International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 06 Issue 13
December 2019

Area And Power Efficient Mac Unit

Kottapalli Nageswari[1], S. Mahaboob Basha[2]
[1]P.G. Scholar, [2]Head of the Department
[1,2] Branch:ECE (VLSI)
[1,2] Geethanjali College of Engineering and Technology, Nannur
Email Id:[1,2] nageswarinandu1995@gmail.com

ABSTRACT

In the field of semiconductor design industry which, in the contemporary times, has observed exceptional, explosive and exhilarating growth in the development of portable communication devices like mobile phones, IPADS and note books. These real time processing systems perform high computational operations, mainly in the form of butterfly and Multiply Accumulate (MAC). However, these systems are expected to consume high power and are characterized by high data throughput rate. Of the two, MAC is a major component used in portable applications and communication sectors like Wireless Code Division Multiple Access (WCDMA), base station receivers, Successive Interference Canceller (SIC), Orthogonal Frequency Division Multiplexing (OFDM) based wireless devices, channel estimators and carrier synchronizers. In general the MAC block resides in the critical path, which governs the complete power and speed of the system. The efficient utilization of MAC in terms of speed and power depends upon the type of architecture, logic technology style, the fundamental block and primitive cell realization. This study vividly presents the bird eye view on the hitherto work concerning the existing MAC unit in terms of its power performance factors, which helps the future researcher for opting suitable MAC block which can be used in Field Programmable Gate Array (FPGA) and Application Specific Integrated Circuit (ASIC) for signal processing applications. The comparative analysis is based on architecture/size, number of clock cycles, Partial Product Reduction Tree (PPRT), functional module, power saving method, logic technology, fabrication process and speed-voltage performance.

Keywords: Multiplier-and-Accumulator (MAC), Modified Gate Diffusion Input Technique (MGDI), Power consumption, Digital Signal Processor (DSP)

INTRODUCTION

Overview

MAC unit is an inevitable component in many digital signal processing (DSP) applications involving multiplications and/or accumulations. MAC unit is used for high performance digital signal processing systems. The DSP applications include filtering, convolution, and inner products. Most of digital signal processing methods use nonlinear functions such as discrete cosine transform (DCT) or discrete wavelet transforms (DWT). Because they are basically accomplished by repetitive application of multiplication and addition, the speed of the multiplication and addition arithmetic determines the execution speed and performance of the entire calculation. Multiplication-and-accumulate operations are typical for digital filters. Therefore, the functionality of the MAC unit enables high-speed filtering and other processing typical for DSP applications. Since the MAC unit operates completely

independent of the CPU, it can process data separately and thereby reduce CPU load. The application like optical communication systems which is based on DSP , require extremely fast processing of huge amount of digital data. The Fast Fourier Transform (FFT) also requires addition and multiplication. 64 bit can handle larger bits and have more memory.

A MAC unit consists of a multiplier and an accumulator containing the sum of the previous successive products. The MAC inputs are obtained from the memory location and given to the multiplier block. The design consists of 64 bit modified Wallace multiplier, 128 bit carry save adder and a register.

A design of high performance 64 bit Multiplier-and-Accumulator (MAC) is implemented in this paper. MAC unit performs important operation in many of the digital signal processing (DSP) applications. The multiplier is designed using modified Wallace multiplier and the adder is done with carry save adder. The total design is coded with verilog-HDL and the synthesis is done using Cadence RTL complier using typical libraries of TSMC 0.18um technology. The total MAC unit operates at 217 MHz. The total power dissipation is 177.732 mW.

The goal of this project is to design and implement a MAC unit and an Arithmetic Logic Unit (ALU). The MAC unit is a 16x16-bit 2's complement multiplier with a 40-bit accumulator.The ALU performs 16-bit arithmetic and includes saturating addition/subtraction logic. The imple-mentation includes a full custom layout and veri_cation of all cell necessary to complete the units.

We perform all our simulations for the TSMC 0.18 _m process, and the chip that uses these units will be fabricated. The priorities of this project, in order of importance, are:Robust and safe circuits.Design timeArea/speed balance. The most important priority during this project is to ensure that it works. Thus, the circuits must be robust and safe, and we must choose designs that are largely immune to noise and generate full rail-to-rail swing on the outputs. To be safe, all our circuits are designed with static CMOS, which means at every point in time, each gate output is connected to either Vdd or Gnd [4].

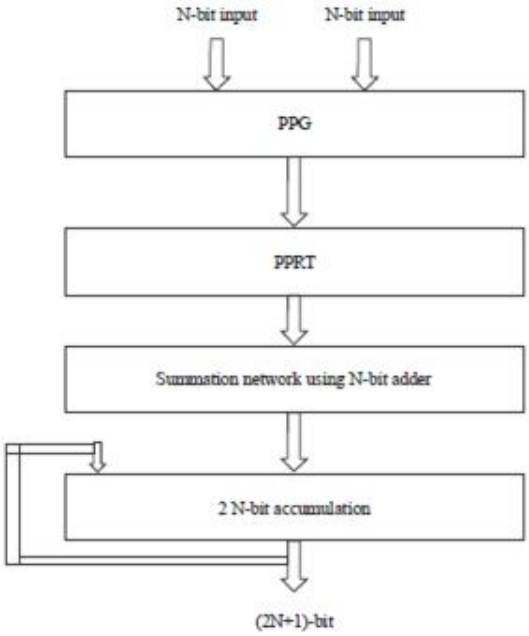
Basically, the architecture of MAC is classified as parallel, recursive and shared segmented structure. The recursive architecture incorporates "divide and conquer" tactic, where the computation of large size data is segmented into smaller units. The multiply-accumulation is achieved using iterative calculation of smaller module through several clock cycles. The latency and throughput of the MAC depends on the number of multipliers and adders, which are recursively called for each cycle. In the first cycle, data is fetched from the internal memory, the second cycle involves multiplication process, during the third cycle summation takes place, while in the fourth cycle the function of A.B+Acc is performed and finally the last output is latched within the internal memory. This approach utilizes minimum hardware by using reusability of resources with increased latency. These types of MAC architectures are deployed in embedded Advanced RISC (Reduced-Instruction-Set-Computing) Machine (ARM) core. The parallel MAC architecture can be constructed by expanding the component of recursive MAC model. The complexity of the structure increases quadratically with the number of inputs. For instance, to implement a 32 bit MAC unit requires 32 bit multiplier, 64 bit adder and 128 bit accumulate unit. This type

International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 06 Issue 13
December 2019

of architecture supports mode dependent logic to support full and half precision multiply or MAC operation. The number of clock cycles is reduced by three, when compared to recursive architecture by means of embedding the accumulator module within the partial product summation network. This MAC architecture is mostly used in Field Programmable Gate Array (FPGA) of Xilinx Corporation and as coprocessor for the LEON2 RISC processor. The shared segmented MAC architecture integrates the structures of split parallel unit and recursive characteristics which operate parallel with moderate resources supporting mode-dependent logic. This kind of architecture lacks from throughput limitations owing to compound PPRT. This architecture is capable of supporting full and half precision multiply or MAC operation. This MAC operates for SIMD (Single Instruction Multiple Data) with reduced clock cycle, when compared to recursive MAC and reduced hardware with respect to parallel MAC. This type of architecture is used in MIPS Technologies (Million Instructions per Second).

1.2 POWER PERFORMANCE FACTORS OF MAC UNIT

The performance of MAC unit depends on the following parameters:



Power: The three major components of power are: Transient or dynamic, short circuit and leakage power. The short circuit power is owing to the current conducting path between GND and VDD. The leakage power is due to the reverse bias diode and sub threshold leakage. These two powers are due to the logic style and technology, through which the MAC is realized. The transient or dynamic power is due to the total number of nodes and capacitors charged/discharged in a transition which is expressed as follows:

$$P_{dynamic} = \alpha_{transition} C_{pd} f_{clk} VDD^2 \quad (3)$$

where, $\alpha_{transition}$ is the total number of nodes active per transition (node activity factor), C_{pd} is the dynamic power capacitors, f_{clk} is the clock frequency (Input/output) and VDD is the supply voltage. So, in the MAC unit the major portion of power is contributed due to transient power that mainly depends upon the node activity factor and dynamic capacitors.

Node activity factor: The node activity factor represents the total number of nodes active per transition, which is divided into two parts, namely, the function and parasitic part. The

function (part) switching depends on the type of architecture used in the design, it depends on the logic function and block such as AND, OR, NAND, multiplier and adder, the signal statistics and the choice of logic style. The second part is due to glitches, which is caused due to signal skews (different input signal arrival time) and the signal statistics. The parasitic part in the MAC can be reduced using balanced delay path, gate sizing and reducing parasitic capacitances.

Clock frequency: It is one of the significant parameters persuading the functional power dissipation of MAC unit. The power factor is directly proportional to clock frequency of the MAC unit, therefore reducing clock frequency may proportionally reduce power, on the other hand, the MAC speed and throughput simultaneously reduced. In order to preserve the throughput for reduced clock frequency parallelism and pipelined architecture have to be considered. The MAC architectural power (block level) is characterized in terms of bits of the component (multipliers and adders) and their operating frequency which can be expressed as:

$$P_{functional-block} = \Delta_1 \sum_{input\ i} f_i + \Delta_2 \sum_{input\ j} f_j = \Delta_1 f_{in} + \Delta_2 f_{out} \quad (4)$$

where, f_{in} and f_{out} is the input and output frequency of MAC unit and Δ_1, Δ_2 are the empirical coefficients derived from gate-level simulation.

Architecture selection: Low power architecture design becomes imperative in MAC block. The architecture selection typically involves the organization of functional blocks in MAC and the number of pipes involved in the computation stage. In architectural level, low power can be achieved through clocking strategy, parallelism, pipelining and component organization. By deploying parallelism the throughput and performance of MAC unit can be improved without increasing the operating frequency.

In recursive architecture, the resource utilization is minimum, therefore, area reduction is achieved but the throughput of the system is considerably very low. Due to the smaller bit-size, component and reusability, the dynamic power is reduced. For parallel MAC the number of components will be doubled, when compared to recursive MAC architecture to achieve high speed and throughput at the expense of increased chip area twice that of recursive MAC. To reduce the area penalty of parallel MAC, split-pipelined MAC architecture is suitable, when trade-off results with less area overhead but more complexity in controller design due to multi-mode operation. When pipelined structures are implemented then the propagation delay is reduced to half when compared to the recursive MAC. On the other hand, consideration should be taken for pipelined MAC to maintain the throughput, when it is operating at lower voltages.

Logic technique: Till date, the majority of the circuit designs have been implemented using Complementary Metal Oxide Semiconductor (CMOS) logic style. It is very attractive due its reliable operation at low voltages. The CMOS logic style incorporates large PMOS (P-type Metal Oxide Semiconductor) in circuit realization. As a result, the propagation delay is higher, because of large node capacitances. The power dissipation is very high at high operating frequencies due to increase in the input loads. From the literature survey it has been observed that the MAC block has been implemented using static Complementary Metal Oxide Semiconductor (CMOS), Clocked-

transmission Gate Adiabatic Logic (CTGAL), Low Voltage Swing Restoration Technique (LVST), Pass Transistor Logic (PTL), Complementary Pass Transistor Logic (CPL), mixed static CMOS-CPL and Swing Restored PTL (SRPL).

The PTL provides improved performance, when compared to CMOS logic due to less number of transistors, as a result, the overall parasitic capacitances is reduced. The dynamic power dissipation is very minimal due to faster switching time. One of the short falls associated with the PMOS logic is threshold drop variation. As a result, the noise margin of the circuit is reduced this in turn degrades the driving capability and leads to unreliable operation. The static power dissipation is very high in PTL due to threshold drop variation. The LVST logic detects the input voltages even, when it is less than 100 mV and performs reliable operation in low voltages. The logic structure realization is very complicated, with three level stages with true and complementary inputs, which add on the number of inverters in the design, in turn, it increases the static power dissipation of the circuit and poses moderate noise immunity.

The CPL logic required large number of transistors or gates to implement simple circuit. Due to large transistor the short circuit current is high and wiring overhead owing to the dual-rail signals. The SRPL circuit design is similar to LVST with three stages supporting true and complementary inputs, which differentiate two low inputs and regenerative operation is established through sense amplifier. In SRPL, when proper device scaling is not provided then discharging the output for 1-0 transition becomes bottleneck and consequently the output degrades. The MAC constructed using SRPL utilizes high transistor count and fair noise margin.

Functional blocks: The indispensable component necessary to implement the MAC blocks are adder and multiplier circuits. As mentioned earlier, the first stage of the MAC unit involves the generation of partial products, which can be established through multiplier circuits and the second stage is accumulation of PPG, which can be accomplished using adder circuits. From the literature survey, it have been observed that various multiplier and adder circuits like distributed arithmetic, parallel, serial-parallel, complementary (Booth encoding), Wallace using CSA, row-column bypass, modulo diminishing-1 and wave pipelining multipliers. The MAC architecture with complementary booth multiplier reduces the generation of number of partial products. Major bottle neck, when deploying booth multiplier is hardware complexity due to fundamental components of encoder and shift registers to produce PPG. Due to high interconnect the power dissipation is very high in this type of MAC architecture. While, deploying Wallace multiplier scheme the time complexity is reduced by N/2, when compared to array structure but dissipates high power due to irregular interconnects.

The accumulation and PP addition in MAC unit are performed using, various adder schemes. The general structures deployed are Parallel Prefix Adder (PPA), Ripple Carry Adder (RCA), Carry Skip Adder (CSkA), Carry Propagate Adder (CPA), Carry Save Adder (CSA) and Carry Select Adder (CSelA). The adder structures deployed in the final stage of MAC have been implemented using Carry Save Adder (CSA), Carry Select Adder (CSelA),

Carry Look Ahead Adder CLAA) and Parallel Prefix Adder (PPA). These adders however exhibits power dissipation and delay due to interconnect scheme and data distribution.

Figure of merit: The Figure of Merit (FOM) is expressed as:

$$FOM = \frac{f}{PV} \times 100 \quad (5)$$

where, P is the total power of MAC unit for the given voltage (V) operating at a given frequency and this performance parameter should be minimum.

Throughput: The throughput of the MAC design is computed with respect to the clock frequency f_{clk} and latency in various pipe stages. The throughput of the MAC can be expressed as

$$MAC_{operations\ per\ second} = \frac{Frequency}{Cycle\ each\ MAC\ operation} \times 1 \quad (6)$$

In above Eq. 6 the term parallel MAC denotes the number of MAC unit deployed to execute the instruction, generally it is expressed as Mega Operation Per Second (MOPS).

1.3 MAC ARCHITECTURE

The architecture selection for MAC unit generally depends upon the type of applications. For embedded microprocessor or microcontroller applications the memory usage is limited and the operand size is also small and therefore, recursive architecture is suitable, when power and area is important. This recursive MAC unit is deployed in image processing application such as Fast Fourier Transform (FFT) and digital filtering.

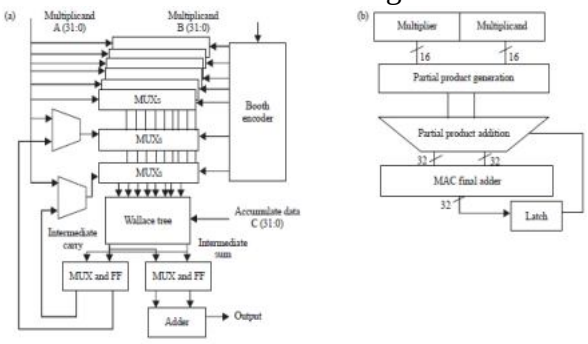
For high performance applications like notepads, laptops and desktops require large set of data computation therefore parallel architecture will be suitable. To perform multi-mode logic dependent operation, where the speed and power constraint is considered then shared segmented architecture is preferable, which is mainly used in embedded medical equipments and in communication systems, such as Orthogonal Frequency Division Multiplexing (OFDM) based wireless devices, subcarrier frequency domain operations, channel estimator and carrier synchronizer. The MAC architecture can be implemented in ASIC and FPGA. The implementation of MAC structure using FPGAs will have limited resources and fixed logic technology while in ASIC it is semi-custom or full custom so that optimization can be achieved from the architectural level to transistor level.

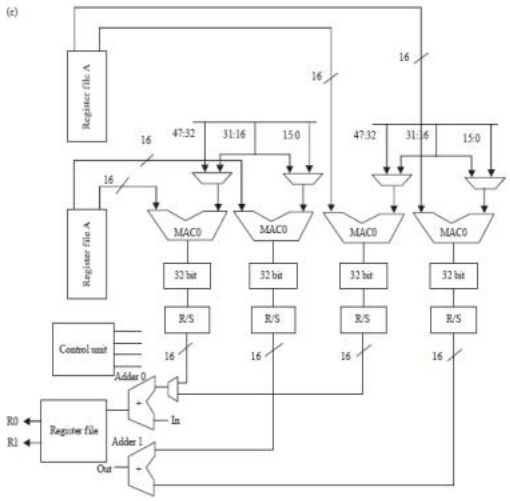
Recursive MAC: A recursive MAC proposed by Matsui *et al.* (1994) use novel Sense-Amplifying Flip-Flop (SA-FF) in combination with NMOS (N-type Metal Oxide Semiconductor) differential logic (Fig. 2a). The MAC unit has been embedded in 2-D DCT which operates at 200 MHz with 350 mW power dissipation for the supply voltage of 3.3 V. The macrocell was fabricated using 0.8 µm base rule CMOS technology. The SA-FF technique acts as a sense amplifier to regenerate low-swing differential inputs. The proposed MAC block reduces the propagation time and macrocell size using SA-FF technique. The MAC operation has been executed using the Distributed Arithmetic (DA) on a bit-by bit data. The intermediate addition process has been implemented using conventional RCA and Carry Propagation Adder (CPA). The final accumulation and summation network has been designed through Carry Skip Adder (CSkA).

A new Swing Restored Pass Transistor Logic (SRPL) non-pipelined recursive unsigned MAC has been proposed by Parameswar *et al.* (1996) for multimedia applications and it was fabricated in double metal 0.4 µm

International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 06 Issue 13
December 2019

CMOS technology, which operates at a maximum speed of 150 MHz consuming 34 mW with one cycle delay of 6.7 ns for bit size of 16 bit wide. The speed of the MAC unit has been improved using Gate sizing optimization approach, where the aspect ratio W/L (Width/Length) of NMOS transistors connected close to the output signal was reduced, when compared to NMOS transistors that were far away from the output. Another important speed optimization was achieved by using moderately scaled PMOS devices in the swing restoring network. The Partial Product (PP) was obtained using Booth encoding scheme and the same is added using CSAs.





The functional block has been implemented using static CMOS logic. The final stage of addition operation has been constructed using, conditional-sum addition that incorporates Parallel Prefix Structure (PPS). The synchronization of clock circuit has been implemented with single rail domino logic.

What is VLSI?

VLSI stands for "Very Large Scale Integration". This is the field which involves packing more and more logic devices into smaller and smaller areas.

VLSI

- Simply we say Integrated circuit is many transistors on one chip.
- Design/manufacturing of extremely small, complex circuitry using modified semiconductor material
- Integrated circuit (IC) may contain millions of transistors, each a few mm in size
- Applications wide ranging: most electronic logic devices

1. 5.3 History of Scale Integration:

- late 40s Transistor invented at Bell Labs
- late 50s First IC (JK-FF by Jack Kilby at TI)
- early 60s Small Scale Integration (SSI)
  - 10s of transistors on a chip
- late 60s Medium Scale Integration (MSI)
  - 100s of transistors on a chip
- early 70s Large Scale Integration (LSI)
  - 1000s of transistor on a chip
- early 80s VLSI 10,000s of transistors on a
  - chip (later 100,000s & now 1,000,000s)
- Ultra LSI is sometimes used for 1,000,000s

- SSI - Small-Scale Integration (0-102)
- MSI - Medium-Scale Integration (102-103)
- LSI - Large-Scale Integration (103-105)
- VLSI - Very Large-Scale Integration (105-107)
- ULSI - Ultra Large-Scale Integration (>=107)

Advantages of ICs over discrete components:

While we will concentrate on integrated circuits , the properties of integrated

circuits-what we can and cannot efficiently put in an integrated circuit-largely determine the architecture of the entire system. Integrated circuits improve system characteristics in several critical ways. ICs have three key advantages over digital circuits built from discrete components:

- Size. Integrated circuits are much smaller-both transistors and wires are shrunk to micrometer sizes, compared to the millimeter or centimeter scales of discrete components. Small size leads to advantages in speed and power consumption, since smaller components have smaller parasitic resistances, capacitances, and inductances.
- Speed. Signals can be switched between logic 0 and logic 1 much quicker within a chip than they can between chips. Communication within a chip can occur hundreds of times faster than communication between chips on a printed circuit board. The high speed of circuits on-chip is due to their small size-smaller components and wires have smaller parasitic capacitances to slow down the signal.
- Power consumption. Logic operations within a chip also take much less power. Once again, lower power consumption is largely due to the small size of circuits on the chip-smaller parasitic capacitances and resistances require less power to drive them.

VLSI and Systems:

These advantages of integrated circuits translate into advantages at the system level:

- Smaller physical size. Smallness is often an advantage in itself-consider portable televisions or handheld cellular telephones.
- Lower power consumption. Replacing a handful of standard parts with a single chip reduces total power consumption. Reducing power consumption has a ripple effect on the rest of the system: a smaller, cheaper power supply can be used; since less power consumption means less heat, a fan may no longer be necessary; a simpler cabinet with less shielding for electromagnetic shielding may be feasible, too.
- Reduced cost. Reducing the number of components, the power supply requirements, cabinet costs, and so on, will inevitably reduce system cost. The ripple effect of integration is such that the cost of a system built from custom ICs can be less, even though the individual ICs cost more than the standard parts they replace.

Understanding why integrated circuit technology has such profound influence on the design of digital systems requires understanding both the technology of IC manufacturing and the economics of ICs and digital systems.

Applications

- Electronic system in cars.
- Digital electronics control VCRs
- Transaction processing system, ATM
- Personal computers and Workstations
- Medical electronic systems.
- Etc….

Applications of VLSI:

Electronic systems now perform a wide variety of tasks in daily life. Electronic systems in some cases have replaced mechanisms that operated mechanically, hydraulically, or by other means; electronics are usually smaller, more flexible, and easier to service. In other cases electronic systems have created totally new applications. Electronic systems perform a variety of tasks, some of them visible, some more hidden:

- Personal entertainment systems such as portable MP3 players and DVD players

perform sophisticated algorithms with remarkably little energy.

- Electronic systems in cars operate stereo systems and displays; they also control fuel injection systems, adjust suspensions to varying terrain, and perform the control functions required for anti-lock braking (ABS) systems.
- Digital electronics compress and decompress video, even at high-definition data rates, on-the-fly in consumer electronics.
- Low-cost terminals for Web browsing still require sophisticated electronics, despite their dedicated function.
- Personal computers and workstations provide word-processing, financial analysis, and games. Computers include both central processing units (CPUs) and special-purpose hardware for disk access, faster screen display, *etc.*
- Medical electronic systems measure bodily functions and perform complex processing algorithms to warn about unusual conditions. The availability of these complex systems, far from overwhelming consumers, only creates demand for even more complex systems.

The growing sophistication of applications continually pushes the design and manufacturing of integrated circuits and electronic systems to new levels of complexity. And perhaps the most amazing characteristic of this collection of systems is its variety-as systems become more complex, we build not a few general-purpose computers but an ever wider range of special-purpose systems. Our ability to do so is a testament to our growing mastery of both integrated circuit manufacturing and design, but the increasing demands of customers continue to test the limits of design and manufacturing

1.1.1. 5.7 ASIC:

An Application-Specific Integrated Circuit (ASIC) is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use. For example, a chip designed solely to run a cell phone is an ASIC. Intermediate between ASICs and industry standard integrated circuits, like the 7400 or the 4000 series, are application specific standard products (ASSPs).

As feature sizes have shrunk and design tools improved over the years, the maximum complexity (and hence functionality) possible in an ASIC has grown from 5,000 gates to over 100 million. Modern ASICs often include entire 32-bit processors, memory blocks including ROM, RAM, EEPROM, Flash and other large building blocks. Such an ASIC is often termed a SoC (system-on-a-chip). Designers of digital ASICs use a hardware description language (HDL), such as Verilog or VHDL, to describe the functionality of ASICs.

Field-programmable gate arrays (FPGA) are the modern-day technology for building a breadboard or prototype from standard parts; programmable logic blocks and programmable interconnects allow the same FPGA to be used in many different applications. For smaller designs and/or lower production volumes, FPGAs may be more cost effective than an ASIC design even in production.

- An application-specific integrated circuit (ASIC) is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use.
- A Structured ASIC falls between an FPGA and a Standard Cell-based ASIC
- Structured ASIC's are used mainly for mid-volume level design. The design task for structured ASIC's is to map the circuit into a fixed arrangement of known cells.

INTRODUCTION TO XILINX

Migrating Projects from Previous ISE Software Releases:

When you open a project file from a previous release, the ISE® software prompts you to migrate your project. If you click Backup and Migrate or Migrate Only, the software automatically converts your project file to the current release. If you click Cancel, the software does *not* convert your project and, instead, opens Project Navigator with no project loaded.

Note: After you convert your project, you *cannot* open it in previous versions of the ISE software, such as the ISE 11 software. However, you can optionally create a backup of the original project as part of project migration, as described below.

To Migrate a Project

1. In the ISE 12 Project Navigator, select **File > Open Project**.
2. In the Open Project dialog box, select the .xise file to migrate.

Note : You may need to change the extension in the Files of type field to display .npl (ISE 5 and ISE 6 software) or .ise (ISE 7 through ISE 10 software) project files.

3. In the dialog box that appears, select **Backup and Migrate** or **Migrate Only**.
4. The ISE software automatically converts your project to an ISE 12 project.

Note : If you chose to Backup and Migrate, a backup of the original project is created at *project_name*_ise12migration.zip.

5. Implement the design using the new version of the software.

Note : Implementation status is *not* maintained after migration.

Properties:

For information on properties that have changed in the ISE 12 software, see ISE 11 to ISE 12 Properties Conversion.

6.3 IP Modules:

If your design includes IP modules that were created using CORE Generator™ software or Xilinx® Platform Studio (XPS) and you need to modify these modules, you may be required to update the core. However, if the core netlist is present and you do not need to modify the core, updates are not required and the existing netlist is used during implementation.

6.4 Obsolete Source File Types:

The ISE 12 software supports all of the source types that were supported in the ISE 11 software.

If you are working with projects from previous releases, state diagram source files (.dia), ABEL source files (.abl), and test bench waveform source files (.tbw) are no longer supported. For state diagram and ABEL source files, the software finds an associated HDL file and adds it to the project, if possible. For test bench waveform files, the software automatically converts the TBW file to an HDL test bench and adds it to the project. To convert a TBW file *after* project migration, see Converting a TBW File to an HDL Test Bench

6.5 Using ISE Example Projects:

To help familiarize you with the ISE® software and with FPGA and CPLD designs, a set of example designs is provided with Project Navigator. The examples show different design techniques and source types, such as VHDL, Verilog, schematic, or EDIF, and include different constraints and IP.

To Open an Example

1. Select **File > Open Example**.
2. In the Open Example dialog box, select the Sample Project Name.

Note To help you choose an example project, the Project Description field describes each project. In addition, you can scroll to the right to see additional fields, which provide details about the project.

3. In the Destination Directory field, enter a directory name or browse to the directory.
4. Click **OK**.

The example project is extracted to the directory you specified in the Destination Directory field and is automatically opened in Project Navigator. You can then run processes on the example project and save any changes.

Note : If you modified an example project and want to overwrite it with the original example project, select **File > Open Example**, select the Sample Project Name, and specify the same Destination Directory you originally used. In the dialog box that appears, select **Overwrite the existing project** and click **OK**.

6.6 Creating a Project:

Project Navigator allows you to manage your FPGA and CPLD designs using an ISE® project, which contains all the source files and settings specific to your design. First, you must create a project and then, add source files, and set process properties. After you create a project, you can run processes to implement, constrain, and analyze your design. Project Navigator provides a wizard to help you create a project as follows.

Note : If you prefer, you can create a project using the **New Project dialog box** instead of the New Project Wizard. To use the New Project dialog box, deselect the **Use New Project wizard** option in the **ISE General page** of the Preferences dialog box.

To Create a Project

1. Select **File > New Project** to launch the New Project Wizard.
2. In the **Create New Project page,** set the name, location, and project type, and click **Next**.
3. *For EDIF or NGC/NGO projects only*: In the **Import EDIF/NGC Project page**, select the input and constraint file for the project, and click **Next**.
4. In the **Project Settings page**, set the device and project properties, and click **Next**.
5. In the **Project Summary page**, review the information, and click **Finish** to create the project

Project Navigator creates the project file (*project_name*.xise) in the directory you specified. After you add source files to the project, the files appear in the Hierarchy pane of the

6.7 **Design panel**:

Project Navigator manages your project based on the design properties (top-level module type, device type, synthesis tool, and language) you selected when you created the project. It organizes all the parts of your design and keeps track of the processes necessary to move the design from design entry through implementation to programming the targeted Xilinx® device.

Note For information on changing design properties, see **Changing Design Properties.**

You can now perform any of the following:

- Create new source files for your project.
- Add existing source files to your project.
  Run processes on your source files.
  Modify process properties.

6.8 Creating a Copy of a Project:

You can create a copy of a project to experiment with different source options and implementations. Depending on your needs, the design source files for the copied project and their location can vary as follows:

- Design source files are left in their existing location, and the copied project points to these files.

- Design source files, including generated files, are copied and placed in a specified directory.
- Design source files, excluding generated files, are copied and placed in a specified directory.

Copied projects are the same as other projects in both form and function. For example, you can do the following with copied projects:

- Open the copied project using the File > Open Project menu command.
- View, modify, and implement the copied project.
- Use the Project Browser to view key summary data for the copied project and then, open the copied project for further analysis and implementation, as described in

Using the Project Browser:

Alternatively, you can create an archive of your project, which puts all of the project contents into a ZIP file. Archived projects must be unzipped before being opened in Project Navigator. For information on archiving, see **Creating a Project Archive.**

To Create a Copy of a Project

1. Select **File > Copy Project**.
2. In the Copy Project dialog box, enter the **Name** for the copy.

Note The name for the copy can be the same as the name for the project, as long as you specify a different location.

3. Enter a directory **Location** to store the copied project.
4. Optionally, enter a **Working directory**.

By default, this is blank, and the working directory is the same as the project directory. However, you can specify a working directory if you want to keep your ISE® project file (.xise extension) separate from your working area.

5. Optionally, enter a **Description** for the copy.

The description can be useful in identifying key traits of the project for reference later.

6. In the Source options area, do the following:

Select one of the following options:

- **Keep sources in their current locations** - to leave the design source files in their existing location.

If you select this option, the copied project points to the files in their existing location. If you edit the files in the copied project, the changes also appear in the original project, because the source files are shared between the two projects.

- **Copy sources to the new location** - to make a copy of all the design source files and place them in the specified Location directory.

If you select this option, the copied project points to the files in the specified directory. If you edit the files in the copied project, the changes do *not* appear in the original project, because the source files are not shared between the two projects.

Optionally, select **Copy files from Macro Search Path directories** to copy files from the directories you specify in the Macro Search Path property in the **Translate Properties** dialog box. All files from the specified directories are copied, not just the files used by the design.

Note: If you added a net list source file directly to the project as described in **Working with Net list-Based IP**, the file is automatically copied as part of Copy Project because it is a project source file. Adding net list source files to the project is the preferred method for incorporating net list modules into your design, because the files are managed automatically by Project Navigator.

Optionally, click **Copy Additional Files** to copy files that were not included in the

original project. In the Copy Additional Files dialog box, use the **Add Files** and **Remove Files** buttons to update the list of additional files to copy. Additional files are copied to the copied project location after all other files are copied.To exclude generated files from the copy, such as implementation results and reports, select

6.10 Exclude generated files from the copy:

When you select this option, the copied project opens in a state in which processes have not yet been run.

7. To automatically open the copy after creating it, select **Open the copied project**.

Note By default, this option is disabled. If you leave this option disabled, the original project remains open after the copy is made.

Click **OK**.

6.11 Creating a Project Archive:

A project archive is a single, compressed ZIP file with a .zip extension. By default, it contains all project files, source files, and generated files, including the following:

- User-added sources and associated files
- Remote sources
- Verilog `include files
- Files in the macro search path
- Generated files
- Non-project files

To Archive a Project:

1. Select **Project > Archive**.
2. In the Project Archive dialog box, specify a file name and directory for the ZIP file.
3. Optionally, select **Exclude generated files from the archive** to exclude generated files and non-project files from the archive.
4. Click **OK**.

A ZIP file is created in the specified directory. To open the archived project, you must first unzip the ZIP file, and then, you can open the project.

Note Sources that reside outside of the project directory are copied into a remote_sources subdirectory in the project archive. When the archive is unzipped and opened, you must either specify the location of these files in the remote_sources subdirectory for the unzipped project, or manually copy the sources into their original location.

INTRODUCTION TO VERILOG

In the semiconductor and electronic design industry, **Verilog** is a hardware description language(HDL) used to model electronic systems. *Verilog HDL*, not to be confused with VHDL (a competing language), is most commonly used in the design, verification, and implementation ofdigital logic chips at the register-transfer level of abstraction. It is also used in the verification ofanalog and mixed-signal circuits.

Overview

Hardware description languages such as Verilog differ from software programming languages because they include ways of describing the propagation of time and signal dependencies (sensitivity). There are two assignment operators, a blocking assignment (=), and a non-blocking (<=) assignment. The non-blocking assignment allows designers to describe a state-machine update without needing to declare and use temporary storage variables (in any general programming language we need to define some temporary storage spaces for the operands to be operated on subsequently; those are temporary storage variables). Since these concepts are part of Verilog's language semantics, designers could quickly write descriptions of large circuits in a relatively compact and concise form. At the time of Verilog's introduction (1984), Verilog represented a tremendous

productivity improvement for circuit designers who were already using graphical schematic capturesoftware and specially-written software programs to document and simulate electronic circuits.

The designers of Verilog wanted a language with syntax similar to the C programming language, which was already widely used in engineering software development. Verilog is case-sensitive, has a basic preprocessor (though less sophisticated than that of ANSI C/C++), and equivalent control flow keywords (if/else, for, while, case, etc.), and compatible operator precedence. Syntactic differences include variable declaration (Verilog requires bit-widths on net/reg types[*clarification needed*]), demarcation of procedural blocks (begin/end instead of curly braces {}), and many other minor differences.

A Verilog design consists of a hierarchy of modules. Modules encapsulate *design hierarchy*, and communicate with other modules through a set of declared input, output, and bidirectional ports. Internally, a module can contain any combination of the following: net/variable declarations (wire, reg, integer, etc.), concurrent and sequential statement blocks, and instances of other modules (sub-hierarchies). Sequential statements are placed inside a begin/end block and executed in sequential order within the block. But the blocks themselves are executed concurrently, qualifying Verilog as a dataflow language.

Verilog's concept of 'wire' consists of both signal values (4-state: "1, 0, floating, undefined") and strengths (strong, weak, etc.). This system allows abstract modeling of shared signal lines, where multiple sources drive a common net. When a wire has multiple drivers, the wire's (readable) value is resolved by a function of the source drivers and their strengths.

A subset of statements in the Verilog language is synthesizable. Verilog modules that conform to a synthesizable coding style, known as RTL (register-transfer level), can be physically realized by synthesis software. Synthesis software algorithmically transforms the (abstract) Verilog source into a net list, a logically equivalent description consisting only of elementary logic primitives (AND, OR, NOT, flip-flops, etc.) that are available in a specific FPGA or VLSI technology. Further manipulations to the net list ultimately lead to a circuit fabrication blueprint (such as a photo mask set for an ASIC or a bit stream file for an FPGA).

History

Beginning

Verilog was the first modern hardware description language to be invented. It was created by Phil Moorby and Prabhu Goel during the winter of 1983/1984. The wording for this process was "Automated Integrated Design Systems" (later renamed to Gateway Design Automation in 1985) as a hardware modeling language. Gateway Design Automation was purchased by Cadence Design Systems in 1990. Cadence now has full proprietary rights to Gateway's Verilog and the Verilog-XL, the HDL-simulator that would become the de-facto standard (of Verilog logic simulators) for the next decade. Originally, Verilog was intended to describe and allow simulation; only afterwards was support for synthesis added.

Verilog-95

With the increasing success of VHDL at the time, Cadence decided to make the language available for open standardization. Cadence transferred Verilog into the public domain under the Open Verilog International (OVI) (now known as Accellera) organization. Verilog was later submitted to IEEE and became IEEE Standard 1364-1995, commonly referred to as Verilog-95.

In the same time frame Cadence initiated the creation of Verilog-A to put standards support behind its analog simulator Spectre. Verilog-A was never intended to be a standalone language and is a subset of Verilog-AMS which encompassed Verilog-95.

Verilog 2001

Extensions to Verilog-95 were submitted back to IEEE to cover the deficiencies that users had found in the original Verilog standard. These extensions became IEEE Standard 1364-2001 known as Verilog-2001.

Verilog-2001 is a significant upgrade from Verilog-95. First, it adds explicit support for (2's complement) signed nets and variables. Previously, code authors had to perform signed operations using awkward bit-level manipulations (for example, the carry-out bit of a simple 8-bit addition required an explicit description of the Boolean algebra to determine its correct value). The same function under Verilog-2001 can be more succinctly described by one of the built-in operators: +, -, /, *, >>>. A generate/endgenerate construct (similar to VHDL's generate/endgenerate) allows Verilog-2001 to control instance and statement instantiation through normal decision operators (case/if/else). Using generate/endgenerate, Verilog-2001 can instantiate an array of instances, with control over the connectivity of the individual instances. File I/O has been improved by several new system tasks. And finally, a few syntax additions were introduced to improve code readability (e.g. always @*, named parameter override, C-style function/task/module header declaration).

Verilog-2001 is the dominant flavor of Verilog supported by the majority of commercial EDA software packages.

Verilog 2005

Not to be confused with SystemVerilog, *Verilog 2005* (IEEE Standard 1364-2005) consists of minor corrections, spec clarifications, and a few new language features (such as the uwire keyword).

A separate part of the Verilog standard, Verilog-AMS, attempts to integrate analog and mixed signal modeling with traditional Verilog.

SystemVerilog

SystemVerilog is a superset of Verilog-2005, with many new features and capabilities to aid design verification and design modeling. As of 2009, the SystemVerilog and Verilog language standards were merged into SystemVerilog 2009 (IEEE Standard 1800-2009).

The advent of hardware verification languages such as OpenVera, and Verisity's e language encouraged the development of Superlog by Co-Design Automation Inc. Co-Design Automation Inc was later purchased by Synopsys. The foundations of Superlog and Vera were donated to Accellera, which later became the IEEE standard P1800-2005: SystemVerilog.

In the late 1990s, the Verilog Hardware Description Language (HDL) became the most widely used language for describing hardware for simulation and synthesis. However, the first two versions standardized by the IEEE (1364-1995 and 1364-2001) had only simple constructs for creating tests. As design sizes outgrew the verification capabilities of the language, commercial Hardware Verification Languages (HVL) such as Open Vera and *e* were created. Companies that did not want to pay for these tools instead spent hundreds of man-years creating their own custom tools. This productivity crisis (along with a similar one on the design side) led to the creation of Accellera, a consortium of EDA

companies and users who wanted to create the next generation of Verilog. The donation of the Open-Vera language formed the basis for the HVL features of SystemVerilog.Accellera's goal was met in November 2005 with the adoption of the IEEE standard P1800-2005 for SystemVerilog, IEEE (2005).

The most valuable benefit of SystemVerilog is that it allows the user to construct reliable, repeatable verification environments, in a consistent syntax, that can be used across multiple projects

Some of the typical features of an HVL that distinguish it from a Hardware Description Language such as Verilog or VHDL are

- Constrained-random stimulus generation
- Functional coverage
- Higher-level structures, especially Object Oriented Programming
- Multi-threading and interprocess communication
- Support for HDL types such as Verilog's 4-state values
- Tight integration with event-simulator for control of the design

There are many other useful features, but these allow you to create test benches at a higher level of abstraction than you are able to achieve with an HDL or a programming language such as C.

System Verilog provides the best framework to achieve coverage-driven verification (CDV). CDV combines automatic test generation, self-checking testbenches, and coverage metrics to significantly reduce the time spent verifying a design. The purpose of CDV is to:

- Eliminate the effort and time spent creating hundreds of tests.
- Ensure thorough verification using up-front goal setting.
- Receive early error notifications and deploy run-time checking and error analysis to simplify debugging.

Examples

Ex1: A hello world program looks like this:

```
module main;
  initial
      begin
            $display("Hello
  world!");
            $finish;
      end
endmodule
```

Ex2: A simple example of two flip-flops follows:

```
module toplevel(clock,reset);
  input clock;
  input reset;

  reg flop1;
  reg flop2;

 always @ (posedge reset or posedge
  clock)
  if (reset)
            begin
             flop1 <= 0;
            flop2 <= 1;
            end
  else
            begin
            flop1 <= flop2;
             flop2 <= flop1;
            end
endmodule
```

The "<=" operator in Verilog is another aspect of its being a hardware description language as opposed to a normal procedural language. This is known as a "non-blocking" assignment. Its action doesn't register until the next clock cycle. This means that the order of the assignments are irrelevant and will produce the same result: flop1 and flop2 will swap values every clock.

The other assignment operator, "=", is referred to as a blocking assignment. When "=" assignment is used, for the purposes of logic, the target variable is

updated immediately. In the above example, had the statements used the "=" blocking operator instead of "<=", flop1 and flop2 would not have been swapped. Instead, as in traditional programming, the compiler would understand to simply set flop1 equal to flop2 (and subsequently ignore the redundant logic to set flop2 equal to flop1.)

Ex3: An example counter circuit follows:

```
module Div20x (rst, clk, cet, cep, count, tc);
// TITLE 'Divide-by-20 Counter with
   enables'
// enable CEP is a clock enable only
// enable CET is a clock enable and
// enables the TC output
// a counter using the Verilog language
parameter size = 5;
parameter length = 20;
input rst; // These inputs/outputs represent
input clk; // connections to the module.
input cet;
input cep;
output [size-1:0] count;
output tc;
reg [size-1:0] count; // Signals assigned
                      // within an always
                      // (or initial)block
                      // must be of type reg
 wire tc; // Other signals are of type wire
 // The always statement below is a parallel
// execution statement that
// executes any time the signals
// rst or clk transition from low to high
 always @ (posedge clk or posedge rst)
  if (rst) // This causes reset of the cntr
    count <= {size{1'b0}};
  else
  if (cet && cep) // Enables both  true
    begin
      if (count == length-1)
        count <= {size{1'b0}};
      else
        count <= count + 1'b1;
    end
 // the value of tc is continuously assigned
// the value of the expression
assign tc = (cet && (count == length-1));
endmodule
```

Ex4: An example of delays:...

```
reg a, b, c, d;
wire e;
...
always @(b or e)
  begin
    a = b & e;
    b = a | b;
    #5 c = b;
    d = #6 c ^ e;
  end
```

The always clause above illustrates the other type of method of use, i.e. the always clause executes any time any of the entities in the list change, i.e. the b or e change. When one of these changes, immediately a is assigned a new value, and due to the blocking assignment b is assigned a new value afterward (taking into account the new value of a.) After a delay of 5 time units, c is assigned the value of b and the value of c ^ e is tucked away in an invisible store. Then after 6 more time units, d is assigned the value that was tucked away.

Signals that are driven from within a process (an initial or always block) must be of type reg. Signals that are driven from outside a process must be of type wire. The keyword reg does not necessarily imply a hardware register.

7.3 Constants

The definition of constants in Verilog supports the addition of a width parameter. The basic syntax is:

<Width in bits>'<base letter><number>

Examples:

- 12'h123 - Hexadecimal 123 (using 12 bits)
- 20'd44 - Decimal 44 (using 20 bits - 0 extension is automatic)
- 4'b1010 - Binary 1010 (using 4 bits)
- 6'o77 - Octal 77 (using 6 bits)

International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 06 Issue 13
December 2019

7.4 Synthesizable Constructs

There are several statements in Verilog that have no analog in real hardware, e.g. $display. Consequently, much of the language can not be used to describe hardware. The examples presented here are the classic subset of the language that has a direct mapping to real gates.

```
// Mux examples - Three ways to do the same thing.
// The first example uses continuous assignment
wire out;
assign out = sel ? a : b;
// the second example uses a procedure
// to accomplish the same thing.
reg out;
always @(a or b or sel)
  begin
    case(sel)
      1'b0: out = b;
      1'b1: out = a;
    endcase
  end
// Finally - you can use if/else in a
// procedural structure.
reg out;
always @(a or b or sel)
  if (sel)
    out = a;
  else
    out = b;
```

The next interesting structure is a transparent latch; it will pass the input to the output when the gate signal is set for "pass-through", and captures the input and stores it upon transition of the gate signal to "hold". The output will remain stable regardless of the input signal while the gate is set to "hold". In the example below the "pass-through" level of the gate would be when the value of the if clause is true, i.e. gate = 1. This is read "if gate is true, the din is fed to latch_out continuously." Once the if clause is false, the last value at latch_out will remain and is independent of the value of din.

```
EX6: // Transparent latch example
 reg out;
always @(gate or din)
 if(gate)
   out = din; // Pass through state
   // Note that the else isn't required here. The variable
   // out will follow the value of din while gate is high.
   // When gate goes low, out will remain constant.
```

The flip-flop is the next significant template; in Verilog, the D-flop is the simplest, and it can be modeled as:

```
reg q;
always @(posedge clk)
  q <= d;
```

The significant thing to notice in the example is the use of the non-blocking assignment. A basic rule of thumb is to use <= when there is a **posedge** or **negedge** statement within the always clause.

A variant of the D-flop is one with an asynchronous reset; there is a convention that the reset state will be the first if clause within the statement.

```
reg q;
always @(posedge clk or posedge reset)
  if(reset)
    q <= 0;
  else
    q <= d;
```

The next variant is including both an asynchronous reset and asynchronous set condition; again the convention comes into play, i.e. the reset term is followed by the set term.

```
reg q;
always @(posedge clk or posedge reset or posedge set)
 if(reset)
   q <= 0;
 else
 if(set)
```

International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 06 Issue 13
December 2019

```
  q <= 1;
else
  q <= d;
```

Note: If this model is used to model a Set/Reset flip flop then simulation errors can result. Consider the following test sequence of events. 1) reset goes high 2) clk goes high 3) set goes high 4) clk goes high again 5) reset goes low followed by 6) set going low. Assume no setup and hold violations.

In this example the always @ statement would first execute when the rising edge of reset occurs which would place q to a value of 0. The next time the always block executes would be the rising edge of clk which again would keep q at a value of 0. The always block then executes when set goes high which because reset is high forces q to remain at 0. This condition may or may not be correct depending on the actual flip flop. However, this is not the main problem with this model. Notice that when reset goes low, that set is still high. In a real flip flop this will cause the output to go to a 1. However, in this model it will not occur because the always block is triggered by rising edges of set and reset - not levels. A different approach may be necessary for set/reset flip flops.

Note that there are no "initial" blocks mentioned in this description. There is a split between FPGA and ASIC synthesis tools on this structure. FPGA tools allow initial blocks where reg values are established instead of using a "reset" signal. ASIC synthesis tools don't support such a statement. The reason is that an FPGA's initial state is something that is downloaded into the memory tables of the FPGA. An ASIC is an actual hardware implementation.

7.5 Initial Vs Always:

There are two separate ways of declaring a Verilog process. These are the **always** and the **initial** keywords. The **always** keyword indicates a free-running process. The **initial** keyword indicates a process executes exactly once. Both constructs begin execution at simulator time 0, and both execute until the end of the block. Once an **always** block has reached its end, it is rescheduled (again). It is a common misconception to believe that an initial block will execute before an always block. In fact, it is better to think of the **initial**-block as a special-case of the **always**-block, one which terminates after it completes for the first time.

```
//Examples:
initial
  begin
    a = 1; // Assign a value to reg a at time 0
    #1; // Wait 1 time unit
    b = a; // Assign the value of reg a to reg b
  end

always @(a or b) // Any time a or b
    CHANGE, run the process
begin
  if (a)
    c = b;
  else
    d = ~b;
end // Done with this block, now return to
    the top (i.e. the @ event-control)

always @(posedge a)// Run whenever reg a
    has a low to high change
  a <= b;
```

These are the classic uses for these two keywords, but there are two significant additional uses. The most common of these is an **always**keyword without the **@(...)** sensitivity list. It is possible to use always as shown below:

```
always
 begin // Always begins executing at time 0
    and NEVER stops
   clk = 0; // Set clk to 0
   #1; // Wait for 1 time unit
   clk = 1; // Set clk to 1
   #1; // Wait 1 time unit
```

end *// Keeps executing - so continue back at the top of the begin*

The **always** keyword acts similar to the "C" construct **while(1) {..}** in the sense that it will execute forever.

The other interesting exception is the use of the **initial** keyword with the addition of the **forever** keyword.

7.6 Race Condition

The order of execution isn't always guaranteed within Verilog. This can best be illustrated by a classic example. Consider the code snippet below:

```
initial
  a = 0;
initial
  b = a;
initial
  begin
    #1;
    $display("Value a=%b Value of b=%b",a,b);
  end
```

What will be printed out for the values of a and b? Depending on the order of execution of the initial blocks, it could be zero and zero, or alternately zero and some other arbitrary uninitialized value. The $display statement will always execute after both assignment blocks have completed, due to the #1 delay.

System Tasks:

System tasks are available to handle simple I/O, and various design measurement functions. All system tasks are prefixed with $ to distinguish them from user tasks and functions. This section presents a short list of the most often used tasks. It is by no means a comprehensive list.

$display - Print to screen a line followed by an automatic newline.
$write - Write to screen a line without the newline.
$swrite - Print to variable a line without the newline.
$sscanf - Read from variable a format-specified string. (*Verilog-2001)
$fopen - Open a handle to a file (read or write)
$fdisplay - Write to file a line followed by an automatic newline.
$fwrite - Write to file a line without the newline.
$fscanf - Read from file a format-specified string. (*Verilog-2001)
$fclose - Close and release an open file handle.
$readmemh - Read hex file content into a memory array.
$readmemb - Read binary file content into a memory array.
$monitor - Print out all the listed variables when any change value.
$time - Value of current simulation time.
$dumpfile - Declare the VCD (Value Change Dump) format output file name.
$dumpvars - Turn on and dump the variables.
$dumpports - Turn on and dump the variables in Extended-VCD format.
$random - Return a random value.

CONCLUSION

High Performance MAC(Multiplier Accumulator Unit) Is Designed With High Speed, Low power , Less Delay By reducing the number of gates By using Wallace tree algorithm to implement the multiplier and also we are using Carry save adder to implement the adder with less number of gates. This implemented By using Verilog HDL

REFERENCES

1. Baugh, C.R. and B.A. Wooley, 1973. A two's complement parallel array multiplication algorithm. IEEE Trans. Comput., C-22: 1045-1047.
2. Berkeman, A., V. Owall and M. Torkelson, 2000. A low logic depth complex multiplier using distributed arithmetic. IEEE J. Solid-State Circ., 35: 656-659.

p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 06 Issue 13
December 2019

3. Brent, R.P. and H.T. Kung, 1982. A regular layout for parallel adders. IEEE Trans. Comput., C-31: 260-264.
4. Chang, J.K., H. Lee and C.S. Choi, 2009. A power-aware variable-precision multiply-accumulate unit. Proceedings of the 9th International Symposium on Communications and Information Technology, September 28-30, 2009, Icheon, pp: 1336-1339.
5. Chang, T.Y. and M.J. Hsiao, 1998. Carry-select adder using single ripple-carry adder. Electron. Lett., 34: 2101-2103.