# ASIC implementation of TMS320C2X DSP

**Thesis** · June 2014

CITATIONS
0

READS
1,376

**1 author:**

Mohamed Omran
The American University in Cairo
**3** PUBLICATIONS **0** CITATIONS

SEE PROFILE

# ASIC implementation of TMS320C2X DSP As a second phase of CORDOBA DSP

BSc. Graduation Project
July 2014

Prof.Dr.Mohamed Rizk, Alexandria University
Prof.Dr.Amr Wassal, Cairo University

Communication and Electronics Department,
Faculty of Engineering, Alexandria University

# Acknowledgments

First, we thank **Allah** for our achievements through all the phases of the projects.

We would like to express our great appreciation to **Dr. Amr G. Wassal** and **Dr.Mohamed Rizk** for their great support and their helpful during the Project

We would also like to thank **Eng.Amr Abdelaziz Fathi , Eng.Islam Alaa El-din Mostafa , Eng.Kholoud Rabie El-garhy and Eng.Mohamed Ahmed Abdelhameed** ,Andalus project Team for explaining their Project and for their Guidance in the project

We would also like to thank **Eng.Fady , Eng.Mostafa Sakr and Eng.Islam Mostafa** Si-Ware Engineers for their great advice and assistance Through the Project

Finally, we would like to thank **Si-Ware systems** for This Chance to Work with them

**Authors**

**Esraa Ramdan**
01283631383, esraa-ramdan222@yahoo.com

**Fatma Bahy**
01285006459, fatmabahy@gmail.com

**Hany Salah Gamal**
01145570766, hanygamaleldiin@gmail.com

**Khalid Essam El-Sayed**
01061462719, k.e.elsayed@ieee.org

**Mahomud Mohamed Refaat**
010249109810, mahmoud.ref3at@gmail.com

**Mohammed Abd El-Fatah Ewais**
01147954640, mohammad.a.ewais@gmail.com

**Mohammed Ali Omran**
01285089981, omranooo66@gmail.com

**Mohammed Sami**
01273308442, mohamedsami395@gmail.com

**Moustafa Emara**
01145840256, m.s.emara@ieee.org

**Samah Saad Desouky**
01222589241, samah_saad@ymail.com

**Yahya Beltagy**
01113375462, ybeltagy@gmail.com

This page intentionally left blank.

# Contents

This page intentionally left blank.

# List of Figures

# Chapter 1

# Introduction

## 1.1 Andalus-DSP

Andalus-DSP is a series of graduation projects working on the implementation of complete digital signal processor with its software layer, to develop the First Egyptian Digital Signal Processor that is able to adapt with todays digital signal processing challenges. Last year was cordoba phase and this year is toledo phase.

## 1.2 Cordoba phase

In Cordoba phase they were working on the main design and the implementation of synthesizable RTL code of the DSP core, testing it on FPGA and implementing a simple assembler for the instruction set of the DSP.

## 1.3 Toledo Phase

In toledo phase we have worked on RTL design outed from cordoba phase, we followed ASIC design flow, our work was divided into 3 main phases:

1. verification (for RTL code and for layout)
2. Back-end Design (Physical Design)
3. Optimization of RTL design

## 1.4 Book overview

### Chapter 2:

introduces the reader to the basic concepts of digital signal processor, architecture and its applications then gives him background about ASIC design flow and EDA tools that are used in this work.

### Chapter 3:

Architecture of The Toledo DSP

### Chapter 4:

talks about the layout of the DSP and the processes made to implement the Andalus DSP

### Chapter 5:

talks about the verification processes made on the Andalus DSP

# Chapter 2

# Scientific background

## 2.1 DSP

### 2.1.1 What is a DSP?

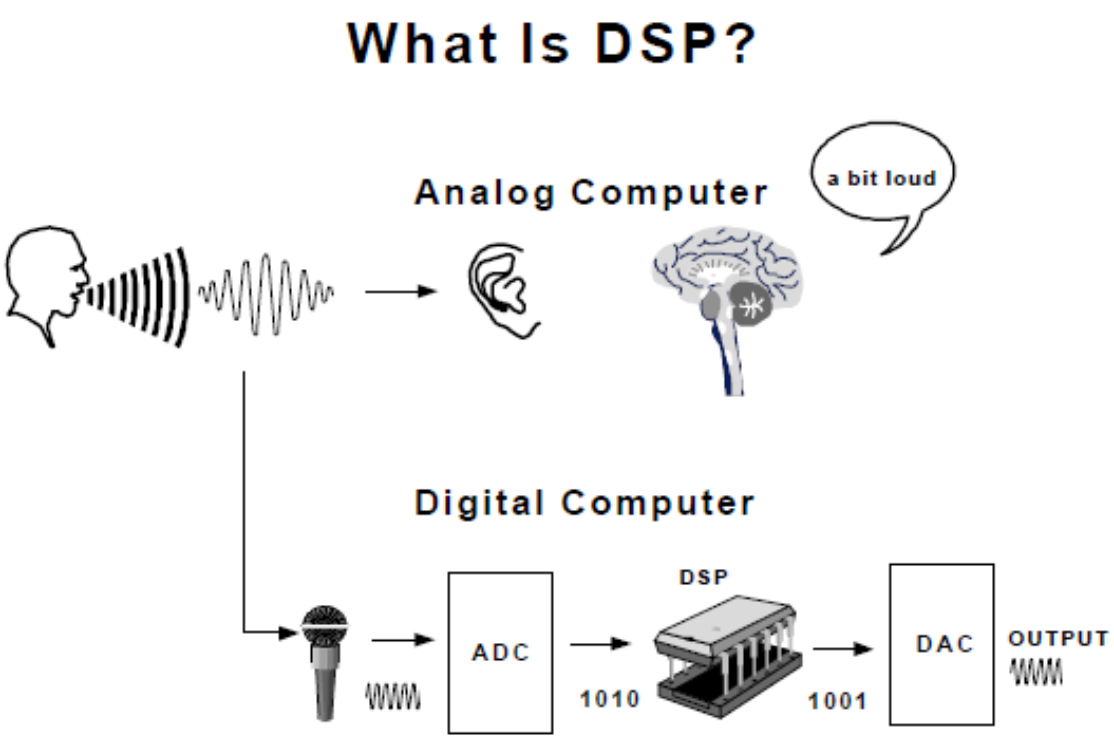



Figure 2.1: what is a DSP?

[3] A Digital Signal Processor, or DSP, is a specialized microprocessor that has an architecture which is optimized for the fast operational needs of digital signal processing. A Digital Signal Processor (DSP) can process data in real time, making it ideal for applications that cant tolerate delays. Digital signal processors take a digital signal and process it to improve the signal into clearer sound, faster data or sharper images. Digital Signal Processors use video, voice, audio, temperature or position signals that have been digitized and mathematically manipulate them. A digital signal processor is designed to perform these mathematical functions rapidly. The signals are processed so the information contained in them can be displayed or converted to another type of signal. DSPs are used in many applications such as communication systems, aerospace, biometrics, military, automation and industrial control.

### 2.1.2 DSP evolution and current scenery

DSPs appeared on the market in the early 1980s. Since then, they have undergone an intense evolution in terms of hardware features, integration, and software development tools. DSPs are now a mature technology. This section gives an overview of the evolution of the DSP over their 25-year life span [1].

## DSP evolution:

- **hardware features:**

  In the late 1970s there were many chips aimed at digital signal processing; however, they are not considered to be digital signal processing owing to either their limited programmability or their lack of hardware features such as hardware multipliers. The first marketed chip to qualify as a programmable DSP was NECs MPD7720, in 1981: it had a hardware multiplier and adopted the Harvard architecture (more information on this architecture is given in Section 2.1.3). Another early DSP was the TMS320C10, marketed by TI in 1982. From a market evolution viewpoint, we can divide the two and a half decades of DSP life span into two phases: a development phase, which lasted until the early 1990s, and a consolidation phase, lasting until now. Figure 2.2 gives an overview of the evolution of DSP features together with the first year of marketing for some DSP families.

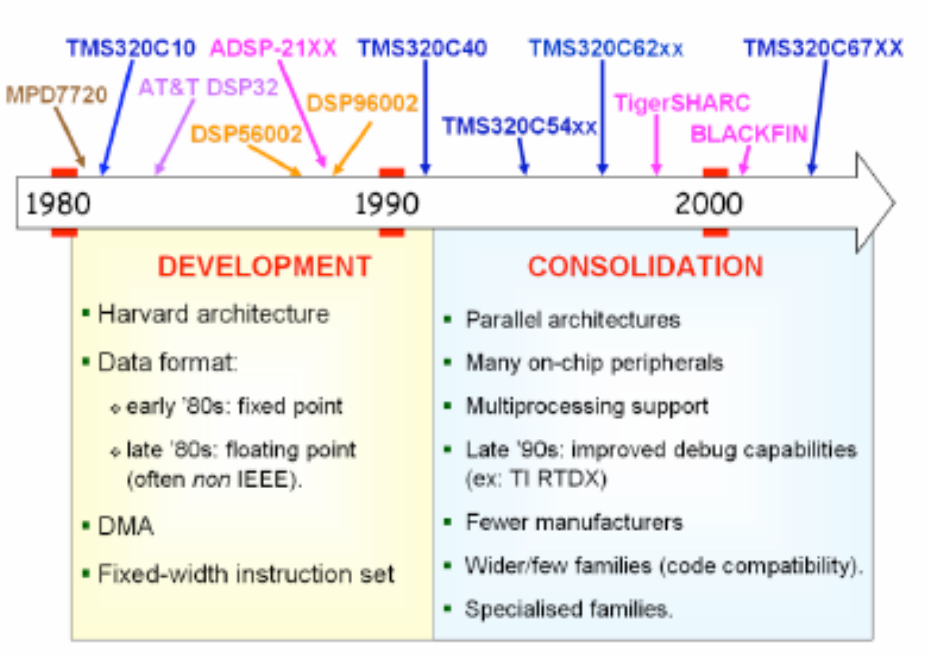



Figure 2.2: Evolution of DSP features from their early days until now

During the market development phase, DSPs were typically based upon the Harvard architecture. The first generation of DSPs included multiply, add, and accumulator units. Examples are TIs TMS320C10 and Analog Devices (ADI) ADSP-2101. The second generation of DSPs retained the architectural structure of the first generation but added features such as pipelining, multiple arithmetic units, special address generator units, and Direct Memory Access (DMA). Examples include TIs TMS320C20 and Motorolas DSP56002. While the first DSPs were capable of fixedpoint operations only, towards the end of the 1980s DSPs with floating point capabilities started to appear. Examples are Motorolas DSP96001 and TIs TMS320C30. It should be noted that the floating-point format was not always IEEE-compatible. For instance, the TMS320C30 internal calculations were carried out in a proprietary format; a hardware chip converter was available to convert to the standard IEEE format. DSPs belonging to the development phase were characterized by fixed-width instruction sets, where one of each instruction was executed per clock cycle. These instructions could be complex, and encompassing several operations. The width of the instruction was typically quite short and did not overcome the DSP native word width. As for DSP producers, the market was nearly equally shared between many manufactur-

ers such as AT & T, Fujitsu, Hitachi, IBM, NEC, Toshiba, Texas Instruments and, towards the end of the 1980s, Motorola, Analog Devices and Zoran.

During the market consolidation phase, enhanced DSP architectures such as Very Long Instruction Word (VLIW) and Single Instruction Multiple Data (SIMD) emerged. These architectures increase the DSP performance through parallelism. Examples of DSPs with enhanced architectures are TIs TMS320C6xxx DSPs, which was the first DSP to implement the VLIW architecture, and ADIs TigerSHARC, that includes both VLIW and SIMD features. The number of on-chip peripherals increased greatly during this phase, as well as the hardware features that allow many processors to work together. Technologies that allow real-time data exchange between host processor and DSP started to appear towards the end of the 1990s. This constituted a real sea change in DSP system debugging and helped the developers enormously. Another phenomenon observed during this phase was the reduction of the number of DSP manufacturers. The number of DSP families was also greatly reduced, in favour of wider families that granted increased code compatibility between DSPs of different generations belonging to the same family. Additionally, many DSP families are not general purpose but are focused on specific digital signal processing applications, such as audio equipment or control loops.

- **device integration:**

  Wafer, die, and feature sizes are the basic key factors that define a chip technology. The wafer size is the diameter of the wafer used in the semiconductor manufacturing process. The die size is the size of the actual chips carved up in a wafer. The feature size is the size of the smallest circuit component (typically a transistor) that can be etched on a wafer; this is used as an overall indicator of the density of an Integrated Circuit (IC) fabrication process. The trend in industry is to go towards larger wafers and chip dies, so as to increase the number of working chips that can be obtained from the same wafer; also called yield. For instance, the current typical wafer size is 12 inches (300 mm), and some leading chip maker companies plan to move to 18 inches (450 mm) within the first half of the next decade. (It should be added that the issue is somewhat controversial, as many equipment manufacturers fear that the 18 inches wafer size will lead to scale problems even worse than for the 12 inches.) Feature size is decreasing, allowing one to either have more functionality on a die or to reduce the die size while keeping the same functionality. Transistors with smaller sizes require less voltage to drive them; this results in a decrease of the core voltage from 5 V to 1.5 V.

The I/O voltage has been lowered as well, with the caveat that it remains compatible with the external devices used and their standard. A lower core voltage has been one of the key factors enabling higher clock frequencies: in fact, the gap between high and low state thresholds is tightened thus allowing a faster logic level transition. Additionally, the reduced die size and lowered core voltage allow lower power consumption, an important factor for portable or mobile system. Finally, the global cost of a chip has decreased by at least a factor 30 over the last 25 years.

The trend towards a faster switching hardware (including chip over-clocking) and smaller feature size carries the benefit of increased processing power and throughput. There is a downside to it, however, represented by the electromigration phenomenon. Electromigration occurs when some of the momentum of a moving electron is transferred to a nearby activated ion, hence causing the ion to move from its original position. Gaps or, on the contrary, unintended electrical connections can develop with time in the conducting material if a significant number of atoms are moved far from their original position. The consequence is the electrical failure of the electronic interconnects and the consequent shortened chip lifetime.

- **software tools:**

The improvement of DSP software tools from the early days until now has been spectacular. Code compilers have evolved greatly to be able to deal with the underlying hardware complexity and the enhanced DSP architectures. At the same time, they allow the developer to program more and more efficiently in high-level languages as opposed to assembly coding. This speeds up considerably the code development time and makes the code itself more portable across different platforms.

Advanced tools now allow the programming of DSPs graphically, i.e., by interconnecting pre-defined blocks that are then converted to DSP code. Examples of these tools are MATLAB Code Generation and embedded target products and National Instruments' LabVIEW DSP Module.

High-performance simulators, emulator and debugging facilities allow the developer to have a high visibility into the DSP with little or no interference on the program execution. Additionally, multiple DSPs can be accessed in the same JTAG chain for both code development and debugging.

### DSP current scenery:

The number of DSP vendors is currently somewhat limited: Analog Devices (ADI), Freescale (formerly Motorola), Texas Instruments (TI), Renesas, Microchip and VeriSilicon are the basic players. Amongst them, the biggest share of the market is taken by only three vendors, namely ADI, TI and Freescale. Table **??** lists the main DSP families for ADI and TI DSPs, together with their typical use and performance.

| Manufacturer | Family | Typical use and performance |
|---|---|---|
| TI | TMS320C2x | Digital signal controllers |
| | TMS320C5x | Power efficient |
| | TMS320C6x | High performance |
| ADI | SHARC | Medium performance. First ADI family (now three generations) |
| | TigerSHARC | High performance for multi-processor systems |
| | Blackfin | High performance and low power |

Figure 2.3: Main ADI and TI DSP families.

## 2.1.3 DSP architecture

One of the biggest bottlenecks in executing DSP algorithms is transferring information to and from memory [7]. This includes data, such as samples from the input signal and the filter coefficients, as well as program instructions, the binary codes that go into the program sequencer. For example, suppose we need to multiply two numbers that reside somewhere in memory. To do this, we must fetch three binary values from memory, the numbers to be multiplied, plus the program instruction describing what to do.

Figure 2.4 shows how this seemingly simple task is done in a traditional microprocessor. This is often called a Von Neumann architecture, after the brilliant American mathematician John Von Neumann (1903-1957).
Von Neumann guided the mathematics of many important discoveries of the early twentieth century. His many achievements include: developing the concept of a stored program computer, formalizing the mathematics of quantum mechanics, and work on the atomic bomb. If it was new and exciting, Von Neumann was there!

As shown in (a), a Von Neumann architecture contains a single memory and a single bus for transferring data into and out of the central processing unit (CPU). Multiplying two numbers requires at least three clock cycles, one to transfer each

of the three numbers over the bus from the memory to the CPU. We don't count the time to transfer the result back to memory, because we assume that it remains in the CPU for additional manipulation (such as the sum of products in an FIR filter).
The Von Neumann design is quite satisfactory when you are content to execute all of the required tasks in serial. In fact, most computers today are of the Von Neumann design. We only need other architectures when very fast processing is required, and we are willing to pay the price of increased complexity.

This leads us to the Harvard architecture, shown in (b). This is named for the work done at Harvard University in the 1940s under the leadership of Howard Aiken (1900-1973).
As shown in this illustration, Aiken insisted on separate memories for data and program instructions, with separate buses for each. Since the buses operate independently, program instructions and data can be fetched at the same time, improving the speed over the single bus design. Most present day DSPs use this dual bus architecture.

Figure (c) illustrates the next level of sophistication, the Super Harvard Architecture. This term was coined by Analog Devices to describe the internal operation of their ADSP-2106x and new ADSP-211xx families of Digital Signal Processors. These are called SHARC DSPs, a contraction of the longer term, Super Harvard Architecture. The idea is to build upon the Harvard architecture by adding features to improve the throughput.
While the SHARC DSPs are optimized in dozens of ways, two areas are important enough to be included in Fig. 2.4 c: an instruction cache, and an I/O controller.

First, let's look at how the instruction cache improves the performance of the Harvard architecture. A handicap of the basic Harvard design is that the data memory bus is busier than the program memory bus. When two numbers are multiplied, two binary values (the numbers) must be passed over the data memory bus, while only one binary value (the program instruction) is passed over the program memory bus.
To improve upon this situation, we start by relocating part of the "data" to program memory. For instance, we might place the filter coefficients in program memory, while keeping the input signal in data memory. (This relocated data is called "secondary data" in the illustration).
At first glance, this doesn't seem to help the situation; now we must transfer one value over the data memory bus (the input signal sample), but two values over the program memory bus (the program instruction and the coefficient). In fact, if we

were executing random instructions, this situation would be no better at all. However, DSP algorithms generally spend most of their execution time in loops. This means that the same set of program instructions will continually pass from program memory to the CPU. The Super Harvard architecture takes advantage of this situation by including an instruction cache in the CPU. This is a small memory that contains about 32 of the most recent program instructions. The first time through a loop, the program instructions must be passed over the program memory bus. This results in slower operation because of the conflict with the coefficients that must also be fetched along this path. However, on additional executions of the loop, the program instructions can be pulled from the instruction cache. This means that all of the memory to CPU information transfers can be accomplished in a single cycle: the sample from the input signal comes over the data memory bus, the coefficient comes over the program memory bus, and the program instruction comes from the instruction cache. In the jargon of the field, this efficient transfer of data is called a high memory access bandwidth.

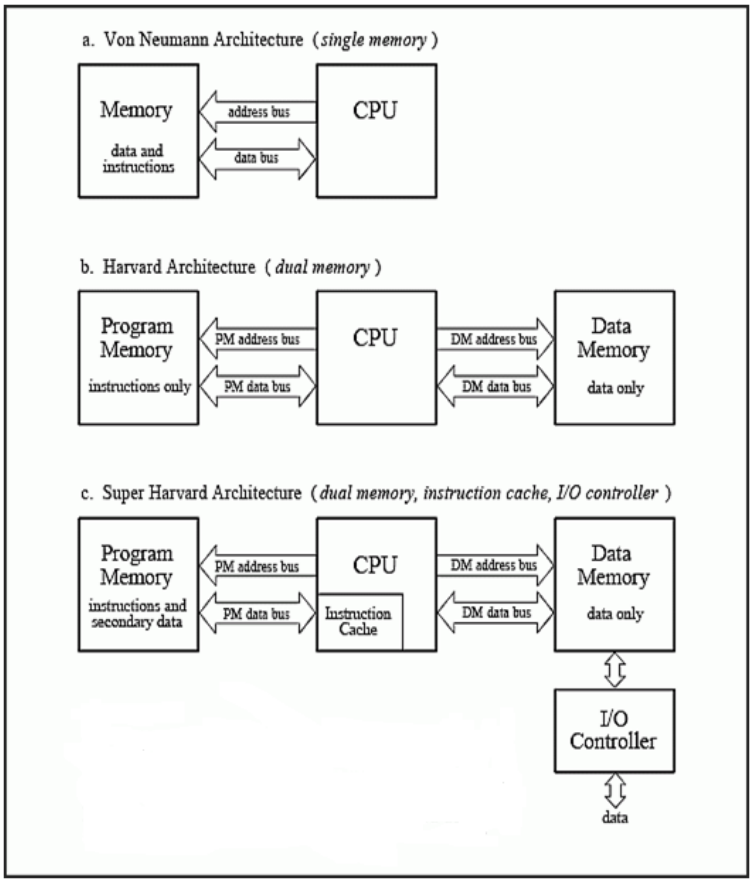



Figure 2.4: Microprocessors architectures.

Figure 2.5 presents a more detailed view of the SHARC architecture, showing the I/O controller connected to data memory. This is how the signals enter and exit the system. For instance, the SHARC DSPs provides both serial and parallel communications ports. These are extremely high speed connections.
For example, at a 40 MHz clock speed, there are two serial ports that operate at 40 Mbits/second each, while six parallel ports each provide a 40 Mbytes/second data transfer. When all six parallel ports are used together, the data transfer rate is an incredible 240 Mbytes/second.
This type of high speed I/O is a key characteristic of DSPs. The overriding goal is to move the data in, perform the math, and move the data out before the next sample is available. Everything else is secondary. Some DSPs have onboard analog-to-digital and digital-to-analog converters, a feature called mixed signal. However, all DSPs can interface with external converters through serial or parallel ports. Now let's look inside the CPU. At the top of the diagram are two blocks labeled Data Address Generator (DAG), one for each of the two memories. These control the addresses sent to the program and data memories, specifying where the information is to be read from or written to. In simpler microprocessors this task is handled as an inherent part of the program sequencer, and is quite transparent to the programmer. However, DSPs are designed to operate with circular buffers, and benefit from the extra hardware to manage them efficiently. This avoids needing to use precious CPU clock cycles to keep track of how the data are stored. For instance, in the SHARC DSPs, each of the two DAGs can control eight circular buffers. This means that each DAG holds 32 variables (4 per buffer), plus the required logic. What is a circular buffer? Circular buffers are limited memory regions where data are stored in a First-In First-Out (FIFO) way; these memory regions are managed in a wrap-around way, i.e., the last memory location is followed by the first memory location. Two sets of pointers are used, one for reading and one for writing; the length of the step at which successive memory locations are accessed is called stride. Address generator units allow striding through the circular buffers without requiring dedicated instructions to determine where to access the following memory location, error detection and so on. Circular buffers allow storing bursts or continuous streams of data and processing them in the order in which they have arrived.

Why so many circular buffers? Some DSP algorithms are best carried out in stages. For instance, IIR filters are more stable if implemented as a cascade of biquads (a stage containing two poles and up to two zeros). Multiple stages require multiple circular buffers for the fastest operation. The DAGs in the SHARC DSPs are also designed to efficiently carry out the Fast Fourier transform. In this mode, the DAGs are configured to generate bit-reversed addresses into the circu-

lar buffers, a necessary part of the FFT algorithm. In addition, an abundance of circular buffers greatly simplifies DSP code generation- both for the human programmer as well as high-level language compilers, such as C.
The data register section of the CPU is used in the same way as in traditional microprocessors. In the ADSP-2106x SHARC DSPs, there are 16 general purpose registers of 40 bits each. These can hold intermediate calculations, prepare data for the math processor, serve as a buffer for data transfer, hold flags for program control, and so on. If needed, these registers can also be used to control loops and counters; however, the SHARC DSPs have extra hardware registers to carry out many of these functions.
The math processing is broken into three sections, a multiplier, an arithmetic logic unit (ALU), and a barrel shifter. The multiplier takes the values from two registers, multiplies them, and places the result into another register. The ALU performs addition, subtraction, absolute value, logical operations (AND, OR, XOR, NOT), conversion between fixed and floating point formats, and similar functions. Elementary binary operations are carried out by the barrel shifter, such as shifting, rotating, extracting and depositing segments, and so on. A powerful feature of the SHARC family is that the multiplier and the ALU can be accessed in parallel. In a single clock cycle, data from registers 0-7 can be passed to the multiplier, data from registers 8-15 can be passed to the ALU, and the two results returned to any of the 16 registers. The DMA controller is a second processor working in parallel with the DSP core and dedicated to transferring information between two memory areas or between peripherals and memory. In doing so the DMA controller frees the DSP core for other processing tasks.
A DMA coprocessor can transfer data as well as program instructions, the latter transfer corresponding typically to the case of code overlay, i.e., of code stored in an external memory and moved to an internal memory (for instance L1) when needed. Multiple and independent DMA channels are also available for greater flexibility. Bus arbitration between the DMA and the DSP core is needed to avoid colliding memory accesses when the DMA and the DSP core share the same bus to access peripherals and/or memories. To prevent bottlenecks, recent DSPs typically fit DMA controllers with dedicated buses.

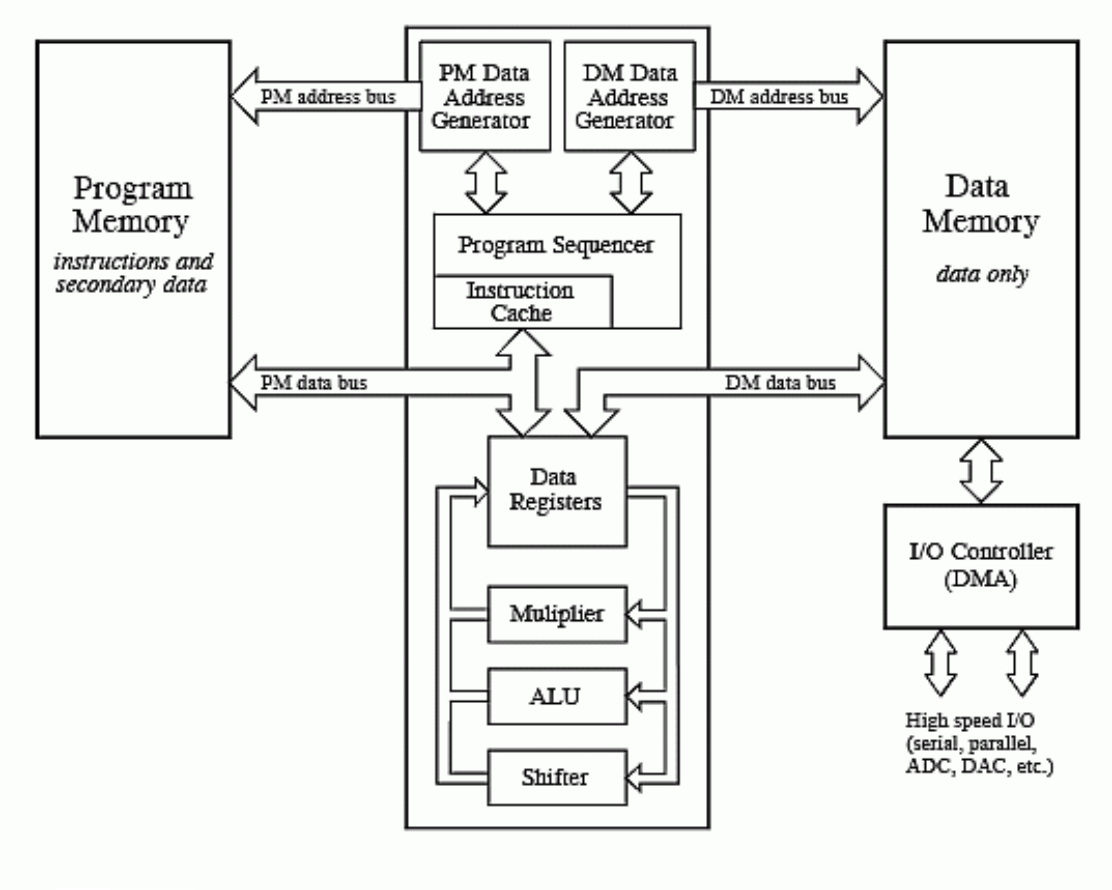



Figure 2.5: Simplified diagram of the Analog Devices SHARC DSP.

### 2.1.4 DSP applications

| Field | | Application |
|---|---|---|
| **Communication** | Broadband | Video conferencing / phone |
| | | Voice / multimedia over IP |
| | | Digital media gateways (VOD) |
| | Wireless | Satellite phone |
| | | Base station |
| **Consumer** | Security | Biometrics |
| | | Video surveillance |
| | Entertainment | Digital still /video camera |
| | | Digital radio |
| | | Portable media player / entertainment console |
| | Toys | Interactive toys |
| | | Video game console |
| **Industrial and entertainment** | Medical | MRI |
| | | Ultrasound |
| | | X-ray |
| | Point of sale | Scanner |
| | | Vending machine |
| | Industrial | Factory automation |
| | | Industrial / machine / motor control |
| | | Vision system |
| **Military and aerospace** | | Guidance (radar, sonar) |
| | | Avionics |
| | | Digital radio |
| | | Smart munitions, target detection |

Figure 2.6: A short selection of DSP fields of use and specific applications

DSPs appeared on the market in the early 1980s. Over the last 15 years they have been the key enabling technology for many electronics products in fields such as communication systems, multimedia, automotive, instrumentation and military.

Figure 2.6 gives an overview of some of these fields and of the corresponding typical DSP applications.

Figure 2.7 shows a real-life DSP application, namely the use of a Texas Instruments (TI) DSP in an Active noise cancellation (ANC) which implements the acoustically adaptive algorithm that cancels the unwanted sound by generating an antisound (antinoise) of equal amplitude and opposite phase. The original, unwanted sound and the antinoise acoustically combine, resulting in the cancellation of both sounds.

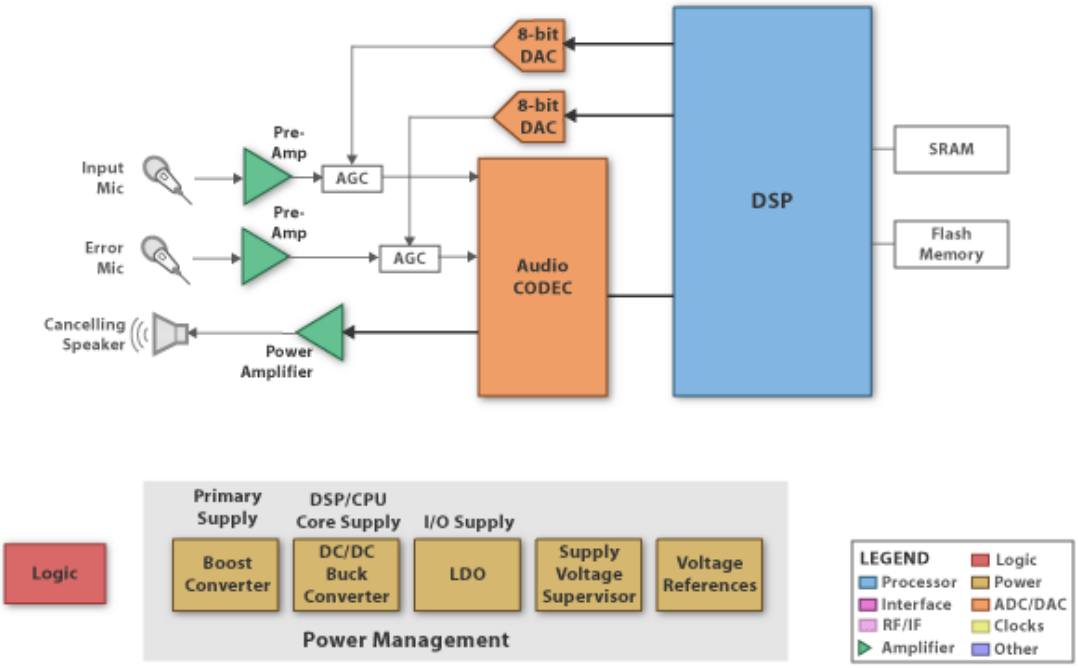



Figure 2.7: Use of Texas Instruments DSP in an ANC system.

## 2.2 ASIC Flow

### 2.2.1 Introduction

Any IC other than a general purpose IC which contain the functionality of- thousands of gates is usually called an ASIC (Application Specific Integrated Circuit). ASICs are designed to fit a certain application.
An ASIC is a digital or mixed-signal circuit designed to meet specifications set by

a specific project

- In integrated circuit design, physical design is a step in the standard design cycle which follows after the circuit design. At this step, circuit representations of the components (devices and interconnects) of the design are converted into geometric representations of shapes which, when manufactured in the corresponding layers of materials, will ensure the required functioning of the components. This geometric representation is called integrated circuit layout. This step is usually split into several sub-steps, which include both design and verification and validation of the layout.

- Modern day Integrated Circuit (IC) design is split up into Front-end design using HDL's, Verification and Back-end Design or Physical Design. The next step after Physical Design is the Manufacturing process or Fabrication Process that is done in the Wafer Fabrication Houses. Fab-houses fabricate designs onto silicon dies which are then packaged into ICs.

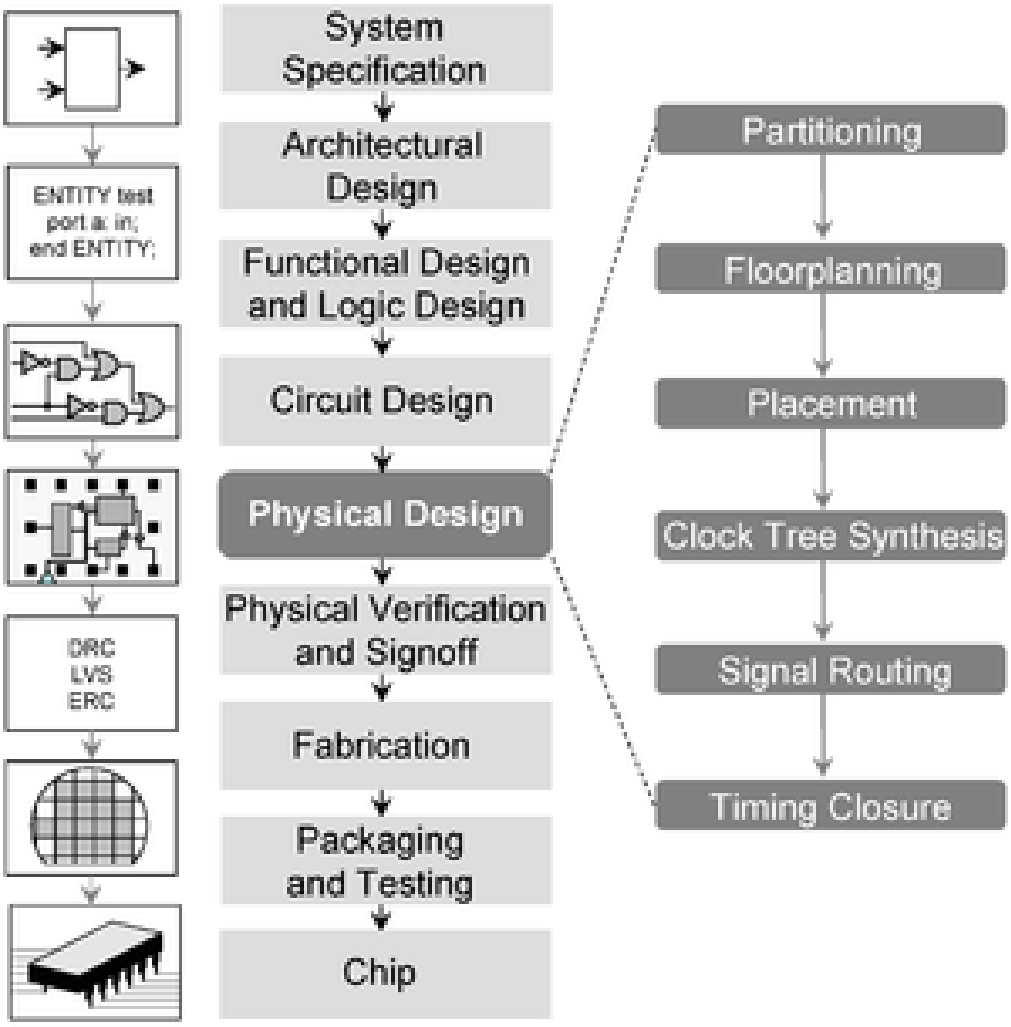
- Each of the phases mentioned above have Design Flows associated with them. These Design Flows lay down the process and guide-lines/framework for that phase. Physical Design flow uses the technology libraries that are provided by the fabrication houses. These technology files provide information regarding the type of Silicon wafer used, the standard-cells used, the layout rules (like DRC in VLSI), etc.

- Typically, the IC physical design is categorised into Full custom & Semi-Custom Design.

1. Full-Custom: Designer has full flexibility on the layout design, no predefined cells are used.

2. Semi-Custom: Pre-designed library cells (preferably tested with DFM) are used; designer has flexibility in placement of the cells & routing.

One can refer ASIC for Full Custom design and FPGA for Semi-Custom design flows.The reason being that one have the flexibility to design/modify design blocks from Vendor provided libraries in ASIC. This flexibility is missing for Semi-Custom flows like FPGA (eg. Altera).




### 2.2.2 Generalized ASIC Design Flow

**High Level Design:**

- Specification Capture
- Design Capture in c, c++, system c, system verilog .
- SW/HW partitioning and ip selection.

**RTL design:**

Verilog/VHDL

**System Timing and Logic Verification:**

- Is the logic working correctly?

**Physical Design:**

Floorplanning, Place and Route, Clock insertion.

**Performance and Manufacturability Verification:**

- Extraction of Physical View
- Verification of timing and signal integrity
- Design Rule Checking/ LVS

In our project , we concentrate on verification and physical design . this part will give you brief explanation about them .

**Logic Design and Verification**

**Design starts with a specification:**

- Text description or system specification language
- Example: C , SystemC , SystemVerilog

**RTL Description:**

- contentAutomated conversion from system specification to RTL possible
- Example: Cadence C-to-Silicon Compiler
- Most often designer manually converts to Verilog or VHDL

**Verification:**

- Generate test-benches and run simulations to verify functionality
- Assertion based verification
- Automated test-bench generation

**RTL Synthesis and Verification**

**RTL Synthesis**

- Automated generation of generic gate description from RTL description
- Logic optimization for speed and area
- State machine decomposition, datapath optimization, power optimization
- Modern tools integrate global place-and-route capabilities

**Library Mapping**

Translates a generic gate level description to a netlist using a target library

**Functional or Formal Verification**

- HDL ambiguities can cause the synthesis tool to produce incorrect netlist
- Rerun functional verification on the gate level netlist
- Formal verification

**Model checking**

- prove that certain assertions are true
- Equivalence checking: compare two design descriptions

### 2.2.3 ASIC physical design flow

The main steps in the ASIC physical design flow are:

- Design Netlist (after synthesis)
- Floorplanning
- Partitioning
- Placement
- Clock-tree Synthesis (CTS)
- Routing
- Physical Verification

- GDSII Generation

These steps are just the basic. There are detailed PD flows that are used depending on the Tools used and the methodology/technology.

Some of the tools/software used in the back-end design are:

- Cadence (Cadence Encounter RTL Compiler, Encounter Digital Implementation, Cadence Voltus IC Power Integrity Solution, Cadence Tempus Timing Signoff Solution)
- Synopsys (Design Compiler, IC Compiler)
- Magma (BlastFusion, etc.)
- Mentor Graphics (Olympus SoC, IC-Station, Calibre)

A more detailed Physical Design Flow is shown in figure 2.8.
Here you can see the exact steps and the tools used in each step outlined.

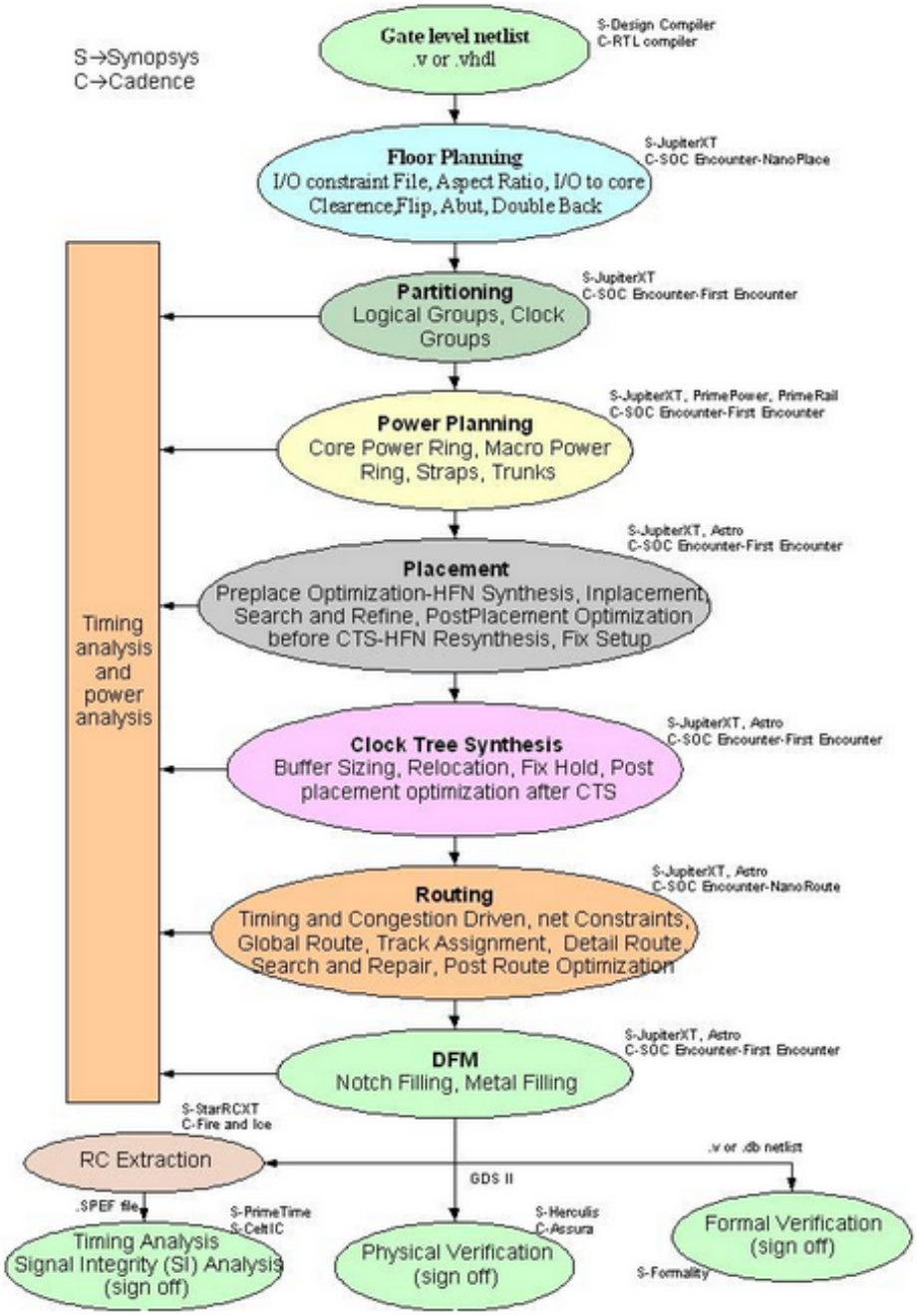



Figure 2.8: Physical Design Flow

The ASIC physical design flow uses the technology libraries that are provided by the fabrication houses. Technologies are commonly classified according to minimal feature size. Standard sizes, in the order of miniaturization, are 2m, 1m , 0.5m , 0.35m, 0.25m, 180nm, 130nm, 90nm, 65nm, 45nm, 28nm, 22nm, 18nm, 14nm, etc. They may be also classified according to major manufacturing approaches: n-Well process, twin-well process, SOI process, etc.

### Design netlist

Physical design is based on a netlist which is the end result of the Synthesis process. Synthesis converts the RTL design usually coded in VHDL or Verilog HDL to gate-level descriptions which the next set of tools can read/understand. This netlist contains information on the cells used, their interconnections, area used, and other details.

Typical synthesis tools are:

- Cadence RTL Compiler/Build Gates/Physically Knowledgeable Synthesis (PKS)
- Synopsys Design Compiler

During the synthesis process, constraints are applied to ensure that the design meets the required functionality and speed (specifications). Only after the netlist is verified for functionality and timing it is sent for the physical design flow.

### Physical Design Steps

1. **Floorplanning**

   The first step in the physical design flow is Floorplanning. Floorplanning is the process of identifying structures that should be placed close together, and allocating space for them in such a manner as to meet the sometimes conflicting goals of available space (cost of the chip), required performance, and the desire to have everything close to everything else.
   Based on the area of the design and the hierarchy, a suitable floorplan is decided upon. Floorplanning takes into account the macros used in the design, memory, other IP cores and their placement needs, the routing possibilities and also the area of the entire design. Floorplanning also decides the IO structure, aspect ratio of the design. A bad floorplan will lead to waste-age of die area and routing congestion.
   In many design methodologies, Area and Speed are considered to be things

that should be traded off against each other. The reason this is so is probably because there are limited routing resources, and the more routing resources that are used, the slower the design will operate. Optimizing for minimum area allows the design to use fewer resources, but also allows the sections of the design to be closer together. This leads to shorter interconnect distances, less routing resources to be used, faster end-to-end signal paths, and even faster and more consistent place and route times. Done correctly, there are no negatives to floorplanning.
As a general rule, data-path sections benefit most from floorplanning, and random logic, state machines and other non-structured logic can safely be left to the placer section of the place and route software.
Data paths are typically the areas of your design where multiple bits are processed in parallel with each bit being modified the same way with maybe some influence from adjacent bits. Example structures that make up data paths are Adders, Subtractors, Counters, Registers, and Muxes.

2. **Partitioning**

   Partitioning is a process of dividing the chip into small blocks. This is done mainly to separate different functional blocks and also to make placement and routing easier.
   Partitioning can be done in the RTL design phase when the design engineer partitions the entire design into sub-blocks and then proceeds to design each module. These modules are linked together in the main module called the TOP LEVEL module. This kind of partitioning is commonly referred to as Logical Partitioning.

3. **Placement**

   Before the start of placement optimization all Wire Load Models (WLM) are removed. Placement uses RC values from Virtual Route (VR) to calculate timing. VR is the shortest Manhattan distance between two pins. VR RCs are more accurate than WLM RCs.

   Placement is performed in four optimization phases:

   (a) Pre-placement optimization

   (b) In placement optimization

   (c) Post Placement Optimization (PPO) before clock tree synthesis (CTS)

   (d) PPO after CTS.

- Pre-placement Optimization optimizes the netlist before placement, HFNs are collapsed. It can also downsize the cells.
- In-placement optimization re-optimizes the logic based on VR. This can perform cell sizing, cell moving, cell bypassing, net splitting, gate duplication, buffer insertion, area recovery. Optimization performs iteration of setup fixing, incremental timing and congestion driven placement.
- Post placement optimization before CTS performs netlist optimization with ideal clocks. It can fix setup, hold, max trans/cap violations. It can do placement optimization based on global routing. It re does HFN synthesis.
- Post placement optimization after CTS optimizes timing with propagated clock. It tries to preserve clock skew.

4. **Clock tree synthesis**
   The goal of clock tree synthesis (CTS) is to minimize skew and insertion

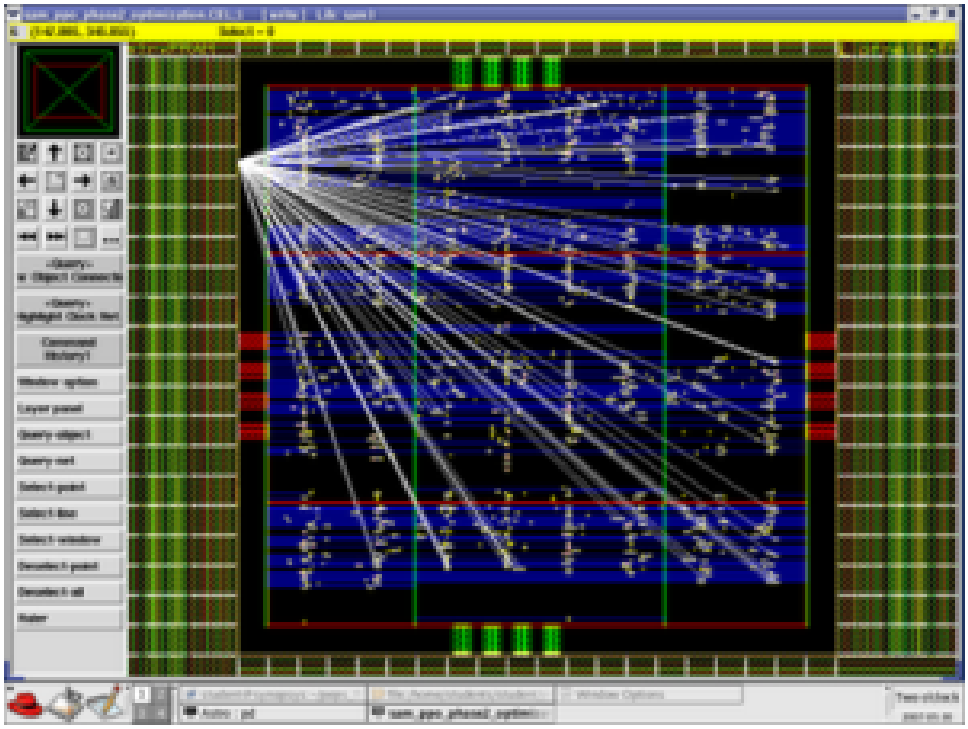


Figure 2.9: Ideal clock before CTS

delay. Clock is not propagated before CTS as shown in the picture. After CTS hold slack should improve. Clock tree begins at .sdc defined clock source and ends at stop pins of flop. There are two types of stop pins known

as ignore pins and sync pins. Dont touch circuits and pins in front end (logic synthesis) are treated as ignore circuits or pins at back end (physical synthesis). Ignore pins are ignored for timing analysis. If clock is divided then separate skew analysis is necessary.

- Global skew achieves zero skew between two synchronous pins without considering logic relationship.
- Local skew achieves zero skew between two synchronous pins while considering logic relationship.
- If clock is skewed intentionally to improve setup slack then it is known as useful skew.

Rigidity is the term coined in Astro to indicate the relaxation of constraints. Higher the rigidity tighter is the constraints

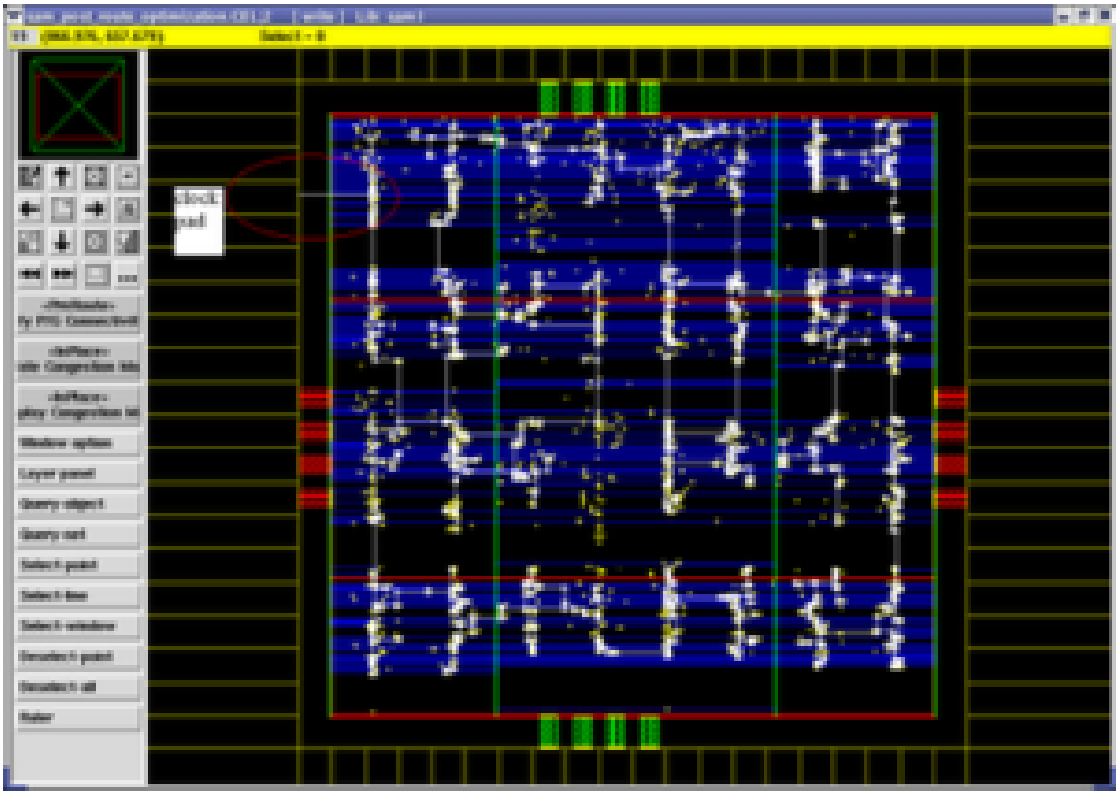



Figure 2.10: Clock After CTS

In clock tree optimization (CTO) clock can be shielded so that noise is not coupled to other signals. But shielding increases area by 12 to 15%. Since the

clock signal is global in nature the same metal layer used for power routing is used for clock also. CTO is achieved by buffer sizing, gate sizing, buffer relocation, level adjustment and HFN synthesis. We try to improve setup slack in pre-placement, in placement and post placement optimization before CTS stages while neglecting hold slack. In post placement optimization after CTS hold slack is improved. As a result of CTS lot of buffers are added. Generally for 100k gates around 650 buffers are added.

5. **Routing**

   There are two types of routing in the physical design process, global routing and detailed routing. Global routing allocates routing resources that are used for connections. Detailed routing assigns routes to specific metal layers and routing tracks within the global routing resources.

6. **Physical Verification**

   Physical verification checks the correctness of the generated layout design. This includes verifying that the layout

   - Complies with all technology requirements Design Rule Checking (DRC)
   - Is consistent with the original netlist Layout vs. Schematic (LVS)
   - Has no antenna effects Antenna Rule Checking
   - Complies with all electrical requirements Electrical Rule Checking (ERC).

# Chapter 3

# Architecture and Modifications

## 3.1 Overview

Toledo Architecture, based on TMS320c2x Architecture from Texas instruments, and provides 139 16-bit fixed point instructions.

Pipeline Operation, Three-level pipeline stages: Fetch, Decode and Execute, which allows three different instructions to be active during any given cycle.

External Memory and I/O Interface, Toledo supports a wide range of system interfacing requirements thus maximizing system throughput.
The system interface consists of:

- A 16-bit parallel data bus (D15D0).
- A 16-bit address bus (A15A0).
- External maskable Interrupts for External devices.
- External nonmaskable interrupt for Reset.
- Various system control signals.

Arithmetic Logic Unit, performs 2s-complement arithmetic s using 32-bit ALU and accumulator.
The ALU is a general purpose arithmetic unit that operates using 16-bit words taken from data RAM or derived from immediate instructions or using the 32-bit result of the multipliers product register.
In addition to the usual arithmetic instructions, the ALU can perform Boolean operations, providing the bit manipulation ability required of a high- speed controller.

Multiplier, The parallel multiplier performs a 16 16-bit 2s-complement multiplication with a 32-bit result in a single instruction cycle.
The multiplier consists of three elements:
T register, P register and multiplier array, The 16-bit T register temporarily stores the multiplicand; the P register stores the 32-bit product. The fast on-chip multiplier allows the device to perform efficiently the fundamental DSP operations such as convolution, correlation, and filtering. The scaling shifter has a 16-bit input connected to the data bus and a 32-bit output connected to the ALU. The scaling shifter produces a left shift of 0 to 16 bits on the input data, as programmed in the instruction. The LSBs of the output are filled with zeros, and the MSBs may be either filled with zeros or sign-extended, depending upon the state of the sign-extension mode bit of status register ST1. Additional shift capabilities enable the processor to perform numerical scaling, bit extraction, extended arithmetic, and overflow prevention [4].

## 3.2 Signals description

| Signal | Width | Direction | Description |
|---|---|---|---|
| VCC | 1 | I | 1.3-V supply pins |
| VSS | 1 | I | Ground pins |
| I_CLKIN_X2 | 1 | I | Input to internal oscillator from crystal or external clock |
| IO_D15-IO_D0 | 16 | I/O/Z | 16-bit data bus D15 (MSB) through D0 (LSB). Multiplexed between program, data, and I/O spaces. |
| O_A15-O_A0 | 16 | O/Z | 16-bit address bus A15 (MSB) through A0 (LSB) |
| O_DS_1 , O_PS_1 | 1 | O/Z | Program and data space select signals |
| O_RW | 1 | O/Z | Read/write signal |
| O_STRB_1 | 1 | O/Z | Strobe signal |
| I_RS_1 | 1 | I | Reset input |
| I_INT0_1 | 1 | I | External user interrupt inputs |
| O_IACK_1 | 1 | O | Interrupt acknowledge signal |

Figure 3.1: Signals description

### 3.2.1 HDL CPU signal description

```
‹module cpu_chip (O_A, IO_D, I_CLKIN_X2, I_INT0_1
I_RS_1, O_DS_1, O_PS_1, O_RW, O_STRB_1, O_IACK_1 );
‹output [15:0] O_A
‹inout [15:0] IO_D
‹input I_CLKIN_X2
‹input I_INT0_1
‹input I_RS_1
‹output O_DS_1
‹output O_PS_1
‹output O_RW
‹output O_STRB_1
‹output O_IACK_1
```

Figure 3.2: HDL CPU signal description

## 3.3 Registers Description

| Register Name | Symbol | Description |
|---|---|---|
| Auxiliary register pointer buffer | ARB | A 3-bit register used to buffer the ARP. Each time the ARP is loaded, the old value is written to the ARB, except during an LST (load status register) instruction. When the ARB is loaded with an LST1, the same value is also copied into ARP. |
| Auxiliary register pointer | ARP | A 3-bit register used to select one of five or eight auxiliary registers. |
| Data memory page pointer | DP | A 9-bit register pointing to the address of the current page. Data pages are 128 words each, resulting in 512 pages of addressable data memory space (some locations are reserved). |
| Interrupt flag register | IFR | A 6-bit flag register used to latch the active-low external user interrupts INT`(2–0), the internal interrupts XINT/RINT (serial port transmit/re-ceive), and TINT (timer) |

| | | |
|---|---|---|
| | | interrupts. The IFR is not accessible through software. |
| Interrupt mask register | IMR | A 6-bit memory-mapped register used to mask interrupts. |
| Instruction register | IR | A 16-bit register used to store the currently executing instruction. |
| Queue instruction register | QIR | A 16-bit register used to store prefetched instructions. |
| Product register | PR | A 32-bit product register used to hold the multiplier product. The PR can also be accessed as the most or least significant words by using the SPH/SPL (store P register high/low) instructions. |
| Period register for timer | PRD | A 16-bit memory-mapped register used to reload the timer. |
| Timer | TIM | A 16-bit memory-mapped timer (counter) for timing control. |
| Temporary register | TR | A 16-bit register that holds either an operand for the multiplier or a shift code for the scaling shifter. |
| Program Counter | PC | A 16-bit program counter used to address program memory. The PC always contains the address of the next instruction to be executed. The PC contents are updated following each instruction decode operation. |
| Prefetch counter | PFC | A 16-bit counter used to prefetch program instructions. The PFC con- tains the address of the instruction currently being prefetched. It is up- dated when a new prefetch is initiated. The PFC is also used to address program memory when using the block move (BLKP), multiply-accu- mulate (MAC/MACD), and table read/write (TBLR/TBLW) instructions and to address data memory when using the block move (BLKD) instruction. |
| Global memory allocation register | GREG | An 8-bit memory-mapped register for allocating the size of the global memory space. |
| Auxiliary registers | AR0-AR7 | A register file containing eight 16-bit auxiliary registers (AR0–AR7), used for addressing data memory, temporary storage, or integer arithmetic processing through the ARAU. |
| Status registers | ST0-ST1 | Two 16-bit status registers that contain status and control bits. A 16-bit register that holds either an operand for the multiplier or a shift code for the scaling shifter. |

## 3.4 Memory Addressing Modes

1. Direct addressing mode
2. Indirect addressing mode
3. Immediate addressing mode

### 3.4.1 Direct Addressing Mode

In the direct memory addressing mode, the instruction word contains the lower seven bits of the data memory address (dma).
This field is concatenated with the nine bits of the data memory page pointer (DP) register to form the full 16-bit data memory address. Thus, the DP register points to one of 512 possible 128-word data memory pages, and the 7-bit address in the instruction points to the specific location within that data memory page. The DP register is loaded through the LDP (load data memory page pointer), LDPK (load data memory page pointer immediate), or LST (load status register ST0) instructions.

### 3.4.2 Indirect Addressing Mode

The auxiliary registers (AR) provide flexible and powerful indirect addressing. Eight auxiliary registers (AR0AR7) are provided. To select a specific auxiliary register, the auxiliary register pointer (ARP) is loaded with a value from 0 through 7 designating AR0 through AR7.
The following symbols are used in indirect addressing, including bit-reversed (BR) addressing:

[*] Contents of AR(ARP) are used as the data memory address.

[*] Contents of AR(ARP) are used as the data memory address, then decremented after the access.

[*+] Contents of AR(ARP) are used as the data memory address, then incremented after the access.

[*0] Contents of AR(ARP) are used as the data memory address, and the contents of AR0 subtracted from it after the access.

[*0+] Contents of AR(ARP) are used as the data memory address, and the contents of AR0 added to it after the access.

[*BR0] Contents of AR(ARP) are used as the data memory address, and the contents of AR0 subtracted from it, with reverse carry (rc) propagation, after the access.

[*BR0+] Contents of AR(ARP) are used as the data memory address, and the contents of AR0 added to it, with reverse carry (rc) propagation, after the access.

Follows the indirect addressing arithmetic operations specified by the op-code:

| Bits 6 | 5 | 4 | Arithmetic Operation |
|---|---|---|---|
| 0 | 0 | 0 | No operation on AR(ARP) |
| 0 | 0 | 1 | AR(ARP) – 1 → AR(ARP) |
| 0 | 1 | 0 | AR(ARP) + 1 → AR(ARP) |
| 0 | 1 | 1 | Reserved |
| 1 | 0 | 0 | AR(ARP) – AR0 → AR(ARP) [reverse carry propagation] |
| 1 | 0 | 1 | AR(ARP) – AR0 → AR(ARP) |
| 1 | 1 | 0 | AR(ARP) + AR0 → AR(ARP) |
| 1 | 1 | 1 | AR(ARP) + AR0 → AR(ARP) [reverse carry propagation] |

Figure 3.3: Indirect Addressing Arithmetic Operations

The bit-3 in the instruction op-code specifies if the ARP will be modified after fetching the address and performing the specified operation on AR(ARP).
If it is one, the ARP will contain the value specified in bits 2-0 in the instruction.

### 3.4.3 Immediate Addressing Mode

In immediate addressing, the instruction word(s) contains the value of the immediate operand. The DSP has both single-word (8-bit and 13-bit constant) short immediate instructions and two-word (16-bit constant) long immediate instructions.
The immediate operand is contained within the instruction word itself in short immediate instructions. In long immediate instructions, the word following the instruction opcode is used as the immediate operand. The following short immediate instructions contain the immediate operand in the instruction word and execute within a single instruction cycle. The length of the constant operand is instruction-dependent.

## 3.5 Instruction Set Architecture

### 3.5.1 Instruction symbols

| Symbol | Meaning |
|---|---|
| **A** | Port address |
| **ACC** | Accumulator |
| **ARB** | Auxiliary register pointer buffer |
| **ARn** | Auxiliary register n (AR0, AR1 assembler symbols equal to 0 or 1) |
| **ARP** | Auxiliary register pointer |
| **B** | 4-bit field specifying a bit code |
| **BIO** | Branch control input |
| **C** | Carry bit |
| **CM** | 2-bit field specifying compare mode |
| **CNF** | On-chip RAM configuration control bit |
| **D** | Data memory address field |
| **DATn** | Label assigned to data memory location n |
| **Dma** | Data memory address |
| **DP** | Data page pointer |
| **FO** | Format status bit |
| **FSM** | Frame synchronization mode bit |
| **HM** | Hold mode bit |
| **INTM** | Interrupt mode flag bit |
| **K** | Immediate operand field |
| **M** | Addressing mode bit |
| **MCS** | Microcall stack |
| **Nnh** | nnh = hexadecimal number (others are decimal values) |
| **OV** | Overflow mode flag bit |
| **OVM** | Overflow mode bit |
| **P** | Product register |
| **PA** | Port address (PA0–PA15 assembler symbols equal to 0 through 15) |
| **PC** | Program counter |
| **PFC** | Prefetch counter |
| **PM** | 2-bit field specifying P register output shift code |
| **Pma** | Program memory address |
| **PRGn** | Label assigned to program memory location n |

Figure 3.4: Instruction Symbols

| Symbol | Meaning |
|---|---|
| R | 3-bit operand field specifying auxiliary register |
| RPTC | Repeat counter |
| S | 4-bit left-shift code |
| STn | Status register n (ST0 or ST1) |
| SXM | Sign-extension mode bit |
| T | Temporary register |
| TC | Test control bit |
| TOS | Top of stack |
| TXM | Transmit mode bit |
| X | 3-bit accumulator left-shift field |
| XF | XF pin status bit |

Figure 3.5: Instruction Symbols (Continued)

### 3.5.2 Instruction set summary

Figure 3.6 shows the instruction set summary for the Andalus processor. Included in the instruction set are four special groups of instructions to improve overall processor throughput and ease of use.

- Extended-precision arithmetic (ADDC, SUBB, MPYU, BC, BNC, SC, and RC)
- Adaptive filtering (MPYA, MPYS, and ZALR)
- Control and I/O (RHM, SHM, RTC, STC, RFSM, and SFSM)
- Accumulator and register (SPH, SPL, ADDK, SUBK, ADRK, SBRK, ROL, and ROR).

| ACCUMULATOR MEMORY REFERENCE INSTRUCTIONS | | | |
|---|---|---|---|
| Mnemonic and Description | | Words | 16-Bit Opcode MSB LSB |
| ABS | Absolute value of accumulator | 1 | 1100 1110 0001 1011 |
| ADD | Add to accumulator with shift | 1 | 0000 SSSS MDDD DDDD |
| ADDC | Add to accumulator with carry | 1 | 0100 0011 MDDD DDDD |
| ADDH | Add to high accumulator | 1 | 0100 1000 MDDD DDDD |
| ADDK | Add to accumulator short immediate | 1 | 1100 1100 KKKK KKKK |
| ADDS | Add to low accumulator with sign-extension suppressed | 1 | 0100 1001 MDDD DDDD |
| ADDT | Add to accumulator with shift specified by T register | 1 | 0100 1010 MDDD DDDD |
| ADLK | Add to accumulator long immediate with shift | 2 | 1101 SSSS 0000 0010 |
| AND | AND with accumulator | 1 | 0100 1110 MDDD DDDD |
| ANDK | AND immediate with accumulator with shift | 2 | 1101 SSSS 0000 0100 |
| CMPL | Complement accumulator | 1 | 1100 1110 0010 0111 |
| LAC | Load accumulator with shift | 1 | 0010 SSSS MDDD DDDD |
| LACK | Load accumulator short immediate | 1 | 1100 1010 KKKK KKKK |
| LACT | Load accumulator with shift specified by T register | 1 | 0100 0010 MDDD DDDD |
| LALK | Load accumulator long immediate with shift | 2 | 1101 SSSS 0000 0001 |
| NEG | Negate accumulator | 1 | 1100 1110 0010 0011 |
| NORM | Normalize contents of accumulator | 1 | 1100 1110 1010 0010 |
| OR | OR with accumulator | 1 | 0100 1101 MDDD DDDD |
| ORK | OR immediate with accumulator with shift | 2 | 1101 SSSS 0000 0101 |
| ROL | Rotate accumulator left | 1 | 1100 1110 0011 0100 |
| ROR | Rotate accumulator right | 1 | 1100 1110 0011 0101 |
| SAC | Store high accumulator with shift | 1 | 0110 1XXX MDDD DDDD |
| SACL | Store low accumulator with shift | 1 | 0110 0XXX MDDD DDDD |
| SBLK | Subtract from accumulator long immediate with shift | 2 | 1101 SSSS 0000 0011 |
| SFL | Shift accumulator left | 1 | 1100 1110 0001 1000 |
| SFR | Shift accumulator right | 1 | 1100 1110 0001 1001 |
| SUB | Subtract from accumulator with shift | 1 | 0001 SSSS MDDD DDDD |
| SUBB | Subtract from accumulator with borrow | 1 | 0100 1111 MDDD DDDD |
| SUBC | Conditional subtract | 1 | 0100 0111 MDDD DDDD |
| SUBH | Subtract from high accumulator | 1 | 0100 0100 MDDD DDDD |
| SUBK | Subtract from accumulator short immediate | 1 | 1100 1101 KKKK KKKK |
| SUBS | Subtract from low accumulator with sign extension suppressed | 1 | 0100 0101 MDDD DDDD |
| SUBT | Subtract from accumulator with shift specified by T register | 1 | 0100 0110 MDDD DDDD |
| XOR | Exclusive-OR with accumulator | 1 | 0100 1100 MDDD DDDD |
| XORK | Exclusive-OR immediate with accumulator with shift | 2 | 1101 SSSS 0000 0110 |
| ZAC | Zero accumulator | 1 | 1100 1010 0000 0000 |
| ZALH | Zero low accumulator and load high accumulator | 1 | 0100 0000 MDDD DDDD |
| ZALR | Zero low accumulator and load high accumulator with rounding | 1 | 0111 1011 MDDD DDDD |
| ZALS | Zero accumulator and load low accumulator with sign extension suppressed | 1 | 0100 0001 MDDD DDDD |

Figure 3.6: Instruction Set summary

| AUXILIARY REGISTERS AND DATA PAGE POINTER INSTRUCTIONS | | | |
|---|---|---|---|
| Mnemonic | Description | Words | 16-Bit Opcode MSB LSB |
| ADRK | Add to auxiliary register short immediate | 1 | 0111 1110 KKKK KKKK |
| CMPR | Compare auxiliary register with auxiliary register AR0 | 1 | 1100 1110 0101 00KK |
| LAR | Load auxiliary register | 1 | 0011 0RRR MDDD DDDD |
| LARK | Load auxiliary register short immediate | 1 | 1100 0RRR KKKK KKKK |
| LARP | Load auxiliary register pointer | 1 | 0101 0101 1000 1RRR |
| LDP | Load data memory page pointer | 1 | 0101 0010 MDDD DDDD |
| LDPK | Load data memory page pointer immediate | 1 | 1100 100K KKKK KKKK |
| LRLK | Load auxiliary register long immediate | 2 | 1101 0RRR 0000 0000 |
| MAR | Modify auxiliary register | 1 | 0101 0101 MDDD DDDD |
| SAR | Store auxiliary register | 1 | 0111 0RRR MDDD DDDD |
| SBRK | Subtract from auxiliary register short immediate | 1 | 0111 1111 KKKK KKKK |

| T REGISTER, P REGISTER, AND MULTIPLY INSTRUCTIONS | | | |
|---|---|---|---|
| Mnemonic | Description | Words | 16-Bit Opcode MSB LSB |
| APAC | Add P register to accumulator | 1 | 1100 1110 0001 0101 |
| LPH | Load high P register | 1 | 0101 0011 MDDD DDDD |
| LT | Load T register | 1 | 0011 1100 MDDD DDDD |
| LTA | Load T register and accumulate previous product | 1 | 0011 1101 MDDD DDDD |
| LTD | Load T register, accumulate previous product and move data | 1 | 0011 1111 MDDD DDDD |
| LTP | Load T register and store P register in accumulator | 1 | 0011 1110 MDDD DDDD |
| LTS | Load T register and subtract previous product | 1 | 0101 1011 MDDD DDDD |
| MAC | Multiply and accumulate | 2 | 0101 1101 MDDD DDDD |
| MACD | Multiply and accumulate with data move | 2 | 0101 1100 MDDD DDDD |
| MPY | Multiply (with T register, store product in P register) | 1 | 0011 1000 MDDD DDDD |
| MPYA | Multiply and accumulate previous product | 1 | 0011 1010 MDDD DDDD |
| MPYK | Multiply immediate | 1 | 101K KKKK KKKK KKKK |
| MPYS | Multiply and subtract previous product | 1 | 0011 1011 MDDD DDDD |
| MPYU | Multiply unsigned | 1 | 1100 1111 MDDD DDDD |
| PAC | Load accumulator with P register | 1 | 1100 1110 0001 0100 |
| SPAC | Subtract P register from accumulator | 1 | 1100 1110 0001 0110 |
| SPH | Store high P register | 1 | 0111 1101 MDDD DDDD |
| SPL | Store low P register | 1 | 0111 1100 MDDD DDDD |
| SPM | Set P register output shift mode | 1 | 1100 1110 0000 10KK |
| SQRA | Square and accumulate | 1 | 0011 1001 MDDD DDDD |
| SQRS | Square and subtract previous product | 1 | 0101 1010 MDDD DDDD |

Figure 3.7: Instruction Set summary(Continued)

| BRANCH/CALL INSTRUCTIONS | | | |
|---|---|---|---|
| Mnemonic and Description | | Words | 16-Bit Opcode MSB LSB |
| I/O AND DATA MEMORY OPERATIONS | | | |
| B | Branch unconditionally | 2 | 1111 1111 1DDD DDDD |
| BACC | Branch to address specified by accumulator | 1 | 1100 1110 0010 0101 |
| BANZ | Branch on auxiliary register not zero | 2 | 1111 1011 1DDD DDDD |
| BBNZ | Branch if TC bit ≠ 0 | 2 | 1111 1001 1DDD DDDD |
| BBZ | Branch if TC bit = 0 | 2 | 1111 1000 1DDD DDDD |
| BC | Branch on carry | 2 | 0101 1110 1DDD DDDD |
| BGEZ | Branch if accumulator ≠ 0 | 2 | 1111 0100 1DDD DDDD |
| BGZ | Branch if accumulator > 0 | 2 | 1111 0001 1DDD DDDD |
| BIOZ | Branch on I/O status = 0 | 2 | 1111 1010 1DDD DDDD |
| BLEZ | Branch if accumulator ≤ 0 | 2 | 1111 0010 1DDD DDDD |
| BLZ | Branch if accumulator < 0 | 2 | 1111 0011 1DDD DDDD |
| BNC | Branch on no carry | 2 | 0101 1111 1DDD DDDD |
| BNV | Branch if no overflow | 2 | 1111 0111 1DDD DDDD |
| BNZ | Branch if accumulator ≠ 0 | 2 | 1111 0101 1DDD DDDD |
| BV | Branch on overflow | 2 | 1111 0000 1DDD DDDD |
| BZ | Branch if accumulator = 0 | 2 | 1111 0110 1DDD DDDD |
| CALA | Call subroutine indirect | 1 | 1100 1110 0010 0100 |
| CALL | Call subroutine | 2 | 1111 1110 1DDD DDDD |
| RET | Return from subroutine | 1 | 1100 1110 0010 0110 |
| TRAP | Software interrupt | 1 | 1100 1110 0001 1110 |
| I/O AND DATA MEMORY OPERATIONS | | | |
| Mnemonic and Description | | Words | 16-Bit Opcode MSB LSB |
| BLKD | Block move from data memory to data memory | 2 | 1111 1101 MDDD DDDD |
| BLKP | Block move from program memory to data memory | 2 | 1111 1100 MDDD DDDD |
| DMOV | Data move in data memory | 1 | 0101 0110 MDDD DDDD |
| FORT | Format serial port registers | 1 | 1100 1110 0000 111K |
| IN | Input data from port | 1 | 1000 AAAA MDDD DDDD |
| OUT | Output data to port | 1 | 1110 AAAA MDDD DDDD |
| RFSM | Reset serial port frame synchronization mode | 1 | 1100 1110 0011 0110 |
| RTXM | Reset serial port transmit mode | 1 | 1100 1110 0010 0000 |
| RXF | Reset external flag | 1 | 1100 1110 0000 1100 |
| SFSM | Set serial port frame synchronization mode | 1 | 1100 1110 0011 0111 |
| STXM | Set serial port transmit mode | 1 | 1100 1110 0010 0001 |
| SXF | Set external flag | 1 | 1100 1110 0000 1101 |
| TBLR | Table read | 1 | 0100 1000 MDDD DDDD |
| TBLW | Table write | 1 | 0101 1001 MDDD DDDD |

Figure 3.8: Instruction Set summary(Continued)

| CONTROL INSTRUCTIONS | | | |
|---|---|---|---|
| Mnemonic and Description | | Words | 16-Bit Opcode MSB LSB |
| BIT | Test bit | 1 | 1001 BBBB MDDD DDDD |
| BITT | Test bit specified by T register | 1 | 0101 0111 MDDD DDDD |
| DINT | Disable interrupt | 1 | 1100 1110 0000 0001 |
| EINT | Enable interrupt | 1 | 1100 1110 0000 0000 |
| IDLE | Idle until interrupt | 1 | 1100 1110 0001 1111 |
| LST | Load status register ST0 | 1 | 0101 0000 MDDD DDDD |
| LST1 | Load status register ST1 | 1 | 0101 0001 MDDD DDDD |
| NOP | No operation | 1 | 0101 0101 0000 0000 |
| POP | Pop top of stack to low accumulator | 1 | 1100 1110 0001 1101 |
| POPD | Pop top of stack to data memory | 1 | 0111 1010 MDDD DDDD |
| PSHD | Push data memory value onto stack | 1 | 0101 0100 MDDD DDDD |
| PUSH | Push low accumulator onto stack | 1 | 1100 1110 0001 1100 |
| RC | Reset carry bit | 1 | 1100 1110 0011 0000 |
| RHM | Reset hold mode | 1 | 1100 1110 0011 1000 |
| ROVM | Reset overflow mode | 1 | 1100 1110 0000 0010 |
| RPT | Repeat instruction as specified by data memory value | 1 | 0100 1011 MDDD DDDD |
| RPTK | Repeat instruction as specified by immediate value | 1 | 1100 1011 KKKK KKKK |
| RSXM | Reset sign-extension mode | 1 | 1100 1110 0000 0110 |
| RTC | Reset test/control flag | 1 | 1100 1110 0011 0010 |
| SC | Set carry bit | 1 | 1100 1110 0011 0001 |
| SHM | Set hold mode | 1 | 1100 1110 0011 1001 |
| SOVM | Set overflow mode | 1 | 1100 1110 0000 0011 |
| SST | Store status register ST0 | 1 | 0111 1000 MDDD DDDD |
| SST1 | Store status register ST1 | 1 | 0111 1001 MDDD DDDD |
| SSXM | Set sign-extension mode | 1 | 1100 1110 0000 0111 |
| STC | Set test/control flag | 1 | 1100 1110 0011 0011 |

Figure 3.9: Instruction Set summary(Continued)

## 3.6 Toledo Block Diagram

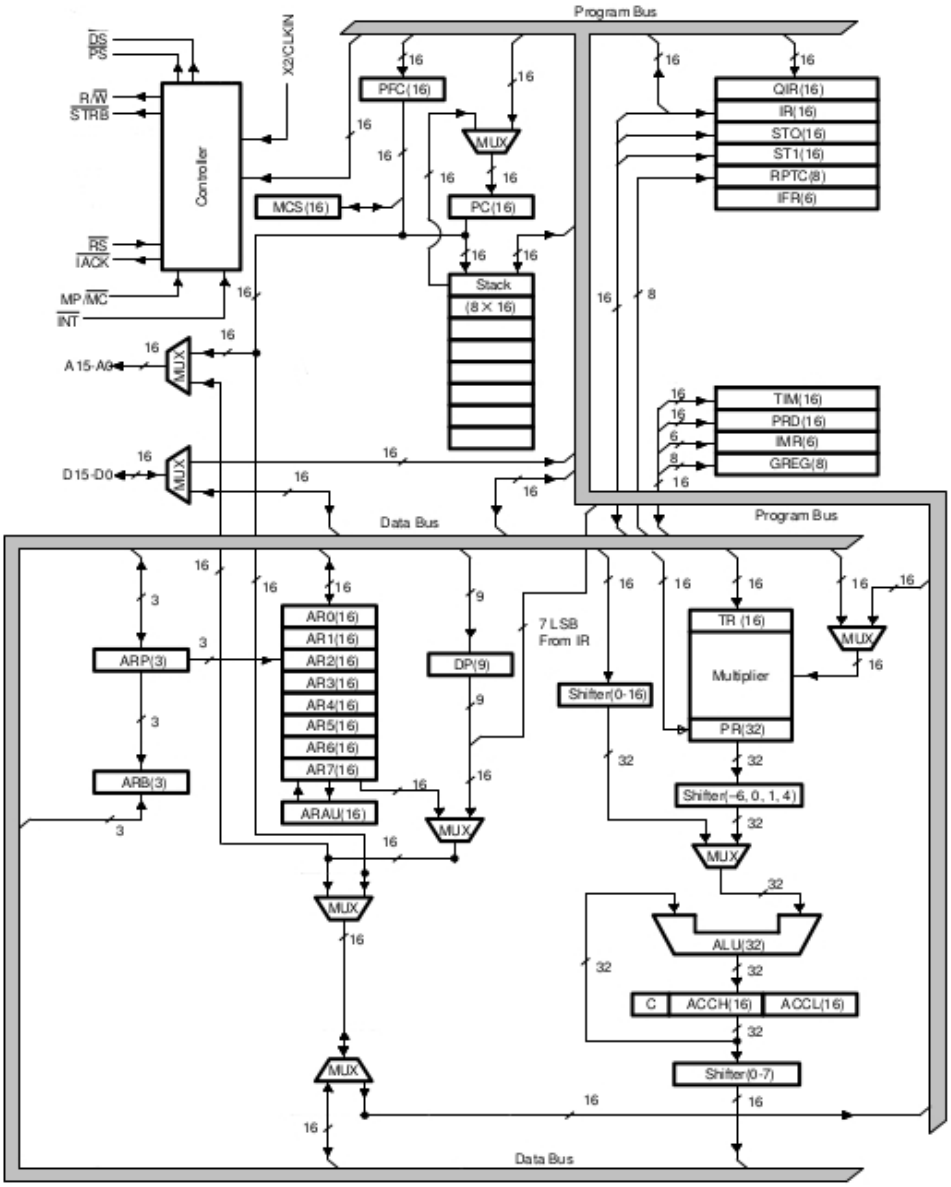



Figure 3.10: Toledo Architecture Block diagram

## 3.7 Control Unit

### 3.7.1 Block Diagram

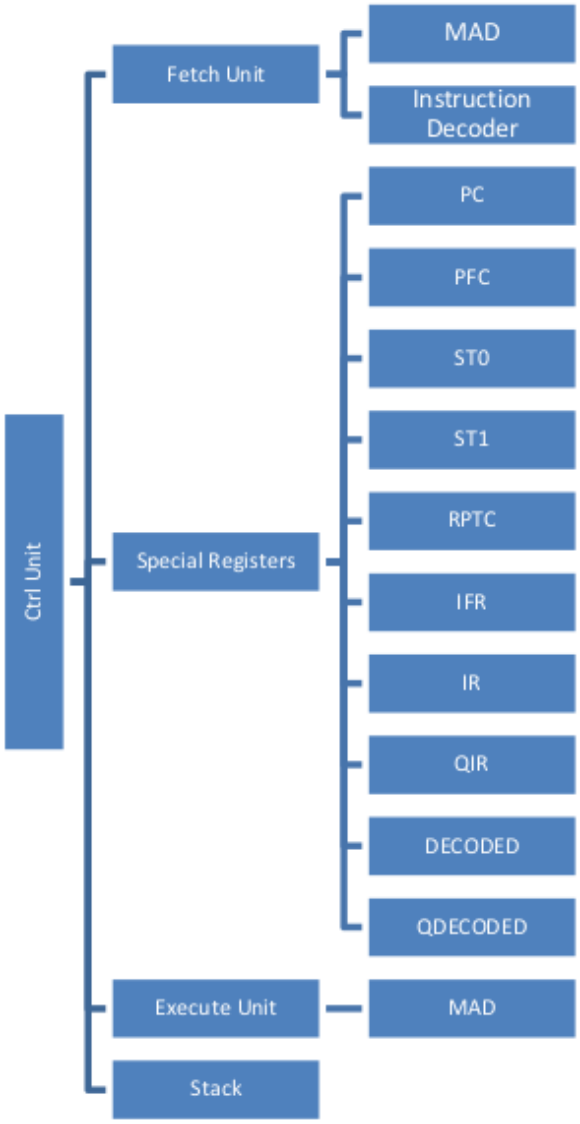



Figure 3.11: Control unit block diagram

### 3.7.2 Block Description

The control unit is the DSP controller, it manages the pipeline and generates the required control signals each clock cycle to manage the DSP operation[5].

### 3.7.3 Internal Blocks Description

1. Stack: It is a 8x16-bit stack that holds the value of the PC in case of CALL or INERRUPT

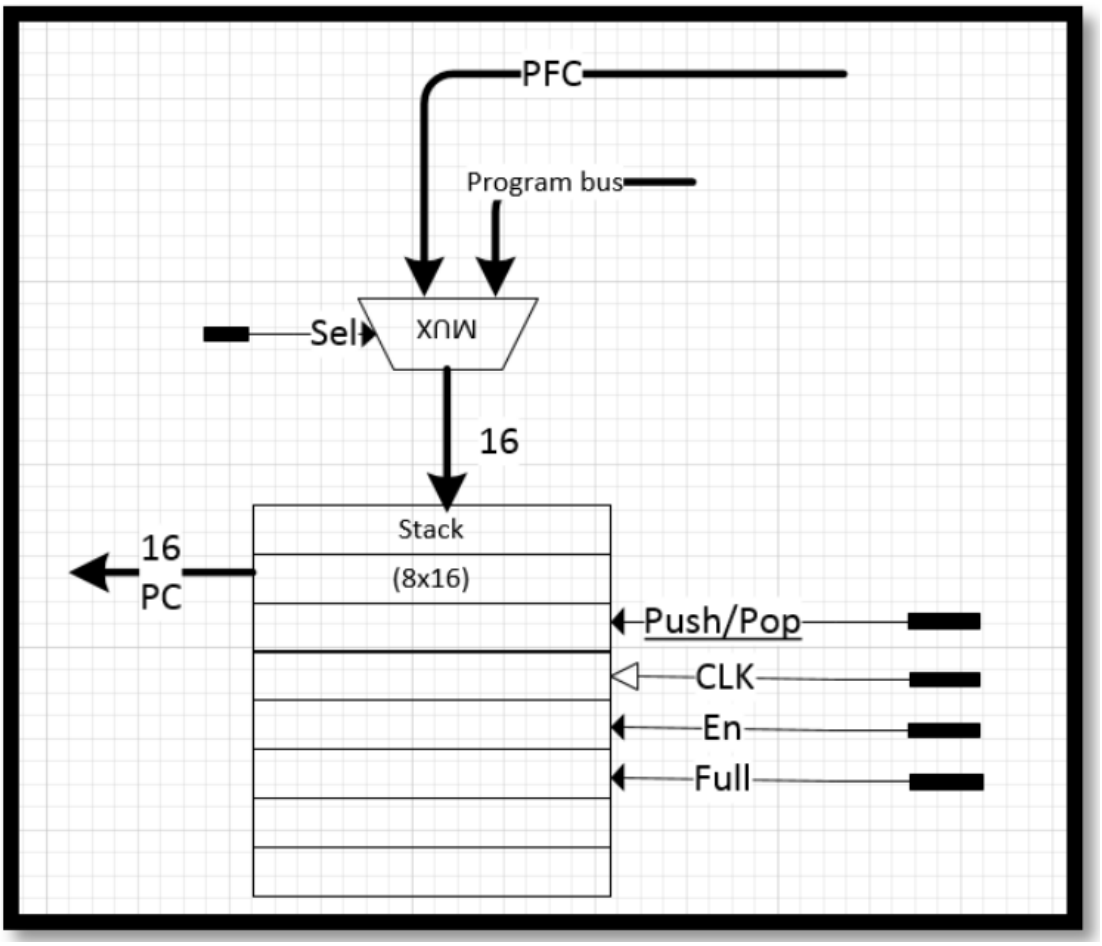


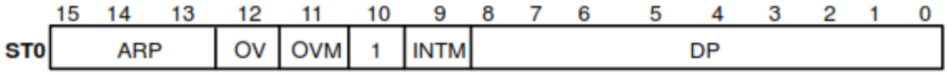

Figure 3.12: Stack block diagram

2. Special Registers: The user has no direct access to those registers, but there are some instructions that affects them directly or indirectly:

   (a) PFC Holds the address of the instruction being fetched

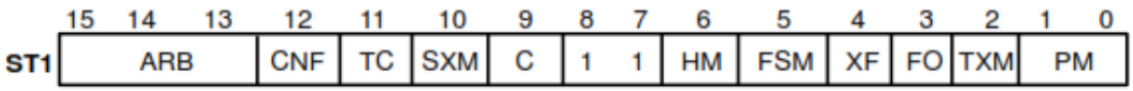
   (b) PC Holds the address of the next instruction to be fetched

   (c) ST0 Status register




| | |
|---|---|
| ARP | Auxiliary register pointer. This three-bit field selects the AR to be used in indirect addressing. When ARP is loaded, the old ARP value is copied to the ARB register. ARP may be modified by memory reference instructions when using indirect addressing, and by the LARP, MAR, and LST instructions. ARP Is also loaded with the same value as ARB when an LST1 instruction is executed. |
| OV | Overflow flag bit. As a latched overflow signal, OV is set to 1 when overflow occurs in the ALU. Once an overflow occurs, the OV remains set until a reset, BV, BNV, or LST instruction clears the OV. |
| OVM | Overflow mode bit. When set to 0, overflowed results overflow normally in the accumulator. When set to 1, the accumulator is set to either its most positive or its most negative value upon encountering an overflow. The SOVM and ROVM instructions set and reset this bit, respectively. LST may also be used to modify the OVM. |
| INTM | Interrupt mode bit. When set to 0, all unmasked interrupts are enabled. When set to 1, all maskable interrupts are disabled. INTM is set and reset by the DINT and EINT instructions. RS and IACK also set INTM. INTM has no effect on the unmaskable RS interrupt. Note that INTM is unaffected by the LST instruction. |
| DP | Data memory page pointer. The 9-bit DP register is concatenated with the 7 LSBs of an instruction word to form a direct memory address of 16 bits. DP may be modified by the LST, LDP, and LDPK instructions. |

(d) ST1 Status register



| | |
|---|---|
| ARB | Auxiliary register pointer buffer. Whenever the ARP is loaded, the old ARP value is copied to the ARB except during an LST instruction. When the ARB is loaded via an LST1 instruction, the same value is also copied to the ARP. |
| CNF | On-chip ram configuration control bit. If set to 0, block B0 is configured as data memory; otherwise, block B0 is configured as program memory. The CNF may be modified by the CNFD, CNFP, and LST1 instructions. RS resets the CNF to 0. |
| TC | Test/control flag bit. The TC bit is affected by the BIT, BITT, CMPR, LST1, and NORM instructions. The TC bit is set to a 1 if a bit tested by BIT or BITT is a 1, if a compare condition tested by CMPR exists between AR0 and another AR pointed to by ARP, or if the exclusive-OR function of the two MSBs of the accumulator is true when tested by a NORM instruction. Two branch instructions, BBZ and BBNZ, provide branching on the status of the TC. |
| SXM | Sign-extension mode bit. SXM = 1 produces sign extension on data as it is passed into the accumulator through the scaling shifter. SXM = 0 suppresses sign extension. SXM does not affect the definition of certain instructions; for example, the ADDS instruction suppresses sign extension regardless of SXM. This bit is set and reset by the SSXM and RSXM instructions, and may also be loaded by LST1. SXM is set to 1 by RS. |

| | |
|---|---|
| C | Carry bit. This bit is set to 1 if the result of an addition generates a carry, or reset to 0 if the result of a subtraction generates a borrow. Otherwise, it is reset after an addition or set after a subtraction, except if the instruction is ADDH or SUBH. ADDH can only set and SUBH only reset the carry bit, but cannot affect it otherwise. These instructions will also affect this bit: SC, RC, LST1, shift, and rotate. Two branch instructions, BC and BNC, have been provided to branch on the status of C. C is set to 1 on a reset. |
| HM | Hold mode bit. When HM = 1, the processor halts internal execution when acknowledging an active HOLD. When HM = 0, the processor may continue execution out of internal program memory but puts its external interface in a high-impedance state. This bit is set to 1 by a reset. |
| FSM | Frame synchronization mode bit. This bit indicates whether the serial port operates with or without frame sync pulses. When FSM = 1, the serial port operation is initiated following a frame sync pulse on the FSX/FSR inputs. When FSM = 0, the FSX/FSR inputs are ignored and the serial port operates continuously with no frame sync pulses required. The bit is set to 1 by a reset. |
| XF | XF pin status bit. This status bit indicates the state of the XF pin, a general purpose output pin. XF is set and reset by the SXF and RXF instructions or may be loaded by LST1. XF is set to 1 by RS. |

| | |
|---|---|
| FO | Format bit. When set to 0, the serial port registers are configured as 16-bit registers. When set to 1, the port registers are configured to receive and transmit eight-bit bytes. FO may be modified by the FORT and LST1 instructions. FO is reset to 0. |
| TXM | Transmit mode bit. TXM = 1 configures the serial port's FSX pin to be an output. In this mode, a pulse is produced on FSX when DXR is loaded. Transmission then starts on the DX pin. |
| | TXM = 0 configures the FSX pin to be an input. TXM is set and reset by the STXM and RTXM instructions and may also be loaded by LST1. RS resets TXM to 0. |
| PM | Product shift mode. If these two bits are 00, the multiplier's 32-bit product is loaded into the ALU with no shift. If PM = 01, the PR output is left-shifted one place and loaded into the ALU, with the LSBs zero-filled. If PM = 10, the PR output is left-shifted by four bits and loaded into the ALU, with the LSBs zero filled. PM = 11 produces a right shift of six bits, sign-extended. Note that the PR contents remain unchanged. The shift takes place when transferring the contents of the PR to the ALU. PM is loaded by the SPM and LST1 instructions. The PM bits are cleared by RS. |

(e) RPTC Is an 8-bit register that holds the value used with the instruction repeat property.

(f) IFR Interrupt flag register

(g) IR Holds the instruction being executed

(h) QIR Holds the instruction fetched by fetch unit

(i) DECODED Holds the decoded value of the instruction being executed

(j) QDECODED Holds the decoded value of the instruction fetched with fetch unit

3. Fetch/Decode Unit The first stage of the pipeline. It fetches the instruction specified by the PFC to the QIR register. In the last cycle of the fetch operation, the instruction is decoded and the decode value is stored in the QDECODED register. If the PFC was pointing to an internal-block address (in ROM or in B0), the fetch and decode operations takes one cycle.

4. Execute Unit In each clock cycle, the execute unit generates the control signals for the current execution state for the instruction in the IR register.

### 3.7.4 Hazards control

1. Structural Hazards The shared resources between the Fetch and Execute units (the pipeline stages) are:

   (a) ROM

   (b) B0 (if it is configured as program memory)

   (c) External Address space

   (d) Program Bus

   (e) Address Bus

   The used solution is to make two utilization registers, one for the execute unit and another for the fetch unit. Each resource from the mentioned has a one bit status in each of the registers. Each of the two units has read access to the other unit status register and a write access to its status register. Before a unit uses one of the previous resources, it makes sure that its bit is not set in the other unit status register, then it sets it in its own status register to indicate allocation of the resource.

2. Data Hazards: The only problem happens in case of jump, in which the fetch unit fetches an instruction which will not be used. In this case, the QIR and QDECODED are flushed out and the execute unit waits till the fetch unit fetches the instruction at the jump destination.

### 3.7.5 Implementation Notes

The implementation of the execute unit is a hard wired control signal generator, to use less area and make the execution faster.

## 3.8 Instruction Decoder

### 3.8.1 Block Diagram

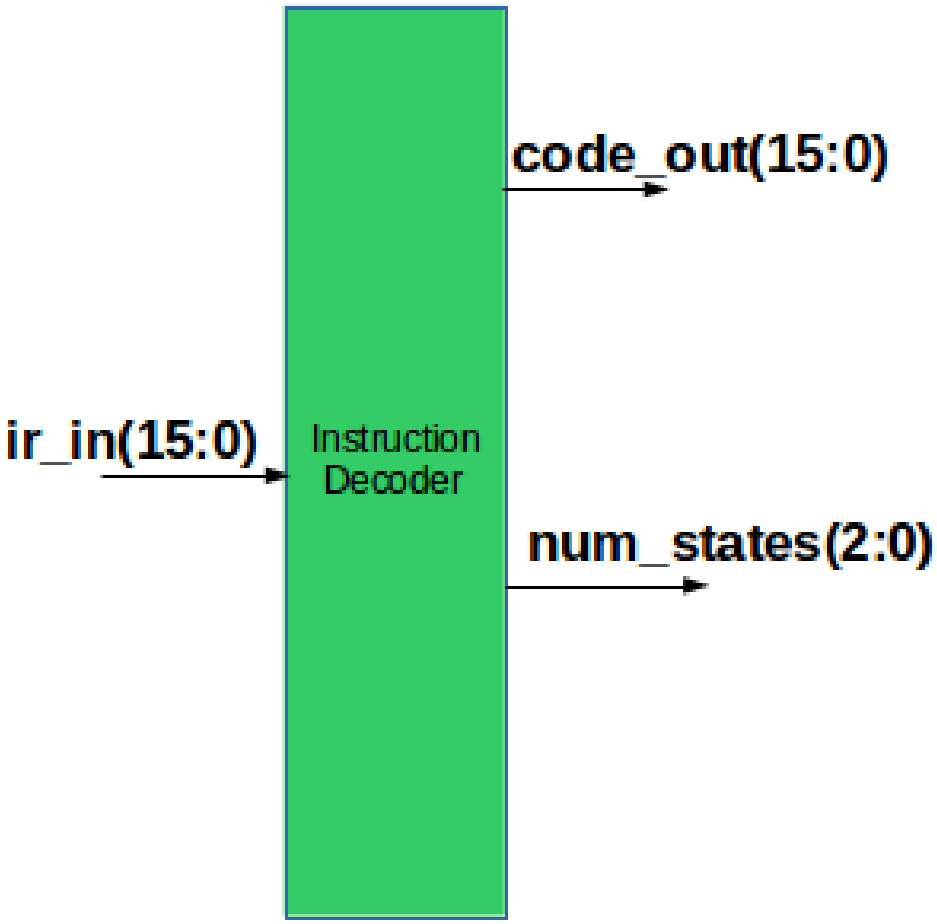



Figure 3.13: Instruction decoder block diagram

### 3.8.2 Block Description

A component that provides the execute unit with some execution information about the instruction to be executed:

1. Number of execution states

2. Long (2 word) or short (1 word) instruction
3. If indirect memory access exists
4. If direct memory access exists
5. The instruction number (8-bits code that specifies the instruction)

### 3.8.3 Ports Description

| Pin Name | Width | Direction | Description |
|---|---|---|---|
| ir_in | 16 | IN | The instruction to be decoded |
| code_out | 11 | OUT | - Bit 10 (MSB) => 1 \| long, 0 \|short<br>- Bit 9 => 1 \| Immediate/indirect exists<br>- Bit 8 => Indirect/DMA bit<br>- Bit 7-0 => Instruction code |
| num_states | 3 | OUT | The number of states (min 1) |

### 3.8.4 Implementation Notes

The component is implemented as nested comparators that checks the instruction bits.

## 3.9 Shifter

### 3.9.1 Block Diagram

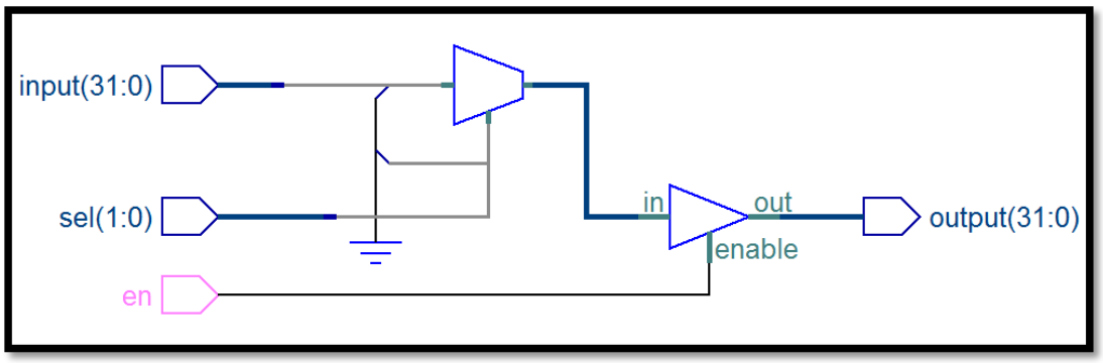



Figure 3.14: Shifter block diagram

### 3.9.2 Block Description

A shifter that shifts the output of the multiplier by the value selected by sel signal as follows:

| Sel value | Shift |
|---|---|
| 00 | No shift |
| 01 | Shift left by one |
| 10 | Shift left by four |
| 11 | Shift right by six |

### 3.9.3 Ports Description

| Pin Name | Width | Direction | Description |
|---|---|---|---|
| input | 32 | IN | The shifter input |
| sel | 2 | IN | Select the shifting value |
| en | 1 | IN | Enable shifter |
| output | 32 | OUT | Shifter output |

### 3.9.4 Implementation Notes

The component is implemented as multiplexers, where each output bit is connected to a 4x1 multiplexer.

## 3.10 Descaling Shifter

### 3.10.1 Block Description

A shifter that shifts left the output of the CALU by value from 0 to 7,selected by sel signal, then its output will be the low 16-bit or the high 16-bit of the shifted result using the low_ high input signal.

### 3.10.2 Ports Description

| Pin Name | Width | Direction | Description |
|---|---|---|---|
| input | 32 | IN | The shifter input |
| sel | 3 | IN | Select the shifting value |
| En | 1 | IN | Enable shifter |
| Low_high | 1 | IN | If 1, output the low 16-bits of shifted value, else output the high 16-bits |
| output | 16 | OUT | Shifter output |

### 3.10.3 Implementation Notes

The component is implemented as multiplexers, where each output bit is connected to a 7x1 multiplexer (a signal from each input bit in shifting range).

## 3.11 Scaling Shifter

### 3.11.1 Block Description

A shifter that shifts left the input of the CALU by value from 0 to 15,selected by sel signal.

# Ports Description

| Pin Name | Width | Direction | Description |
|---|---|---|---|
| input | 16 | IN | The shifter input |
| sel | 4 | IN | Select the shifting value |
| en | 1 | IN | Enable shifter |
| Sx_mode | 1 | IN | The sign extension mode (if 1, use the SXM, else extent with zeros) |
| sxm | 1 | IN | SXM bit from status register |
| output | 32 | OUT | Shifter output |

### 3.11.2 Implementation Notes

The component is implemented as multiplexers, where each output bit is connected to a 32x1 multiplexer (a signal from each input bit and its inverse).

## 3.12 Register File

### 3.12.1 Block Diagram

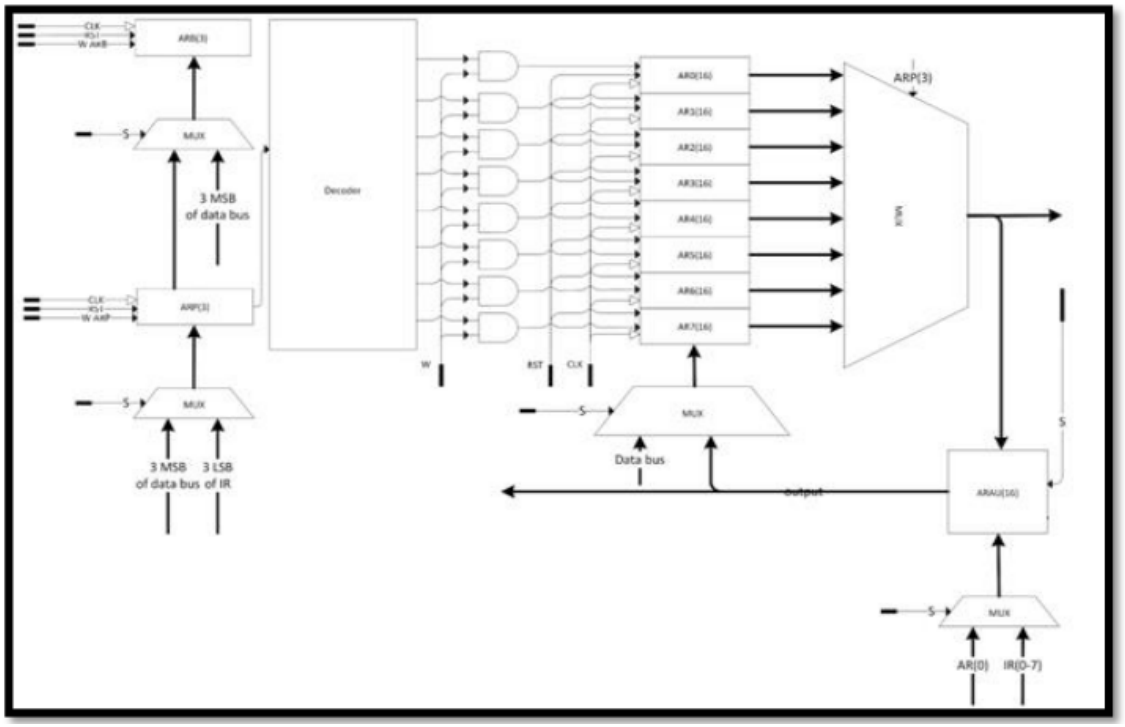



Figure 3.15: Register file block diagram

# Block Description

This unit is a 16-bit register file along with its own ARAU unit. It is mainly used for the different indirect addressing modes supported by the DSP.

### 3.12.2 Implementation Notes

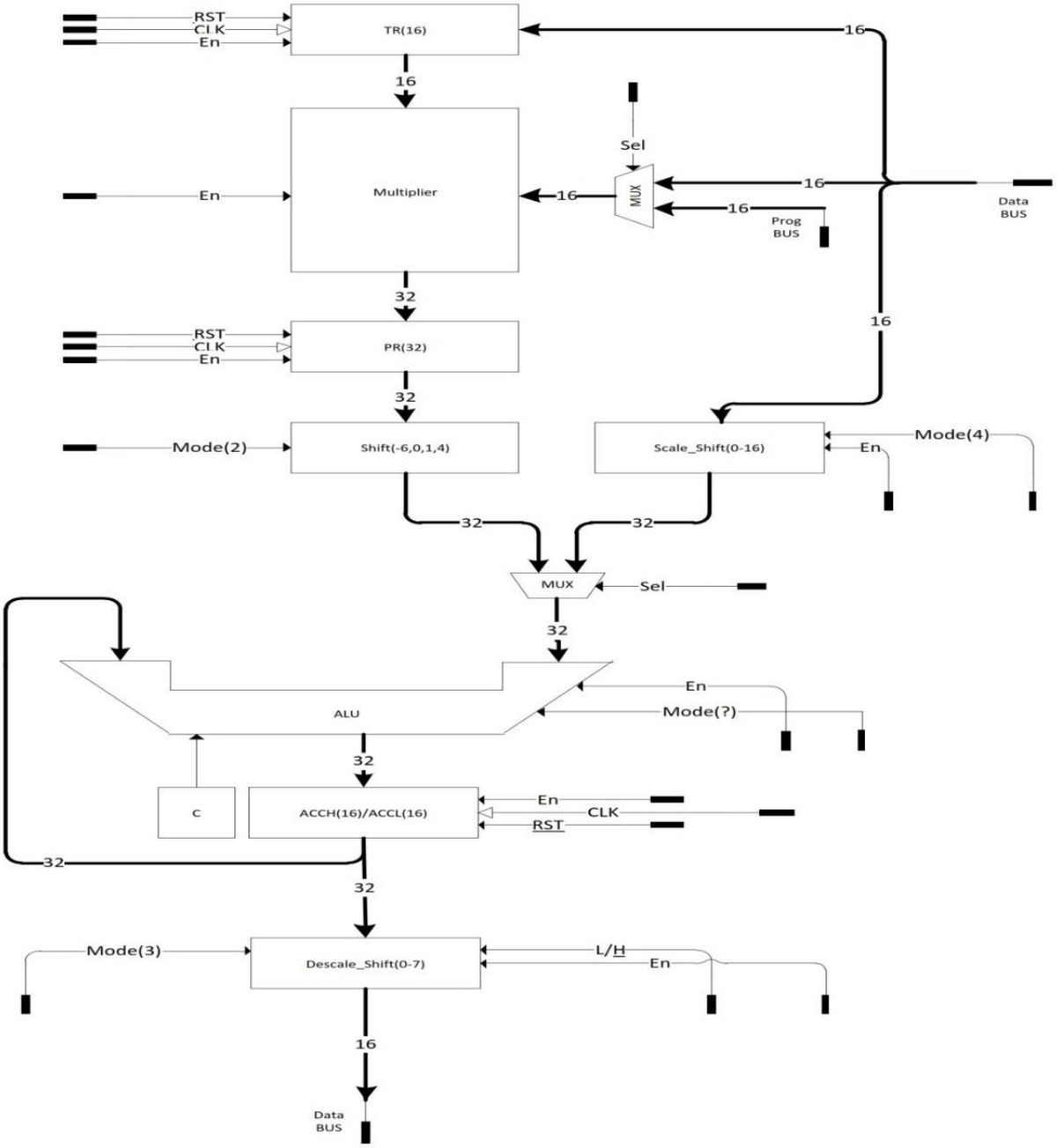
This component is divided into two main components, the register file and the ARAU. The implementation of the register file was straight forward; write to the register specified by the ARP when the write signal is enabled, the output of the register file is always the contents of that specified register. As for the implementation of the ARAU, two units of an n bit adder were used, one for the normal add/subtract operations and one for the reverse carry operations.

| Pin Name | Width | Direction | Description |
|---|---|---|---|
| clk | 1 | IN | DSP clock. |
| reset | 1 | IN | Active low asynchronous reset. |
| write_arp | 1 | IN | The write signal for the arp register. |
| write_arb | 1 | IN | The write signal for the arb register. |
| write_ar | 1 | IN | The write signal for the auxiliary register. |
| arb_select | 1 | IN | Selects 3MSB of data bus (1) /arp output (0) as an input to the arb register. |
| arp_select | 1 | IN | Select 3MSB of data bus (0) /3LSB of program bus (1) as an input to the arp register. |
| ar_select | 1 | IN | Select data bus (0) /arau output (1) as an input to the auxilary register file. |
| Ar_select_mode | 1 | IN | Specifies if the ARP register is used to specify the address of the operation register(0) or bits 10-8 in program bus(1) |
| arau_select | 1 | IN | Select ar0 (0)/ program bus (1) as the second operand for the arau. |
| arau_cm | 2 | IN | Compare mode<br>If CM = 00, test if AR(ARP) = AR0<br>If CM = 01, test if AR(ARP) < AR0<br>If CM = 10, test if AR(ARP) > AR0<br>If CM = 11, test if AR(ARP) != AR0 |
| arau_mode | 3 | IN | 000 add<br>001 subtract<br>010 increment<br>011 decrement<br>100 transfer<br>101 add with reverse carry<br>110 subtract with reverse carry<br>111 compare |
| data_bus | 16 | IN | The data bus. |
| program_bus | 16 | IN | The program bus. |
| arau_tc | 1 | OUT | The TC bit in the status register. |
| arp | 3 | OUT | The contents of the arp register. |
| arb | 3 | OUT | The contents of the arb register. |
| ar_of_arp | 16 | OUT | The contents of AR(ARP) |

| operation | Input0 | Intput1 | Carry |
|---|---|---|---|
| Add/Add with RC | A | B | 0 |
| Sub/Sub with RC | A | Not B | 1 |
| Increment | A | 0x0000 | 1 |
| Decrement | A | 0xffff | 0 |
| Transfer | A | 0x0000 | 0 |
| Compare | A | Not B | 1 |

# 3.13 CALU

## 3.13.1 Block diagram



### 3.13.2 Block Description

The central arithmetic logic unit (CALU) contains a 16-bit scaling shifter, a 16 x 16-bit parallel multiplier, a 32-bit arithmetic logic unit (ALU), a 32-bit accumulator (ACC), and additional shifters at the outputs of both the accumulator and the multiplier.

#### Multiplier

#### Overview

Multiplication is a very important operation for most DSP common algorithms, convolution and filtering are the most common ones. These operations requires big number of multiplications and additions. So, doing the multiplication operation efficiently is an important thing to be considered in DSP design.
The Multiplier utilizes a 16  16-bit hardware multiplier, which is capable of computing a signed or unsigned 32-bit product in a single machine cycle. All multiply instructions, except the MPYU (multiply unsigned) instruction, perform a signed multiply operation in the multiplier. That is, the two numbers being multiplied are treated as 2s complement numbers, and the result is a 32-bit 2s complement number. The following two registers are associated with the multiplier:

1. A 16-bit temporary register (TR) that holds one of the operands for the multiplier,

2. A 32-bit product register (PR) that holds the product.

The output of the product register can be left-shifted 1 or 4 bits. This is useful for implementing fractional arithmetic or justifying fractional products. The output of the PR can also be right-shifted 6 bits to enable the execution of up to 128 consecutive multiply/accumulates without the possibility of overflow.
An LT (load T register) instruction normally loads the TR to provide one operand (from the data bus), and the MPY (multiply) instruction provides the second operand (also from the data bus). A multiplication can also be performed with an immediate operand using the MPYK instruction. In either case, a product can be obtained every two cycles.
Two multiply/accumulate instructions (MAC and MACD) fully utilize the computational bandwidth of the multiplier, allowing both operands to be processed simultaneously. This provides for single-cycle multiply/accumulates when used with repeat (RPT/RPTK) instructions. The SQRA (square/add) and SQRS (square/subtract) instructions pass the same value to both inputs of the multiplier for squaring a data memory value.

The MPYU instruction performs an unsigned multiplication, which greatly facilitates extended-precision arithmetic operations. The unsigned contents of the T register are multiplied by the unsigned contents of the addressed data memory location, with the result placed in the P register. This allows operands of greater than 16 bits to be broken down into 16-bit words and processed separately to generate products of greater than 32 bits.
Four product shift modes (PM) are available at the PR output and are useful when performing multiply/accumulate operations and fractional arithmetic, or when justifying fractional products. The PM field of status register ST1 specifies the PM shift mode, as shown in Table below.

| PM | Result |
|---|---|
| 00 | No shift |
| 01 | Left shift 1bit |
| 10 | Left shift 4 bits |
| 11 | Right shift of 6 bits |

### Radix 4 Multiplier

The most trivial multiplier approach is the array multiplier. Generating a partial products to be summed to form the operation product. For each bit in the multiplicand, a partial product is generated, if the bit chosen is 0, the partial product is all zeros, and if that bit is 1, the partial product is the same as multiplier. For the 16 by 16 bits array multiplier, a 15 summation stages are needed, which is big number of logic gates and lots of power consumption.
A very famous approach to reduce number of summation stages needed is the radix 4 multiplier. Booth multiplier reduces the number of logic levels needed to eight levels for 16x16 operations. That is because we process 3 bits at the time instead of one.
Due to the need for a parallel multiplier, other multiplication options like array multiplication or radix-2 booth multiplication became not applicable because of the large area or the large power consumption. Therefore, radix-4 booth multiplier is the best choice because of its reduced logic levels, and better power consumption.
The multiplier consists of eight units of the booth encoder, each one of them is responsible for processing 3 bits from the input. The operation of each MBE (multiplication booth encoder) goes like the following:

1. Extend the sign bit one bit in case of a logic vector of odd size.
2. Add 0 to the LSB (least significant bit) of the multiplier.

3. According to the 3 bits that are send to the decoder, define the operation as one of the following operations mention in the table.

4. The multiplication by 2 or by -2 is done by shifting and the addition is done using a 16-bit adder.

5. Each MBE produce 2 bits of the final product, called the partial product.

6. Last MBE produces 18 bits not only 2 bits for the final product.

| B(i) | B(i-1) | B(i-2) | Operation |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | +A |
| 0 | 1 | 0 | +A |
| 0 | 1 | 1 | +2A |
| 1 | 0 | 0 | -2A |
| 1 | 0 | 1 | -A |
| 1 | 1 | 0 | -A |
| 1 | 1 | 1 | 0 |

### ALU and Accumulator

32-bit ALU and accumulator implement a wide range of arithmetic and logical operations, Once an operation is performed in the ALU, the result is transferred to the accumulator where additional operations such as shifting may occur. Data that is input to the ALU may be scaled by the scaling shifter.

### ALU

The ALU is a general-purpose arithmetic and logical unit that operates on 16-bit words taken from data bus or derived from immediate instructions. One input to the ALU is always provided from the accumulator, and the other input may be provided from the product register (PR) of the multiplier or the input scaling shifter that has fetched data from the RAM on the data bus. After the ALU has performed the arithmetic or logical operations, the result is stored in the accumulator.

### Accumulator

The 32-bit accumulator is split into two 16-bit segments for storage in data memory: ACCH (accumulator high) and ACCL (accumulator low). Shifters at the output of the accumulator provide a left-shift of 0 to 7 places. This shift is performed while the data is being transferred to the data bus for storage. The contents of the accumulator remain unchanged. When the ACCH data is shifted left, the LSBs are transferred from the ACCL, and the MSBs are lost. When ACCL is shifted left, the LSBs are zero-filled, and the MSBs are lost.

### ALU Operations op-codes

| NO. | Operation | Operation Description | Op-code |
|---|---|---|---|
| 1 | ADD | Add | 00001 |
| 2 | ADDC | Add with carry | 00010 |
| 3 | SUB | Subtract from accumulator | 00011 |
| 4 | SUBB | Subtract from accumulator with borrow | 00100 |
| 5 | AND | AND with accumulator | 00101 |
| 6 | OR | OR with accumulator | 00110 |
| 7 | CMPL | Complement accumulator | 00111 |
| 8 | XOR | Exclusive-OR with accumulator | 01000 |
| 9 | ROL | Rotate accumulator left | 01001 |
| 10 | ROR | Rotate accumulator right | 01010 |
| 11 | SFL | Shift accumulator left | 01011 |
| 12 | SFR | Shift accumulator right | 01100 |
| 13 | ABS | Absolute value of accumulator | 01110 |
| 14 | NEG | Negate accumulator Add with carry | 01111 |
| 15 | LAC | Load Accumulator | 10000 |
| 16 | NORM | Normalize contents of accumulator | 10001 |
| 17 | ZAC | Zero accumulator | 10010 |
| 18 | ZALH | Zero low accumulator and load high accumulator | 10011 |
| 19 | ZALR | Zero low accumulator and load high accumulator with rounding | 10100 |
| 20 | ZALS | Zero accumulator and load low accumulator with sign extension suppressed | 10101 |
| 21 | SUBC | Conditional subtract | 10110 |

## 3.14 Interrupts

The DSP has external maskable user interrupts available for external devices that interrupt the processor. Internal interrupts are generated by the timer (TINT), and by the software interrupt (TRAP) instruction. Interrupts are prioritized with reset (RS) having the highest priority and the serial port transmit interrupt (XINT) having the lowest priority.
When an interrupt occurs, it is stored in the 6-bit interrupt flag register (IFR). This register is set by the external user interrupts and the internal interrupts RINT, XINT, and TINT. Each interrupt is stored in the IFR until it is recognized, and then automatically cleared by the IACK (interrupt acknowledge) signal or the RS (reset) signal. The RS signal is not stored in the IFR. No instructions are provided for reading from or writing to the IFR. The DSP has a memory-mapped interrupt mask register (IMR) for masking external and internal interrupts. The layout of the register is shown in Figure 1. A 1 in bit positions 5 through 0 of the IMR enables the corresponding interrupt, provided that INTM = 0. The IMR is accessible with both read and write operations but cannot be read using BLKD. When the IMR is read, the unused bits (15 through 6) are read as 1s. The lower six bits are used to write to or read from the IMR. Note that RS is not included in the IMR, and therefore the IMR has no effect on reset.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Reserved | Reserved | Reserved | Reserved | Reserved | Reserved | Reserved | Reserved | Reserved | Reserved | XINT | RINT | TINT | INT2' | INT1' | INT0' |

The INTM (interrupt mode) bit, which is bit 9 of status register ST0, enables or disables all maskable interrupts. INTM = 0 enables all the unmasked interrupts, and INTM = 1 disables these interrupts. The INTM is set to 1 by the IACK (interrupt acknowledge) signal, the DINT instruction, or a reset. This bit is reset to 0 by the EINT instruction. Note that the INTM does not actually modify the IMR or IFR.

## 3.15 Architecture Changes

- Internal Memories TMS320C25 architecture employs four different internal memories. Three of which are RAMs called B0, B1 and B2, and the Fourth is a ROM. There are also 6 memory mapped registers.
  The ROM has a size of 4096 words, 16 bits wide, Accessible by a 12 bits

address. It is used only as a program memory.

B0 has a size of 256 words, 16 bits wide. It has the ability to be configured as data memory or program memory at run time. This gives user the ability to add new programs to the chip without actually using an external programmer. One can configure B0 as a data memory, receive new programs through the serial port for example, write it on B0 block, then convert it to a program memory and use the code.

B1 and B2 have a size of 256 and 32 respectively. They can only be used as data memories. The processor uses two separate memory spaces for program and data accessible by 16 bit address. Each space can be divided to 512 pages of 128 words each. The spaces are as shown in figure 3.16 when B0 is used as data memory. And in Figure 3.17 when used as a program memory.

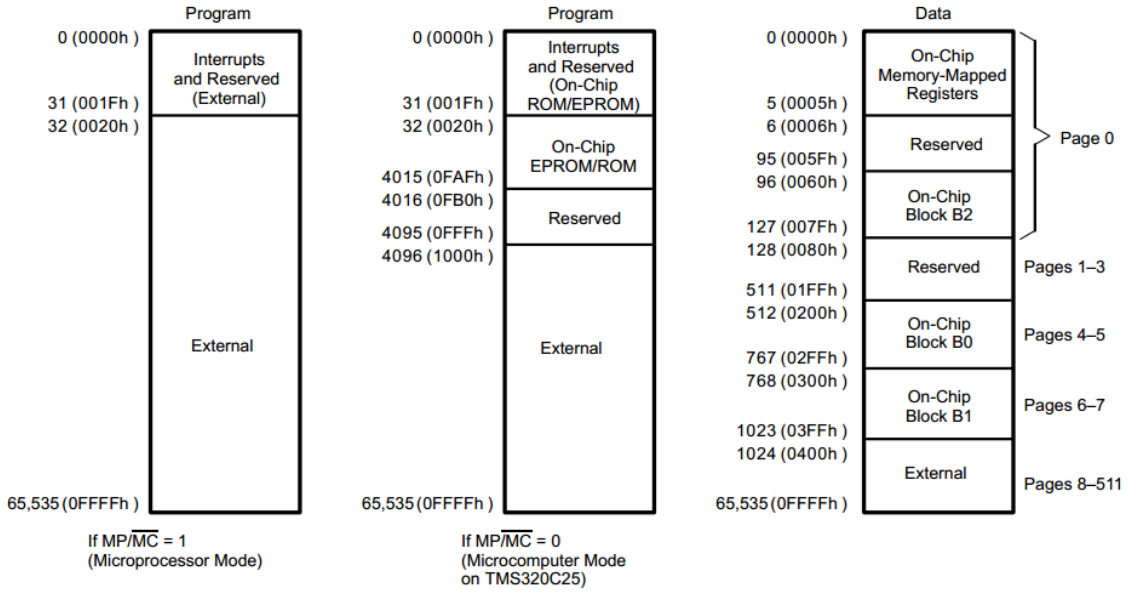



Figure 3.16: When B0 is used as data memory

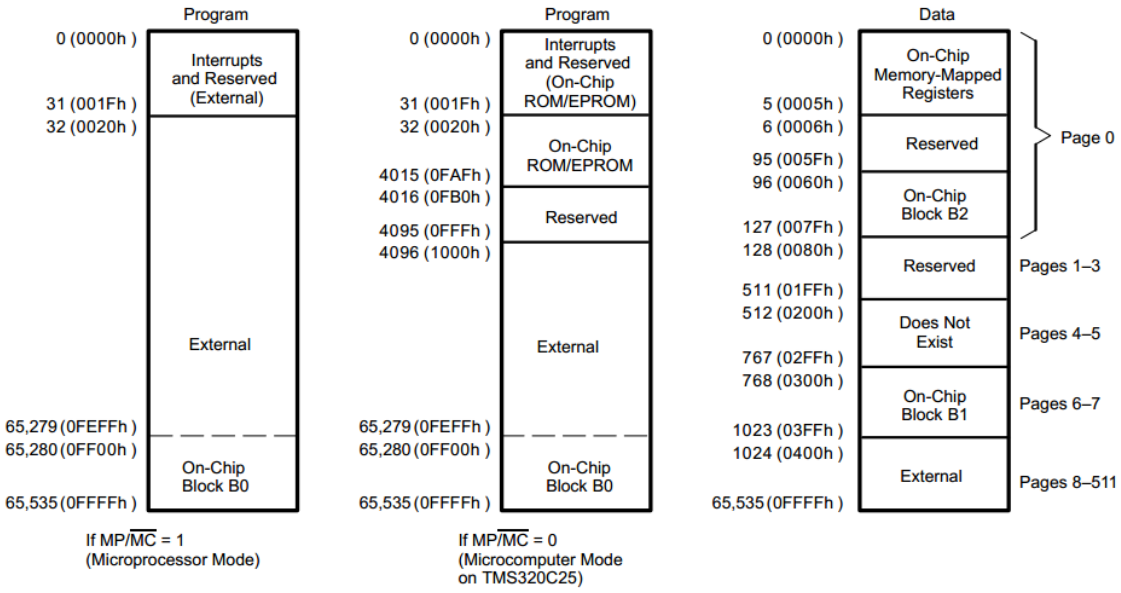



Figure 3.17: when B0 used as a program memory

- **External Asynchronous Memory**

  Because our library was lacking memory cells, and we did not have access to memory compilers, we were forced to remove all internal memories and suffice with just the external memory. For our architecture we chose to use an asynchronous memory of 8ns access delay. This is the best delay of an asynchronous memory we could find.

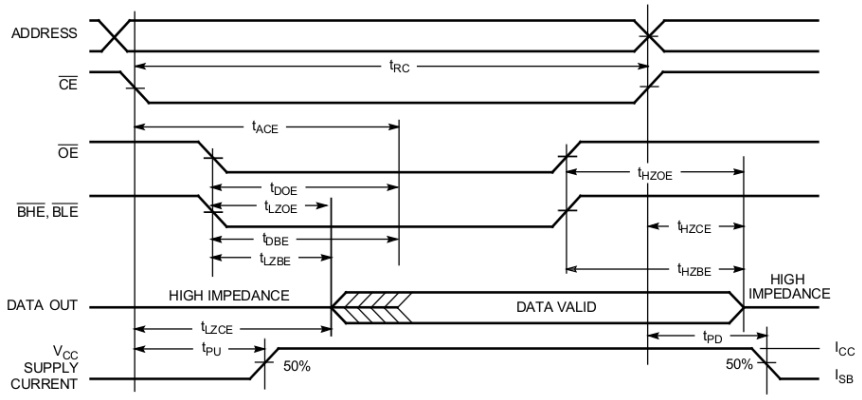



Figure 3.18: First Read Operation

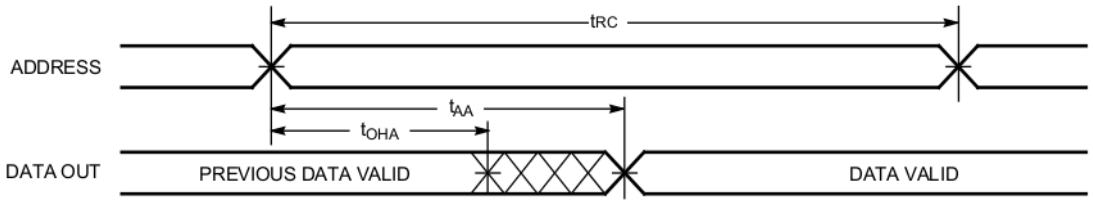



Figure 3.19: Reads in a row

Although it would limit our operating frequency, it was the best solution. A synchronous memory would require burst operation which would force us to further change the architecture.

Our chosen memory IC is 256K words, accessible by an 18 bits address, Second and third quarters are used as data and program memories, the first and last are unused. It has the timing diagrams shown in Fig 3.18 , 3.19. Since we don't use the internal memories, we have activated the reserved areas in the memory map. All the address spaces are now used and accessible from the external memory, except for the memory mapped registers, these are 6 registers tied to the operation of serial port, timer and other peripherals. They cannot be used from external memory. Our address space is that in Figure 3.20.

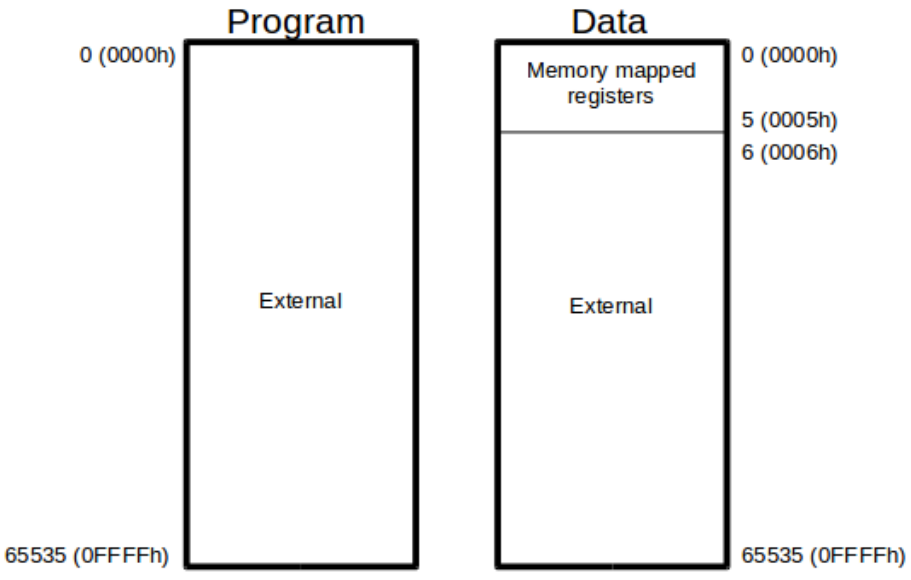



Figure 3.20: Address space

- Required Changes Since the core was only verified for internal memory usage, and the memory interface and control signals were not tested, we had to design those from the beginning.

  The processor has 4 memory interface signals and the address and data ports, we first had to change the memory address decoders inside the processors to map the addresses to the new memory map. And then setup the memory interface signals to work as required. After this was done, major changes in the instruction decoder and control logic were added to allow the memory access delays. Instructions that would previously execute and access internal memory in the same cycle would now take to cycles to do so with external memories. This essentially added a pipeline stage for memory accesses [6].

# Chapter 4

# Synthesis and layout

## 4.1 Synthesis

### 4.1.1 Synthesis Process

The synthesis process infers a possible gate-level realization of the input RTL description that meets user-defined constraints such as area, timings or power consumption.

The design constraints are defined outside the VHDL models by means of tool-specific commands. The targeted logic gates belong to a library that is provided by a foundry or an IP company as part of a so-called design kit.
Typical gate libraries include a few hundreds of combinational and sequential logic gates. Each logic function is implemented in several gates to accommodate several fan-out capabilities or drive strengths. The gate library is described in a tool-specific format that defines, for each gate, its function, its area, its timing and power characteristics and its environmental constraints.

The synthesis step generates several outputs: a gate-level VHDL netlist, a Verilog gatelevel netlist, and a SDF description. The first netlist is typically used for post-synthesis simulation, while the second netlist is better suited as input to the place&route step.

The SDF description includes delay information for simulation. Note that considered delays are at this step correct for the gates but only estimated for the interconnections.

For out project we used Synopsys Design Compiler for the synthesis stage. We provided the RTL, the constraints and script files and the library technology files. It generates the required verilog netlist in addition to SDF, SDC files and the synthesis reports.

### 4.1.2 Faraday's 130nm

The Faraday library is a 0.13m family standard cell library tailored for UMCs 0.13m High Speed (HS) process. The process employs copper wiring and FSG dielectric. This library offers five (5) to eight (8) metal layers. It is optimized for applications requiring high speed and ultra high density.
8-track (3.2m) cell height has the industrys smallest cell layout area (250K gates/mm 2 ). This librarys optimized drive strength is based on Faradays rich experience of over 1,000 successful ASIC projects. It provides an extensive library database that is easy to manage and use.
The library uses a 1.2v supply voltage for core cells and a 3.3v supply for IO cells. It can operate in a range of -40 to 125 C as shown in Fig 4.1.

| Operating Condition | | Min | Typ | Max | Unit |
|---|---|---|---|---|---|
| VCC | Standard cells | 1.08 | 1.2 | 1.32 | V |
| | 3.3V I/O cells | 2.97 | 3.3 | 3.63 | V |
| $T_J$ | Junction operating temperature range | -40 | 25 | 125 | °C |

Figure 4.1: Operating Condition of Standard cells and I/O Cells

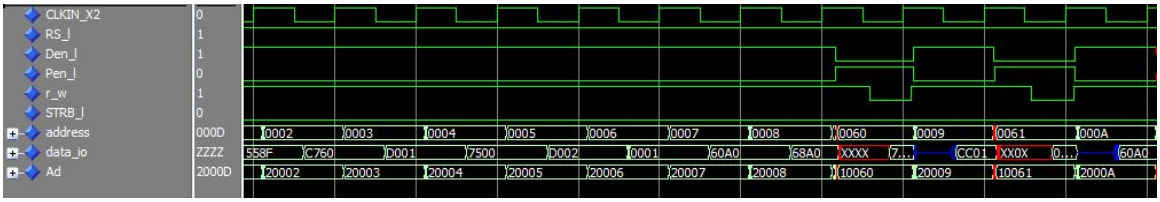
The library consists of three db files (library files input to Design Compiler) one for the typical case (temperature of 25 C) , another for the minimum case and the third is for the maximum case. Each of these libraries contain wire load models to be used for modeling wire resistance and capacitance for different cases and about 400 logic cells.

### 4.1.3 Timing Constraints

**Requirements:**

We were offered a 1425x1425mm chip area from TIEC. And our design was required to operate at a frequency of 100MHz. Our design's timing was further limited by other factors, the first being the memory interface and the second is the limited power available for the pads which would limit the operating frequency to 50MHz.
The memory timing diagrams require that the address is stable before memory control signals become valid. Violating this in case of read operations is harmless, but in case of write operations it might cause data to be written to wrong addresses.
To avoid such erroneous operations our first approach was to use a PLL to introduce phase shifts between the address and control signals, effectively delaying

them after the address is stable. A PLL from our Faraday's library was used to achieve this. This approach was later dropped as the PLL required 2 IO pads, which we could not afford.
Our Second Approach was to extend the write operation to occupy two cycles. Generating the address in the first, holding it for both cycles and generating the control signals in the second. This approach however would limit our performance, and was therefore dropped.
The third approach we used was to activate some memory control signals at the negative clock level. The address and all control signals except RW are output at positive clock edge as normal, then RW signal follows at the negative edge, this gives a time of half clock cycle for the address to be stable. This is shown in Figure below.




# Constraints:

After removing the Internal memories and using only external ones, the area of the architecture was no longer an issue. Our main concern is to meet the timing requirements and be able to achieve the necessary changes for memory interface.
Our design was synthesized at 100MHz, that is a clock cycle of 10ns. Accounting for the input and output delays of each synthesized block, load attached to them, their loads on the driving cells and the clock latency and uncertainty. The RW signal that activates on the negative clock edge was further constrained to make its delay as low as possible. The following is a sample of a written constraints script for one of our designs.

```
read_ddc {mapped/MBM_16x16_radix4.ddc mapped/shifter.ddc mapped/scaling_shifter.ddc
mapped/alu.ddc mapped/descaling_shifter.ddc}

elaborate calu
set_dont_touch [get_cells {ALU32 MUL16 SHIFT SSHIFT DSHIFT}] true

set_svf svf/calu.svf

# Create clock
create_clock -period 10 [get_ports clk]
set_clock_latency .6 [get_clocks clk]
set_clock_transition .3 [get_clocks clk]
set_clock_uncertainty .3 [get_clocks clk]
set_dont_touch_network  [get_clocks clk]

# Create Ideal Network for rst
set_ideal_network [get_ports rst]
set_ideal_latency .6 [get_ports rst]
    set_ideal_transition .3 [get_ports rst]
# Set input delay for all inputs
## All inputs are registered
set_input_delay -clock clk -max .5 [all_inputs]
remove_input_delay [get_ports clk rst]
# Set output delay for all outputs
## Outputs go directly to registers
set_output_delay -clock clk -max .5 [all_outputs]

# Defining loads
## Load of Registered Inputs
set_driving_cell -lib_cell DFFCHD [all_inputs]
remove_driving_cell [get_ports clk rst]

## Load of a DFF
set_load [load_of fsc0h_d_generic_core_tt1p2v25c/DFFCHD/D] [all_outputs]

# Wire loads
set_wire_load_mode enclosed
set auto_wire_load_selection true
set compile_fix_multiple_port_nets true

# Operating Conditions
set_min_library fsc0h_d_generic_core_ss1p08v125c.db -min_version fsc0h_d_generic_core_ff1p32vm4
set_operating_conditions -max WCCOM -max_library fsc0h_d_generic_core_ss1p08v125c
```

```
-min BCCOM -min_library fsc0h_d_generic_core_ff1p32vm40c

# Compile
set_fix_hold clk
set enable_recovery_removal_arcs true
check_design
uniquify
compile_ultra

# Reports
report_constraint -all_violators -verbose > report/DC/calu_constraints.rpt
report_area -hierarchy -physical > report/DC/calu_area.rpt
report_timing -crosstalk_delta -capacitance > report/DC/calu_timing.rpt
report_power > report/DC/calu_power.rpt
report_cell > report/DC/calu_cell.rpt

# save designs
write -hierarchy -format ddc -output "mapped/calu.ddc"
write -hierarchy -format verilog -output "netlist/calu.v"
write_sdc sdc/calu.sdc
write_sdf sdf/calu.sdf
```

### 4.1.4 Synthesis Results

Some of our blocks have slightly missed the required timing requirements, working at 100 MHz, but since we have reduced our operating frequency to 50MHz this was no longer a problem. The core achieved an area of 65519.35, and a total power consumption of 0.8823 mW (without the clock tree). The following is a portion from our generated reports.

```
                 Internal      Switching      Leakage        Total
Power Group      Power         Power          Power          Power   (  %  ) Attrs
--------------------------------------------------------------------------------
io_pad           0.0000        0.0000         0.0000         0.0000  (  0.00%)
memory           0.0000        0.0000         0.0000         0.0000  (  0.00%)
black_box        0.0000        0.0000         0.0000         0.0000  (  0.00%)
clock_network    1.6403e-05    1.0770e-05     4.4961e+03     3.1669e-05  (  0.00%)
register         0.6157        1.9449e-02     3.8482e+06     0.6390  ( 72.42%)
sequential       3.0129e-02    2.5390e-03     2.1684e+06     3.4836e-02  (  3.95%)
combinational    6.1624e-02    0.1341         1.2753e+07     0.2085  ( 23.63%)
--------------------------------------------------------------------------------
Total            0.7075 mW     0.1561 mW      1.8774e+07 pW  0.8823 mW
1
```

| | | |
|---|---|---|
| clock CLKIN_X2 (rise edge) | 10.00 | 10.00 |
| clock network delay (ideal) | 0.60 | 10.60 |
| clock uncertainty | -0.30 | 10.30 |
| cntrl_path/IR_reg_reg[12]/CK (QDFFRSBEHD) | 0.00 | 10.30 r |
| library setup time | -0.12 | 10.18 |
| data required time | 10.18 | |
| data required time | 10.18 | |
| data arrival time | -9.48 | |
| slack (MET) | 0.70 | |

## 4.2 Pads and Power

### 4.2.1 Physical Design

[2] Converting gate-level netlist to physical layout circuit representations of the components (devices and interconnects) of the design are converted into geometric representations of shapes which, when manufactured in the corresponding layers of materials, will ensure the required functioning of the components. This geometric representation is called integrated circuit layout. This step is usually split into several sub-steps, which include both design and verification and validation of the layout.

### 4.2.2 Design Planning

Efficient design implementation of any ASIC requires an appropriate style or planning approach that enhances the implementation cycle time and allows the design goals such as area and performance to be met. There are two style alternatives for design implementation of an ASIC , flat or hierarchical

1. Flat
    - Small to Medium ASIC
    - Better Area Usage Since no reserve space around each sub-design for power/ground
2. Hierarchical
    - For very large design

- When sub-systems are design individually
- Possible only if a design hierarchy exist

## Floor Planning

It is the first step of physical layout implementation a floor plan should include the following decisions :-
- Size of layout
- Core area
- Placement of I/O pad and io pins
- Placement of hard macros

## Step 1 Size of the layout:-

The first step in floor planning is to define the outline of the layout. If the layout is rectangular, only the length and the width of the layout are required.

## Step 2 Core Area :-

The core area is usually defined by specifying the distance between the edge of the layout and the core( core utilization to all chip)

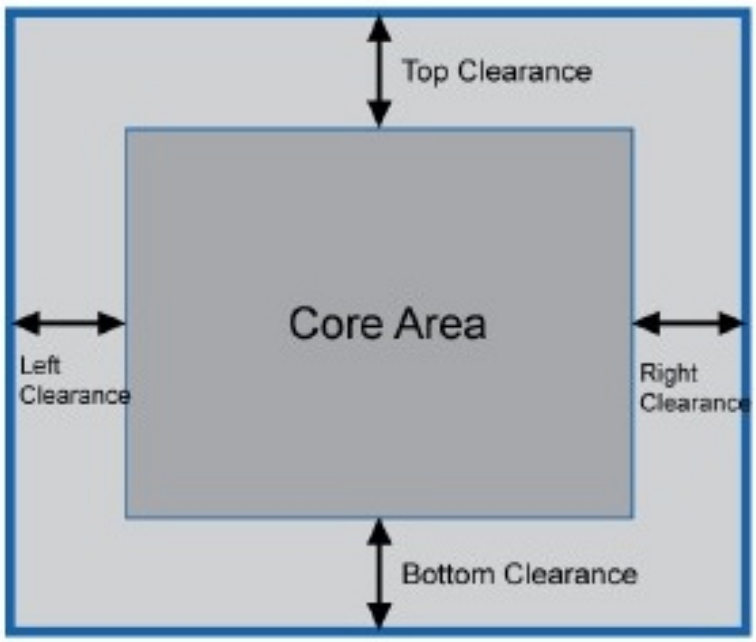



Figure 4.2: core Area

All standard cells must be placed in the core area. I/O pads and macros do not have this restriction although it is common to place macros in the core area. The area outside the core can be used to place the I/O pads, the I/O pins, and the core power rings.
Standard cells are placed in rows, similar to placing books on a book shelf. The rows are called cell rows and are drawn inside the core area. All cell rows have the same height.

## Step 3 Placements of IO Pads and IO Pins Geometries:-

For a chip-level layout, the next step is to place the IO pads. The P&R tool can fill the gaps between the pads with pad filler cells and corner cells. For a block-level layout, the user needs to define the location and geometries (size and metal layer) of every IO pin
I/O pad circuits translate the signal levels used in the ASIC core to the signal levels used outside the ASIC. Additionally, the I/O pad circuits clamp signals to the power and ground rails to limit the voltage at the external connection to the ASIC I/O pad. This clamping reduces signal overshoot and prevents damage from Electrostatic Discharge (ESD).

## good I/O system has the following properties:-

- Drives large capacitances typical of off-chip signals
- Operates at voltage levels compatible with other chips
- Limits slew rates to control high-frequency noise
- Protects chip against damage from electrostatic discharge (ESD)
- Protect against over-voltage damage
- Has a small number of pins (low-cost)
- Pad consists of a square of top-level metal o on a side that is either soldered to bond wire connecting to a package or coated with lead solder ball.
  Pad refers to metal square only or to the complete I/O cell containing the metal, ESD protection circuit, and I/O transistors.
  Also sometimes it contains built in receiver and driver circuits to perform level conversion and amplification.
  Pad size is defined usually by the minimum size to which a bond wire can be attached.

- The spacing of the pads is defined by the minimum pitch at which bonding machines can operate.

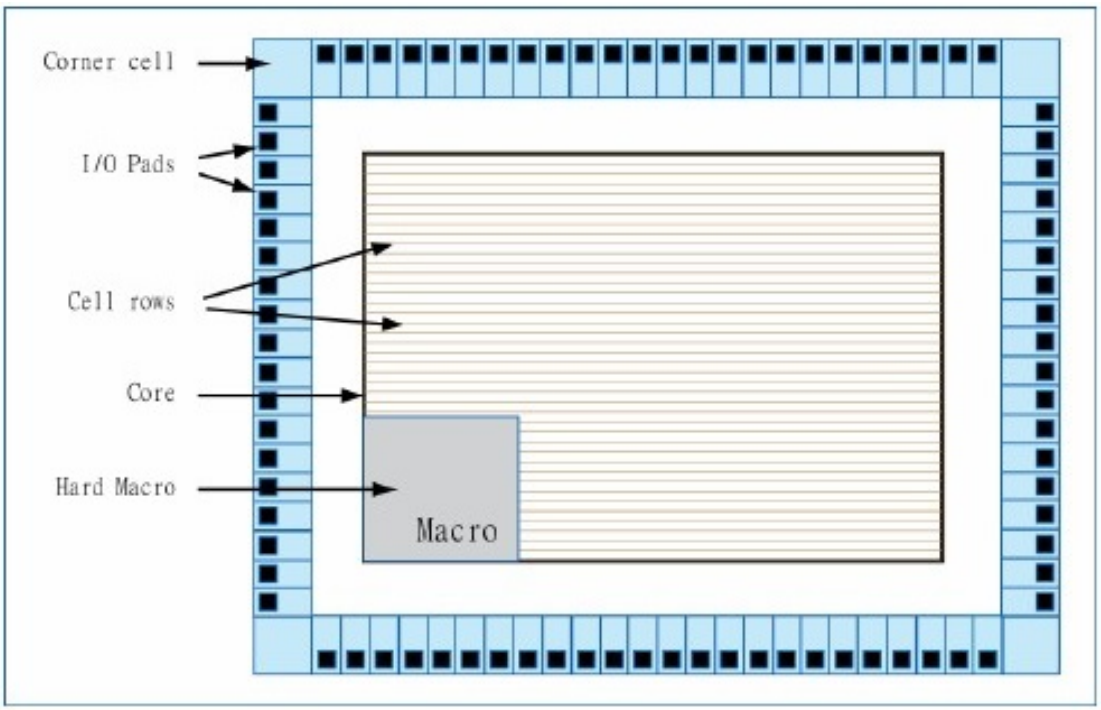



Figure 4.3: Floarplanning

## There are three types of ASIC input and output pads:

- General-purpose input and output pads
- Power and ground pads
- Special purpose input and output pads

Power and ground pads provide connections to the various ASIC power and ground busses. The metal connections from the pad to the power or ground bus within the power and ground pad or to the ASIC core are made as wide as possible and on as many metal layers as practical to minimize their resistance and maximize the current carrying capacity.

Special purpose I/O pads encompass cells with unusual or especially stringent requirements, or cells that are primarily analog. Some examples are crystal oscillator cells, Universal Serial Bus (USB) transceivers, Peripheral Component Interconnect (PCI), Low-Voltage Differential Signaling (LVDS), and isolated analog signal, power, and ground cells

It is critical to functional operation of an ASIC design to insure that the pads have adequate power and ground connections and are placed properly in order to eliminate electromigration and current-switching noise related problems

## Output pads:-

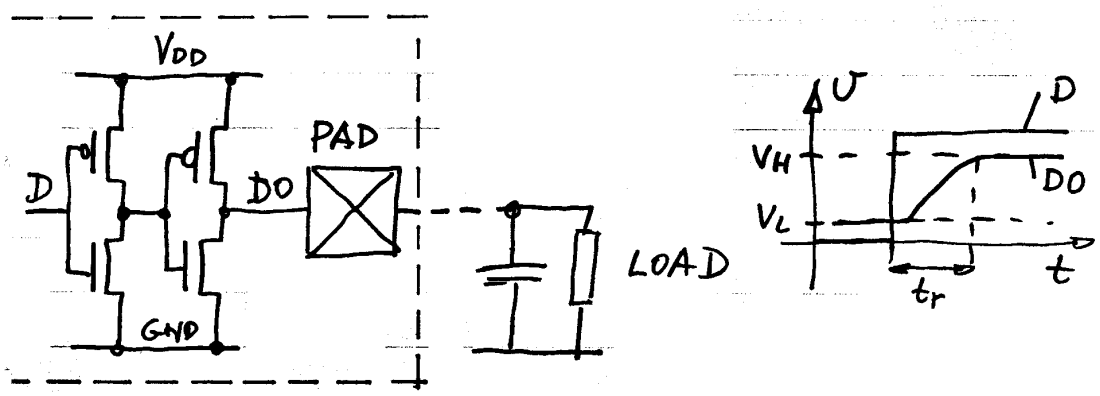



Figure 4.4: Output Pad Circuit

Drive large off-chip loads (2 50 pF) With suitable rise/fall times Requires chain of successively larger buffers Guard rings to protect against latchup Large nMOS output transistor Output Pads must have sufficient drive capability to deliver adequate rise and fall times into given capacitive load.
If the pad drives resistive load it must also deliver enough current to meet the required DC transfer characteristics.
Also, output pads generally contains adequate buffering to reduce the load seen by the on-chip circuitry driving the pad.

**Inputs pads:-**

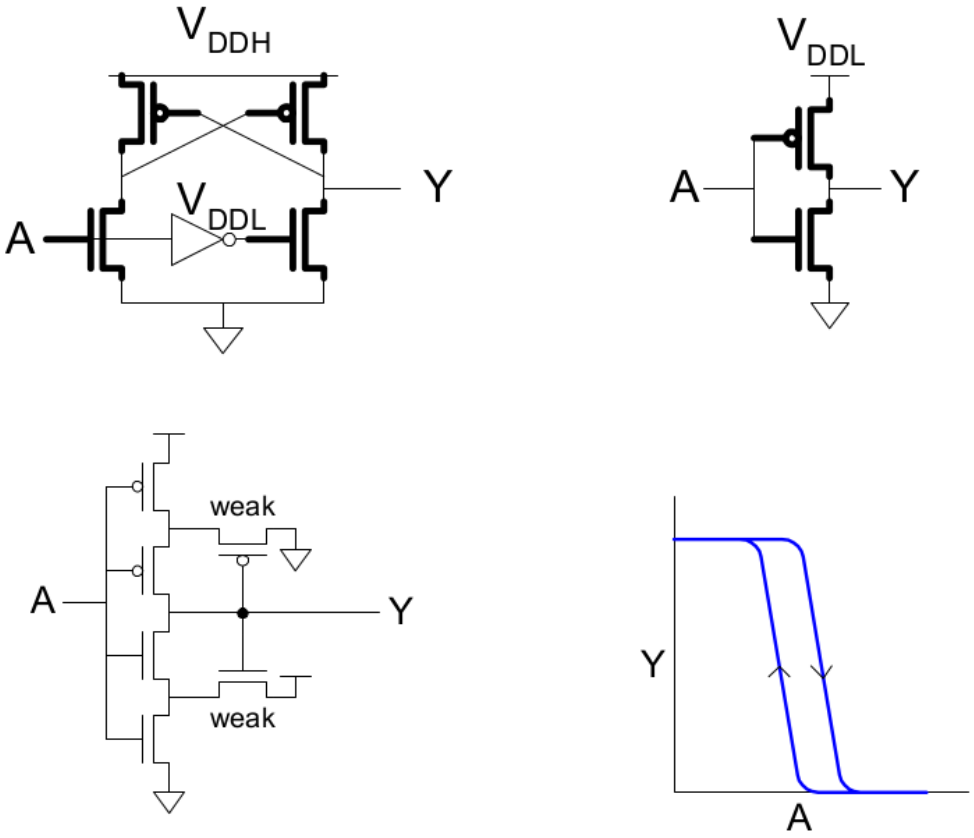



Figure 4.5: Input Pad Circuit

Level conversion Higher or lower off-chip V
Noise filtering Schmitt trigger Hysteresis changes VIH, VIL Protection against electrostatic discharge

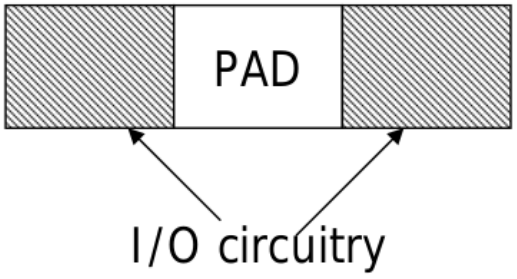
## Input pads ESD protection

Input pads have transistor gates connected directly to the external world. These gates are subject to damage from electrostatic discharge that can puncture and break down the oxide A protection circuit from diodes and resistors is used to clampthe input signal
Pads could be designed according to following criteria:

- Core-limited
- Pad-limited

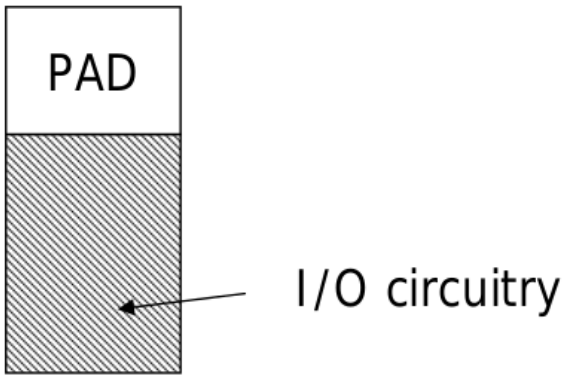
## Core limited:

The internal core of the chip determines the size of the chip, so thin pads are required.
The input/output circuitry is placed on either side of the pad




## PAD limited:

The input/output circuitry is placed toward the center of the chip



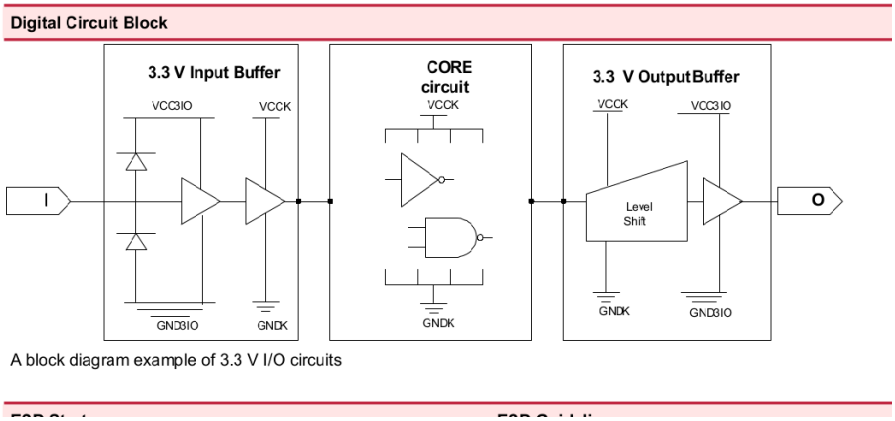



Figure 4.6: A block diagram of 3.3 V I/O circuts

## Some cosideration to be taken when design pads:-

## Electro-Migration (EM)

The Electro-Migration (EM) effect occurs when the Direct Current (DC) density in a power supply line is too high, causing the electron flow to push around the metal grains and wear out the metal line during the lifetime of a chip. The EM effect on a power cell results when the high current demanded by the I/O buffers flows from the power pad through the metal wire of the power lines, inducing the EM effect The EM effect on the ground cell is a similar phenomenon; the current flows back to the ground pad through the metal wire, causing the EM effect. Therefore, the EM effect on the power/ground depends on the current-carrying capacity of its metal wire, and the current-carrying capacity depends on its metal wire width

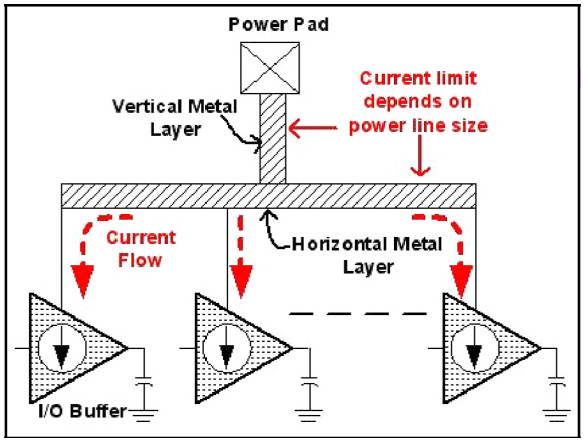



Figure 4.7: EM

### SSO Noise Effects

SSO Push-out:

the extent of the propagation delay caused by the SSO noise during the simultaneous switching of several drivers, which is an important concern in the synchronous circuits. The maximum delay is due to the maximum allowable simultaneous switching output buffers. Different numbers of simultaneous switching output buffers have different delay variations.

False Toggle in Non-Switching Output:

the false state of quiet buffers as a result of SSO noise, which is an important concern in asynchronous circuits. Higher noise is tolerable at the mid-cycle, as long as the voltage settles within the sampling time. Please note that the maximum tolerable noise (The maximum noise immunity of the input receivers) not only depends on the switching noise pulse amplitude, but also on its width.

## ESD

- Your body has a capacitance of a few hundred pF
- You can become charged with static electricity to 10kV
- ESD protection structures dissipate the ESD and prevent gate overvoltage
- The ESD protection structure must be able to dissipate an energy of CV2=10mJ

#### to Calculate I/O-to-P/G Ratio by Considering Electro-Migration

The I/O-to-Power/Ground (I/O-to-P/G) ratio represents the number of I/O buffers that can be connected to one power or ground pad. The operating frequency and loading capacitance of the I/O buffer should be taken into consideration if you want to calculate the I/O-to-P/G ratio with consideration to the EM.
That is, the loading capacitor must be charged and discharged at a rate that is equal to the operating frequency A lot of current is required if many buffers switch simultaneously, and the metal wire of the power or ground pads must have enough current-carrying capacity in order to avoid the EM effect. Suppose that N buffers are connected to one power/ground pad pair Current required N buffer: I = NCdv/dt = NCVf where f is the operating frequency, V is the operation voltage, and C is the capacitor load

### 4.2.3 Chip specifications

**Step 1 Size of the layout**

Rectangular floor plan with size 1525*1525

**Step 2 Core Area**

The core area to all chip 0.1

**Step 3 Placements of IO Pads**

We use pad limited type
Pad size = 75*63
Pad spacing =60
we need 4 power and ground to supply core on chip
f= 75mhz
c=8pf
v=3.3v
i= 102.4 mA

we need 3 power and ground to supply input,output pads
one to supply input buffers it can supply up to N=102.4/75mhz*3.3v *8pf=314 at least
one to supply output buffers it can supply up to N=102.4/75mhz*3.3v *8pf=314
one to supply bidirectional buffers it can supply up to N=102.4/75mhz*3.3v *8pf=314

## Buffers and pads used from tech library:-

XMHA Programmable 3.3V input buffer, pad limit
YA4GSHA 4 to 16 mA programmable 3.3V output buffer, pad limit
VCCKHA Power supply for internal core cells and I/O to core interfaces, pad limit
VCC3IHA Power supply for 3.3V input buffers and output pre-drivers, pad limit
VCC3OHA Power supply for 3.3V output buffers and input ESD protection, pad limit
VCC3IOHA Power supply for 3.3V input/output buffers, pad limit
GNDKHA Ground for internal core cells and I/O to core interfaces , pad limit
GND3IHA Ground for input buffers and output pre-drivers, pad limit
GND3OHA Ground for output buffers and input ESD protection, pad limit
GND3OHA Ground for output buffers and input ESD protection, pad limit

### Power Planning:-

The next step is to plan and create power and ground structures for both I/O pads and core logic. The I/O pads power and ground busses are built into the pad itself and will be connected by abutment.
For core logic, there is a core ring enclosing the core with one or more sets of power and ground rings. A horizontal metal layer is used to define the top and bottom sides, or any other horizontal segment, while the vertical metal layer is utilized for left, right, and any other vertical segment. These vertical and horizontal segments are connected through an appropriate via cut. The next consideration is to construct the standard cell power and ground that is internal to the core logic. These internal core power and ground busses consist of one or two sets of wires or strips that repeat at regular intervals across the core logic, or specified region, within the design. Each of thesepower and ground strips run vertically, horizontally, or in both directions.

## The key consideration for power planning is:

- an acceptable IR-drop from the power pads to all power pins
- meeting electro-migration requirements
- does not result in routing congestion
- compact layout

A power plan consists of several types of power structure.
Figure 4.8 illustrates a typical sequence to construct the power structures.

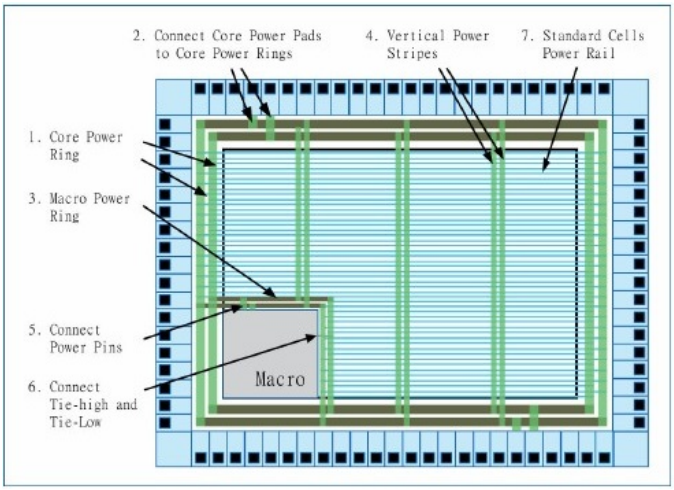



Figure 4.8: illustrates a typical sequence to construct the power structures.

**How to construct the power structures?**

1. core power rings are routed first
2. core power pads are connected to the core power rings
3. the power rings are added around the macros where necessary
4. vertical stripes and horizontal stripes are added to reduce the IR- drop at the power rails of the standard cells and the macros
5. the power pins of the hard macros are tapped to the core rings or the power stripes
6. if tie-high and tie-low cells are not used, the tie-high and tie-low inputs to the hard macros and IO pads are tapped to the power structures
7. the power rails for the standard cell are added to the power plan The power rails can tap the power from the core power rings, the power stripes and the macro power rings

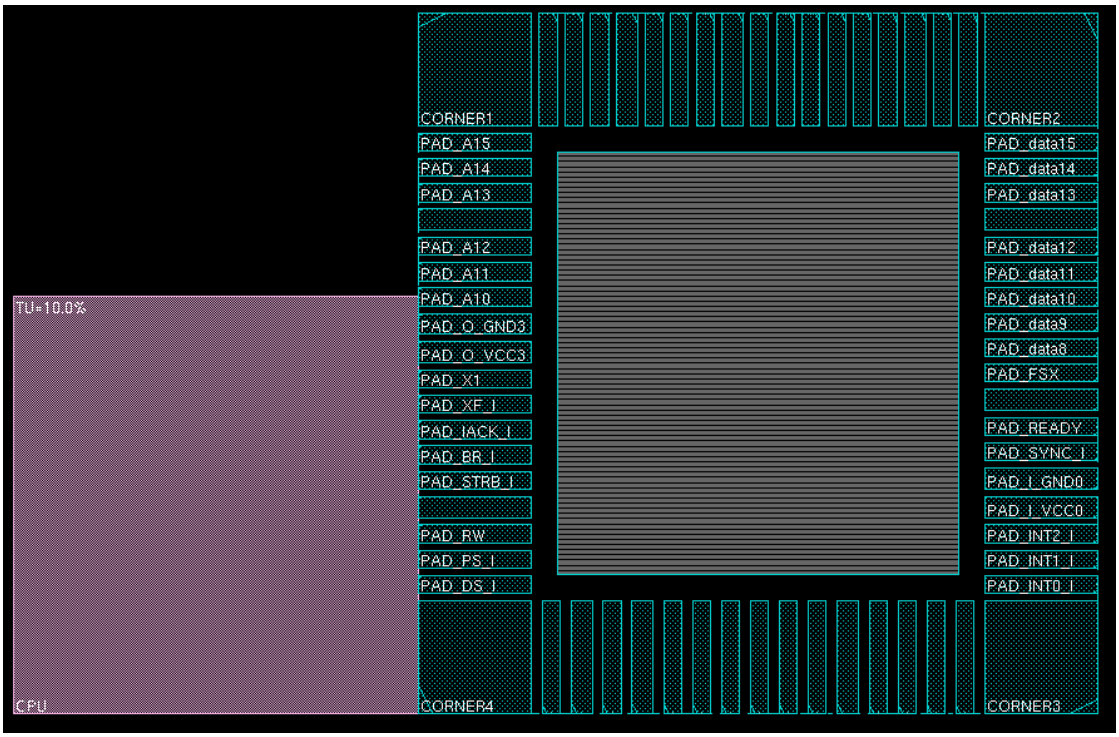



Figure 4.9: pads

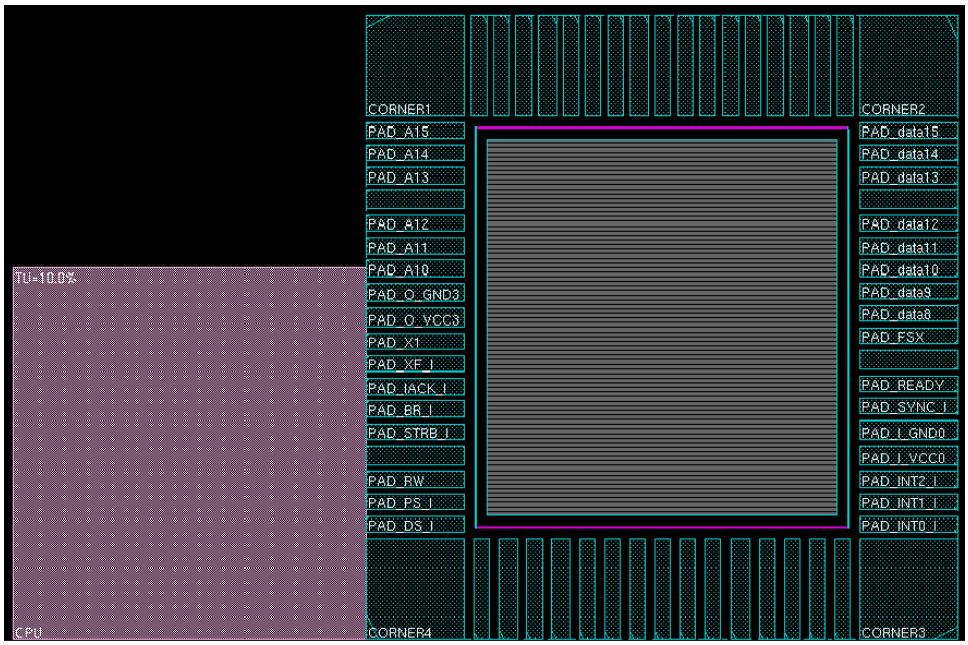



Figure 4.10: Rings

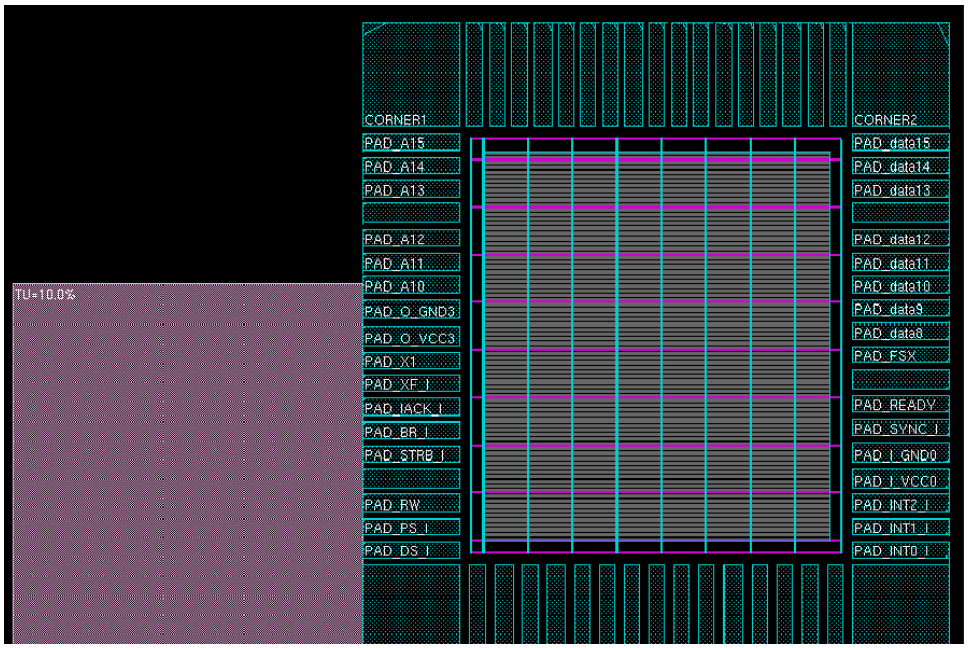



Figure 4.11: STRIPS

Power ring and power strips are made in metal 7,8 Width =0.4 Space=0.4 according to the tech files

### 4.2.4 Bonding pads

Typically it is composed from all the metal layers stacked on top of each other and connected through vias; which allows efficient connection between the IO pads pins through the package leads. The connections between the Bonding pads and the package leads are achieved through sufficient low resistance wire (Gold or Aluminum ), using Aluminum is cheaper than Gold but with a lower performance than Gold,

| Material | Wire diameter (um) | Min. pitch (um) |
|---|---|---|
| Gold (Au) | 18-76 | 35 |
| Aluminum (Al) | 20-51 | 60 |

Figure 4.12: Aluminum and Gold Diameter

The above dimensions is obtained from Quik-Pak Microelectronic Packaging & Assembly Solutions
After comparing Gold allowed pitch with Aluminum we find that using gold will allow us to use higher number of IO pads using same area.
For our design we were allowed to use only 90um Pad Pitch but we were allowed to use a Bonding Pad Over Circuit (POC) configuration, a cross section plot of a POC configuration is given in the next figure.

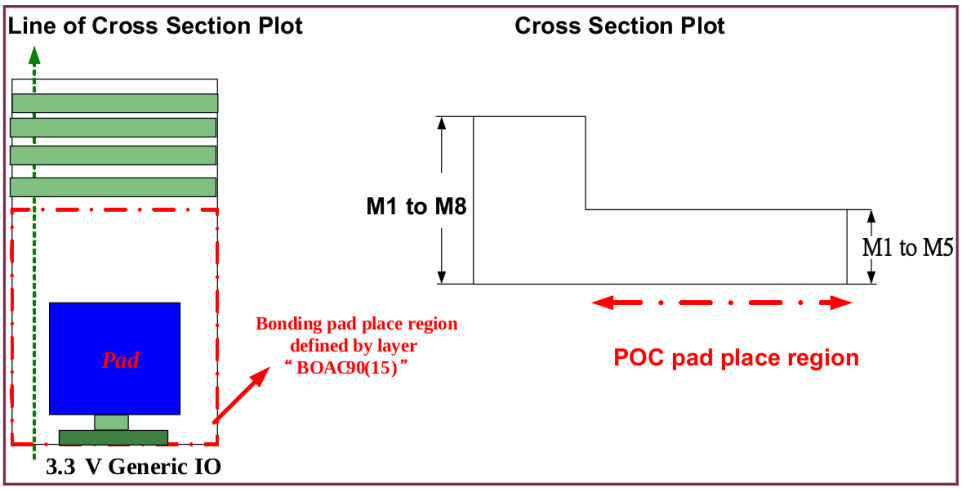



Figure 4.13: Cross Section plot of a POC configuration

Given that we are using:

1. 1525x1525 chip size
2. 60x152 IO pad
3. 47x72 Bonding pad

By calculating the number of Bonding pads we could use for each side:

First, we subtract 1525um (side length) from 152umx2(IO pad length at each side):
1525-152x2=1221

second, we calculate the largest number of bonding pads could be added in each side:
60x13+30x12=1140

having 13 Bonding pads in each side allows us to use 52 Bonding pad.

From the above calculation we found that there are 60um spaces between the IO pads, in which it is highly recommended to insert the empty guard ring cells with the built-in vertical guard ring in the boundary of the power domain. The empty guard ring cells cells with the built-in vertical guard ring are used to prevent the latch-up issue. figure 4.14 illustrates the usage of the empty guard ring cells cell with the built-in vertical guard ring.

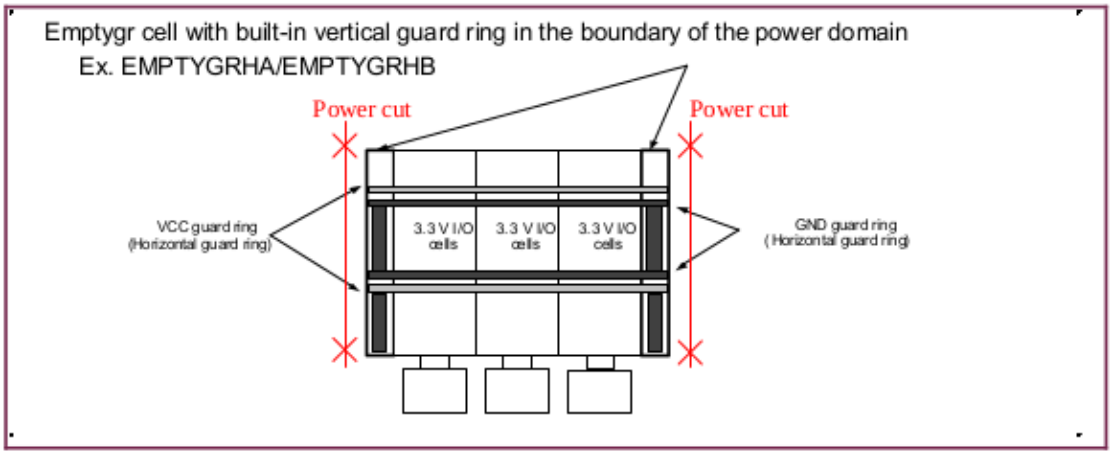



Figure 4.14: illustrates the usage of the empty guard ring cells cell with the built-in vertical guard

## 4.3 Layout

### 4.3.1 Technology Files

The provided technology files are UMC's 130 nm Faraday development kit.
The kit contains three different technology files suited for different purpose.
They are High Speed (HS), Low Leakage (LL) and Standard Process (SP).
All cores fabricated on the same wafer must use the same technology.
All teams participating in FabCat Academic round used HS process, so did we.
HS process supports up to 8 metal layers.

Figure 4.15 shows the characteristics of the process.
The library contains a wide variation of core cells, Figure 4.16 lists the types of core cells in the kit.

| Characteristic | Description |
|---|---|
| Technology | UMC's 0.13µm logic, 1.2V / 3.3V high speed, FSG dielectric, CMOS process |
| Minimum drawn channel length | 0.12µm |
| Supply voltage | • 1.08V to 1.32V for core cells<br>• 2.97V to 3.63V for 3.3V I/O cells |
| Metal layer option | • Five (5) to eight (8) layer options<br>• Please refer to Table 3 below for more details |
| Performance | Td = 17 ps / stage (measured from 101 stage inverter ring in typical process and operated under 1.2V, 25℃) |
| Gate density | 250K gates / $mm^2$ |
| Power consumption | 6 nW / MHz / gate (measured from 2-input NAND, output load = 2 standard load, in typical process and operated under 1.2V, 25℃) |

Figure 4.15: HS UMC 0.13 $\mu m$ characteristics

| Cell Group | Function Description |
|---|---|
| AN | AND gates |
| AO | AND into OR complex gates |
| AOI | AND into NOR complex gates |
| DEL | Delay cells |
| FA / HA | Half and full adders |
| MAO / MOA | Complex gates |
| MUX / MXL | Multiplexers |
| ND | NAND gates |
| NR | NOR gates |
| OA | OR into AND complex gates |
| OAI | OR into NAND complex gates |
| OR | OR gates |
| XNR / XOR | Exclusive NOR and OR gates |
| BUF / BUFT / BUFB | Buffers / tri-state buffers |
| INV / INVT / INVB | Inverters / tri-state inverters |
| DF / DBF / QDF | D-type flip-flops |
| DFE / DFZE | Enable flip-flops |
| DLA / DBA / QDLA / QDBA | Latches |
| CKLD | Clock load cells |
| TIE | Tie 0 / Tie 1 cells |
| GCK / GCB | Gating-clock latches |

Figure 4.16: HS UMC 0.13 $\mu m$ core cells

Special cells are also included,as the clock load cell that is used to solve clock skew problems, delay cells to delay a signal when necessary, variable sized filler cells used to fill inter cell free spaces, and Tie-High and Tie-Low cells that are used to force constant high or low voltage respectively.
There are other special cells included in the library but they are not used in our layout.
All the Input-Output (IO) pads are programmable. They have a programmable schmitt trigger and a programmable pull-up/down resistors.
We used the schmitt trigger to provide noise free input and output from the chip, at the expense of the increased delay.
pull-up/down resistors are not used since the memory interface doesn't requre the bus to be pulled to any level.
The drive capability (driving current) for output and input/output (I/O) pads is also programmable for the values of $\{2mA, 4mA, 8mA, 12mA, 16mA\}$. We have chosen the $8mA$ driving current since it's sufficient for driving the memory pins,

higher driving current needs more power pads. Figure 4.17 shows the I/O pad which can be considered as a general case for input and output pads.

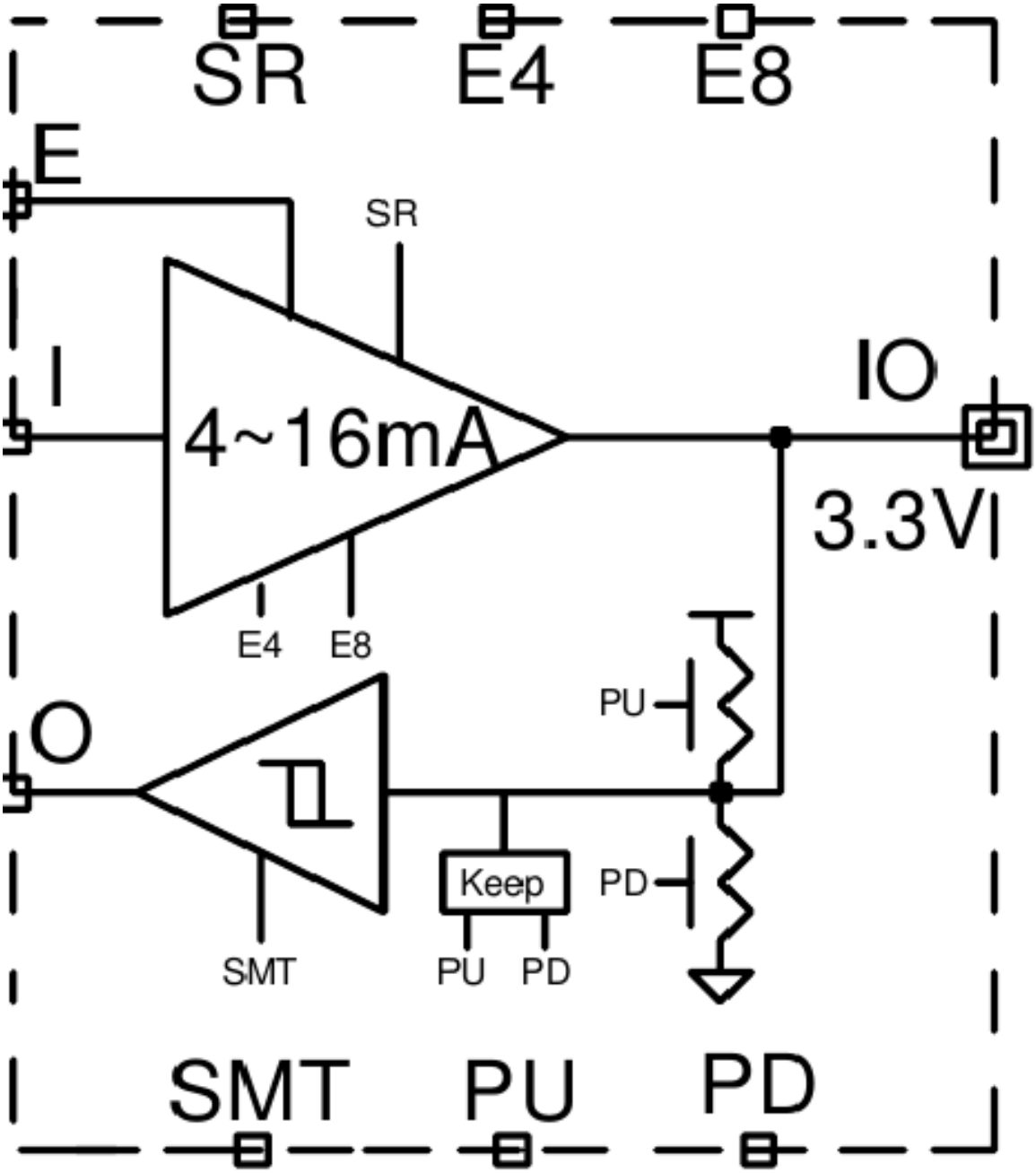



Figure 4.17: Schematic of ZMA4GS bid-directional buffer

### 4.3.2 SoC Encounter

The tool used in the layout process is Cadence's SoC Encounter. SoC Encounter is the most famous EDA tool used in automated digital design layout.

The inputs to the tool is the LEF files, lib files, verilog netlist, timing constraints and pad placement file. LEF file is one of the technology files that provide information about the top view of the core and IO cells. This information include input and output port names and location, metal layers contained within the cell and the cell's total area. lib files includes the timing information of the cells. The verilog netlist is the output file of the synthesis tool. It describes the gates contained in the design and their connection. Timing constraints file describes the timing information of each cell in the netlist. Pad placement file describes the type and location of each IO pad. Outputs files form Encounter are the GDSII stream, timing constraints and various reports, The GDSII file is a description of the layout in terms of layers and masks. The timing constraints file is used for post-layout verification. The reports includes design rule checks (DRC) to check that the layout is consistent with the process's constraints. The reports also includes timing, gate count, clock tree and power reports.

Figure 4.18 shows the used flow in Encounter. The remaining subsections in this chapter describe the design flow on SoC Encounter.

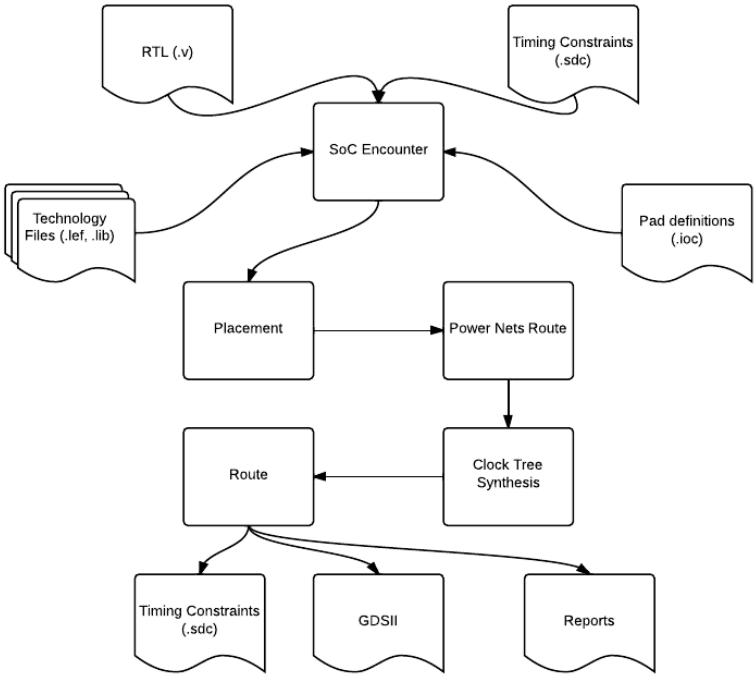



Figure 4.18: The design flow we used in SoC Encounter

### 4.3.3 Core and Pads Placement

Placement is the process constraining the die area, then specifying the location of each logic block. The location of each logic block will greatly influence the wire latency and area. We first placed the IO pads and then let the tool decide the best placement for core logic in order to optimize the latency and the area.
IO pads can be divided into 4 groups; Data, address, control and power pads.
Pads are arranged in a way to minimize the crossing of PCB wires that interfaces the chip to the memory.

### 4.3.4 Clock Tree and Power Analysis

Clock tree is responsible for the distribution of the clock throughout the core.
The clock signal should reach the all the memory elements in the same time, so delay logic is inserted on faster paths to slow them down The clock source should also have strong drive capability since it's connected to all memory elements.
In fact the clock and reset signals have the highest fan-out of all other nets.
To increase the drive capability, buffers are inserted in the clock tree.

### 4.3.5 Bonding Pad Placement

Bonding pads are a set of metal layers soldered to the wires connecting the core to the package.
The connect the package to the pad logic(buffers). Bonding pads must be placed such that the pitch to pitch is no less than $90um$.
This results in the necessity of adding empty cells between pads to separate them by a distance greater than the pitch. Empty cells provide continuity in the pad power ring. Cells of different voltage levels are separated by latch up protection pads.
They prevent the latch up that can happen on the chip power up. Pad On Chip (POC) bonding pads are used, the bonding pad is placed directly above the pad logic.

### 4.3.6 Filler Cell and Route

Core filler cells are used to fill any spaces between regular library cells to avoid planarity problems.
The logo "Toledo CU-AU 2014" is placed on the north west corner of the chip.
It helps the fab identify which side is north by rotating the chip until the logo is readable.

Final design routing is done after successful clock tree synthesis. The automated router tries to optimize the wire lengths in order to meet the timing constraints. Figure 4.19 shows the design after wire routing.

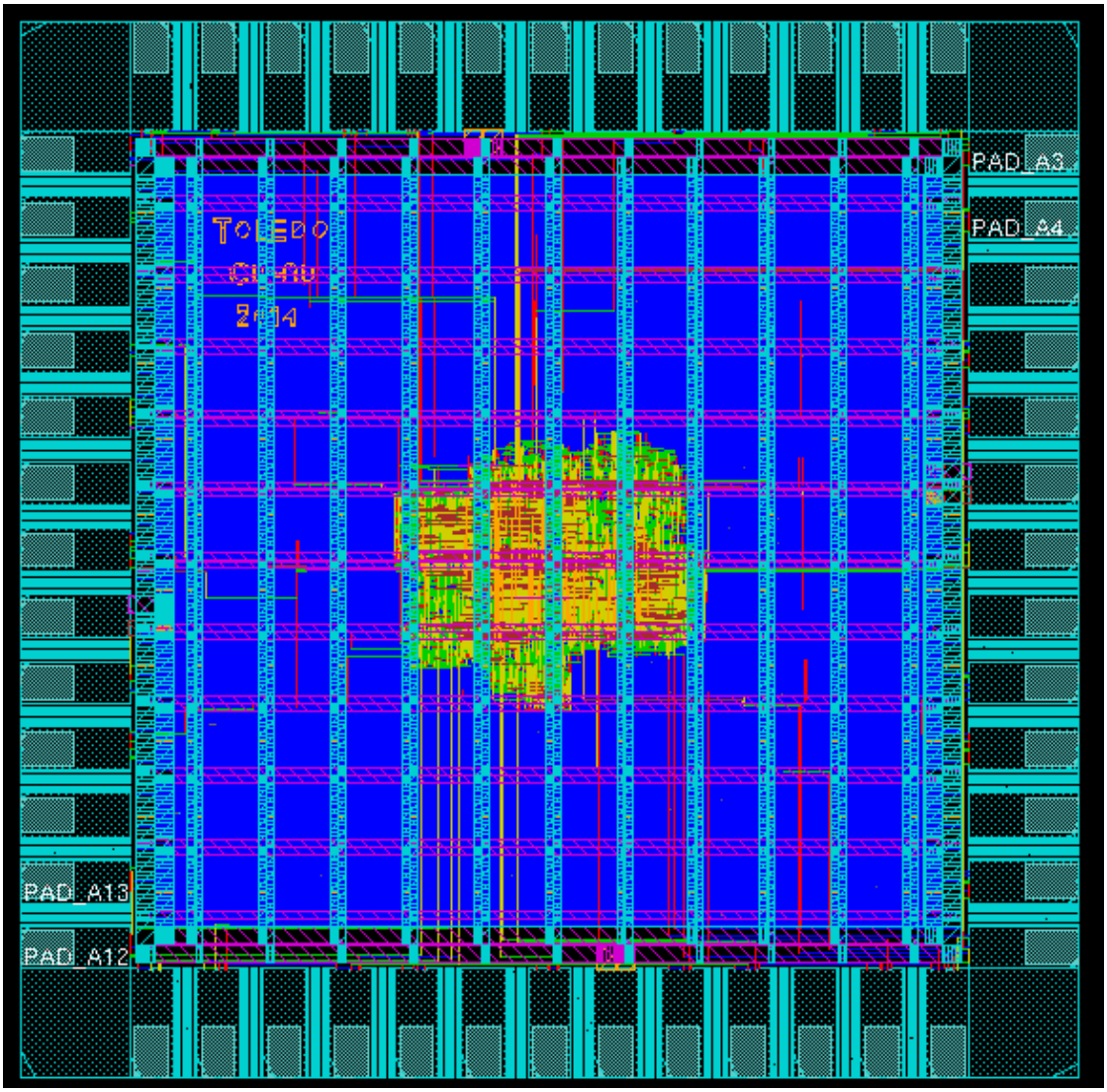



Figure 4.19: Routed design

## 4.4 Tapeout

### 4.4.1 Introduction

#### why we need tapeout

Chip building is time consuming and challenging because there are many things to be considered and much more that can be easily overlooked. On top of such time constraints and complexity is the reality of operating the chip under unforgiving physical rules. If its not done properly, it doesnt work, It is the rules that the fabrication lab gives it to us it is DRC/LVS rules.
The final result of the design cycle for integrated circuits, the point at which the artwork for the photomask of a circuit is sent for manufacture.
First tapeout is rarely the end of work for the design team. Most chips will go through a set of spins in which fixes are implemented after testing the first article.

Many different factors can cause a spin, including:

First tapeout is rarely the end of work for the design team. Most chips will go through a set of spins in which fixes are implemented after testing the first article. Many different factors can cause a spin, including

- The taped-out design fails final checks at the foundry due to problems manufacturing the design itself.
- The design is successfully fabricated, but the first article fails functionality tests.

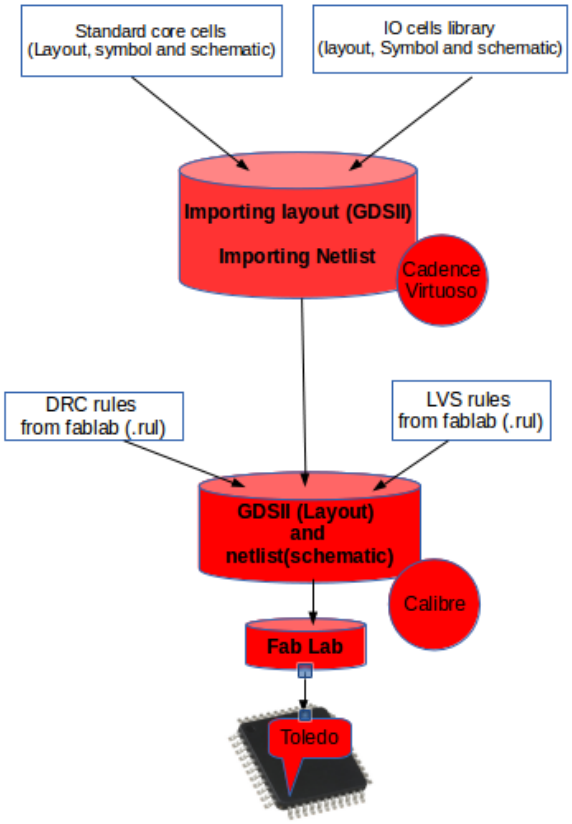



Figure 4.20: Tapeout flow

First define the standard core cells library and the IO cells library by three different views:

1. Layout
2. Schematic
3. Symbol

Second, import the Placed and routed GDSII to a custom design tool (Cadence Virtuoso were used for that), also we import the Netlist Schematic. Finally, import the design from custom design tool libraries to DRC and LVS tool (Calibre were used for this).

### 4.4.2 Defining the standard core cells and IO cells library

The custom design tool must use a reference for the design files you are importing, such that every imported cell has a Layout, symbol and a schematic.

The first refrence for the custom design tool must be the manufacturer techology files used that will be the first thing you will define in the custom design tool. Then it comes to the part that you must import the libraries that define the layout, symbol and schematic. That part is not straight through the custom design tool; you may have to export the schematic of a symbol or a layout to get those views referenced.

Next, is an example of layout view of an inverter cell used in the core and its symbol used followed by I/O cell layout and schematic view:

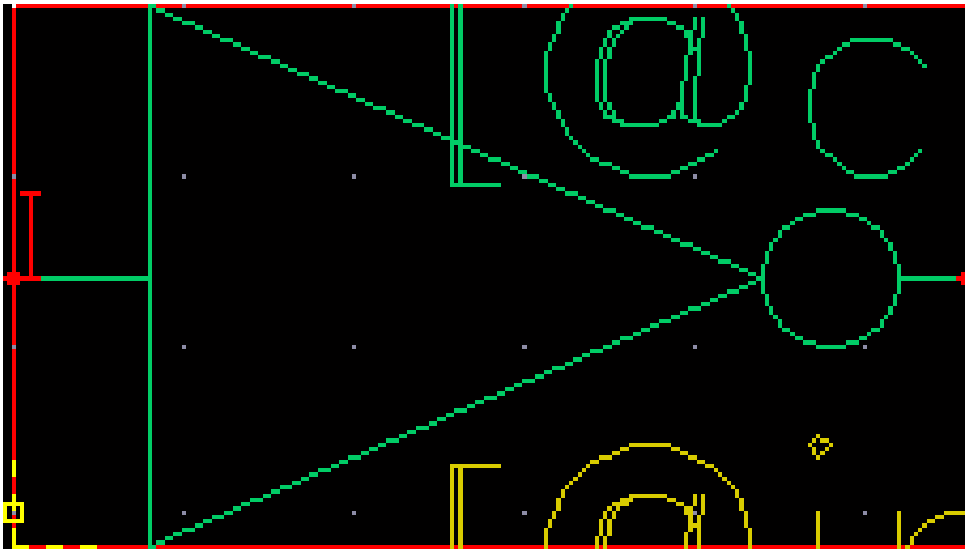


Figure 4.21: symbol view of an inverter cell

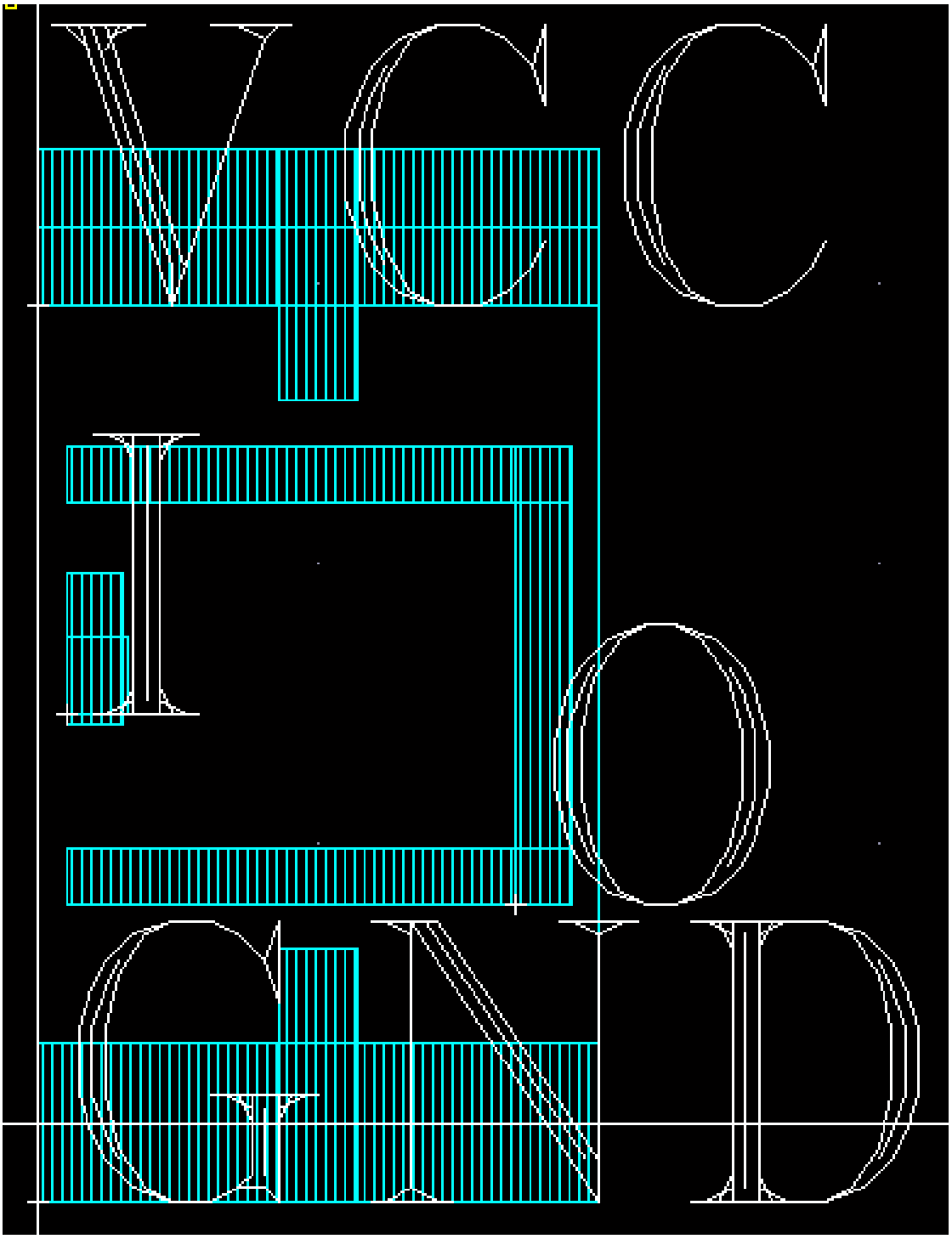


Figure 4.22: Layout view of an inverter cell

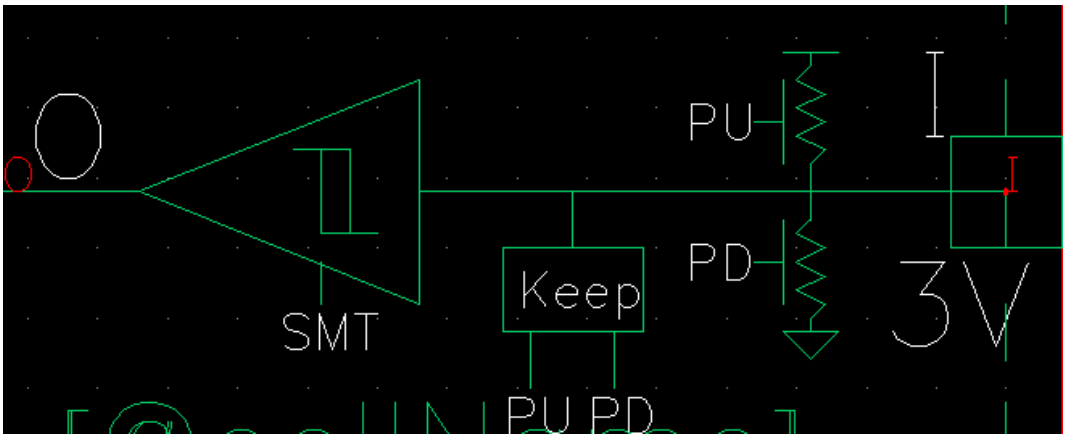



Figure 4.23: symbol view of an I/O cell

## 4.5 Importing GDSII

First, let us define the GDS format and how it look like to the manufacturer, GDS = Graphic Database System Is a database file format which is the factory industry standard for data exchange of integrated circuit or IC layout artwork. It is a binary file format representing planar geometric shapes, text labels, and other information about the layout in hierarchical form. The data can be used to reconstruct all or part of the artwork to be used in sharing layouts, transferring artwork between different tools, or creating photomasks. During tapeout stages the design passes through different operation and checks which causes the GDSII to be modified several times. Next is a Sample of a 3D view of GDSII and the layout of Toledo after exporting the GDS:

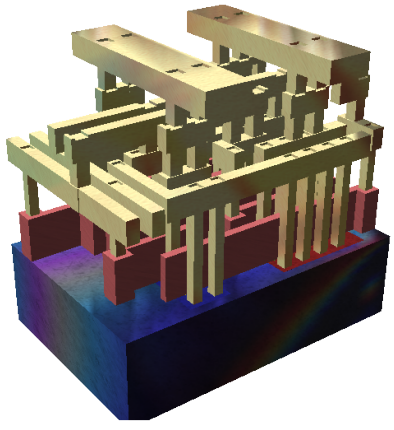


Figure 4.24: GDSII 3D view

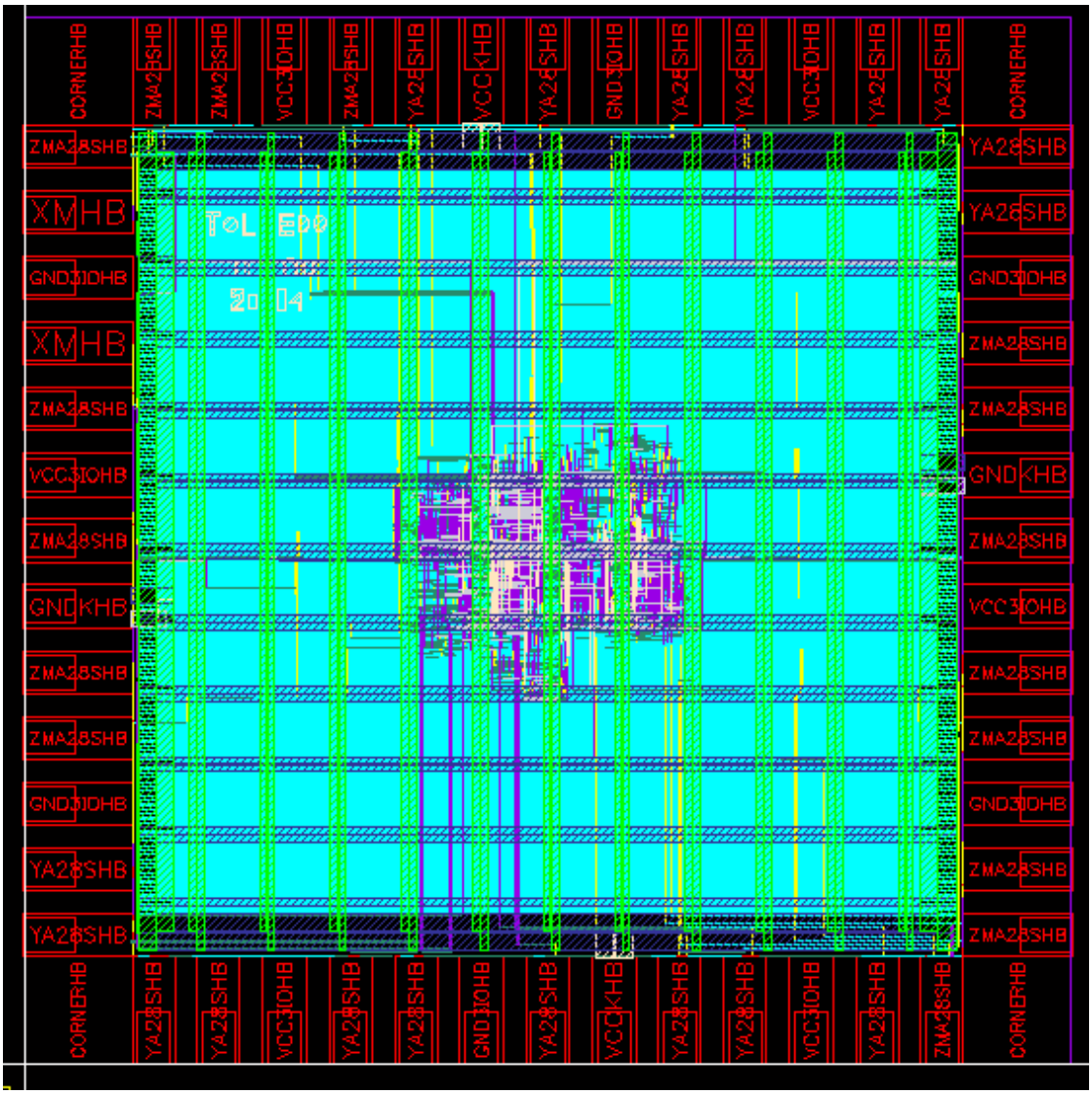


Figure 4.25: Layout view of Toledo

### 4.5.1 Importing netlist

First, when exporting the netlist we must refer to the technology library we used (umc 0.13) you must notice also that the netlist we are exporting must be fully defined; because in the next step(LVS) it will be our reference for comparing with the exported layout and checking that both has the same functionality.
Next is the Netlist of Toledo :

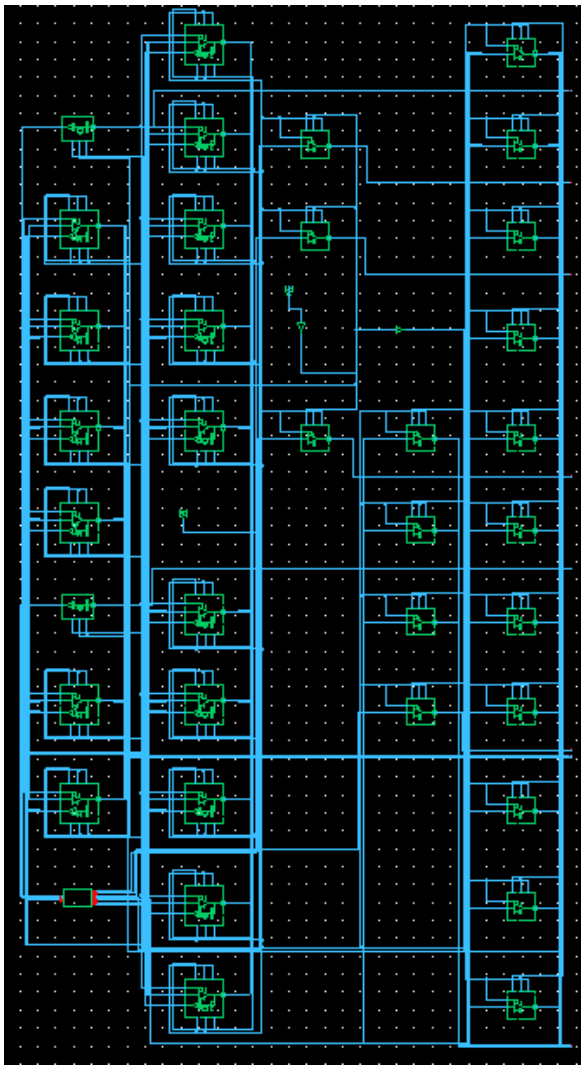


Figure 4.26: Schematic view of Toledo

### 4.5.2 DRC

Design rule check (DRC) examines conformity of layout with geometric rules imposed by the target process.
Next is an example of a DRC error called skewedges:

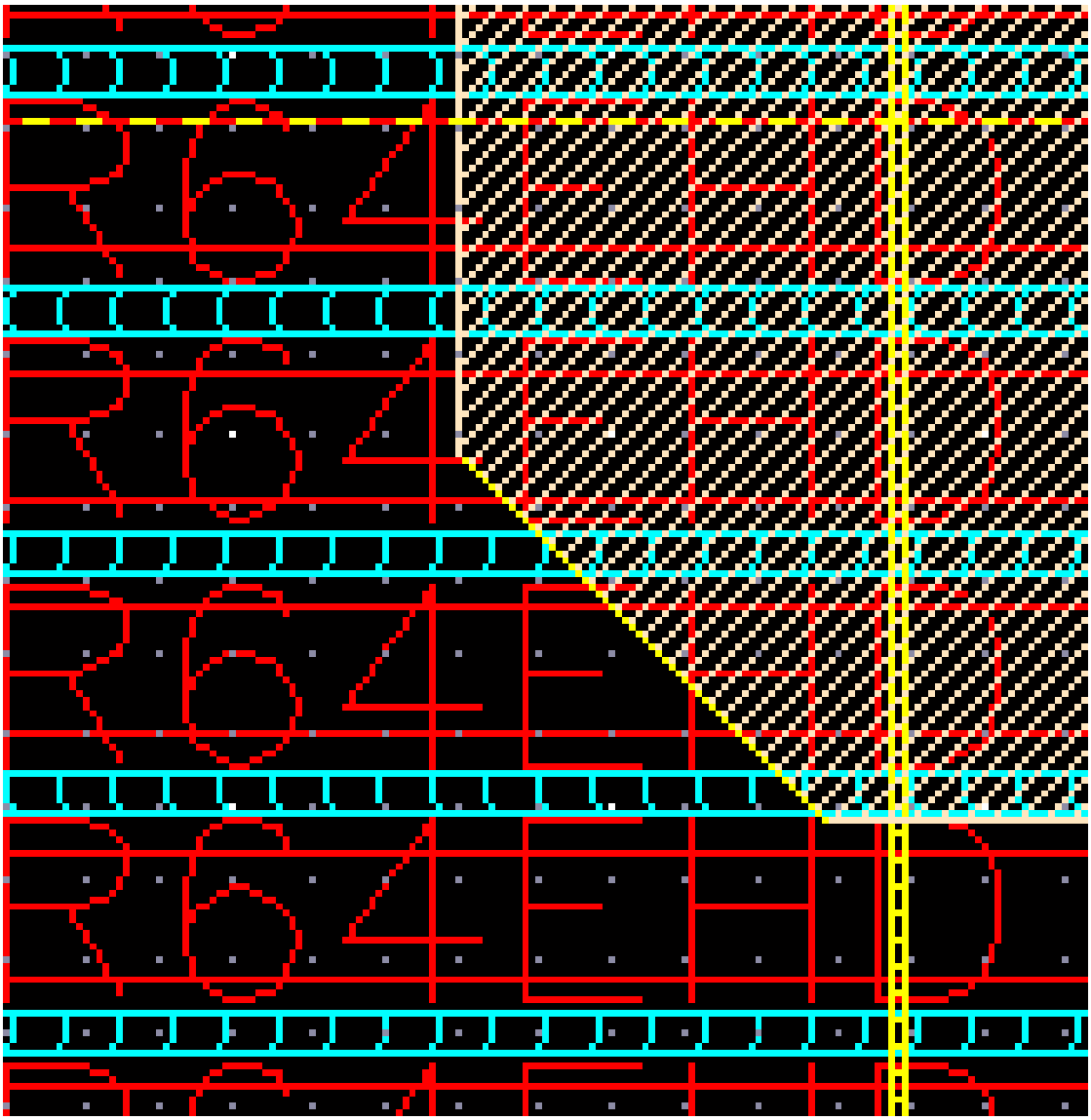


Figure 4.27: Skewedges DRC error

### 4.5.3 LVS

This is the step of comparing the layout generated in the custom design tool (Cadence virtuoso) vs. the netlist.

Common errors in this step is :

- Error: Different numbers of nets.

Here the number of nets in the layout is different from the number of nets in the schematic.

- Error: Different numbers of instances.
  Here the number of instances in the layout is different from the number of instances in the schematic.

This page intentionally left blank.

# Chapter 5

# Verification

## 5.1 What is verification

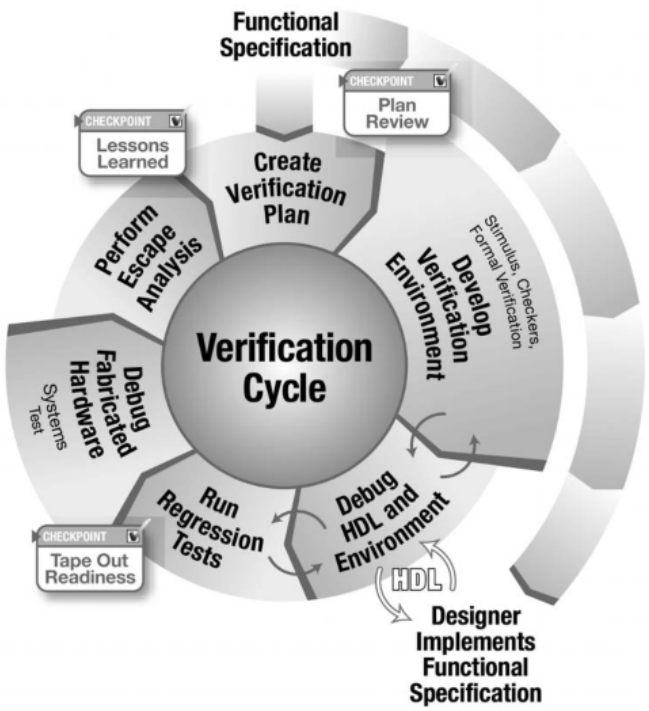



Figure 5.1: verification cycle

Verification is the act of reviewing, inspecting or testing, in order to establish and document that a product, service or system meets the specifications that is meant for Verification ( in Digital Design ) is a process that parallels the process of the

design . A designer reads the hardware specification for a block, interprets the human language description, and creates the corresponding logic in a machine-readable form, usually RTL code [9].
As a verification engineer, you must read the hardware specification, create the verification plan, and then follow it to build tests showing the RTL code correctly implements the features. By having more than one person perform the same interpretation, you have added redundancy to the design process. As the verification engineer, your job is to read the same hardware specifications and make an independent assessment of what they mean. Your tests then exercise the RTL to show that it matches your interpretation .

## 5.2 Why do we need verification

This section talk about the importance of verification and why do we have to spend time and money in verification .
Today, in the era of multi-million gate ASICs and FPGAs, reusable intellectual property (IP), and system-on-a-chip (SoC) designs, verification consumes about 70% of the design effort. Design teams, properly staffed to address the verification challenge, include engineers dedicated to verification. The number of verification engineers can be up to twice the number of RTL designers . Due to the complexity of SOCs and IPs designs these days , the verification process has been created to solve this problem and give methodologies to assure that design is working properly. As we have said that this process takes a lot of time and effort which could reach to 70% of the project time , Parallelism is there to reduce the verification process time .
If efforts can be parallelized, additional resources can be applied effectively to reduce the total verification time. For example, digging a hole in the ground can be parallelized by providing more workers armed with shovels. To parallelize the verification effort, it is necessary to be able to write and debugtestbenches in parallel with each other as well as in parallel with the implementation ofthe design. Providing higher abstraction levels enables you to work more efficiently without worrying about low-level details. Using a backhoe to dig the same hole mentioned above is an example of using a higher abstraction level.

## 5.3 What to Verify ?

One of the most important questions you must be able to answer is:"What are you verifying?" The purpose of verification is to ensure that the result of some transformation is as intended or as expected. For example, the purpose of balancing

a checkbook is to ensure that all transactions have been recorded accurately and confirm that the balance in the register reflects the amount of available funds.


Figure 5.2: Reconvergent paths in Verification

5.2 shows that verification of a transformation can be accomplished only through a second reconvergent path with a common source. The transformation can be any process that takes an input and produces an output.
RTL coding from a specification, insertion of a scan chain, synthesizing RTL code into a gate-level netlist and layout of a gate-level netlist are some of the transformations performed in a hardware design project. The verification process reconciles the result with the starting point. If there is no starting point common to the transformation and the verification, no verification takes place.
What types of bugs are lurking in the design? The easiest ones to detect are at the block level, in modules created by a single person. Did the ALU correctly add two numbers? Did every bus transaction successfully complete?
Did all the packets make it through a portion of a network switch? It is almost trivial to write directed tests to find these bugs as they are contained entirely within one block of the design. After the block level, the next place to look for discrepancies is at boundaries between blocks. Interesting problems arise when two or more designers read the same description yet have different interpretations. For a given protocol, what signals change and when? The first designer builds a bus driver with one view of the specification, while a second builds a receiver with a slightly different view. Your job is to find the disputed areas of logic and maybe even help reconcile these two different views.
To simulate a single design block, you need to create tests that generate stimuli from all the surrounding blocks  a difficult chore. The benefit is that these low-level simulations run very fast. However, you may find bugs in both the design and testbench as the latter will have a great deal of code to provide stimuli from

the missing blocks. As you start to integrate design blocks, they can stimulate each other, reducing your workload. These multiple block simulations may uncover more bugs, but they also run slower.
At the highest level of the DUT, the entire system is tested, but the simulation performance is greatly reduced. Your tests should strive to have all blocks performing interesting activities concurrently. All I/O ports are active, processors are crunching data, and caches are being refilled. With all this action, data alignment and timing bugs are sure to occur. At this level you are able to run sophisticated tests that have the DUT executing multiple operations concurrently so that as many blocks as possible are active. What happens if an MP3 player is playing music and the user tries to download new music from the host computer? Then, during the download, the user presses several of the buttons on the player? You know that when the real device is being used, someone is going to do all this, so why not try it out before it is built? This testing makes the difference between a product that is seen as easy to use and one that locks up over and over. Once you have verified that the DUT performs its designated functions correctly, you need to see how it operates when there are errors. Can the design handle a partial transaction, or one with corrupted data or control fields? Just trying to enumerate all the possible problems is difficult, not to mention how the design should recover from them. Error injection and handling can be the most challenging part of verification.

## 5.4 Different kinds of Verification

There are different kinds of verification , There is :

1. Functional Verification
2. Formal Verification
3. Static timing Verification

- We will focus on Functional Verification in this book

## 5.5 Functional Verification

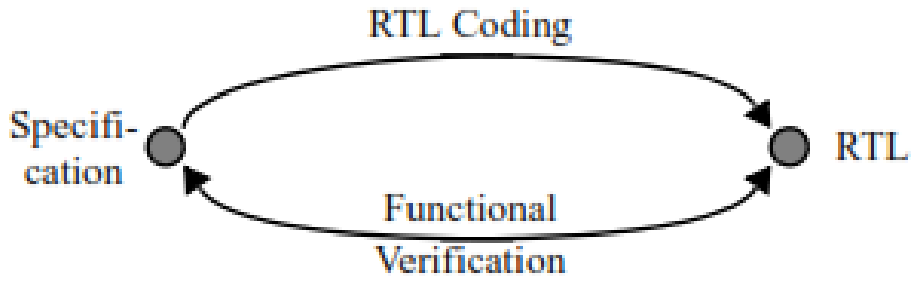



Figure 5.3: Functional Verification paths

It is the task of verifying that the logic design conforms to specification. In everyday terms, functional verification attempts to answer the question "Does this proposed design do what is intended?" This is a complex task, and takes the majority of time and effort in most large electronic system design projects. Functional verification is a part of more encompassing design verification, which, besides functional verification, considers non-functional aspects like timing and layout [8].

### 5.5.1 Functional Verification Approaches

#### 1. Black-Box Verification

Black-box verification cannot look at or know about the inside of a design (RTL specification).With a black-box approach, functional verification is performed without any knowledge of the actual implementation of a design. All verification is accomplished through the available interfaces, without direct access to the internal state of the design, without knowledge of its structure and implementation. This method suffers from an obvious lack of visibility and controllability. It is often difficult to set up an interesting state combination or to isolate some functionality. It is equally difficult to observe the response from the input and locate the source of the problem. This difficulty arises from the frequent long delays between the occurrence of a problem and the appearance of its symptom on the designs outputs.

**- Test Case is independent of implementation**

The advantage of black-box verification is that it does not depend on any specific implementation. Whether the design is implemented in a single ASIC, RTL code, transaction-level model, gates, multiple FPGAs, a circuit board or entirely in software, is irrelevant.
A black-box functional verification approach forms a true conformance verification that can be used to show that a particular design implements the intent of a specification, regardless of its implementation. A set of black-box testbenches can be developed on a transaction-level model of the design and run, unmodified, on the RTL model of the design to demonstrate that they are equivalent. Black-box testbenches can be used as a set of golden testbenches.

**- In black-box verification, it is difficult to control and observe specific features.**

The pure black-box approach is impractical in todays large designs. A multimillion gates ASIC possesses too many internal signals and states to effectively verify all of its functionality from its periphery. Critical functions, deep into the design, will be difficult to control and observe. Furthermore, a design fault may not readily present symptoms of a flaw at the outputs of the ASIC. For example, the black-box ASIC-level testbench in Figure is used to verify a critical round-robin arbiter. If the arbiter is not completely fair in its implementation, what symptoms would be visible at the outputs? This type of fault could only be found through performance analysis using several long simulations to identify discrepancies between the actual throughput of a channel compared with its theoretical throughput.

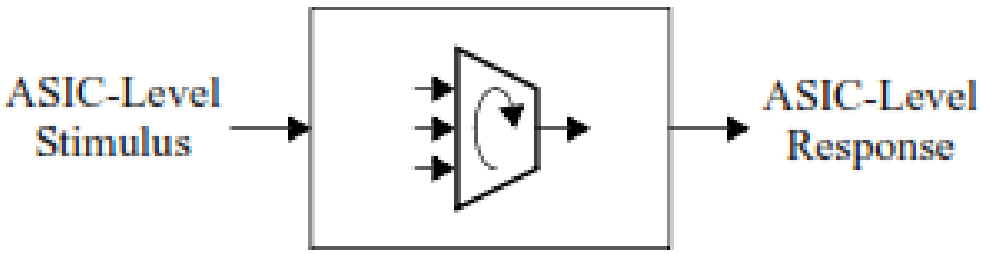



Figure 5.4: Black-box verification of a low-level feature

## 2. White-Box Verification

**- White box verification has intimate knowledge and control of the internals of a design.**

As the name suggests, a white-box approach has full visibility and controllability of the internal structure and implementation of the design being verified. This method has the advantage of being able to set up an interesting combination of states and inputs quickly, or to isolate a particular function. It can then easily observe the results as the verification progresses and immediately report any discrepancies from the expected behavior.

**- White-box verification is tied to a specific implementation.**

However, this approach is tightly integrated with a particular implementation. Changes in the design may require changes in the testbench. Furthermore, those testbenches cannot be used in gate-level simulations, on alternative implementations or future redesigns. It also requires detailed knowledge of the design implementation to know which significant conditions to create and which results to observe.

**- White-box techniques can augment black-box approaches.**

White-box verification is a useful complement to black-box verification. This approach can ensure that low-level implementation specific features behave properly, such as counters rolling over after reaching their end count value or datapaths being appropriately steered and sequenced. The white-box approach can be used only to verify the correctness of the functionality, while still relying on the black- or grey-box stimulus. Assertions are ideal for implementing white-box checks in RTL code.
For example, Figure 5.4 shows the black-box ASIC-level environment shown in Figure 5.5 augmented with assertions to verify the functional correctness of the round-robin arbiter. Should fairness not be implemented correctly, the white-box checks would immediately report a failure. The reported error would also make it easier to identify and confirm the cause of the problem, compared to a two percent throughput discrepancy.

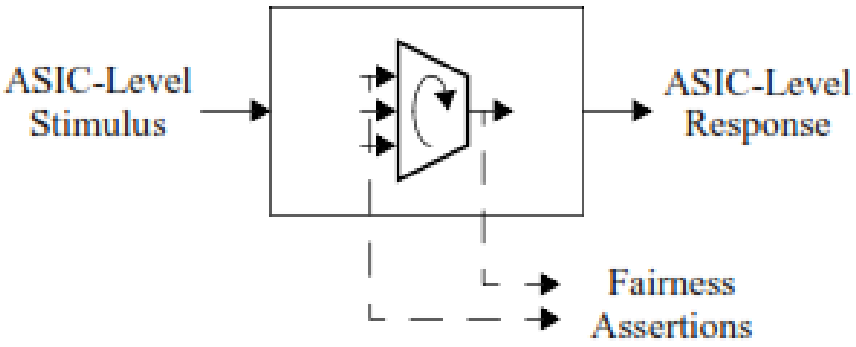



Figure 5.5: White-box checks in black-box environment

**- Checkered-box is used in system-level verification.**

A checkered-box verification approach is often used on SoC design and system-level verification. A system is defined as a design composed of independently designed and verified components. The objective of system-level verification is to verify the system-level features, not re-verify the individual components. Because of the large number of possible states and the difficulty in setting up interesting conditions, system-level verification is often accomplished by treating it as a collection of black-boxes. The independently designed components are treated as black-boxes, but the system itself is treated as a white-box, with full controllability and observability.

## 3. Grey-Box Verification

Grey-box verification is a compromise between the aloofness of a black-box verification and the dependence on the implementation of white-box verification. The former may not fully exercise all parts of a design, while the latter is not portable.

**- Test case may not be relevant on another implementation.**

As in black-box verification, a grey-box approach controls and observes a design entirely through its top-level interfaces. However, the particular verification being accomplished is intended to exercise significant features specific to the implementation. The same verification of a different implementation would be successful, but the verification may not be particularly more in-

teresting than any other black-box verification. A typical grey-box test case is one written to increase coverage metrics. The input stimulus is designed to execute specific lines of code or create a specific set of conditions in the design. Should the structure (but not the function) of the design change, this test case, while still correct, may no longer contribute toward better coverage.

**- Add functions to the design to increase controllability and observability .**

A typical grey-box strategy is to include some non-functional modifications to provide additional visibility and controllability. Examples include additional software-accessible registers to control or observe internal states, speed up a real-time counter, force the raising of exceptions or modify the size of the processed data to minimize verification time.
These registers and features would not be used during normal operations, but they are often valuable during the integration phase of the first prototype systems.

**- Verification must influence the design.**

For non-functional features required by the verification process to exist in a design, verification must be considered as an integral part of a design. When architecting a design, the verifiability of that architecture must be assessed at the same time. If some architectural features promise to be difficult to verify or exercise, additional observability or controllability features must be added. This process is called design-for-verification.

**- White-box cannot be used in parallel with design.**

The black-box and grey-box approaches are the only ones that can be used if the functional verification is to be implemented in parallel with the implementation using a transaction-level model of the design. Because there is no detailed implementation to know about beforehand, these two verification strategies are the only possible avenue.

## 5.6 Hardware Verification Languages (HVLs)

A hardware verification language, or HVL, is a programming language used to verify the designs of electronic circuits written in a hardware description language. HVLs typically include features of a high-level programming language like C++

or Java as well as features for easy bit-level manipulation similar to those found in HDLs. Many HVLs will provide constrained random stimulus generation, and functional coverage constructs to assist with complex hardware verification.

**- Most of Verification Engineers use the following languages for verifying complex digital designs , these languages are :**

1. OpenVera
2. e
3. System C
4. System Verilog

We will take a look on each of these languages in the this section.

1. ## OpenVera :

   it is a hardware verification language developed, and managed by Synopsys. OpenVera is an interoperable, open hardware verification language for testbench creation. The OpenVera language was used as the basis for the advanced verification features in the IEEE Std. 1800 SystemVerilog standard, for the benefit of the entire verification community including companies in the semiconductor, systems, IP and EDA industries along with verification services.

   **- Vendors supporting OpenVera**

   - Nusym Technology
   - Synopsys
   - Axiom Design Automation
   - Reference Verification Methodology (RVM)

2. ## e (Verification Language) :

   e is a hardware verification language (HVL) which is tailored to implementing highly flexible and reusable verification testbenches .
   e was first developed in 1992 in Israel by Yoav Hollander for his Specman software. In 1995 he founded a company, InSpec (later renamed Verisity), to commercialize the software. The product was introduced at the 1996 Design Automation Conference.Verisity has since been acquired by Cadence Design

Systems

**- Features of e-language :**

- Random and constrained random stimulus generation
- Functional coverage metric definition and collection
- Temporal language that can be used for writing assertions
- Aspect-oriented programming language with reflection capability
- Language is DUT-neutral in that you can use a single e testbench to verify a SystemC/C++ model, an RTL model, a gate level model, or even a DUT residing in a hardware acceleration box (using the UVM Acceleration for e Methodology)
- Can create highly reusable code, especially when the testbench is written following the Universal Verification Methodology (UVM)
- Formerly known as e Re-use Methodology (eRM)

3. **SystemC :**

SystemC is a set of C++ classes and macros which provide an event-driven simulation interface in C++ (see also discrete event simulation). These facilities enable a designer to simulate concurrent processes, each described using plain C++ syntax. SystemC processes can communicate in a simulated real-time environment, using signals of all the data types offered by C++, some additional ones offered by the SystemC library, as well as user defined. In certain respects, SystemC deliberately mimics the hardware description languages VHDL and Verilog, but is more aptly described as a system-level modeling language.

**- Language features**

(a) Modules Modules are the basic building blocks of a SystemC design hierarchy. A SystemC model usually consists of several modules which communicate via ports. The modules can be thought of as a building block of SystemC.

(b) Ports Ports allow communication from inside a module to the outside (usually to other modules) via channels.

(c) Exports Exports incorporate channels and allow communication from inside a module to the outside (usually to other modules).

(d) Processes Processes are the main computation elements. They are concurrent.

(e) Channels Channels are the communication elements of SystemC. They can be either simple wires or complex communication mechanisms like FIFOs or bus channels.

Elementary channels:

- signal: the equivalent of a wire
- buffer
- fifo
- mutex
- semaphore

(f) Interfaces Ports use interfaces to communicate with channels.

(g) Events Events allow synchronization between processes and must be defined during initialization.

(h) Data types SystemC introduces several data types which support the modeling of hardware.

Extended standard types:

- sc _ int < n > n-bit signed integer
- sc _ uint < n > n-bit unsigned integer
- sc _ bigint < n > n-bit signed integer for n>64
- sc _ biguint < n > n-bit unsigned integer for n>64

Logic types:

- sc _ bit 2-valued single bit
- sc _ logic 4-valued single bit
- sc _ bv<n> vector of length n of sc _ bit
- sc _ lv<n> vector of length n of sc _ logic

Fixed point types:

- sc _ fixed<> templated signed fixed point
- sc _ ufixed<> templated unsigned fixed point
- sc _ fix untemplated signed fixed point
- sc _ ufix untemplated unsigned fixed point

## 4. System Verilog :

In the semiconductor and electronic design industry, SystemVerilog is a combined hardware description language and hardware verification language based on extensions to Verilog.
SystemVerilog started with the donation of the Superlog language to Accellera in 2002. The bulk of the verification functionality is based on the OpenVera language donated by Synopsys. In 2005, SystemVerilog was adopted as IEEE Standard 1800-2005. In 2009, the standard was merged with the base Verilog (IEEE 1364-2005) standard, creating IEEE Standard 1800-2009. The current version is IEEE standard 1800-2012.

**- The feature-set of SystemVerilog can be divided into two distinct roles:**

(a) SystemVerilog for RTL design is an extension of Verilog-2005; all features of that language are available in SystemVerilog.

(b) SystemVerilog for verification uses extensive object-oriented programming techniques and is more closely related to Java than Verilog.

**- General improvements to classical Verilog :**

In addition to the new features above, SystemVerilog enhances the usability of Verilog's existing language features.

The following are some of these enhancements:

- The procedural assignment operator(s) (<=, =) can now operate directly on arrays.
- Port (inout, input, output) definitions are now expanded to support a wider variety of datatypes: struct, enum, real, and multi-dimensional types are supported.
- The for-loop construct now allows automatic variable declaration inside the for statement. And loop-control is improved by the continue and break statements.
- SystemVerilog adds a do/while to the while construct.
- Constant variables, i.e. those designated as non-changing during runtime, can be designated by use of const.
- Variable initialization can now operate on arrays.

- The preprocessor has improved 'define macro-substitution capabilities, specifically substitution within literal-strings (""), as well as concatenation of multiple macro-tokens into a single word.
- The fork/join construct has been expanded with join _ none and join _ any.
- Additions to the 'timescale directive allow simulation timescale to be controlled more predictably in a large simulation environment, with each source-file using a local timescale.
- Task ports can now be declared ref. A reference gives the task body direct access to the source arguments. in the caller's scope. Since it is operating on the original variable itself, rather than a copy of the argument's value, the task/function can modify variables (but not nets) in the caller's scope in realtime. The inout/output port-declarations pass variables by value, and defer updating the caller-scope variable until the moment the task exits.
- Functions can now be declared void, which means it returns no value.
- Parameters can be declared any type, including user-defined typedefs.

Besides this, SystemVerilog allows convenient interface to foreign languages (like C/C++), by SystemVerilog DPI (Direct Programming Interface).

## 5.7 Directed vs Random

As designs grow larger, it becomes more difficult to create a complete set of stimuli needed to check their functionality. You can write a directed test case to check a certain set of features, but you cannot write enough directed test cases when the number of features keeps doubling on each project. Worse yet, the interactions between all these features are the source for the most devious bugs and are the least likely to be caught by going through a laundry list of features. The solution is to create test cases automatically using constrained-random tests (CRT). A directed test finds the bugs you think are there, but a CRT finds bugs you never thought about, by using random stimulus. You restrict the test scenarios to those that are both valid and of interest by using constraints. Creating the environment for a CRT takes more work than creating one for directed tests. A simple directed test just applies stimulus, and then you manually check the result. These results are captured as a golden log file and compared with future simulations to see whether the test passes or fails. A CRT requires an environment to predict the result, using a reference model, transfer function, or other techniques, plus functional coverage to measure the effectiveness of the stimulus. However, once this environment is

in place, you can run hundreds of tests without having to hand-check the results, thereby improving your productivity. This trade off of test-authoring time (your work) for CPU time (machine work) is what makes CRT so valuable.
The first thing you may think of are the data fields. These are the easiest to create. The problem is that this approach has a very low payback in terms of bugs found: you only find data-path bugs, perhaps with bit-level mistakes. The test is still inherently directed. The challenging bugs are in the control logic. As a result, you need to randomize all decision points in your DUT. Everywhere control paths diverge, randomization increases the probability that youll take a different path in each test case

- **Device Configuration**

  What is the most common reason why bugs are missed during testing of the RTL design? Not enough different configurations have been tried! Most tests just use the design as it comes out of reset, or apply a fixed set of initialization vectors to put it into a known state. This is like testing a PCs operating system right after it has been installed, and without any applications; of course the performance is fine, and there are no crashes.

- **Environment Configuration**

  The device that you are designing operates in an environment containing other devices. When you are verifying the DUT, it is connected to a testbench that mimics this environment. You should randomize the entire environment, including the number of objects and how they are configured.

- **Primary Input Data**

  This is what you probably thought of first when you read about random stimulus: take a transaction such as a bus write or ATM cell and fi ll it with some random values. How hard can that be? Actually it is fairly straightforward as long as you carefully prepare your transaction classes. You should anticipate any layered protocols and error injection.

- **Encapsulated Input Data**

  Many devices process multiple layers of stimulus. For example, a device may create TCP traffic that is then encoded in the IP protocol, and fi nally sent out inside Ethernet packets. Each level has its own control fields that can be randomized to try new combinations. So you are randomizing the data and the layers that surround it. You need to write constraints that create valid control fields but that also allow injecting errors.

- **Protocols, Exceptions and Violations**

  Anything that can go wrong, will, eventually. The most challenging part of design and verification is how to handle errors in the system. You need to anticipate all the cases where things can go wrong, inject them into the system, and make sure the design handles them gracefully, without locking up or going into an illegal state. A good verification engineer tests the behavior of the design to the edge of the functional specification and sometimes even beyond. When two devices communicate, what happens if the transfer stops partway through? Can your testbench simulate these breaks? If there are error detection and correction fields, you must make sure all combinations are tried. The random component of these errors is that your testbench should be able to send functionally correct stimuli and then, with the flip of a configuration bit, start injecting random types of errors at random intervals.

- **Delays**

  Many communication protocols specify ranges of delays. The bus grant comes one to three cycles after request. Data from the memory is valid in the fourth to tenth bus cycle. However, many directed tests, optimized for the fastest simulation, use the shortest latency, except for that one test that only tries various delays. Your testbench should always use random, legal delays during every test to try to find that (hopefully) one combination that exposes a design bug. Below the cycle level, some designs are sensitive to clock jitter. By sliding the clock edges back and forth by small amounts, you can make sure your design is not overly sensitive to small changes in the clock cycle. The clock generator should be in a module outside the testbench so that it creates events in the Active region along with other design events. However, the generator should have parameters such as frequency and offset that can be set by the testbench during the configuration phase.

## 5.8 Universal Verification Methodology (UVM)

This phase is considered to be independent to main Verification Progress and has no effect on the current processes. The main target of UVM phase is to proceed preamble step in the verification process through the next version of Andalus DSP.

### 5.8.1 Design Specifications

The TMS320C2x central arithmetic logic unit (CALU) contains

- 16-bit scaling shifter
  - has a 16-bit input connected to the data bus
  - has a 32-bit output connected to the ALU.
  - The scaling shifter produces a left shift of 0 to 16 bits on the input data, as programmed in the instruction.
  - The LSBs of the output are filled with zeros, and the MSBs may be either filled with zeros or sign-extended
- 16 16-bit parallel multiplier
  - compute a signed or unsigned 32-bit product in a single machine cycle.
  - A 16-bit temporary register (TR) that holds one of the operands for the multiplier.
  - A 32-bit product register (PR) that holds the product.
  - The output of the product register can be left-shifted 1, 4 or 6 bits.
  - TR is loaded via Load T-Register instructions specified in Instruction set
- 32-bit arithmetic logic unit (ALU)
  - The major of operations execute in a single clock cycle
  - Perform Arithmetic and Boolean Operation
  - One input to the ALU is always provided from the accumulator, and the other input may be provided from the product register (PR) of the multiplier or the input scaling shifter that has fetched data from the RAM on the data bus.
- 32-bit accumulator (ACC)
  - The 32-bit accumulator is split into two 16-bit segments for storage in data memory: ACCH (accumulator high) and ACCL (accumulator low).
  - Shifters at the output of the accumulator provide a left-shift of 0 to 7 places
  - This shift is performed while the data is being transferred to the data bus for storage.

- The contents of the accumulator remain unchanged. When the ACCH data is shifted left, the LSBs are transferred from the ACCL, and the MSBs are lost. When ACCL is shifted left, the LSBs are zero-filled, and the MSBs are lost.
- additional shifters at the outputs of both the accumulator and the multiplier.

### 5.8.2 UVM Environment

UVM Environment mainly aims to generate constrained random stimulus, push them towards the Device Under Test, check the DUT behavior correctness and finally check the overall performance of The verification process.
The Goals mentioned above can be achieved through many realizations of simulation environment and we are going to discuss one of them through the rest of this section.

#### Sequence

is considered to be a sequence of stimulus formed to be introduced to the DUT. Actually sequence items form aren't the actual inputs of the DUT, instead we encapsulate them into abstract form called Transaction, usage of transaction instead the actual inputs considered to be more efficient than traditional stimulus representation and will be explained later.

#### Driver

is the unit which responsible of transform the stimulus from Transaction domain into Signal-level domain to introduce it to the DUT. Actually driver task is to represent the physical interaction with the DUT and requires full-understand of the applied test scenarios.

#### Monitor

is the unit which receive the DUT response to different stimulus generated by sequence, check the response validity (Parity, CRC) and check the overall performance of the verification progress through the coverage statistics

#### Scoreboard

is the unit which receive the stimulus generated by sequence, calculate the expected output, compare it with the actual output of DUT and record all defected test cases so as to be reported. The theoretical response obtained by apply the stimulus sent

by sequence to behavioral model of DUT called Reference Model. This model is designed according the design specification forwarded from system level designer or customer.

### TLM

Transaction Level Modeling is considered to be highly efficient way to obtain good stimulus generation. This concept can be explained clearly beyond our DUT. Transaction in this environment is abstract operations of CALU like addition, subtraction, multiplication . etc.
Our DUT have 67 pins of input signals and to traditionally verify its behavior, we need to test all possible inputs (i.e 2 $\hat{6}7$) possible inputs, but on other hand, one can notice that not all inputs should affect all DUT operations. For instance, overflow and carry inputs affect the addition and subtraction operations but never affect the logical operations like XORing, XNORing and ANDing. Consequently there is no meaning to test these bits through the logical operations' test.

### CRT

Through the last section we realize the great role of transaction representation of stimulus and here we raise the importance of constrained random test which follow the plan made by the verification team and discussed in previous sections.

### Sequence

the sequence planed to apply for the DUT aim to divide the test into three layers of tests named weak, normal and strength. Applying these layers will be done in the order they explained. Actually these strategy is preferred to avoid wasting of time, since the design with trivial defects will cause a lot of defected test cases in high strength layer.

### 5.8.3 weak layer

| R.N | Description |
|---|---|
| 001 | Addition |
| 002 | Subtraction |
| 003 | Multiplication |
| 004 | Shift Left |
| 005 | Shift Right |
| 006 | Rotate Left |
| 007 | Rotate Right |
| 008 | AND |
| 009 | OR |
| 010 | XOR |
| 011 | CMPL |
| 012 | NEG |
| 013 | Store ACC |
| 014 | Load ACC |
| 015 | Store P |
| 016 | Load T |

### 5.8.4 Normal layer

| | |
|---|---|
| 100 | Addition with 0 ~ 7 shift with sign extension mode enabled and sx =1 |
| 101 | Addition with 8 ~15 shift with sign extension mode enabled and sx=0 |
| 102 | Addition with 16 shift with sign extension mode suppressed |
| 103 | Addition with 10 shift with sign extension mode enable and sx = 1 and high = 1 |
| 104 | Addition to high Accumulator byte |
| 105 | Load T Register with < 16 and Add to Accumulator with shift specified by T Register |
| 106 | Load T Register with > 16 and Add to Accumulator with shift specified by T Register |
| 107 | Addition long Value ( Double Word Instruction ) |
| 108 | Addition with Carry |
| 200 | Subtraction with 0 ~ 7 shift with sign extension mode enabled and sx =1 |
| 201 | Subtraction with 8 ~15 shift with sign extension mode enabled and sx=0 |
| 202 | Subtraction with 16 shift with sign extension mode suppressed |
| 203 | Subtraction with 0 ~ 7 shift with sign extension mode suppressed |
| 204 | Subtraction with 10 shift with sign extension mode enable and sx = 1 and high = 1 |
| 205 | Subtraction to high Accumulator byte |
| 206 | Load T Register with < 16 and Subtract to Accumulator with shift specified by T Register |
| 207 | Load T Register with > 16 and Subtract to Accumulator with shift specified by T Register |
| 208 | Subtraction long Value ( Double Word Instruction ) |
| 209 | Basic Subtraction to result in Borrow flag set |
| 210 | Subtraction with Borrow |
| 400 | Shift left with sign extension mode enabled and sx = 0 |
| 401 | Shift left with sign extension mode enabled and sx = 1 |

| | |
|---|---|
| 402 | Shift left with sign extension mode disabled and sx =0 |
| 403 | Shift left with sign extension mode disabled and sx = 1 |
| 500 | Shift right with sign extension mode enabled and sx = 0 |
| 501 | Shift right with sign extension mode enabled and sx = 1 |
| 502 | Shift right with sign extension mode disabled and sx =0 |
| 503 | Shift right with sign extension mode disabled and sx = 1 |
| 600 | Rotate left with sign extension mode enabled and sx = 0 |
| 601 | Rotate left with sign extension mode enabled and sx = 1 |
| 602 | Rotate left with sign extension mode disabled and sx =0 |
| 603 | Rotate left with sign extension mode disabled and sx = 1 |
| 700 | Rotate right with sign extension mode enabled and sx = 0 |
| 701 | Rotate right with sign extension mode enabled and sx = 1 |
| 702 | Rotate right with sign extension mode disabled and sx =0 |
| 703 | Rotate right with sign extension mode disabled and sx = 1 |
| 800 | And ACC with data 1 cycle |
| 801 | And ACC with data 2 cycle |
| 900 | OR ACC with data 1 cycle |
| 901 | OR ACC with data 2 cycle |
| 1000 | XOR ACC with data 1 cycle |
| 1001 | XOR ACC with data 2 cycle |

### 5.8.5 strength layer

Through this one, we target the DUT to be stressed under unpredictable sequence of operations regardless of operations dependency. And it could be formed with randomize the instructions of the normal layer so that we could be able to pick only one random instruction from each category and followed it by random operation extracted from another category.

### 5.8.6 Driver

Our driver designed to contact with the DUT on the level of signals' pins, Driver ask its sequencer to pass him the handle of next transaction, then translate it to signals level to begin to drive the DUT with the proper inputs.

### 5.8.7 Monitor

Monitor is designed to receive DUT signal-level response to different transactions and repack it into transaction form to facilitate coverage and checker analysis.

### 5.8.8 Scoreboard

Scoreboard is created as checker component that mainly aims to compare results and store defected test case in associative array. Scoreboard also include reference model which determine the expected results according design specifications described above.

### 5.8.9 Reference Models

Reference model has been chosen at the field of checkers over other possibilities due to large combination of inputs could be tested which make it the best choice and designed in abstract level to receive transaction from sequencer and proceed its theoretical output to Scoreboard. Its hierarchy consists of control function which receive the transaction and call another sub-control function which in turn call another computational function that predict the expected output.

## 5.9 verification plan

Verification plan in this project had been modified too many times to satisfy needs and to find design defects. This plan consists of phases, every phase has its mission.

- First fire.

- Block function Verification.
- Block access repetition.
- Integration phase.

### 5.9.1 1st phase: First fire: (simple instructions)

In this phase, our goal is to fire the DSP core and to make sure this design need fixing not redesigning. This phase is the smallest phase with respect to time. Its top aim is to validate that clock drive the core, works PC (program counter) counts without errors, Fetching and Decoding from memory fine without any confliction.

### 5.9.2 2nd phase: Block function Verification:

This phase aims to test the functionality of each block and treat it as black box. With the assembler, small codes are written to access whole core blocks deeply. This phase is not only testing blocks validation but also test flags.

**Ex1:**

LACK 30h
ADDLK 20h

In this example, the code written to access the core adder and check its functionality

**Ex2:**

LACK 30h
LT 20h
MPYK 60h

In this example, this code accesses the Multiplier block. Note that , in this phase our 1st goal to test functionality of blocks only, observe that immediate addressing mode to avoid any bugs (if existing) comes from memory as possible as we can.

**Ex3:**

LAC 30h
SACH 40h

SACL 41h

Here this code aims to access data memory and check reading and writing operations with direct address mode and so on to all block.

This phase consumes too much times because this phase time needs RTL edit and to fix all bugs in the blocks design, otherwise testing all Instruction set.

### 5.9.3 3rd phase: Block access repetition:

This phase is not testing blocks functionality, but the main task is to repeat accessing same block to check Fetching, Decoding, executing.

**Ex1:**

ADDK 20h
ADLK 0FECAh
ADDC *+
SUBK 30h
SBLK 15h
SUB *+
SUBB *+

Note that, this code is to repeat accessing on Adder/Subtractor in ALU unit.

**Ex2:**

OR *+
ORK 30h
ANDK 4Ch
AND *+
XOR *+

Here this simple code is to access the Logic operations in ALU unit multiple times

**Ex3**

LALK 0C165h
SFR
SFL

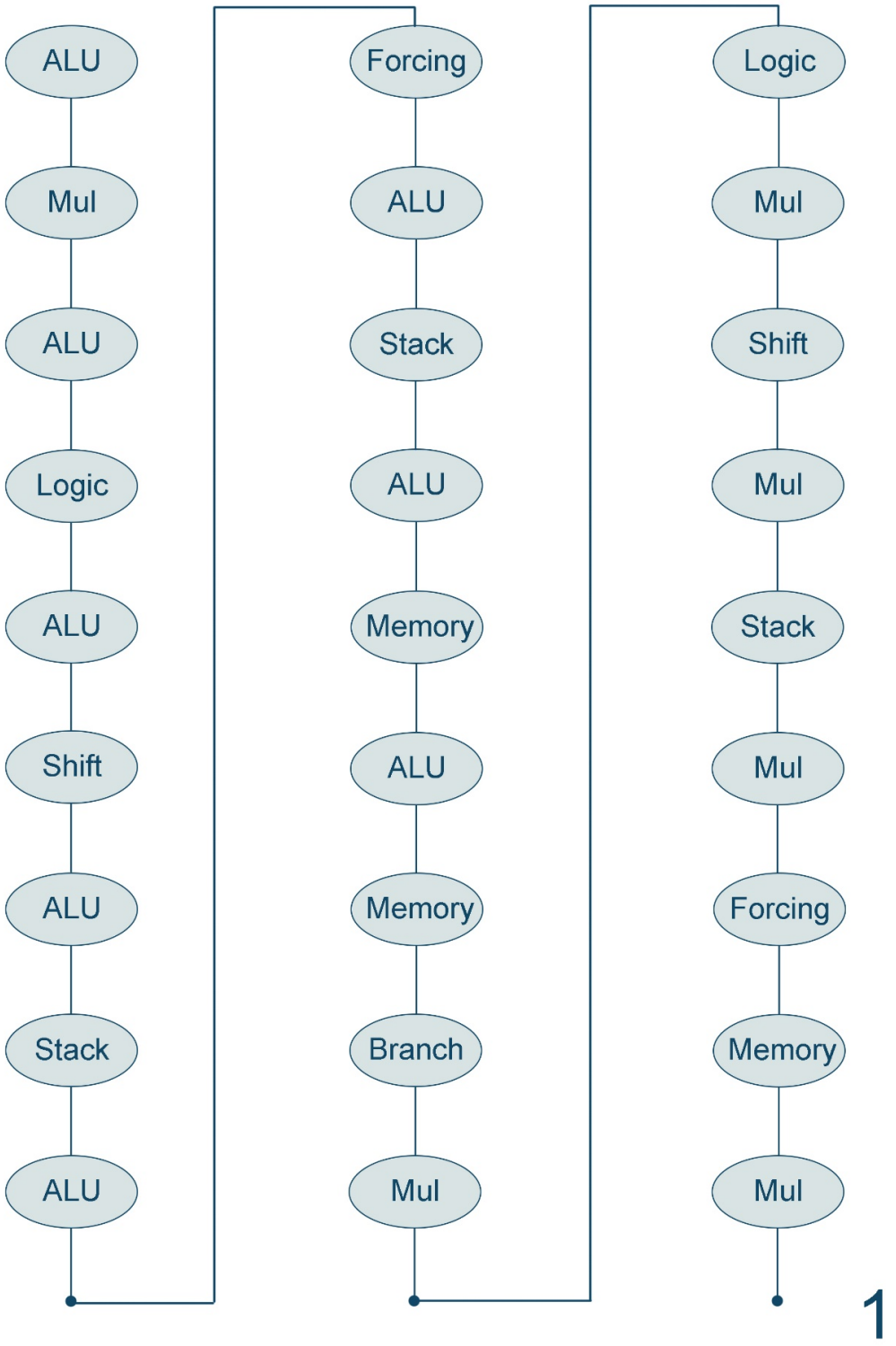
ROL
ROR

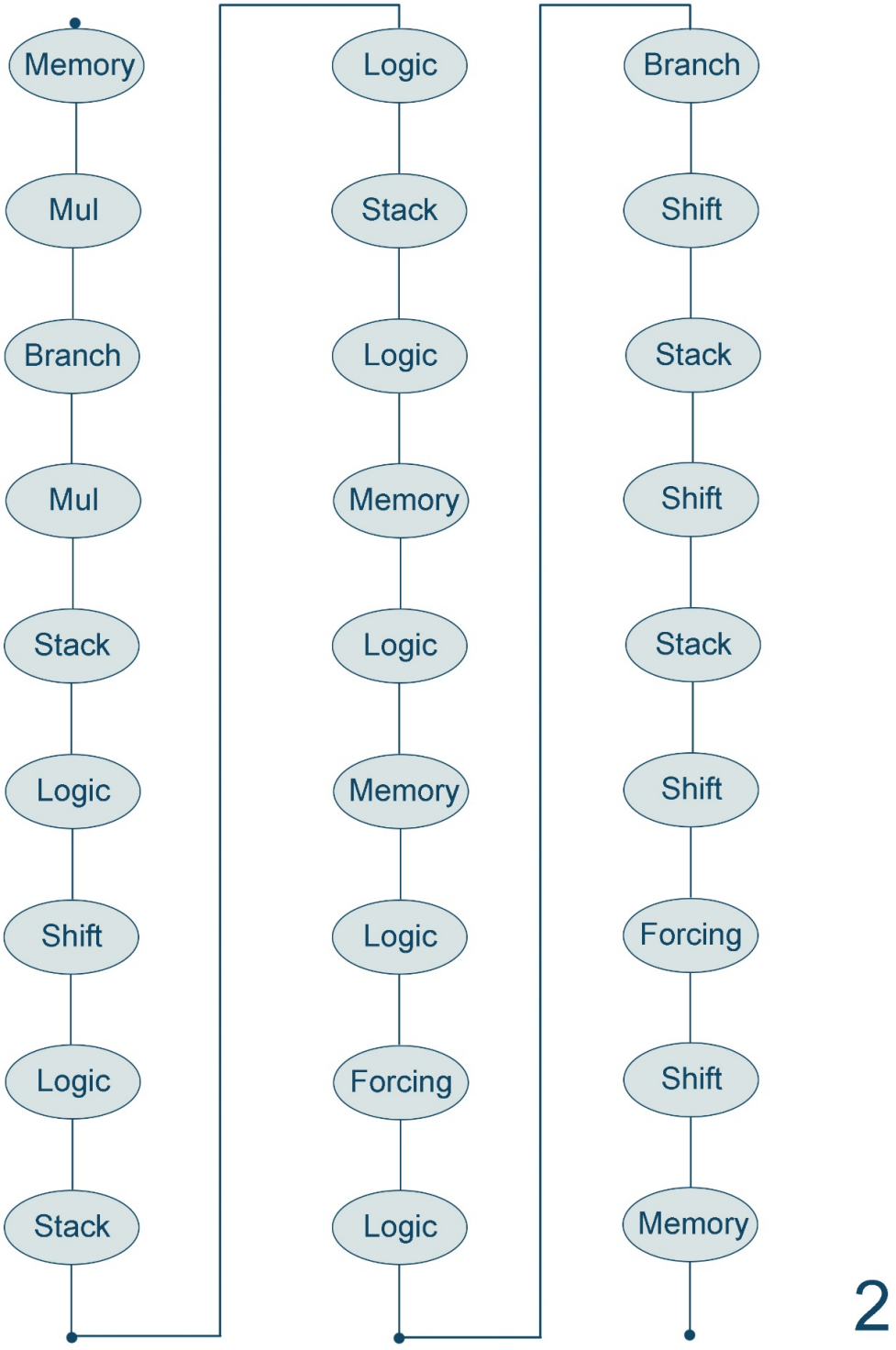
Note that, this simple code aims to access shifter unit in ALU.

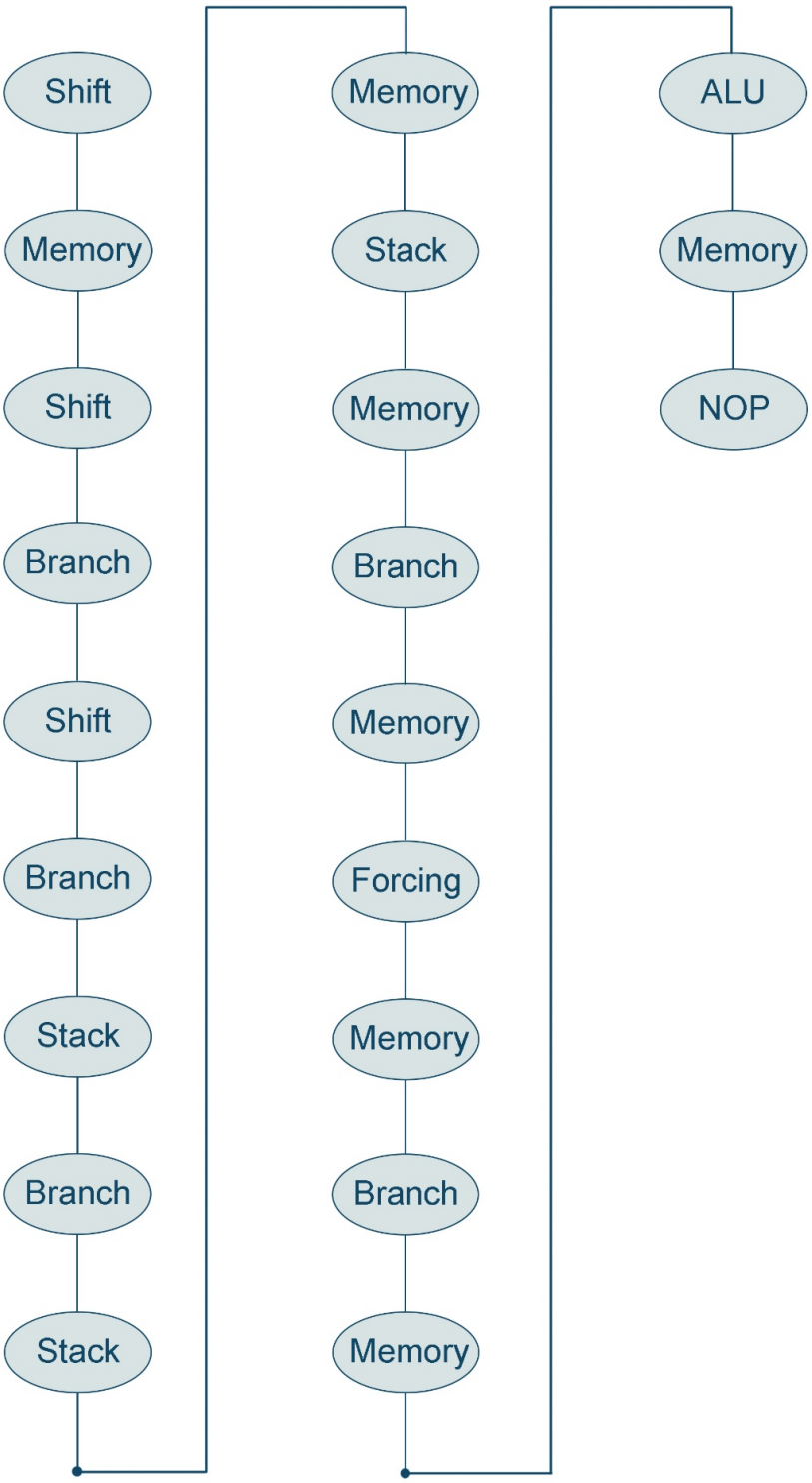
### 5.9.4 4th phase: Integration phase:

This phase is the longest phase in functional Verification,
This phase mission is to test core with respect to all previous types of test, like, Memory Operations:
Data Memory Reading and Writing, Storing, Branching, Block functionality, Program Memory, Control unit:
Fetching, Decoding, block functionality: Adder, Shifter, Register files, Addressing modes, Stack: PUSH, POP, Etc.
Figure blow shows flow diagram of Verification Integration phase:

- ALU: ALU operations (Addition, Subtraction)
- Logic: Logic Operations.
- Shift: Shifting Operations.
- Stack: PUSH and POP operations.
- Mul: Multiplication operations.
- Memory: Read & Write operations on Data memory.
- Branch: Branching and Jumping Conditional and non-Conditional.
- Forcing: Zeroing Accumulator.


ALU
Mul
ALU
Logic
ALU
Shift
ALU
Stack
ALU
Forcing
ALU
Stack
ALU
Memory
ALU
Memory
Branch
Mul
Logic
Mul
Shift
Mul
Stack
Mul
Forcing
Memory
Mul
1


Memory
Mul
Branch
Mul
Stack
Logic
Shift
Logic
Stack
Logic
Memory
Logic
Memory
Logic
Forcing
Logic
Branch
Shift
Stack
Shift
Stack
Shift
Forcing
Shift
Memory
2


Shift
Memory
Shift
Branch
Shift
Branch
Stack
Branch
Stack
Memory
Stack
Memory
Branch
Memory
Forcing
Memory
Branch
Memory
ALU
Memory
NOP
3

Each blue bubble in this figure every time runs with different Assembly code for example, Writing code to represent first bubble in figure(1), Bubble is ALU, here we run the code with ALU representation as Instruction like, ADDK 20h , the next time we run the code with replacing ADDK 20h with ADD 30h here we tested 2 kinds of addressing mode with the same bubble to cover almost all Instructions Sequence can be written in the future by programmers or Users.
Integration phase aims to penetrate all micro-element in the design and to test the full validity of the DSP core.
Before Eliminating Interrupt and Serial Units due to problems occur with Backend and Design team, the Integration phase comprise this units and has its independent tests and codes, and these blocks had been tested and fixed from bugs ,and It was working fine as RTL design.
An example achieves the Integration phase, (Matches figures):

```
LDPK 0
LARP AR7
LARK AR7, 60h
ADDK 53h
LT *+
MPYK 0CFh
ZAC
APAC
ANDK 00FFh
ROR
ADDC *+
SFL
SUBK 0Fh
PUSH
ADDK 20h
ZAC
ADD 78h
POP
ADLK 6CEDh
SACH *+
SACL *+
SUBK 01h
LALK 6543h
B frst
* Unreachable statements
SFL
```

```
LACK 5h
frst NOP * bubble insertion (eliminate branching hazards)
LT *+
MPYK 0C1h
ZAC
APAC
XORK 0F0F0h
ROL
PUSH
ZAC
MPYK 11h
APAC
BNZ scnd
MPYK 21h
scnd NOP
POP
CMPL
SFL
OR *+
PUSH
AND *+
POP
NEG
LAC *+
XOR *+
SACL *+
SACH *+
PUSH
AND *+
POP
CMPL
ZAC
BZ Branch1
* Unreachable statements
PUSH
SACL *+, 5
NEG
Branch1 NOP
LAC *+
ROL
```

PUSH
SFL
POP
SFR
ZALH 63h
ROL
SACL *+
SACH *+
SFL
LALK 0A55Ah
SFR
BNC Branch2
* Unreachable statements
ZAC
SACL *+
SACH *+
Branch2 NOP
ROR
BNC Branch3
PUSH
Branch3 NOP
BGEZ Branch4
* Unreachable statements
NEG
ZAC
Branch4 NOP
POP
SACL *-
SACH *-
PUSH
LAC *+
BLZ Branch5
* Unreachable statements
LT 50h
MPYK 47h
ZAC
APAC
SACL *+
SACH *+
Branch5 NOP

SACL *+
SACH *+
ZAC
B Branch6
*Un reachable statements begin
LT 50h
MPYK 47h
APAC
LALK 0ffffh
SACL *+
SACH *-
*Un reachable statements end
Branch6 NOP
LAC *+
ADLK 0FA02h
NOP
SACL *+
SACH *+
NOP

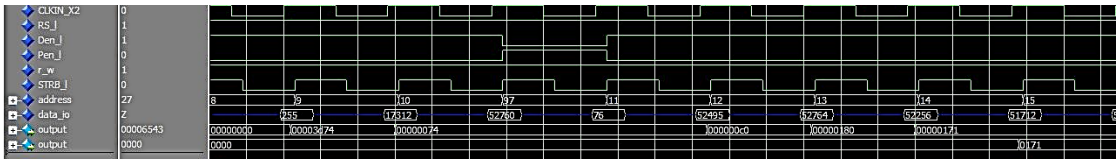



Figure 5.6: Part of simulation of Post layout

**Bugs discovered: - without Hazards (next section)-**

- Combinational loops. Two signals were looping on each other in an infinite loop.

- Bugs in some Instructions. Some instructions have a problems in Decoding, as

  CMPL Fixed.

  OR Fixed.

  PAC Not fixed.

  SBLK Not fixed.

- Interfacing with External memory. Interfacing with External memory with its model had has some challenges like timing in Writing cycles and reading cycles.

### 5.9.5 Hazards:

- Data hazards: Data hazards occur when instructions that exhibit data dependence modify data in different stages of a pipeline. Ignoring potential data hazards can result in race conditions. There are three situations in which a data hazard can occur:

  1. Read after write (RAW).
  2. Write after read (WAR).
  3. Write after write (WAW).

  **1. Read after write (RAW):**

  In these hazards, the read process happens after the write process, although both processes happen in the same clock cycle. If the write process takes a long time, it may not complete by the time the read occurs, which will produce incorrect data.

  **Ex:**

  SACL *+ #save accumulator to memory
  LAC *+ #load accumulator from memory

  **2. Write after read (WAR):**

  In a WAR hazard, the write from a previous instruction will not complete before the successive read instruction. This means that the next value read will be a previous value, not the correct current value.

  **Ex:**

  LT *+ #load T register
  SPL*+ #store low P register to memory

3. **Write after write (WAW):**

   WAW hazards occur when two processes try to write to a data storage element at the same time. If this occurs in a single clock cycle, there will be no time in between to read the intermediate value. If the instructions execute out of order, the incorrect value may be left in the register and the current written value may be a previous written value.

   **Ex:**

   SACL *+
   SACH *+

- **Control Hazards (Branching Hazards):**

  Control hazards occur when a branch instruction is processed. While the branch instruction is traveling through the pipeline, the instruction fetch module will continue to read sequential instructions from the instruction memory. The problem is that because of the branch, the next instructions might execute out of order, which will cause problems. DSP doesn't know which instructions to execute after the branch until he knows whether the branch was taken or not.

  **Ex:**

  B Branch1
  * Unreachable statements
  Branch1 LAC *+

**Hazards Eliminating**

**Data hazards:**

In our Design data hazards solved in 3 stages:

- First treatment is inserting bubble instruction (NOP No Operation)

**Ex:**

SACL *+

NOP
SACH *+

-Second choice is to insert PLL (Phase-locked loop), that control the memory perfectly and tuning timing.


**Control Hazards (Branching Hazards):**

Eliminating control hazards done by inserting (NOP) after branching label, this solution called (Software solution) or (programmer solution) Due to programmer involvement.

**Ex:**

B Branch1
* Unreachable statements
Branch1 NOP
LAC *+

# References


[1] Maria Elena Angoletta. Digital signal processor fundamentals and system design. 2008.

[2] Khosrow Golshan. *Physical design essentials: An ASIC design implementation perspective*. Springer, 2007.

[3] Steven W Smith. *Digital signal processing: a practical guide for engineers and scientists*. Newnes, 2003.

[4] Texas Instruments. Tms320c25 users guide. *Digital Signal Processor Products, preliminary*, 1986.

[5] Andalus  cordoba final report.

[6] Cy7c1041cv33 data sheet.

[7] www.ti.com.

[8] Chris Spear. Systemverilog for verification-a guide to learning the testbench language features. sl, 2008.

[9] Bruce Wile, John C Goss, and Wolfgang Roesner. *Comprehensive Functional Verification: The Complete Industry Cycle*. Morgan Kaufmann, 2005.