

Apex Instruction Set Architecture Simulator

(`apex-sim`)

Phase 2 Documentation

Matthew Cole
mcole8@binghamton.edu

Brian Gracin
bgracin1@binghamton.edu

19 November 2016

Contents

1	Design	1
1.1	Driver Program	1
1.2	Classes	3
1.2.1	Issue Queue	3
1.2.2	Reorder Buffer	4
1.2.3	Registers	4
1.2.4	CPU and Stages	4
2	Implementation	4
2.1	Reverse-Ordered Execution	5
2.1.1	Committing	5
2.1.2	Advancing	5
2.1.3	Working	5
2.1.4	Forwarding	5
2.2	Register Renaming and Allocation	5
2.2.1	Allocation	5
2.2.2	Renaming	5
2.3	Dispatch	5
2.3.1	Stalling for Multiple Control Flow Instructions	5
2.4	Issue	5
2.5	Commit	5
2.6	Statistics	5
3	Work Log	5
A	Appendix: Screen Captures	9

List of Figures

1	APEX pipeline and class data flows.	2
---	---	---

List of Tables

1	APEX Forwarding Scenarios	6
2	APEX Instruction Source and Destination Sets	7
3	Chronological Work Log	8

1 Design

`apex-sim` is a simulator for the *Architecture Pipeline EXample* (APEX) Instruction Set Architecture (ISA). `apex-sim` consists of the following components:

- `main.cpp` contains the driver program. The driver program provides file input for instructions, user interface operations, maintaining persistent simulator state and statistics monitoring. This component is discussed in section 1.1.
- `apex.cpp` contains helper functions for `main.cpp`. These include wrapper functions that delegate interface actions down to individual classes.
- Several source files provide the objects modeling components of the pipeline. These components are discussed in section 1.2. Briefly, they are
 - `code.cpp` models the simulator’s read-only instructions file.
 - `cpu.cpp` (plus its associated helper functions in `simulate.cpp`) models the stages in the pipeline and interact with the Instruction Queue (IQ) and Reorder Buffer (ROB). It is responsible for overall execution of a single cycle through its helper function `simulate`.
 - `data.cpp` models the simulator’s read-write main memory.
 - `iq.cpp` models the simulator’s IQ.
 - `registers.cpp` models the simulator’s unified register file.
 - `stage.cpp` models a single stage in the pipeline. It also doubles as an inflight instruction or entry in the IQ or ROB. This allows advancement of an inflight instruction to be greatly simplified.
- `simulate.cpp` provides the functions that allow the CPU to simulate working on each of its stages, inter-stage communication through advancement, stalls for basic inter-stage interlocks, out-of-order execution and reordering, and forwarding. These implementation details are described in section 2.

Figure 1 shows class interactions and data flow between each of the stages and support classes. Finally, we discuss our team’s work log in section 3.

1.1 Driver Program

The `apex-sim` entry point file is `main.cpp`. Besides maintaining simulator state variables for the current cycle, program counter and instructions filename, this program shepherds execution through the lifecycle of the program and provides a user interface for interacting with the simulator. The functionality of the driver program is as follows:

1. Verify sanity of command line inputs (lines 23-32).
2. Instantiate class instances for the simulator (lines 34-40).
3. Perform the initialization of each pipeline stage (line 43).

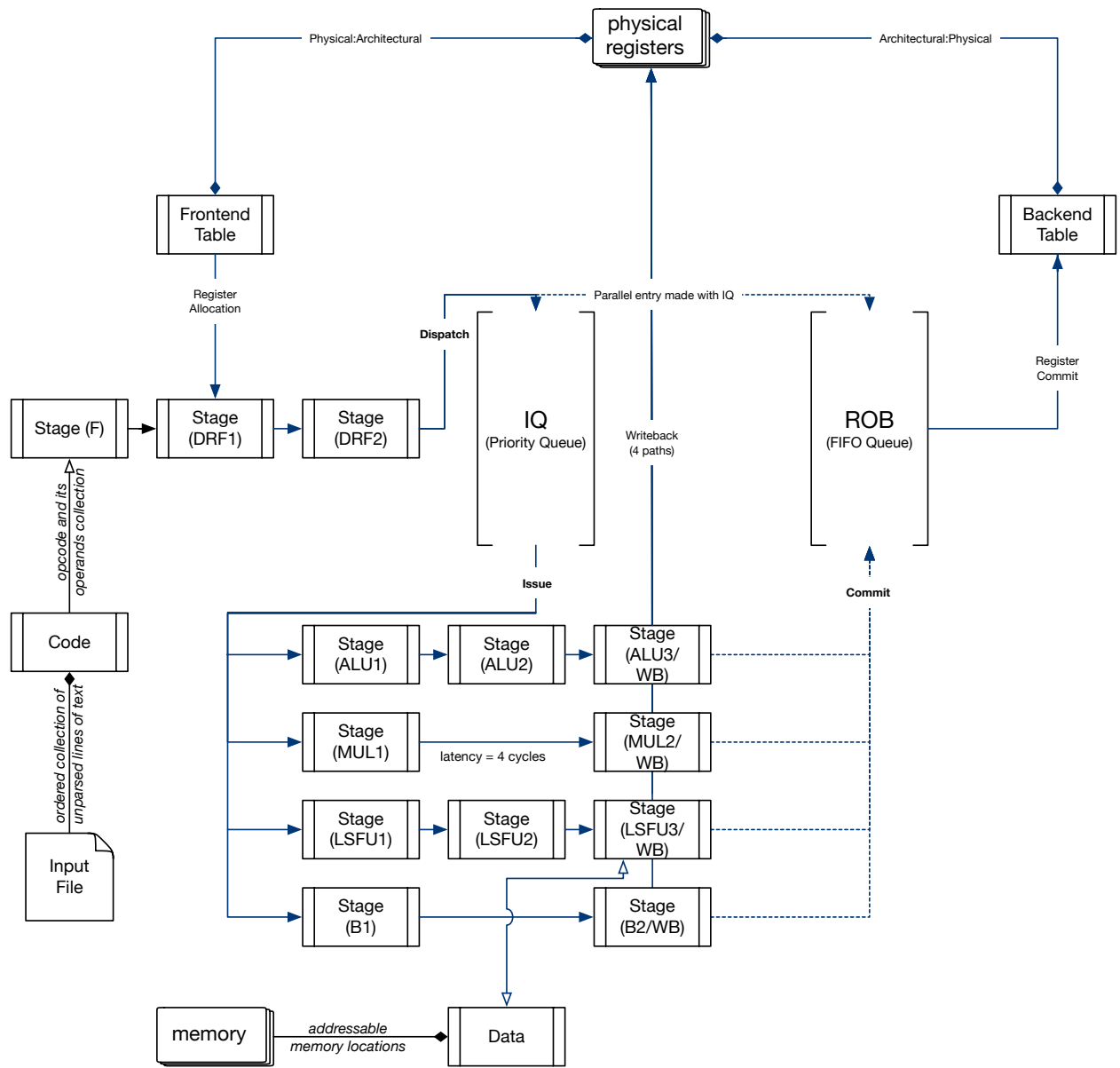


Figure 1: APEX pipeline and class data flows.

4. Prepare and begin the simulator user interface's operations (lines 48-54).
5. Parse user interface inputs and delegate actions to interface helper functions (lines 57-127).

`main.cpp` also provides helper functions which delegate work down to class instances in `apex.cpp`. These functions are:

- `help()` displays the user interface keyboard shortcuts. It's invoked at startup, on request from the user, and whenever the user provides an input which is not recognized.
- `initialize()` resets the simulator state, and invokes the class instances' own `classname.initialize` functions which reset the instances' internal state.
- `display()` displays the simulator internal state variables as well as delegated calls to each class' `classname.display()` function.
- `stats()` displays simulator execution statistics.
- `simulate()` is the most important of the helper functions. It is responsible for controlling simulation of the `CPU.simulate()` function for a given number of cycles, and allowing the CPU class to communicate that it has encountered an error, reached EOF of a code file without a HALT instruction, or has processed a HALT instruction through the pipeline.
- `quit()` gracefully halts the simulator and triggers a final call to `display()`.

1.2 Classes

`apex-sim` models each major component of the APEX system as a standalone class. Unless mentioned below, these classes did not change appreciably from release v1.0 and are not discussed further in this report. Please see the release v1.0 documentation for discussion of these unchanged classes.

1.2.1 Issue Queue

`apex-sim`'s issue queue (IQ) uses C++11's Standard Template Library (STL) deque container to model a priority queue. Each entry in the deque is a Stage class instance. This allows inflight instructions to seamlessly dispatch from the DRF2 stage into the IQ, and issue out into one of the function units. Entries are inherently sorted by timestamp of fetching, ensuring that removing entries with earlier program-ordered are preferentially removed before later program-ordered instructions if both are ready. Additionally, this mechanism allows us to determine if an later LOAD instruction is attempting to bypass an earlier STORE instruction and prevent its issue. The use of the deque container was precipitated by the queue container not having a standard iterator.

1.2.2 Reorder Buffer

The reorder buffer (ROB) likewise uses a deque container for the same design reasons as the IQ, however it is a strict FIFO queue. This allows instructions which may have been issued out-of-program order to be committed in-program order. Whenever an entry is made in the IQ, a parallel entry is made and queued in the ROB. This ensures that the head of the ROB always points to the earliest dispatched instruction. Each of the function units' (FU) writeback (WB) stage contents are compared by opcode and timestamp to the ROB head once they become ready. When a match occurs, this allows that FU to commit its contents to the back-end register file and de-queue the ROB entry. In the case of the LSFU, it also allows memory access to occur. This enforces memory in-order execution.

1.2.3 Registers

The Registers class retains its basic structure as a register file, using an `STL::map` whose key is a register "tag" (a string), and whose value is a two-tuple of value (an integer) and validity (a bool). However, in phase two, we added three components. First, a front-end rename table mapping physical register tag to architectural register tag (`STL::map`). Second, a back-end rename table mapping architectural register tag to physical register tag (`STL::map`). Third, a free list (`STL::vector`), listing free physical registers, sorted by their tag's integral value. This assures that the lowest numbered physical register is preferentially allocated.

1.2.4 CPU and Stages

The functionality and design of the Stage class is largely unchanged from phase 1. However, the CPU's structure changed dramatically from phase 1.

- DRF stage was split into two stages.
- MUL FU was split from the larger ALU FU.
- MEM stage was replaced by a LSFU.
- WB stage was replaced by a specialized WB stage for each FU

Figure 1 shows the interactions between these new classes.

2 Implementation

In this section, we will discuss key aspects of our simulator's execution: the stage-wise reverse-ordered execution, register renaming, dispatch, issue, commit, register value forwarding.

2.1 Reverse-Ordered Execution

2.1.1 Committing

2.1.2 Advancing

2.1.3 Working

2.1.4 Forwarding

Table 1 describes scenarios where `apex-sim` performs forwarding from later stages to earlier stages in order to reduce bubbles caused by waiting for flow dependencies to resolve. In `apex-sim` this is accomplished in `simulate.cpp` during the forwarding phase (lines). The instructions' source sets and destination sets are enumerated in table 2.

2.2 Register Renaming and Allocation

2.2.1 Allocation

2.2.2 Renaming

2.3 Dispatch

2.3.1 Stalling for Multiple Control Flow Instructions

2.4 Issue

2.5 Commit

2.6 Statistics

3 Work Log

We open-sourced `apex-sim` under the MIT license, and developed it using a GitHub repository.¹ This repository contains this documentation, all source code, reference materials on the APEX ISA semantics, and other related materials. Additionally, it contains an in-depth look at our work progress over the course of this project at a much finer grain than this report contains. As of writing this report, commits were made with a total of lines of code. Naturally, such a volume of code and the required levels of collaboration would have

¹The repository is available at <https://github.com/colehatt/apex-sim>

Add forwarding code's line numbers

Add discussion of allocating in ascending order of tag. See para 5 in spec.

Add discussion of issuing LSFU in program order. See para 4 in spec.

Add discussion of how

Table 1: APEX Forwarding Scenarios

From	To	Description
B2	B1	Occurs when B2 has a BAL instruction, and B1 has the X register in its source set.
B2	IQ	Occurs when B2 has a BAL instruction, and one or more IQ entries is a BAL or JUMP with the X register in its source set.
ALU3	IQ	Occurs when ALU3's destination set has a union with an IQ entry's source set.
ALU3	ALU2	Occurs when ALU3's destination set has a union with ALU2's source set.
ALU3	MUL1	Occurs when ALU3's destination set has a union with MUL1's source set.
ALU2	ALU1	Occurs when ALU2's destination set has a union with ALU1's source set.
ALU2	MUL1	Occurs when ALU2's destination set has a union with MUL1's source set.
ALU2	B1	B1's opcode is BAL or JUMP , ALU2's destination set has a union with B1's source set.
ALU2	LSFU2	LSFU2's opcode is LOAD or STORE , ALU2's destination set has a union with LSFU2's source set.
ALU2	LSFU1	LSFU2's opcode is LOAD or STORE , ALU2's destination set has a union with LSFU1's source set.
ALU2	IQ	ALU2's destination set has a union with an IQ entry's source set.
MUL2	IQ	MUL2's destination set has a union with an IQ entry's source set.
MUL1	ALU1	MUL1's destination set has a union with ALU1's source set.
MUL1	B1	B1's opcode is BAL or JUMP and MUL1's destination set has a union with B1's source set.
MUL1	LSFU2	MUL1's destination set has a union with LSFU2's source set.
MUL1	LSFU1	MUL1's destination set has a union with LSFU1's source set.
MUL1	IQ	MUL1's destination set has a union with an IQ entry's source set.
LSFU3	LSFU2	LSFU3's destination set has a union with LSFU2's source set.
LSFU3	LSFU1	LSFU3's destination set has a union with LSFU1's source set.
LSFU3	ALU1	LSFU3's destination set has a union with ALU1's source set.
LSFU3	MUL1	LSFU3's destination set has a union with MUL1's source set.
LSFU3	LIQ	LSFU3's destination set has a union with an entry in the IQ's source set.

Table 2: APEX Instruction Source and Destination Sets		
Instruction	Destination Set Operand Indices	Source Set Operand Indices
Arithmetic	0	1,2
MOVC	0	-
LOAD	0	1
STORE	1	0
BAL, JUMP	-	0
BZ, BNZ	-	-
HALT, NOP	-	-

been nearly impossible without the use of some sort of repository. We encourage the curious reader to see these statistics in depth using the **Pulse** and **Graphs** tabs available on the repository.

Table 3 is a broad, chronological overview of work performed by each member.

Table 3: Chronological Work Log

Date	Matthew's Task	Brian's Task
Dec 5, 2016	Moved source files into their own directory, updated Makefile.	Began modifying Registers class to support URF operations. Created Front-end, Back-end table, Free-list. Prototyped API functions to class.
Dec 6, 2016	Updated UI for new commands specified in Phase 2. Added stats mechanisms. Updated display methods.	Further work on Registers class, updated function to create new instances of registers. Began work on ROB class.
Dec 7, 2016	Allowed templating Stages to IQ and ROB dequeues. Continued work on final report.	Completed ROB and IQ class architectures. Added commit and head-compare utility functions to ROB and IQ for programmer convenience.
Dec 8, 2016	Visibility and inter-class communications pathways. Added halting logic.	Initial compilation and end-to-end testing. Added issue utility function to IQ for programmer convenience.
Dec 9, 2016	Continued work on documentation.	Continued work on documentation.
Dec 10, 2016	Continued work on documentation. Fixed blocking for advance functions. Began work phase code for all FUs.	Completed issue, committing and writeback design. Extensive end-to-end troubleshooting.
Dec 11, 2016	Finalized Work phase, Advancing phase and Forwarding phase in <code>simulate()</code> function. Squashed 3 issues.	Continued troubleshooting on all classes.
Dec 12, 2016	Completed final report, troubleshooting, screen captures and submission package.	Substantial troubleshooting and Checkpointed Release v.2.0!

A Appendix: Screen Captures

Add
screen
cap-
tures.