CSEE 3827: Fundamentals of Computer Systems

Project #5

Pipelined KeyValue Store

Due 12/5/17 at 11:59pm

1 Key-Value Store

Your objective in this assignment is to *pipeline a key-value store*. The scaffolding provides a functional but slow baseline which you should aim to improve via pipelining.

Interface Our Key-Value store uses 8-bit keys and 8-bit values. It supports two instructions: read and write.

They are formatted as 17-bit commands.

- Read: [0][Key (8bits)][X (8bits)][1]
- Write: [1][Key (8bits)][Value (8bits)]

Each command produces a result that is formatted identically to the command, but with the read results carrying the read value.

- Read: [0][Key (8bits)][Value (8bits)]
- Write: [1][Key (8bits)][Value (8bits)]

Behavior Initially all entries in the store are zero, meaning any unitialized reads return zero values. Reads of initialized values should reflect the most recently written value. The store should produce results in the same order the commands arrive. Remember that the reset operation should work asynchronously, including when asserted for multiple clock cycles.

SLOW_KEYVAL In the scaffolding you will find the slow baseline. It uses a component called SLOW_KEYVAL which takes commands and produces results, using our usual latency-insensitive handshake. This slow module accepts a command, is busy for 10 cycles, issues the result and then becomes ready again to accept the next command. It is completely serial, processing one command at a time and using 10 cycles per command.

The baseline design simply routes commands to this slow unit, thereby taking $10 \times n$ cycles to complete n commands.

Implementing PIPE_KEYVAL Your task is to design and implement a pipeline for read and write instructions. In other words, find a way to overlap the execution of multiple instructions thereby improving the throughput of the system beyond the baseline. As with any pipeline, the objective is to minimize cycles per instruction or CPI.

2 Test Harness

Because this assignment centers on measuring performance, the test harness has some new features.

[1] The Value field of incoming read commands does not matter.

Cycle Count Output We have added a new output, CYCLES, which simply counts the number of clock cycles from reset until halt.

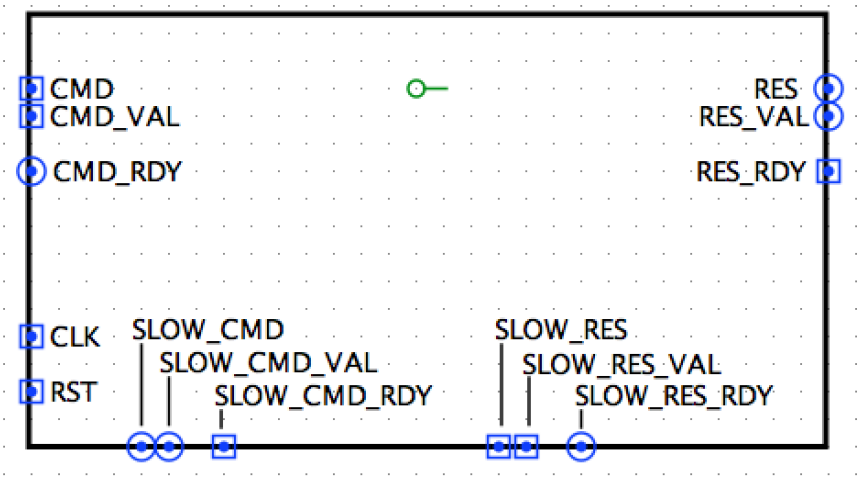
Measurement Modes There are now four measurement modes, set via the following two configuration bits:

perf_mode	1	harness always proffering/accepting data
	0	harness randomly proffers/accepts data, as in prior assignments
latency_mode	1	harness will wait until one command completes before proffering the next
	0	harness will send commands to PIPE_KEYVAL anytime it indicates it is ready

Note that in the initial scaffolding is completely serial, so you will see no difference in runtime between latency_mode settings.

Workloads The scaffolding includes a sequence of 100 instructions, which is 75% reads and 25% writes. The keys follow a Gaussian distribution with a mean of 127 and standard deviation of 1. The values are drawn from a uniform random distribution. For your convenience, we have provided the commands and results, in plaintext, in Canvas.

PIPE_KEYVAL Port Locations Note the port locations shown below. Your submission must match this interface.



3 Scoring Process and Rubric

We will test your submission on sequences of commands of various lengths. Each [Update 11/26/17: performance measurement] sequence on will have the statistical properties described above. [Functional check sequences will have arbitrary key distributions.]

To be graded, your submission must:

- not use any RAM or ROM in PIPE_KEYVAL,
- use only builtin Memory compoents to hold state (no buiding your own flip-flops),
- not exceed 256 bits of total state in PIPE_KEYVAL,
- [Update 11/17/17: upon halt, have the state of SLOW_KEYVAL reflect the commands supplied by the harness,]
- not change the name or appearance of the PIPE_KEYVAL module,

- halt without error (e.g., oscillations or undefined wire values), and
- be uploaded via courseworks.

We will first perform some basic checks on each submission. The first checks functionality, running a small sequence of two commands, and confirming that

- $NUM\_PASS_{\overline{perf\_mode} \cdot latency\_mode} == 10$
- $NUM\_PASS_{\overline{perf\_mode} \cdot \overline{latency\_mode}} == 10$

We will also check to see that your design is actually pipelined. The essence of a pipeline is that it improves throughput *without reducing the latency of individual operations*. We will thus check, again using ten commands, that individual operation latency has not been reduced:

- $CYCLES_{perf\_mode \cdot latency\_mode} >= 100$

If your submission fails to pass any of these three basic checks, we will not proceed further in scoring it and it will receive a zero.

Submissions that pass all three checks will be assessed for function and performance on sequences of 1000 instructions. We use a modified CPI formula, computed with $perf\_mode \cdot \overline{latency\_mode}$:

$$CPI = \frac{CYCLES + 20 \times (1000 - NUM\_PASS)}{1000}$$

If the simulation times out we will set $CYCLES$ to the runtime of the baseline, or 10000. Note that this formula admits some failed corner cases, but with a time penalty of 20 cycles per incorrect result. We will compute the average CPI across five random sequences of 1000 instructions.

We will assess the *resource* usage of your design by counting $StateBits$, the total bits of state contained in PIPE_KEYVAL.

	Total	Formula
Perf	80pts	$\lceil \frac{80}{9} \times (10 - CPI_{avg}) \rceil$ Points in proportion to reduction in CPI. Will not be negative or exceed 80.
Res	20pts	~~[20]~~ $\lceil \frac{20}{256} \times (256 - StateBits) \rceil$ Points in proportion to reduction in state bits.
Style	10 pts	If $Perf + Res > 50$, the usual extra credit for most beautiful schematics.

4 Hints

- I strongly encourage you to develop and test incrementally. Implement basic behaviors first and test to ensure the progressively more optimized versions work under all four scaffolding modes.
- Remember that when you are simulating you can drill down into the state of component blocks (and continue the simulation there) by clicking on the magnifying glass icon that appears when you hover over a module during simulation.
- Logisim has several handy debugging features: http://www.cburch.com/logisim/docs/2.1.0/guide/log/index.
- You can induce hazards (and thus test them) but using a single key for all operations.