CSL718 Architecture of High Performance Computers
Tejas : Software Architecture

1 High Level Architecture

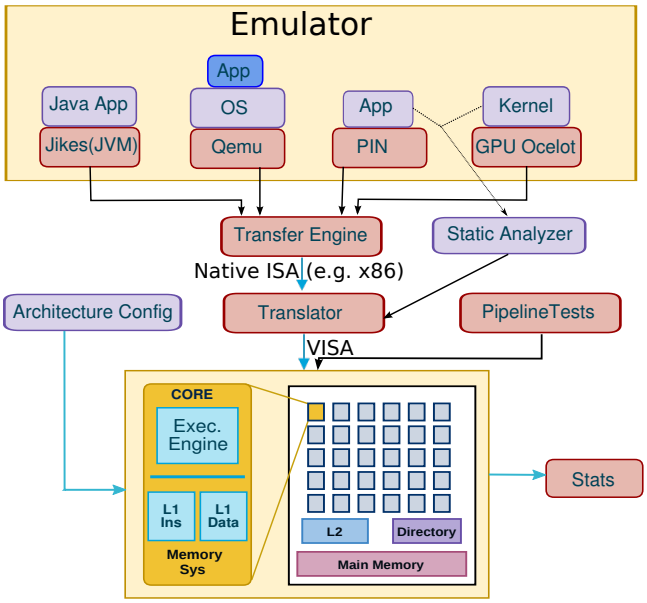

Figure 1: High Level Architecture

Figure 1 presents a high level view of Tejas. The given benchmark is executed by an emulator (like Intel Pin). The emulator collects runtime traces and provides these to a translator module. The translator converts the traces into VISA microps (see Section 2). These microps are then fed to the simulated pipelines.

An alternative way to exercise the pipelines is through the "PipelineTests" framework (see Section 8).

Its important to note that Tejas concerns itself with *timing*, and not *values*.

This is best explained with an example. Consider the instruction `add r1, r2, r3`. We do not concern ourselves with the values of `r1` and `r2`, nor do we perform the addition and store the result in `r3`. Instead, we are concerned with when `r1` and `r2` will be ready, how long will the addition take, and consequently when will `r3` be ready so that some dependant instruction can begin execution. Similarly, in the caches, we concern ourself with whether a given address is present or not. We are not concerned with the value present in a particular cache line.

2 Virtual Instruction Set Architecture (VISA)

The pipelines in Tejas simulate the execution of a custom instruction set known as VISA. Therefore, the instructions belonging to a native instruction set like x86 have to be translated to VISA microps. This is done by the translator module.

2.1 VISA Micro-Ops

VISA is a simple RISC instruction set.

```
public class Instruction {
       private OperationType type;  //Instruction type : see
           OperationType.java
       private Operand sourceOperand1;
       private Operand sourceOperand2;
       private Operand destinationOperand;
       private long ciscProgramCounter;//use this for instruction fetch
       private boolean branchTaken;
}
```

The instruction types are *integerALU, integerMul, integerDiv, floatALU, floatMul, floatDiv, load, store, jump, branch, mov, xchg, nop, sync*. See Table 1 for details regarding the different instructions.

2.2 VISA Operands

A VISA micro-op generally takes two source operands, and one destination operand.

```
public class Operand {
  private OperandType type;
  private long value;//if register type, value indicates which register
}
```

An operand can be one of four types as shown in Table 2.

Instruction Class	Instruction Type(s)	Description
Compute	integerALU integerMul integerDiv floatALU floatMul floatDiv	Can take two register operands OR two immediate operands OR one register and one immediate operand; Result is placed in the destination register
Memory	load store	Operand 1: address; destination operand: read value placed here Operand 1: address; operand 2: value to be written
Control Flow	jump branch	unconditional jump conditional jump
Data Movement	mov xchg	Move operand 1 to destination operand Exchange operand 1 and operand 2
	nop inValid	No operation To mark the end of the input stream. Indicator to stop simulation.

Table 1: VISA Instruction Set

Operand Type	Description
integerRegister	Integer Register
floatRegister	Floating Point Register
immediate	Immediate Value
memory	Used to specify the address in loads and stores. A memory operand is recursively defined in terms of two operands. See Section 2.4.

Table 2: Operand Types

2.3 Program Counter

The field `ciscProgramCounter` gives the program counter of an instruction. This value is to be used when issuing a request to the *iCache* during the instruction fetch stage.

In a CISC machine (say x86), the fetch stage fetches a CISC instruction. During the decode stage, the CISC instruction is broken down into multiple RISC instructions. In Tejas, the translation module translates a single CISC instruction into multiple VISA micro-ops. For example, a single x86 instruction performs load-add-store. This is broken down into 3 VISA micro-ops. Suppose the x86 instruction is at PC 0x1000. In the sequence of VISA instructions, the load gets `ciscProgramCounter` 0x1000. The add and the store are given a `ciscProgramCounter` of -1. The fetch stage in the pipeline issues an instruction fetch only for non-negative program counters. This ensures that like the actual machine, fetches are issued only once per CISC instruction.

2.4 Memory Instructions

Consider a load instruction *ldInst*: `mov 5(eax), ebx`. PIN tells us that the actual address that the load took place from was $0x1234$.

- `ldInst.getSourceOperand1MemValue()`: gives the address ($0x1234$)

- `ldInst.getDestinationOperand()`: gives the register where the load value will be placed (operand of type `integerRegister`, and value 2 (assuming *eax* has the mapping 1, *ebx* has the mapping 2 and so on))
- `ldInst.getSourceOperand1.getMemoryLocationFirstOperand()`: gives the operand used to generate the first part of the address (operand of type `immediate`, and value 5)
- `ldInst.getSourceOperand1.getMemoryLocationSecondOperand()`: gives the operand used to generate the second part of the address (operand of type `integerRegister`, and value 1)

2.5 Branch Instructions

`branchInst.isBranchTaken()` tells us if a branch is taken or not. To see how to perform branch prediction, see the file `DecodeUnitIn_MII.java` from lines 138 onwards. It must be noted that, the stream of instructions coming from PIN are always in the correct execution order. Therefore, there is no need for a pipeline flush on a branch misprediction. You can simply make the pipeline do nothing for a fixed number of cycles (mentioned in *config.xml*, tag `<BranchMispredPenalty>`).

2.6 `inputToPipeline`

The stream of instructions from PIN reaches the pipeline through a structure called *inputToPipeline*. Go through `FetchUnitIn_MII.java` to understand how to read from the input stream.

3 Semi-Event Driven Model

Tejas follows a hybrid model for simulation, which is a combination of iterative and event-driven models.

In a purely iterative model, the simulator basically runs a loop. In each iteration of the loop, (i) each structure in the simulated architecture is advanced through one cycle, that is, its FSM is exercised through one cycle, and (ii) the clock is incremented by one cycle. This loop runs until all instructions are simulated.

An alternative strategy is a discrete-event simulator. Essentially, an event queue is maintained. In each iteration of the loop, (i) all events scheduled for the current cycle is serviced, creating more events, and (ii) the clock is incremented by one cycle. Consider a load operation. An last-level cache miss would typically take around 200 cycles to complete. This amounts to 200 iterations in the iterative model. In the discrete-event simulator model however, we can directly skip to the 200th cycle.

The event queue is basically a priority queue. A priority queue is an expensive structure when the size of the queue is large. Thus, in Tejas, for activities

that happen every cycle like fetch and decode, we employ the iterative model. For activities that have long or unpredictable latencies, we employ the discrete-event simulator model.

4 Pipeline Interface

A pipeline must implement the pipeline interface (`src/simulator/pipeline/PipelineInterface.java`). For every cycle of the simulation, the function oneCycleOperation() in the interface is called. As an example, the interface to the in-order pipeline is as follows :

```
public void oneCycleOperation() {
        writeback();

        drainEventQueue();

        mem();
        exec();
        decode();
        fetch();
}
```

You may notice that the various stages of the pipeline are invoked in reverse order : writeback first and fetch last. This is because we are sequentially simulating a parallel operation : the input to the $(k+1)^{th}$ stage at time t, is the output of the k^{th} stage generated in time $t-1$. If we invoke the stages in the forward order, the input to the $(k+1)^{th}$ stage at time t will be the output of the k^{th} stage generated in time t, instead of $t-1$.

`drainEventQueue()` refers to processing the vents scheduled for that particular cycle. All interactions with the memory system are through events. You may choose to handle the completion of execution of the functional units also through events.

Finishing Simulation The `inValid` instruction is used to indicate the end of the stream. When this instruction is encountered, a flag needs to be set to true to indicate to the driving loop of the simulator to break out. This can be done from the pipeline by executing `containingExecutionEngine.setExecutionComplete(true)`. See line 67 in the file `WriteBackUnitIn_MII` for usage details.

5 Memory system interface

As mentioned earlier, all memory operations are through events. The memory system interface (src/simulator/memorysystem/CoreMemorySystem.java) allows you to place load or store requests for either data or instructions, and receive responses for load requests. Every pipeline must implement this interface.

Minor modifications to
`src/simulator/pipeline/multi_issue_inorder/InorderCoreMemorySystem_MII.java`,
as to how to process responses for load requests, should prove sufficient.

6 Pooling

To optimize Tejas' heap behavior, we try and avoid all dynamic allocation of memory. Therefore, we pre-allocate a "pool" of *Instructions* and *Operands*, and use these throughout the course of the simulation. Upon finishing the simulation of an instruction, the pipeline must return the instruction back to the pool. See line 121 of `WriteBackUnitIn_MII.java` to see how this is done.

7 Statistics

The contents of the result file are the counters defined in `Statistics.java`. See lines $70 - 71$ in `WriteBackUnitIn_MII.java` to see how to set these counters.

8 PipelineTests

The framework provided in `src/simulator/pipeline/PipelineTests.java` can be used to generate sequences of micro-ops and feed these to the pipelines. This can help creating targeted test cases, to test different aspects of the pipeline/memory-system. Some example test cases are already provided in this file. They attempt to test the renaming logic and the data hazard and control hazard enforcement logic.

```
public static void renameTest() {
  //generate instruction sequence
  Instruction newInst;
  for(int i = 0; i < 100; i++) {
        newInst = Instruction.getFloatingPointDivision(
            Operand.getFloatRegister(0),
               Operand.getFloatRegister(0),
               Operand.getFloatRegister(0));

        inputToPipeline.enqueue(newInst);
  }
  inputToPipeline.enqueue(Instruction.getInvalidInstruction());//end-of-stream

  //simulate pipeline
  while(ArchitecturalComponent.getCores()[0].getPipelineInterface().isExecutionComplete()
      == false) {
        ArchitecturalComponent.getCores()[0].getPipelineInterface().oneCycleOperation();
        GlobalClock.incrementClock();
  }
}
```

To run PipelineTests, use the `main()` function in this file instead of `src/simulator/main/Main.java`. It takes a single command line argument – the configuration file (*config.xml*).