System Architecture Of Tejas

Meenakshi Atkade

Overview

- What is a simulator?
- Why do we need a simulator?
- What is an emulator?
- System Architecture of Tejas
- Application of tejas
- Conclusion

What is a Simulator?

- A simulator is a software tool used in research to model and emulate the behavior of systems or specific components.
- Simulators replicate the functionality and performance characteristics of the target architecture, enabling researchers to run software workloads and observe system behavior.

Why do we need Simulators?

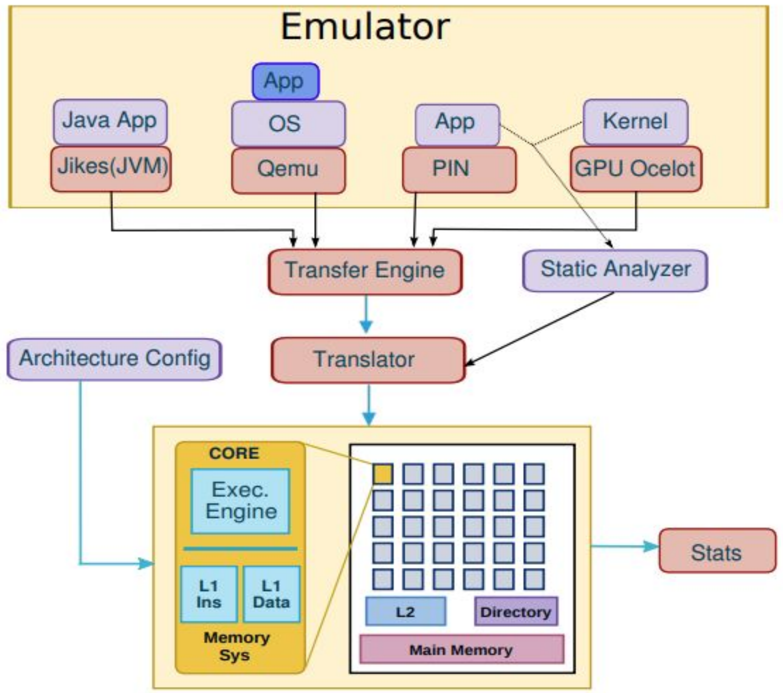
- It serves as a cost-effective alternative to physically designing new systems, allowing researchers to experiment and analyze various architectural ideas.
- This enables iterative refinement of architectural designs, hypothesis testing, and performance optimization.

About Tejas

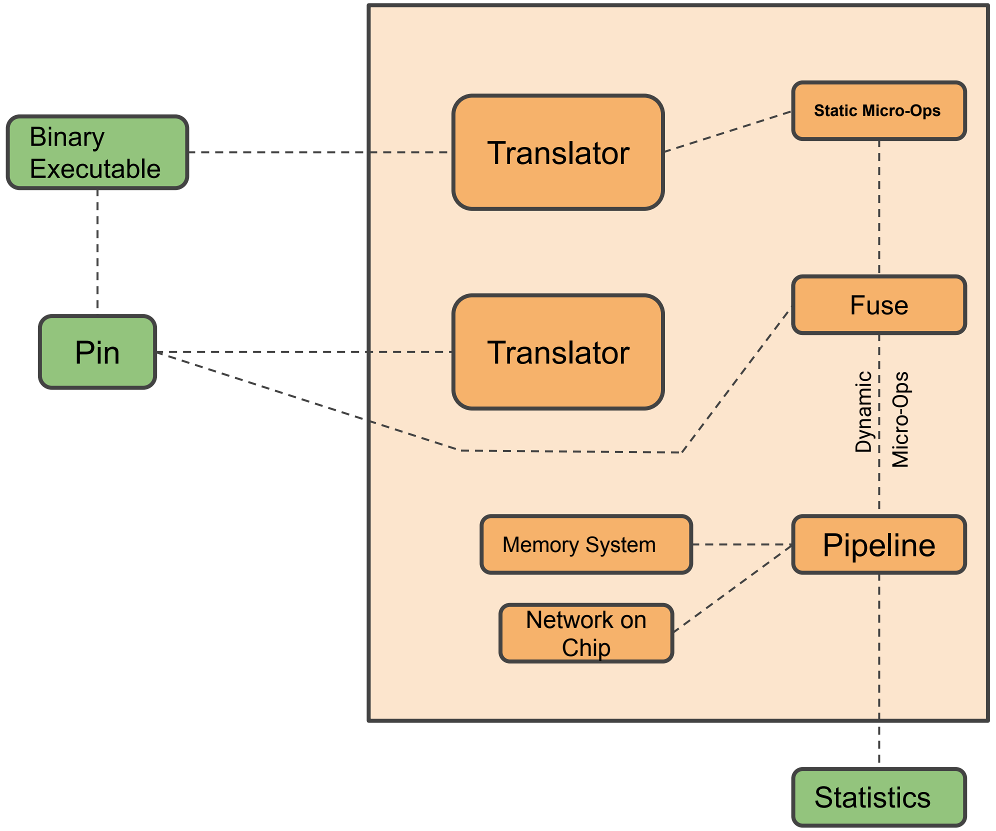
- Tejas Simulator® is an open-source, Java-based multicore architectural simulator built by the **Srishti research group, IIT Delhi**. It includes support for Java applications simulation, multicore simulation, cache coherence, NUCA protocols, and power simulation.
- It has an extremely modular component-oriented design, and it is easy to add new features.
- Tejas simulates the x86 ISA (32 and 64 bits). It is easy to add support for other ISAs as well. Simulation speeds achieved by Tejas are at par, if not better than most open source simulators.

What is an Emulator?

An emulator is a software or hardware tool that enables a computer system to imitate the behavior of another computer system or device, allowing software designed for one system to run on a different system by providing a virtual environment.



System Architecture Of Tejas



Communicating the Trace to the Simulator

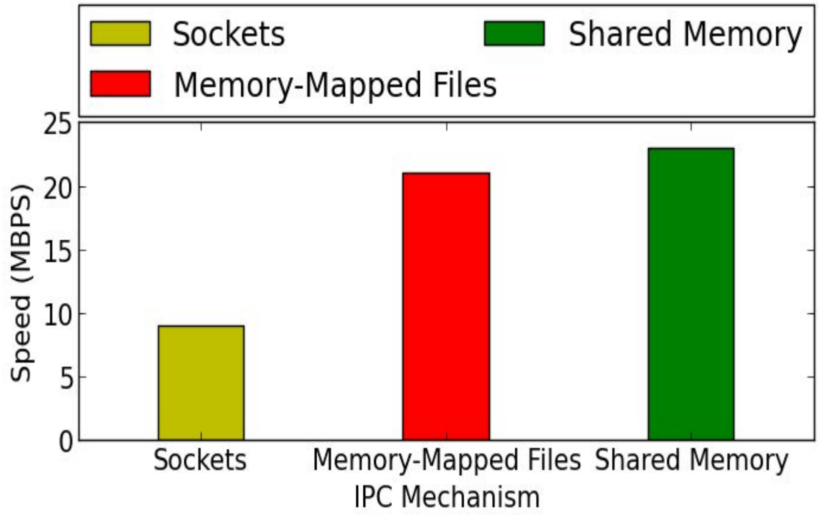
The application threads run in parallel, and potentially generate gigabytes of data per second.

It is necessary to send all of this data to Tejas using a high throughput channel. Options that were evaluated are

- ❖ Network sockets
- ❖ Memory mapped files
- ❖ Shared memory

Inter-process Communication Mechanisms

- ❖ Sockets are the slowest (10 MBps). This is because they make costly system calls to transfer data across the processes, and buffering is done by the Kernel.
- ❖ For high throughput, shared memory is the best option (24 MBps).
- ❖ Communication with memory mapped files is slower, because we need to synchronize data with the hard disk, or the disk cache in main memory.



VISA (Virtual Instruction Set Architecture)

- The VISA instruction set is fairly abstract, and it has sufficient information to perform a timing simulation.
- It is not concerned with different behavioral aspects of the instruction set.
- Almost all major architectural simulators break down emulated instructions into a simpler instruction set.

VISA (Virtual Instruction Set Architecture)

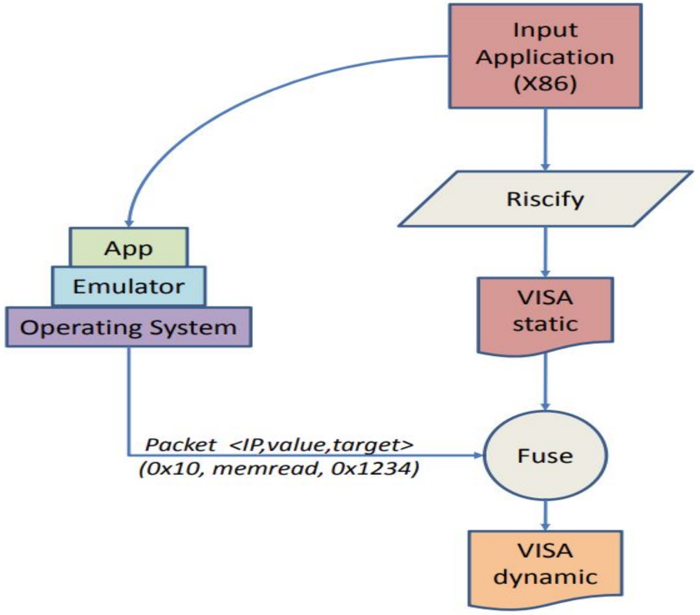
Types of operands supported

- ➢ integer register
- ➢ floating point register
- ➢ immediate value
- ➢ memory operand

VISA (Virtual Instruction Set Architecture)

Types of operations supported on integer and floating point operands

- ➢ ALU, Multiplication and Division.
- ➢ Load can fetch a value from memory into an integer or floating point register.
- ➢ Jump represents an unconditional transfer control to a different point in the program, whereas branch uses a conditional statement.



```
add %rax, (%rbx)
```

```
add  : load, int-alu, store
rax  : reg0
rbx  : reg1
```

```
load <(%reg1), ?>, %tmpreg1
int-alu %reg0, %tmpreg1, %tmpreg1
store %tmpreg1, <(%reg1), ?>
```

```
load <(%reg1), 0x1234>, %tmpreg1
int-alu %reg0, %tmpreg1, %tmpreg1
store %tmpreg1, <(%reg1), 0x1234>
```

Translator

- ❖ The translator module in the simulator is responsible for mapping different instruction sets to VISA.
- ❖ To optimize performance, the mapping focuses on commonly used instructions and skips rare instructions that are not architecturally significant.
- ❖ The design of the simulator prioritizes simplicity and scalability, but it ensures correctness by executing the program on a separate emulator or native machine.

Measure completeness of Translator

- Static Coverage
  - Static Coverage of the translator is the fraction of instructions in the object executable that could be translated into VISA.
- Dynamic Coverage
  - Dynamic Coverage of the simulator to the fraction of dynamically executed instructions that were translated to VISA.

Architecture Specification

Modular Structure

Tejas is designed as a modular system, allowing for easy addition of new features or modifications. This means that new types of pipelines, branch predictors, cache replacement policies, network-on-chip (NOC) topologies, and routing algorithms can be integrated into Tejas without significant challenges.

Architecture Specification

Scalability:

Tejas can simulate systems with varying numbers of cores. Extensive experiments have been conducted with 128-core systems without encountering significant performance issues. This scalability demonstrates Tejas' ability to handle large-scale simulations effectively.

Architecture Specification

Configurability:

Tejas offers high configurability for different components. Cores are implemented as pipelines with private cache memory, and various types of pipelines can be chosen based on the requirements. The memory system can be specified with great flexibility, allowing customization according to specific needs. The NOC implementation supports multiple topologies and routing algorithms, providing options for efficient interconnect design.

Pipeline

Tejas provides two pipeline types

- ❖ Multi-Issue In-Order
- ❖ Out-of-Order

Features Common to Both Pipelines

- A unified pipeline interface exposed to the rest of the system
  - A queue of instructions forms the input to the pipeline. The translator feeds instructions into this queue.
  - A function that describes the pipeline's one cycle operation. This function is called in the principal simulation loop, where the global time is advanced by one cycle in each iteration.
- Branch predictor
  - The different predictors provided are Always Taken, Always Not Taken, Bimodal, GAg, GAp, PAg, PAp, GShare, and tournament predictors.
  - New predictors can be easily added.

In-order Pipeline

❖ **Instruction Fetch**

- An instruction is fetched from the input queue (that is filled by the translator). A request to the i-cache is made, the address equal to the program counter of the instruction.
- Meanwhile, the instruction is placed in a fixed size buffer called the iCacheBuffer. The instruction resides here until the i-cache responds.
- Once it does, the entry from the iCacheBuffer is removed, and the instruction fetch is deemed complete.

In-order Pipeline

❖ **Instruction Decode**

- An instruction stays in the decode stage until both its source operands are ready, and the corresponding functional unit is available.
- Once available, depending on the type of operation, the destination register's time when ready is set. For a load instruction, whose latency cannot be determined at decode time, the time when ready is set to infinity.
- Branch prediction is performed in the decode stage -- a mis-prediction results in a penalty of further instruction fetches being stalled for a predetermined number of cycles.

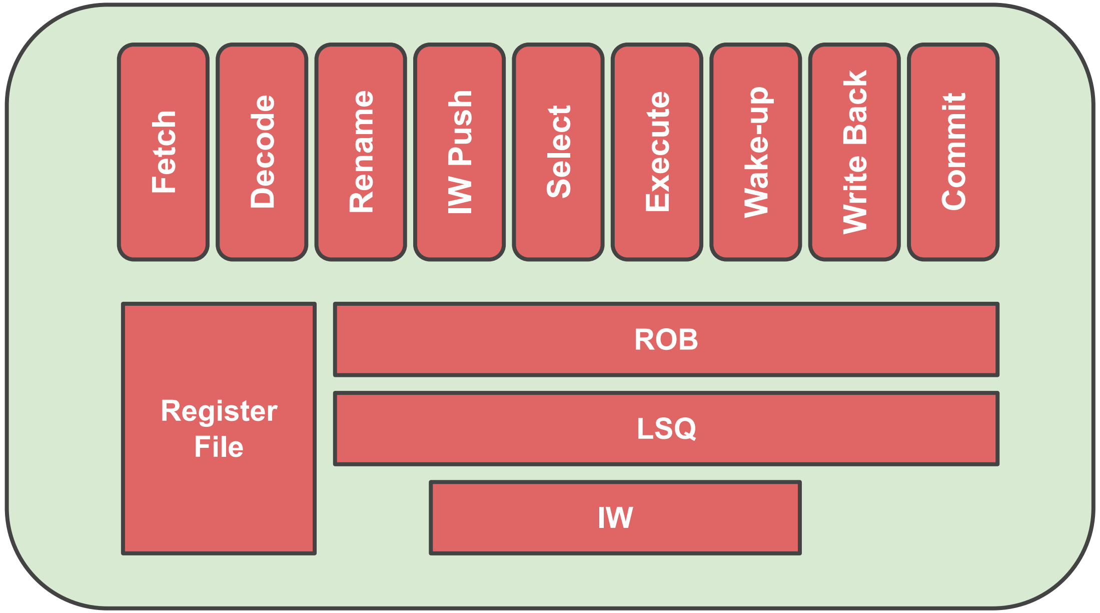
In-order Pipeline

- **Execute Stage** An instruction occupies the execute stage until the corresponding functional unit completes.
- **Memory Stage** If a load instruction, it occupies this stage until the memory system responds. The corresponding destination register's time when ready is set to the current time when the load completes.
- **Write-back Stage** Simulated as a 1 cycle operation.

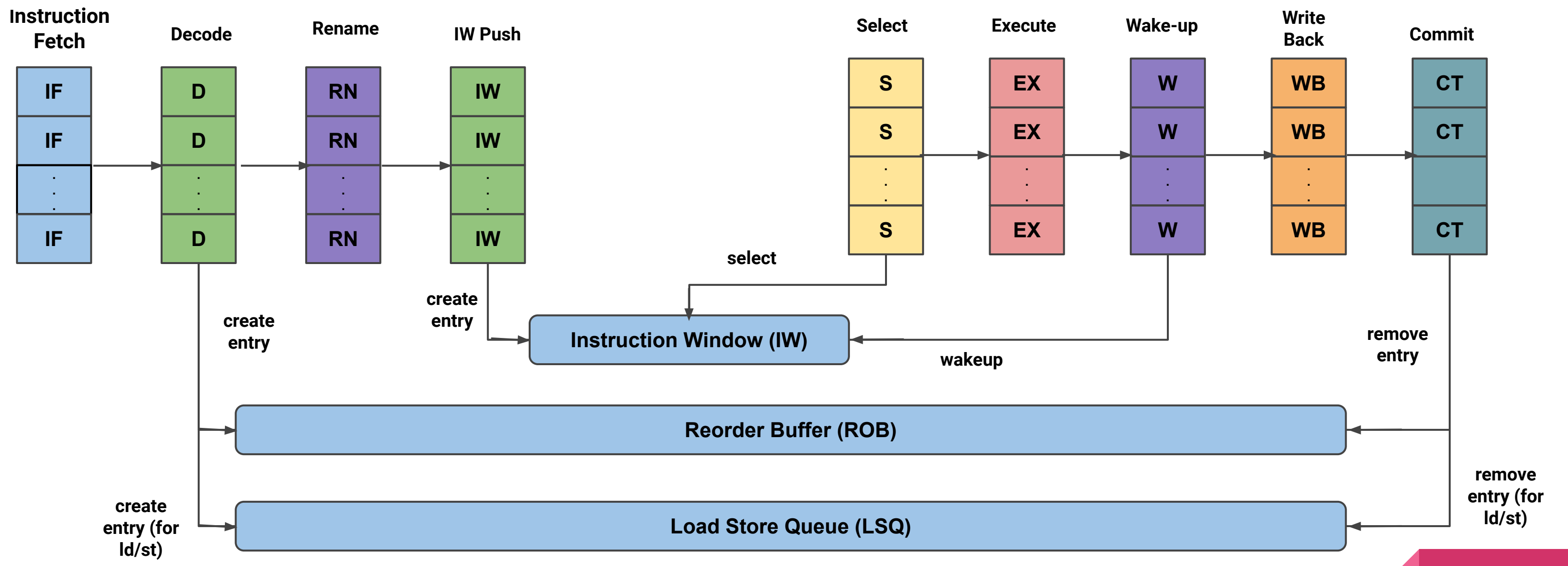
Multi-issue In-order Pipeline

- The width of the pipeline can be more than 1 (set <IssueWidth> tag in the configuration file). The architecture implemented is based on the Intel Pentium processor (Alpert et. al., 1993).
- Multiple instructions can be processed by a stage in 1 cycle. This processing happens in-order, brought about by modeling the latches between stages as circular queues of size equal to the pipeline width. All types of hazards are strictly handled.

Out-of-Order Pipeline



Out-of-Order Pipeline



Instruction Fetch

- An instruction is fetched from the input queue (that is filled by the translator).
- A request to the i-cache is made, the address equal to the program counter of the instruction.
- Meanwhile, the instruction is placed in a fixed size buffer called the iCacheBuffer.
- The instruction resides here until the i-cache responds. Once it does, the entry from the iCacheBuffer is removed, and the instruction fetch is deemed complete.

Instruction Decode

- Once fetched, “instruction decode” is simulated. Again, since all details are known, only timing is simulated by advancing the clock.
- A Reorder Buffer entry and a Load-Store Queue entry (if memory operation) are made at this point.
- Unavailability of free entries causes the pipeline to stall till entries are available.

Rename

- An available physical register is assigned to the destination operand. Unavailability causes the pipeline to stall.
- The physical registers corresponding to the source operands are determined, and their availability, that is, whether or not their values are available in the register file, is ascertained.
- The rename logic is made up of a register alias table (RAT) and a list of available registers.

Instruction Window Push

- The next stage involves the creation of an Instruction Window entry. Unavailability causes pipeline stalling.
- The size of the Instruction Window, Reorder Buffer and Load-Store Queue can be specified in the configuration file.

Instruction Select

- ❖ The select logic, in every cycle, processes the entries in the IW, looking for ready instructions. If the operands of an instruction in the Instruction Window are available, and a functional unit that it can execute on is available, the instruction is issued for execution.
- ❖ The issue width can be set in the configuration file. The IW entries are processed in-order -- so if more than issue width number of instructions are found ready, the ones that entered the window earlier are given preference.

Execute

- The instruction stays in the execute stage until the execution completes. Based on the type of operation, execution times vary -- upon issue, an event signalling completion of execution is scheduled for n cycles from the current time, where n is the latency of the corresponding functional unit.
- The latency of the functional units can be set in the configuration file. A load instruction can be serviced through load-store forwarding, if a store to the same address occupies an earlier position in the queue.

Wake-up

❖ Once execution completes, the instructions waiting for the result (dependent instructions) are woken up to begin execution in the very next cycle.

Write-back

❖ The register corresponding to the destination operand is marked ready.

Commit

- The last stage is the commit of the instruction. First, if the instruction was a branch, a prediction is performed. The prediction is compared with the actual outcome. If they differ, then the penalty for misprediction is simulated by stalling all stages of the pipeline for a pre-specified (in the configuration file) number of cycles.
- If the instruction being committed is a store, the Load-Store Queue is intimated that it may allow the value to be written to the memory hierarchy.

Wake-up Select Logic

- Wake-up Select logic allows an instruction j waiting for instruction i's result, to begin execution in the immediately next cycle after instruction i completes execution. When instruction i completes, it wakes up all dependent instructions. Its result is forwarded to the dependent instruction through the by-pass path, essentially storage associated with the functional units.

Wake-up Select Logic

- The wake-up signal is modelled as an event, “Broadcast Event”, as its time cannot be statically determined. In simulation, Select is performed before the events are processed. Therefore, the broadcast event is scheduled at the same time as the execution complete event. Doing this would set the availability of the operand in this cycle (the cycle when the producing instruction completed execution). This instruction then becomes a candidate for selection in the next cycle.

Two cases exist that require special handling

❖ Suppose the consuming instruction j was to be selected for execution in the same cycle (begins execution in the next cycle) as when instruction i completed execution. This isn't possible with the above described solution, as the wake-up is performed in the same cycle as i's completion. The selection can happen only in the next cycle, and the execution would thus begin in the cycle after that.

Two cases exist that require special handling

- ❖ Suppose the consuming instruction j is in the rename stage when the producing instruction i completes execution. j looks at the register file, and deems it's operand unavailable. The broadcast signal sent by i touches only the instruction window, and is thus not seen by j. Subsequently, j enters the instruction window, and i updates the register file. Effectively, j has missed the wake-up signal and thus, remains perpetually in the pipeline.

Simultaneous Multi-Threading (SMT)

- The out-of-order pipeline is capable of simultaneous multi-threading. The pipeline is provided with a decode width (configurable) number of input queues that the translator can populate -- each queue corresponds to one thread.
- The pipeline reads from these queues in round-robin fashion -- one queue each cycle. Each instruction has a thread id field to help distinguish it when data dependencies are handled. When one thread faces a branch misprediction, all threads are stalled for the predefined penalty period.

Reference :

https://www.cse.iitd.ac.in/tejas/overview.html

Thank You