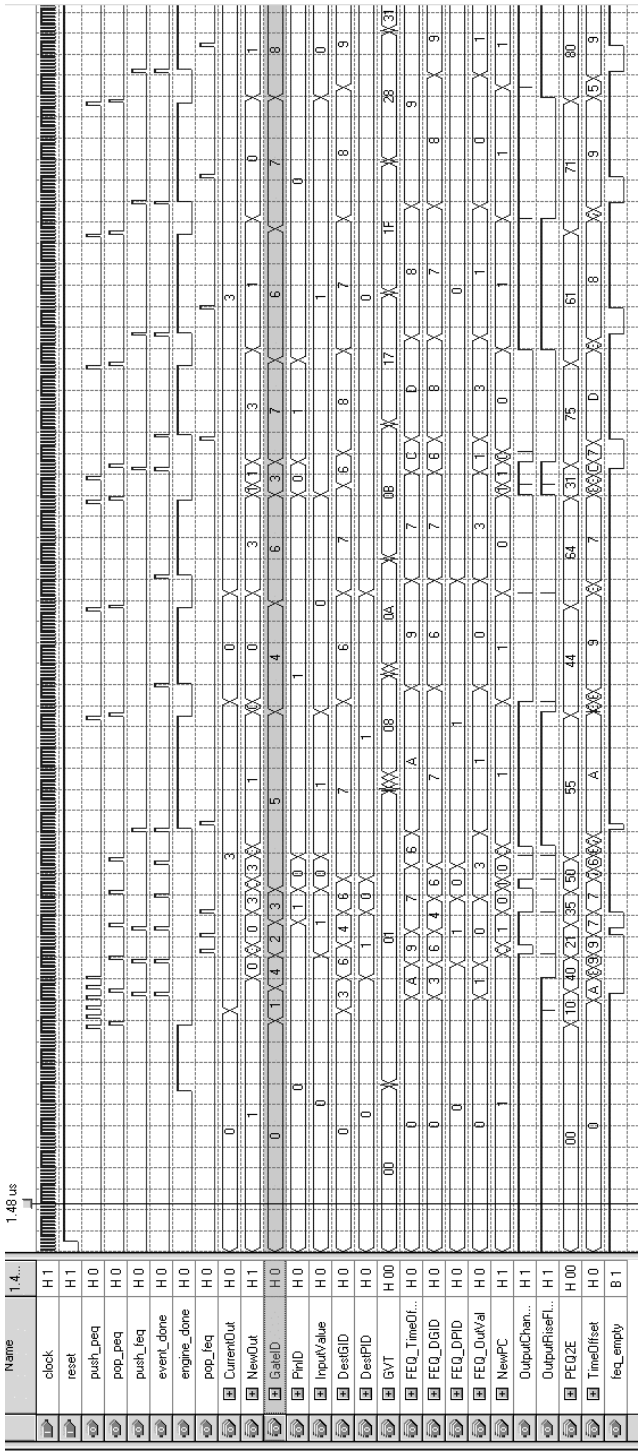


Figure 68 Simulation Waveform for Prototype Circuit

7.2.6 Pre-processing Software and Data Structure

The pre-processing software parses the verilog file for net-list information and the SDF file for delay information. The parser software reads the net-list input file (verilog file) and organizes the connectivity information by matching the output name of a gate to the input name, which the current gate drives (cross-linking). It then looks up the delay information file (SDF file) to assign the delay values to each gate.

The software then generates the initial memory map of the hardware according to the input files. The data structure of the software follows the memory architecture of the hardware so that the output of the software can be directly loaded into the memory of the hardware design. Figure 69 shows the data structure used by the hardware and parser software.

```
struct {
   unsigned int Delay_Base_Address;
   unsigned int Power_Count;
   unsigned int Number_of_Inputs;
   unsigned int List_of_Input_Values[i];
   unsigned int Level_1_Input_Inversion_Flag[l];
   unsigned int Level_2_Input_Inversion_Flag[l];
   unsigned int Number_of_Outputs;
   unsigned int List_of_Output_Values[o];
   unsigned int Output_Inversion_Flag;
   unsigned int List_of_Dest_GateID[n];
   unsigned int List_of_Dest_PinID[n]
   unsigned int Mask1;
   unsigned int Mask2;
   unsigned int FunctionGroup;
};
```

Figure 69 Data structure for Hardware and Software

Delay values are organized as rise/fall time pairs and stacked up in the Delay memory, as described in Chapter 5. Most of the gates in the library only use one rise/fall time pair as their delay information (fixed delay), but some gates such as XORs contain multiple entries of rise/fall pairs since they exhibit state dependent I/O-path delays. Multiple delay entries are also linearly stacked in the delay memory, and the SDF parsing software provides the pre-computed starting address of the delay memory (Delay Base Address).

7.3 Scalability of the Architecture

There are two aspects of the scalability. One is the scalability of the gate model, which addresses the gate size. The other is system scalability, which determines the capacity of the logic simulation system.

Our components are pre-scaled to support up to 8 inputs for single level logic gates, and up to 64 inputs for two level logic cells such as AO/OA gates. Table 45 shows the resource usage report by the Quartus-II compiler. Speed was measured by inserting registers on input and output ports (register to register delay) so that the IO port delay does not hinder the performance of the primitives. Every component runs at 10ns or faster, except the UG_AoOa8x8 primitive. This is because our UG_AoOa8x8 primitive is implemented as 2-level logic.

Notice that our prototype design runs at 39.8 MHz (shown in Section 7.2.2). This is a normal phenomenon for chip design due to the place and route process. This

becomes more severe when we use an FPGA as target platform, because unlike ASIC designs, where the transistors can be physically placed and resized, FPGAs only deal with assigning the functionality to the existing hardware resources (Logic Elements), and connecting these hardware resources. The numbers shown in Table 45 can be changed when different chips with different technologies are used and will not be discussed any further.

Table 45 Resource Usage and Speed for Logic Primitives

Altera's EP20K200EFC672-1X		
Primitive	LE	Speed
Univ_AndOr8	145	112.98MHz
Univ_AoOa8x8	686	77.17MHz
Univ_XOR8	143	138.29MHz
MUX41	95	103.98MHz
FA	48	117.08MHz
D-FF	21	161.34MHz

Table 46 shows the width of each component when our design scales to a 100,000-gate capacity. The assumptions made are:

- Maximum number of inputs for 1-level logic cells: 8
- Maximum number of inputs for 2-level logic cells: $8 \times 8 = 64$
- Maximum number of outputs for any cells: 2
- Average number of fan-out per gate: 5
- Average number of rise/fall delay value pair per gate: 5
- Total delay memory space: 500,000 (rise/fall time pair)

Based on above assumptions, the items in Table 46 were computed. For example, the width of "Gate ID" is computed as *ceiling*(log_2(100,000)) = 17, therefore the gate address space is 17 bits wide. To support up to 64 inputs for 2-level gates, 6-bits (*ceiling*(log_2(64)) = 6) are required to encode the input Pin ID. Also, the "List of Input Value" item grows quite large, (2×64 = 128 bits). With the maximum fan-out of 5, the "List of destination gate ID" has to be 85-bits (5×17 = 85), and the "List of destination Pin ID" should be 30-bits (= 5×6).

Table 46 Data Width for 100,000 Gate Simulation

Data item	Bits	Comments
Mask1	2	Mask values for 0, 1, Z, X
Mask2	2	Mask values for 0, 1, Z, X
Function Group	4	16 different functions
number of inputs	6	total 64 inputs
number of outputs	1	total 2 outputs
Level 1 output inversion flag	8	1 flag bit for level-1 gates, 8 total
Level 1 input inversion flag	8	1 flag bit for each input or level-1 gate
Level 2 output inversion flag	2	1 flag bit for each output
Delay RAM base address	19	*ceiling*(*log*(delay space))
List of Input Value	128	64 total inputs, 2 bits total
List of Output Value	4	2 outputs, 2 bits each
Power Count	20	1 million switching count
List of destination Gate ID	170	GateID * average fan-out * max output
List of destination Pin ID	60	encoded input * average fan-out * max output
TOTAL	434	

As was shown from the Table 46, 434 bits are required to perform a 100,000 gate design logic simulation task. This is about 13.5 times wider than the memory bus width of a 32-bit generic workstation. And therefore, workstations require multiple memory accesses to read and write this wide data. On the other hand, our design performs this 434-bit wide memory access in one shot, and achieves the performance gain.

7.4 Performance Comparison

As was shown in Figure 68, our experimental results for the prototype show that our design can process one event every 44 cycles. If our system runs at 200MHz, then for every 220 ns we can process one event. This is equivalent to $1/(220\times10^{-9}) = 4.55$ million events per second.

As a comparison, we have measured the software performance of Modelsim [34]. To acquire simple performance numbers in full-timing mode, we supplied 100 inverters in series and measured the run time. We have also measured the performance with 3,000 inverters in series in the same manner. In the 100 inverter case, Modelsim reported a time of 62.84 micro-seconds per event, which is equivalent to $1/(62.84\times10^{-6})\approx16{,}000$ events per second. In the 3,000 inverter case, Modelsim reported 75 micro-seconds per event, which is equivalent to $1/(75\times10^{-6})\approx13{,}333$ events per second. As an average, we will use 70 micro-second per event ($\approx$ 14,000 events per second) as a software performance measure. In comparison, our architecture achieved a speed up of 325 (= 4.55 million / 14,000) over a software logic simulation.

Our performance can be further improved if we employ a multi-port memory and pipeline the architecture. The logic evaluation task endemically involves a *read-modify-writeback* to the *same memory location*. Memory pipelining is not possible if we have to "lock" the memory location unless we use a multi-port memory, so that memory-read and memory-write can be performed independently.

Our bottleneck also comes from slow memory performance on both the scheduler block and logic evaluation block, because logic simulation is a memory intensive task. Having a faster memory will benefit both hardware and software approaches. But our performance comes from a wide memory access and a simplified hardware structure by avoiding overheads caused by the operating system, virtual memory management, and multiple instructions run on a general purpose processors.

IBM's LSM and YSE [14, 15] have 63,000 and 64,000 gate capacity, respectively, with a speed of 12.5 M gates/sec. They can handle 4-level signal strength, but cannot handle full-timing simulation. IBM's last product, EVE [16], uses 200 processors with a top speed of 2.2 billion gates/sec (assumed linear) and a two million gate capacity, but they still cannot handle full-timing simulation. All of IBM's accelerators limit the maximum number of inputs to four. In comparison, IBM's accelerator is 275% faster than our design because their lack of full-timing capability simplifies the hardware and therefore, improves the speed. Also, limiting the number of inputs require that a single cell with more than 4 inputs has to be broken up into multiple pieces before each piece can be evaluated and merged. This will drop the actual event per second performance number. As a conclusion, our hardware will out-feature IBM's design with a trade off of speed.

ZyCAD's LE system performs 3.75 million gates per second per processor and has a top speed of 60 million gates per second (assumed linear) with 16 processors [18]. It can also handle 4-level signal strength without full-timing simulation. In comparison, our design out-performs ZyCAD by 21%.

The most recent product is NSIM, developed by IKOS [19]. Their performance claim is 40 to 100 times faster than software. The reason for varying performance is due to the IKOS primitive. In the IKOS simulation environment, the circuit under test (CUT) has to be mapped according to the IKOS provided primitives, while our approach is based on modeling the cell on an "as is" basis, as was shown in Chapter 4. Therefore, re-mapping the CUT into IKOS primitives consumes time and reduces the capacity (complex cells have to be broken into tens of IKOS primitives). This re-mapping of IKOS primitives also causes complex cells to generate more events, which reduces the performance. NSIM does handle full-timing simulation. As discussed above, the average software performance (Modelsim) is approximately 70 micro-second per event (14,000 events/second) for full-timing simulation. When NSIM is 40 times faster than software, then its performance is $14000 \times 40 = 560{,}000$ events per second. If NSIM is 100 times faster than software, then its performance is $14000 \times 100 = 1.4$ million events per second. Therefore, our design out-performs NSIM with similar features. Our design is $4.55/0.56 = 8.125$ times faster than NSIM if NSIM is 40 times faster than software, and $4.55/1.4 = 3.25$ times faster if NSIM is 100 times faster than software.

The emulation acceleration hardware (Quickturn and IKOS) cannot be directly compared with simulation accelerators, because emulators directly map the circuit design into the given platform (usually an array of FPGAs) and physically run the design in the system. Therefore, they loose all the technology dependent information and ignore the timing information. These emulators are extremely fast, but they can only be used for

verifying logical correctness, because they lack all the features that most circuit verifiers need.

Since our design allows one to handle co-simulation with a large timing resolution, the performance of our scheduler severely impacts the overall performance. Table 47 shows the effect caused by different event memory configurations of the scheduler. The memory depth indicates the depth of a single segment of event memory for a local minimum search. As we can see from the table, our performance drops as the depth of the event memory increases, because our scheduler is based on a linear search.

Table 47 Event Memory Depth vs. Performance

Memory Depth (*k*)	M Events/sec
8	4.55
16	2.63
32	1.43
64	0.75
128	0.38
256	0.19

We have compared the performance of our design with the performance of IKOS NSIM, which is currently available in the market. The result is shown in Figure 70. Up to the memory depth (*k*) of 32, our performance is better or similar to their peak performance (100 times faster than software), and from 32 to 64 our performance is still better than their 40 times performance claim. However, when the memory depth is increased to 128 or more, our performance drops below their performance.

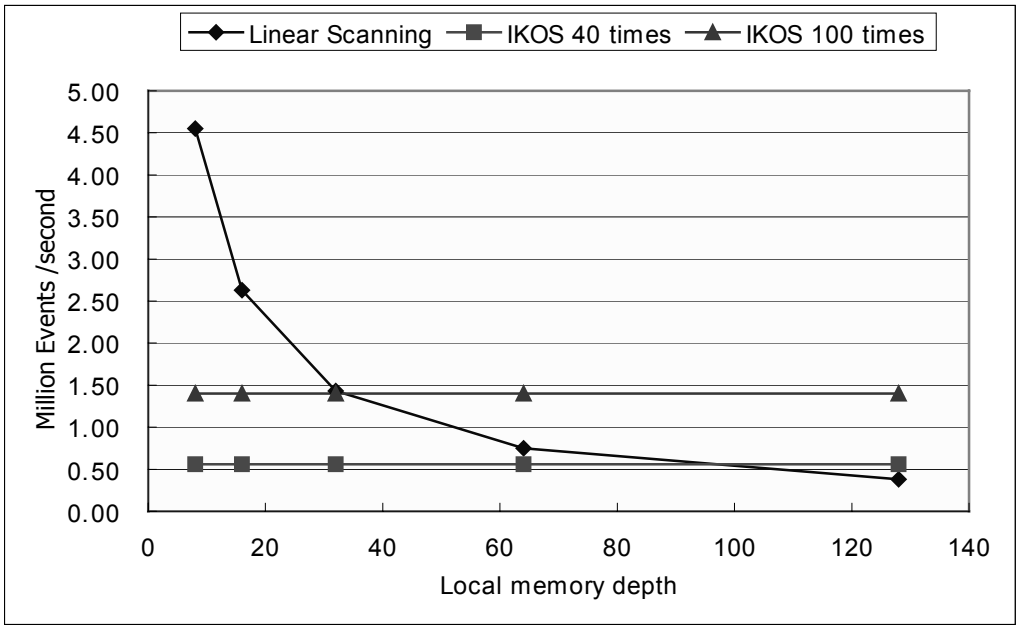


Figure 70 Performance Comparison between Our Design and IKOS

As discussed in a previous chapter, an event wheel can improve the scheduler's performance. IKOS uses the event wheel as their scheduler implementation and therefore their scheduling capability is limited in nature.

As a conclusion, the performance of our design is faster than IKOS and similar to ZyCAD, but slower than IBM. But our design out-features all hardware acceleration systems. Table 48 summarizes the performance and feature comparison.

Table 48 Performance and Feature Comparison

	Capacity	Full Timing	Multi-level Signal Strength	Co-simulation	Power & Heat	Events/Sec
IBM	63K to 2M	no	Yes	n/a	no	12.5M
ZyCAD	1M	yes	Yes	n/a	no	3.75M
IKOS	8M	yes	Yes	n/a	no	560K to 1.4M
ModelSim	scalable	yes	Yes	n/a	no	14K
Univ. Pitt	100K+	yes	Yes	possible	yes	4.55M

As was shown in Figure 68, our design has a mechanism to record the power count. This is a new feature built into our architecture, which can guide the designer to isolate the thermal hot spots in the design. If our design is used in the pre-technology mapping stage, this information can guide the layout process so that the hot- spots in the chip can be more evenly distributed, allowing the chip to run cooler. Adding up all of the output change counts will also provide the designer with a measure of power consumption. The equation for dynamic power consumption was discussed in Chapter 1.

We have successfully demonstrated that our design can out perform the software logic simulation. We also compared our performance and features to the existing hardware simulation accelerators, and have shown that our design has better or equivalent performance, and that our design provides more features than existing hardware simulators.

8.0 CONCLUSIONS

8.1 Summary

With the increasing complexity of modern digital systems, and tight time to market restrictions, design verification has become one of the most important steps in the Electronic Design Automation (EDA) area. Design verification through full-timing logic simulation has been relying on software running on a fast workstation with a general-purpose architecture. Due to the growing complexity of circuits, this software solution no longer provides sufficient performance.

In Chapter 1, the need for a fast and accurate logic simulation mechanism was discussed. Chapter 2 described various algorithms used in the software approach. The performance bottleneck was identified and discussed, as was related research. In Chapter 3, a new hardware architecture was proposed and its architectural component modules and tasks were defined. Chapter 4 introduced the concept of behavioral modeling for each of the logic cells and primitives. We introduced a new set of primitives called the Any( ) and All( ) functions. With these new primitive functions, the logic cell evaluation design was optimized and implemented. The size of the standard lookup table approach was computed and compared with our approach. In an 8-input Universal Gate for AND/OR/NAND/NOR implementation example, the size reduction factor of 21,845 was achieved. To implement a full-timing simulation, various delay models were introduced and different memory architectures were explored and compared in Chapter 5. In

Chapter 6, we analyzed the existing scheduling algorithm and identified the related problems and the performance bottlenecks. We have proposed a parallel sub-scanning scheduler design, which can handle mixed timing resolution so that it can be expanded into the hardware/software co-simulation. In Chapter 7, a proof of concept prototype of our architecture was implemented, and its performance was measured and compared to the existing software and hardware simulators. Experimental results show that our architecture can achieve a speed up of 325 over the software logic simulation.

8.2 Contributions

This thesis seeks an architecture design to enhance the performance of logic simulation in hardware. The primary contributions of this work are as follows:

1. Software Performance Bottleneck Analysis: The logic simulation software algorithm and performance were analyzed and the performance bottleneck was identified as the memory activity of "*read-modify-writeback*".
2. Hardware Concurrency: Each sub-task is designed in a designated hardware to utilize parallelism within the logic simulation algorithm.
3. Behavioral Modeling and Universal Gate: A cell library was examined and cells were modeled according to their behavior. Based on the behavioral model, the concept of a "Universal Gate" was developed and implemented in hardware to simplify the logic evaluation process. This Universal Gate was also used to implement multilevel logic cells such as AO and OA logic cells. The new

hardware primitives designed for the behavioral modeling includes: Any/All circuit, equivalence checker and edge detector circuits. Emulation is also included into the evaluation to compute XOR and XNOR logic functions. Universal gates are reused for evaluating multi-level logic cells.

4. The Scheduler: The scheduler circuit is designed and implemented to provide more accurate timing (fine grain timing resolution). The memory structure of the scheduler is divided to exploit the concurrency in the scheduling algorithm.
5. Multi-level Signal Strengths: The architecture handles the 4-level signal strengths and a full-timing delay model.
6. Scalable Architecture: The architecture is capable of computing up to 8 inputs for single level gates, and 64 inputs for two level gate cells. The architecture is also designed to scale to over a 100,000 gate capacity to accommodate the complexity of the modern digital systems.
7. Speedup: Our architecture has a speed up of 325 over a software logic simulator.
8. Power Count: The output change count mechanism is built into the architecture so that it can be used in test coverage, stuck-at fault simulation, and thermal topology analysis.
9. Pre-processing Software: Parsing software was implemented to provide accurate memory map information for the architecture. The software shares the same data structure with our architecture, and plays a crucial role in hardware logic simulation.

8.3 Future Work

As was discussed in previous chapters, the performance bottleneck of the logic simulation task comes from memory performance. In terms of memory activity, the logic simulation task can be viewed as "*read-modify-writeback*" to the same memory location. To enhance the system performance, a pipelined architecture can be employed. If a pipeline is implemented, the architecture will have the performance benefit as long as the system does not attempt to access the same gate information in the net-list and configuration memory consecutively. In such cases, pipeline has to be stalled until the net-list memory finishes its update phase (*writeback*). If successive pending event points to the same gate frequently, the system will not gain any performance improvement. Even with a multi-port memory, the pipeline architecture can still face the hazard situation. If two or more events point to the same gate within the pipeline cycle, the pipeline has to be stalled. Otherwise, the pipeline architecture can utilize the concurrency in the simulation task.

Our architecture has been prototyped in an FPGA, on which the performance has degraded considerably due to the nature of a PLD device. If the architecture is implemented using ASIC technology, all of the functional blocks we have designed can maintain the delay characteristic within the unit. Therefore, with an addition of a small routing delay, the architecture can perform in a more predictable manner.

BIBLIOGRAPHY

BIBLIOGRAPHY

1. International Technology Roadmap for Semiconductors, http://public.itrs.net/
2. Gajski, Daniel D., Dutt, Nikkil D., Wu, Allen C-H, and Lin, Steve Y-L., "High-Level Synthesis; Introduction to Chip and System Design," Kluwer Academic Publishers, Boston, 1992.
3. Advanced Micro Devices (AMD) Inc., "Athlon data sheet", http://www.amd.com
4. Intel Corporation, "Pentium-III data sheet", http://www.intel.com
5. N. Weste, *et. al.*, "Principles of CMOS VLSI Design A Systems Perspective," Addison-Wesley Publishing Company, 1988.
6. Prithviraj Banerjee, "Parallel Algorithms for VLSI Computer-Aided Design," Prentice Hall, 1994.
7. M. A. Breuer and A. D. Friedman, "Diagnosis and Reliable Design of Digital Systems," Computer Science Press, 1976.
8. L. Soule and T. Blank, "Parallel logic simulation on General Purpose Machines," Proc. Design Automation Conference, pp. 166-171, June, 1988.
9. R. M. Fujimoto, "Parallel Discrete Event Simulation," Communications of the ACM, 33(3):30-53, Oct. 1990.
10. K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," Communications of the ACM, 24(11), pp. 198-206, April. 1981.
11. R. E. Bryant, "A Switch-level Model and Simulator for MOS Digital Systems," IEEE Transactions on Computers, C-33(2):160-177, Feb. 1984.

12. D. Jefferson, "Virtual Time," ACM Trans. Programming Languages Systems, pp. 404-425, July, 1985.
13. J. Briner, "Parallel Mixed Level Simulation of Digital Circuits using Virtual Time," Technical Report, Ph. D. Thesis, Department of Electrical Engineering, Duke University, Durham, NC, 1990.
14. J. K. Haroward, *et. al.*, "Introduction to the IBM Los Gatos Logic Simulation Machine," Proc. IEEE Int. Conf. Computer Design: VLSI in Computers, pp. 580-583, Oct., 1983.
15. G. Pfister, "The Yorktown Simulation Engine," Proc. 19th Design Automation Conference, pp. 51-54, June, 1992.
16. L. N. Dunn, "IBM's Engineering Design System Support for VLSI Design and Verification," IEEE Design and Test of Computers, pp. 30-40, Feb., 1984.
17. Fehr, Edward Scott, "An Array-based Hardware Accelerator for Digital Logic Simulation", Ph. D. Thesis, University of Texas at Austin, 1992.
18. ZYCAD Co., "Zycad Logic Evaluator LE-1000 series – Product Description," Technical Report, St. Paul, MN, 1987.
19. IKOS, "NSIM," http://www.ikos.com
20. P. Agrawal, *et. al.*, "Architecture and Design of the MARS Hardware Accelerator," Proc. 24th Design Automation Conference, pp. 108-113, June, 1987.
21. S. Walters, "Computer-Aided Prototyping for ASIC-based Systems," IEEE Design and Test of Computers, 8(2):4-10, June, 1991.

22. IKOS, “Virtual Logic Emulator,” http://www.ikos.com

23. ISCAS Benchmark suite, http://ftp.cbl.ncsu.edu/www/benchmarks

24. IEEE std. 1164-1993, “IEEE standard multivalue logic system for VHDL model interoperability (stdlogic1164),” May 1993.

25. Gary D. Hachtel, *et. al.*, “Logic Synthesis and Verification Algorithms,” Kluwer Academic Publishers, 1998.

26. Yunmo Chung, “Logic Simulation on Massively Parallel SIMD Machines,” Ph. D. Thesis, Michigan State University, 1990.

27. John L. Hennessy and David A. Patterson, “Computer Architecture A Quantitative Approach,” Morgan Kaufmann, 1996.

28. R. E. Bryant, “Simulation of Packet Communications Architecture Computer Systems,” Technical Report MIT-LCS-TR-188, Laboratory of Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1977.

29. Phillip Rens Prins, “Parallel Logic Simulation,” Ph. D. Thesis, Department of Electrical Engineering, University of Idaho, 1995.

30. Altera Corporation, “Apex 20K data sheet”, http://www.altera.com

31. IEEE std 1364-1995, “IEEE Standard Hardware Description Language based on the Verilog Hardware Description Language,” August 1996.

32. IEEE std P1497, “IEEE Draft Standard for Standard Delay Format (SDF) for the Electronic Design Process,” June 2000.

33. T. H. Cormen, *et. al.*, “Introduction to Algorithms,” McGraw-Hill, 1994.

34. Mentor Graphics Corporation, “ModelSim”, http://www.mentor.com

35. James T. Cain, Marlin H. Mickle, and Lawrence P. McNamee, "Compiler Level Simulation of Edge Sensitive Flip-Flops," AFIPS Conference Proceedings Volume 30, April 1967.
36. W. E. Cohen, *et. al.*, "Dynamic Barrier Architecture for Multi-Mode Fine-Grain Parallelism Using Conventional Processors; Part I: Barrier Architecture," Purdue University School of Electrical Engineering, Technical Report TR-EE 94-9, March 1994.
37. H. G. Dietz, *et. al.*, "A Fine-Grain Parallel Architecture Based On Barrier Synchronization," Proceedings of the International Conference on Parallel Processing, August 1996.