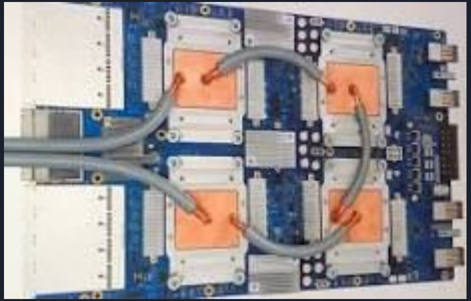
CPU Presentation

Extra Learning
I have a copy if necessary.
Memory presentation
Google form questions
Gpu presentation
Git tutorial
OS document

Hardware interaction

The reality is that when you want to make a computer faster at a certain specific task then you can reliably assume that designing the hardware to be optimized specifically for your need it will be faster than if you just implemented it with software. For that reason the OS and the hardware interact with each other in specific ways to speed up certain necessary algorithms such as threading and other things.

TPU (tensor processing unit) A processor built specifically for handling tensors which are common in AI->



Operating System

An Operating System(OS) is the software that handles your essential functions of your computer. After the computer has run through its essential booting (Powering on the computer and running the POST test) the OS is loaded and is run until the computer is turned off.

The main part of the operating system is the kernel. The kernel is responsible for most of the absolutely essential functions the operating system handles.

There are three main functionalities of an operating system. The first is to handle all instances of code that your computer wants to run. The second is handling specific hardware constructs to speed up the running of said code. The third involves all things related to files.

A simple way to use an os is to let the hardware speak for itself. What that means is that you let the os tell the cpu what block of code to run and then let it run it. The way it does that is through processes.

Do note there are more way a computer can be architected.

Processes

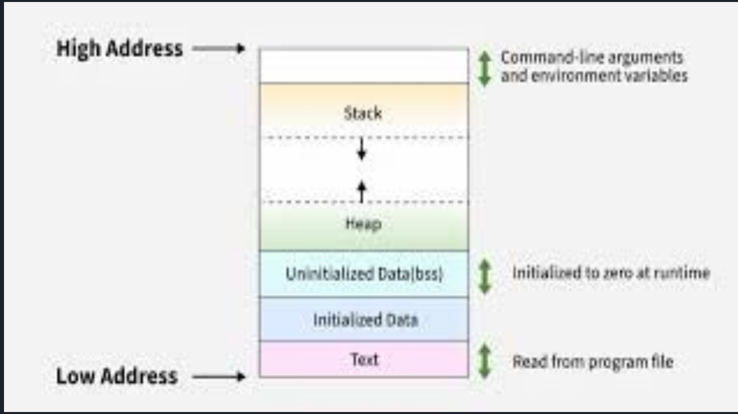
A process is an instance of a program.

A program is a collection of executable files (sometimes .exe, sometimes others) that make up anything software on your computer from a browser, to an antivirus software, to a flight simulator.

A program is a memory element until it is called upon. When that happens, it turns into a process which is like a memory element that is being used. Processes are os managed and come with a bunch of necessary elements that are built into a process control block(pcb). You can think of a process control block as a data structure that holds a list of necessary information for a process.

Process sections/pages

Pages are stored in page tables, page tables are tables of all the pages with their memory locations on the RAM/ROM. Pages are usually divided up into sections for variables, heap, stack, and the actual code. Divisions like this make it easy to separate out all the code over memory so they don't have to be 1 continuous block. This is called non-contiguous memory space. Sometimes a process can be stored on physical memory, like RAM, and needs Virtual memory on the hard drive. The Virtual memory (or VM) is controlled by the OS, make sure you remember the distinction.



Process Life

An OS evaluates processes based on how they can be scheduled via lifecycles. A typical 5 stage starts with a process being loaded into memory. This is called the “new” state.

In this state a pcb, process control block, is created. A pcb can be thought of as a list of essential information that needs to be created.

When the process is loaded into memory it changes to the “ready” state. In this state the process is ready to be run at a moments notice.

There are not always the same number of stages on every os. They match the case you are using them for.

Process Life Ext.

A process changes into the running state when the process begins being executed by the cpu.

Sometimes a process runs into an instruction that requires information that is not already in any of the processes memory segments. When this happens a process requests extra information and the process goes into the blocked state.

When the process finishes running then it turns to the “terminated state”.

<u>There are not always the same number of stages on every os. They match the case you are using them for.</u>

Process Scheduling

The process of choosing which process to run is called process scheduling. There are two subtypes of scheduling.

Non preemptive scheduling is a method in which the CPU allows a process to fully complete itself while on the CPU. This includes waiting for I/O. This method is extremely slow, inefficient, and can lead to serious security problems such as malware taking up the CPU for extended time.

Preemptive scheduling allows the CPU to switch between different processes, making sure a security problem such as in non preemptive scheduling won't happen. When a process is ran on the CPU, a special timer starts counting, and if it reaches a certain value without the process needing I/O or other interrupts, then the process has its state saved to its allocated memory, and a new process is ran. If a process does require I/O before the timer reaches its value, then the process is saved while waiting for the I/O, and a new process takes time on the CPU. This allows for your CPU to constantly be working with as little breaks as possible.

Process Scheduling factors

The CPU has many many processes that are all trying to use the CPU, se we need to manage them and put them on a schedule.

The factors to consider when making a schedule are:

Arrival time: When will the process arrive to use the CPU (the ready queue)?

Completion time: How long until that process completes?

Burst time: The time required by a process for the CPU.

Turn around Time: The time completion and arrival time.

Waiting Time: The time between Turnaround and Burst.

Process scheduling factors

When we create our algorithm, our goal is to keep the CPU busy at all time. This is called CPU Utilization.

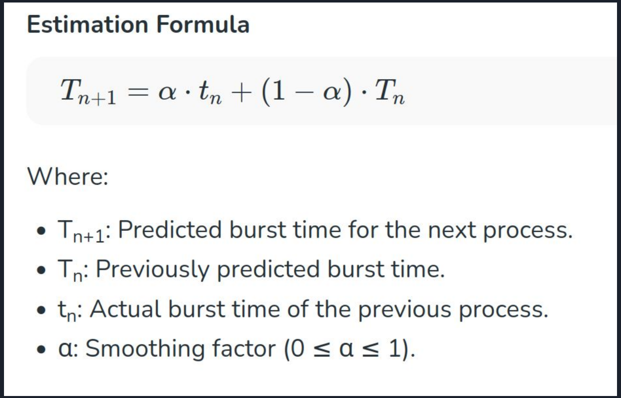
Another factor to consider here is Throughput, or, the average amount of processes the CPU can complete within 1 unit of time.

Furthermore, Waiting time. The algorithm hast to minimize each processes time waiting in the ready queue.

And response time, basically how long it takes for the process to be submitted to returning an output.

Shortest Job FIrst(SJF) / Shortest Job Next(SJN)

This is a Nonpreemptive system where it selects the process with the shortest wait time and allows that to be executed on the CPU. SJF might be preemptive or not, depending. SJF determines the time on the PCU or the “Burst Time” either by the process telling the OS, or a special formula.



Estimation Formula

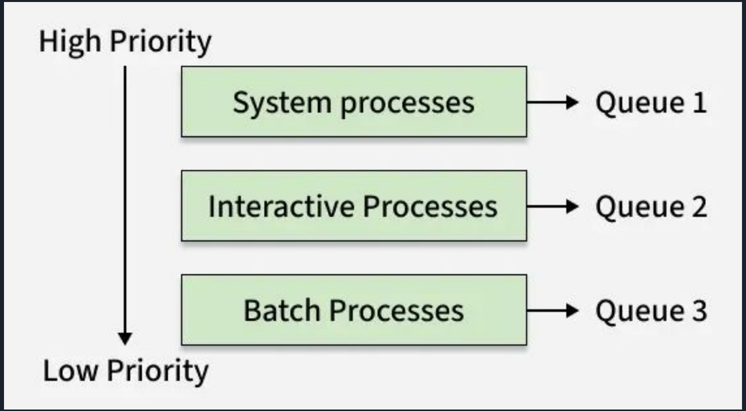
$$T_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot T_n$$

Where:

- T_{n+1}: Predicted burst time for the next process.
- T_n: Previously predicted burst time.
- t_n: Actual burst time of the previous process.
- α: Smoothing factor ($0 \leq \alpha \leq 1$).

MultiLevel Feedback Queue Scheduling(MLQ)

MLQ is very interesting, this system uses multiple queues based on a processes priority. Lower priority programs have lower queues than high priority processes. HPP can interrupt LPP. Each queue has authority over the lower queues. The benefits are that it's much more organized, and it's easy to administrate. However, Low processes still are at risk of getting starved, multiple queues require more resources, and certain queues might dominate CPU time.



Priority Scheduling

Priority Scheduling is a very good type of CPU scheduling, where in each process is given a priority number, with 0-1 or 0-3451, the number is arbitrary. When a process with a priority of lets say 6 enters the ready queue, it surpasses all the processes with a priority numbers lower than theirs.

One big issue with this, is as the number of higher priority processes increase, the less CPU time lower priority processes get, leading to what is called “starvation”, as they are starved for CPU time. A solution to this would be called “aging”, where the more time in the ready queue a process has, the higher their priority number grows.

The priority number is 1st allocated based on a few factors, memory usage, time to execute, and other I/O,

Round Robin

Round Robin is one of the most efficient CPU scheduling methods. The way it works is for each process in the ready queue, once they are running on the CPU, there is a timer that goes off, if the timer reaches 0 and the process has not hit an IO call, then it will be interrupted, snapshotted, and sent to the back of the ready queue.

The reason for this is so 1 process does not eat up all of the CPU’s time and energy. Processes might do this if they are poorly programmed or malware.

System Interrupts

System Interrupts are whenever your OS detects something urgent has occurred. Your mouse moving on the screen, a data transfer completing, or even specific errors.

These interrupts all processed by the Interrupt Vector Table(IVT) that is simply there to direct each interrupt to their specific handling function in the Interrupt Service Routine(ISR) table, where each function has their own specific function to handle it. Each function is ran by the CPU on special wires called IRQ lines that are made for those specific Interrupts. These are physical lines in the CPU and processed by a small chip. A more modern way of this is by using the PCI BUS to transfer the interrupts to the CPU interrupt manager instead of dedicated lines.

Types of interrupts

There are 3 major interrupts:

Faults: This is when the CPU tries to access non existing or inaccessible memory.

Page faults, dividing by 0 errors, and protection faults.

Trap: These are basically just whenever the system detects a system call or a program needs a breakpoint in the code.

Abort: These are the worst, they signal that there is something seriously wrong, either to the physical hardware or something else. These are when the system aborts the process or aborts itself to protect itself.

Other types of interrupts, while still important to the CPU, are more of what I have said, about your mouse moving, or your keyboard being pressed, etc.

Dispatch Latency

When a process changes from process to process it needs to perform a context switch. Context switching is the “saving” of process where its pcb (process control block) is saved and a different process is loaded in. A process control block contains essential information such as registers and the process id.

Dispatch latency is the time it takes for a process to begin context switching and then have the fresh process in ready state. The dispatch latency is ideally as low as possible as no real utility is had during a context switch.

User mode v Kernel mode

If you know anything about an OS you might have hear about the Kernel. What does it do though? The kernel is kind of like the middle man between the user and the hardware, usually requiring a password, it protects the user from highly sensitive and critical information and devices. The kernel talks to the actual hardware, for example, if you wanted to save some text to a file, you would send what is called a system call to the Kernel. A system call is just when you asking the kernel to do something for you. The kernel gets this message, analysis it for stuff such as if you have the ability to save files, and if the request is possible, etc. Then, it takes all your data and turns it into binary, then pushes it to the drive, finds the fils, and makes the changes.

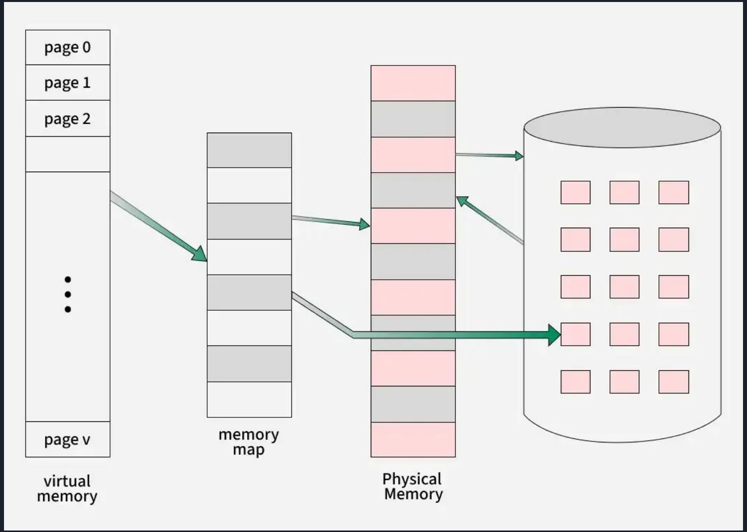
VIrtual memory

When you want more memory than the RAM can offer you use virtual memory. We find space on then the disk(hard drive, nvme, ssd, ect). Virtual memory directs new data to the hard drive if no space on the RAM/ROM is available. This is very very very slow, up to 1000x times slower than RAM/ROM. Virtual memory on the disk are called page file sor swap spaces.

Virtual memory works by creating a memory map of that specific process, where all the data is. It holds all the pointers to the stack, the heap, and the disk if needed.

RAM/ROM is physical memory, while the disk is usually virtual memory

When a component that a process requests is not there a page miss is activated and it triggers the OS to look for it on disk.



Done with the slides for the meeting

CPU Essential information

ADI

Some types of CPU have an instruction cycle with 3 stages. It is called a fetch-decode-execute cycle. A cpu can only execute one process at a time.

The fetch reads an instruction from the process.

The decode stage takes the fetched instruction and tells it where to go.

The execute stage is where the instruction is actually executed. This can be something being sent to the alu, a jump or something else.

There are other stages that may exist in some types of cpus

CPU Fetch/Decode/Execute/Store steps

ADI

The steps the CPU follows to execute instructions from memory are Fetching the instruction from RAM using the programs program counter.
The program counter (PC), called instruction counter in intel x86, it is a special register in the CPU that hold the memory address of the next instruction to be executed by the CPU for a specific process.
Then the instruction is properly decoded into signals sent to different components in the CPU, such as the ALU of FPU.
Then the instruction is actually ran, the signals are actually sent.
Then the data is stored written to the process storage space.

The instruction is very much not like something you would program in c. The instruction is machine code in this state. The machine code consists of an opcode and a couple of other things which differentiates the types of instructions such as an add, subtract and others.

Once decoding is done the instruction is sent to its appropriate place to be handled.

Changing the program counter

ADI

The program counter is set by a jmp. The jmp sends a value to the program counter that it is set to. By doing this you can change the program counter to be something else.

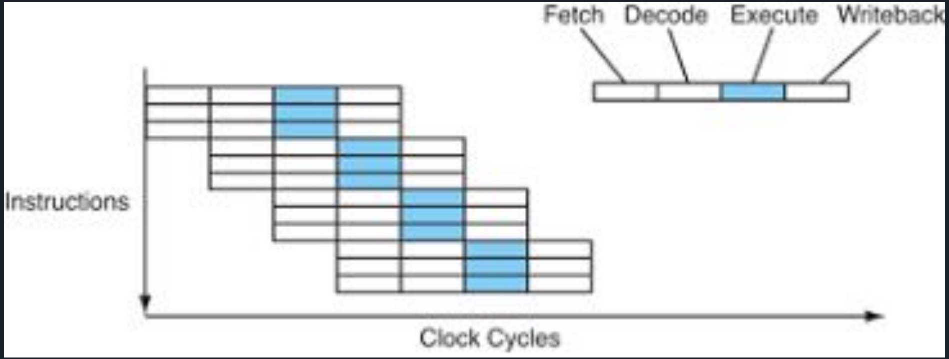
When you don’t set a program counter to an entirely different value then the computer just adds enough to see the next instruction. Some architectures dedicate this operation to a whole step in the CPU cycle called increment.

Parallelism

NIKESH

The 3 basic steps of the instruction cycle get run and then wait for the other steps to run before being run again. What this means is that the fetch step will start and then it will have to wait for both the decoding and executing to be done before it is run again. This is slow. A way to fix this is to have all three steps run at the same time. That mostly means the fetch stage, the decode stage and the execute stage would run at the same time.

The image shows a different number of steps in the instruction cycle and a different architecture. The point still stands though.



Threads

NIKESH

A thread is a process in the same address space as another process. When running multiple threads you should know that threads run depending on what the os scheduler wants. A main problem that comes with that is that you could share one essential variable and have both processes modify that necessary variable. (Race condition).

You can fix that by restricting the scheduling of threads so that only one thread is running within a program. This is called locking (it is also sometimes known as a mutex).

<u>There are problems involving threads that we don't talk about. Know that threading is a very complex idea and a short explanation does not give it all the attention it deserves.</u>

Threads ext.

NIKESH

Do note that most times the complicated nature of threads requires hardware support. Most modern day threads have numerous algorithms such as load-linked, store-conditional, fetch-and-add, yielding and many more.

Compiler

A compiler is what translates the code you write into code that a cpu can work with. The lowest level of code is assembly or machine code. Assembly is entirely based on what computer you are running it on. X86, arm and mips are common examples. Compilers are separated into stages.

Some compilers contain preprocessors. Preprocessors handle macros and primarily are used to make certain things known when a compiler is running. This means that the compiler will take certain things into account while it is running such as constant variables.

The first is usually the lexer. It converts the written contents into “tokens” which the compiler uses.

Compiler

The parser interprets these tokens and generates an IR (intermediate representation) representation. An IR is machine code of a program. IRs can be more or less detailed. These are compiler specific and later optimization is based off of this.

This IR is then optimized for factors such as speed, pipelining efficiency and memory use. These optimizations are then sent to a linker.

Linking is the process of taking external files such as libraries or files that you explicitly marked as for sharing between files (such as header files) and bridging them together. Linking is the slowest step of optimization. The component that does this is called a Linker.

These are generalizations. Though this number of steps is common it is not guaranteed that certain parts such as a linker may be used.

Concurrency

NIKESH

Concurrency is just the way the CPU handles programs. When multiple processes need the CPU, the CPU breaks up each part of the code of each program in the queue where the code is waiting on system calls, user input, or anything else. While the CPU is waiting, it takes a snapshot of the CPU state, and moves on to another process to run its code until that process also has to wait on a system call or user input. Then it moves onto the next process in the queue. The CPU does this very very fast, making it seem as if each of these processes are working at the same time, giving it the name.

When a process is waiting on a system call or user input, the move from the cpu to the back of the queue and waits.

IPC

NIKESH

IPC or inter process communication is a way 2 processes can interact with each other. The way they share data is by allocating a special little are on the heap, called the shared memory location, this is where they dump all data the the other process will need.

Hyper-Threading

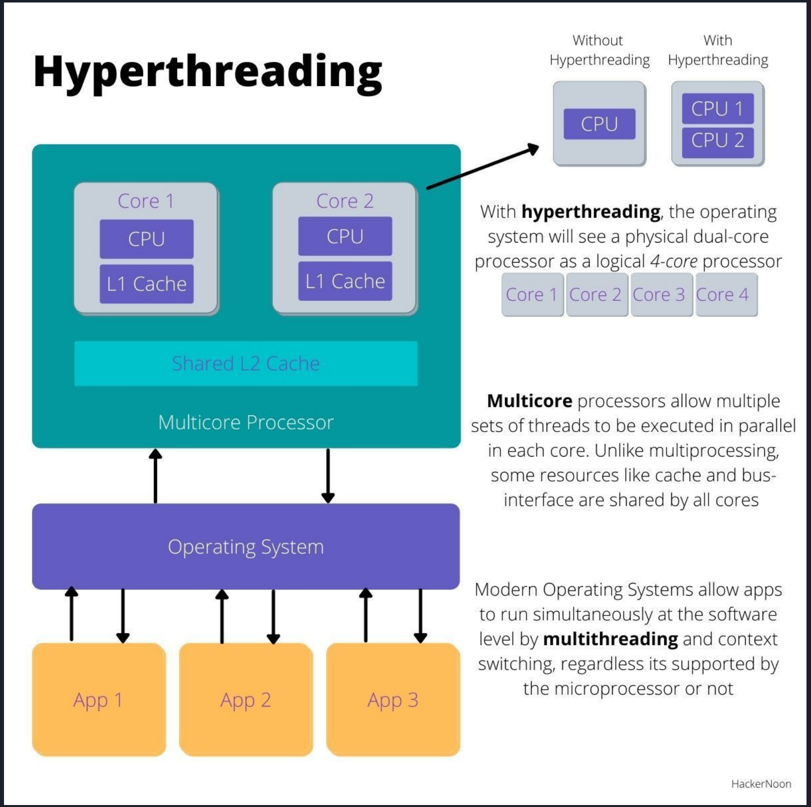
Hyper-Threading and Threads are 2 very distinct topics in the computing world.

Hyperthreading is the physical version of threading.

For example let's say there is an 8 core CPU with hyperthreading (In a factor of 2).

So there would be 8 Physical Cores, and 16 Logical Processors (Each Physical Core is a Logical Processor, but not all Logical Processors are actually Physical Cores)

Hyperthreading in a way "separates that core so that the CPU can execute things simultaneously (These "new-found cores" are often called **Logical Processors**). You can find that Hyperthreading will come in factors of 2 or more.


Hyperthreading
Without Hyperthreading
With Hyperthreading
CPU
CPU 1
CPU 2
Core 1
CPU
L1 Cache
Core 2
CPU
L1 Cache
Shared L2 Cache
Multicore Processor
With **hyperthreading**, the operating system will see a physical dual-core processor as a logical *4-core* processor
Core 1
Core 2
Core 3
Core 4
Multicore processors allow multiple sets of threads to be executed in parallel in each core. Unlike multiprocessing, some resources like cache and bus-interface are shared by all cores
Operating System
Modern Operating Systems allow apps to run simultaneously at the software level by **multithreading** and context switching, regardless its supported by the microprocessor or not
App 1
App 2
App 3
HackerNoon

Von Neumann CPU architecture

The Von Neumann architecture was created in 1945 as a way to organize CPU’s. Don't let age fool you, it's still very important and widely used.

THIS VON

Bit by Bit, Byte by Byte *(A Refresh Guide)*

b -Bit:

A bit contains either a 1 or 0

B - Byte:

8 bits = 1 Byte

kB - Kilobyte:

1024 Bytes

mB - Megabyte:

1024 Kilobytes

gB - Gigabyte:

1024 Megabytes

(I think you can start to see the pattern as we go higher from here…)

Future plans -

Marketing - first impressions (make better) , actual marketing (unsure currently)

In future - Analyse an arduino or something (not likely), possible competitions, and

Future slides - GPU, Analog, Verilog, in-depth circuits, SPICE, look at pcbs (possibly)

Outside of club obligations (projects) - Required obligations may include github and git skills. Obligations may include either a list of things specified by us which give points. When you reach enough points you no longer have to do these projects. Possible Examples include looking at the number of lines in gimple (gnu parser), installing linux, etc. Other options are a personal hardware related project which is approved by us and monitored via weekly check-ins (something like this). Final options as of this moment is helping us make content.

Building understanding of community - Doing statistics and stuff like that.

Extra learning additions - More specific textbooks, more in depth look at all topics presented in the slides, RAM error handling, Finish ASM slides, Verilog slides, SPICE slides, boolean algebra, discrete maths, calculus related to hardware, x64 guide, etc.

Todo list

1. Digital Design Projects 💻

AES Encryption/Decryption using VHDL/Verilog: Secure your DATA TRANSMISSION with encryption techniques. 🔒

ALU (Arithmetic Logic Unit) Design: Implement a 16-BIT ALU with various ARITHMETIC and LOGIC OPERATIONS. ➗✖️

CPU Design (Simple RISC Processor): Build a BASIC RISC-BASED CPU to execute instructions. 🖥️

Floating Point Unit (FPU) Design: Implement IEEE-754 floating point arithmetic for accurate NUMBER REPRESENTATION.

Digital Clock with Alarm: A fun FPGA-BASED CLOCK with LCD/LED DISPLAY and an ALARM FEATURE. ⏰

2. Verification & Testing Projects 🧪

Verification of UART Protocol using SystemVerilog & UVM: Test the UART PROTOCOL functionality and performance. 🧑‍💻

I2C/SPI Communication Protocol Verification: Verify the I2C/SPI PROTOCOLS for proper data communication. 🔄

AXI Protocol Implementation & Verification: Implement and verify the AXI PROTOCOL for HIGH-PERFORMANCE DATA TRANSFER. 🚀

Memory Controller Verification: Test interfaces like DDR3/DDR4 MEMORY CONTROLLERS for seamless memory access. 💾

3. Low-Power VLSI Design Projects ⚡

Low-Power Flip-Flop Design: Use CLOCK GATING techniques to OPTIMIZE POWER CONSUMPTION in flip-flops.

Power Optimization in FIR Filter Design: Apply APPROXIMATION TECHNIQUES to reduce POWER USAGE in FIR filters. 💡

Low-Power Multiplier Design: Implement a BOOTH'S ALGORITHM or WALLACE TREE multiplier with LOW POWER CONSUMPTION. 🔋

4. FPGA-Based Projects 🎛️

Image Processing using FPGA: Work on FACE DETECTION and EDGE DETECTION algorithms for IMAGE PROCESSING. 🖼️

Traffic Light Controller using FPGA: Design a TRAFFIC LIGHT SYSTEM based on REAL-TIME DATA inputs. 🚦

Speech Recognition System on FPGA: Create a SPEECH RECOGNITION SYSTEM to convert speech into text. 🗣️

Cryptographic Hash Function Implementation on FPGA: Implement HASHING ALGORITHMS like SHA on FPGA for secure applications. 🔐

5. Analog & Mixed-Signal VLSI Projects ⚙️

SAR ADC Design: Build a SUCCESSIVE APPROXIMATION REGISTER (SAR) ADC for ANALOG-TO-DIGITAL CONVERSION. 🧮

CMOS Operational Amplifier Design: Design a LOW-POWER CMOS OP-AMP for signal processing. 🔌

PLL (Phase-Locked Loop) Design: Develop a PLL CIRCUIT for generating STABLE CLOCK SIGNALS. ⏲️

Low Noise Amplifier (LNA) for RF APPLICATIONS: Design an LNA to amplify weak signals while minimizing NOISE. 📶

6. AI/ML-Enabled VLSI Projects 🤖

Edge AI Accelerator for TinyxML: Create a HARDWARE ACCELERATOR to run AI/ML MODELS on tiny devices. 💥

Hardware Implementation of CNNs on FPGA: Design an FPGA-BASED SYSTEM for running CONVOLUTIONAL NEURAL NETWORKS (CNNs). 🧠

Systolic Array Design for Deep Learning: Implement a SYSTOLIC ARRAY architecture to optimize DEEP LEARNING APPLICATIONS. 📊