Apex Instruction Set Architecture Simulator (`apex-sim`) Phase 2 Documentation

Matthew Cole
`mcole8@binghamton.edu`

Brian Gracin
`bgracin1@binghamton.edu`

12 December 2016

Contents

List of Figures

List of Tables

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).

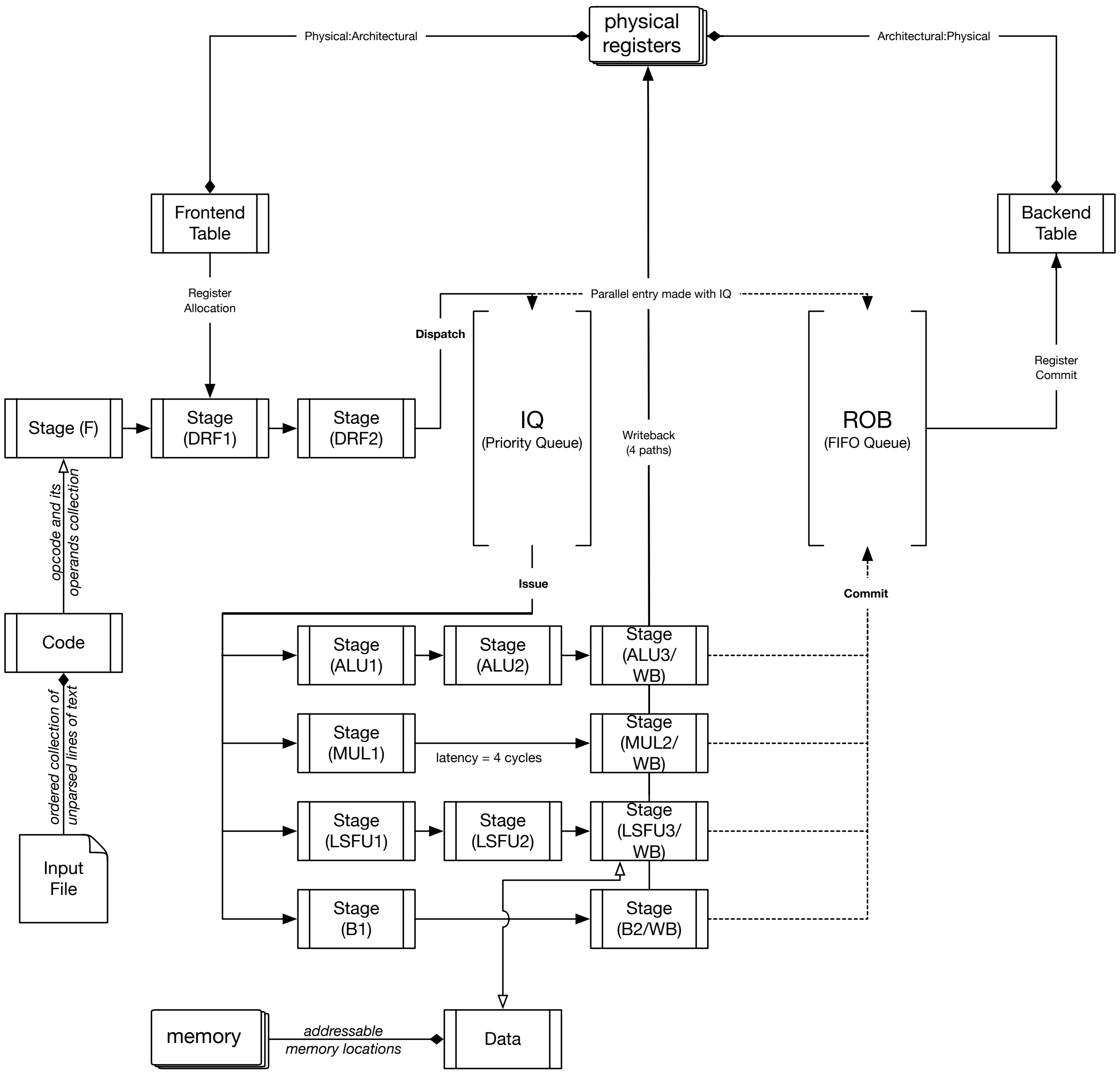

Figure 1: APEX pipeline and class data flows.

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::deque), listing free physical registers, sorted by their tag's integral value. This assures that the lowest numbered physical register is preferentially allocated. Section 2.2 discusses this further.

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

When we say *reverse-ordered execution*, what we mean is that the `simulate()` function generally proceeds in what appears to be the opposite order of the pipeline flow. That is, later stages in the pipeline perform their tasks in any given phase before earlier stages, and the Branch FU performs its work before the other function units. This scheme prevents runaway by an inflight instruction or crossing up values between stages. This function is broken up into four phases, each providing one general task of the overall cycle's worth of simulation.

2.1.1 Forwarding

Stages attempt to forward. 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.

Add forwarding code's line numbers

2.1.2 Committing

In this phase, the stages' contents of each of the four WB stages are compared to the ROB's head. If the stage is ready, not empty, and the opcode and timestamp match between stage and ROB head, the instruction is committed. Committing allows register values to writeback, allows memory access for the LSFU, deallocates registers back into the free list, pops the ROB head, and empties the WB stage. This phase occurs in lines , and takes advantage of the ROB class' `commit()` function to perform each of the tasks listed above.

add committing source code line numbers

2.1.3 Advancing

Next, stages other than those which commit are allowed to advance. Advancing occurs only if the following conditions are all met:

- The sending stage is not empty.
- The sending stage is ready (i.e. it has completed its work in a previous cycle's working phase).
- The receiving stage is empty (i.e. it advanced its contents in this cycle or a previous cycle).

Advancing takes advantage of the Stage class' `advance()` function to perform this logic and to copy contents. This phase occurs in lines .

add advancing phase source code lines

2.1.4 Working

Then, stages perform their work as one might expect. For example, the ALU performs either arithmetic, logical operations or a constant move into a register. Other stages have their own working phase tasks, and they can be seen in lines .

add working

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	-	-

2.2 Register Renaming and Allocation

We use the free list to determine if a physical register can be allocated as a stand-in for an architectural register. If it can, the register is removed from the free list, and mappings are updated in the front-end and back-end rename tables. After the instruction using that physical register as a source reaches the commit step, the physical register is deallocated by making sure that the mappings are updated, and the register is added to the free list. When the register is added to the free-list, the list is sorted to ensure that lower-numbered physical registers are preferentially used. While this is not required to strictly support register renaming, we have done so to support deterministic evaluation. These operations are enabled by use of the Registers class' `getRenamed()`, `commit()` and `deallocate()` functions. Furthermore, both physical and architectural registers have respective write, read and validity check accessors and mutators for reading and writing to the register file.

2.3 Dispatch

Instructions are dispatched from the DRF2 stage to the IQ using the IQ class' `dispatchInst(Stage&)` function when DRF2 is ready and not empty. At the same time, an identical entry is added to the ROB using the ROB class' `addStage(Stage&)` function. There are three exceptions to this flow. First, if DRF2 contains a `HALT` instruction, it is not dispatched because keeping the `HALT` in DRF2 prevents further instructions from entering the IQ (unless a branch instruction flushes DRF2), and because it is part of the halting detection logic. Second, if DRF2 contains a `NOP` instruction, it is not dispatched because the instruction is idempotent and belongs to no FU. Third, if there is already a control flow instruction in the IQ, design limitations given by the specification prevent a second control flow instruction from entering the IQ. This is checked using the IQ's `hasEntryWithOpcode(String opcode)` function with each of the control flow instructions. Once dispatch occurs, or it is determined that no dispatch can occur, the relevant statistics are updated within the CPU simulation function.

2.4 Issue

As noted in section 1.2.1, the IQ is a priority queue. This means that we issue from the IQ to a waiting function unit using the IQ's `issue(...)` function. This function uses the following logic to select the highest priority entry for issue:

- Only ready entries can be issued. This means all operands needed in the first stage of the applicable FU are ready.
- The ready entry with the lowest time stamp should be issued first. This is accomplished by selecting the entry closest to the IQ head with the assumption that since entries were added in program order, they remain stored within the container in program order no matter how many dispatch and issue actions occur.

Once it is determined that issue occurs or no issue can occur, the relevant statistics are updated within the CPU simulation function.

2.5 Commit

Commits occur when inflight instructions reach one of the four WB stages. Their opcode and timestamp are compared against the ROB's head to determine which (if any) should commit. As noted above, the ROB is a true FIFO queue, even though it is implemented as a dual-ended deque container. This decision was made in order to permit the use of an iterator to traverse the ROB for flushing.

2.6 Flushing

Whenever a branch is taken, entries in the IQ, ROB and any stage must be flushed if its timestamp is *later* than the timestamp of the instruction causing the branch to be taken. This is achieved using `flush()` functions. These functions receive an integral value for the timestamp from the instruction generating the flush, and - in the case of the IQ and Stage functions - a handle to the registers so that they can cause the allocated registers they hold to be deallocated. For the IQ and ROB's version of the `flush()` function, an iterator across the collection simplifies the process.

2.7 Statistics

As noted above, statistics counters are updated throughout the execution span of the simulator. They are calculated using the `stats()` function in `apex.cpp`. This will also display the stats to the console either when the `stats()` or `display()` functions are called.

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 been nearly

Add number of commits

Add number of

[1] The repository is available at `https://github.com/colematt/apex-sim`

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 deques. 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 `simulate()` 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!**

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. Brian Gracin was the team lead for this release of the project. He is behind the overall design, register renaming and the overwhelming majority of troubleshooting. Matthew Cole rebuilt the simulation functions and user interfaces from release version 1.0, and maintained the repository and documentation.

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

A Appendix: Screen Captures

Add screen cap-tures.