

Summer Training Report

Vedant Pahariya — Priyanshi Jain

Contents

1	TinyML	3
2	Neural Networks	3
2.1	Non-Linearity	3
2.2	Backpropagation	4
3	PyTorch	4
3.1	What are Tensors?	5
3.2	Concept of seeding	5
3.3	Autograd	6
3.4	Training a Basic Neural Network	7
3.4.1	Import Required Libraries	7
3.4.2	Pre-Processing the Data	7
3.4.3	Define the Neural Network	8
3.4.4	Writing Training Pipeline	8
3.4.5	Evaluation	9
3.4.6	Basics of OOPS in Python	9
3.5	Using NN Pytorch Module	9
3.6	Dataset & DataLoader Class	9
3.6.1	Dataset Class	10
3.6.2	DataLoader Class	11
4	RISC-V	11
4.1	RISC vs. CISC: Instruction Sets and Code Density	12
4.2	RISC-V Name	12
4.3	RISC-V Instruction Set Variants	13
4.4	Shakti Processors	13
4.4.1	Shakti Processor Variants	13
4.5	Vega Processors	13
4.5.1	Vega Processor Variants	14
5	SystemVerilog	14
5.1	About RTL	14
5.2	Verilog Vs SystemVerilog	14
5.3	Data Types	15
5.3.1	2-State Data Types	15

5.3.2	Struct Data Type	16
5.3.3	Enumerated Data Types	17
5.3.4	Fixed Arrays (Packed/Unpacked)	18
5.3.5	Dynamic Arrays	20
5.3.6	Queue	22
5.3.7	Associative Arrays	22
5.3.8	Summary	24
5.4	Display Output	25
5.4.1	Format Specifiers	26
5.4.2	Zero-Padding/Space-Padding	26
5.4.3	System Task	26
5.5	Tasks & Functions	27
5.5.1	Enhancements in SystemVerilog:	27
5.5.2	Types of Tasks	27
5.5.3	Syntax of Task	28
5.5.4	Passing arguments to Tasks	29
5.6	Interface	30
5.6.1	Syntax of Interface	30
5.6.2	Modports	31
5.7	Blocking & Non-Blocking Assignments	32
5.7.1	Delay-Based Timing Control	33
5.8	Event Scheduler	34
5.8.1	Simulation Time & Time Slot	34
6	CVA6	36
6.1	Setting up CVA6	37
6.2	Stages in CVA6	40
6.2.1	PC generation Stage	41
6.2.2	Instruction Fetch Stage	41
6.2.3	Instruction Decode Stage	42
6.3	Understanding Spike & Verilator	43
6.4	Scoreboard	43
6.5	Model on Hardware	43
6.5.1	Tests in CVA6	44
6.6	Performance Modelling	45
6.6.1	RVFI Trace	45
7	PULP-TrainLib	46
8	UVM	46
9	On-Device Training	46
9.0.1	Subsection	46

1 TinyML

Reference: [What is TinyML?](#) by datacamp

Many Machine learning applications require a lot of computational power and memory. Because of this demand, they are usually run on powerful servers or cloud computing platforms.

In machine learning, the workflow consists of two main phases:

- **Training** - When the model learns from data by adjusting its parameters
- **Inference** - When the trained model is used to make predictions on new data

In addition to these models being computationally expensive to train, running inference on them is often quite expensive too because of following reasons:

- They require a lot of memory to store the model parameters
- They require a lot of computational power to run the model and make predictions
- More energy than tiny devices can provide

If machine learning is to expand its reach and penetrate additional domains, a solution that allows machine learning models to run inference on smaller, more resource-constrained devices is required. The pursuit of this solution is what has led to the subfield of machine learning called Tiny Machine Learning (TinyML).

TinyML is a type of machine learning that allows models to run on smaller, less powerful devices. It involves hardware, algorithms, and software that can analyze sensor data on these devices with very low power consumption, making it ideal for always-on use-cases and battery-operated devices.

Benefits of TinyML:

- Latency
- Energy savings
- Reduced bandwidth
- Data privacy

2 Neural Networks

Neural networks are also called artificial neural networks (ANNs). Put simply, neural networks form the basis of architectures that mimic how biological neurons signal to one another.

2.1 Non-Linearity

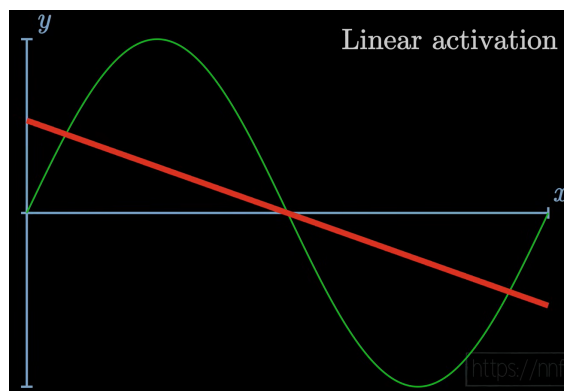
Non-linearity is crucial in neural networks because it allows them to learn complex patterns in the data. Without non-linear activation functions, a neural network would essentially behave like a linear regression model, regardless of the number of layers it has. This means it could only learn linear relationships, which are often insufficient for real-world tasks.

What is the use of the activation function in a neural network? Why we need Non-Linearity?

Reference: [Watch here](#)

As shown in the video above, using linear functions, we can't construct a complex function like sinusoidal function. The output of a linear function is always a straight line, irrespective of the no. of layers in the network, the final equivalent function boils down to simple $y = mx + c$, which means it can only represent linear relationships between inputs and outputs. Common non-linear activation functions include:

- Sigmoid
- Tanh
- ReLU (Rectified Linear Unit)



2.2 Backpropagation

Reference: [Lecture by Justin Johnson](#)

3 PyTorch

Reference: [Deep Learning using PyTorch - YouTube Playlist](#)

In 2002, Torch was a scientific computing framework with wide support for machine learning algorithms. But there were two problems in Torch: first, it is written in Lua, which is not widely used in the industry; second, it was using the static computational graph, which is not flexible and dynamic like PyTorch/ TensorFlow.

These problems are fixed by PyTorch which is an open-source machine learning library developed by Facebook's AI Research lab. It is widely used for deep learning applications and provides a flexible and dynamic computational graph, making it easy to build and train neural networks.

Core Features of PyTorch include:

- **Tensor Computations:** PyTorch provides a multi-dimensional array (tensor) library that is similar to NumPy but with GPU acceleration.

- **GPU Acceleration:** PyTorch can utilize GPUs for faster computation, making it suitable for deep learning tasks.
- **Dynamic Computation Graph:** PyTorch uses a dynamic computation graph, allowing for more flexibility in building and modifying neural networks.
- **Automatic Differentiation:** PyTorch uses a technique called automatic differentiation to compute gradients for high optimization tasks.
- **Distributed Training:** Training models on multiple GPUs or across multiple machines rather than a single GPU.
- **Interoperability with other libraries:** PyTorch can easily integrate with other popular libraries such as NumPy, SciPy, and Cython.

3.1 What are Tensors?

Tensors are a fundamental data structure in PyTorch, representing multi-dimensional arrays. They are similar to NumPy arrays but with additional capabilities for GPU acceleration and automatic differentiation. Tensors can be created from Python lists or NumPy arrays and can be manipulated using a variety of operations.

Examples:

- **0D Tensor (Scalar):** A scalar tensor is a single value, which can be created from a Python number or a NumPy scalar. Output of Loss function is a scalar value.
- **1D Tensor:** A 1D tensor is an array or vector, which can be created from a Python list or NumPy array. The feature vector of text is a 1D tensor, also known as embedding vector.
- **2D Tensor:** A 2D tensor is a matrix, which can be created from a list of lists or a 2D NumPy array. Gray Scale images are 2D tensors, where each pixel is represented by a single value like MNIST dataset.
- **3D Tensor:** A 3D tensor is a cube, which can be created from a list of 2D arrays or a 3D NumPy array. RGB images are 3D tensors, where each pixel is represented by three values (R, G, B).
- **4D Tensor:** A 4D tensor is a hypercube, which can be created from a list of 3D arrays or a 4D NumPy array. Video frames or Batch of RGB images are 4D tensors, where each frame is represented by a 3D tensor (RGB image).
- **5D Tensor:** Video data can be represented as a 5D tensor, where each frame is a 3D tensor (RGB image) and the batch size is the first dimension. For example, a batch of 10 video clips, each with 30 frames, can be represented as a 5D tensor with shape (10, 30, height, width, 3 RGB channels).

3.2 Concept of seeding

Seeding is a technique used to ensure that the random number generation in PyTorch is reproducible. By setting a seed value, we can ensure that the same random numbers are generated each time we run the code, which is useful for debugging and

testing purposes. A seed is a number that initializes the random number generator. If you use the same seed, you'll get the same sequence of random numbers.

Function	Purpose
<code>torch.manual_seed(seed)</code>	Sets the seed for CPU random number generation in PyTorch.
<code>torch.cuda.manual_seed(seed)</code>	Sets the seed for current GPU .
<code>torch.cuda.manual_seed_all(seed)</code>	Sets the seed for all GPUs .
<code>torch.backends.cudnn.deterministic = True</code>	Forces cuDNN to use deterministic algorithms (slower, but reproducible).
<code>torch.backends.cudnn.benchmark = False</code>	Disables the auto-tuner that may introduce non-determinism.

3.3 Autograd

Autograd is PyTorch's automatic differentiation engine that powers neural network training. It allows us to compute gradients automatically, which is essential for optimizing model parameters during training.

Autograd keeps a record of data (tensors) and all executed operations in a directed acyclic graph (DAG) consisting of Function objects. For DAG, leaves are the input tensors, and roots are the output tensors. By traversing this graph in reverse (back-propagation), we can compute gradients for all tensors involved in the computation.

In a forward pass, autograd does two things simultaneously:

- run the requested operation to compute a resulting tensor
- maintain the operation's gradient function in the DAG.

The backward pass kicks off when `.backward()` is called on the DAG root. autograd then:

- computes the gradients from each `.grad_fn`,
- accumulates them in the respective tensor's `.grad` attribute
- using the chain rule, propagates all the way to the leaf tensors.

Clearing Gradients

After the backward pass, it's important to clear the gradients of the model parameters to prevent accumulation from previous iterations. This is typically done using the following command:

```
{Leaf(input)_Variable_Name}.grad.zero_()
```

The Underscore after zero indicates that the operation is done in-place.

There is other command `optimizer.zero_grad()` which is used to clear the gradients of all model parameters in a single call. This is often used in training loops to reset gradients before the next forward and backward pass. This function sets the gradients of all model parameters to zero, ensuring that the next forward and backward pass starts with a clean slate.

Stopping the Backpropagation

After the model is trained, we may want to stop the backpropagation for certain tensors/operations. This can be done in three ways:

- `detach()` - Returns a new tensor that shares the same data but does not require gradients. This is useful when we want to use a tensor in a computation without tracking its gradients.
- `with torch.no_grad():` - A context manager that temporarily disables gradient tracking. This is useful for inference or evaluation, where we don't need to compute gradients.
- `requires_grad=False` - Setting this attribute on a tensor prevents it from being included in the computation graph and stops gradient tracking for that tensor.

```
1 import torch
2 # Create a tensor with requires_grad=True
3 x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
4 # Perform some operations
5 y = x * 2 + 1
6 # Compute gradients
7 y.backward(torch.tensor([1.0, 1.0, 1.0])) # Backpropagate
8 print(x.grad) # Print gradients
9 # Detach the tensor from the computation graph
10 x_detached = x.detach() # Detach x from the graph
11 # Perform operations without tracking gradients
12 y_no_grad = x_detached * 2 + 1 # No gradients will be computed
13 # Use with torch.no_grad() context
14 with torch.no_grad():
15     y_no_grad = x * 2 + 1 # No gradients will be computed
16 # Set requires_grad=False
17 x.requires_grad = False # Stop tracking gradients for x
```

3.4 Training a Basic Neural Network

[Youtube Video by CampusX](#)

Here, we are discussing few important points discussed in the video above, which is a good introduction to training a basic neural network using PyTorch. Following are the steps to train a basic neural network using PyTorch:

3.4.1 Import Required Libraries

3.4.2 Pre-Processing the Data

Before training a neural network, we need to prepare the data. This involves loading the dataset, preprocessing it, and splitting it into training and validation sets. PyTorch provides several utilities for data loading and preprocessing, such as the `torchvision` library for image datasets.

The data values should be normalized to a range of 0 to 1 or -1 to 1, depending on the activation function used in the neural network. This helps in faster convergence

during training.

We use methods from scikit learn for data scaling, such as `StandardScaler` for standardization.

```
from sklearn.preprocessing import StandardScaler

# Create a StandardScaler object
scaler = StandardScaler()

# Fit the scaler on the training data
scaler.fit(X_train)

# Transform the training and validation data
X_train_scaled = scaler.transform(X_train)
X_val_scaled = scaler.transform(X_val)
# Note: Do not fit the scaler on validation data, only transform it.

# Convert the scaled data to PyTorch tensors
X_train_tensor = torch.tensor(X_train_scaled, dtype=torch.float32)
X_val_tensor = torch.tensor(X_val_scaled, dtype=torch.float32)

# Convert the labels to PyTorch tensors
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
y_val_tensor = torch.tensor(y_val, dtype=torch.long)
```

While normalizing the data, keep in mind to use the same normalization parameters statistics (mean and standard deviation) for both training and validation datasets. This ensures that the model sees the same scale of data during both training and evaluation. This is done by using the `fit` method on the training data and then using the `transform` method on both training and validation data as shown in above code snippet.

3.4.3 Define the Neural Network

3.4.4 Writing Training Pipeline

From the Autograd section, we know that we need to compute the gradients of the loss function with respect to the model parameters. This is done using the `backward()` method on the loss tensor as shown below:

```
loss.backward()
```

This loss can be computed using a loss function which we can get from the `torch.nn` module or can be defined explicitly in Neural Network class.

There are two things we have to keep in mind,

- To make the gradients zero before the backward pass, we need to call the `zero_grad()` method on the model parameters. This is important because PyTorch accumulates gradients by default, and we want to avoid accumulating gradients from previous iterations.

- To turn off the gradient tracking for the model parameters while updating the parameters or running the models for evaluation, we can use the `torch.no_grad()` context manager or this is handled by the optimizer itself while using `optimizer.step()`.

3.4.5 Evaluation

3.4.6 Basics of OOPS in Python

Reference: [Video-1](#) (Intro to OOPS) [Video-2](#) (Classes & Objects) [Video-3](#) (Constructors) [Video-4](#) (Inheritance)

Coding Neural Networks in PyTorch involves defining classes, objects and methods. This also includes concept related to inheritance, polymorphism, and encapsulation. So, its good to go through the above videos to understand the basics of OOPS in Python.

3.5 Using NN Pytorch Module

The `torch.nn` module in PyTorch provides a high-level interface for building neural networks. It includes pre-defined layers, loss functions, and optimizers that make it easier to construct and train neural networks. Key components of `torch.nn` include:

- **Modules:** The `nn.Module` class is the base class for all neural network modules in PyTorch. It provides methods for defining the forward pass and managing parameters.
- **Layers:** Pre-defined layers like `nn.Linear`, `nn.Conv2d`, and `nn.ReLU` for building neural networks.
- **Activation Functions:** Pre-defined activation functions like `nn.ReLU`, `nn.Sigmoid`, and `nn.Tanh` for introducing non-linearity in the network, allowing it to learn complex patterns.
- **Loss Functions:** Pre-defined loss functions like `nn.CrossEntropyLoss` and `nn.MSELoss` for computing the loss during training.
- **Sequential:** The `nn.Sequential` class allows us to define a neural network as a sequence of layers, making it easier to build simple feedforward networks.
- **Optimizers:** Pre-defined optimizers like `torch.optim.SGD` and `torch.optim.Adam` for updating model parameters during training.
- **Dropout and Batch Normalization:** Layers like `nn.Dropout` and `nn.BatchNorm2d` for regularization and normalization. The `nn.Dropout` layer is used to prevent overfitting by randomly setting a fraction of input units to zero during training.

3.6 Dataset & DataLoader Class

There are the following problems with the traditional way of loading data in PyTorch:

- Loading the entire dataset into memory at once can be inefficient and may not fit into memory for large datasets.

- Data augmentation techniques like random cropping, flipping, and rotation are often needed to improve model generalization.
- Shuffling the dataset is important to ensure that the model does not learn any unintended patterns from the order of the data.
- Batching the data with efficient parallelism is necessary to speed up training and make better use of hardware resources.

These problems are addressed by the `Dataset` and `DataLoader` classes in PyTorch. The `Dataset` class is an abstract class that represents a dataset and provides methods for accessing individual data samples. The `DataLoader` class is responsible for loading data from a `Dataset` and providing it in batches, with support for shuffling, parallel loading, and data augmentation.

3.6.1 Dataset Class

The `Dataset` class is essentially a blueprint. When we create a custom `Dataset`, we decide how data is loaded and returned. It defines:

- `__init__()` method: Initializes the dataset, loads data from files, and performs any necessary preprocessing.
- `__len__()` method: Returns the total number of samples in the dataset.
- `__getitem__(index)` method: Returns a single sample from the dataset at the specified index. This is where we can apply data transformations or augmentations.

```
import torch
from torch.utils.data import Dataset

class CustomDataset(Dataset):
    def __init__(self, data, labels, transform=None):
        self.data = data
        self.labels = labels
        self.transform = transform

    def __len__(self):
        return self.data.shape[0] # Return the total number of samples

    def __getitem__(self, index):
        sample = self.data[index]
        label = self.labels[index]
        if self.transform:
            sample = self.transform(sample) # Apply any transformations
            if provided
        return sample, label
```

Here, `data` is the input and `labels` are the corresponding labels/Output for the input data. The `transform` parameter is optional and can be used to apply any transformations to the data samples, such as normalization or data augmentation.

3.6.2 DataLoader Class

The DataLoader class is responsible for loading data from a Dataset and providing it in batches. At the start of each epoch, the DataLoader (if shuffle=True) shuffles indices (using a sampler). It divides the indices into chunks of batch size. For each index in the chunk, data samples are fetched from the Dataset object. The samples are then collected and combined into a batch (using collate_fn). The batch is returned to the main training loop.

We don't need to write class for DataLoader, as it is already provided by PyTorch. We can use it directly to load data from our custom Dataset as following:

```
from torch.utils.data import DataLoader
# Create an instance of the custom dataset
dataset = CustomDataset(data, labels, transform=transform)

# Create a DataLoader instance
dataloader = DataLoader(dataset, batch_size=32, shuffle=True,
                        num_workers=4)

for epoch in range(num_epochs):
    # Iterate through the DataLoader
    for batch_data, batch_labels in dataloader:
        # Perform training or evaluation on the batch
        pass
```

4 RISC-V

RISC-V is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles. The specification for this set of instructions is the 5th generation of RISC processors, which have been in development since the 1980s, thus we call it RISC-V. RISC-V ecosystem consists of following elements:

- Physical hardware: Processors, development boards, System-on-Chips (SoCs), System-on-Modules (SoMs), and other physical systems.
- “Soft” IP processor cores that can be loaded into emulators, field-programmable gate arrays (FPGAs), or implemented in silicon.
- The entire software stack, from bootloaders and firmware, up to full operating systems and applications.
- Educational material including courseware, curricula, lesson plans, online courses like this one, tutorials, podcasts, lab assignments, and even books.
- Services including verification, custom board design, and many more.

4.1 RISC vs. CISC: Instruction Sets and Code Density

RISC-V follows the RISC (Reduced Instruction Set Computer) philosophy, which contrasts with the CISC (Complex Instruction Set Computer) approach used in architectures like x86. A key distinction between these approaches involves two separate concepts that are often confused:

- **Instruction Set Size:** The number of unique instruction types defined by the architecture
- **Instruction Count in Programs:** The number of instruction instances needed to implement a specific task

CISC architectures typically have larger instruction sets (more unique instructions) where most of which have access to memory but require fewer instructions to implement a given program. For example, Intel’s 80386 introduced in 1985 supported over 150 distinct instructions.

In contrast, RISC architectures like RISC-V have smaller instruction sets (fewer unique instructions) with memory access restricted to a few Load and Store instructions but may require more instructions to implement the same functionality. The RISC-V base integer instruction set includes only 40 instructions.

To illustrate this difference, consider a simple operation of adding a value from memory to a register:

CISC Approach	RISC Approach
ADD REG, [ADDR] (One instruction)	LOAD REG2, [ADDR] ADD REG1, REG1, REG2 (Two instructions)

This design choice in RISC architectures enables simpler hardware implementations, more efficient pipelining, and often better performance despite requiring more instructions to accomplish the same tasks. The tradeoff of using more, simpler instructions instead of fewer, complex instructions has proven beneficial for most modern processor designs.

Prof. Krste Asanović and graduate students Yunsup Lee and Andrew Waterman started the RISC-V instruction set in May 2010 as part of the Parallel Computing Laboratory (Par Lab) at UC Berkeley, of which Prof. David Patterson was Director. The Par Lab was a five-year project to advance parallel computing funded by Intel and Microsoft for \$10M over 5 years, from 2008 to 2013.

4.2 RISC-V Name

The name RISC-V was chosen to represent the fifth major RISC ISA design from UC Berkeley (RISC-I [15], RISC-II [8], SOAR [21], and SPUR [11] were the first four). We also pun on the use of the Roman numeral “V” to signify “variations” and “vectors”, as support for a range of architecture research, including various data-parallel accelerators, is an explicit goal of the ISA design.

4.3 RISC-V Instruction Set Variants

RISC-V is not just one instruction set, but a family of related ISAs. The core part of each ISA is called a base integer instruction set, and there are currently four main versions:

- **RV32I** – 32-bit registers and address space ($XLEN = 32$)
- **RV64I** – 64-bit registers and address space ($XLEN = 64$)
- **RV32E** – A smaller version of RV32I, with only 16 integer registers (instead of 32), made for small microcontrollers
- **RV128I** – A future version with 128-bit registers and address space ($XLEN = 128$)

All these versions use two's-complement to represent signed integers. $XLEN$ is the term used to refer to the register width (32, 64, or 128 bits).

4.4 Shakti Processors

Resources: [https://en.wikipedia.org/wiki/SHAKTI_\(microprocessor\)](https://en.wikipedia.org/wiki/SHAKTI_(microprocessor))
<https://github.com/platformio/platform-shakti>

Shakti is an Indian-developed, open-source RISC-V processor started as an academic initiative back in 2014 by the Reconfigurable Intelligent Systems Engineering (RISE) group at IIT-Madras.

4.4.1 Shakti Processor Variants

The Shakti processor family includes several variants, each designed for different applications and performance levels. The variants are categorized into classes based on their intended use:

Variant	Description
E-class	Embedded microcontroller class (RV32IMA), no MMU
C-class	Controller-class (with MMU), runs Linux
I-class	Industrial-class, superscalar
M-class	Multicore high-performance
S-class	Server-grade
H-class	High-performance, out-of-order execution

There are experimental variants as well, such as the Shakti H-class, which is a high-performance processor with out-of-order execution capabilities.

4.5 Vega Processors

Resources: https://en.wikipedia.org/wiki/VEGA_Microprocessors

Vega processors are developed by the Centre for Development of Advanced Computing (C-DAC) in India.

An out-of-order processor is a type of processor that executes instructions in a different order than they are written in the program, as long as it maintains the correct order of execution to ensure the program works correctly.

4.5.1 Vega Processor Variants

The Vega processor family includes several variants:

Variant	Description
Vega ET1031	Embedded 32-bit processor with floating-point unit (FPU) support
Vega ET1040	Higher performance variant with memory management unit (MMU)
Vega AS1061	Security-focused variant with enhanced protection features

The Vega Series is entirely open-source and compatible with multiple toolchains, making it flexible for various development environments.

5 SystemVerilog

Refer: [Youtube Playlist](#)

EDA Playground Link for Practicing : <https://edaplayground.com/x/t7M2>

Drawback of Verilog: Not able to extensively verify the design. Missing the Corner Cases

5.1 About RTL

Refer: [Reddit Post](#)

RTL stands for Register Transfer Level, which is a design abstraction used in digital circuit design. The above Reddit post link gives the detailed RTL description, its uses and importance in VLSI design.

5.2 Verilog Vs SystemVerilog

Refer: [Reddit Post](#)

Verilog is subset of SystemVerilog. SystemVerilog is an extension of Verilog that includes additional features and capabilities, particularly for verification and design abstraction. Refer the above Reddit post for more details.

Think of SystemVerilog as C++ and Verilog as C. Everything you can write in C will work in C++, but C++ offers much more.

Vanilla Verilog is the pure, standard Verilog — without any SystemVerilog features. When someone says "vanilla Verilog", they usually mean "just Verilog, no SystemVerilog".

5.3 Data Types

Recalling for the data types in Verilog, we have `reg`, `wire`, `integer`, `time`, `real` etc. All these are 4-state data types, meaning they can take values 0, 1, X (unknown), and Z (high impedance).

In case of SystemVerilog, we have the following data types:

Data Type	States	Description
<code>logic</code>	4-state	Can be used in both procedural and continuous assignments (one at a time)
<code>bit</code>	2-state	Can only take values 0 or 1, more efficient for synthesis
<code>enum</code>	Variable	Enumerated data type allowing named constant values
<code>struct</code>	Variable	Composite data type grouping related variables
<code>typedef</code>	N/A	User-defined type declarations for code reusability

Values of `reg` can only be assigned in procedural blocks like `always` and `initial` blocks, while `wire` can only be assigned in continuous assignments using `assign` statements. Sometimes, we get confused between what to use (`wire` or `reg`) when and where. So to avoid this confusion, we can use `logic` data type in SystemVerilog, which can be used in both procedural and continuous assignments (one at a time).

Defining a variable as `logic` will let us use it in either procedural blocks (like `always` or `initial`) or continuous assignments (like `assign` statements).

Example Usage:

```
1 // SystemVerilog logic data type
2 logic [7:0] data_bus; // Can be used flexibly
3 logic      clock;     // Single bit logic
4
5 // Traditional Verilog approach
6 reg [7:0] data_reg;    // Only in procedural blocks
7 wire [7:0] data_wire; // Only in continuous assignments
```

default value of both `reg` and `logic` is X (unknown) in simulation.

5.3.1 2-State Data Types

These data types can only take values 0 or 1, making them more efficient for synthesis and simulation. Following are the 2-state data types in SystemVerilog:

Data Type	Description
bit	Unsigned single bit data type that can take values 0 or 1.
byte	An 8-bit signed data type that can take values from -128 to 127. It is a convenient way to represent small integers.
shortint	A 16-bit signed integer data type that can take values from -32768 to 32767. It is useful for representing small signed integers.
int	A 32-bit signed integer data type that can take values from -2147483648 to 2147483647. It is the default integer type in SystemVerilog.
longint	A 64-bit signed integer data type that can take values from -9223372036854775808 to 9223372036854775807. It is useful for representing large signed integers.

Important Note: What will happen if we assign x or z to these 2-state data types? If x or z values are assigned to 2-state data types, then it will be automatically converted to 0.

5.3.2 Struct Data Type

The difference between **struct** and **array** is that grouping of different data types in **struct** while **array** is grouping of same data types.

Following is the syntax for defining a struct in SystemVerilog:

```
// SystemVerilog struct data type
struct packed {
    logic [7:0] data; // 8-bit data field
    logic valid;      // Validity flag
} my_struct;         // Instance of the struct

// Accessing struct fields
my_struct.data = 8'hFF; // Assigning value to data field
my_struct.valid = 1'b1; // Assigning value to valid field
// Using struct in a module
```

Packed vs Unpacked Structs:

Above code defines a *packed* struct in SystemVerilog.

Packed structs are stored in a contiguous block of memory, with no padding between fields. This makes them more efficient for hardware representation. Good for synthesizable code or bit-level operations.

Unpacked structs, on the other hand, allow for padding and can have variable-sized fields. They are more flexible but less efficient for hardware representation. Each member may be separately stored.

Packed vs Unpacked Structs

typedef struct packed {		typedef struct {
-------------------------	--	------------------

<pre> logic [3:0] a; logic [3:0] b; } my_struct_t; +-----+-----+ bits 7:4 bits 3:0 a b +-----+-----+ </pre>	<pre> logic [3:0] a; logic [3:0] b; } my_struct_t; </pre>
---	---

Using Typedef

Typedef allows us to create new data types based on existing ones. This can help improve code readability and maintainability by giving meaningful names to complex data types.

Verilog Code using Typedef

<pre> logic [7:0] byte_t; // Here, byte_t is a variable logic [7:0] data1; logic [7:0] data1; </pre>	<pre> typedef logic [7:0] byte_t; // Define a new type byte_t byte_t data1; byte_t data2; // variables of type byte_t </pre>
--	--

Good Example

Good Example of Packed Struct Usage with Typedef

```

typedef struct packed {
    logic [3:0] a;
    logic [3:0] b;
} my_struct_t;

my_struct_t s;
initial begin
    s = 8'b10001100; // assign whole struct as 8-bit value
end

```

In the above example, the packed struct `my_struct_t` is defined with two 4-bit fields `a` and `b`. The struct can be assigned a single 8-bit value where first four bits correspond to `a` and the last four bits correspond to `b`. So here, `a= 1000 (8)` and `b=1100 (12)`.

5.3.3 Enumerated Data Types

Enumerated data types allow us to define a set of named values, which can make our code more readable and maintainable. They are particularly useful for representing states or modes in our design.

Enumerated Data Type Example

```

typedef enum {IDLE, RUNNING, DONE} state_t; // Define an enumerated
type

```

```

state_t current_state; // Declare a variable of the enumerated type
// current_state is a variable of enum data type which can take
// values IDLE, RUNNING, or DONE
initial begin
    current_state = IDLE; // Assign a named value to the variable
end

```

5.3.4 Fixed Arrays (Packed/Unpacked)

Arrays in SystemVerilog can be one-dimensional or multi-dimensional, and they can be packed or unpacked. Packed arrays are stored in a contiguous block of memory, while unpacked arrays allow for variable-sized elements.

Packed arrays are typically used when you need to manipulate the entire array as a single unit at bit level operations, particularly for tasks like bit slicing or packing/unpacking data.

Note: Packed arrays have restrictions on the data types they can use. We cannot use the 2-state data types like `int`, `byte`, `shortint`, or `longint` with packed arrays. Packed arrays can only be used with data types like `logic`, `bit`, and `enum`.

Example of Packed Array: Following is the syntax for one-dimensional two bit packed array:

```
bit [1:0] array;
```

Note: Here, we don't need to use the keyword `packed` explicitly like we did for structs.

For multi-dimensional packed arrays, the data will be stored in contiguous memory locations only like in linear one dimensional fashion. The syntax for a two-dimensional packed array is as follows:

```
bit [1:0][2:0] array;
```

In unpacked arrays, the array dimensions are mentioned after the variable name and they can be of any data type like `int`, `byte`, `shortint`, or `longint`.

Example of Unpacked Array:

```
int array [1:0][2:0];
```

This defines a two-dimensional unpacked array where the first dimension has 2 elements and the second dimension has 3 elements. The data type of the elements is `int`.

```
bit [1:0] packed_array [3:0];
```

Here, we have defined a packed array named `packed_array` with 4 elements, each of which is 2 bits wide. The elements can be accessed using indices like `packed_array[0]`, `packed_array[1]`, etc. It is packed along one dimension that is `[1:0]` is continuous memory locations but unpacked along the other dimension that is `[3:0]` is not con-

tinuous memory locations.

Assigning Values to Arrays:

Assigning Values to Array

<pre>int unpacked_array [3:0]; // 4 int elements packed_array[0] = 2'b01; packed_array[1] = 2'b10; packed_array[2] = 2'b11; packed_array[3] = 2'b00;</pre>		<pre>int unpacked_array [3:0]; unpacked_array = '{10,20,30,40}; // Use ' to assign values // directly to the entire array</pre>
--	--	--

In the above example, using apostrophe (') allows us to assign values directly to the entire array in one go. This is particularly useful for initializing arrays with known values. This will give `unpacked_array[0] = 40` `unpacked_array[1] = 30` and so on.

Accessing Array Elements:

To access all elements of an array, we can use a `foreach` loop in SystemVerilog. This loop iterates over each element of the array, allowing us to perform operations on them. This is similar to the `for` loop in C/C++. We can also use `for` loop to access the elements of the array.

Accessing Array Elements using foreach

```
foreach (packed_array[i]) begin  
    // Access each element of packed_array  
    $display("Element %0d: %b", i, packed_array[i]);  
end  
  
// Using for loop to access elements  
for (int i = 0; i < $size(packed_array); i++) begin  
    $display("Element %0d: %b", i, packed_array[i]);  
end
```

Note: The `foreach` loop is particularly useful for iterating over arrays with dynamic sizes, as it automatically adjusts to the size of the array.

Packed Array Example:

Packed Array Example

```
module example;  
    bit [2:0][3:0][7:0] data;  
  
    initial  
        begin  
            data = 96'hffff_ffef_ffef_aaaa_aaaa_bbbb_bbbb;  
            $display("the value of data is: %b", data);  
  
            // Display array values  
            foreach (data[i])  
                begin
```

```

        $display("data[%0d] = %b", i, data[i]);
    foreach (data[i][j])
        begin
            $display("data[%0d] [%0d] = %b", i, j, data[i][j]);
        end
    end
end
endmodule

```

Output:

```

the value of data is: 1111111111011111111111111111011111010101010101010101010101010101110111011101110111011
data[2] = 111111111101111111111111111101111
data[2][3] = 11111111
data[2][2] = 11101111
data[2][1] = 11111111
data[2][0] = 11101111
data[1] = 10101010101010101010101010101010
data[1][3] = 10101010
data[1][2] = 10101010
data[1][1] = 10101010
data[1][0] = 10101010
data[0] = 10111011101110111011101110111011
data[0][3] = 10111011
data[0][2] = 10111011
data[0][1] = 10111011
data[0][0] = 10111011

```

5.3.5 Dynamic Arrays

Compile Time vs Runtime

Compile-time refers to the phase in the development process when the source code is translated into machine code (or intermediate code) by a compiler. This phase happens before the program is run. The goal of compile-time activities is to check and transform the code so that it is ready for execution.

In HDLs like SystemVerilog, compile-time refers to the time when the compiler checks for the syntax, and the HDL code is converted into a netlist or hardware structure before execution of program.

Runtime is the phase when the compiled code is executed on a computer or hardware. During runtime, the program is running, and it can perform operations, access memory, produce output and interact with the user or other systems.

Dynamic arrays are a type of array in SystemVerilog that can change size during simulation. They are particularly useful when the size of the array is not known at compile time and can be adjusted based on runtime conditions.

In case of Fixed Arrays, the size of the array is fixed at compile time, meaning it cannot be changed during runtime. Dynamic array do not have a fixed size at compile-time. The size is not known or set during compilation, and no memory is allocated for the array at this stage.

To declare a dynamic array, you use the following syntax:

```
int dynamic_array [];
```

In this example, we declare a dynamic array of integers. The size of the array can be set at runtime using the `new()` method:

```
dynamic_array = new[10]; // Create
                        an array with 10 elements
```

Dynamic arrays can also be resized using the `resize()` method:

```
dynamic_array.resize(20); // Resize
                        the array to 20 elements
```

Copying the elements of Dynamic Array:

Without explicitly creating the memory for the second dynamic array, we can copy the elements of one dynamic array to another just by using the following syntax:

```
int dynamic_array2[] = dynamic_array;
// Copy elements to another dynamic array
```

If we make any changes to `dynamic_array`, it will not affect `dynamic_array2`. SystemVerilog creates an independent copy of the array. Both arrays will have separate memory locations, and changes to one array will not affect the other.

Increasing the size of Dynamic Array:

There are two methods to increase the size of a dynamic array in SystemVerilog:

- The existing elements will be deleted and size will be increased.
- The existing elements will remain as it is and size will be increased.

Resizing Dynamic Array

<pre>dyn1 = new[20]; // Existing elements will be deleted</pre>	<pre> dyn1 = new[20](dyn1); // Existing elements will remain // Size of dyn1 will be increased to 20</pre>
---	---

Lets say the initial size of `dyn1` is 10, then after resizing it to 20, the first method will delete the existing 10 elements and create a new array of size 20. The second method will keep the existing 10 elements and increase the size to 20, leaving the last 10 elements uninitialized (default value is 0).

Builtin Functions for Dynamic Arrays:

- `size()` - Returns the number of elements in the dynamic array.
- `delete()` - Deletes the dynamic array and frees up memory.
- `push_back()` - Adds an element to the end of the dynamic array.
- `pop_back()` - Removes the last element from the dynamic array.

[Here](#) is the link of good Dynamic Array example code.

Dynamic Array Example:

5.3.6 Queue

In SystemVerilog, a queue is a variable-size, ordered collection of homogeneous elements (all elements are of the same type). Unlike static arrays and dynamic arrays, the size of a queue can change dynamically during runtime as elements are added or removed. Queues provide a powerful, flexible data structure that operates like a FIFO (First In, First Out) or LIFO (Last In, First Out) mechanism depending on how you manipulate the elements.

In dynamic arrays, we have to explicitly create the memory for the array using the `new()` method before inserting the elements, but in queues, we don't have to do that. Memory is allocated in the queue automatically when we add elements to them.

Key Characteristics:

- **Dynamic Size:** Queues can grow or shrink as needed as elements are deleted or inserted, allowing for flexible memory usage.
- **Operations:** Queues support various operations such as inserting, deleting elements to the front or back, and querying the queue size.
- **Homogeneous Elements:** All elements in a queue must be of the same data type.
- **FIFO Behavior:** Elements are typically added to the end and removed from the front, following a first-in-first-out order.

Here is the general syntax for declaring a queue in SystemVerilog:

```
data_type queue_name [$];
```

For example, to declare a queue of integers, you would use:

```
int my_queue [$];
```

This creates a queue named `my_queue` that can hold integers.

Builtin Functions

- `size()` - Returns the number of elements in the queue.
- `insert(index, value)` - Inserts a value at the specified index in the queue.
- `delete(index)` - Deletes the value at specified index. In case if index is not mentioned, it deletes all elements in the queue and frees up memory.
- `push_back(value)` - Adds an element to the end of the queue.
- `push_front(value)` - Adds an element to the front of the queue.
- `pop_front()` - Removes and returns the first element from the queue.
- `pop_back()` - Removes and returns the last element from the queue.

5.3.7 Associative Arrays

As we have discussed earlier about dynamic arrays and fixed arrays, there we have allocate the fixed memory space either in the compile time or at the runtime irrespective of the fact that we are using that memory or not. For example, if we declare a dynamic array of size 100, then 100 memory locations will be allocated for that array even if we are using only 10 elements in that array, remaining 90 elements are just waste of memory.

Associative arrays provide a way to store data where the index or key doesn't need to be an integer (can also be a string) and doesn't need to be consecutive, unlike dynamic arrays or fixed-size arrays. This makes them similar to dictionaries or maps in other programming languages. They are very useful when the index values are sparse or irregular, allowing flexible and dynamic data storage.

Following is the syntax for declaring an associative array in SystemVerilog:

```
data_type associative_array_name [key_type];
```

key_type can be of any data type, including integers, strings, or enumerated types. * (Wildcard type) index type inferred at first use. data_type is the data type of the elements stored in the associative array.

For example, to declare an associative array of integers with string keys, you would use:

```
module example;
    int marks[string];
    initial begin
        marks["Alice"] = 85;
        marks["Bob"] = 90;
        marks["Charlie"] = 78;

        $display("Alice's marks: %0d", marks["Alice"]);
        $display("Bob's marks: %0d", marks["Bob"]);
        $display("Charlie's marks: %0d", marks["Charlie"]);
    end
endmodule
```

Builtin Functions

- num() - Returns the number of elements currently stored in the associative array.
- first(var) - Returns the first key in the associative array and stores it in the variable var.
- last(var) - Returns the last key in the associative array and stores it in the variable var.
- next(key) - Returns the next key after the specified key in the associative array.
- prev(key) - Returns the previous key before the specified key in the associative array.
- delete(key) - Deletes the element associated with the specified key.
- exists(key) - Checks if the specified key exists in the associative array.

Example using next()

Example using next()

```
module associative_array();

    int fruits[string];
    initial begin
```

```

fruits= '{"apple":6,"orange":2, "guava":3, "watermelon": 9,"grape":1}';
begin
  string f; // Default value of f would be lowest index i.e. "apple"
  while(fruits.next(f))
    $display(" Next fruit of is [%s] = %0d",f,fruits[f]);
    //It'll stop as it's not cyclic in nature unlike enum
  end
end
endmodule

```

Output:

```

““
Next fruit of is [apple] = 6
Next fruit of is [grape] = 1
Next fruit of is [guava] = 3
Next fruit of is [orange] = 2
Next fruit of is [watermelon] = 9
““

```

5.3.8 Summary

ARRAY TYPE	DESCRIPTION	APPLICABLE METHODS
DYNAMIC ARRAYS	Arrays with a variable size, determined at runtime.	size(), delete(index), sort(), reverse(), shuffle(), min(), max(), find_with(), unique()
Associative Arrays	Arrays indexed by an arbitrary integral type (e.g., int, string)	num(), exists(index), delete(index), delete() (no argument), first(var), last(var), next(var), prev(var)
Queues	Ordered, variable-sized lists supporting push and pop operations	size(), push_front(item), push_back(item), pop_front(), pop_back(), insert(index, item), delete(index), sort(), reverse(), shuffle(), min(), max(), find_with(), unique()

Method	Description	Applicable Arrays
<code>size()</code>	Returns the number of elements.	Dynamic Arrays, Queues
<code>insert(index, item)</code>	Inserts an item at a specific index.	Queues
<code>delete(index)</code>	Removes an element at a specific index.	Dynamic Arrays, Queues, Associative
<code>delete()</code>	Deletes all elements.	Dynamic Arrays, Associative Arrays
<code>pop_front()</code>	Removes and returns the first element.	Queues
<code>pop_back()</code>	Removes and returns the last element.	Queues
<code>push_front(item)</code>	Adds an element to the front of a queue.	Queues
<code>push_back(item)</code>	Adds an element to the back of a queue.	Queues
<code>num()</code>	Returns the number of elements.	Associative Arrays
<code>first(var)</code>	Returns the first index.	Associative Arrays
<code>last(var)</code>	Returns the last index.	Associative Arrays
<code>next(var)</code>	Returns the next index after the given index.	Associative Arrays
<code>prev(var)</code>	Returns the previous index before the given index.	Associative Arrays
<code>exists(index)</code>	Checks if an index exists.	Associative Arrays
<code>sort()</code>	Sorts elements in ascending order (custom order possible).	Dynamic Arrays, Queues
<code>reverse()</code>	Reverses the order of elements.	Dynamic Arrays, Queues
<code>shuffle()</code>	Randomly shuffles elements.	Dynamic Arrays, Queues
<code>min()</code>	Returns the smallest element.	Dynamic Arrays, Queues
<code>max()</code>	Returns the largest element.	Dynamic Arrays, Queues
<code>find_with()</code>	Finds the first element that satisfies a condition.	Dynamic Arrays, Queues

5.4 Display Output

In SystemVerilog, we can use the `$display` function to print output to the console/Terminal. The `$display` function is similar to the `printf` function in C/C++. It allows us to format the output and display variables, strings, and other data types.

5.4.1 Format Specifiers

The following table shows the format specifiers used with `$display`/`$write`/`$monitor`/`$fwrite` in SystemVerilog:

Specifier	Meaning	Example Output
%d	Decimal (signed integer)	42
%0d	Decimal with zero-padding (optional field width)	00042 (if width is given)
%b	Binary	1010
%h	Hexadecimal	A
%o	Octal	12
%c	Character	A
%s	String	hello
%p	Aggregate (arrays, structs, etc.)	{10, 20, 30}
%t	Time (used in simulations)	#100
%%	Prints a literal %	%

Note: The `%p` format specifier is particularly useful for printing dynamic arrays, queues, associative arrays, and struct data types without needing to write explicit loops to display each element.

5.4.2 Zero-Padding/Space-Padding

In SystemVerilog, we can also specify the width of the output and whether to use zero-padding or space-padding. The following table summarizes the different formats:

Format	Name	Example Output (for <code>x = 7</code>)
%0d	Zero-padding (no width \rightarrow acts like <code>%d</code>)	7 (<i>no padding seen</i>)
%04d	Zero-padding with width	0007
%4d	Space-padding (right-aligned)	7 (<i>3 spaces before 7</i>)
%-4d	Left-aligned space-padding	7 (<i>3 spaces after 7</i>)

5.4.3 System Task

Anything in verilog starting with `$` is called a system task. System tasks are used to perform various operations such as displaying output, reading input, and controlling simulation time. Some commonly used system tasks are:

- `$display`: Used to display output to the console.
- `$write`: Similar to `$display`, but does not add a newline at the end of the output.
- `$strobe`:
- `$monitor`: Used to monitor changes in signals and display output when a change occurs.

- `$finish`: Used to terminate the simulation.
- `$stop`: Used to pause the simulation.
- `$time`: Returns the current simulation time.

5.5 Tasks & Functions

In SystemVerilog, tasks and functions are procedural blocks that allows for executing a series of statements. Tasks are particularly useful when you need to group code that performs operations involving multiple inputs/outputs or when your code involves time-consuming operations (such as delays, waiting for an event, etc.). Tasks can be reused by calling them multiple times, helping to make your design more modular and easier to maintain.

Function is a subroutine that encapsulates reusable code and returns a single value. Functions are primarily used to perform calculations or operations that don't require simulation time, meaning they cannot contain any timing controls such as delays or event triggers.

5.5.1 Enhancements in SystemVerilog:

- In Verilog, code inside task is guarded by `begin` and `end` keywords, while in SystemVerilog, no need to write these keywords.
- Function can only have input arguments in Verilog, while in SystemVerilog, it can have both input, inout and output arguments.
- We can't have void function in Verilog, while in SystemVerilog, we can have void function.

Note: SystemVerilog doesn't always support C-like inline initialization of local variables in task and function blocks — depending on tool and context. So it's safer to split declaration and assignment.

Example: `int i = 10 + 20;` inside the task/function block will return `i=0`.

5.5.2 Types of Tasks

There are two types of tasks in SystemVerilog:

- **Static Tasks:** A static task in SystemVerilog is a task that does not maintain any state between invocations and shares its local variables across all calls. Static tasks are instantiated once and their variables persist between calls, so changes made to variables in one invocation are visible in other invocations.

When the static task block is invoked/called for the first time, a memory space is allocated for the task and all member variables will be initialized. When it is called again, the same memory space is used and the modification will be done in the same variables only.

By default, all tasks in SystemVerilog are static tasks. This means that if you don't specify otherwise, the task will behave as a static task.

- **Automatic Tasks:** A automatic task in SystemVerilog is a task that maintains its own state between invocations and does not share its local variables across calls. Automatic tasks are instantiated each time they are called, and their variables are initialized anew for each invocation.

When the automatic task block is invoked/called for the first time, a memory space is allocated for the task and all member variables will be initialized. The memory space is freed up when the task execution is completed.

Everytime the automatic task is called, a new memory space is allocated for the block of code and all member variables will be initialized again. This means that the automatic tasks does not retain any state between calls.

keyword `automatic` is used to declare a task as automatic.

Important Note: Static Task is better when the task is never recursive, never called in parallel, and performance is key, since automatic memory allocation has overhead. Automatic Task is preferred in case of multithreaded, or recursive use cases.

5.5.3 Syntax of Task

Static Task Syntax

```
module task_example;

    int a;
    task increase();
        a=a+1;
    endtask

    task automatic increment();
        int i;
        i = i+1; // can't do int i = i+1;
        $display("The value of a after automatic increment is %0d", i);
    endtask

    initial begin
        increase();
        increment();
        $display("The value of a after 1st increment is %0d", a);

        increase();
        increment();
        $display("The value of a after 2nd increment is %0d", a);
    end

endmodule
```

5.5.4 Passing arguments to Tasks

In SystemVerilog, we can pass arguments to tasks in the following ways:

- Argument passing by name

```
task example_task(int a, int b);
    $display("The value of a is %0d and b is %0d", a, b);
endtask
initial begin
    example_task(.a(5), .b(10)); // Passing arguments by name
end
```

- Argument passing by value

```
task example_task(int a, int b);
    $display("The value of a is %0d and b is %0d", a, b);
endtask
initial begin
    int x = 5;
    int y = 10;
    example_task(x, y); // Passing arguments by value
end
```

- Argument passing by reference

Keyword **ref** is used to pass arguments by reference. This means that the task can modify the original variables passed to it, and those changes will be reflected outside the task.

The task must be declared **automatic** to use reference arguments. This is because passing by reference implies that the task or function can be re-entrant, meaning it can be called multiple times concurrently (or recursively), and each call needs its own independent set of local variables.

```
task automatic example_task(ref int a, ref int b);
    a = a + 1;
    b = b + 2;
    $display("The value of a is %0d and b is %0d", a, b);
endtask

initial begin
    int x = 5;
    int y = 10;
    example_task(x, y); // Passing arguments by reference
    $display("The value of x after task call is %0d and y is %0d", x, y);
end
```

Default Arguments

In SystemVerilog, we can also specify default values for task arguments. This allows us to call the task without providing all the arguments, and the default values will be used for any missing arguments.

Default Arguments in Task

```
module task_example;

    task example_task(int a = 5, int b = 10);
        $display("The value of a is %0d and b is %0d", a, b);
    endtask

    initial begin
        example_task(); // Using default values
        example_task( , 20); // Using default value for b, leaving blank
                             // for a
        example_task(30, 40); // Using custom values
    end
endmodule
```

5.6 Interface

In SystemVerilog, an interface is a construct that allows you to group related signals and variables together, providing a way to define a common communication protocol between different modules.

Lets say we are using the same set of signals in multiple modules, then instead of declaring the same signals in each module, we can declare them in an interface and use that interface in all the modules. This helps to reduce code duplication and makes the design more modular and easier to maintain.

5.6.1 Syntax of Interface

Interface Syntax

```
interface my_if();
    int a;
    int b;
endinterface

// Here, I have declared the interface of two signals. This means that
// if I want to use these signals in any module then I can just create a
// instance of this inferface and include it directly.

module add(my_if intf);
    initial begin
        $display("sum is %0d", intf.a + intf.b);
    end
endmodule
```

```

module sub(my_if intf);
    initial begin
        $display("sum is %0d", intf.a - intf.b);
    end
endmodule

module operation;
    my_if intf();
    // my_if is the interface name
    // intf is the interface instance

    initial begin
        intf.a= 3;
        intf.b= 4;
    end

    add adder(.intf(intf));
    sub subtractor(.intf(intf));
endmodule

```

We can pass any argument to the interface instance, it will be helpful in putting conditioning for the inputs signals. For example, if we want the signals to change with clock edge, then we can pass the clock signal to the interface instance and use it in the modules to modify the signals.

Interface with Clock Signal

```

interface my_if(input logic clk);
    logic [7:0] data;
    logic valid;

    // Clocked process to update data and valid signals
    always_ff @(posedge clk) begin
        data <= data + 1; // Increment data on clock edge
        valid <= (data < 255); // Set valid based on data value
    end
endinterface

```

5.6.2 Modports

Modports (short for module ports) are a feature in SystemVerilog interfaces that allow you to define different access permissions rules (read/write directions) for the signals within the interface. It tells which signals are inputs, outputs, or inout for a specific module using that interface.

When multiple modules connect through the same interface, they may use the interface differently:

- One module might need to write to a signal, while another module might only

need to read from it.

- One module might drive the signals (output). Another module might read the same signals (input).

So, we use modport to define how each module can access the interface signals.

Modports in Interface

```
interface inf();
    int a;
    int b;
    int c;
    int d;

    modport master(input a, input b, output c);
    modport slave(input a, input b, input c, output d);
endinterface

module add_two(inf.master infc);
    assign infc.c = infc.a + infc.b;
endmodule

module add_three(inf.slave infc);
    assign infc.d = infc.a + infc.b + infc.c;
endmodule

module example_mud();
    inf my_inf();

    initial begin
        my_inf.a = 1;
        my_inf.b = 2;
    end

    add_two add1(my_inf); // same as add_two add1(.inf (my_inf))
    add_three add2(my_inf);

    initial begin
        #1;
        $display("The value of a and b is %0d, %0d",my_inf.c, my_inf.d);
    end
endmodule
```

5.7 Blocking & Non-Blocking Assignments

In Verilog, there are two types of assignments: blocking and non-blocking.

- **Blocking assignments** (using the '=' operator) are executed sequentially. The next statement will not execute until the current statement is complete. This is similar to how assignments work in most programming languages.

- **Non-blocking assignments** (using the '`=>`' operator) allow the next statement to execute without waiting for the current statement to complete. This is useful in sequential logic, where you want to model flip-flops and other memory elements.

5.7.1 Delay-Based Timing Control

In Verilog, we can use delay-based timing control to specify the time delay for the execution of a statement. This is done using the '`#`' symbol followed by the time delay value. This can be done in two ways:

- **Inter Assignment Delay:** This is used to specify the time delay for the execution of a statement. The statement will be executed after the specified time delay. The delay value is specified on the left hand side of the expression.
- **Intra Assignment Delay:** This is used to specify the time delay for the execution of a statement within a blocking assignment. The RHS to LHS assignment happens after the specified time delay. The delay value is specified on the right hand side of the assignment operator.

For both cases, the delay is specified in time units (e.g., nanoseconds, picoseconds, etc.) which is mentioned in the timescale directive at the top of the Verilog file.

Example-1:

Delay-Based Timing Control

<code>// Inter</code>		<code>// Intra</code>
<code>initial begin</code>		<code>initial begin</code>
<code> #10 b = 1;</code>		<code> c = #10 a + b;</code>
<code> #5 c = 5;</code>		<code> d = #5 3;</code>
<code>end</code>		<code>end</code>

In the above example, in case of inter assignment delay, the value of `b` will be assigned after 10 time units, and the value of `c` will be assigned after 15 time units (10 + 5). In case of intra assignment delay, the value of `c` will be assigned after 10 time units but the value of `a + b` will be evaluated at time = 0 only. The value of `d` will be assigned after 15 time units (10 + 5).

Example-2:

Inter Assignment Delay in Non-Blocking Assignments

<code>// Case-1</code>		<code>// Case-2</code>		<code>// Case-3</code>
<code>initial begin</code>		<code>initial begin</code>		<code>initial begin</code>
<code> a <= 0;</code>		<code> a <= 1;</code>		<code> a <= 1;</code>
<code> b <= 1;</code>		<code> #5 b <= a + 1;</code>		<code> b <= #5 a + 1;</code>
<code> c <= 2;</code>		<code> #10 c <= 3;</code>		<code> c <= #10 3;</code>
<code> d <= 3;</code>		<code> #15 d <= 4;</code>		<code> d <= #15 4;</code>
<code>end</code>		<code>end</code>		<code>end</code>

Case-1: This is simple non-blocking assignment execution. All the assignments will be executed in parallel and the values of **a**, **b**, **c**, and **d** will be 0, 1, 2, and 3 respectively at time $t = 0$.

Case-2: Even though we are using the non-blocking assignments, this will be executed similar to blocking assignments because of inter assignment delays. The assignments to **b**, **c**, and **d** have delays specified. The value of **b** will be assigned after 5 time units, **c** after 15 time units, and **d** after 30 (5+10+15) time units. The values of **a**, **b**, **c**, and **d** will still be 1, 2, 3, and 4 respectively after the execution of the initial block.

Case-3: In this case, the assignments to **b**, **c**, and **d** have delays specified. The value of **b** will be assigned after 5 time units, **c** after 10 time units, and **d** after 15 time units. However, the values of **a**, **b**, **c**, and **d** will be 1, 1, 3, and 4 respectively. Here, the value of **b** is different from Case-2 assigned after 5 time units, but the value of $a + 1$ is evaluated at time $t = 0$ only. So, at time $t=0$, the value of **a** = 0, so the value of **b** will become 1.

Example-3:

Intra Assignment Delay in Non-Blocking Assignments

```
initial begin
    a <= 0;
    b <= #5 1;
    c = #10 2;
    d <= #3 3;
    e <= #5 4;
end
```

Here, the value of **a**, **b**, **c**, will be assigned after 0, 5, 10 time units respectively. The **c** statement will block the execution of further statements until it is executed completely. So, the value of **d** and **e** will be assigned after 13 time units (10 + 3) and 15 time units (10 + 5) respectively.

5.8 Event Scheduler

[Event Scheduler Paper](#)

5.8.1 Simulation Time & Time Slot

In SystemVerilog, simulation time is divided into discrete time slots. Each time slot represents a specific point in time during the simulation. Time slot = all events processed at a single simulation time.

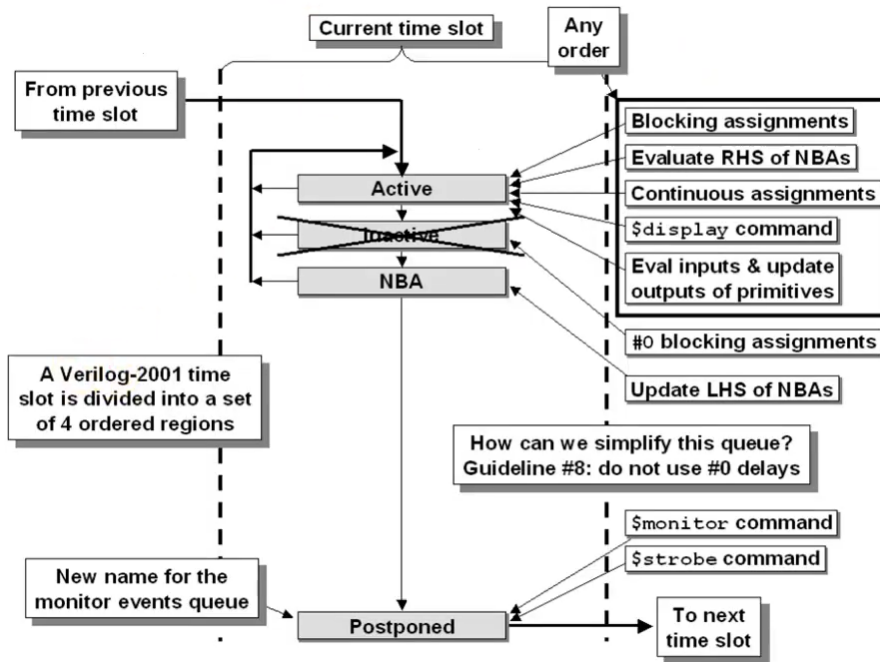


Figure 1: Event Scheduler in Verilog

To improve the the racing conditions, SystemVerilog introduces five more evnet regions in each time slot as shown below:

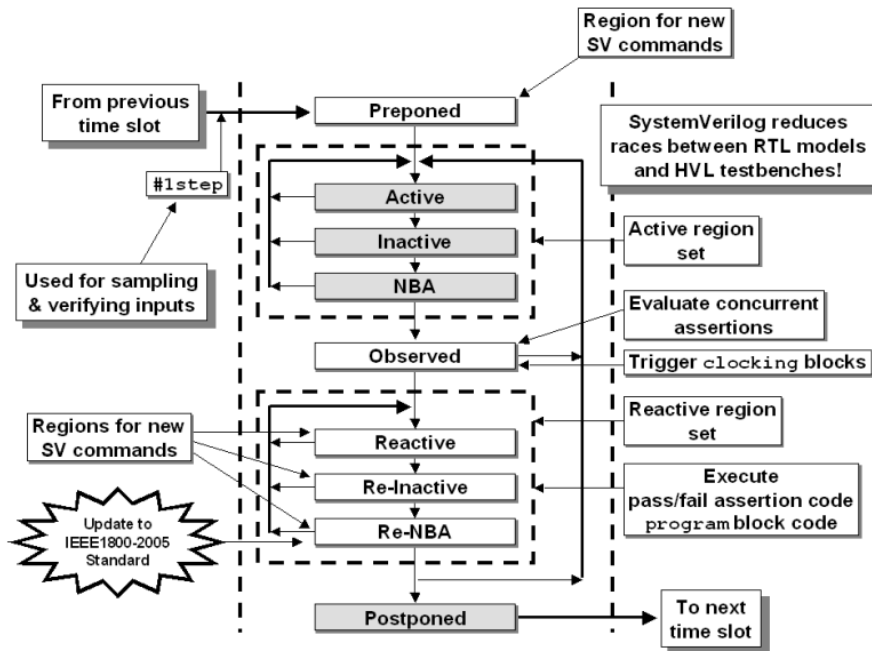


Figure 2: Event Scheduler in SystemVerilog

Simulation time moves in time slots, but within a time slot, a lot happens in steps (Preponed → Active → Inactive → NBA → Observed...).

Preponed Region

The Preponed region happens first in a time slot, right after advancing simulation time, before anything else happens.

The Preponed region is read-only, meaning it doesn't change any signal values. Its main purpose is to capture (sample) signal values for assertions before any updates or evaluations start.

This allows the testbench to capture the last known values before any design changes are applied.

Active Events Region

The principal function of this region is to evaluate and execute all current module activity. There is no role of testbench in this region.

- Execute all `module` blocking assignments.
- Evaluate the Right-Hand-Side (RHS) of all nonblocking assignments and schedule updates into the NBA region.
- Execute all `module` continuous assignments
- Evaluate inputs and update outputs of Verilog primitives.
- Execute the `$display` and `$finish` commands.

Inactive Events Region

[ChatGPT Chat](#) for more details.

Non-Blocking Assignments (NBA) Region

The principal function of this region is to execute the updates to the LeftHand-Side (LHS) variables that were scheduled in the Active region for all currently executing nonblocking assignments.

Observed Region

The principal function of this region is to evaluate the concurrent assertions using the values sampled in the Preponed region.

Reactive Region Set

This is a set dual of the corresponding Active regions. The Reactive region is used to run the testbench code, which is not part of the design under test (DUT), allowing the testbench to respond to the design events that happened earlier in the same time slot.

Postponed Region

The principal function of this region is to execute the `$strobe` and `$monitor` commands that will show the final updated values for the current time slot.

6 CVA6

References: [GitHub Repository](#) & [Setup Tutorial](#)

Official Documentations : [CVA6 User Manual](#) & [Main Theory](#)

CVA6 GitHub Lab : [Hands-on-Experience with CVA6](#)

CVA6 stands for CORE-V version of Ariane Core, which is a RISC-V based open-source processor core developed by the OpenHW Group. It is a 6-stage, single-issue, in-order CPU which implements the 64-bit RISC-V instruction set.

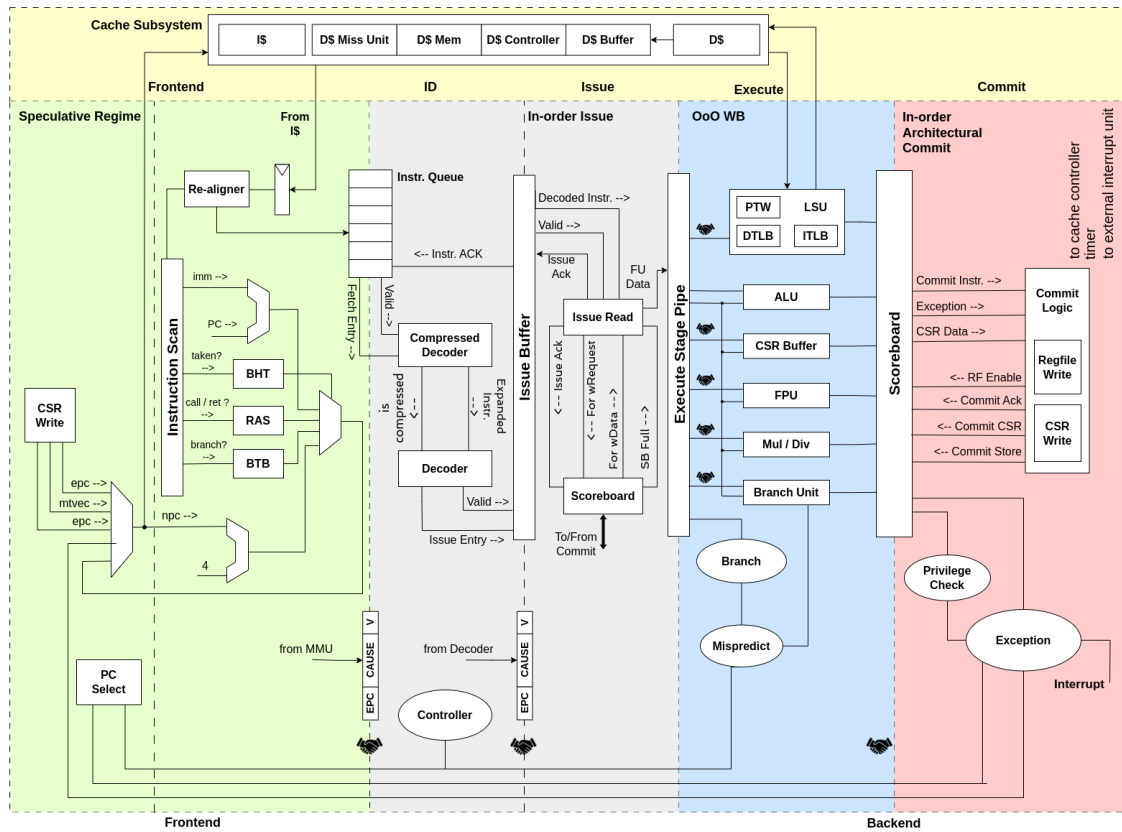


Figure 3: CVA6 Architecture

Note: CVA6 is an in-order processor, which is true in the sense that instruction issue and commit happen in program order. However, its write-back and execution complete out-of-order. Click [Here](#) for details (ChatGPT chat with sources).

CVA6 supports I, M, A, and C extensions in RISC-V. [Extracted details from Manual \(GPT\)](#) & [RISC-V Instruction Set Manual](#)

It implements three privilege levels M, S, U to fully support a Unix-like operating system. [GPT chat](#) for more details.

6.1 Setting up CVA6

To set up CVA6, you need to follow these steps:

1. Fork the CVA6 repository and then clone from GitHub.
Don't clone directly from the main repository, as it will not allow you to push changes (contribute back) in the future.
2. Set upstream for pull the latest update

```
git remote add upstream https://github.com/openhwgroup/cva6.git
git remote -v
```

3. Initialize and update all submodules:

```
git submodule update --init --recursive
```

Running this command ensures all submodules inside `.gitmodules`—down to the deepest nested level—are initialized, cloned, and checked out to the exact commit expected by the main project. It's required to build and simulate complex repos like CVA6 smoothly. [Link](#)

4. Install cmake using following command:

```
sudo apt install cmake
cmake --version
```

5. Building the compiler Toolchain ([Steps](#))

[Why & What Toolchain?](#)

Several standard packages are required to build the compiler toolchain. You can install them using the following command:

```
sudo apt-get install autoconf automake autotools-dev curl git
libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex
texinfo gperf libtool bc zlib1g-dev
```

Under the `cva6/util/toolchain-builder/config`, we have different toolchain configurations for different RISC-V targets.

To build the specific toolchain, you can run the following command:

```
bash get-toolchain.sh gcc-13.1.0-baremetal
```

The important step here is to set the configuration file

<https://chatgpt.com/share/685e9b22-605c-8005-ad4d-3c5d990b4081>

```
export RISCV=$HOME/riscv
export NUM_JOBS= 8
./build-toolchain.sh -f gcc-13.1.0-baremetal $RISCV
```

To check if it is installed correctly, you can run the following command:

```
$HOME/riscv/bin/riscv-none-elf-gcc --version
OR
home/vedant/riscv/bin/riscv-none-elf-gcc --version
```

6. Installing the Verilator and Spike

```
export DV_SIMULATORS=veri-testharness,spike

bash veril/regress/smoke-tests-cv64a6_imafdc_sv39.sh
```

Under CVA6/tools, we can clearly check two folders, spike and verilator-v5.008
Run this to set them in path:

```
export
    PATH=$(pwd)/tools/verilator-v5.008/bin:$(pwd)/tools/spike/bin:$PATH

export
    PATH=/home/vedant/Desktop/Summer_Project/cva6/tools/spike/bin:$PATH
export
    PATH=/home/vedant/Desktop/Summer_Project/cva6/tools/verilator-v5.008/bin:$PATH
verilator --version
spike --version
```

Then we can check the verilator and spike version.

7. To run a C program, we can use the following code:

```
source verif/sim/setup-env.sh
# run the above command in CVA6 directory

cd verif/sim

python3 cva6.py \
    --target cv64a6_imafdc_sv39 \
    --iss=veri-testharness \
    --iss_yaml=cva6.yaml \
    --c_tests ../tests/custom/hello_world/hello_world.c \
    --linker=../../config/gen_from_riscv_config/linker/link.ld \
    --gcc_opts="-static -mmodel=medany -fvisibility=hidden -nostdlib \
    -nostartfiles -g ../tests/custom/common/syscalls.c \
    ../tests/custom/common/crt.S -lgcc \
    -I../tests/custom/env -I../tests/custom/common"
```

```
export RISCVC=$HOME/riscv
export RISCVC_CC=/home/vedant/riscv/bin/riscv-none-elf-gcc
export SPIKE_PATH=/home/vedant/Desktop/Summer_Project/cva6/tools/spike/bin
```

Before running the below commands, run the above lines.
For converting .c file into .elf file:

```
export PATH=$HOME/riscv/bin:$PATH
riscv-none-elf-gcc --version
riscv-none-elf-gcc -o hello1.elf hello.c
# run this command where hello.c is present
```

```
/home/vedant/riscv/bin/riscv-none-elf-gcc \
verif/tests/custom/mytest/hello.c \
-T config/gen_from_riscv_config/linker/link.ld \
```

```
-static -mcmmodel=medany -fvisibility=hidden -nostdlib -nostartfiles \  
-g verif/tests/custom/common/syscalls.c verif/tests/custom/common/crt.S -lgcc \  
-Iverif/tests/custom/env -Iverif/tests/custom/common -Iverif/env/p \  
-o verif/sim/out_2025-07-03/directed_tests/hello.elf \  
-march=rv64gc -mabi=lp64
```

Run the above command from CVA6 repo

https://chatgpt.com/s/t_6862946b5dbc81919ee5c69cf3d0d9a6

Run all the setup here

```
export RISCV=$HOME/riscv  
source setup-env.sh # run this in CVA6/verif/sim  
export NUM_JOBS=8  
export DV_SIMULATORS=veri-testharness,spike  
  
python3 cva6.py \  
--target cv64a6_imafdc_sv39 \  
--iss=$DV_SIMULATORS \  
--iss_yaml=cva6.yaml \  
--c_tests ../tests/custom/hello_world/hello_world.c \  
--linker=../../config/gen_from_riscv_config/linker/link.ld \  
--gcc_opts="-static -mcmmodel=medany -fvisibility=hidden -nostdlib \  
-nostartfiles -g ../tests/custom/common/syscalls.c \  
../tests/custom/common/crt.S -lgcc \  
-I../tests/custom/env -I../tests/custom/common"
```

For waveform generation, use the following code:

[GitHub Official Instructions](#)

```
export TRACE_FAST=1  
export TRACE_COMPACT=1
```

6.2 Stages in CVA6

There are 6 stages in CVA6, which are as follows:

- **PC generation Stage:** The Program Counter (PC) is generated to point to the next instruction to be executed.
- **Instruction Fetch Stage:** The instruction is fetched from memory.
- **Instruction Decode Stage:** The fetched instruction is decoded to determine the operation.
- **Issue Stage:** The instruction is issued to the appropriate execution unit.
- **Execute Stage:** The operation is performed (e.g., ALU operations, memory access).
- **Commit Stage:** The instruction is committed, and the next instruction is fetched.

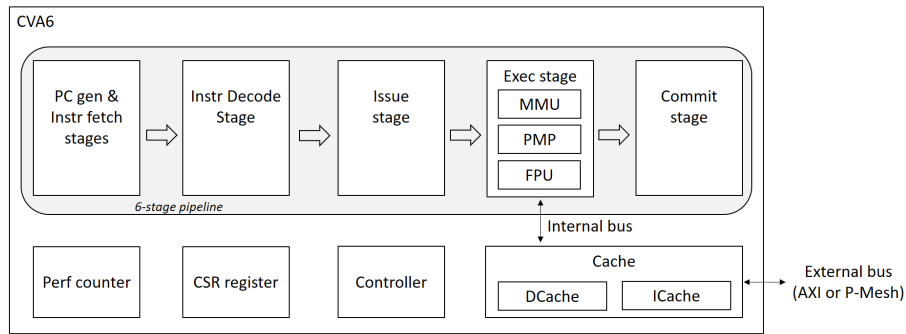


Figure 4: Stages in CVA6

6.2.1 PC generation Stage

The Instruction Cache (\$I) has 1 cycle access latency on hit while Data Cache (\$D) has longer latency of 3 cycles access latency on hit.

This stage is part of the frontend which includes branch prediction structures like BTB (Branch Target Buffer) and BHT (Branch History Table). The PC is generated based on the current PC value, branch prediction, and instruction fetch.

Control and Status Register (CSR)

These are the special registers which tell the CPU where to jump for special events (errors, OS calls, debug) by providing the addresses for exception/interrupt handling, environment call returns, and debug entry. and where to return after handling them, ensuring correct control flow in all situations (ensuring correct PC selection during exceptional control flow events).

6.2.2 Instruction Fetch Stage

In this stage, the instruction is fetched from the instruction cache (\$I) based on the generated PC based on the handshake between the Instruction Fetch Unit (IFU) and the Instruction Cache.

The handshake is a classic "valid/ready" protocol:

- The frontend sets `icache_dreq_o.req` high when it wants to fetch an instruction.
- The instruction cache sets `icache_dreq_i.ready` high when it can accept a fetch request.
- Only when both signals are high in the same cycle does the fetch actually happen.

Frontend wants to fetch:

- `instr_queue_ready = 1` (queue has space)
- `halt_frontend_i = 0` (not halted)

→ `icache_dreq_o.req = 1`

Instruction cache is ready:

- `icache_dreq_i.ready = 1`

```
assign icache_dreq_o.req = instr_queue_ready & ~halt_frontend_i;
```

```
assign if_ready = icache_dreq_i.ready & instr_queue_ready & ~halt_frontend_i;
```

- `instr_queue_ready` ensures the instruction queue can accept new instructions.
- `halt_frontend_i` can block fetching (e.g., during a fence or halt).

Fetch occurs:

- `if_ready = 1`
- The PC is updated, and the instruction cache returns the instruction data.
- If either side is not ready, the fetch is stalled until both are ready.

Why do we need an Instruction Cache and an Instruction Queue?

Instruction Cache (I\$)

It stores recently fetched instructions close to the CPU for fast access and reduces the need to access slow main memory for every instruction fetch.

Memory is much slower than the CPU. The cache bridges this speed gap, allowing the CPU to fetch instructions quickly and keep the pipeline full.

Instruction Queue

It buffers instructions fetched from the cache before they are sent to the decode stage and decouples the fetch stage from the decode/issue stages.

The instruction fetch and decode/issue stages may not always run at the same speed: The cache might deliver multiple instructions at once (burst fetch). The decode/issue stage might stall (e.g., due to pipeline hazards). The queue allows the frontend to keep fetching instructions even if the backend is temporarily stalled, improving throughput and hiding memory/cache latency.

Branch Prediction

Branch prediction data in CVA6 consists of:

- **BTB (Branch Target Buffer):** Remembers the target addresses of recently seen branches.
- **BHT (Branch History Table):** Remembers the recent outcomes (taken/not taken) of branches, usually using a small counter.
- **Index:** Subset of bits from the PC used to select (index into) a particular entry in the BTB/BHT tables. It provides the fast lookup into these tables using part of the PC.
- **Tag:** Subset of higher-order bits from the PC stored with each entry to help distinguish between different branches that might map to the same table entry (to avoid aliasing).

Note/Fact: The last bit(s) of the PC is always zero because when fetching instructions, the hardware fetches in words (32 bits = 4 bytes). This means the PC used for fetching is always aligned to 4 bytes, so the lowest two bits are zero. Even if the instruction is compressed (16 bits), the PC is still aligned to 2 bytes, so the lowest bit is zero.

6.2.3 Instruction Decode Stage

Because of the compressed instructions, this stage gets little complicated. It consists of three main components:

- **Instruction Re-aligner:** Handles compressed (16-bit) and regular (32-bit)

instructions alignment across cache line boundaries.

- **Compressed Decoder:** raw 32-bit instruction words are transformed into executable control structures called scoreboard entries
- **Decoder** Contains the CPU's registers, which are used to store intermediate values during execution.

6.3 Understanding Spike & Verilator

Spike is the official RISC-V instruction set simulator (ISS) developed by UC Berkeley and maintained by the RISC-V community. It is referred as a golden reference software model for RISC-V processors.

Note: It is pre-written C++ code that implements RISC-V instructions functionally. It is not a simulator in the sense of simulating hardware, but rather a software model that executes RISC-V instructions.

It executes RISC-V binaries (.ELF) according to the ISA spec. Spike simulates an idealized RISC-V processor that follows only the ISA specification - it doesn't know or care about CVA6's specific design choices.

Spike is like a "reference answer key" for what any RISC-V processor should do functionally, while Verilator shows what YOUR specific CVA6 processor actually does!

6.4 Scoreboard

Scoreboard is a hardware structure that tracks which instructions are in-flight (issued but not yet completed), which registers are being written, and which instructions are waiting for data.

Memory (data RAM/cache) accesses can take several cycles. If the CPU waits (stalls) every time it needs data from memory, performance drops. If an instruction is waiting for data (e.g., a load from memory), but there are other instructions that do not depend on that data, the CPU can execute those independent instructions while waiting. This helps in hiding latency.

6.5 Model on Hardware

Step by step instruction

grep is a powerful command-line tool used in Unix-like operating systems, such as Linux and macOS, for searching text files based on patterns. It stands for "global regular expression print". grep searches for lines in files that match a specified pattern (which can be a simple string or a more complex regular expression) and then prints those matching lines to the console.

The RISC-V testing infrastructure (like Spike or the CVA6 test framework) checks the special memory-mapped variable tohost. This value is used to signal success or failure of the program to the simulator. When program exits, the runtime writes the return value to tohost. If it is zero means success else failure.

6.5.1 Tests in CVA6

There are following tests like:

- **run-asm-tests:** Runs all RISC-V assembly tests (from ci/riscv-asm-tests.list).
- **run-amo-tests:** Runs atomic memory operation tests.
- **run-mul-tests:** Runs multiplication instruction tests.
- **run-fp-tests:** Runs floating-point instruction tests.
- **run-benchmarks:** Runs benchmark programs.

Each of these targets will simulate a suite of tests using the test lists defined in the ci/ directory.

To run a specific test, you can use the following command:

```
make run-mul-tests
```

Log File Format:

```
1 core 0: 0x0000000000010000 (0x00100413) li s0, 1
2 3 0x0000000000010000 (0x00100413) x 8 0x0000000000000001
3 core 0: 0x0000000000010004 (0x01f41413) slli s0, s0, 31
4 3 0x0000000000010004 (0x01f41413) x 8 0x0000000000000000
5 core 0: 0x0000000000010008 (0xf1402573) csrrs a0, mhartid, zero
6 3 0x0000000000010008 (0xf1402573) x10 0x0000000000000000
7 core 0: 0x000000000001000c (0x00000597) auipc al, 0x0
8 3 0x000000000001000c (0x00000597) x11 0x000000000001000c
9 core 0: 0x0000000000010010 (0x07458593) addi al, al, 116
10 3 0x0000000000010010 (0x07458593) x11 0x0000000000010080
11 core 0: 0x0000000000010014 (0x00040067) jr s0
12 3 0x0000000000010014 (0x00040067)
```

Figure 5: CVA6 Log File Instructions

Each instruction is logged in two parts:

- **Instruction Decode Line:**

```
core 0: 0x0000000000010000 (0x00100413) li s0, 1
```

This shows that core 0 executed the instruction `li s0, 1` at address `0x0000000000010000` (PC Value), which has 32-bit RISC-V machine code (actual instruction) in Hex (`0x00100413`) stored at that address.

- **Writeback Line (if register is written):**

```
3 0x0000000000010000 (0x00100413) x 8 0x0000000000000001
```

Here, the number 3 indicates Machine mode, register `x8` was updated with value `0x1`. PC and Instruction is repeated. There are other modes like 0 for User mode and 1 for Supervisor mode. The instruction was executed in Machine Mode, the highest privilege level in RISC-V.

Note: `li s0, 1` → which is a pseudoinstruction for: `addi s0, zero, 1`

6.6 Performance Modelling

6.6.1 RVFI Trace

RVFI (RISC-V Formal Interface) trace is a standardized way to capture and analyze the execution of RISC-V instructions in a simulation environment. It records:

- Instruction execution
- Register updates
- Memory accesses
- Program Counter (PC) values
- Timing information

RVFI Analyser

This is a script written by copilot to analyse the RVFI trace file generated by the CVA6 simulator. To run the script, You can run the script using the following command:

```
python3 rvfi_analyser.py <rvfi_trace_file>
```

Example:

```
python3 rvfi_analyzer.py \
verif/sim/out_2025-07-02/veri-testharness_sim/hello_world.cv64a6_imafdc_sv39.log
```

There are four files for analysing log files:

- **rvfi_analyzer.py**: Instruction type distribution, proper log instruction printing
- **simple_rvfi_analyzer.py**: Simple analysis of RVFI traces
- **analysis.py**: Subset result of model.py. RVFI analysis for issue=commit=2
- **demo_issue_commit.py**: Issue and Commit width analysis with Hazard and Improvement Analysis. Varies the issue and commit width, put the data for each of them
- **model.py**: issue/commit matrix

Complete Execution Flow

This is python script to completely analyse the instructions

```
python3 perf_analysis.py
../verif/sim/out_2025-07-08/veri-testharness_sim/multiply.cv64a6_imafdc_sv39.log
--performance --save-terminal-output corrected_test.txt --colors
```

Getting the Scoreboard Result

For printing the scoreboard debug, run the following:

```
python3 debug_model.py
../verif/sim/out_2025-07-08/veri-testharness_sim/multiply.cv64a6_imafdc_sv39.log
```

This will generate the output file `scoreboard_debug_<filename>.log`

7 PULP-TrainLib

8 UVM

Reference: [Playlist1](#) [Playlist2](#) are the links to the UVM playlists.

9 On-Device Training

9.0.1 Subsection

Introduce about the Title here.

Reference: <https://www.youtube.com/watch?v=ic1UMeuCBA8>
[GeeksforGeeks](#)

- 1:
- 2:

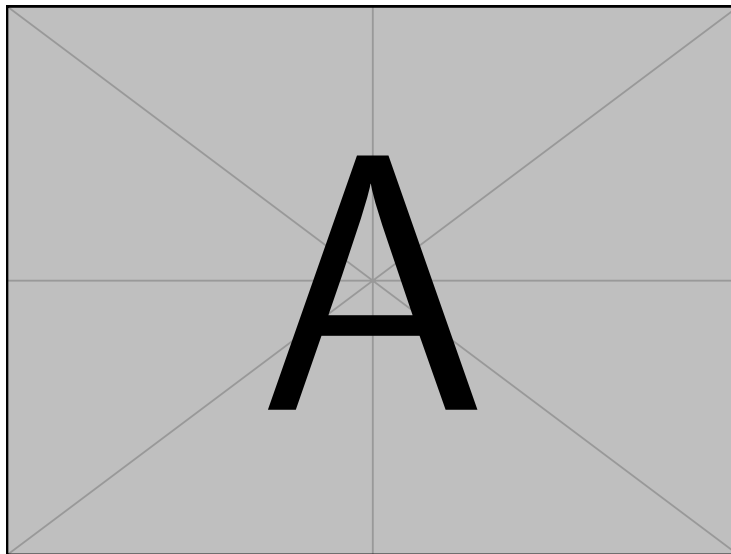


Figure 6: Sample Image

Verilog is a Case Sensitive language.

The term “module” refers to the text enclosed by the keyword pair **module ... endmodule**.

Module is the fundamental descriptive unit in Verilog language.

Keyword “module” is followed by the name of the design (ABC here) and parenthesis
- enclosed list of ports.