



Cairo University



Faculty of Engineering

# Hardware Accelerator for Deep Learning Inference

---

Prepared by

**Ahmed Mahmoud Huseiny Atya**

**Islam Nasser Mahmoud**

**Hassan Ahmed Hassan El-Sman**

**Abdelnaeem Abdelbaset Abdelnaeem**

**Khaled Khalifa Abd El-Hay**

**Ahmed Alaa El-deen Fathy Mahmoud**

Supervised by

**Dr. Hassan Mostafa**

**Dr. Hassan Aboushady**

A Graduation Project Thesis Submitted to  
The Faculty of Engineering at Cairo University. In Partial Fulfillment of  
The Requirements for The

**Degree of  
Bachelor of Science**

In

Electronics and Communications Engineering  
Faculty of Engineering, Cairo University

Giza, Egypt

July 2023

## Acknowledgments

We would like to express our heartfelt gratitude to Dr. Hassan Mostafa and Dr. Hassan Aboushady for their invaluable guidance and support throughout the journey of our graduation project. Their unwavering commitment to excellence, depth of knowledge, and unwavering dedication have been instrumental in shaping the success of our project.

Dr. Hassan Mostafa, your mentorship and expertise have been pivotal in shaping our research methodology and ensuring the accuracy and precision of our findings. Your guidance and insightful feedback have continually challenged us to think critically and strive for excellence. We are truly grateful for your unwavering support and belief in our capabilities.

Dr. Hassan Aboushady, we are immensely grateful for your constant encouragement and motivation. Your vast knowledge and passion for the subject matter have served as an inspiration to us. Your ability to simplify complex concepts and provide us with a clear direction has been invaluable to the development of our project.

Lastly, we would like to express our heartfelt appreciation to our families and friends for their unconditional love, understanding, and encouragement throughout this challenging yet rewarding journey. Their unwavering support has been a constant source of strength and motivation.

To everyone who has contributed to our graduation project, directly or indirectly, we extend our sincerest thanks. Your support and belief in our abilities have played a significant role in shaping the success of this project.

## Abstract

The increasing demand for high-performance deep-learning models has led to the development of specialized hardware accelerators. Among these accelerators, Gemmini stands out as a configurable and efficient solution for deploying deep learning models. This graduation project aims to explore the process of configuring and implementing the Gemmini accelerator for the deployment of deep learning models.

The project begins with an in-depth study of Gemmini's architecture, focusing on its design principles and key components. This understanding forms the basis for configuring the accelerator to support various types of deep learning models. By adapting the architecture to accommodate specific model requirements, the Gemmini accelerator can maximize performance and efficiency.

This graduation project focuses on estimating the area and power results of the Gemmini accelerator using a digital backend synthesis, Formal Verification and Place-and-Route (PNR) flow. The flow involves logic synthesis, technology mapping, placement, routing, physical verification, and area and power estimation. Through this flow, designers can evaluate the efficiency and scalability of the accelerator, enabling informed decisions during the deployment of the Gemmini accelerator in deep learning model implementations. The project aims to provide valuable insights into the area and power characteristics of the Gemmini accelerator, facilitating its integration into various hardware setups for deep learning applications.

# Table of Contents

Acknowledgments.....	2
Abstract.....	3
1. Introduction.....	1
1.1 Cognitive radio networks.....	1
1.2 Modulation classification.....	3
1.3 Wireless signal model.....	3
1.4 Fading.....	5
1.4.1 large-scale fading.....	5
1.4.2 Small-scale fading.....	6
1.5 Data acquisition.....	8
1.6 Deep learning.....	8
1.7 Deep learning acceleration.....	11
2 Gemmini and Chipyard Framework.....	15
2.1 Why Gemmini: The Choice Among Other Tools:.....	15
2.2 Chipyard Ecosystem:.....	16
2.2.1 Chisel:.....	16
2.2.2 FIRRTL:.....	16
2.2.3 RISC-V:.....	17
2.2.4 Rocket core and BOOM core:.....	17
2.2.5 RoCC (Rocket Chip Coprocessor) Accelerator generators:.....	17
2.2.6 FireSim:.....	18
2.2.7 HAMMER:.....	18
3 Methodology and Design Flow (Frontend):.....	19
3.1 Starting with DL Model:.....	19
3.2 Gemmini accelerator Design Flow for designing accelerators:.....	20
3.3 Custom SoC configuration for Different RTL generators:.....	20
3.4 RTL Build Process:.....	21
3.5 Software RTL Simulations:.....	22
3.5.1 Low-Level tests:.....	23
3.5.2 Gemmini Flow Automation Script:.....	23
3.5.3 High-level test:.....	24
4 Gemmini Framework.....	26
4.1 Background.....	26
4.1.1 Converting Convolution to General Matrix Multiplication (GEMM).....	26

4.1.2 Gemmini framework.....	28
4.2 Gemmini architecture.....	28
4.2.1 Introduction .....	28
4.2.2 RoCC interface .....	29
4.2.3 LoopConv and LoopMatMul modules .....	30
4.2.4 LoadController and StoreController.....	30
4.2.5 ExecuteController and image to column (im2col) .....	31
4.2.6 ScratchPad .....	33
4.3 Configurations.....	34
4.3.1 Introduction .....	34
4.3.2 Configurable parameters.....	34
4.3.3 Implemented configurations .....	34
5 Design flow .....	36
5.1 Synthesis flow using design compiler:.....	37
5.1.1 Develop HDL files.....	37
5.1.2 Specify libraries (PDKS).....	38
5.1.3 Read Design.....	39
5.1.4 Define design environment.....	40
5.1.5 Set design constrains .....	41
5.1.6 Compile strategy.....	43
5.1.7 Synthesize and optimize the design.....	43
5.1.8 Reports and netlist .....	44
5.2 Results of the synthesis step .....	45
5.2.1 Timing -setup.....	45
5.2.2 Timing -hold.....	46
5.2.3 Area .....	47
6 Formal Verification.....	48
6.1 Formal Verification Components.....	49
6.1.1 Logic Cones.....	49
6.1.2 Compare Points.....	50
6.1.3 Containers.....	51
6.2 Formality Flow.....	52
6.2.1 Start Formality .....	53
6.2.2 Load Guidance.....	53
6.2.3 Load Reference and Implementation Designs.....	53

6.2.4 Perform Setup .....	54
6.2.5 Matching Compare Points .....	54
6.2.6 Performing Verification .....	56
6.2.7 Reporting and Interpreting Results .....	56
6.3 Result of the Formal Verification.....	57
7 CHAPTER FIVE: PLACE AND ROUTE.....	58
7.1 Introduction to PnR .....	58
7.2 PnR Flow.....	59
7.3 Detailed PnR Flow for Queue_56 .....	60
7.3.1 Design Import.....	60
7.3.2 Floor Planning .....	61
7.3.3 Power Planning.....	62
7.3.4 Placement.....	62
7.3.5 Timing analysis and optimization preCTS .....	63
7.3.6 Clock Tree Synthesis (CTS).....	64
7.3.7 Detailed Routing.....	64
7.3.8 Verification.....	65
7.3.9 Filler Insertion .....	66
7.3.10 Summary.....	67
7.4 Results .....	67
7.4.1 Area Results.....	67
7.4.2 Area comparison.....	68
7.4.3 Power Results .....	68
8 Conclusion .....	70
9 Future work.....	71
References.....	73

# List of Figures

Figure 1: Spectrum management framework [2].....	1
Figure 2: Transmitter diagram .....	3
Figure 3: Receiver diagram.....	4
Figure 4: transmission gain with respect to logarithm of the distance. ....	5
Figure 5: Multipath propagation illustration. [4].....	6
Figure 6: Response of non-stationary channel. [4].....	7
Figure 7: Neural network example. ....	9
Figure 8: Perceptron.....	9
Figure 9: Common activation functions. ....	10
Figure 10: a fully connected layer before and after pruning.....	10
Figure 11: convolution example. ....	11
Figure 12: CNN example.....	11
Figure 13: max pooling example .....	12
Figure 14: Processing engine [6] .....	12
Figure 15: time sharing in deep learning acceleration. [6].....	13
Figure 16: weight stationary. [6] .....	13
Figure 17: output stationary. [6].....	14
Figure 18 Chipyard Ecosystem.....	16
Figure 19 Process of converting Chisel to Verilog [8] .....	17
Figure 20 Rocket SoC for accelerators .....	18
Figure 21 Approaches of running DL models. ....	19
Figure 22 Gemmini Accelerator Different design flows. ....	20
Figure 23 Snapshots of chisel configuration files.....	21
Figure 24 Snapshot of output generated files from Gemmini.....	22
Figure 25 Gemmini different levels of visibility. ....	22
Figure 26 Output of the final configuration Low level tests.....	24
Figure 27 (a) Direct convolution. (b) im2col-based GEMM operation (C)Systolic array operation .....	26
Figure 28 MAC architecture .....	27
Figure 29 Full SoC architecture.....	28
Figure 30 Gemmini accelerator architecture.....	29
Figure 31 RoCC interface signals.....	29
Figure 32 Command unrolling modules .....	30
Figure 33 DMA engine .....	31
Figure 34 SRAM controllers (a) im2col module (b)ExecuteController .....	31
Figure 35 MeshWithDelays module .....	32
Figure 36 MeshWithDelays module .....	32
Figure 37 Scratchpad module .....	33
Figure 38 (a) ScratchpadBanks (b) AccumulatorMem.....	33
Figure 39 Overview of how Design Compiler fits into the design flow.....	36
Figure 40 (computer-aided design) CAD tools.....	37
Figure 41 the Synthesis flow using DC tool.....	38
Figure 42 Design compiler take VHDL and Verilog files only.....	38
Figure 43 TSMC PDK and its PVT flavours .....	38
Figure 44 Structure of design.....	39
Figure 45 Design environment to estimate the reality .....	40

Figure 46 setting the design environment components.....	40
Figure 47 Different processes in the different operating conditions.....	41
Figure 48 Max fanout to have good STA estimation.....	41
Figure 49 : constrain the blocks of design. ....	42
Figure 50 Input to output timing path from Design compiler.....	42
Figure 51 Max delay to solve the problem. ....	42
Figure 52 Hierarchy of design .....	43
Figure 53: problem of Top-down strategy .....	43
Figure 54 the output files from the synthesis step .....	44
Figure 55 parameters under concern.....	45
Figure 56 The Logic Cone concept.....	49
Figure 57 Compare points of the cones of logic .....	50
Figure 58 Containers in a Hierarchical Design.....	51
Figure 59 Containers can hold design and technology libraries .....	52
Figure 60 the flow of The formality tool .....	52
Figure 61 Formality Read-Design Process Flow .....	54
Figure 62 Matches Corresponding Points between Designs.....	55
Figure 63 compare points is matched by their name between Designs .....	55
Figure 64 Verifies logical equivalence for each logic cone between Designs .....	56
Figure 65 PnR Flow .....	59
Figure 66 Detailed PnR flow .....	60
Figure 67 Design modules .....	60
Figure 68 Final Floor Plan .....	61
Figure 69 Power Planning (stripes).....	62
Figure 70 Power Planning (Rings).....	62
Figure 71 Trial Routing Results.....	62
Figure 72 Chip after Placement and Trial Routing.....	63
Figure 73 Timing and DRV's preCTS .....	63
Figure 74 Timing and DRV's after optimization .....	64
Figure 75 Detailed Routing Results.....	65
Figure 76 Chip after Detailed Routing.....	65
Figure 77 Connectivity Report.....	66
Figure 78 Geometry Report .....	66
Figure 79 Density after Filler insertion.....	66
Figure 80 Chip after Filler insertion .....	66
Figure 81 Area Distribution .....	68
Figure 82 Power Distribution.....	69
Figure 83 an overview of the top-module.....	71
Figure 84the top-module layout before merging .....	72
Figure 85 the top-module layout after merging .....	72



## List of Tables

Table 1 comparison between Gemmini and other DNN accelerator generators .....	15
Table 2 the outcomes of running inference on three images using our configuration for ResNet50 .....	25
Table 3 Setup timing result from Synthesis step .....	45
Table 4 Hold timing result from Synthesis step.....	46
Table 5 Table 4 Area result from Synthesis step .....	47
Table 6 1the result of the formal verification step.....	57
Table 7 Area Results .....	67
Table 8 Area Results for 22 nm FinFet technology [11] .....	68
Table 9 Power Results .....	69

# List of Abbreviations

"

". JSON"  
(JavaScript Object Notation) 21

(

(Design for Test)  
DFT 71

**A**

additive white Gaussian noise  
(AWGN) 4  
artificial neural networks  
(ANN) 9  
Automatic modulation classification  
(AMC) 3

**C**

central processing unit  
(CPU) 15  
Cognitive radio networks  
(CRN)  
cognitive radio users  
(CR) 1  
convolution neural network  
(CNN) 11

**D**

Deep Learning Model  
(DL) 19  
Deep neural network  
(DNN) 15  
Direct memory access  
(DMA) 30  
Dynamic random access memory  
(DRAM) 13

**F**

feature map  
(fmap) 10  
feature-based  
(FB) 3  
Field Programmable Gate Arrays  
(FPGAs) 18  
finite impulse response  
(FIR) 4  
Flexible Intermediate Representation for RTL  
(FIRRTL) 16

x

## G

General matrix multiplication  
(GEMM) 23  
Graphics processing unit  
(GPU) 19

## H

HAMMER  
(High-level Abstraction for Modularity, Extensibility, and Reusability) 18  
Hardware  
(HW) 17  
hardware description language  
(HDL) 16

## I

instruction set architecture  
(ISA) 17  
integrated circuits  
(ICs) 18  
intellectual property core  
(IP) 15

## L

likelihood-based  
(LB) 3  
local oscillator  
(LO) 4

## M

multiple regression  
(MR) 24  
Multiply And Accumulate  
(MAC) 27

## O

Open Neural Network Exchange  
(ONNX) 23  
Output Stationary  
(OS) 27

## P

Place-and-Route  
PNR *See*  
primary users  
(PU) 1  
processing element  
(PE) 13

## R

received signal strength  
(RSS) 2

rectified linear unit  
(ReLU) 9  
Register transfer level  
(RTL) 17  
RoCC  
(Rocket Custom Coprocessor) 17

## S

Signal-to-noise ratio  
(SNR) 2  
Static random-access memory  
(SRAM) 31  
systems-on-chip  
(SoC) 16  
Systolic Array  
(SA) 27

## T

Tensor Processing Unit  
(TPU) 27  
The Berkeley Out-of-Order Machine  
(BOOM) 17  
the radio frequency  
(RF) 5

## W

Weight Stationary  
(WS) 27

# 1. Introduction

## 1.1 Cognitive radio networks

With the enormous increasing demand in communication systems and the spectrum is a very scarce resource, new methodologies of communication are emerging to take advantage of the whole spectrum. The spectrum is not used efficiently as it's very congested in some parts while other parts are not used efficiently, the unlicensed bands in the spectrum which is the bands that could be used without a license like bands used by the Bluetooth or WIFI interfaces could be used for other types of communications like mobile communications to make use of these parts of the spectrum which are not congested like the licensed bands. [1]

The problem with that paradigm is that it requires the communication network to be intelligent and cognitive, to sense the spectral environment around it and find the spectral opportunities available. The network should also be able to change its operating parameters to dynamically make use of its spectral environment for an efficient spectrum utilization. [1]

Cognitive radio networks or CRN for short are networks that are capable of sensing and changing their operating parameters to make use from the spectral environment around them. As seen in Figure 1 To sense it's environment, information about the environment is gathered from devices of the primary users (PU) and cognitive radio users (CR) and other IoT devices and is sent to a near base stations and then these data are sent for processing to datacenters using internet, after processing the data from all network users the datacenters send back the decisions to the base stations. [2]

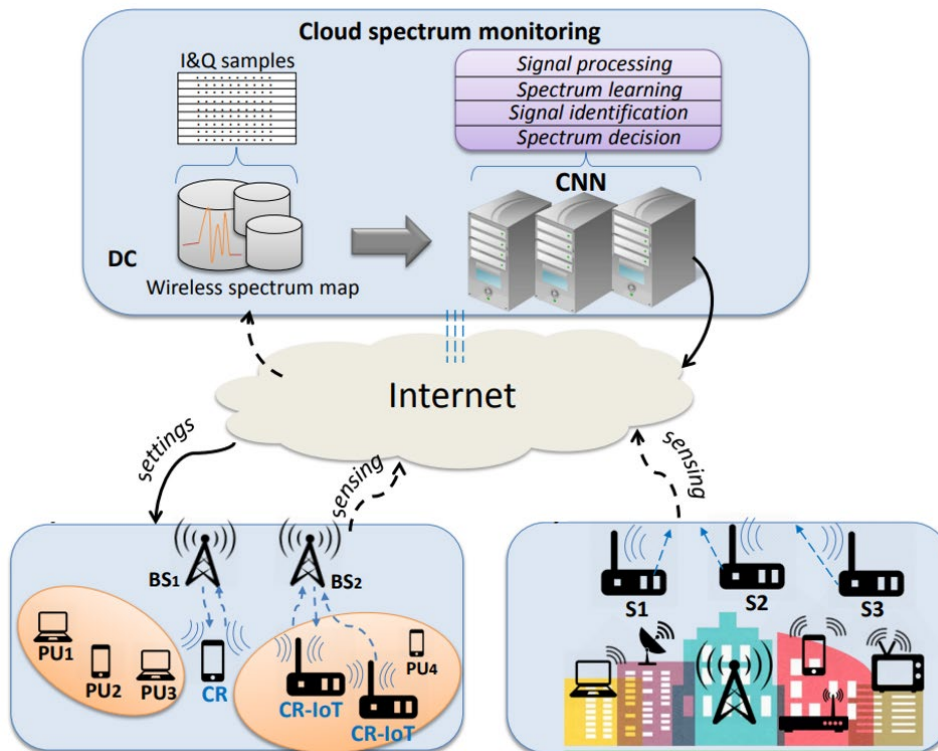


Figure 1: Spectrum management framework [2]

Cognitive radio networks have some challenges especially in the case of multiuser access, some are caused by fading, shadowing and interference.

Certain challenges faced during multiuser access are described as follows: [1]

1. Uncertainty in allocating the channel resource: Inconsistency in the received signal strength (RSS) due to channel fading can cause the CR to make wrong decision, which leads to violation of cognitive policies laid out for reasoning and learning from the communication environment. Highly sensitive radio devices can differentiate between faded primary signals and white spaces. The decision-making process is often challenged when the fading effect is severe, and the results are nonconclusive. In such circumstances, local readings of all the neighboring CRs are collated and processed to deduce the signal strength, frequency, and PU channels. Adaptive learning and sophisticated reasoning algorithms help the radio to allocate appropriate channels, thereby establishing seamless communication links to the primary licensed users.
2. Noise and attenuation uncertainty: Communication signal in the environment is susceptible to noise and attenuation losses. The factor of attenuation may vary in CRN; therefore, the impact of noise needs to be reduced for quality interpretation of signal in a consistent and coherent manner. Therefore, SNR is deduced, which compares the level of desired signal strength to random noise.
3. Uncertainty due to aggregated interference: Communication links of CRs are affected by the interference caused by different environmental variables present in the surrounding vicinity. Hence, inconsistency produced from the immediate environment can hamper the quality of communication of the PUs. Also, owing to the learning capability of CRs, aggregated interference deduced by a set of radios in the neighborhood having a similar set of environmental variables may lead to incorrect detection of the PU frequency, and may eventually cause trouble in sensing the entire network of cognitive users. The only solution here is to calibrate the sensitivity of the radios and compensate for the difference of aggregate interference at each radio or at the controller.
4. Interference limit: A couple of major cases contributing to this context are localization difficulty and listen-only mode. In first case, when the geographic location of the licensed receiver is unknown to the secondary user, the aggregate interference and compensation fail to be suitably analyzed. In the second case, the PU is calibrated to be in a listen-only mode, in which no feedback or communication could be acquired from the receiver radio to the PU. In such scenario, one-way communication takes place, because of which the primary transmitter/controller fails to deduce the aggregate interference at the receiving end. The preceding two cases inhibit the analysis of interference limits that could further lead to disturbances in the communication link of the PUs.

## 1.2 Modulation classification

Automatic modulation classification (AMC) that identifies the modulation type of the received signal is an essential part of noncooperative communication systems. The AMC plays an important role in many civil and military applications such as cognitive radio, adaptive communication, and electronic reconnaissance. In these systems, transmitters can freely choose the modulation type of signals; however, the knowledge of modulation type is necessary to the receivers to demodulate the signals so that the transmission can be successful. AMC is a sufficient way to solve this problem with no effects on spectrum efficiency. [3]

AMC algorithms have been widely studied in the past 20 years. In general, conventional AMC algorithms can be divided into two categories: likelihood-based (LB) and feature-based (FB). LB methods are based on the likelihood function of the received signal, and FB methods depend on feature extraction and classifier design. [3]

## 1.3 Wireless signal model

A wireless communication system transmits information from one point to another through a wireless medium which is called a channel. At the system level, a wireless communication model consists of the following parts:

**Transmitter.** The transmitter transforms the message, i.e., a stream of bits, produced by the source of information into an appropriate form for transmission over the wireless channel. Figure 3 shows the processing chain at the transmitter side.

First, the bits  $b_k \in \{0, 1\}$  are mapped into a new binary sequence by a coding technique. The resulting sequence is mapped to symbols  $s_k$  from an alphabet or constellation which might be real or complex. This process is called modulation.

In the modulation step, the created symbols are mapped to a discrete waveform or signal via a pulse shaping filter and sent to the digital to analog converter module (D/A) where the waveform is transformed into an analog continuous time signal,  $s_b(t)$ . The resulting signal is a baseband signal that is frequency shifted by the carrier frequency  $f_c$  to produce the wireless signal  $s(t)$  that is defined by:

$$s(t) = \Re\{s_b(t)e^{j2\pi f_c t}\} = \Re\{s_b(t)\} \cos(2\pi f_c t) - \Im\{s_b(t)\} \sin(2\pi f_c t) \quad (1)$$

Where  $s(t)$  is a real-valued bandpass signal,  $f_c$  is the carrier frequency for the channel and  $s_b(t)$  is the baseband signal.

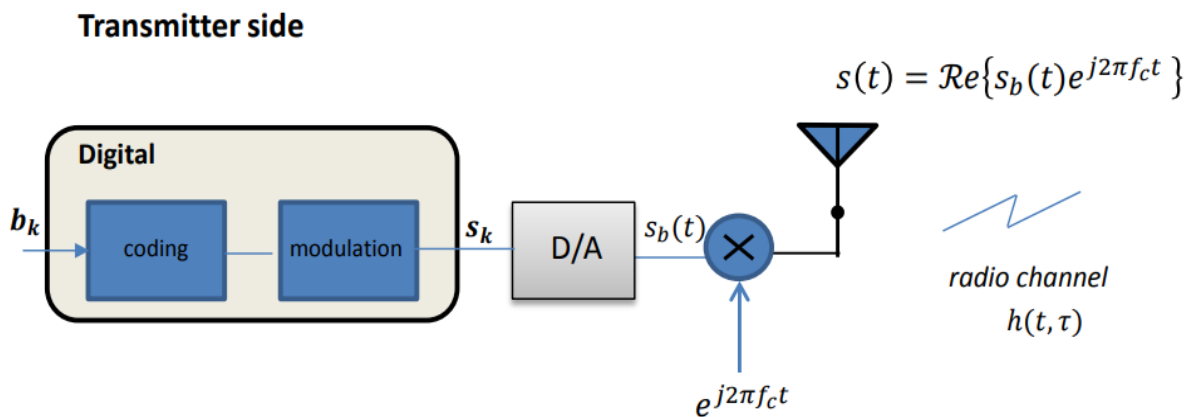


Figure 2: Transmitter diagram

**Wireless channel.** The wireless channel is characterized by the variations of the channel strength over time and over frequency. The variations are modeled as (i) *large-scale fading*, which characterizes the path loss of the channel as a function of distance and shadowing by large objects such as buildings and hills, and (ii) *small-scale fading*, which models constructive and destructive interference of the multiple propagation paths between the transmitter and receiver. The channel effects can be modeled as a linear time-varying system described by a complex finite impulse response (**FIR**) filter  $h(t, \tau)$ . If  $r(t)$  is the signal at the channel output, the input/output relation is given by:

$$r(t) = s(t) * h(t, \tau)$$

where  $h(t, \tau)$  is the band-limited bandpass channel impulse response, while  $*$  denotes the convolution operation.

**Receiver.** The wireless signal at the receiver output will be a corrupted version of the transmitted signal due to channel impairments and hardware imperfections of the transmitter and receiver. Typical hardware related impairments are:

- Noise caused by the resistive components such as the receiver antenna. This thermal noise may be modelled as additive white Gaussian noise (AWGN),  $n \sim N(0, \sigma^2)$ .
- Frequency offset caused by the slightly different local oscillator (LO) signal frequencies at the transmitter,  $f_c$ , and receiver,  $f'_c$ .
- Phase Noise,  $\phi(t)$ , caused by the frequency drift in the LOs used to demodulate the received wireless signal. It causes the angle of the LO signals to drift around its intended instantaneous phase  $2\pi f_c t$ .
- Timing drift caused by the difference in sample rates at the receiver and transmitter.
- The received wireless signal model can be given by

$$r(t) = \Re\{r_b(t)e^{j\omega_c t}\} \quad (2)$$

Where  $r_b(t)$  is the baseband complex envelop defined by

$$r_b(t) = (s_b(t) * h_b(t, \tau)) \cdot \frac{1}{2} e^{j(\omega_c - \omega'_c)t} + n(t) \quad (3)$$

Where  $h_b(t, \tau)$  is the baseband channel equivalent given by

$$h_b(t, \tau) = \sum_{i=0}^l \alpha_i(t, \tau) e^{j\omega_c \tau_i(t) + \phi_i(t, \tau)} \delta(\tau - \tau_i(t)) \quad (4)$$

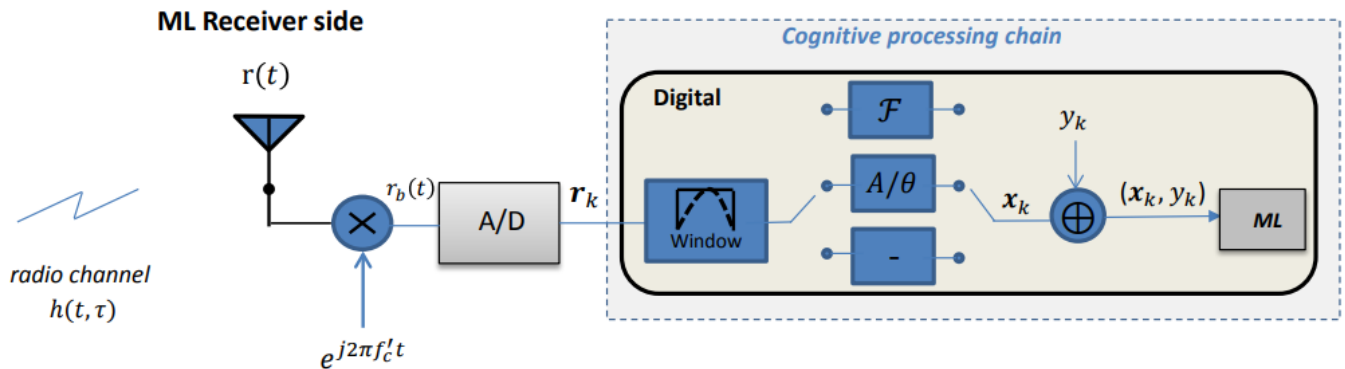


Figure 3: Receiver diagram



## 1.4 Fading

Fading is the happens due to reflection, refraction and pathloss that happens for in the propagation of electromagnetic waves. Reflection is when the electromagnetic signal encounters a surface with particles with large dimensions than the wavelength of the signal, while refraction happens in case of particles with dimensions near the wavelength of the signal. [4]

There are many propagation models used in modeling the fading effect, but they're categorized into two categories, large-scale and small-scale propagation models. Large-scale propagation models account for the mean of the signal which is used to define the radio frequency (RF for short) signal coverage. Small-scale models on the other hand are used to represent the rapid fluctuations around the large-scale loss in the received signal in small distances or small periods of time. [4]

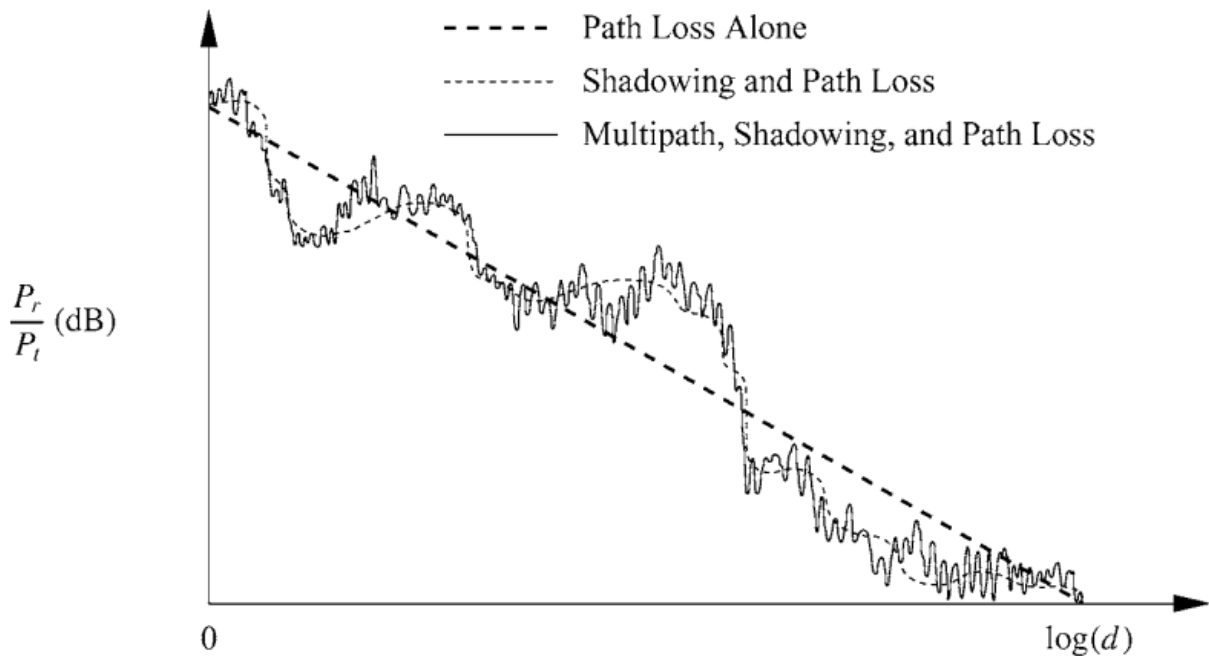


Figure 4: transmission gain with respect to logarithm of the distance.

### 1.4.1 large-scale fading

One of the large-scale propagation models that is used is the Friis free space path loss model given in equation 1 :

$$FS(dB) = 10 \log_{10} \left( \left( \frac{\lambda}{4\pi d} \right)^2 \right) \quad (5)$$

Where:

$\lambda$ : wavelength

$d$ : Transmitter to receiver separation distance

$$P_r(dB) = P_t + G_t + G_r - FS(dB) \quad (6)$$

Where:

$P_r$ : received power

$P_t$ : transmitted power

$G_t$ : transmitter gain

$G_r$ : receiver gain

This model is used for simple path loss estimations because of its simple form and limited number of required parameters.

### 1.4.2 Small-scale fading

In the figure shown the transmission gain is calculated through distance, the small fluctuations around the large-scale fading. The small fluctuations caused by the multipaths the signal takes from the transmitter to the receiver is the main reason for small-scale fading and it could be viewed as a time-varying channel response.

The figure shows the response of a non-stationary channel where its response is given as mentioned in eq.3.

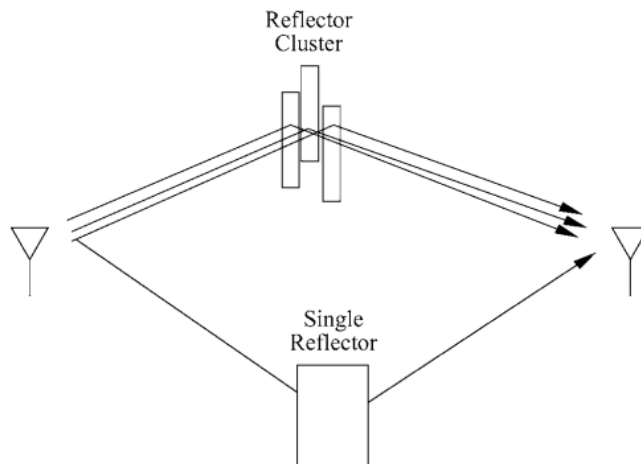


Figure 5: Multipath propagation illustration. [4]

If a single pulse is transmitted over a multipath channel, then the received signal will appear as a pulse train, with each pulse in the train corresponding to the line-of-sight component or a distinct multipath component associated with a distinct scatterer or cluster of scatterers. The time delay spread of a multipath channel can result in significant distortion of the received signal. This delay spread equals the time delay between the arrival of the first received signal component (LOS or multipath) and the last received signal component associated with a single transmitted pulse. If the delay spread is small compared to the inverse of the signal bandwidth, then there

is little time spreading in the received signal. However, if the delay spread is relatively large then there is significant time spreading of the received signal, which can lead to substantial signal distortion. [4]

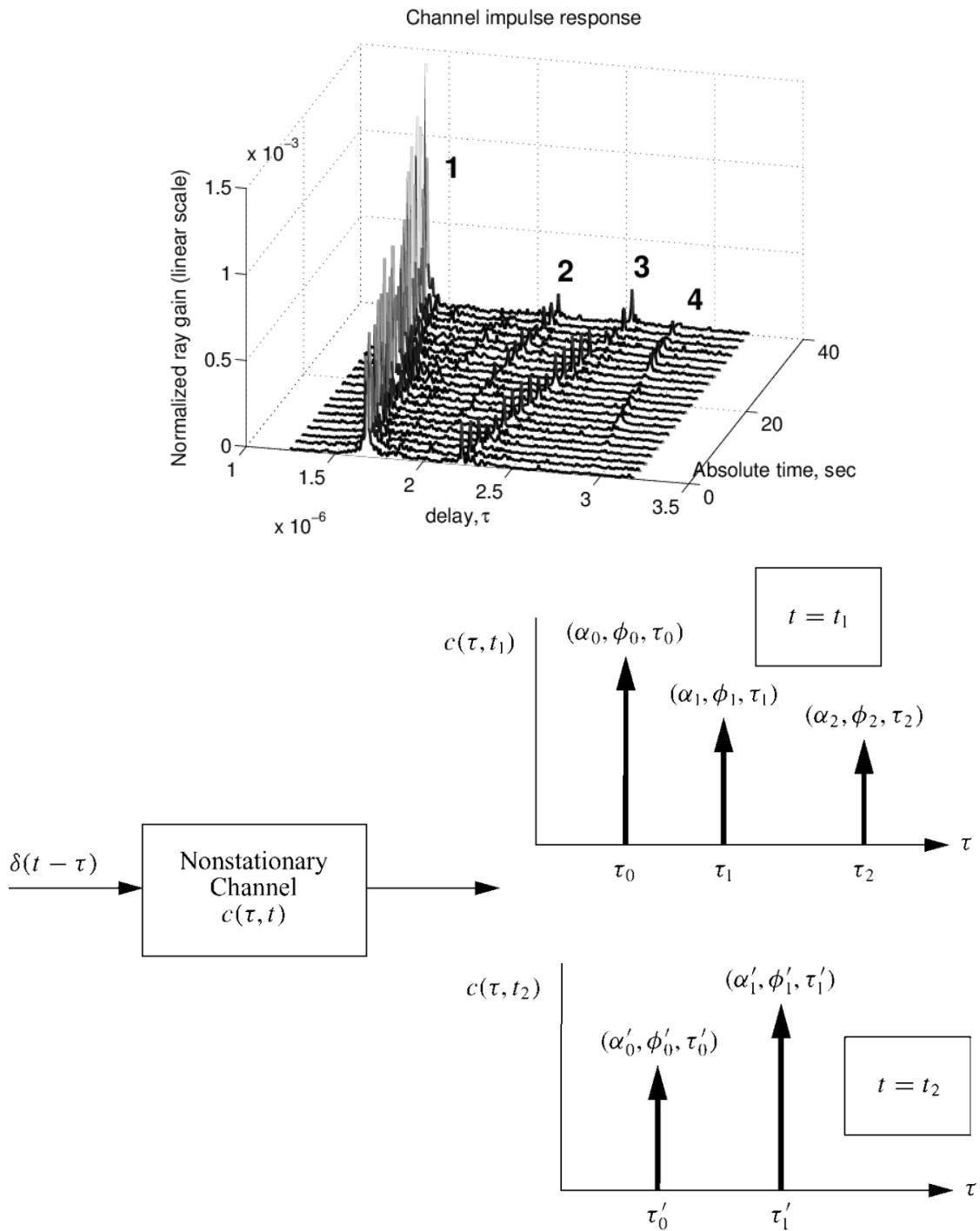


Figure 6: Response of non-stationary channel. [4]

## 1.5 Data acquisition

Figure 3 summarizes the data acquisition process for collecting wireless signal features. The received signal,  $r(t)$ , is first amplified, mixed, low-pass filtered and then sent to the analog to digital (A/D) converter, which samples the continuous-time signal at a rate  $f_s = \frac{1}{T_s}$  samples per second and generates the discrete version  $r_n$ . The discrete signal  $r_n = r[nT_s]$  consists of two components, the in-phase,  $r_I$ , and quadrature component,  $r_Q$ , i.e., [2]

$$r_n := r[n] = r_I[n] + jr_Q[n] \quad (7)$$

Suppose we sample for a period  $T$  and collect a batch of  $N$  samples. The signal samples  $r[n] \in \mathbb{C}$ ,  $n = 0, \dots, N - 1$ , are a time-series of complex raw samples which may be represented as a data vector. The  $k$ -th data vector can be denoted as

$$r_k = [r[0], r[1], \dots, r[N - 1]]^T \quad (8)$$

These data vectors  $r_k$  are windowed or segmented representations of the received continuous sample stream, similarly, as is seen in audio signal processing. They carry information for assessing which type of wireless signal is sensed. This may be the type of modulation, the type of wireless technology, interferer, etc. [2]

The data used in the classification is derived in frequency form by using Fourier transformation to each sampled vector of the time series data, and the amplitude phase form as each form have more accuracy when used in certain classification tasks in CRN as shown in the figure. [2]

## 1.6 Deep learning

CRN uses many deep learning techniques to tackle problems like modulation classification and interface detection so an introduction about machine learning and its uses is discussed in this section. Followed by a short discussion about how to accelerate machine learning algorithms to be used efficiently in battery-based applications where power and computational resources are very limited.

Deep learning algorithms are a part of the machine learning field which uses multiple layers nodes to approximate highly non-linear relations in highly dimensional vector spaces with a computationally efficient processing nodes. Considering the processing node, it's simply a weighted sum of the inputs with an activations function before the output as shown in figure 8. [5]

Each node is called a perceptron and is derived based on the theory of how neurons work, it's assumed that neurons sense the electrical current from its dendrites and uses a weighted sum for the currents and an activation function to produce the final output signal through the axon. And for that reason, structures containing many layers of nodes are referred to as neural networks. [5]

The need for activation function is extra important, given the structure of a neural network shown in figure 7, if there isn't a non-linear activation function the structure would act as a single perceptron due to the linear function in each node. Non-linear activation functions help the structure to predict the non-linear relation between the input and the output with a high accuracy.

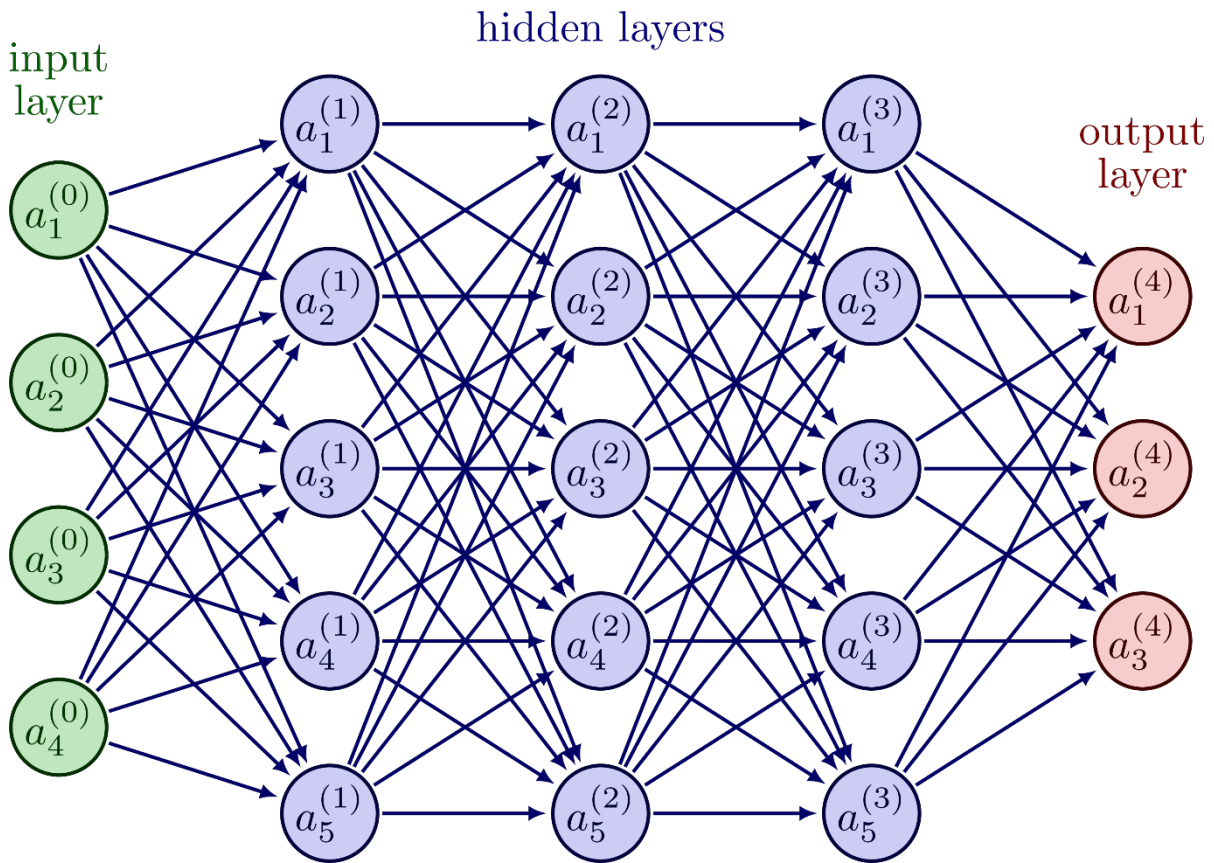


Figure 7: Neural network example.

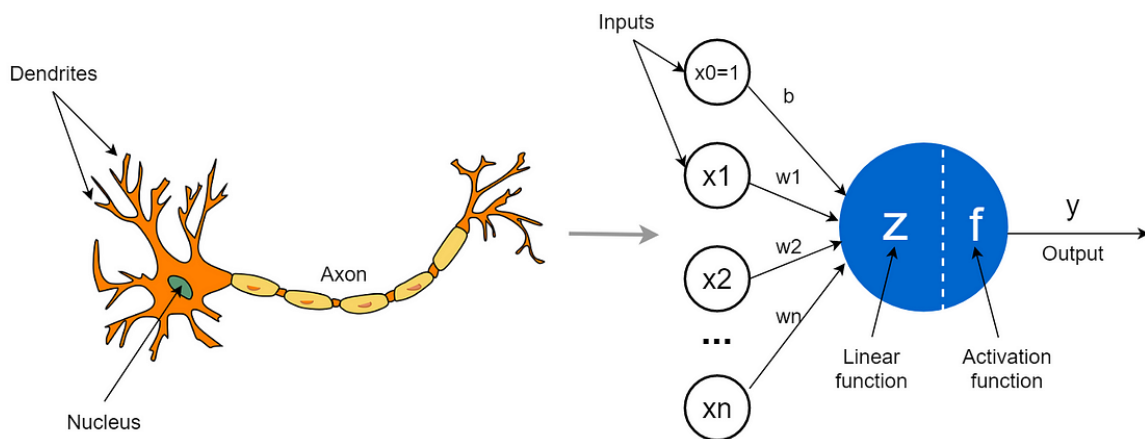


Figure 8: Perceptron.

One of the famous activation functions used in deep neural networks are shown in figure below, in hidden layers it's common to use the rectified linear unit (ReLU) function due to its very simple implementation. Some problems with the ReLU function arise like the dead-ReLU problem so some artificial neural networks (ANN) use the leaky-ReLU function to overcome the problem. [6]

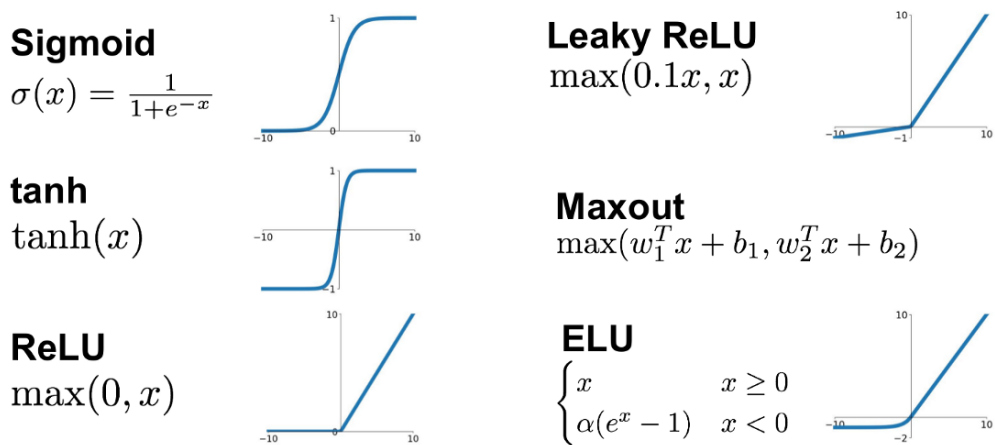


Figure 9: Common activation functions.

The neural network mentioned above is usually referred to as fully connected network as each neuron has a different weight for each input synapse. Most of the time there's some common distribution between the inputs to the neurons so it's more convenient to share weights between them. The redundancy of the fully connected architecture is obvious when pruning the structure to remove the less effective weights and redo training for the network to find the appropriate weights (figure 10). [6]

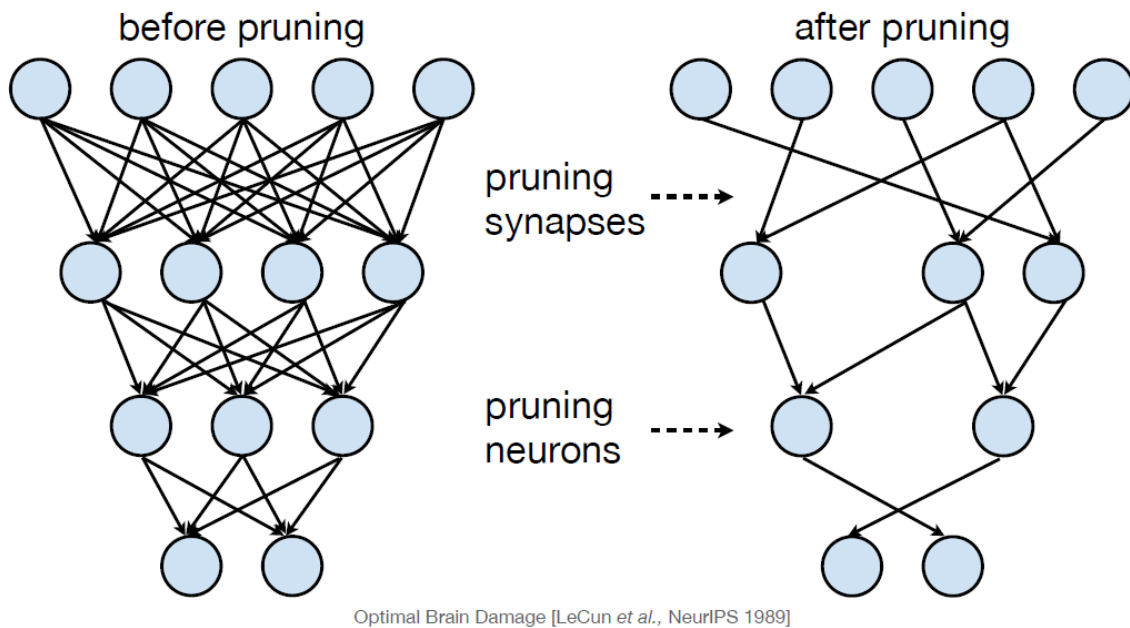


Figure 10: a fully connected layer before and after pruning.

The weight sharing paradigm could be achieved using a kernel being convolved with the output of each layer, and each node in the kernel is a weight and after each convolution with the kernel there's an activation function. The benefit of convolution is that different elements in the input vector to each layer share the same weight making the network more efficient with less redundancy in the structure. The figure below illustrates how convolution between a kernel and input feature map (fmap) to the convolution layer. [6]

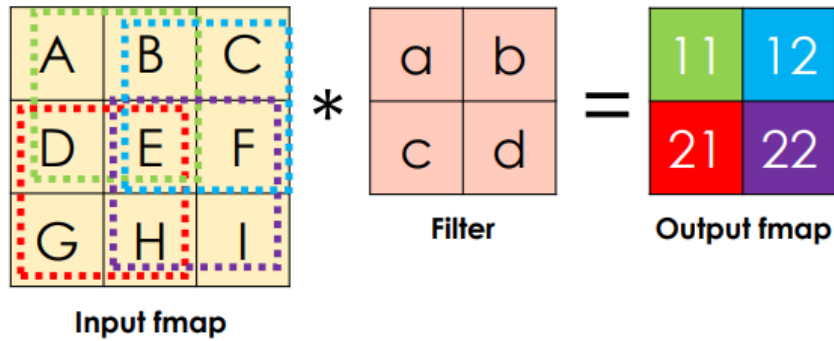


Figure 11: convolution example.

Figure 12 shows a convolution neural network (CNN) example, between each two convolution layers there's a max pooling layer which choose the max feature in each 2-by-2 block in the feature maps, an example of max pooling is shown in figure 13. [6]

### 1.7 Deep learning acceleration

Deep learning algorithms infer lots of computation especially multiplication as it requires more power and time than simple addition and subtraction. Another problem with deep learning is the precision of calculations, floating point operations is more time and power consuming than fixed point operations, and the fact that deep learning algorithms are developed and tested in the floating point domain, transforming the model -mainly the weights- to the fixed-point domain is needed to reduce the area and power of the accelerator design with taking much care of the loss in accuracy could be resulted due to that. [6]

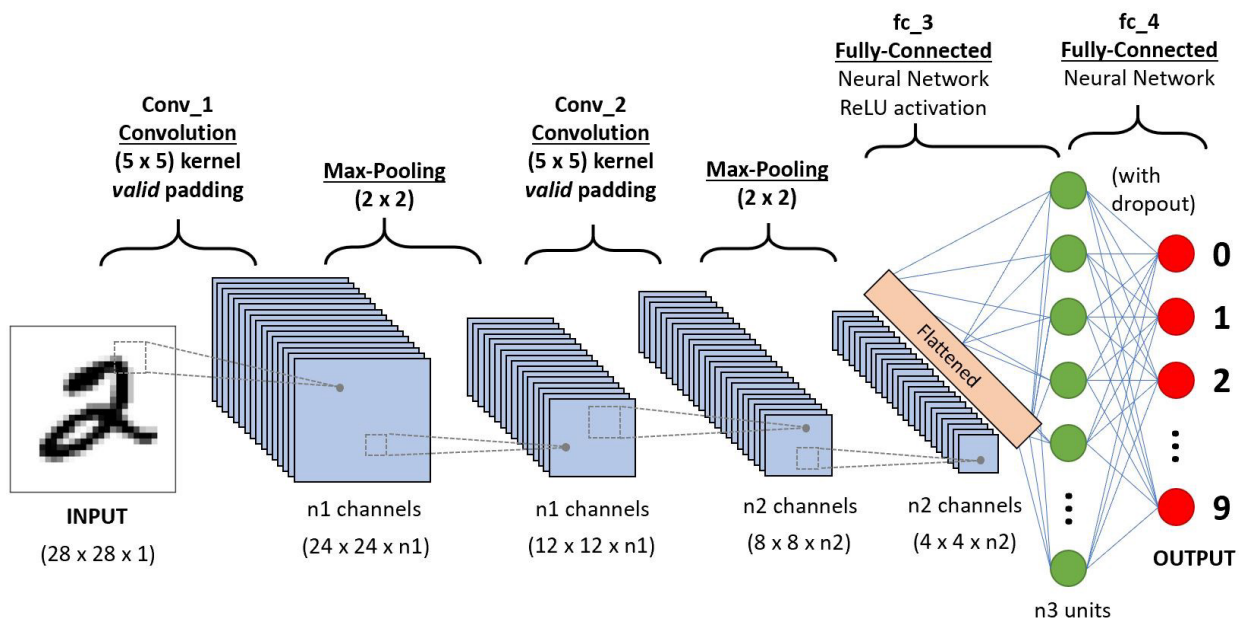


Figure 12: CNN example



Figure 13: max pooling example

The digital design for each node is simply a multiply and accumulate unit with some internal registers to store the input activation (shown in figure 14), weight and output partial sum. As seen from previous figures ANN architectures commonly contain lots of computational nodes that makes it almost impossible to make a direct implementation by simply interchanging each node in the signal flow graph with its implementation, the design will need more redundant area and power without benefiting from the regularity in the deep learning algorithm. [6]

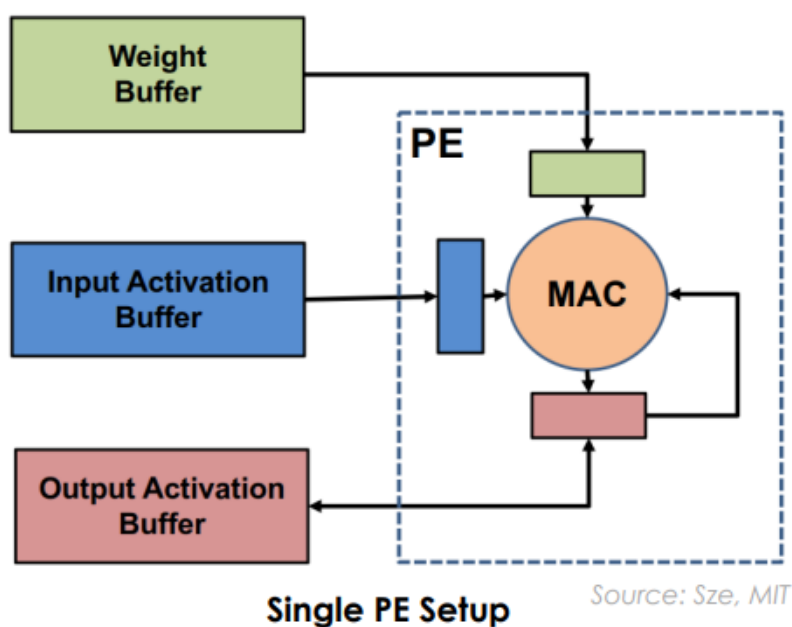


Figure 14: Processing engine [6]

To overcome that problem, the use of time sharing is mandatory. The problem of time sharing is that it complicates the controller part of the design. The figure below shows an example of the use of time sharing in deep learning acceleration with each processing engine having a control unit to control the flow of signals and data, and registers to store inputs and outputs. [6]



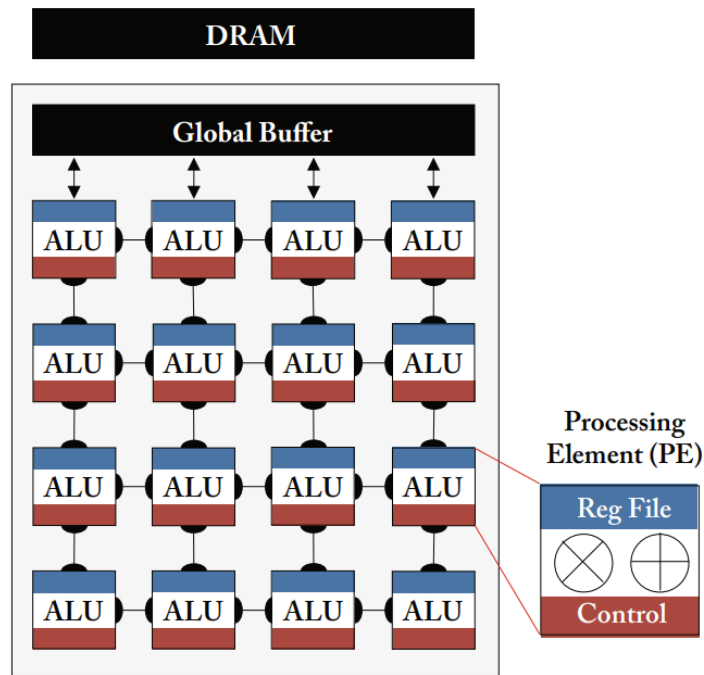


Figure 15: time sharing in deep learning acceleration. [6]

There're different ways for the data flow between the accelerators to benefit from the regularity in computation of convolution, figure 16 shows an example of weight stationary data flow taxonomy. The weights of the kernel are loaded in the PEs and the input activations is loaded in parallel and partial sums are moved sequentially in the PE chain and after each output feature element is computed, the input activations is changed with the weights being stationary until all outputs associated with these weights are computed to the movement of weights from the DRAM to the buffer is minimized. [6]

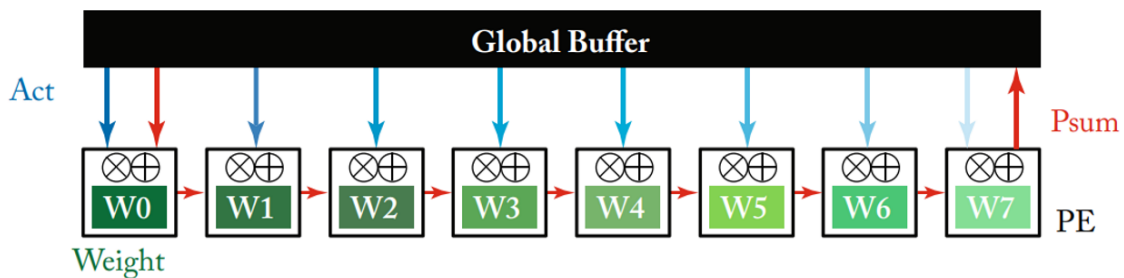


Figure 16: weight stationary. [6]

The output stationary is also a common taxonomy used in deep learning acceleration, as shown in figure 17, it's same as the output stationary but with the partial sums being stationery and input activations and weights of different kernels is loaded in parallel and partial sums are moved sequentially after being computed in each PE. [6]

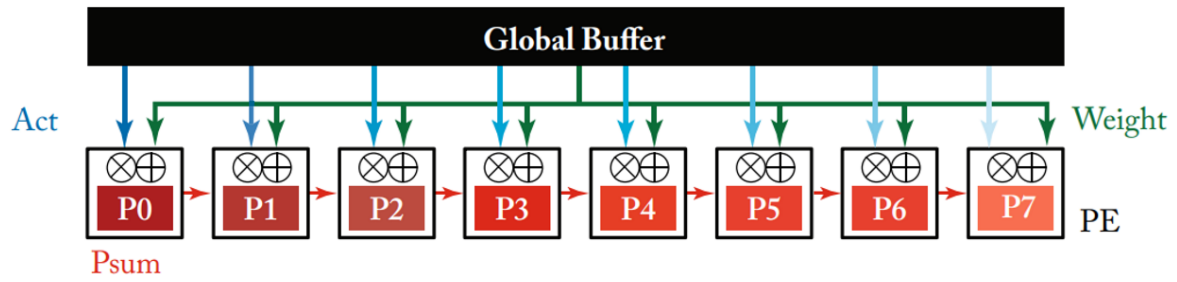


Figure 17: output stationary. [6]

## 2 Gemini and Chipyard Framework

As we discussed in chapter one Hardware accelerators play a crucial role in addressing the ever-increasing computational demands of modern applications, particularly in fields such as artificial intelligence, data analytics, and scientific simulations. These specialized hardware components, like the Gemini accelerator, are designed to efficiently perform specific tasks by offloading computation from general-purpose processors. The importance of hardware accelerators lies in their ability to significantly enhance performance and energy efficiency, enabling faster processing and reduced power consumption compared to traditional CPUs. However, implementing hardware accelerators comes with its own set of challenges. One major challenge is the design complexity involved in creating specialized architectures tailored to specific workloads.

To address the complexity involved in designing hardware accelerators, several strategies can be employed, leveraging frameworks such as NVDLA, Gemini, and TPU, among others. These frameworks provide valuable resources and tools that simplify the accelerator design process and help overcome challenges.

One approach is to utilize one of these frameworks which offer pre-designed and verified hardware accelerator IP cores specifically tailored for deep learning and matrix multiplication tasks, respectively. These frameworks provide a starting point for us, enabling us to focus on customization and integration rather than starting from scratch. By leveraging these specialized frameworks, we can accelerate development, reduce design complexity, and benefit from the expertise embedded in these ready-to-use accelerator designs

### 2.1 Why Gemini: The Choice Among Other Tools:

We have chosen Gemini in our project as the hardware accelerator due to its unique features and advantages compared to other existing options. One key factor is Gemini's flexibility in providing parameterizable architectural templates, enabling the generation of a wide variety of hardware and software instances. This feature significantly improves hardware design productivity and allows for systematic evaluation of DNN architecture.

*Table 1 Comparison between Gemini and other DNN accelerator generators*

	Property	NVDLA	VTA	PolySA	DNNBuilder	MAGNet	DNNWeaver	MAERI	Gemini
Hardware Architecture Template	Datatypes	Int/Float	Int	Int	Int	Int	Int	Int	Int/Float
	Dataflows	✗	✗	✓	✓	✓	✓	✓	✓
	Spatial Array	vector	vector	systolic	systolic	vector	vector	vector	vector/systolic
	Direct Convolution	✓	✗	✗	✓	✓	✓	✓	✓
Programming Support	Software Ecosystem	Compiler	TVM	SDAccel	Caffe	C	Caffe	Custom	ONNX/C
	Virtual Memory	✗	✗	✗	✗	✗	✗	✗	✓
System Support	Full SoC	✗	✗	✗	✗	✗	✗	✗	✓
	OS Support	✓	✓	✗	✗	✗	✗	✗	✓

As shown in Table 1 which compares the Gemini with other DNN accelerator generators Gemini stands out by offering flexible architectural templates that cover various DNN accelerators, each suitable for different execution environments and area/power/performance targets. Unlike many existing generators that focus on specific representations or dataflows, Gemini supports both floating and fixed-point data types, multiple dataflows, and both vector and systolic spatial array architectures. This versatility enables a quantitative comparison of efficiency and scalability differences, making it easier to evaluate and compare against other accelerators.

Another advantage of Gemini is its multi-level programming interface, catering to different user requirements. It provides an easy-to-use programming interface for DNN application practitioners who prefer working with high-level frameworks, such as PyTorch or TVM. At the same time, it offers low-level

programming capabilities in C/C++ or assembly for framework developers and system programmers who need precise control over hardware states.

Furthermore, Gemmini addresses the critical aspect of system-level integration, which is often overlooked by other DNN accelerator generators. It supports full SoC integration with host CPUs, shared resources like caches and system buses, and even booting Linux. This capability allows architects to evaluate subtle trade-offs at the system level, considering the interaction and integration of the accelerator within a larger system.

## 2.2 Chipyard Ecosystem:

Gemmini is part of the Chipyard ecosystem which is a framework for designing and evaluating the full system. It is composed of a collection of tools and libraries designed to provide an integration between open-source and commercial tools for the development of systems-on-chip (SoC). The next figure shows the Chipyard ecosystem. [7]

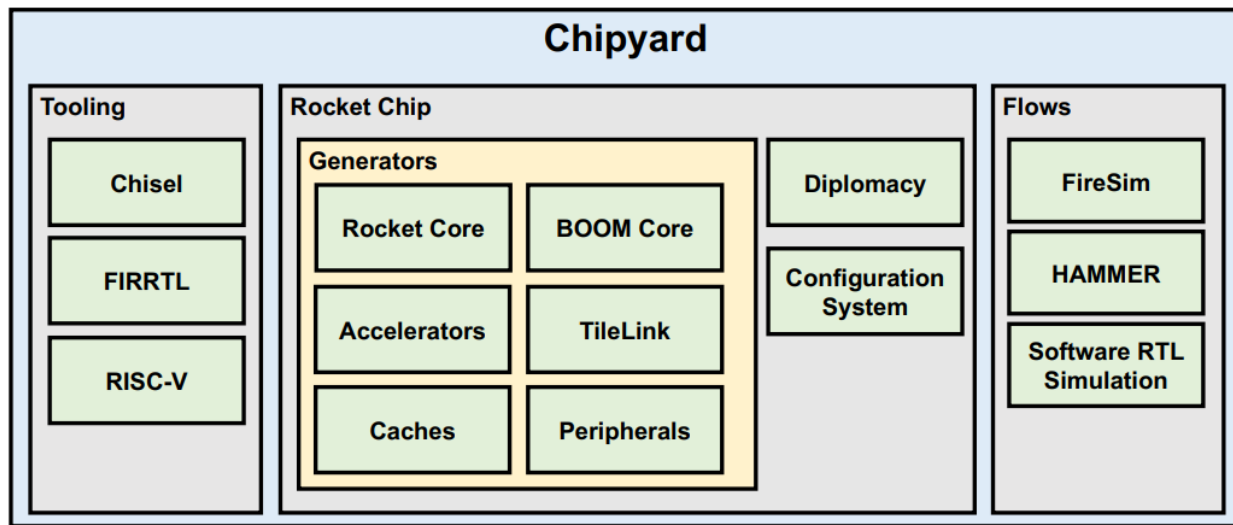


Figure 18 Chipyard Ecosystem

From the previous figure before starting the Gemmini design flow, we will summarize some important components in the chipyard ecosystem and their functionality.

### 2.2.1 Chisel:

Chisel is a hardware description language (HDL) embedded within the Scala programming language. It provides a concise and expressive syntax for describing digital circuits and enables the generation of optimized synthesizable Verilog or VHDL code. Chisel combines the power of Scala's functional programming constructs with a hardware construction library, allowing us to create reusable and composable hardware components. Its integration with the Scala ecosystem provides access to a wide range of tools, libraries, and utilities, enhancing productivity in the hardware design process.

### 2.2.2 FIRRTL:

FIRRTL (Flexible Intermediate Representation for RTL) is an intermediate hardware representation language developed by Berkeley. It serves as a common format for specifying digital circuits and allows for efficient transformations and optimizations. It acts as the bridge between Chisel and the subsequent stages of the

design flow. The FIRRTL compiler takes the Chisel-generated FIRRTL code and performs various transformations and optimizations to generate the corresponding RTL. Figure 19 shows the steps of converting the Chisel to synthesizable Verilog RTL codes ready for HW implementations. [7] [8] [9]

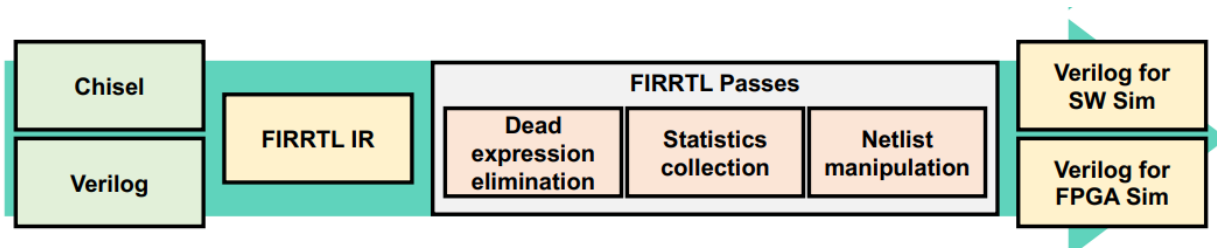


Figure 19 Process of converting Chisel to Verilog [8]

### 2.2.3 RISC-V:

RISC-V is an open-source instruction set architecture (ISA) developed at Berkeley. It is designed to be simple, modular, and extensible, making it highly versatile and adaptable to various computing applications. RISC-V's open nature allows for unrestricted use, modification, and implementation, fostering innovation and collaboration in the development of processors and related hardware. [7]

### 2.2.4 Rocket core and BOOM core:

Rocket core and BOOM core are open-source processor cores based on the RISC-V instruction set architecture. Rocket core is an in-order processor core, while BOOM core is an out-of-order core. These cores serve as versatile starting points for designing custom RISC-V processors, with Rocket Chip generator framework in the Chipyard ecosystem enabling the generation of customizable RISC-V SoCs. [7]

### 2.2.5 RoCC (Rocket Chip Coprocessor) Accelerator generators:

RoCC (Rocket Custom Coprocessor) accelerator generators are a framework within the Rocket Chip generator that enables the design and integration of custom hardware accelerators. With the RoCC accelerator generators, we can create specialized coprocessors like **Gemmini**, Hwacha, and SHA3 that extend the functionality of the Rocket processor. These generators provide a customizable template and infrastructure for designing and integrating custom accelerators, facilitating seamless communication and interaction between the accelerators and the Rocket processor.

The architecture of the Rocket chip encompasses various essential components such as peripherals, memory units, and system buses. These elements play significant roles in enabling the functionality and overall operation of the Rocket chips. Figure 20 below provides an overview of the architecture, showcasing the integration and interconnection of these components within the Rocket Chip. [8] [9]

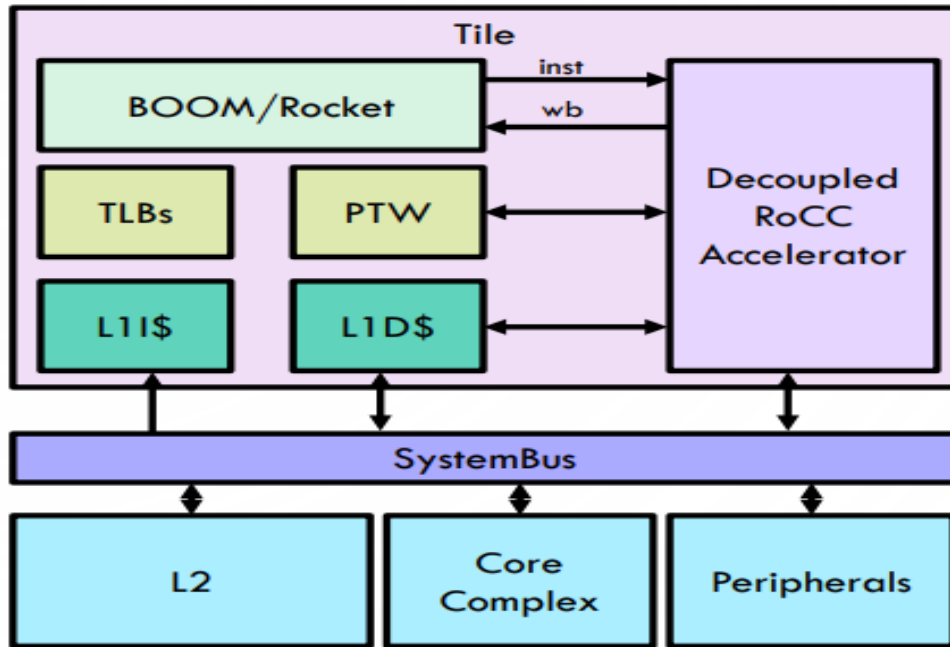


Figure 20 Rocket SoC for accelerators

In the next chapter, we will discuss the detailed architecture of the Gemmini accelerator and how it communicates with the whole system.

## 2.2.6 FireSim:

FireSim is a key component of the Chipyard ecosystem developed by Berkeley. It is a cycle-accurate, FPGA-accelerated simulation framework that allows for full-system simulation of RISC-V SoCs. FireSim enables us to evaluate and validate their designs before physical fabrication by emulating the entire system, including the Rocket processor, accelerators, and peripheral devices. This powerful tool provides a high level of fidelity and performance for architectural exploration, software development, and verification in the Chipyard environment. FireSim supports running on Amazon EC2 F1 FPGA-enabled cloud instances and locally managed Linux machines with FPGAs attached. [7] [10]

## 2.2.7 HAMMER:

HAMMER (High-level Abstraction for Modularity, Extensibility, and Reusability) is an influential physical design flow tool that plays a crucial role within the Chipyard ecosystem. This tool offers a modular and reusable approach to physical design, streamlining the process of creating custom integrated circuits (ICs) and accelerating the development cycle. HAMMER enables us to work at a higher level of abstraction by providing a set of well-defined interfaces and design abstractions. It allows for the seamless integration of different design components, such as cores, memories, and interconnects, fostering modularity and reusability. By utilizing HAMMER, we can focus on the specific functionality and performance of their custom ICs, while the tool automatically handles aspects like placement, routing, and optimization. This modular and reusable design flow not only enhances productivity but also promotes collaboration and innovation within the Chipyard ecosystem, ultimately enabling the rapid development of high-performance and energy-efficient hardware solutions. [7] [11] [12]

### 3 Methodology and Design Flow (Frontend):

After exploring the various components of the Chipyard ecosystem in the previous chapter, we now shift our focus to the Gemmini accelerator frontend design flow in this chapter. The frontend design flow is a crucial stage in the development of Gemmini accelerators, encompassing several essential steps. In this chapter, we will delve into each of these steps, providing an overview of the process and highlighting key considerations and methodologies employed. By understanding the Gemmini accelerator frontend design flow, we can gain insights into the intricate design decisions and optimizations involved in building high-performance and efficient accelerators within the Chipyard framework.

#### 3.1 Starting with DL Model:

As we aim to implement a Hardware computing device (Accelerator) to run a Deep Learning Model (DL) in our case is a convolution neural network model making a Modulation recognition inference so, our starting point will be with it.

Figure 21 shows the possible ways to run the DL model. There are three methods to execute and run DL models. The first approach is to run the model on the host processor or host GPU of a computer. However, this is not an ideal choice as it consumes a significant amount of time to run inference since these processors are not specifically designed for DL tasks, which are computationally intensive. [13] [14]

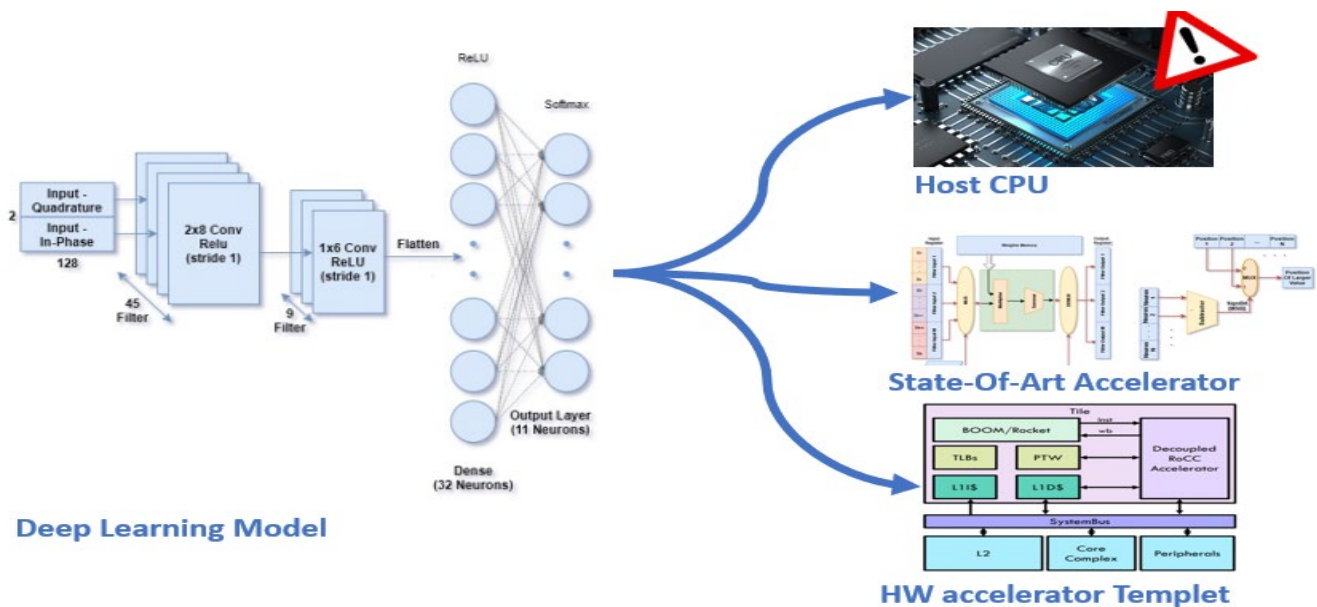


Figure 21 Approaches of running DL models.

The second method involves designing a specialized hardware (HW) accelerator tailored to the DL model. This approach has the potential to offer the best performance in terms of area, power, and speed. However, it is a complex and time-consuming process that requires designing and verifying the HW accelerator specifically for the DL model. Moreover, this specialized HW accelerator is not capable of running inferences for other DL models, limiting its versatility. [13]

The third and final approach is to utilize a pre-designed HW template, optimized for running DL models efficiently in terms of power and area. In this method, we write configuration files using a high-level language, and tools convert our design specifications into a synthesizable RTL (Register Transfer Level). This approach

also encompasses additional tasks such as designing the entire System-on-Chip (SoC), testing, and profiling performance. Essentially, we begin with the DL model and use the configurations to design the accelerator accordingly. In the following chapter, we will provide a brief overview of how these configurations are written. [11] [8]

### 3.2 Gemmini accelerator Design Flow for designing accelerators:

Once we have thoroughly studied the DL model or the specific requirements of the generic accelerator, we proceed with the design flow as depicted in the following figure. This flow involves several steps, beginning with the creation of configuration files that describe the behavior of the HW architecture design.

After completing the configuration files, we move on to testing the RTL (Register Transfer Level) implementation. This step ensures that the design functions as intended and meets the desired specifications. Through rigorous testing, we can identify and address any potential issues or bugs.

Once the RTL has been successfully tested, we proceed to evaluate the performance of the accelerator. This can be done using an FPGA (Field-Programmable Gate Array) or by obtaining the final layout of the design and its performance. [8] [9]

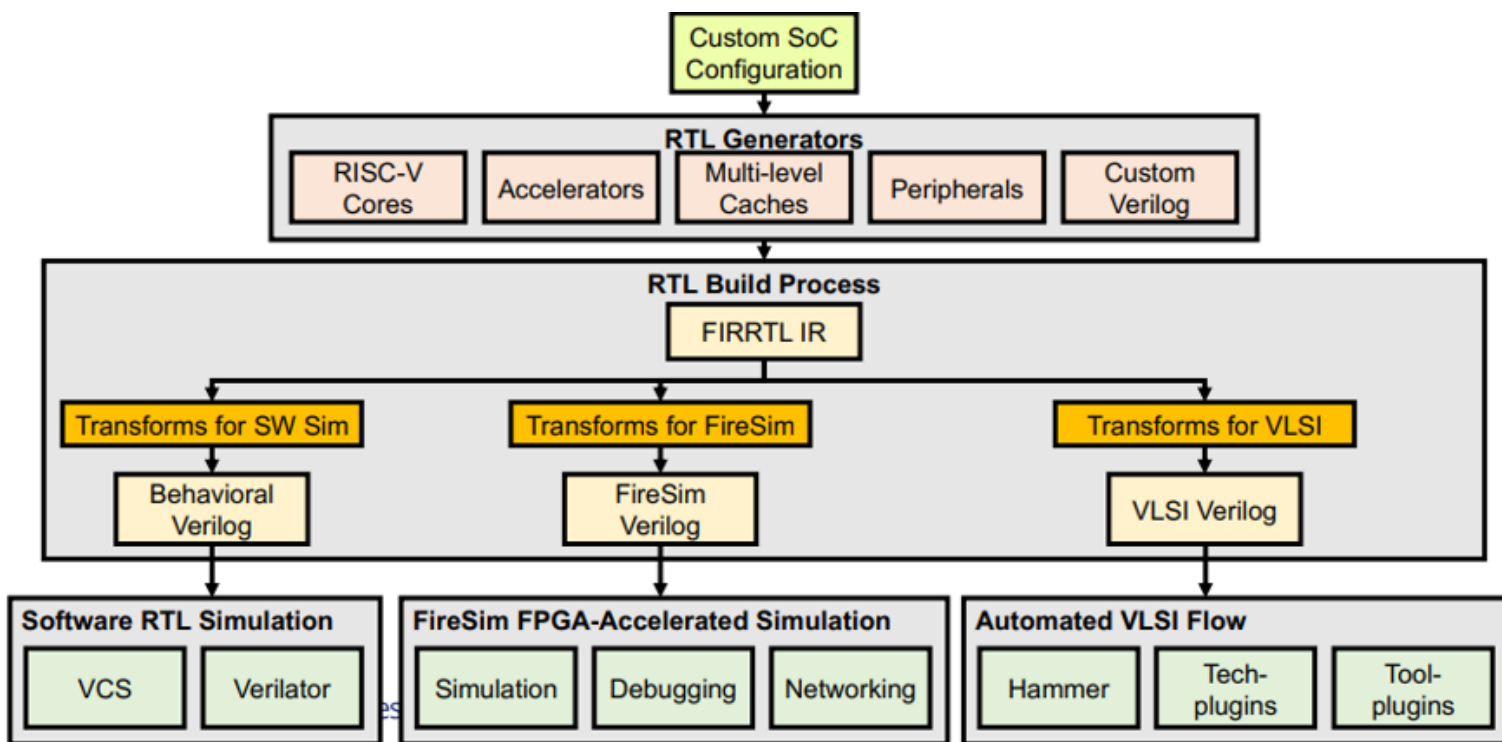


Figure 22 Gemmini Accelerator Different design flows.

### 3.3 Custom SoC configuration for Different RTL generators:

Once we have conducted a thorough study of the DL model and determined the requirements and specifications for the HW accelerator designed for spectrum sensing purposes, we begin the process of writing Chisel configurations for the accelerator. These configurations will later be converted to RTL using a specialized tool.

In addition to configuring the accelerator, we also have control over the entire system's configurations, as depicted in Figure 20. This includes specifying configurations for the host CPU, L2 Cache, DRAM, and other



peripherals. Since we have limited time to fully comprehend the Chisel language, we primarily rely on default configurations and make only edits to some parameters, such as adjusting memory size and systolic array size. We also remove unnecessary parts that are not important in spectrum sensing, which is a simpler task compared to image processing. [15]

However, due to our limited expertise with Chisel and the specific configurations required for our use case, we often encounter compatibility issues between the accelerator and the surrounding SoC. This necessitates reconfiguring parameters and sometimes adding extra memory size that may not be needed to ensure proper functionality and integration.

The key components we configure consist of four Chisel files. In the subsequent chapter, we will provide a brief overview of how we configured these files, outlining the specific modifications made to tailor the accelerator for spectrum sensing purposes. These files are **GemminiCustomConfigs.scala**, **CPUConfigs.scala**, **SoCConfigs.scala**, and **GemminiDefaultConfigs.scala**. Figure 23 shows examples of these configuration files.

```
class CustomGemminiSoCConfig extends Config(
  new gemmini.GemminiCustomConfig ++

  // Set your custom L2 con
  new chipyard.config.M2HL

  new Freechips.RocketChip(
    nBanks = 1,
    nWays = 8,
    capacityKB = 16,
    outerLatencyCycles = 40
  ) ++

  // Set the number of CPUs
  new chipyard.CustomGemmi

  new chipyard.config.Abstr
)

val roses = defaultConfig.copy(
  // DNN options
  has_training_convs = false,
  has_max_pool = false,

  //input and accumulator size
  inputType = SInt(16.W),
  accType = SInt(32.W),

  // Spatial array PE options
  dataflow = Dataflow.WS,

  // Scratchpad and accumulator
  sp_capacity = CapacityInKilobytes(64),
  acc_capacity = CapacityInKilobytes(32),

  sp_banks = 4,
  acc_banks = 2,
)
```

Figure 23 Snapshots of chisel configuration files

### 3.4 RTL Build Process:

Once we have written our configuration files, Gemmini provides automated scripts that facilitate the conversion of these Chisel configuration files into Verilog RTL code. This conversion process involves several steps. First, the Chisel codes are transformed into FIRRTL, which we discussed in detail in the previous chapter. Subsequently, the FIRRTL Compiler, an open-source tool, translates the FIRRTL code into Verilog. [7] [9]

The resulting generated source code is located in the "generated\_src" directory within the Gemmini system. This directory comprises several automatically generated files based on the provided configurations as shown in Figure 24. The core of this generated source code consists of RTL source files written in Verilog hardware description languages. These files define the digital logic and components of the Gemmini system, including the hardware accelerator, memory interfaces, control units, and interconnects. [15]

Testbench files were also generated which are fundamental for simulating and verifying the functionality of the Gemmini system. These files contain stimulus generation modules, test cases, and simulation infrastructure, allowing us to assess the correctness and performance of the system.

There are also ".JSON" files, which stand for JavaScript Object Notation, which are used in the Gemmini system for configuration purposes. These files store system settings and specifications in a structured readable format. They have parameters such as memory sizes, data precision, and systolic array dimensions. The Gemmini

system's automated scripts process .JSON files during system generation, ensuring that the specified settings are applied to generate customized RTL code and other related components. [15]

```

root@059a4b566a55:~/chipyard/generators/gemmini/generated-src/verilator/chipyard.TestHarness.CustomGemminiSoCConfig# ls
ClockDividerN.sv
EICG_wrapper.v
IOCell.v
SimDRAM.cc
SimDRAM.v
SimDTM.cc
SimJTAG.cc
SimJTAG.v
SimSerial.cc
SimSerial.v
SimUART.cc
SimUART.v
TestHarness.anno.json
bootrom_rv32.img
bootrom_rv64.img
chipyard.TestHarness.CustomGemminiSoCConfig
chipyard.TestHarness.CustomGemminiSoCConfig.0x0.0.regmap.json
chipyard.TestHarness.CustomGemminiSoCConfig.0x0.1.regmap.json
chipyard.TestHarness.CustomGemminiSoCConfig.0x100000.0.regmap.json
chipyard.TestHarness.CustomGemminiSoCConfig.0x110000.0.regmap.json
chipyard.TestHarness.CustomGemminiSoCConfig.0x200000.0.regmap.json
chipyard.TestHarness.CustomGemminiSoCConfig.0x2010000.0.regmap.json
chipyard.TestHarness.CustomGemminiSoCConfig.0x40.0.regmap.json
chipyard.TestHarness.CustomGemminiSoCConfig.0x4000.0.regmap.json
chipyard.TestHarness.CustomGemminiSoCConfig.0x54000000.0.regmap.json
chipyard.TestHarness.CustomGemminiSoCConfig.0xc000000.0.regmap.json
chipyard.TestHarness.CustomGemminiSoCConfig.anno.json
chipyard.TestHarness.CustomGemminiSoCConfig.buildTopClockGenerator.Freq.summary
chipyard.TestHarness.CustomGemminiSoCConfig.core.config
chipyard.TestHarness.CustomGemminiSoCConfig.d
chipyard.TestHarness.CustomGemminiSoCConfig.dromajo_params.h
chipyard.TestHarness.CustomGemminiSoCConfig.dts
chipyard.TestHarness.CustomGemminiSoCConfig.fir
chipyard.TestHarness.CustomGemminiSoCConfig.graphml
chipyard.TestHarness.CustomGemminiSoCConfig.graphml
chipyard.TestHarness.CustomGemminiSoCConfig.harness.anno.json
chipyard.TestHarness.CustomGemminiSoCConfig.harness.fir
chipyard.TestHarness.CustomGemminiSoCConfig.harness.mems.conf
chipyard.TestHarness.CustomGemminiSoCConfig.harness.mems.v
chipyard.TestHarness.CustomGemminiSoCConfig.harness.v
chipyard.TestHarness.CustomGemminiSoCConfig.harnessDividerOnlyClockGenerator.Freq.summary
chipyard.TestHarness.CustomGemminiSoCConfig.json
chipyard.TestHarness.CustomGemminiSoCConfig.l2.json
chipyard.TestHarness.CustomGemminiSoCConfig.memmap.json
chipyard.TestHarness.CustomGemminiSoCConfig.plusArgs
chipyard.TestHarness.CustomGemminiSoCConfig.rom.conf
chipyard.TestHarness.CustomGemminiSoCConfig.top.anno.json
chipyard.TestHarness.CustomGemminiSoCConfig.top.fir
chipyard.TestHarness.CustomGemminiSoCConfig.top.fir
emulator.cc
firrtl_black_box_resource_files.harness.f
firrtl_black_box_resource_files.top.f
mm.cc
mm.h
mm_dramsim2.cc
mm_dramsim2.h
plusarg_reader.v
remote_bitbang.v
remote_bitbang.h
sim_files.common.f
sim_files.f
testchip_tsi.cc
testchip_tsi.h
uart.cc
uart.h
verilator.h
    
```

Verilog RTL

Figure 24 Snapshot of output generated files from Gemmini

### 3.5 Software RTL Simulations:

After generating the RTL code for the accelerator, it is crucial to ensure its proper functionality. This involves testing and evaluating its performance using cycle-accurate simulators. The gemmini full-system visibility SoC provides a convenient solution for this purpose.



Figure 25 Gemmini different levels of visibility.

The software code compilation flow for the Gemmini hardware accelerator offers multiple levels of control and flexibility as shown in Figure 26. At the lowest level, we have the option to engage in low-level programming using assembly language or direct machine configuration. This level of programming grants the highest level of control, allowing for precise manipulation and configuration of the Gemmini hardware accelerator. It provides a

deeper understanding of the underlying hardware and enables us to squeeze out every bit of performance by carefully crafting code. [10] [14]

For more fine-grained control and optimization, the mid-level programming option provides access to kernel libraries. We can leverage these libraries to optimize the performance of specific operations, such as matrix multiplication (GEMM). This level of programming enables customization and tuning of the Gemmini accelerator to achieve optimal results for specific tasks or workloads.

At the highest level, we can utilize high-level programming languages to compile their ONNX models, providing a convenient and abstract approach. This level of programming allows for easy integration and utilization of the Gemmini accelerator without delving into lower-level details.

### **3.5.1 Low-Level tests:**

During the development and testing phase of the Gemmini hardware accelerator, we started with the lowest-level templated tests with minor modifications. These tests are designed to assess the functionality and performance of individual components within Gemmini, such as matrix multiplication, tiled matrix multiplication, and specific workloads like Mobilenet. [15]

The templated tests provide a foundation for evaluating the behavior of Gemmini at a low level, allowing us to validate the functionality of each component and verify that they are functioning as intended. By starting with these tests, we can ensure that the fundamental building blocks of the accelerator are working correctly before moving on to more complex and comprehensive evaluations.

There are tools for testing Gemmini like SPIKE and Verilator. SPIKE is a functional RISC-V ISA (Instruction Set Architecture) simulator that emulates the behavior of a RISC-V processor. It enables us to execute programs and observe the expected behavior without the need for physical hardware. Verilator, on the other hand, is a cycle-accurate Verilog simulator. It allows us to simulate and analyze the behavior of Gemmini's hardware design at a more detailed level, including the timing and interactions between different components. [7] [15]

### **3.5.2 Gemmini Flow Automation Script:**

During our exploration and experimentation with Gemmini, we encountered failures in the tests, as we mentioned earlier. These failures were primarily attributed to our limited experience with Chisel and the complexities of configuring the system. As a result, reverting to default configurations took a considerable amount of time. To streamline this process, we developed an automated script that facilitates the generation and testing of Gemmini. Whenever a test fails, we simply modify the configuration files and rerun the script. This iterative approach allows us to make multiple attempts with different configurations until we successfully pass all the tests it is shown in Figure 26. The details of the final configurations adopted are discussed briefly in the subsequent chapter.

```

=====
Gemmini extension configured with:
  dim = 16
Flush Gemmini TLB of stale virtual addresses
Initialize our input and output matrices in main memory
Calculate the scratchpad addresses of all our matrices
  Note: The scratchpad is "row-addressed", where each address contains one matrix row
Move "In" matrix from main memory into Gemmini's scratchpad
Move "Identity" matrix from main memory into Gemmini's scratchpad
Multiply "In" matrix with "Identity" matrix with a bias of 0
Move "Out" matrix from Gemmini's scratchpad into main memory
Fence till Gemmini completes all memory operations
Check whether "In" and "Out" matrices are identical
Input and output matrices are identical, as expected
=====

Prediction: 75 (score: 127)
Prediction: 900 (score: 127)
Prediction: 125 (score: 103)
Prediction: 897 (score: 98)

Total cycles: 5665836 (100%)
Matmul cycles: 347757 (6%)
Im2col cycles: 0 (0%)
Conv cycles: 86663 (1%)
Pooling cycles: 0 (0%)
Depthwise convolution cycles: 3356976
Res add cycles: 394717 (6%)
Other cycles: 1479723 (26%)
PASS

=====
Gemmini extension configured with:
  dim = 16
MAT_DIM_I: 64
MAT_DIM_J: 64
MAT_DIM_K: 64
Starting slow CPU matmul
Cycles taken: 2130388
Starting gemmini matmul
Cycles taken: 94
=====

=====
Flush Gemmini TLB of stale virtual addresses
Initialize our input and output matrices in main memory
Calculate the scratchpad addresses of all our matrices
  Note: The scratchpad is "row-addressed", where each address contains one matrix row
Move "In" matrix from main memory into Gemmini's scratchpad
Move "Identity" matrix from main memory into Gemmini's scratchpad
Multiply "In" matrix with "Identity" matrix with a bias of 0
Move "Out" matrix from Gemmini's scratchpad into main memory
Fence till Gemmini completes all memory operations
Check whether "In" and "Out" matrices are identical
Input and output matrices are identical, as expected
[UART] UART0 is here (stdin/stdout).
All commands completed
=====

```




Figure 26 Output of the final configuration Low level tests

### 3.5.3 High-level test:

After iterating through multiple configurations and tests, we arrived at our final configuration for Gemmini. To validate the performance and functionality of the accelerator at a higher level, we conducted tests using the ONNX runtime with a quantized model of ResNet-50 as our test case. We executed the test and successfully passed it, demonstrating the compatibility of Gemmini with real-world workloads. However, it's worth noting that achieving the desired results requires many clock cycles.

During our experimentation with the Gemmini accelerator, we encountered challenges when utilizing the ONNX Runtime for the modulation recognition CNN (Convolutional Neural Network) model. While we successfully employed the ONNX Runtime with the ResNet 50 test case, we faced difficulties with the MR CNN model. The MR CNN model requires input in the form of a 2x128 dimension array, which differed from the image runner compiler used in the ResNet 50 test. As a result, we were limited to testing the Gemmini accelerator exclusively with the ResNet 50 quantized model. Next table illustrates the outcomes of running inference on three images using our configuration.

Table 2 the outcomes of running inference on three images using our configuration for ResNet50

<pre>Element count 1000. Top 5 classes: 0.031456 giant schnauzer 0.075702 curly-coated retriever 0.087432 Great Dane 0.271946 Labrador retriever 0.361813 Rottweiler Done! Inference took 297272646 cycles</pre>	
<pre>Element count 1000. Top 5 classes: 0.049711 orange 0.050172 paintbrush 0.051713 bucket, pail 0.251607 tennis ball 0.303495 lemon Done! Inference took 297273936 cycles</pre>	
<pre>Called into systolic add Element count 1000. Top 5 classes: 0.036013 space bar 0.038994 desktop computer 0.155010 notebook, notebook computer 0.204533 laptop, laptop computer 0.329461 printer Done! Inference took 297275247 cycles</pre>	

During our evaluation, we observed that the accuracy of the model utilized in the Gemmini accelerator fell short of our expectations. It's important to note that the diminished performance can be attributed to the quantization of the ResNet 50 model itself, rather than any hardware limitations. Quantization involves reducing the precision of numerical values in the model to enhance computational efficiency. However, this process can lead to a decrease in model accuracy. Thus, the performance we experienced can be attributed to the inherent characteristics of the deep learning model rather than any shortcomings in the Gemmini hardware accelerator. Moving forward, further optimizations or alternative models may be explored to address the accuracy concerns.

# 4 Gemini Framework

## 4.1 Background

### 4.1.1 Converting Convolution to General Matrix Multiplication (GEMM)

Convolution is the main operation of deep learning and what it does is that it extracts features from the training data [16] [17],The convolution operation can be expressed as a dot product between the input and the kernel, figure 27a shows the direct convolution of an input matrix A with height  $i_h$  and width  $i_w$  and a kernel of height  $k_h$  and width  $k_w$  and the output matrix C of height  $o_h$  width  $o_w$ , C matrix is produced by sliding the kernel B over matrix A with a parameter called stride and applying matrix multiplication.

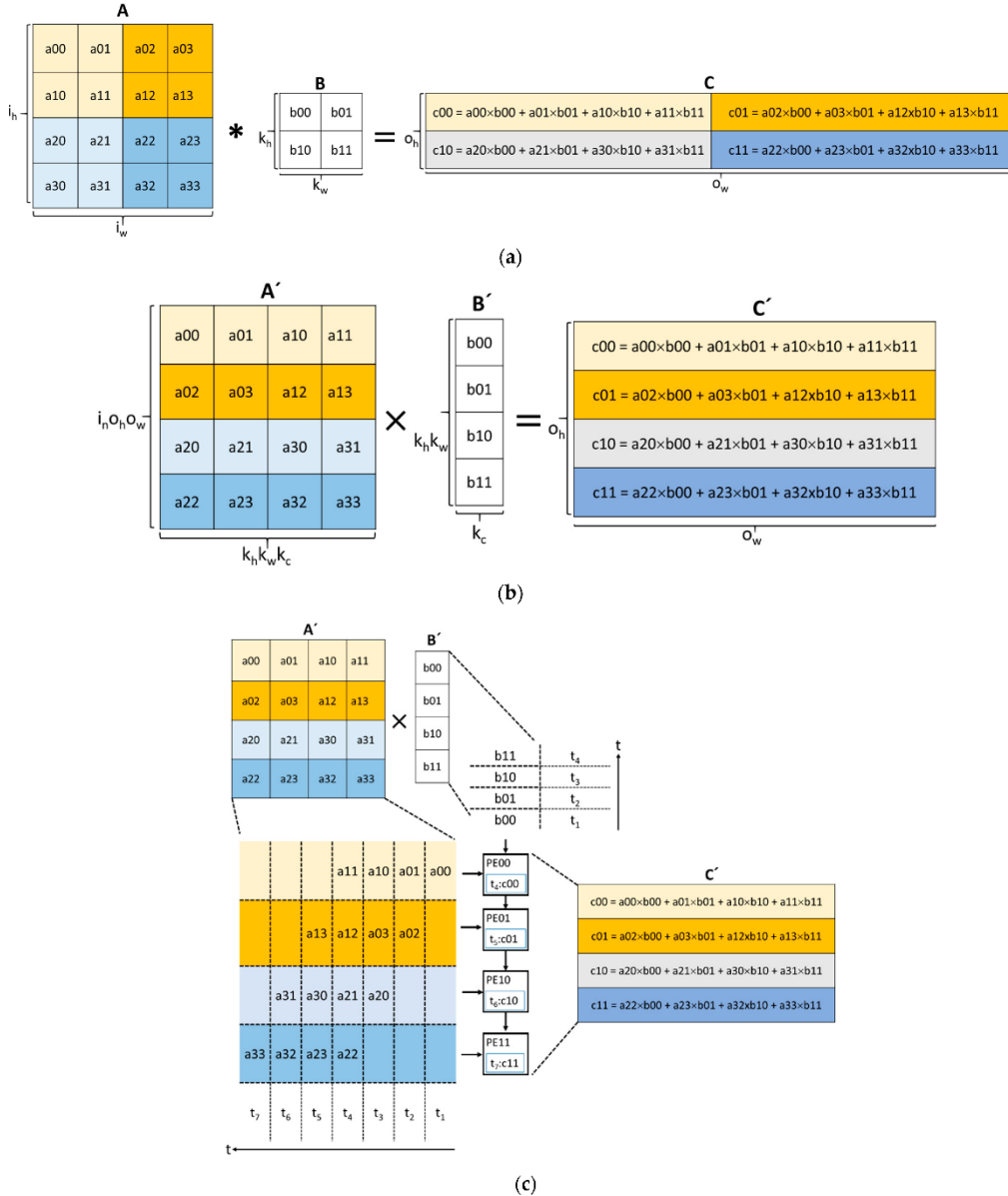


Figure 27 (a) Direct convolution. (b)  $im2col$ -based GEMM operation (C) Systolic array operation

For more efficient operations the inputs and kernels are lowered into matrices using im2col operation and applying GEMM operation [18] [19], as shown in figure 27b the matrices A and B are transformed into A' and B' respectively using the im2col method, Each submatrix of A with a height of  $k_h$  and width of  $k_w$  is stretched into a row of A'. A' ends up with a dimension of  $i_n o_h o_w \times k_h k_w k_c$ , where  $i_n$  is the total number of inputs and  $k_c$  is the number of kernel channels. The im2col method is capable of identifying the overlapping data among the rows of A'. The transformed kernel B' is also generated by stretching B from Figure 27a into a column. The im2col transformation enables the computation of the output matrix C' with dimensions  $o_h$  and  $o_w$  by performing a matrix multiplication between matrices A' and B'. By leveraging optimized primitives or hardware accelerators, the im2col-GEMM approach can efficiently execute this operation.

The im2col approach led to a lot of innovations in hardware accelerators one of them is the use of the Systolic Array (SA) [20], SA was designed in 1979 but it grabbed the attention again when Google used it in Google Tensor Processing Unit (TPU) to accelerate GEMM operations [21], an SA is made of mesh of processing elements (PEs) which includes Multiply And Accumulate (MAC) unit which architecture is shown in figure 28. PEs communicate with neighbors to send input data and compute data from MACs, each clock cycle data are loaded from memory to the PEs that they are connected to, then to the neighbor PEs then to the last PEs then back to the memory to save the results. Figure 27c shows the operation of 4\*1 SA used to multiply A' and B' to produce C', A' is fed to the SA from the left and B' is fed to the SA from the top in each clock cycle. Each PE computes the MAC operation until the last input value is received. The final generation of individual elements of matrix C' starts and ends in cycle  $t_4$  and cycle  $t_7$ , respectively. The SA method implements two main dataflows, which include OS and WS [22].

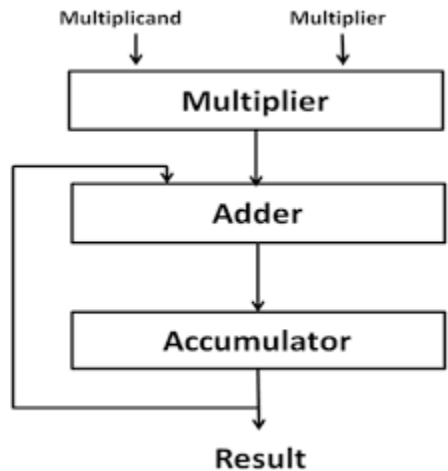


Figure 28 MAC architecture

The dataflow scheme illustrates how the processing elements (PEs) communicate with each other and how data is accessed and transferred between memory and the PEs. Two different dataflow schemes, OS and WS, are described.

In the OS (Output Stationary) dataflow scheme, the partial sums of C' remain within the PEs until the final output is computed. This design minimizes the movement of partial sums between the PEs and memory. The PEs communicate with each other to perform the necessary computations, reducing the need for data transfers to and from memory.

On the other hand, the WS (Weight Stationary) dataflow scheme involves pre-loading the kernel matrix B' into the appropriate PEs before the computations start. This allows for the reuse of the kernel matrix without accessing memory repeatedly. By storing the necessary data within the PEs, the WS dataflow minimizes memory access and enhances overall efficiency. The Gemmini accelerator framework, which is the focus of this work,

utilizes the SA method to configure and accelerate General Matrix Multiplication (GEMM) operations of various dimensions. The SA method leverages the dataflow schemes, OS and WS, to optimize the computation and minimize data movement, resulting in improved performance and efficiency for GEMM operations.

### 4.1.2 Gemmini framework

The name is not an abbreviation of anything but it may be a combination of the two words Gemini and GEMM, the Gemmini accelerator frame work has been around since 2019, it is capable of generating a full System on Chip (SoC) accelerator including peripherals, memories, processors and their interconnections, from the beginning the researches have worked on improving and enhancing some components of the architecture

## 4.2 Gemmini architecture

### 4.2.1 Introduction

Gemmini is a part of the open source RSIC-V chipyard framework [7], The architecture is always under development, the language used in the framework is Chisel which is a new Hardware Description Language (HDL) designed to generate configurable hardware design with parameters, Chisel is embedded in Scala to provide high level object oriented programming and functional primitives, Then the Chisel HDL is compiled to generate Verilog HDL code and since, it does not only generate the SA but it generated a full SoC for simulations and implementations [23]

Gemmini accelerator can be generated alone or in a full SoC as shown in figure 29, the Gemmini accelerator is based on non-standard RSIC-V instruction set architecture so it needs a rocket processor to communicate with to receive commands and send results, the framework gives the ability to generated a rocket processor on chip with Gemmini accelerator or only the Gemmini accelerator to be integrated later into a system that already has a RSIC-V processor.

We need to understand the architecture of the Gemmini accelerator to be able to read the configurations and configure our accelerator to our deep learning model.

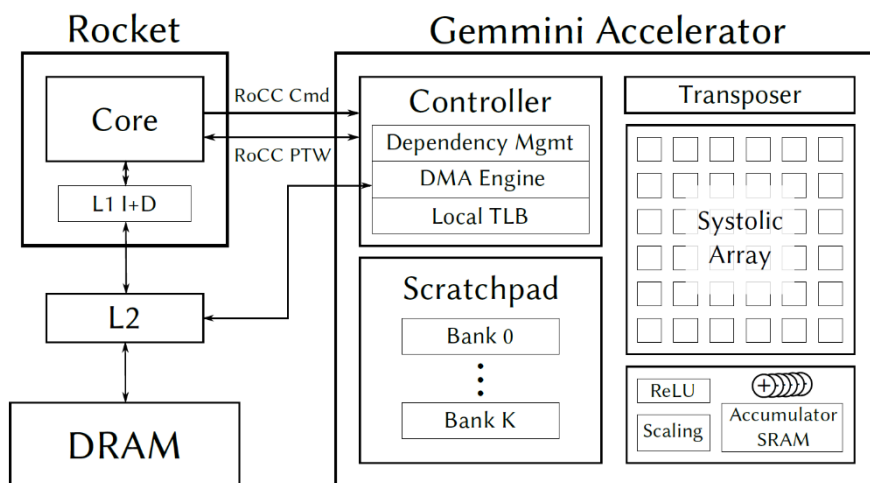


Figure 29 Full SoC architecture



In the next section we describe the architecture of the standalone Gemmini accelerator shown in figure 30.

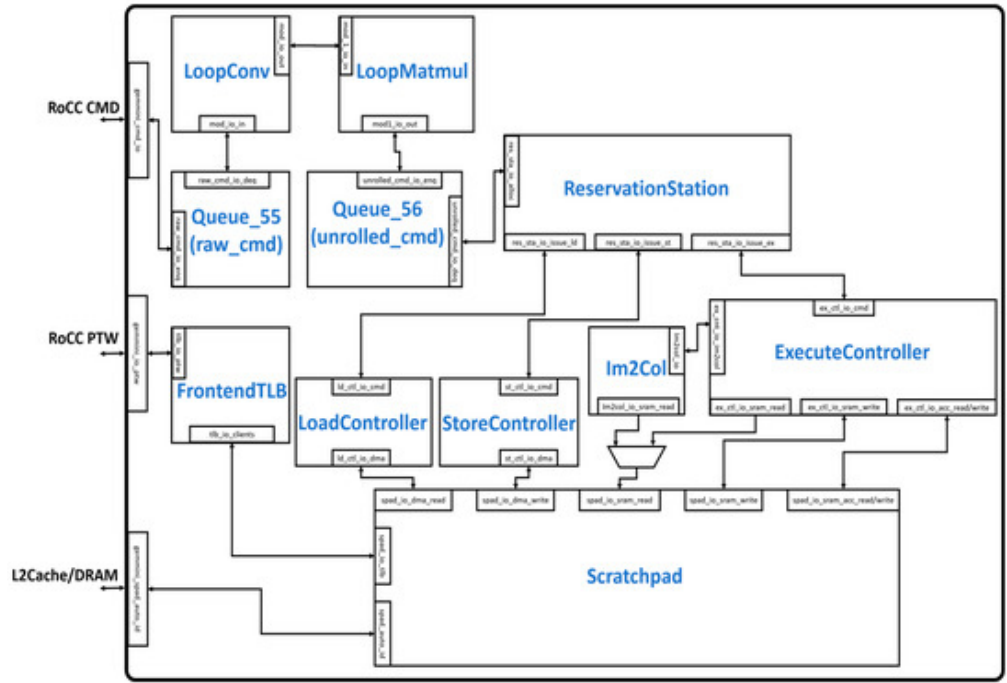


Figure 30 Gemmini accelerator architecture

### 4.2.2 RoCC interface

The interface is designed to enable decoupled communication between the core processor and the Gemmini accelerator [24], as shown in figure 31, The RoCC interface consists of command (RoCCCommand) and response (RoCCResponse), the core processor sends commands and related data to Gemmini accelerator through the RoCCCommand and receives the results from accelerator through RoCCResponse interface.

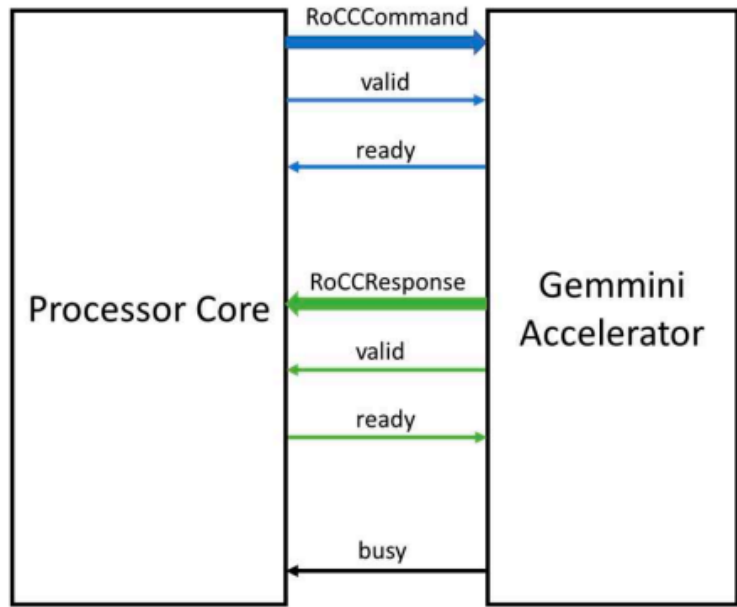


Figure 31 RoCC interface signals

### 4.2.3 LoopConv and LoopMatMul modules

The LoopConv and LoopMatMul unroll the large convolutions and matrix multiplications to fit the systolic array dimensions.

As shown in figure 32.a LoopConv consists of modules such as LoopConvLdBias, LoopConvLdWeight, and LoopConvLdInput. Which are responsible for loading the main inputs of a machine learning models (inputs, bias, weights). The LoopConvExecute and LoopConvst modules are responsible for executing the convolution operations and storing the results respectively.

As shown in figure 32.b LoopMatMul consists of modules such as LoopMatmulLdA, LoopMatmulLdB, and LoopMatmulLdD which are responsible for loading the matrices A, B and D respectively. LoopMatmulExecute is responsible for generating commands for the matrix multiplications. LoopMatmulStC is responsible for storing the result matrix C.

The Queue\_56 module in both LoopConv and LoopMatmul stores the input commands in a buffer.

The unrolled commands are then sent to Reservation station module.

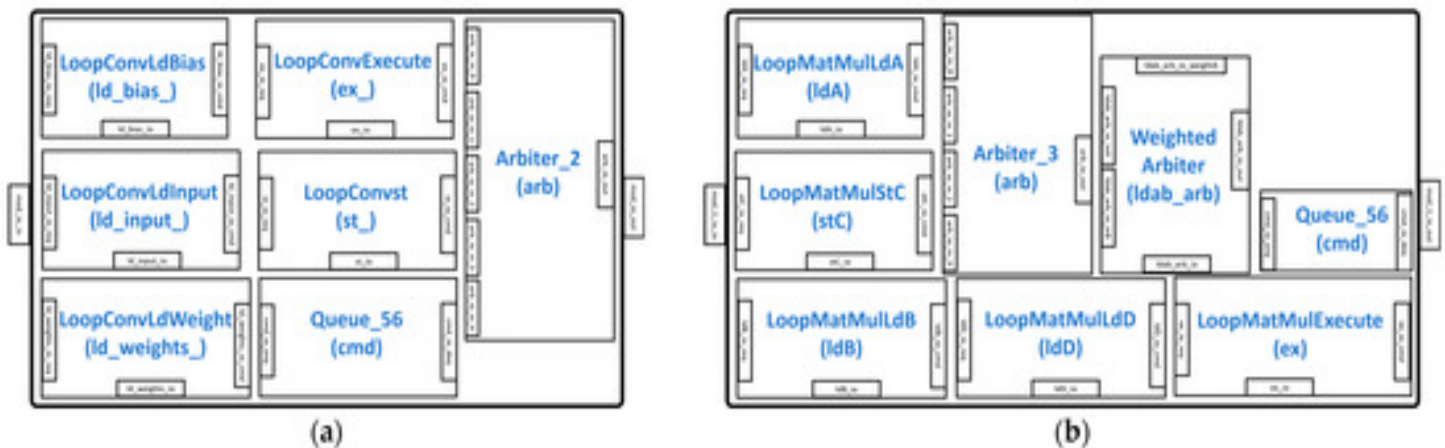


Figure 32 Command unrolling modules

### 4.2.4 LoadController and StoreController

Load and store controllers are similar in design and they are responsible for data movement instructions from DRAM to the scratchpad module in Gemmini accelerator and from the scratchpad in the Gemmini accelerator to DRAM respectively. Both modules use a finite state machine. The finite state machine has three states *waiting\_for\_cmd*, *sending\_rows*, and *waiting\_for\_dma\_req\_ready*. In the *waiting\_for\_cmd* state, the controllers configure themselves and load the commands from the reservations station module into the DMACommandTracker and DMACommandTracker\_1 for load and store controllers respectively, they also keep track of total amount of data to be loaded the scratchpad and stored in the DRAM, respectively. In the *waiting\_for\_req\_ready* state. The controllers wait until the DMA configure itself to store and read data. In the *sending\_row* state, the controllers keep working until they load all the data needed which we know the size of needed data from the command trackers.

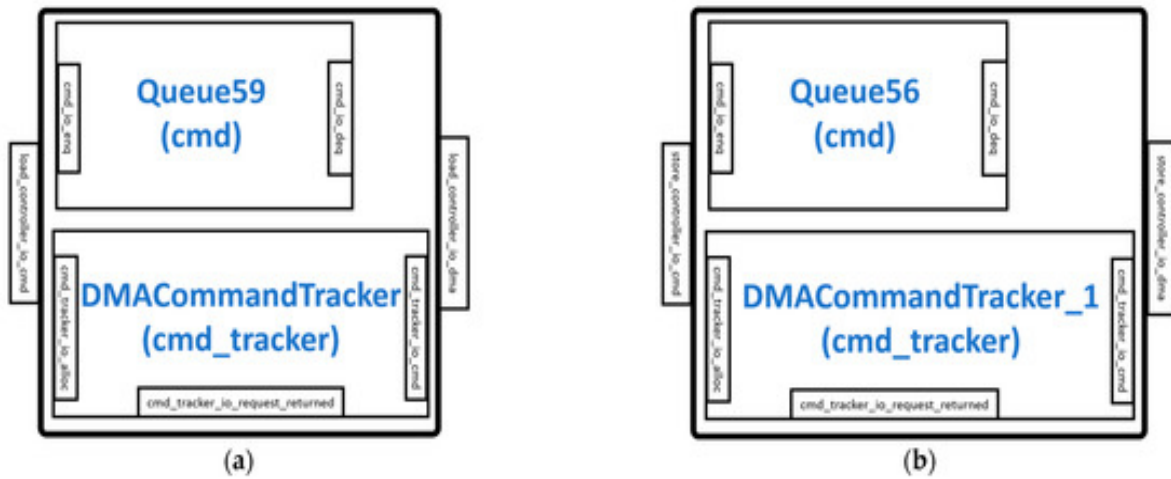


Figure 33 DMA engine

#### 4.2.5 ExecuteController and image to column (im2col)

ExecuteController illustrated in figure 34b and im2col illustrated in figure 34a both modules connect to the SRAMs, im2col module can be generated as a hardware module or the operation can be handled by the core processor.

Im2col module reads instructions from execute controller and data from SRAM interface of the scratchpad module, the module makes sure that every PE is used as much as possible, it operates depending on a finite state machine with states *nothing\_to\_do*, *waiting\_for\_im2col*, *preparing\_im2col*, and *im2col\_done*.

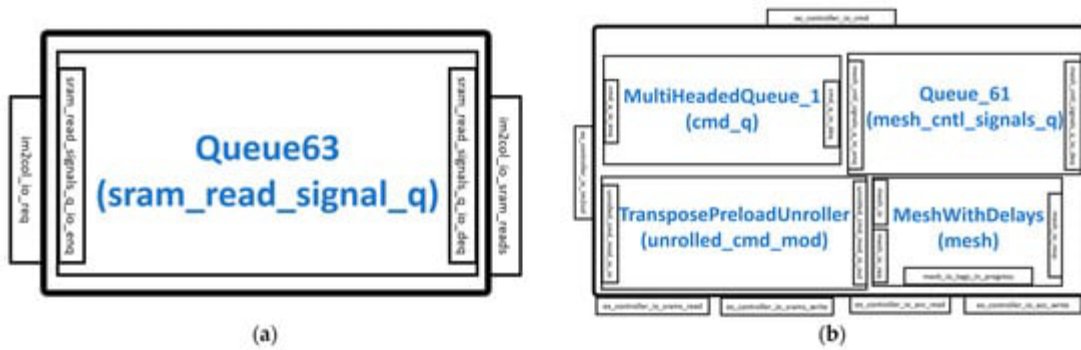


Figure 34 SRAM controllers (a) im2col module (b)ExecuteController

The execute controller connects to im2col and SRAM interfaces and reservation station, the received command from reservation station is unrolled by TransposePreloadUnroller module and stored in the MultiHeadedQueue\_1 module. MultiHeadedQueue\_1 module is a buffer and it classifies commands to execute instructions which are then buffered in Queue\_61 module then sent to MeshWithDelays module illustrated in figure 35, and memory instructions to read data from the SRAMs of the scratchpad module.

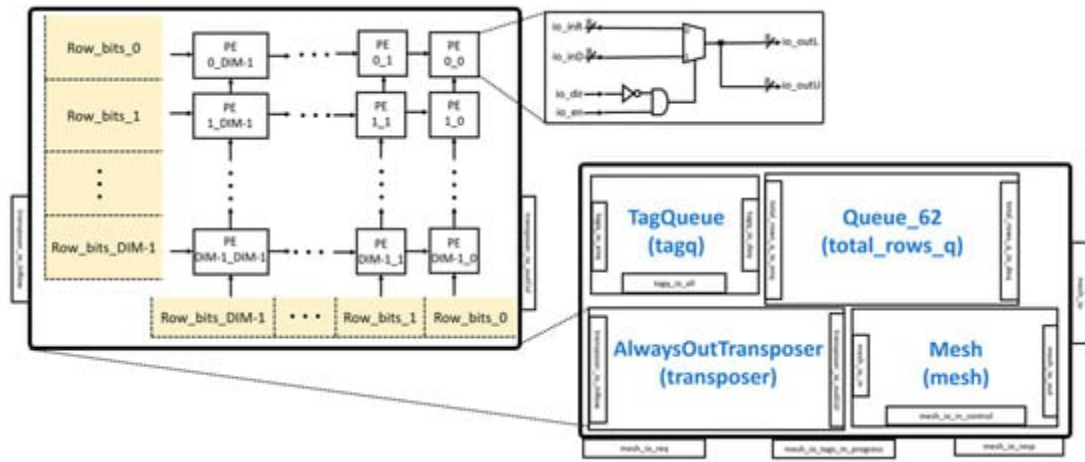


Figure 35 MeshWithDelays module

MeshWithDelays module consists of a mesh and AlwaysOutTransposer, AlwaysOutTransposer is responsible for transposing the matrix before it enters the mesh, It is implemented as a configurable systolic array.

Mesh module is illustrated in figure 36, it consists of configurable number of rows and columns of PEs, there are registers to make sure all input matrices arrive at the same time to the PEs before starting the multiplication operation, one of the configurations that we might have multiple meshes each mesh contains one tile that is made of PEs, so we conclude that PEs are the most important part of the mesh.

PEs in Gemmini can have three data flows implementations WS, OS and both, we choose what to have in the configuration part. Also illustrated in figure 36.

As discussed before Mesh module takes the inputs from AlwaysOutTransposer unit and sends the results to the SRAM of accumulator in scratchpad.

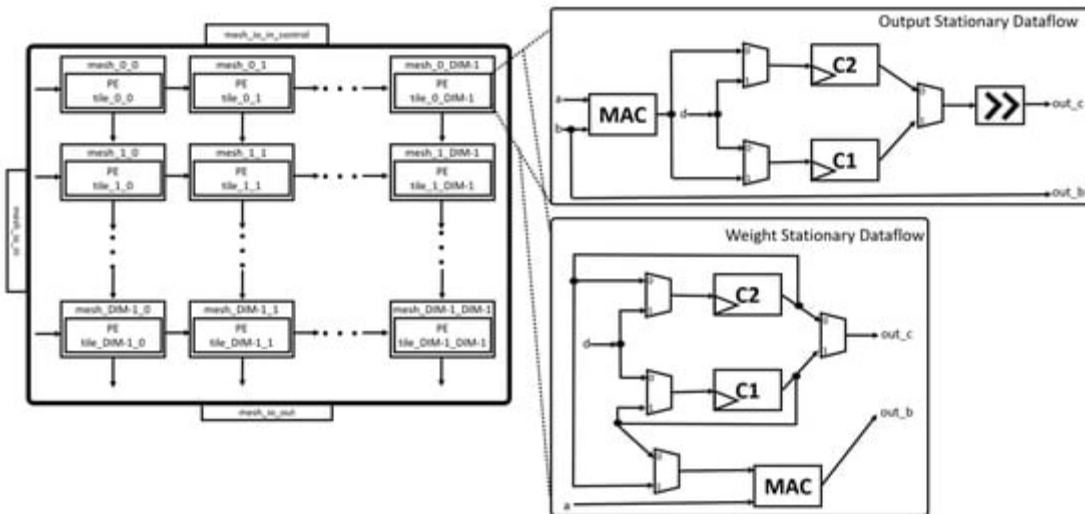


Figure 36 MeshWithDelays module

## 4.2.6 ScratchPad

It is responsible for storing the output data of the matrices convolution and multiplication. Figure 37 illustrates the structure of the scratchpad module. It contains multiple pipelines to pipeline DMA and ExecuteController read operation. StreamWriter module is responsible for storing the data before sending it to the DRAM, DMA control signals interacts with StreamWriter and StreamReader to read or read their contents.

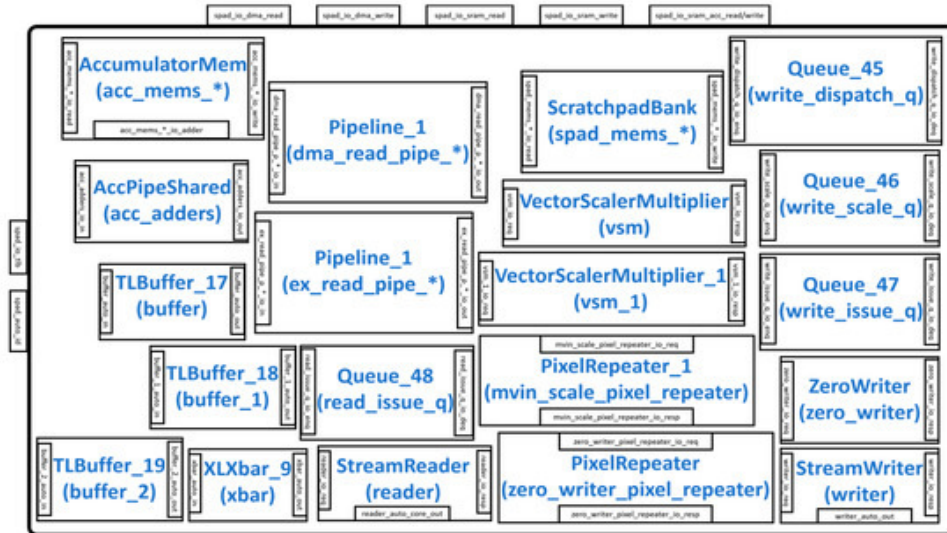


Figure 37 Scratchpad module

Figure 38a illustrates the structure of the scratchpadBanks in the scratchpad module. The number of banks that are SRAM banks and their sizes are configurable (The default number of banks is four banks), while multiple banks enables concurrent memory access. They are responsible for storing the input and output matrices A, B, D and C.

Figure 38b illustrates the structure of the AccumulatorMem in the scratchpad module. The number of banks that are SRAM banks and their sizes are configurable (The default number of banks is two banks), the AccumulatorMem holds the intermediate and final results [10].

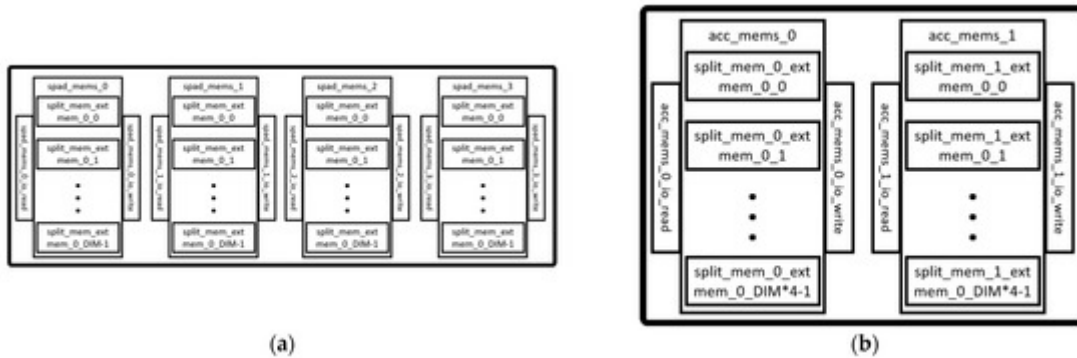


Figure 38 (a) ScratchpadBanks (b) AccumulatorMem

## 4.3 Configurations

### 4.3.1 Introduction

As the goal is to achieve higher performance configurations play a crucial rule to achieve that goal, In this section we will discuss the available configurations and the configurations we chose whether we chose to stick with the default configurations or chose to change it to fit our goal.

### 4.3.2 Configurable parameters

#### 4.3.2.1 Spatial Array parameters

As discussed before spatial array can has multiple tiles while each tile has multiple number of PEs, the default configuration is to have 1 tile and a mesh of 16x16 PEs.

The dataflow options are weight stationary and output stationary and to have both, the default configuration is to have both.

#### 4.3.2.2 Data types

Data types in Gemmini can be fixed point or floating point, we can configure the input data type, the accumulator data type and the spatial array data type, the default is the input data type is an 8-bit signed integer, the accumulator data type is 32-bit signed integer and the spatial array data type to be 20-bit signed integer.

#### 4.3.2.3 Scratchpad and accumulator

Gemmini gives you the ability to choose the size of the scratchpad memory and accumulator memory and to divide each one of them into banks to enable concurrent access, the default configuration for the scratchpad memory is to be 256kB divided to 4 banks, and for the accumulator the default is to have 64kB memory divided to 2 banks.

#### 4.3.2.4 Deep neural network options

As Gemmini accelerator is a deep learning accelerator it give the ability to generate hardware modules to do some of the common functionalities of the deep learning, it can be configured to be able to learn and change the weight of the neural network in the run time or to only do the inference, can be configured to have max pool operation and nonlinear activation (ex: RELU), the default is to have them all.

### 4.3.3 Implemented configurations

In the spatial array we chose to stick with the default configuration as it fits our purpose the only change is in the dataflow type, we chose to have only weight stationary as it have been proved to be faster and execute the operation in less clock cycles than output stationary in small designs [10].

In the data types we chose the input to be 16-bit signed integer as of the deep learning model we are trying to run, and the accumulator input type and the spatial array output type we chose to stick with default configurations.

The scratch pad memory size we chose it to be 64kB and to be divided into 4 banks as the deep learning model we are trying to run is relatively small to the normal deep learning models, the accumulator memory size is 32kB and divided into 2 banks.

In the deep learning options as the goal of the deep learning model we are trying is to only do the inference we chose the option that the accelerator does not learn in the run time, as the deep learning model we are trying to run does not have a maxpool function we chose not to have it, the deep learning model we are trying to run have a RELU function so we chose to implement it.

We chose to have a hardware module to do the image to column operation.

## 5 Design flow

We use Design Compiler for logic synthesis, which is the process of converting a design description written in a hardware description language such as Verilog or VHDL into an optimized gate-level netlist mapped to a specific technology library. The steps in the synthesis process are as follows:

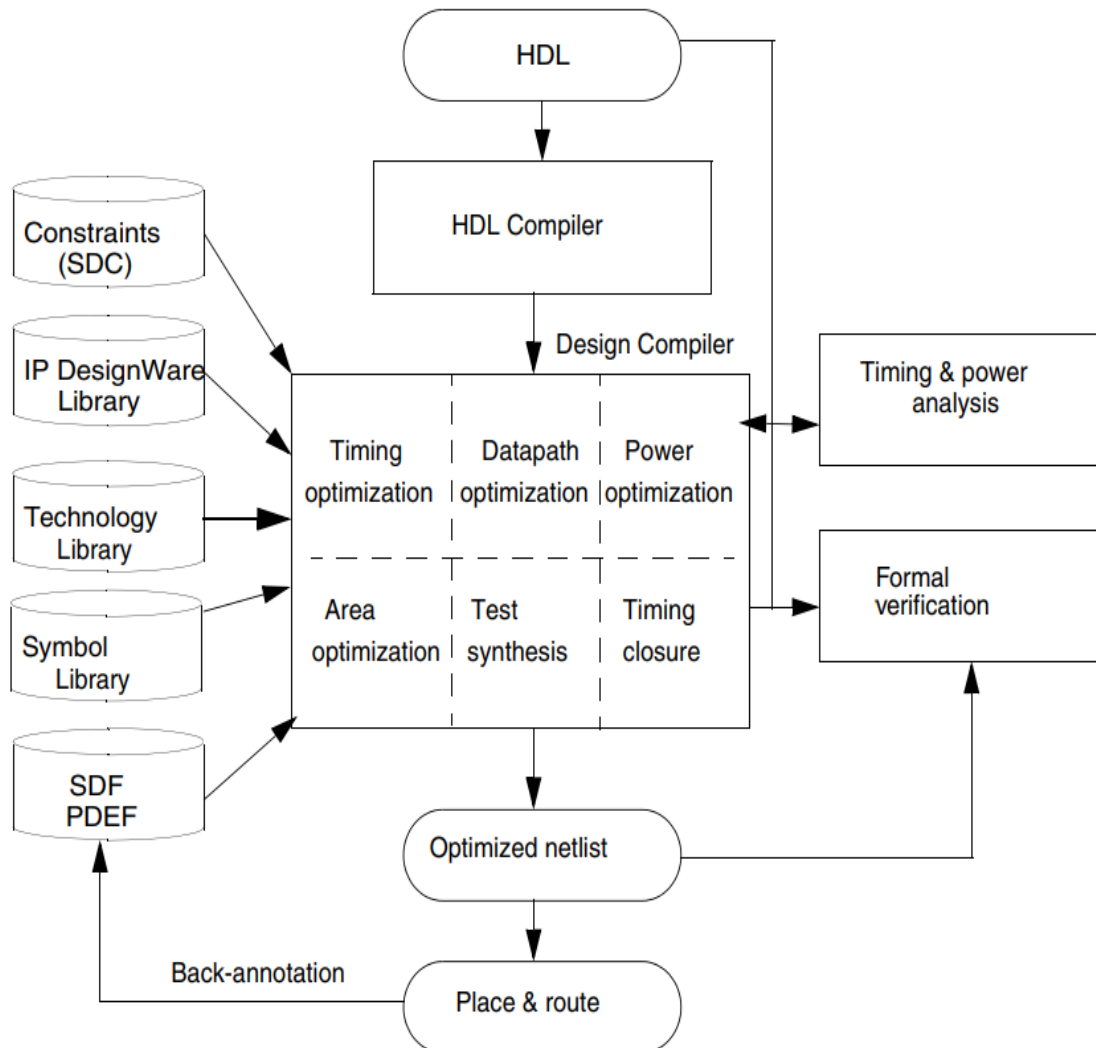


Figure 39 Overview of how Design Compiler fits into the design flow

1. The input design files for Design Compiler are often written using a hardware description language (HDL) such as Verilog or VHDL.
2. Design Compiler uses technology libraries, synthetic or DesignWare libraries, and symbol libraries to implement synthesis and to display synthesis results graphically. During the synthesis process, Design Compiler translates the HDL description to components extracted from the generic technology (GTECH) library and DesignWare library. The GTECH library consists of basic logic gates and flip-flops. The DesignWare library contains more complex cells such as adders and comparators. Both the GTECH and DesignWare libraries are technology independent, that is, they are not mapped to a specific technology library. Design Compiler uses the symbol library to generate the design schematic [25].



3. After translating the HDL description to gates, Design Compiler optimizes and maps the design to a specific technology library, known as the target library. The process is constraint driven. Constraints are the designer's specification of timing and environmental restrictions under which synthesis is to be performed.
4. After the design is optimized, it is ready for test synthesis. Test synthesis is the process by which designers can integrate test logic into a design during logic synthesis. Test synthesis enables designers to ensure that a design is testable and resolve any test issues early in the design cycle. The result of the logic synthesis process is an optimized gate-level netlist, which is a list of circuit elements and their interconnections.
5. After test synthesis, the design is ready for the place and route tools, which place and interconnect cells in the design. Based on the physical routing, the designer can back-annotate the design with actual interconnect delays; Design Compiler can then resynthesize the design for more accurate timing analysis.

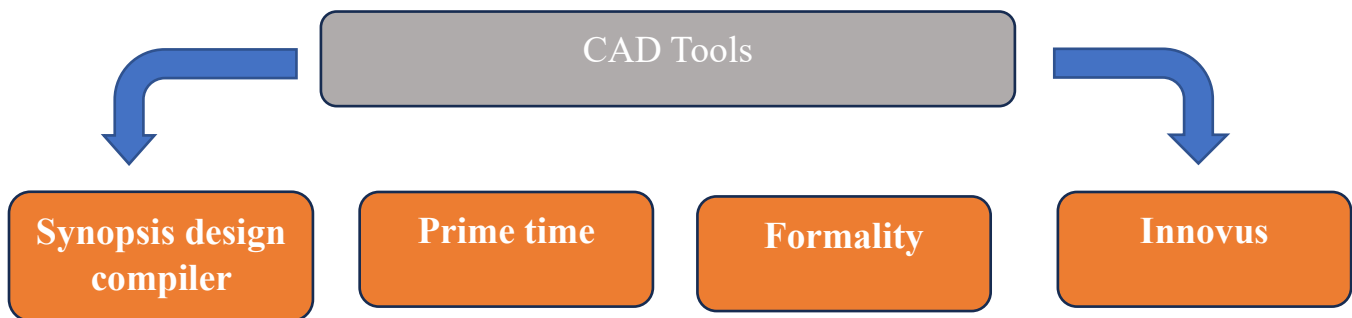


Figure 40 (computer-aided design) CAD tools

Synthesis tool > Design compiler from Synopsis

STA tool > Prime time from Synopsis

Formal Verification tool > Formality from Synopsis

PnR tool > Innovus from Cadence

## 5.1 Synthesis flow using design compiler:

As shown in Figure 41.:

### 5.1.1 Develop HDL files

- First, we extract Verilog files from Gemini (design compiler accept RTL written in VHDL and Verilog as a HDLs)

- Understand the nature of RTL extracted from Gemini and how arrange files to input them as source files to Design compiler compiler

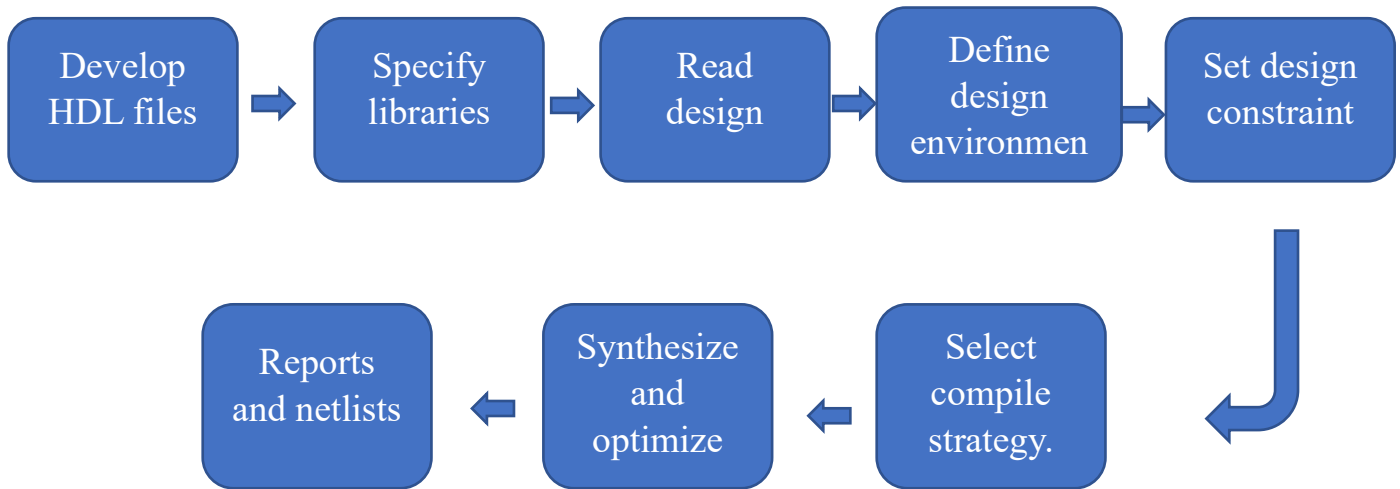


Figure 41 the Synthesis flow using DC tool

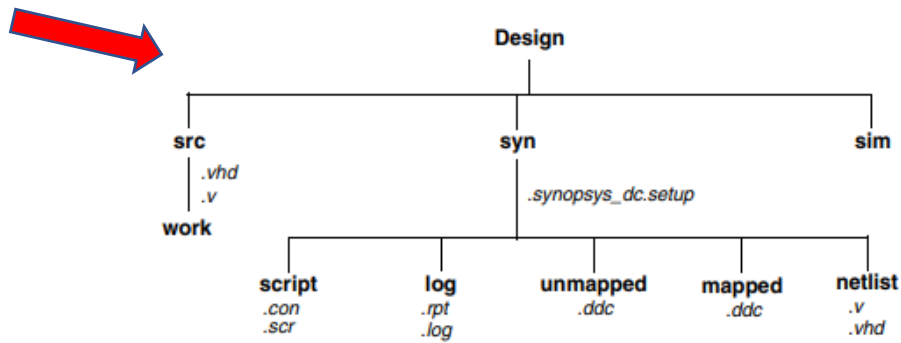


Figure 42 Design compiler take VHDL and Verilog files only.

### 5.1.2 Specify libraries (PDKS)

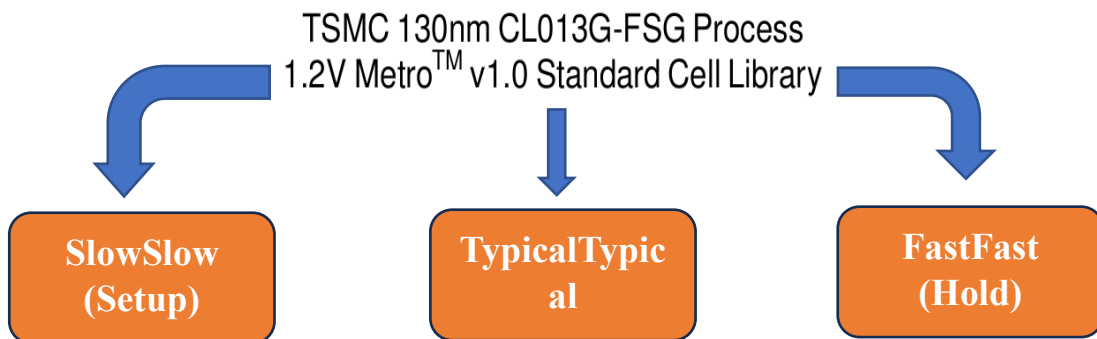


Figure 43 TSMC PDK and its PVT flavours

we specify the link, targets and synthetic libraries for Design Compiler by using the link\_library, target\_library and synthetic\_library commands.

The link and target libraries are technology libraries that define the semiconductor vendor's set of cells and related information, such as cell names, cell pin names, delay arcs, pin loading, design rules, and operating conditions.

The synthetic library is the DesignWare libraries where you take the adder, subtractor, multiplier and divisor and that which make us able to write the arithmetic symbols like (+,-,\*,/) in our RTL code [25]

### 5.1.3 Read Design

Design Compiler provides the following ways to read design files:

- The analyze and elaborate commands
- The read\_file command (we used this way)

Our design consists of 11 submodules, and we make synthesis to every submodule separately as the code of the RTL was so big and that led to full of our laptops rams the design compiler crashes we will discuss this problem later in details inshallah.

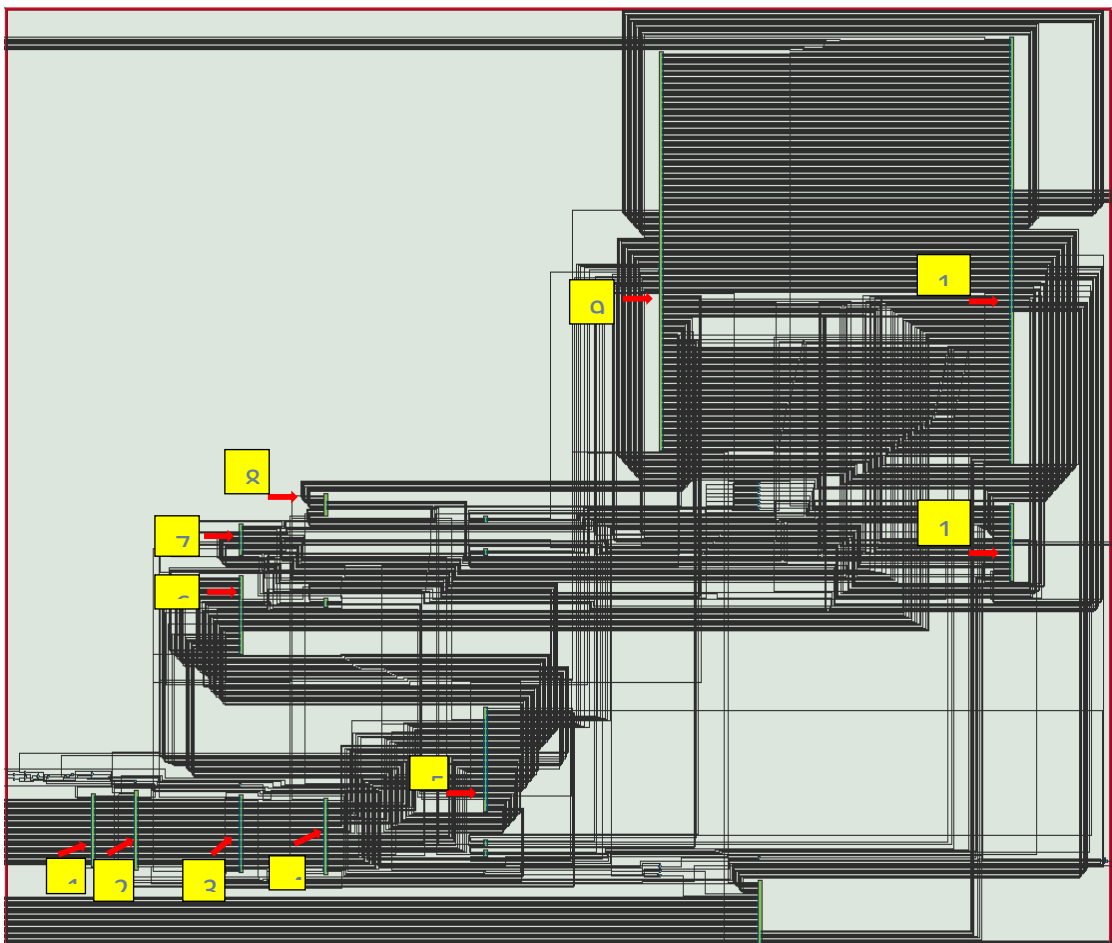


Figure 44 Structure of design

## 5.1.4 Define design environment

Design Compiler requires you to model the environment of the design to be synthesized. This model comprises the external operating conditions (manufacturing process, temperature, and voltage), loads, drives, fanouts, and wire load models. It directly influences design synthesis and optimization result.

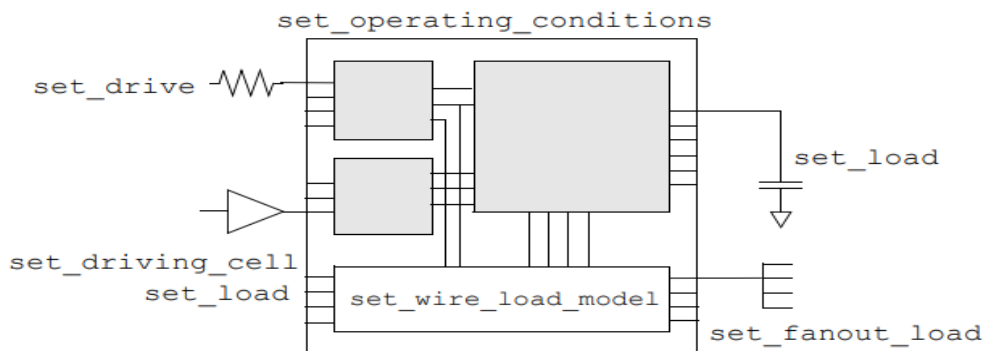


Figure 45 Design environment to estimate the reality

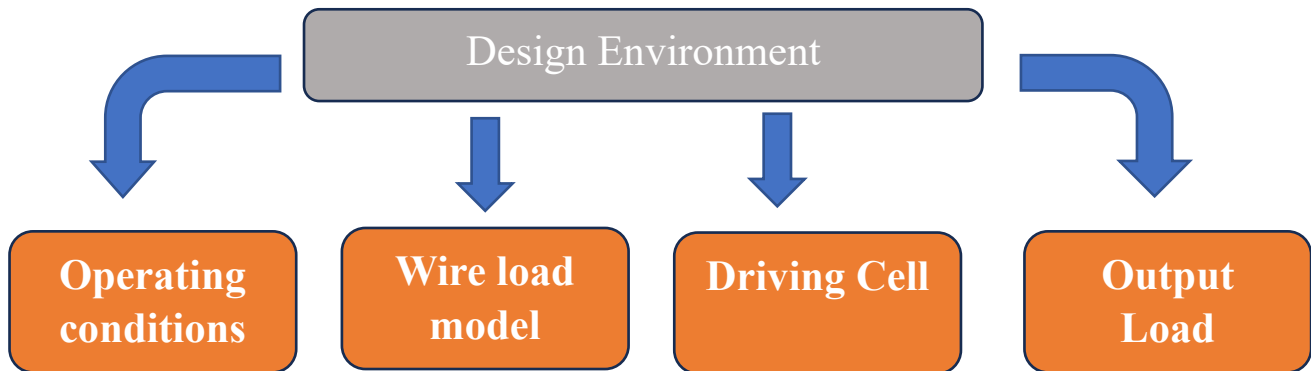


Figure 46 setting the design environment components

### 5.1.4.1 Operating conditions

As shown Figure 47, The technology library contains operating condition specifications such as (FF, SS, TT) with different voltages, temperature, and process to make the design take in account all corner cases and changes may occur in the chip while processing.

Goal of operating conditions: To ensure that chip will operate correctly in the different environments.

### 5.1.4.2 Wire load model

Wire load models are used only when Design Compiler is not operating in topographical mode. Wire load modeling allows you to estimate the effect of wire length and fanout on the resistance, capacitance, and area of nets. Design Compiler uses these physical values to calculate wire delays and circuit speeds. Semiconductor vendors develop wire load models, based on statistical information specific to the vendors' process. The models include coefficients for area, capacitance, and resistance per unit length, and a fanout-to-length table for estimating net lengths (the number of fanouts determines a nominal length) [25].

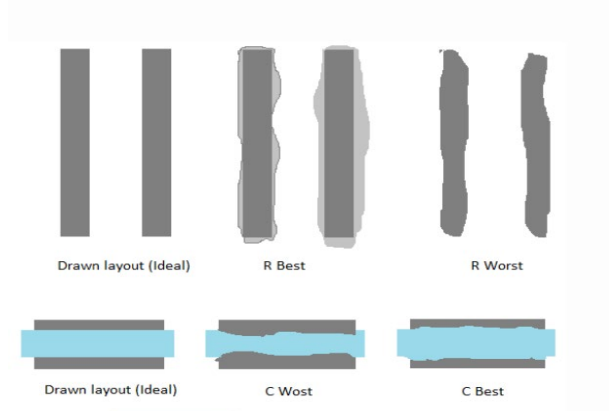


Figure 47 Different processes in the different operating conditions

Goal of WLM: To estimate the parasitics of routing between cells and not be more optimistic in STA.

Choice of WLD depends on design size.

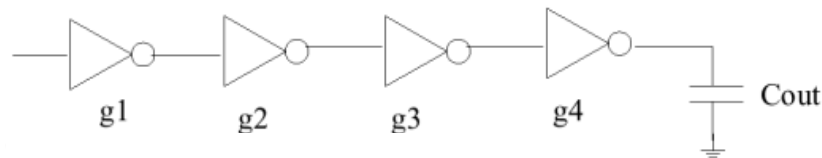


Figure 48 logic chain

### 5.1.4.3 Driving cell and output load

Our design consists of hundreds of logic chains like in

the figure and to calculate the delay of the chain you need the input transition and the output capacitance, so we set driving cell to estimate the input transition and set the output load.

### 5.4.5 Set design constrains

We have two types from constrains:

Logical DRC:

- Max fanout
- Max transition
- Max capacitance

```
#####logical constrains#####
set max area 0
set max fanout 10 [get designs ExecuteController]
set max capacitance 3.0 [get designs SystemBus]
```

Figure 48 Max fanout to have good STA estimation.

Optimization:

- Area:
  - Set max area
- Speed:
  - Clock
  - Input delay
  - Output delay

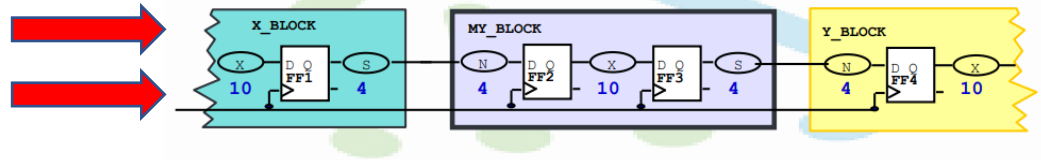


Figure 49 : constrain the blocks of design.

We put 60% from the clock cycle as input delay for the logic outside my block towards input ports and also 60% from the clock cycle to the logic outside my block towards output ports.

Problem: In the timing paths between input to output we have 120% from the clock cycle reserved outside my block and that cause me setup violation

To solve this problem, we use `set_max_delay` command to constrain these specific paths only.

```

16 Startpoint: io_resp_ready
17 (input port) clocked by REF_CLK
18 Endpoint: io_req_ready
19 (output port) clocked by REF_CLK
20 Path Group: REF_CLK
21 Path Type: max
22
23 Des/Clust/Port   Wire Load Model   Library
24 -----
25 VectorScalarMultiplier 1
26     tsmc13_wl50           scmetro_tsmc_cl013g_rvt_ss_1p08v_125c
27
28 Point           Incr           Path
29 -----
30 input external delay           12.00          12.00 r
31 io_resp_ready (in)             0.07           12.07 r
32 U3877/Y (A021X2M)              0.74           12.80 r
33 U4325/Y (CLKBUF12M)             0.78           13.58 r
34 io_req_ready (out)              0.00           13.58 r
35 data arrival time                13.58
36
37 max delay                    2.00           2.00
38 clock uncertainty              -0.20           1.80
39 output external delay          -12.00          -10.20
40 data required time             -10.20
41 -----
42 data required time             -10.20
43 data arrival time              -13.58
44 -----
45 slack (VIOLATED)                -23.78

```

Figure 50 Input to output timing path from Design compiler

```

145 #####additional_timing_constrains #
146 #set_max_delay "[expr 0.28*$ref_per]" -to [all outputs]
147 set_max_delay "[expr 0.1*$ref_per]" -to [get_ports io
148

```

Figure 51 Max delay to solve the problem.

## 5.4.6 Compile strategy

Top-down strategy:

Top-down compile, in which the top-level design and all its sub designs are compiled together.

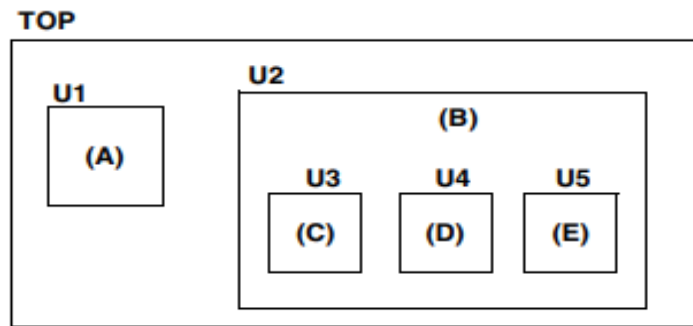


Figure 52 Hierarchy of design

When we use Top-down compile the Ram become full and the design compiler crashes. So, we had to compile every submodule separately with Its constraining and script and integrate them again.

- We synthesis the TOP

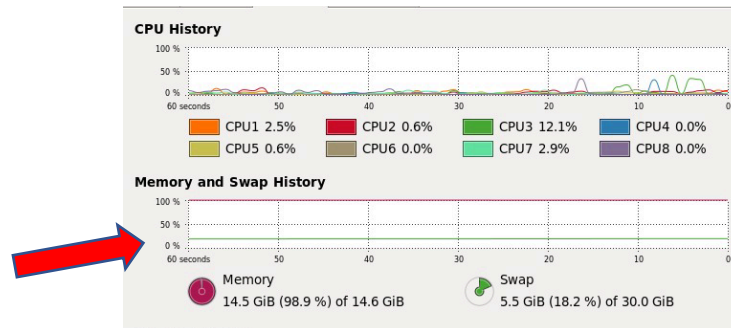


Figure 53: problem of Top-down strategy

Bottom-up strategy:

Bottom-up compile, in which the individual sub designs are compiled separately, starting from the bottom of the hierarchy, and proceeding up through the levels of the hierarchy until the top-level design is compiled.

-We synthesis {U3, U4, U5} then U2 with U1 then TOP

## 5.4.7 Synthesize and optimize the design

We have multiple efforts in the tool:

Most powerful : Compile\_ultra more optimization and maximum effort from the DC to meet the constrains

Colmpile -map\_effort:

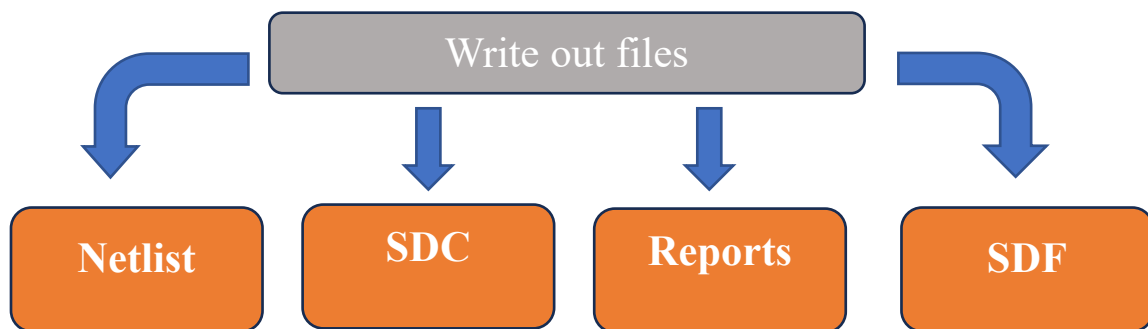
- High

- medium
- low

optimization order:

- Delay (WLM)
- design rule
- power
- area

### 5.4.8 Reports and netlist



*Figure 54 the output files from the synthesis step*

**Netlist:** The optimized gate-level file that mapped to the technology libraries (.libs) and has two formats Verilog format (.v) and binary format (.ddc) which is Synopsys database format. It is most recommended by Synopsys.

**SDC:** it has the constrains applied on the design with its ports and nets, It is a short form of “Synopsys Design Constraint” which is a common format for constraining the design which is supported by almost all Synthesis.

**Reports:** They are files reported from the Design compiler tool to see the result of the synthesis process and we have extracted 6 reports:

- Area
- Setup
- Hold
- Constrains
- Clocks
- Power



SDF: It stands for Standard delay format. It gives information on the timing data extensively used in backend VLSI design flows like gate level simulation the SDF after synthesis isn't accurate as it depends on WLM.

## 5.2 Results of the synthesis step

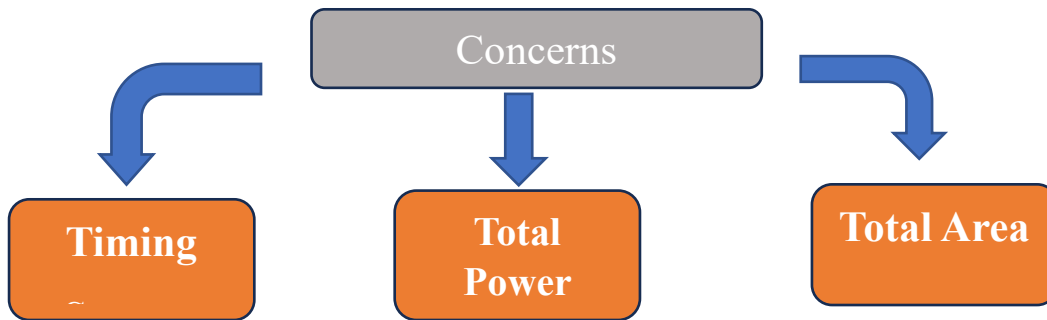


Figure 55 parameters under concern

### 5.2.1 Timing -setup

Table 3 Setup timing result from Synthesis step

Instance name	Setup state	Slack -critical path	Period
Execute controller	Met	0	20ns
FrontendTLB	Met	0	20ns
im2col	Met	4.57	20ns
LoadController	Met	0	20ns
Loopconv	Met	0	20ns
LoopMatmul	Met	0	20ns
Queue_57	Met	1.2	20ns
Queue_58	Met	0.38	20ns
ReservationStation	Met	0	20ns
StoreController	Met	0	20ns
Scratchpad	Met	0	20ns
Total accelerator	Met	-	20ns

## 5.2.2 Timing -hold

Table 4 Hold timing result from Synthesis step

Instance name	Setup state	Slack -critical path	Period
Execute controller	Met	0.47	20ns
FrontendTLB	Met	0.62	20ns
im2col	Met	12.9	20ns
LoadController	Met	0.89	20ns
Loopconv	Met	0.32	20ns
LoopMatmul	Met	0.81	20ns
Queue_57	Met	1.16	20ns
Queue_58	Met	1.16	20ns
ReservationStation	Met	0.61	20ns
StoreController	Met	0.77	20ns
Scratchpad	Violated	0	20ns
Total accelerator	Met	–	20ns

### 5.2.3 Area

Table 5 Table 4 Area result from Synthesis step

Instance name	Number of cells	cell area	Interconnect area	Total area	Percentage
Execute controller	192210.0	3098539.4	116024229.0	119122768.4	6.88%
FrontendTLB	3937.0	46637.3	2513013.2	2559650.5	0.15%
im2col	67.0	734.3	26600.1	27334.4	0.00%
LoadController	5565.0	77618.7	3466885.7	3544504.4	0.20%
Loopconv	59142.0	986664.1	35674748.9	36661413.0	2.12%
LoopMatmul	37754.0	517273.8	18498994.4	19016268.2	1.10%
Queue_57	1607.0	23604.6	1120939.4	1144544.0	0.07%
Queue_58	1636.0	24349.5	1155006.3	1179355.8	0.07%
ReservationStation	31897.0	346019.2	18678435.3	19024454.5	1.10%
StoreController	4168.0	68223.9	2310945.3	2379169.2	0.14%
Scratchpad	2677679	38857777.8	1487934334.8	1526792112.7	88.18%
Total accelerator	<b>337983.0</b>	<b>5189664.8</b>	<b>199469797.7</b>	<b>1529171281.9</b> <b>um2</b> <b>1.529 cm2</b>	<b>100%</b>

## 6 Formal Verification

Formal Verification or Logic Equivalence Check, popularly known as LEC is one of the most important parts of the ASIC VLSI design. Formal verification is done to check the functional/logical equivalence of a changed/modified design, which is the netlist of the design, concerning a golden design. The golden design (Verilog RTL model) is called the Reference design, and the design which needs to be checked, is the Implementation design. Formal Verification is done to make sure that unexpected changes have not happened in the design in each implementation step.

Formal Verification needs to be done in the following circumstances. usually, two kinds of verification are common in Formal Verification. I would use (ref) for reference and (impl) implementation:

- RTL (ref) vs Netlist(impl):
  - a) Golden RTL design VS Post-synthesis (Synth) netlist.
  - b) Golden RTL design VS Post-design for test (DFT) netlist.
  - c) Golden RTL design VS Post-place and route (PNR) netlist.
- Netlist before a specific step in ASIC flow VS Netlist after this specific step:
  - a) Pre-Synthesis netlist VS Post-Synthesis netlist.
  - b) Pre-DFT netlist VS Netlist Post-DFT netlist
  - c) Pre-PNR netlist VS Netlist Post-PNR netlist

RTL vs Netlist is used to verify that the synthesis has been doing the right job, i.e. the resulting netlist is functionally equal to the RTL. we can always simulate the netlist to verify that the netlist is ok, but netlist simulations take time, they can run for hours days, and in some cases even weeks. Now suppose that after running days of simulation, you found a very small bug, you fix it in RTL, synthesize it, and produce a new netlist. Now without the existence of any formal verification, you would run days of simulation again. With formal verification, you can quickly verify that the netlist out of synthesis is corresponding to what the RTL is, thus saving you a lot of time [26] .

Netlist vs Netlist may be done to verify:

1. Pre-layout and Post-layout netlists: After PNR, you produce a 'post-layout' netlist to check if this netlist is functionally equivalent to the 'pre-layout netlist', formal verification is a popular choice.
2. Engineering Change Order (ECO) changes which is the practice of introducing logic directly into the gate-level netlist corresponding to a change that happens in the RTL. i.e., Suppose you change a gate or two for fixing timing or for any other purpose. For example, if you insert a buffer to fix a hold violation in the netlist manually, you would then like to be sure that nothing else has broken as a result.

In any typical SOC design, there are several electronic design automation (EDA) tools in the Formal Verification flow. The major tools which do formal checking are Synopsys Formality, Cadence Co-formal, and Mentor Formal-pro.

Our EDA tool choice in this step of the ASIC flow was Synopsys Formality tool as it has a good pack of features such as Offering a graphical user interface (GUI) and a shell command-line interface (fm\_shell) environment. Using the shell command-line interface makes creating an automated TCL script to be run to all

modules of the Gemini architecture without the need to open the GUI of the Formality tool as we use the bottom-up technique in the ASIC flow.

The most important features of the formality tool were performing RTL-to-RTL, RTL-to-gate, and gate-to-gate design verifications, Reading Synopsys internal (.db or .ddc) formats. And Reading synthesizable System Verilog, Verilog, and VHDL.

## 6.1 Formal Verification Components

This section presents components and concepts that help you effectively use the Formality tool. It includes the following sections:

### 6.1.1 Logic Cones

A logic cone consists of combinational logic originating from a particular design object such as an output port, register, latch, black box input pin, or net driven by multiple drivers. The logic cone is terminated at certain design object outputs such as mentioned above. The design objects where logic cones are terminated are those used by Formality to create compare points that will be shown in the next section.

These design objects are primary outputs, internal registers, black box input pins, and nets driven by multiple drivers where at least one driver is a port or black box. The design objects at which logic cones terminate are primary inputs and those that Formality uses to create compare points.

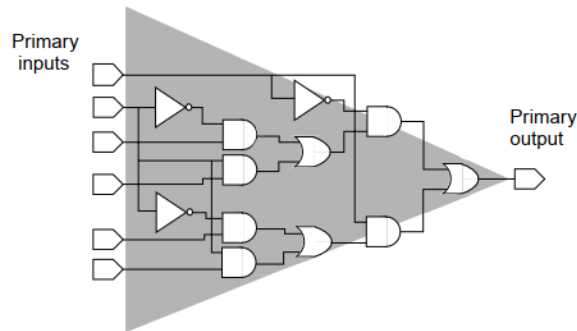


Figure 56 The Logic Cone concept

In Figure 56, the design object of concern is a primary output. Formality compares this design object (compare point) to a comparable object (compare point) in a second design during verification. The shaded area of the figure represents the logic cone for the primary output. The cone begins at the input net of the port and works back toward the termination points. In this illustration, the termination points are nets connected to primary inputs [27].

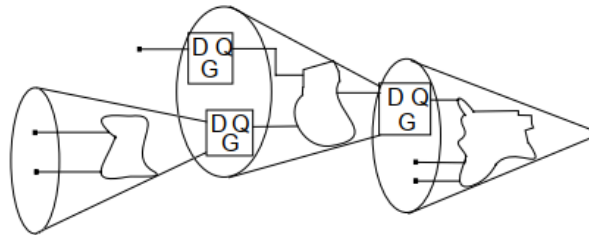
## 6.1.2 Compare Points

A compare point is a design object used as a combinational logic endpoint during verification. A compare point can be an output port, register, latch, black box input pin, or net driven by multiple drivers.

Formality uses the following design objects to automatically create compare points:

- Primary outputs
- Sequential elements
- Black box input pins
- Nets driven by multiple drivers, where at least one driver is a port or black box

As shown in Figure 57, Formality verifies a compare point by comparing the logic cone from a compare point in the implementation design against a logic cone for a matching compare point from the reference design.



*Figure 57 Compare points of the cones of logic*

When functions defining the cones of logic for a matched pair of compare points (one from the reference design and one from the implementation design) are proved by Formality to be functionally equivalent, the result is that the compare points in both the reference and implementation designs have passing status. If all compare points in the reference design pass verification, the final verification result for the entire design is a successful verification [27].

Before design verification, Formality tries to match each primary output, sequential element, black box input pin, and qualified net in the implementation design with a comparable design object in the reference design.

For Formality to perform a complete verification, all compare points must be verifiable. There must be a one-to-one correspondence between the design objects in the reference and implementation designs. However, the following cases do not require a one-to-one correspondence to get complete verification when you are testing for design consistency:

- Implementation design contains extra primary outputs.
- Either the implementation or reference design contains extra registers, and no compare points fail during verification.

Compare points are primarily matched by object names in the designs. If the object names in the designs are different, Formality uses various methods to match up these compare points automatically. You can also manually match these object names when all automatic methods fail.

Compare-point matching techniques in Formality can be broadly divided into two categories:

- Name-based matching techniques
- Non-name-based matching techniques

Unmatched design objects from either the implementation or reference design are reported as failing compare points, with a note indicating that there is no comparable design object in the reference design [26].

### 6.1.3 Containers

A container is a complete, self-contained space into which Formality reads designs. It is typical for one container to hold the reference design while another holds the implementation design.

A container typically includes a set of related technology libraries and design libraries that fully describe a design that is to be compared against another design. A technology library is a collection of “parts” associated with a particular vendor and design technology. A design library is a collection of designs associated with a single design effort. Designs contain design objects such as cells, ports, nets, and pins. A cell can be a primitive or an instance of another design as shown in figure 58 [26, 27].

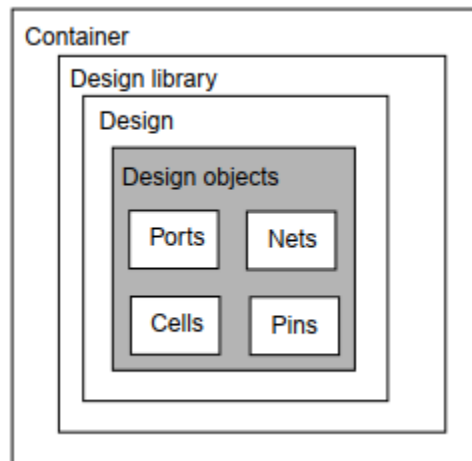


Figure 58 Containers in a Hierarchical Design

In general, to perform a design comparison, you should load all of the information about one design into a container (the reference), and all the information about the other design into another container (the implementation).

As shown the Figure 59, Each container can hold many design and technology libraries, and each library can hold many designs and cells. Components of a hierarchical design must reside in the same container [26].

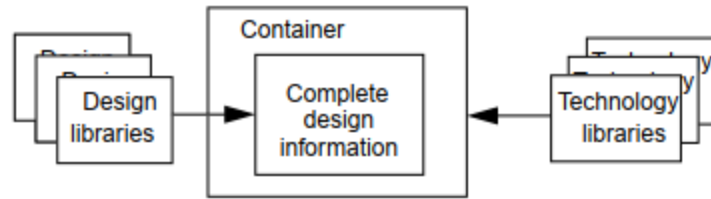


Figure 59 Containers can hold design and technology libraries

## 6.2 Formality Flow

The flow chart in Figure 60 provides an overall flow to the Formality design verification process. That we will go through each step in the flow.

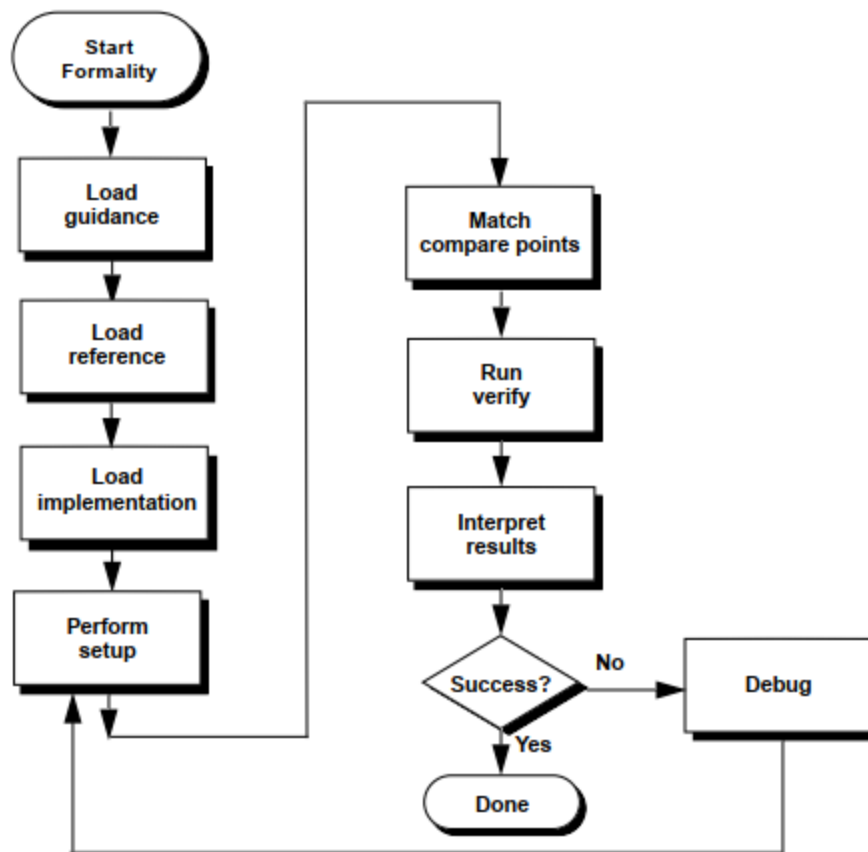


Figure 60 the flow of The formality tool



## 6.2.1 Start Formality

Formality offers two working environments:

- The Formality shell, `fm_shell`, is the command-line interface. The `fm_shell` commands are made up of command names, arguments, and variable assignments. Commands use the tool command language that is used in many applications in the EDA industry.
- The Formality GUI is the graphical, menu-driven interface. It allows you to perform verification and provides schematic and logic cone views to help you debug failed verifications.

Our choice was using the shell command-line interface makes creating an automated TCL script to be run to all modules of the Gemini architecture without the need to open the GUI of the Formality tool as we use the bottom-up technique in the ASIC [27].

## 6.2.2 Load Guidance

Load guidance or the Automated Setup File is the step before specifying the reference and implementation designs, it is optional to load an automated setup file (.svf) into Formality. The benefit of an automated setup file (.svf) is that it allows the implementation tool to automatically provide setup information to Formality. It helps Formality understand and process design changes caused by other tools that were used in the design flow. Formality uses this information to assist compare point matching and correctly set up verification without your intervention. It eliminates the need to enter setup information manually, a task that is time-consuming and error-prone [27].

For example, during synthesis, the phase of a register might be inverted. This change is recorded in the automated setup file (.svf). When you read the automated setup file into Formality, the tool can account for the phase inversion during compare point matching and verification.

## 6.2.3 Load Reference and Implementation Designs

To run Formality, it is a must to read in both a reference and an implementation design and any related technology libraries. Optionally, you can pass additional setup information from Design Compiler to Formality, and you can load automated setup files from other related tools.

As shown in Figure 6, you first read in these automated setup files. Next, read only the libraries (.db) and designs (.v) that are needed for the reference, then immediately specify its top-level design. Read only the files that you need for the implementation design (netlist) (.v), then immediately specify its top-level design. Finally, you must set the top-level design for the reference design before proceeding to the implementation design [26].

The formality tool uses the concept of container as shown previously in the container section to create two containers one for the reference design, called the reference container, and another for the implementation design, called the implementation container, As shown in Figure 61.

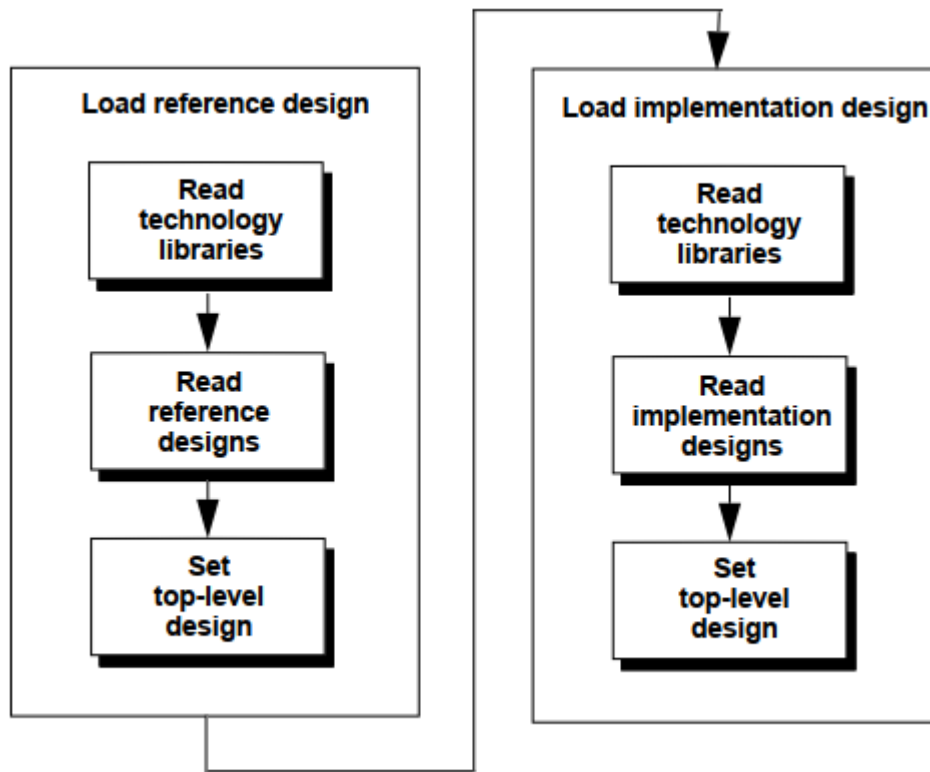


Figure 61 Formality Read-Design Process Flow

## 6.2.4 Perform Setup

After reading designs into the Formality environment and linking them, sometimes it needs to set design-specific options to help Formality perform verification. For example, if someone is aware of certain areas in a design that Formality cannot verify, you might want to prevent the tool from attempting to verify those areas. Or, if you want to speed up verification, you might declare blocks in two separate designs black boxes.

There are some cases you need to use setup commands:

- Exclude some compare points from the verification step.
- Force some ports or pins to have a constant value whether 0 or 1.
- Allow the formality tool to treat undriven nets and pins as don't care values during the verification step.

## 6.2.5 Matching Compare Points

After preparing the verification environment and setting up your design, the design is ready to match compare points and then verify the design.

As described in “Compare Points”, compare point matching is either named-based, as shown in Figure 7, or non-name-based. The following matching techniques occur by default when you match compare points.

Name-based matching:

- Exact-name matching
- Compare point matching based on net names
- Name filtering

Non-name-based matching:

- Topological equivalence
- Signature and topological analysis

These matching techniques are executed in the following order:

- Exact-name matching
- Name filtering
- Topological equivalence
- Signature analysis
- Compare point matching based on net names

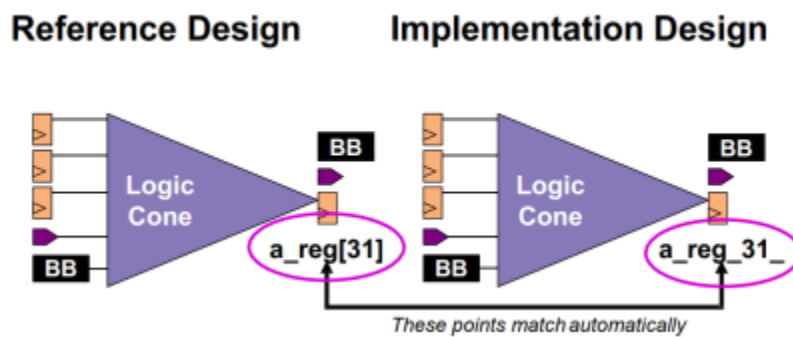


Figure 63 compare points is matched by their name between Designs

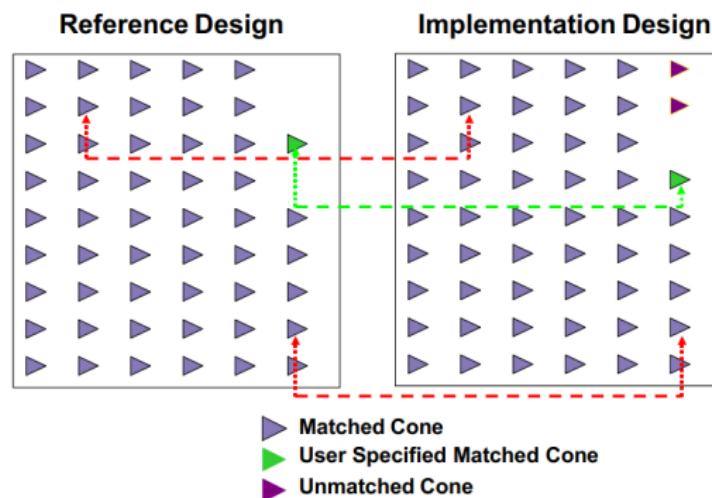


Figure 62 Matches Corresponding Points between Designs

After a technique succeeds in matching a compare point in one design to a compare point in the other design, that compare point becomes exempt from processing by other matching techniques and is passed to verify step [27].

## 6.2.6 Performing Verification

When the verify command is used, the Formality attempts to verify logical equivalence for each logic cone and its compare points in the implementation design and the reference design. As shown in the figure below.

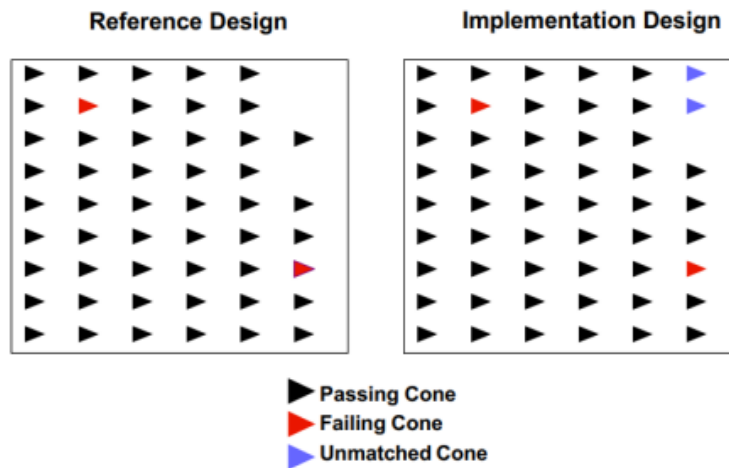


Figure 64 Verifies logical equivalence for each logic cone between Designs

## 6.2.7 Reporting and Interpreting Results

As part of your troubleshooting efforts, Formality allows you to report on passing, failing, unverified, and aborted compare points. During verification, Formality assigns one of three types of status messages for each compare point it identifies:

- **Passing**  
A passing point represents a compare point match that passes verification. Passing verification means that Formality determined that the functions that define the values of the two compare point design objects are functionally equivalent.
- **Failing**  
A failing point represents a compare point match that does not pass verification. Failing verification means that Formality determined that the two design objects that constitute the compare point are not functionally equivalent [26].
- **Unverified**  
An unverified point represents a compare point that has not yet been verified. Formality normally stops verification after 20 failing points have been found.
- **Aborted**  
An aborted point represents a compare point that Formality did not determine to be either passing or failing. The cause can be a compare point that is too difficult to verify [27].

Based on the previous categories, Formality classifies final verification results in one of the following ways:

- **Succeeded**  
The implementation design was determined to be functionally equivalent to the reference design. All compare points passed verification.
- **Failed**  
The implementation design was determined to be not functionally equivalent to the reference design. Formality could not successfully match all design objects in the reference design with

comparable objects in the implementation design, or at least one design object in the reference design was determined nonequivalent to its comparable object in the implementation design (a compare point failure).

- Inconclusive

Formality could not determine whether the reference and implementation designs are equivalent in the following cases:

- A matched pair of compare points was too difficult to verify, causing an “aborted” compare point, and no failing points.
- The verification was interrupted.

### 6.3 Result of the Formal Verification

As we use the bottom-up technique in the ASIC flow. We perform the formality step for each module of the Gemini architecture. We choose to run formality for only compare Golden RTL between Post-synthesis Netlist and Post-PNR Netlist as comparing a Netlist before a specific step in ASIC flow and after this specific step will not get a useful indication to the formal verification but comparing Golden RTL and a netlist is always giving a better indication what changes have happened to the RTL after a specific step in ASIC flow.

*Table 6 1the result of the formal verification step*

Module name	Golden RTL VS Post-synthesis Netlist	Golden RTL VS Post-PNR Netlist
ExecuteController	Verification Succeeded	Verification Succeeded
FrontendTLB	Verification Succeeded	Verification Succeeded
Im2col	Verification Succeeded	Verification Succeeded
LoadController	Verification Succeeded	Verification Succeeded
Loopconv	Verification Succeeded	Verification Succeeded
LoopMatmul	Verification Succeeded	Verification Succeeded
Queue_57	Verification Succeeded	Verification Succeeded
Queue_58	Verification Succeeded	Verification Succeeded
ReservationStation	Verification Succeeded	Verification Succeeded
Scratchpad	Verification Succeeded	Verification Succeeded
StoreController	Verification Succeeded	Verification Succeeded

As shown in the previous table each module of the Gemini architecture has passed the formal verification check. That means the functional/logical operation of a changed/modified design (netlist) after synthesis and PNR steps are the same as the functional/logical operation of the RTL before synthesis and PNR steps. It also means the synthesis tool and PNR tool have done the right job.

## 7 CHAPTER FIVE: PLACE AND ROUTE

### 7.1 Introduction to PnR

The rapid advancement of integrated circuit technology has led to the development of complex and highly integrated electronic systems, these systems often consist of numerous components, such as logic gates, memory elements, and interconnects, all interconnected on a silicon chip. However, designing the physical layout of these components and efficiently routing interconnects poses significant challenges.

Place and route (PnR) is a crucial step in the physical design process of electronic systems, where the locations of components are determined and the interconnections between them are established. PnR plays a fundamental role in translating the logical representation of a circuit into a physical layout that meets various design objectives, including performance, area utilization, power consumption, and manufacturability.

The primary goal of PnR is to optimize the physical layout of the components on a chip, taking into account the specified design constraints and objectives. The placement step determines the positions of the components on the chip's surface, aiming to minimize wire length, optimize timing, and reduce congestion. On the other hand, the routing step involves establishing the interconnections between the components while considering factors such as wire length, signal integrity, and congestion.

PnR algorithms leverage various techniques, including mathematical optimization, heuristic algorithms, and machine learning approaches, to achieve efficient and effective physical design. These algorithms consider numerous factors, such as signal delays, power consumption, thermal constraints, and manufacturing constraints, to produce a physically realizable and high-performing layout.

The challenges in PnR arise due to the increasing complexity and scale of modern electronic systems. The large number of components, the interconnect patterns, and the requirement to satisfy multiple conflicting design objectives make PnR a computationally intensive task. Moreover, as technology advances, the need for faster and more energy-efficient electronic systems further intensifies the demand for sophisticated PnR techniques.

Efficient PnR plays a vital role in the success of electronic systems, impacting their performance, power consumption, and cost. A well-designed PnR process can significantly improve the yield of integrated circuits, reduce the TTM (time-to-market), and enhance the overall reliability and manufacturability of electronic systems.

In conclusion, PnR is a critical step in the physical design process of electronic systems, aiming to optimize the placement and routing of components on a chip. Efficient PnR algorithms and techniques play a crucial role in achieving high-performance, low-power, and cost-effective designs. As the complexity of electronic systems continues to grow, advancements in PnR are crucial for meeting the evolving demands of modern integrated circuit design.

## 7.2 PnR Flow

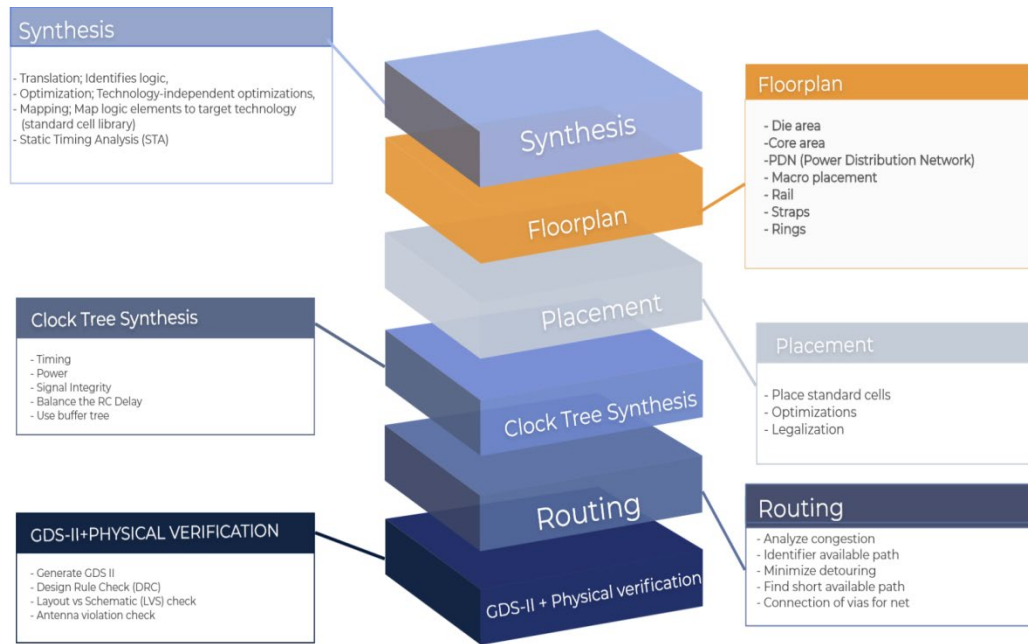


Figure 65 PnR Flow

As shown in the above figure the PnR is the step that follows the synthesis step as it take the Gate Level Netlist that is produced from the synthesis step and the general flow of the PnR consist of :-

1. Floor planning
2. Placement
3. CTS (clock tree synthesis)
4. Detailed Routing
5. Physical verification
6. GDS-II steam

And in the next section we will explain this flow in details in a test case (Queue\_57.v)

The tool used: - **Cadence Innovus 2019.**

## 7.3 Detailed PnR Flow for Queue\_56

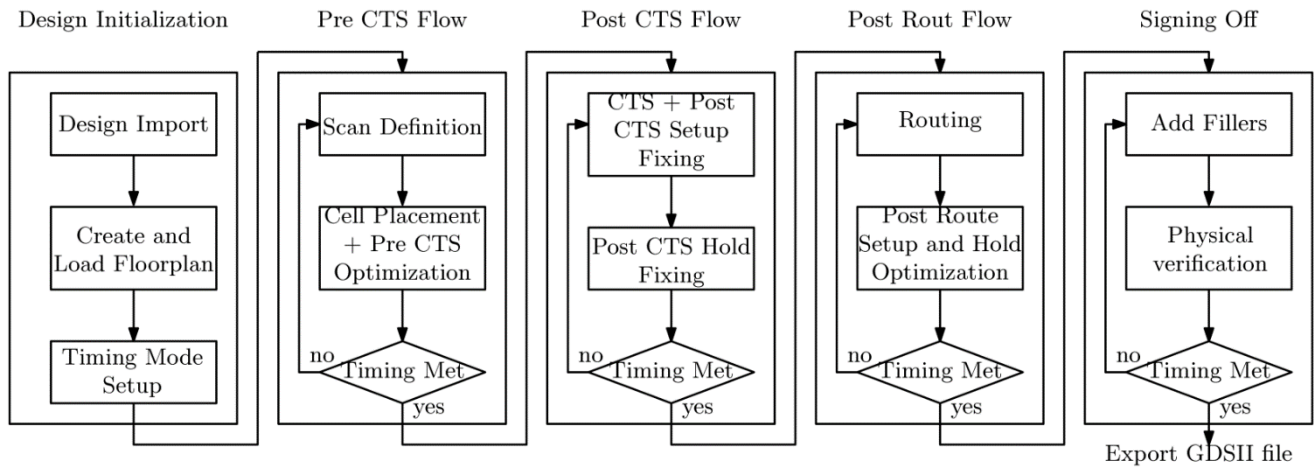


Figure 66 Detailed PnR flow

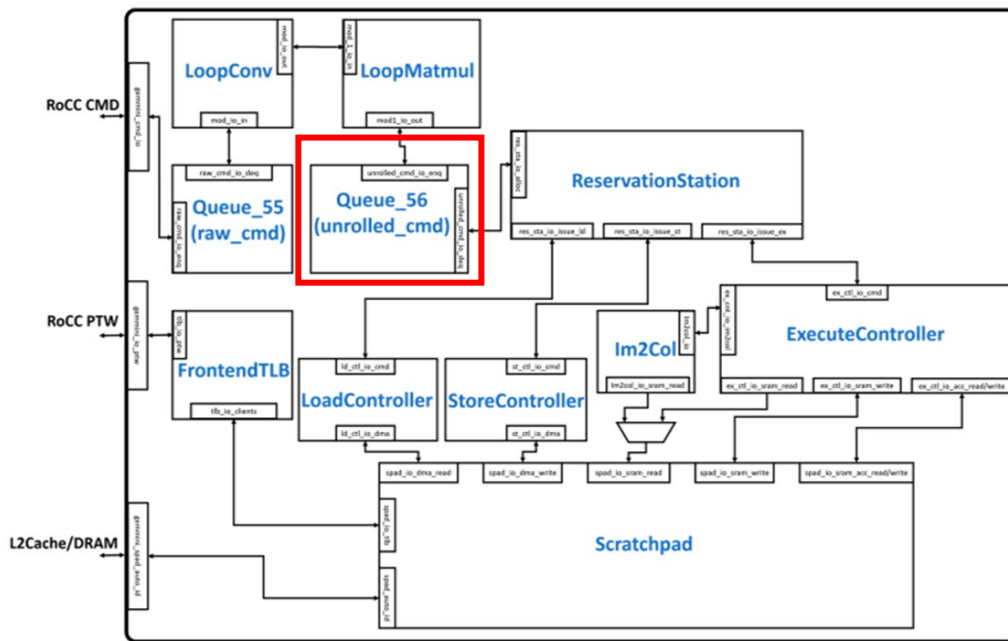


Figure 67 Design modules

### 7.3.1 Design Import

This is the first step in the PnR flow where we load the netlist from the synthesis operation and also load the LEF and MMMC files

**LEF Files:**-Library Exchange Format (LEF) is a specification for representing the physical layout of an integrated circuit in an ASCII format. It includes design rules and abstract information about the cells. There are three kinds of LEF files which are Technology LEF File, Standard Cell LEF file, IO LEF file.



**Multi-Mode Multi Corner (MMMC) View Definition File:-** The MMMC view definition file contains pointers to two type of files which are Liberty Timing Models Files (LIB), Design Constraints (SDC). LIB files contain timing information about the standard cells and IO pads and SDC file contains the different timing constraints

### 7.3.2 Floor Planning

In this step we choose the chip area (Innovus calculate it automatically based on the synthesis results) and choose the core utilization (the area of the standard cells divided by the total area) for the placement step which is **0.7** in our case to make easy routing and also choose the core margins in all direction which is **5** in our case to place the power a  
it 1 resulting in sq

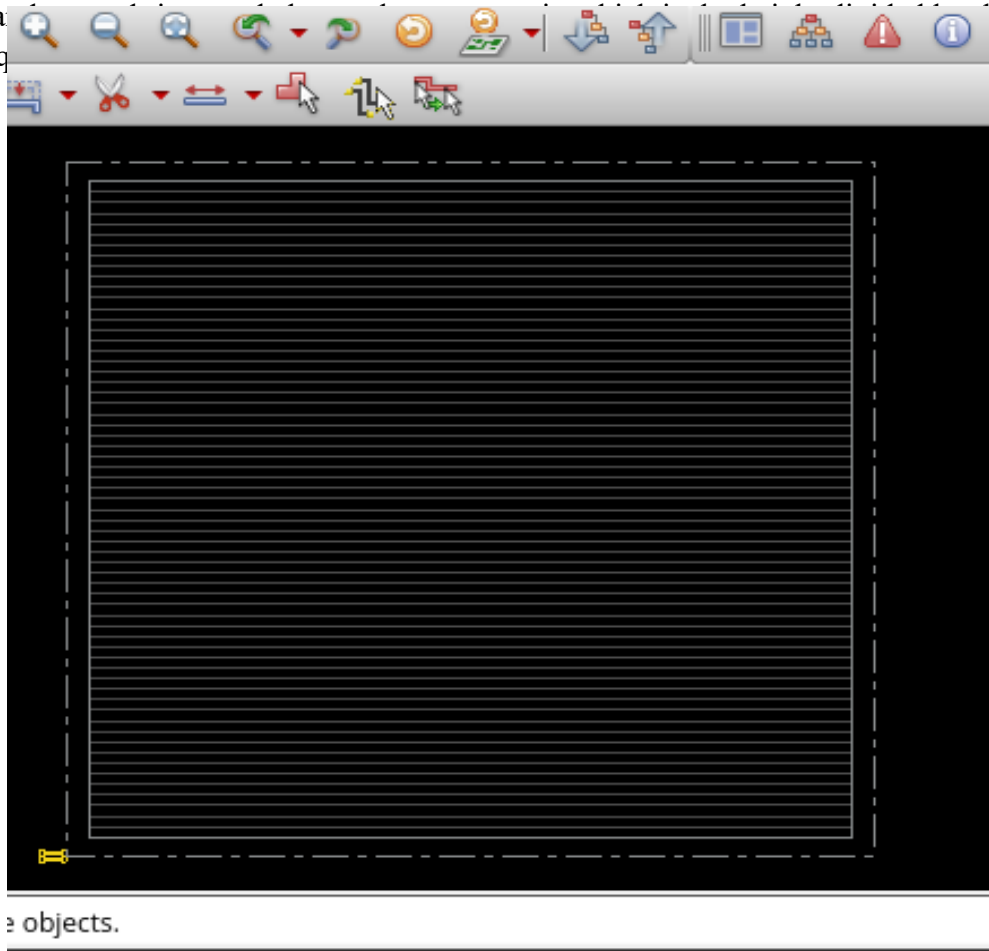


Figure 68 Final Floor Plan

### 7.3.3 Power Planning

In this step we add the power and ground rings in the margin area to feed all the chip cells then add the stripes to establish a (VDD, GND) network

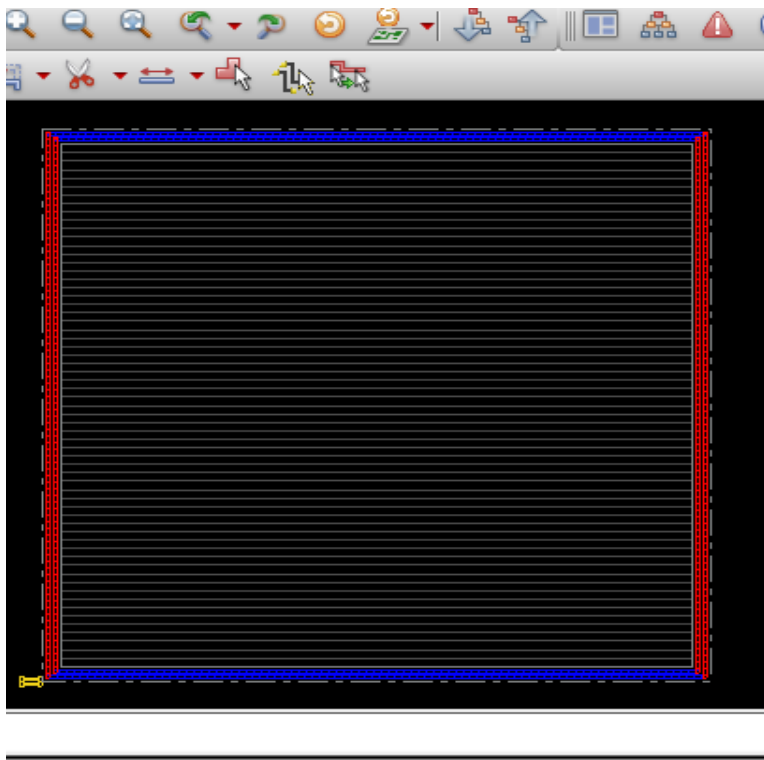


Figure 70 Power Planning (Rings)

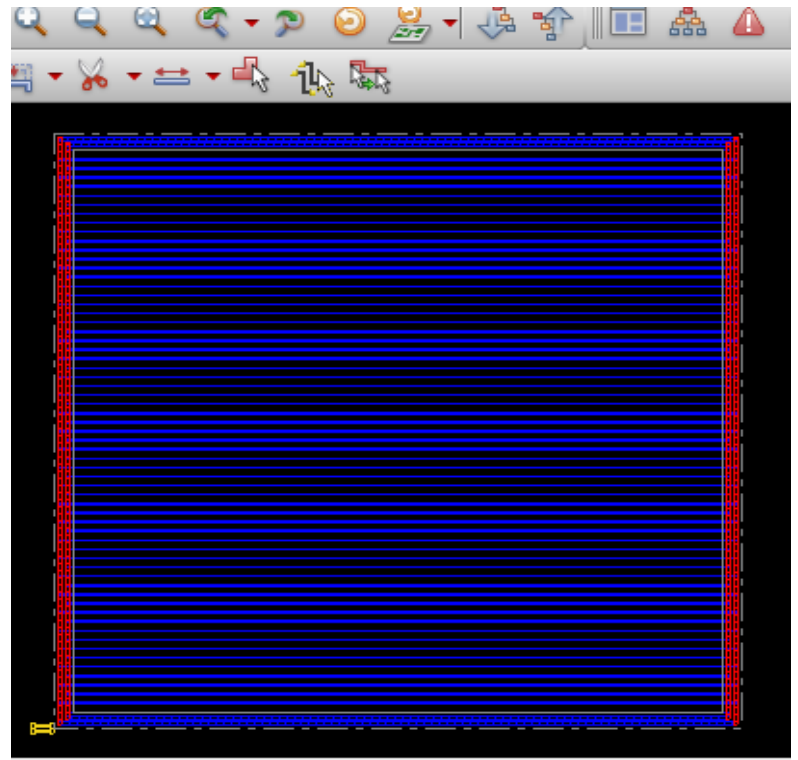


Figure 69 Power Planning (stripes)

### 7.3.4 Placement

Placement is the process of placing the standard cells inside the core boundary in an optimal location. The tool tries to place the standard cell in such a way that the design should have minimal congestions and the best timing. Every PnR tool provides various commands/switches so that users can optimize the design in a better way in terms of timing, congestion, area, and power as per their requirements. Based on the preferences set by the user, the tool tray to place and optimize the Placement, does not place only the standard cells present in the synthesized netlist but also places many physical only cells and adds buffers/inverters as per the requirement to meet the timings, DRV, and foundry requirements. Also in this step we do the trial route which is not the actual route to estimate some design metrics.

```
METAL1 (1F) length: 0.000000e+00um, number of vias: 5641
METAL2 (2V) length: 1.741906e+04um, number of vias: 8834
METAL3 (3H) length: 1.879645e+04um, number of vias: 718
METAL4 (4V) length: 6.326710e+03um, number of vias: 266
METAL5 (5H) length: 5.307040e+03um, number of vias: 132
METAL6 (6V) length: 1.965130e+03um, number of vias: 30
Total length: 4.981439e+04um, number of vias: 15621
```

Figure 71 Trial Routing Results

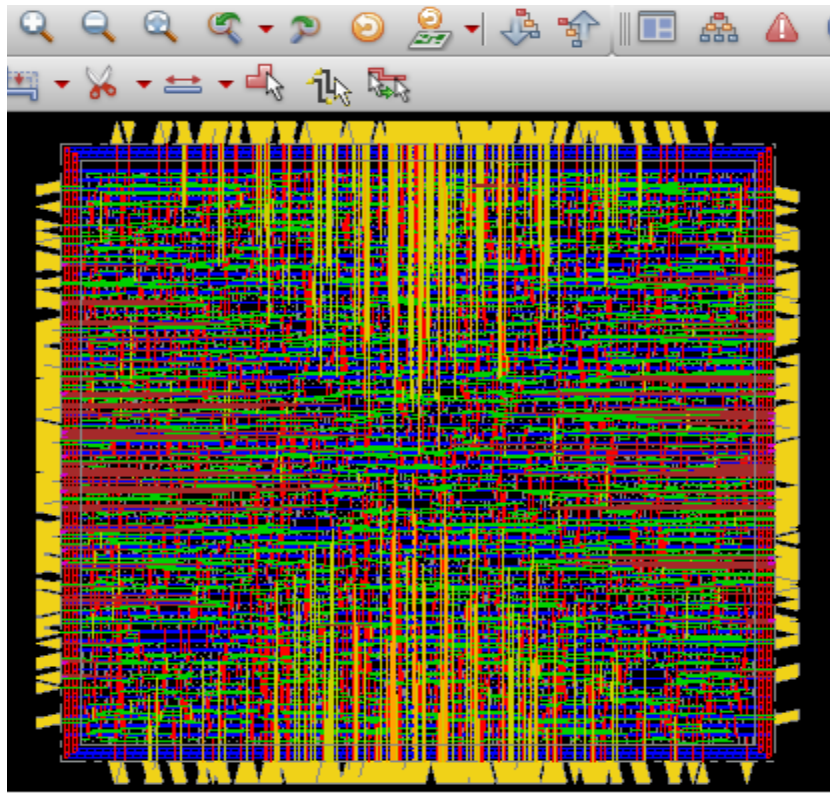


Figure 72 Chip after Placement and Trial Routing

### 7.3.5 Timing analysis and optimization preCTS

After we finish the placement and before adding the clock tree (CTS) we should check the timing and the DRV's of the design and if any violation happens, we should solve them by the optimization step.

```

Setup views included:
  setup_analysis_view_func
  
```

Setup mode	all	reg2reg	default
WNS (ns):	-8.397	1.810	-8.397
TNS (ns):	-1088.3	0.000	-1088.3
Violating Paths:	267	0	267
All Paths:	800	533	800

DRVs	Real		Total
	Nr nets (terms)	Worst Vio	Nr nets (terms)
max_cap	7 (7)	-1.794	7 (7)
max_tran	17 (1617)	-9.030	17 (1618)
max_fanout	6 (6)	-189	7 (7)
max_length	0 (0)	0	0 (0)

Density: 64.509%  
Routing Overflow: 0.00% H and 0.00% V

Figure 73 Timing and DRV's preCTS

As seen, there is a timing violation and some DRV's so we need the optimization step

```

Setup views included:
  setup_analysis_view_func
  
```

Setup mode	all	reg2reg	default
WNS (ns):	3.578	11.217	3.578
TNS (ns):	0.000	0.000	0.000
Violating Paths:	0	0	0
All Paths:	800	533	800

DRVs	Real		Total
	Nr nets (terms)	Worst Vio	Nr nets (terms)
max_cap	0 (0)	0.000	0 (0)
max_tran	0 (0)	0.000	0 (0)
max_fanout	0 (0)	0	1 (1)
max_length	0 (0)	0	0 (0)

Density: 65.669%  
Routing Overflow: 0.00% H and 0.00% V

Figure 74 Timing and DRV's after optimization

As seen the density increase from **64.5%** to **65.67%** but we solve the violations.

### 7.3.6 Clock Tree Synthesis (CTS)

In this step we create the clock tree for our clock (REF\_CLK) and check the timing and the DRV's same as the previous step and make sure there is no violations

The only change from the previous step is that the density increase from **65.669** to **66.087** as a result for the CTS creation

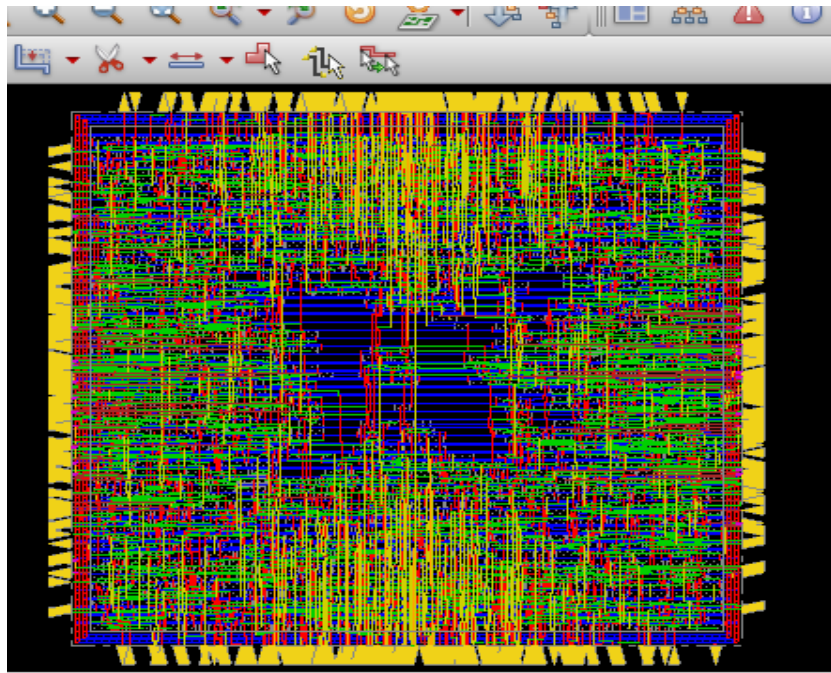
### 7.3.7 Detailed Routing

As mentioned before in the placement step we make a trial routing which is not the final routing and it's done for the purpose of estimation of some design metrics but in this step we make the detailed (actual) routing for all the cells in the chip and calculate the actual delay for all the interconnects and save them in the **SDF** file after finishing the flow

**SDF File**:-Standard Delay Format File it gives information about the delay of the paths and interconnects and also used in the final gate level simulation

```
#Post Route wire spread is done.
#Total number of nets with non-default rule or having extra
#Total wire length = 48704 um.
#Total half perimeter of net bounding box = 39022 um.
#Total wire length on LAYER METAL1 = 2179 um.
#Total wire length on LAYER METAL2 = 17007 um.
#Total wire length on LAYER METAL3 = 18088 um.
#Total wire length on LAYER METAL4 = 8407 um.
#Total wire length on LAYER METAL5 = 2422 um.
#Total wire length on LAYER METAL6 = 601 um.
#Total wire length on LAYER METAL7 = 0 um.
#Total number of vias = 11680
#Up-Via Summary (total 11680):
```

*Figure 75 Detailed Routing Results*



*Figure 76 Chip after Detailed Routing*

After the detailed routing we also check timing and make optimization if needed, the density doesn't change

### **7.3.8 Verification**

In this step we verify the geometry and the connectivity of the design to make sure there is no violation of the DRC.

```

# Design:      Queue_57      # Design:      Queue_57
# Command:    verifyGeometry # Command:    verifyConnectivity
#####
#####
Verify Connectivity Report is created on

Begin Summary ...
Cells       : 0
SameNet     : 0
Wiring      : 0
Antenna     : 0
Short       : 0
Overlap     : 0
End Summary

No DRC violations were found

Begin Summary
Found no problems or warnings.
End Summary

```

Figure 78 Geometry Report

Figure 77 Connectivity Report

As seen, there are no geometry or connectivity violations.

### 7.3.9 Filler Insertion

In this step we add some filler cells to increase the utilization of the chip and fill the empty places in the chip, the library provide various sizes of the filler cells such as {FILL1M FILL2M FILL4M FILL8M FILL16M FILL32M FILL64M} the number indicate the filler width, after adding the filler the density is 100%

Density: 100.000%

Figure 79 Density after Filler insertion

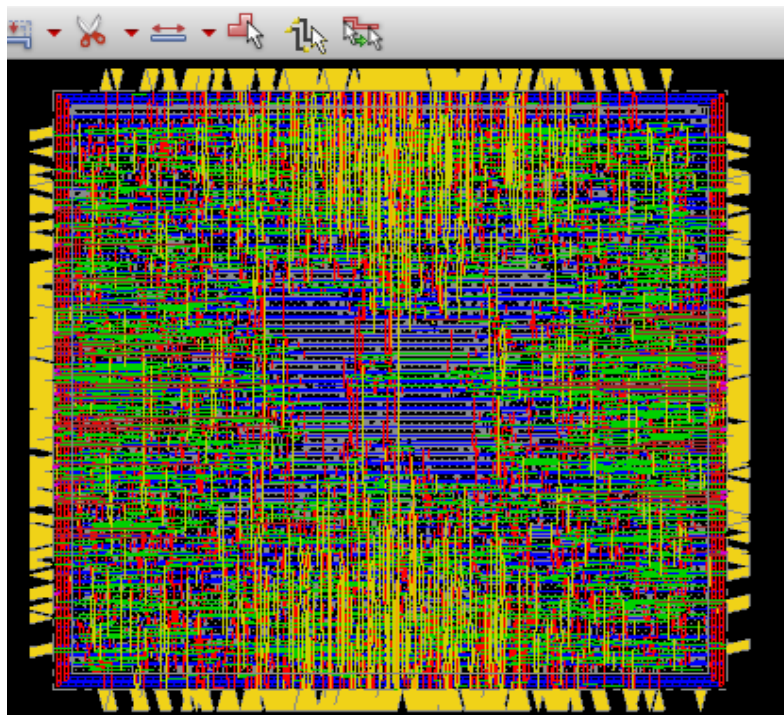


Figure 80 Chip after Filler insertion

### 7.3.10 Summary

After done with this flow we generate the **GDSII** file and the **SDF** file and some reports (Area, power, Timing) that will be discuss in the next section.

**Graphic Design System (GDSII Files):** This file is required for layout extraction at the time of layout generation we need the mapping file to produce actual GDS file of the IC.



## 7.4 Results

### 7.4.1 Area Results

In the next table all the modules final areas will be listed and its percentage from the total area.

Table 7 Area Results

Module name	Area in ( $\mu\text{m}^2$ )	Percentage %
ExecuteController	7323129	14.335%
FrontendTLB	33848	0.066%
Im2col	614	0.001%
LoadController	113450	0.222%
Loopconv	1065238	2.085%
LoopMatmul	380153	0.744%
Queue_57	22179	0.043%
Queue_58	23063	0.045%
ReservationStation	284168	0.556%
Scratchpad	41792113	81.81%
StoreController	46337	0.091%
<b>Total</b>	51084292	100%

The expected final area will be larger than the **51 mm<sup>2</sup>** as this area is the summation of all the modules area after integrating them with each other and the largest 2 modules are the Scratchpad and the ExecuteController as they take **96%**

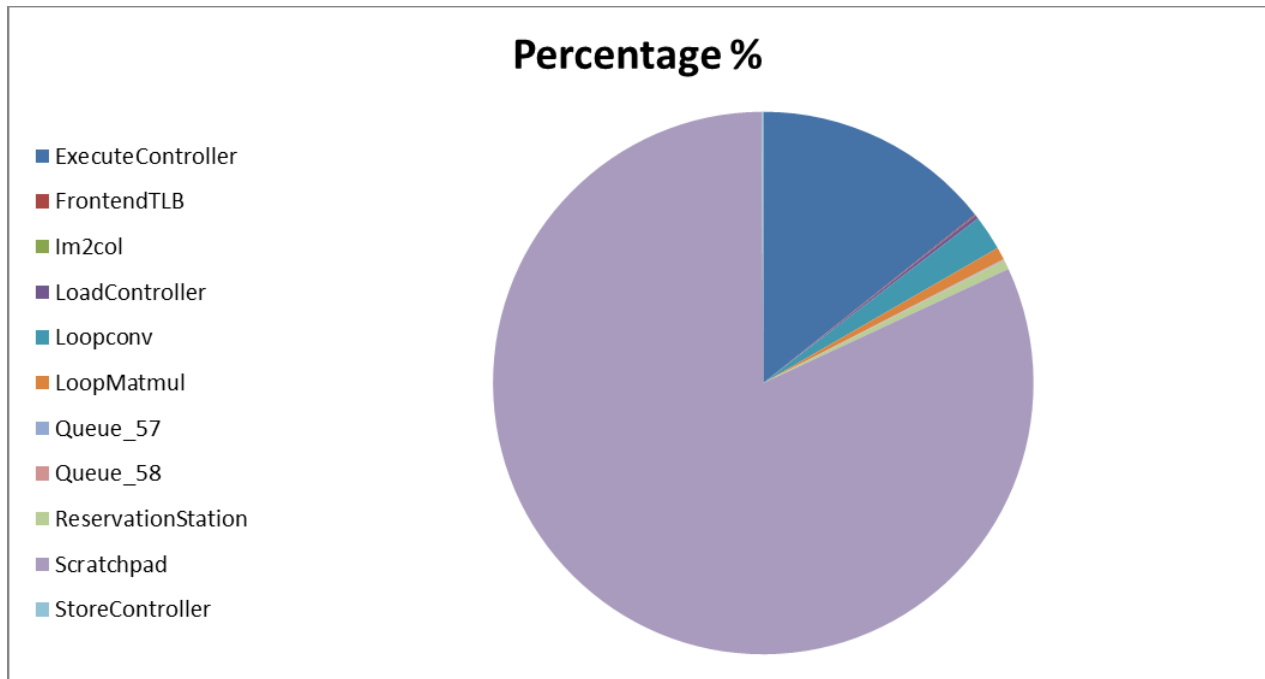


Figure 81 Area Distribution

## 7.4.2 Area comparison

In the next table we list the area results for implement the Gemmini accelerator with the 22nm FFL process technology and the final total area is **1.029 mm<sup>2</sup>** compared to our **51 mm<sup>2</sup>** implementation with TSMC 130nm process technology we found area increase by **50X** which is reasonable and close to the rough estimate which is the square of the technology scaling (130/22) square which is **36X**.

Table 8 Area Results for 22 nm FinFet technology [11]

Module name	Area (µm)	Percentage %
Spatial Array	116k	11.3%
Scratchpad	690k	67.1%
CPU	171k	16.6%
Other	52k	5.0%
<b>Total</b>	1029k	100%

## 7.4.3 Power Results

In the next table all the modules total power will be listed and its percentage from the total chip power.



Table 9 Power Results

Module name	Power in (mW)	Percentage %
ExecuteController	125.6	13.064%
FrontendTLB	1.72	0.179%
Im2col	0.13	0.014%
LoadController	3.6	0.374%
Loopconv	39.2	4.077%
LoopMatmul	13.6	1.415%
Queue_57	1.86	0.193%
Queue_58	2.05	0.213%
ReservationStation	11.82	1.229%
Scratchpad	758	78.842%
StoreController	3.85	0.4%
<b>Total</b>	<b>961.43</b>	<b>100%</b>

The total power is approximately **1W** and the 2 most power-hungry modules are the Scratchpad and the ExecuteController as they consume about **92%** from the chip total power

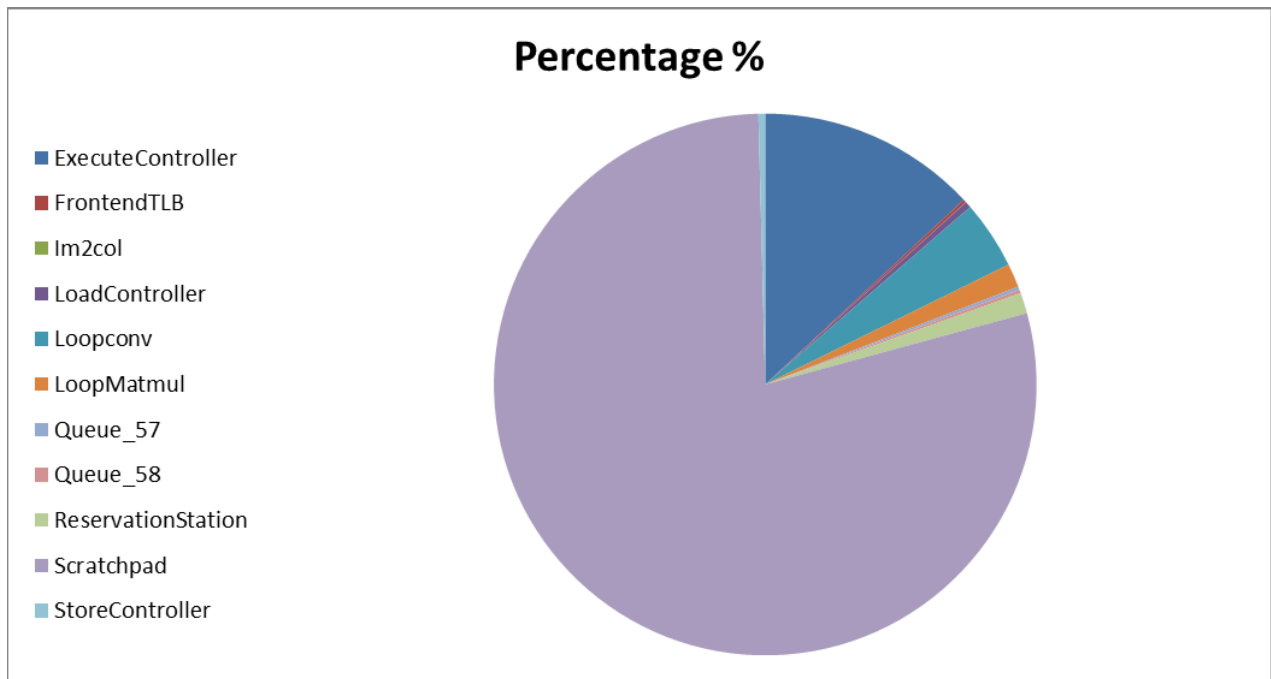


Figure 82 Power Distribution

## 8 Conclusion

In conclusion, our graduation project focused on understanding and implementing the Gemmini accelerator for deep learning model inference. We began by acquiring a comprehensive understanding of hardware architectures specifically designed for deep learning. Subsequently, we delved into the Gemmini framework, studying its architecture and configuration options.

Based on a published paper analysis, we identified the suitable configuration for our model and generated the Register Transfer Level (RTL) representation. This allowed us to verify the correctness of the Gemmini accelerator by testing it on the ResNet-50 model, demonstrating its ability to make deep learning inference.

To bring our implementation to fruition, we employed a Digital Backend flow, synthesis, placement, and routing stages. This flow facilitated the transformation of the RTL design into a physical layout, ensuring efficient placement of components and optimal interconnections. Then list the final area and power for all the modules and we end with the GDS2 file for the Gemmini modules

## 9 Future work

Although we achieved some satisfactory results for us, but there were and still are some challenges that, unfortunately, due to lack of time, we were unable to accomplish them. Therefore, we will present those problems that may one day be solved by us or by other engineers :

1. Multi modes (capture and scan modes) and (Design for Test) DFT

Due the huge number of lines in the generated RTL code of Gemmini architecture, almost half million line of Verilog, and it's not easy to read and debugged it. Therefore, it was a hard and time-consuming mission to identify each combinational logic exist between 2 flipflops and replace them by the scan-flipflop version.

2. As the the Scratchpad is power-hungry module. so, finding a better implementation (hard macro or IP) to replace the Scratchpad module to achieve a reasonable area and power consumption is a need.
3. Integrate all the generated GDS files for all modules to only one GDS representing the top module for Gemmini architecture. This step needs a good experience in physical layout engineering to draw the layout of the top module and leave a good area to the layout of the 11 modules of Gemmini architecture to merge the GDS files later by using cadence innovus tool or Calibre.

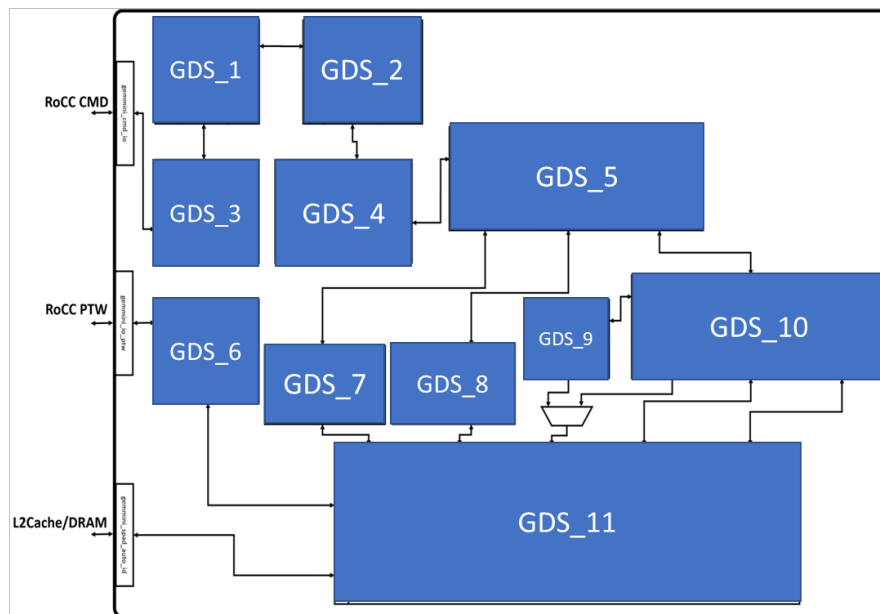


Figure 83 an overview of the top-module

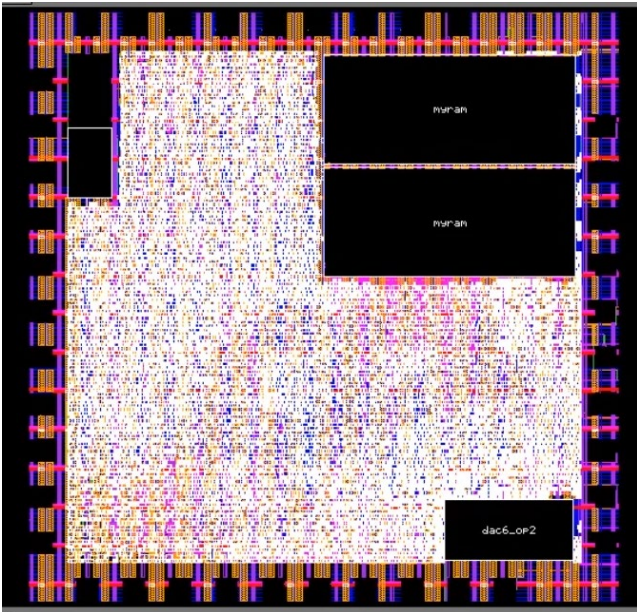


Figure 84 the top-module layout before merging

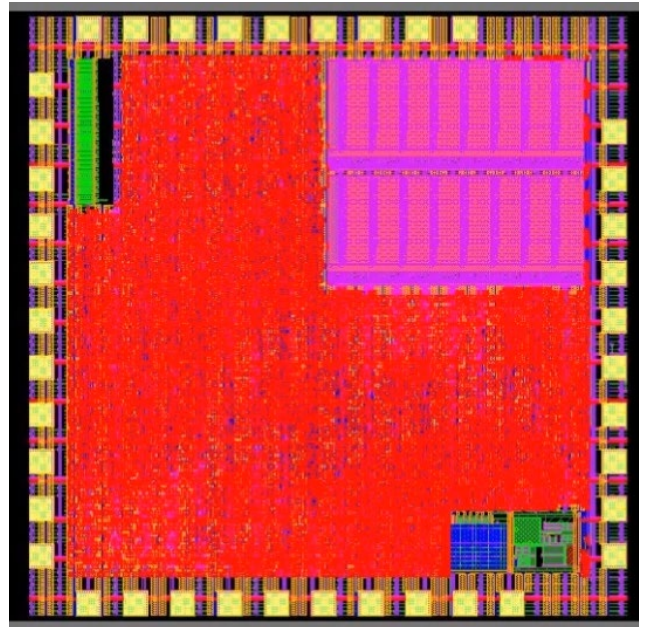


Figure 85 the top-module layout after merging

## References

- [1] Introduction to cognitive radio networks and applications, Boca Raton: CRC Press, 2016.
- [2] T. K. I. m. E. d. P. Merima Kulin, "End-to-end learning from spectrum data: a deep learning approach for wireless signal identification in spectrum monitoring applications.," *IEEE Access*, vol. 6, pp. 18484-18501, 2018.
- [3] A. M. ., N. M. ., a. A. M. V. Valeria Loscri, "Wireless cognitive network technologies and protocols," in *Modeling and Simulation of Computer Networks and Systems Methodologies and Applications*, Waltham, Elsevier, 2015, pp. 119-153.
- [4] A. Goldsmith, *Wireless Communications*, New York: Cambridge University Press, 2005.
- [5] C. M. Bishop, *Pattern Recognition and Machine learning*, Singapore: Springer, 2006.
- [6] Y.-H. C. T.-J. Y. a. J. S. E. Vivienne Sze, *Efficient Processing of deep neural networks*, Morgan Claypool: Springer, 2020.
- [7] Chipyard's documentation, <https://chipyard.readthedocs.io/en/stable/>, version "1.10.0" accessed on 10 November 2022.
- [8] EE290-2, Spring 2021: Hardware for Machine Learning course in UC Berkely.
- [9] EE241B : Advanced Digital Circuits course in UC Berkely.
- [10] Gookyi, D.A.N.; Lee, E.; Kim, K.; Jang, S.-J.; Lee, S.-S. Deep Learning Accelerators' Configuration Space Exploration Effect on Performance and Resource Utilization: A Gemmini Case Study. *Sensors* 2023, 23, 2380. <https://doi.org/10.3390/s23052380>.
- [11] H. Genc et al., "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration," 2021 58th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 2021, pp. 769-774, doi: 10.1109/DAC18074.2021.9586216..
- [12] D. G. J. W. C. S. H Liew, Hammer: a modular and reusable physical design flow tool, *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 1335-1338, 2022.
- [13] A. Emad et al., "Deep Learning Modulation Recognition for RF Spectrum Monitoring," 2021 IEEE International Symposium on Circuits and Systems (ISCAS), Daegu, Korea, 2021, pp. 1-5, doi: 10.1109/ISCAS51556.2021.9401658..
- [14] Gemmini tutorial mlsys 2022 ,Gemmini Tutorial MLSys 2022. Available at: <https://sites.google.com/berkeley.edu/gemmini-tutorial-mlsys-2022/> ..
- [15] Gemmini. Available online: <https://github.com/ucb-bar/gemmini> (accessed on 10 November 2022)..

- [16] Howard, A.G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; Adam, H. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv 2017, arXiv:1704.04861. preprint..
- [17] Krizhevsky, A.; Sutskever, I.; Hinton, G.E. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 2017, 60, 84–90. [CrossRef].
- [18] Chen, J.; Xiong, N.; Liang, X.; Tao, D.; Li, S.; Ouyang, K.; Zhao, K.; DeBardleben, N.; Guan, Q.; Chen, Z. TSM2: Optimizing Tall-and-Skinny Matrix-Matrix Multiplication on GPUs. In Proceedings of the ACM International Conference on Supercomputing, Phoeni.
- [19] Anderson, A.; Gregg, D. MobileNets: Optimal DNN Primitive Selection with Partitioned Boolean Quadratic Programming. arXiv 2018, arXiv:1710.01079. preprint..
- [20] Kung, H.T. Why Systolic Architectures? *IEEE Comput.* 1982, 15, 37–46. [CrossRef].
- [21] Jouppi, N.P.; Young, C.; Patil, N.; Patterson, D.; Agrawal, G.; Bajwa, R.; Bates, S.; Bhatia, S.; Boden, N.; Borchers, A.; et al. Eyeriss: In-Datcenter Performance Analysis of a Tensor Processing Unit. In Proceedings of the Annual International Symposium, Architecture, Toronto, ON, Canada, 24–28 June 2017; IEEE: San Francisco, CA, USA, 2017..
- [22] Sze, V.; Chen, Y.H.; Yang, T.J.; Emer, J.S. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* 2017,, 105, 2295–2329. [CrossRef].
- [23] Digital Design with Chisel. Available online: <https://github.com/schoeberl/chisel-book> (accessed on 10 November 2022)..
- [24] Rocket Core. Available online: <https://chipyard.readthedocs.io/en/stable/Generators/Rocket.html> (accessed on 10 November 2022)..
- [25] Design Compiler® User Guide Version L-2016.03-SP2, June 2016.
- [26] I. Synopsys, Formality User Guide, Synopsys, Inc, 2010.
- [27] Formality® User Guide Version Z-2007.06, June 2007.