Your First Accelerator: Matrix Sum

CS250 Laboratory 3 (Version 100913)
Written by Ben Keller

Overview

In this lab, you will write your first coprocessor, a matrix sum accelerator. Matrices are common data structures in a wide range of fields, and basic operations like row and column summations are frequent operations performed on them. Your accelerator will take advantage of the redundancy inherent in a sequence of row summation (rowsum) and column summation (colsum) commands to perform these commands considerably faster (in aggregate) than a scalar processor could. To accomplish this, your accelerator will buffer and store intermediate summation results. For the first part of this assignment, you will use registers (flip-flops) to implement the storage for these buffers. Later, you will replace the flip-flop arrays with SRAM macros to produce a smaller and more energy efficient implementation.

Deliverables

This lab is due **Tuesday, October 15 at 2PM**. The deliverables for this lab are:

- (a) your working Chisel RTL checked into your private git repository at Github
- (b) Reports (only!) generated by DC Compiler, IC Compiler, and PrimeTime PX checked into your git repo for the three implementations of your design
- (c) written answers to the questions given at the end of this document checked into your git repository as `writeup/lab3-report.pdf` or `writeup/lab3-report.txt`

You are encouraged to discuss your design with others in the class, but you must write your own code and turn in your own work.

Accelerated Rowsum and Colsum

Your accelerator will act on an n-by-n matrix consisting of 64-bit unsigned integers stored sequentially in memory. Based on instructions passed in from the Rocket scalar core, it should access memory to fetch the relevant elements of the matrix along a specified row or column and return the sum of the row or column to the scalar core. For the purposes of this lab, each matrix will be the target of several rowsum and colsum commands. In particular, your accelerator will receive instructions for several rowsums followed by several colsums (or vice versa). This will allow you to speed up the computation of the second set of sums by avoiding redundant memory accesses.

Consider the example 3-by-3 matrix shown in Figure 1. To compute an unaccelerated rowsum or colsum would take at least three cycles, because accessing each of the three elements in a row or column requires a separate memory access (the interface to the memory subsystem is 64 bits wide). Now consider the sequence of rowsums and colsums shown in Table 1. The three rowsums are filled by sequential memory accesses as described above. However, while the results of the rowsums are accumulated, the partial colsums are also accumulated and recorded in anticipation of future colsum commands. By the end of the third rowsum, each colsum has been completely computed and can be returned immediately upon request, without any further memory accesses or computation.

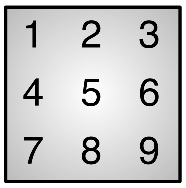

Figure 1: A 3-by-3 matrix populated with integer values.

Table 1: An accelerated sequence for computing sums for the matrix shown in Figure 1.

Sum	Result	Memory Accesses	Partial Colsum 0	Partial Colsum 1	Partial Colsum 2
Rowsum 0	6	3	1	2	3
Rowsum 1	15	3	5	7	9
Rowsum 2	24	3	12	15	18
Colsum 0	12	0	12	15	18
Colsum 1	15	0	12	15	18
Colsum 2	18	0	12	15	18

Note that this same technique can be used whether rowsums precede colsums or vice versa. The technique is also useful even when the first set of commands are not exhaustive. (For example, if only two rowsums had been computed in the example above, only one memory request would be required to compute each colsum.)

The RoCC Interface

In the Rocket coprocessor interface, the Rocket core processes custom Rocket Custom Coprocessor (RoCC) instructions and passes requests to the accelerator. In this lab, you will not instantiate a full Rocket core, but instead wrap your accelerator in an emulated RoCC interface. Your accelerator will be responsible for handling three distinct coprocesor instructions: SETUP, ROWSUM, and COLSUM.

RoCC Instructions

In general, 32-bit RoCC instructions extend the RISC-V ISA and are formatted as shown in Figure 2.

31 25	24 20	19 15	14	13	12	11 7	6 0
funct7	rs2	rs1	xd	xs1	xs2	rd	opcode
7	5	5	1	1	1	5	7

Figure 2: The RoCC instruction encoding.

The *xs1*, *xs2*, and *xd* bits control how the base integer registers are read and written by the coprocessor instruction.

If *xs1* is a 1, then the 64-bit value in the integer register specified by *rs1* is passed to the coprocessor. If the *xs1* bit is clear, no value is passed over the RoCC interface. The *xs2* bit similarly controls whether a second integer register specified by *rs2* is read and passed to the RoCC interface.

If the *xd* bit is a 1 and *rd* is not `x0`, the core will wait for a value to be returned by the coprocessor over the RoCC interface after issuing the instruction to the coprocessor. The value is then written to the integer register specifed by *rd*. If the *xd* is 0 or *rd* is `x0`, the core will not wait for a value from the coprocessor.

In all cases, bits 31–7 of the instruction are also passed to the coprocessor for further decoding, along with a 2-bit field indicating the major opcode (*0/1/2/3*). The coprocessor is responsible for signalling illegal instructions back to the core. Note that only 2 bits of the opcode are passed to the coprocessor, as five bits are required to define the instruction as a custom RoCC instruction.

RoCC Instructions for Matrix Sum

Your coprocessor will implement three distinct instructions: SETUP, ROWSUM, and COLSUM.

SETUP

The SETUP instruction defines the size of the (square) matrix that the accelerator will operate on, as well as the memory location of the first element in the matrix. The SETUP instruction also indicates that the accelerator should discard any state retained from previous matrix operations.

- **opcode** is set to 0 for all matrix sum instructions.

- **xd** is set low, as no return is expected from the SETUP command.
- **xs1** is set high. **rs1** contains the starting memory address of the matrix.
- **xs2** is set high. **rs2** contains the size of the matrix (up to 64).
- **funct** is set to 0 to indicate the SETUP command.

ROWSUM

The ROWSUM instruction indicates that your accelerator should sum the values of the indicated row of the matrix, and return the 64-bit result to the destination register. In case of an overflow, the accelerator should return the maximum 64-bit unsigned integer value (i.e., 0xFFFFFFFFFFFFFFFF).

- **opcode** is set to 0 for all matrix sum instructions.
- **xd** is set high. **rd** contains the destination register for the command; the accelerator must return this destination register value along with the result.
- **xs1** is set high. **rs1** contains the (0-indexed) row to be summed.
- **xs2** is set low.
- **funct** is set to 1 to indicate the ROWSUM command.

COLSUM

The COLSUM instruction indicates that your accelerator should sum the values of the indicated column of the matrix, and return the 64-bit result to the destination register. In case of an overflow, the accelerator should return the maximum 64-bit unsigned integer value.

- **opcode** is set to 0 for all matrix sum instructions.
- **xd** is set high. **rd** contains the destination register for the command; the accelerator must return this destination register value along with the result.
- **xs1** is set high. **rs1** contains the (0-indexed) column to be summed.
- **xs2** is set low.
- **funct** is set to 2 to indicate the COLSUM command.

Instruction Ordering

For the purposes of this lab, you can assume that the SETUP command will be followed by either:

- zero or more consecutive ROWSUM commands and then zero or more consecutive COLSUM commands, or
- zero or more consecutive COLSUM commands and then zero or more consecutive ROWSUM commands.

Each block of consecutive ROWSUM or COLSUM commands may not be in matrix order, and the commands may not sum every row or column of the matrix, but no ROWSUM or COLSUM command will be repeated after a given SETUP command.

RoCC Interface

Your accelerator will interact with the Rocket processor and the shared memory system via the standard RoCC interface. Each portion of the interface is decoupled by standard queues and ready-valid signals (with one minor exception, detailed below). You can view the Chisel implementation of the RoCC interface in the Rocket source file `rocc.scala`.

The Rocket core initiates an accelerator command by passing most of the RoCC instruction directly to the accelerator, as well as the relevant register values.

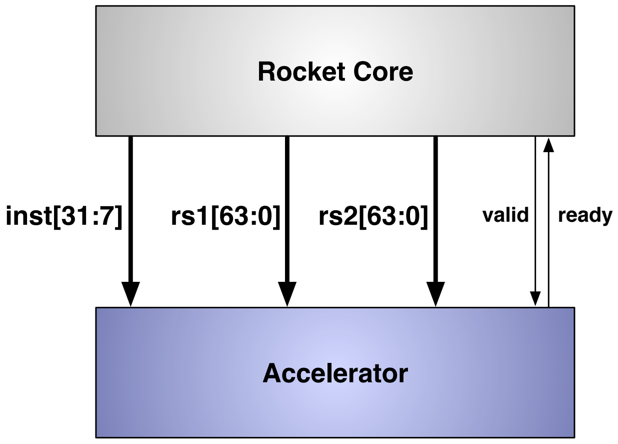

Figure 3: The RoCC request interface.

- **inst** includes all but the last bits of the RoCC instruction. For the purposes of this lab, you need only access the *funct* field of those bits to distinguish between the different types of commands. Note that you can take advantage of Chisel's nested interface syntax and address each instruction field by name, rather than by selecting bits from the bus (e.g., `io.cmd.bits.inst.funct`).
- **rs1** and **rs2** contain the contents of the registers specified in the instruction if *xs1* and *xs2* are high, respectively. Note that if the control signals in the instruction are low, then these fields will not contain valid data, even though a single valid signal is responsible for the entire interface.

If the instruction supplied to the accelerator set the *xd* bit, then the accelerator must eventually supply a response over the RoCC response interface.

- **rd** is the destination register supplied by the original instruction.
- **data** is the 64 bits of data to be written to that register.

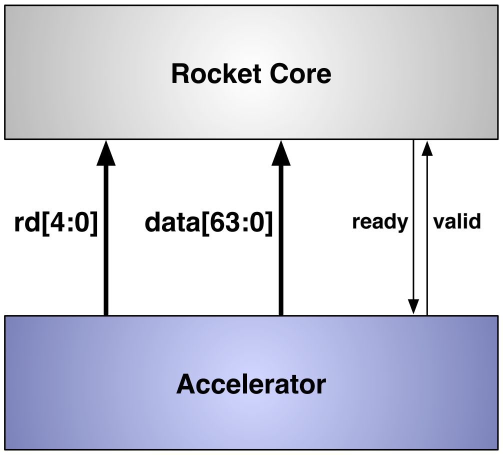

Figure 4: The RoCC response interface.

Your accelerator will use the standard Rocket interface to communicate with the DCache. This interface is named HellaCacheIO (HellaCache is the name of the non-blocking L1 data cache used in the Rocket core). You can view the Chisel implementation of this interface in the Rocket source file `nbdcache.scala`. Many of signals in this interface can be ignored for this lab. (In fact, setting some of these signals to values other than their defaults may make your memory interface non-functional.) The relevant signals that you should use to send and receive memory requests are detailed below.

The relevant signals that you need to send memory requests are shown in Figure 5.

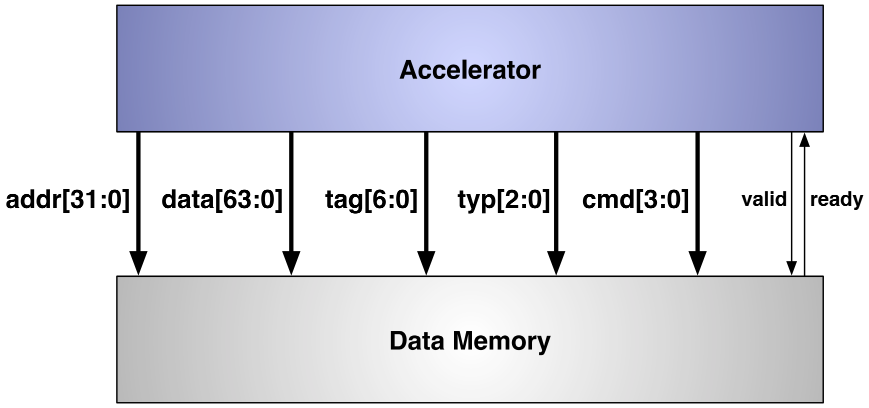

Figure 5: The DCache request interface.

- **addr** is the memory address targeted by the load or store.
- **data** is the 64-bit data value to be stored in memory. Since you are only performing loads in this lab, you can set this to 0.

- **tag** is a 7-bit tag that your accelerator must assign to each request. Since memory requests may not be fulfilled in the order in which they were requested, the tags will allow you to associate responses with their initiating request.
- **typ** describes the width of the request. You should set this to `011` to specify a 64-bit memory request. (Other values would cause fewer than 64 bits to be loaded or stored.)
- **cmd** specifies the type of memory request, typically a load or a store. You should set this to `0000` to specify a memory read. (`0001` would be a write.)

Note that you can use the constant values defined in `uncore/src/main/scala/consts.scala` to set *typ* and *cmd*. For example, you could set *typ* to `MT_D` to specify a 64-bit response instead of setting it to `011` directly.

The relevant signals needed to receive memory responses are shown in Figure 6.

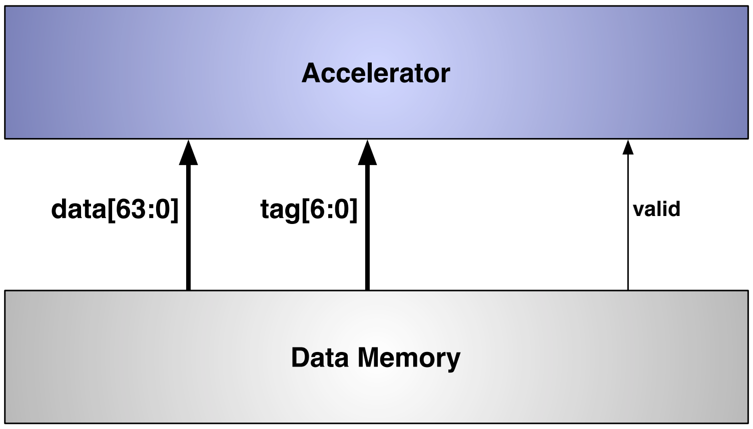

Figure 6: The DCache response interface.

- **data** is the data associated with a memory load response.
- **tag** is the tag that was sent with the response. Note that the memory subsystem returns a tag on a completed store request as well as a completed load request.

The memory response interface has only a valid signal, not a ready-valid pair. This means that your accelerator cannot exert back-pressure on the memory system, and must be ready to accept a new memory response every cycle. (In other words, it must always be "ready".) This should not be a problem for your matrix sum accelerator, as it should be able to fully process each returned data value in just one cycle.

Your accelerator provides two control bits of information back to the Rocket core.

- **busy** must be held high while the accelerator has active memory requests in flight. In this lab, you should assert *busy* for the duration of the accelerator computation, regardless of the particulars of the memory subsystem.

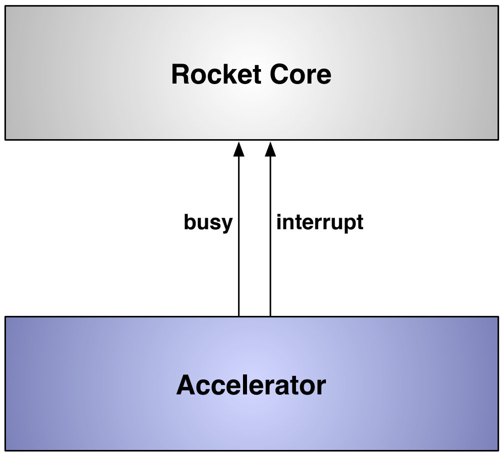

Figure 7: The RoCC control interface.

- **interrupt** allows the accelerator to trigger a system interrupt in the event of an invalid instruction or other problem. You will not need to trigger any interrupts in this lab, so you should hold this signal low.

Accelerator Hardware

Your accelerator can be conceptualized as three distinct hardware modules: a simple accumulator, a storage element for the partial sums, and a more complicated control block. Figure 8 shows the relationship between the three modules.

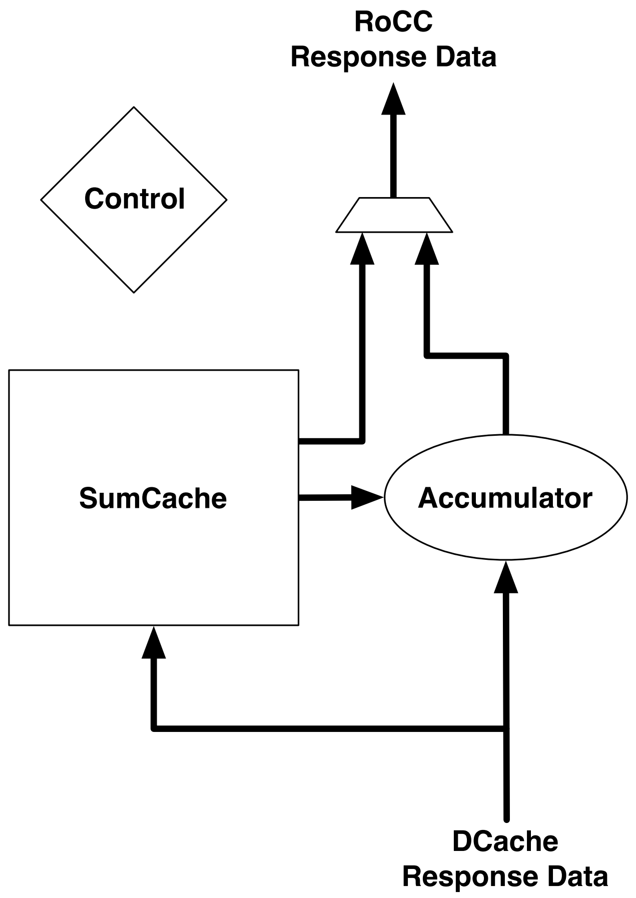

Figure 8: A system block diagram. Control signals are not shown.

Accumulator

The accumulator block is responsible for summing matrix values passed in by the memory interface. After resetting or initializing to 0, it simply sums its input with the stored 64-bit value. Note that your accumulator should handle overflows by outputting the maximum 64-bit value. To do this, you should use a 65-bit adder, with the MSB acting as the accumulator bit. If you do reach overflow, you can signal the control block to return this sum result to the Rocket core immediately, potentially skipping additional memory accesses and summations. Sometimes, only partial sums will have been completed when the matrix operations switch direction. In this case, the partial sum will need to be passed to the accumulator so that the final sum can be completed.

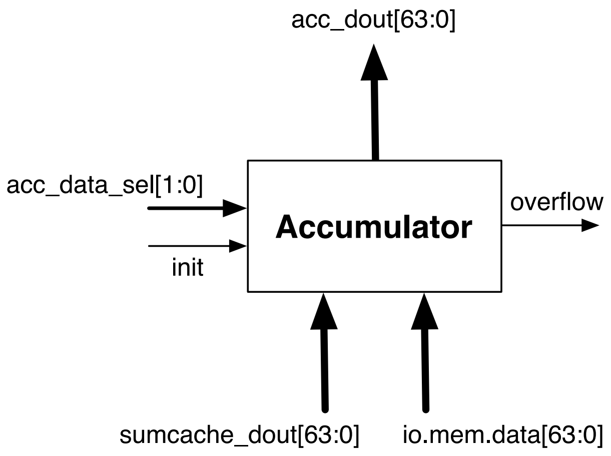

Figure 9: The accumulator module.

Partial Sum Cache

The partial sum cache is a local storage element that should store the partial sums in the direction orthogonal to the first set of matrix sums, as described in the algorithm above. This block must also be able to increment a particular sum each cycle as additional matrix values are returned from memory, handling overflows as described above. This element can be constructed out of an adder, several multiplexers, and a number of registers equal to the largest allowable matrix size. The partial sum cache will take advantage of Chisel's parameterized generation feature. It takes an input parameter called `maxSize` that represents the dimension of the largest possible matrix that your accelerator might be asked to process. Be sure to parameterize your design to generate only the hardware necessary to handle a matrix of this maximum size.

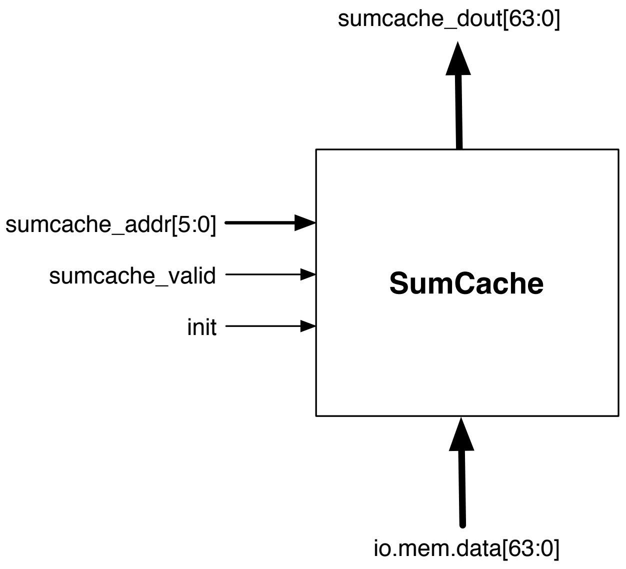

Figure 10: The sum cache module.

Note that for a maximum matrix size of 64, your cache will have 64 64-bit registers, or 4096 flip-flops in total. That's a lot of flops! You will eventually modify this module to use a two-ported SRAM in place of this large block of state, saving considerable area.

Control

Your control block is responsible for handling all of the control bits in the RoCC and DCache interfaces. It must interpret commands sent by the processor and send appropriate memory requests to load data for the accumulator and cache. It must also keep track of whether to return the result from the accumulator, or the cached precomputed result.

Take care in your handling of memory requests, particularly given that memory responses may come back in an order different than the order of requests. It is recommended that you implement your control block such that your coprocessor handles one instruction entirely before moving on to the next one. You are welcome to design a scheme in which you have multiple requests in flight simultaneously, but this may complicate your control scheme considerably.

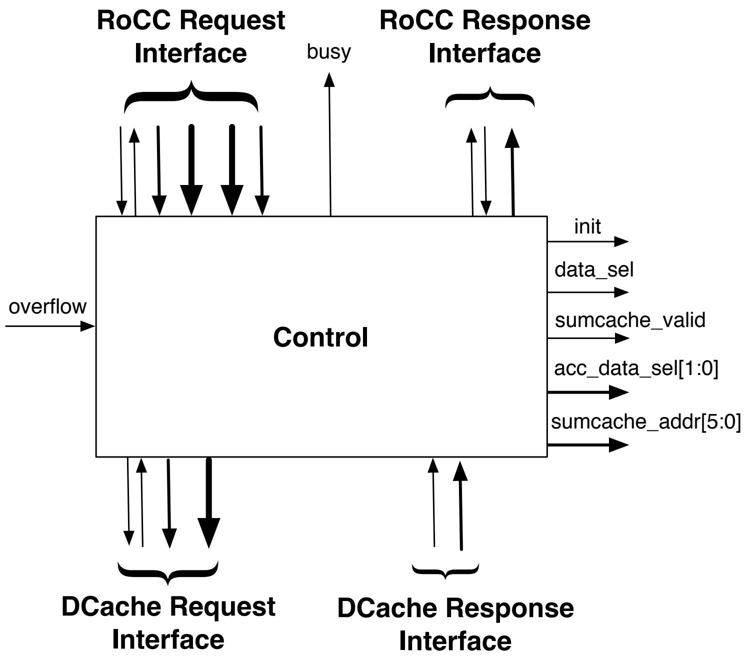

Figure 11: The control module.

You may find it helpful to define additional components within your control block. For instance, you could create a submodule that is solely responsible for handling the DCache interface.

Building Your Accelerator

To get the files for Lab 3, go to the working directory you used for Lab 1 (or create a new one, following the instructions in the Lab 1 handout) and run `git pull template master`. That should create a `lab3` directory: this is the top-level directory for your Lab 3 design files. The skeleton

Chisel code we provide will compile without errors, but will fail verification.

We provide code templates for the module hierarchy described in the previous section. `sumaccel.scala` is the top-level module for the accelerator. `accum.scala` contains the accumulator circuit. `sumcache.scala` contains state elements for cached partial sums. `control.scala` contains the control block. You are welcome to define additional modules or signals between them.

One way to get started constructing your accelerator is to first build a “slow” accelerator that does not calculate and store partial sums. By implementing only the accumulator and a simplified control scheme, you should be able to construct hardware that returns the correct response to each instruction. This approach would allow you to master the accelerator interfaces before adding the complexity of your sum caching scheme.

Test Infrastructure

Unlike the previous labs, you will be performing the bulk of your testing and debugging using Scala and the Chisel `Tester` class. The `Tester` class allows you to write test cases in Scala, and compile and run them in C++ just as you build your design itself. Additionally, we have implemented a new class, `AdvTester`, that extends `Tester` and provides additional functionality to the testharness.

To test your design, type `make test` in the top-level Lab 3 directory. This will instantiate an accelerator block with `maxSize=64`. The Chisel testharness will then emulate the role of the Rocket processor and memory, sending responses and checking the results sent by your accelerator.

We have not implemented a separate C++ testharness for this lab (although you are welcome to write one yourself if you would like). The Verilog testharness is not as robust as the provided Chisel tests, so you should use it only as a check after you have fully verified your design using the Scala tester.

Using the SumAccel Tester

The testharness that defines the interface and testvectors that are applied to your accelerator is located in `test_sumaccel.scala`. Open up this file and take a look at the code. Near the top, you can see where the three instructions for your accelerator are defined, as well as the instantiation of the `AdvTester` that will drive your accelerator.

Scroll down until you find `defTests`, where the testvectors for the testharness are defined. Let’s examine one of these tests:

```
...
println("Test 3: Execute 3x3 matrix (no address offset) rowsum on row 1")
Cmd_IHandler.inputs.enqueue(TestCmd.rowsum(1, Some(12)))
until( Cmd_IHandler.isIdle && !dutBusy ) { instantMemory.process() }
assert(!Resp_OHandler.outputs.isEmpty, "Test 3 failed: Expected response")
val t3_resp = Resp_OHandler.outputs.dequeue()
assert( t3_resp.data==do_rowsum(1, testmatrix_n3, 3), "Test 3 failed... )
assert( t3_resp.rd==12, "Test 3 failed: Bad rd returned" )
...
```

The testing steps are readable from this Scala code. First, a rowsum command is put in the queue to send to the accelerator. Its arguments describe which row to sum (row 1) and which register value to return in the `rd` output (register 12). The testharness then waits for the response to come back from the accelerator. A series of assert statements check that a response is returned that matches what the correct response ought to be.

Note that this test calls a memory emulator named `instantMemory`. This memory always returns a response a single cycle after requests are sent in, so there are no delays and responses are always returned in the order in which the corresponding requests were received. Later tests instead use a marginally more realistic memory named `slowMemory`. This emulated memory does not accept requests every cycle, and may delay responses by several cycles. This means that requests submitted to this memory may not be returned in the same order that they were sent out. Your accelerator should function correctly with either memory model.

You can write your own tests by copying the provided examples and modifying them. Several provided functions may be helpful in this process, including `generate_test_matrix` and `print_matrix`. Take a look through the remaining code for more details.

Note that the Scala testharness can take quite some time to run for the longer tests. Feel free to modify or comment out these tests while you are debugging your accelerator. However, make sure that your final code can pass each of the tests that was originally provided before you submit it.

Debugging with the SumAccel Tester

If your accelerator fails one of the provided tests, the tester will print a message to the console based on the assert statement that did not pass, providing some information about the mode of failure:

```
ASSERT FAILED: Test 7 failed: Bad response for col 33
```

However, you may need more information to debug your design. One way to do this is to add more assert statements to query the state of the system and print more information to the console. You can access signals within your accelerator by querying the `dut` object (e.g., `dut.control.io.dout`).

By default, the `Tester` class prints all input and output signals to the console each cycle. We have disabled this for the provided test infrastructure, but you can re-enable it by adding the following line anywhere within the provided test cases:

```
  printTrace = true
  // Tester prints inputs and outputs each cycle...
  printTrace = false
  // Tester stops printing inputs and outputs.
```

In some cases, it may be helpful to view all of the internal signals of your accelerator. Chisel can output a `vcd` file that can be converted into a `vpd` file for use with a waveform viewer. To generate such a waveform, type `make test-debug` in the top-level Lab 3 directory. This will generate a file called `SumAccel.vpd` that you can view with DVE.

Unit Testing

While we are using Scala testharnesses to test your entire design in this lab, the `Tester` class is perhaps most useful for writing unit tests that test the functionality of individual submodules in your code. This helps you debug each module separately before attempting to assembly them.

We have written a simple unit test for your accumulator module. This test is located at the end of the `accum.scala` file. The test vectors (inputs and expected outputs) are defined in the `accumulatorTest` class. The code we have provided simply uses the numbers 1 to 10 as inputs for the `accumulator` component, accumulates them cycle by cycle, and checks the expected result against the component's output.

To build and run this test harness, type `make test-accum` in the top-level Lab 3 directory. Note that the unit test we have provided is not very robust: it only checks small inputs for a few cycles, does not exercise the `init` or `overflow` signals, and only accepts values to one of the input ports of the module. If you'd like, you can improve this unit test to ensure that your accumulator functions properly before testing the whole system.

While probably not necessary for this lab, writing unit tests for each of your modules is generally a good idea for large systems. To create a new Scala test harness, you need to implement a subclass of `Tester` for the component that you want to test. You will also need to add a few lines to `top.scala` and `Makefile` so that you can invoke the unit test. If you'd like to try this out, use the existing lines for the accumulator unit test as a template.

The VLSI Flow

Once you have verified your Chisel, you should build it using the standard VLSI toolflow described in Labs 1 and 2. The `vlsi/build-rvt` directory contains subdirectories with Makefiles. Use the Makefile in `vcs-sim-rtl` to build and verify a Verilog version of your hardware. By stepping through DC, ICC, and PrimeTime, you will be able to produce a layout for your chip, as well as reports on area, power, and timing. Be sure to simulate your design with VCS after each stage to make sure it still works.

Multi-V_t Flow

Thus far in CS250, you have used standard cells from a single library to build your design. Now you will have an opportunity to take advantage a modern VLSI power-saving techinque: multi-V_t design. The Synopsys educational libraries provide different "flavors" of standard cells. In this lab, you'll be using both regular-V_t and high-V_t standard cells. Recall that transistors with a higher threshold voltage are slower but leak less. If provided with multiple standard-cell libraries, the tools will use the faster regular-V_t devices only on the critical path, and will swap in slower devices elsewhere to save power.

Take a look at the top-level `Makefrag` in the `vlsi/build-multi-vt`. Note that different libraries are now included. The other build scripts are symlinked to the original flow to save space. By linking to each library and providing a few key commands, Design Compiler can take advantage of the different types of standard cells to better optimize the design.

Build your design (using the same Chisel code) using the multi-V_t flow.

SRAM

As noted in the hardware description, implementing all of your sum caching state as flip-flops introduces considerable overhead in area and energy. With some simple modifications, you can instantiate an SRAM instead.

Chisel's `Mem` class can be used to implement memories with arbitrary numbers of read and write ports. Writes always happen on the positive clock edge, but the timing of reads can be either combinational (result available during the same cycle as address is provided) or sequential (address is registered on positive clock edge, result available during the next cycle). However, the ASIC SRAMs we will be using in this course all use sequential reads. Therefore, to enable the Chisel compiler to map a `Mem`-generated memory to an ASIC SRAM macro (as opposed to an array of flip-flops), you must specify sequential read timing and put a register at the output of the SRAM. Below is an example of how to do this for a one read, one write memory (the kind you should use in your design). You should instantiate the memory for your sum cache in a similar fashion. For more about the `Mem` class, see Chapter 9 of the Chisel manual.

```
val ram1r1w = Mem(UInt(width = 64), 64, seqRead = true)
val reg_raddr = Reg(UInt())
when (wen) { ram1r1w(waddr) := wdata }
when (ren) { reg_raddr := raddr }
val rdata = ram1r1w(reg_raddr)
```

In addition to using the `Mem` class, you may need to make one other modification to your sum cache microarchitecture. When using registers, it is sensible to "read" from a particular register output, modify the value, and "write" to that register's input, all in the same cycle. However, it is not permitted to read from and write to the same address in an SRAM in the same cycle. Furthermore, a sequential SRAM takes one full cycle to return the data from the provided address. Therefore, you will need to add some registers to correctly time reads and writes so that your read-modify-write operation performs correctly. Make sure that your control logic accounts for cases in which you might want to read from your cache before the latest value to an address has actually been written!

You must instantiate 4096-bit SRAM as sized for a `maxSize` of 64, as we have only given you one SRAM macro to use in this design. The macro is called `sram8t64x64`; it is dual-ported, 64 bits wide, and 64 entries deep. Its library files and Verilog model (used for simulation) are located in the `generated-rams` directory. This macro was generated using Cacti, an SRAM modeling suite. Make sure you the memory you instantiate using `Mem` has the same depth and width at the SRAM macro. If you had been using 65-bit registers to keep track of overflow, you will need to have a separate register bank of overflow bits, as these extra bits will not fit in the SRAM.

If you use the `Mem` class in a way that does not imply a sequential memory, Chisel will just instantiate an array of flip-flops instead. Take care to define and register the ports in the order shown above. Be sure to check the generated Verilog files to see that the SRAM macro is being instantiated.

Once you have successfully instantiated an SRAM, build your design with the provided flow in `build-sram`. You can look at the `Makefrag` to see how the build setup has been modified to include this extra macro.

Submission and Writeup

Write (or reuse) a python script to collect the data necessary to fill in the following tables. Use the reports generated by PrimeTime to obtain the power data, and the reports generated by ICC for area and timing information.

Area		RVT	Multi-VT	SRAM
Sum Cache	um^2			
Total	um^2			
Cell Count	RVT			
Cell Count	HVT			

Power		RVT	Multi-VT	SRAM
Entire Design				
Leakage	uW			
Total	uW			
Sum Cache				
Leakage	uW			
Total	uW			

Questions: Multiple VT Flow

1. What impact does switching from a single VT to a multi-VT flow have on area? Does this match your expectations?

2. What effect does switching to a multi-VT have on the power consumption of your design? How much does it reduce leakage power?

3. Can you think of a reason why you wouldn't want to use multiple VT cells to implement a design?

4. What portion of the cells used in the multi-VT version of your design are regular VT? Does this match your expectations?

5. Where in your design do regular VT cells appear? Why do you think this is the case?

Questions: SRAM

1. How much area do you save by using an SRAM instead of registers to implement storage for your window buffer?

2. What is the area of the SRAM macro?

3. Assuming that the individual bitcells have an area of 0.415 um^2, what is the efficiency (area used for bitcells over total area) of this SRAM macro?

4. Would you expect this SRAM to use 6T or 8T bitcells, and why?

5. What are the other components of an SRAM, besides the bitcells?

Questions: Power and Performance

1. Estimate how much faster your accelerator can calculate all row and column sums for a 64-by-64 matrix than an unaccelerated computation performed by the Rocket core. How much acceleration do you get for other use cases?

2. Assuming a clock frequency of 1 GHz, calculate the amount of energy (in Joules) used to compute all row and column sums for a 8-by-8 matrix for all three versions of your design. Then extrapolate this result to determine the energy required to compute all row and column sums for a 64-by-64 matrix.

3. How might your design change if each matrix element were 8 bits wide instead of 64 bits wide? Would you see more or less "acceleration" relative to the Rocket core in this case?

Submission

To complete this lab, you should commit the following files to your private Github repository:

- Your working Chisel code.
- The `reports` directories from DC, ICC, and Primetime.
- Your Python script.
- Your answers to the questions above, in a file called `writeup.txt` or `writeup.pdf`.

Some general reminders about lab submission:

- If you are using one or more late days for this lab, please make a note of it in your writeup. If you do not, your TA will assume that whatever was committed at the deadline represents your submission for the lab, and any later commits will be disregarded.
- Please note in your writeup if you discussed or received help with the lab from others in the course. This will not affect your grade, but is useful in the interest of full disclosure.
- Please note in your writeup (roughly) how many hours you spent on this lab in total.

Acknowledgements

Parts of this lab, particularly the section on SRAMs, were originally written by Rimas Avizienis for CS250 Fall 2012 Lab 2. The `AdvTester` Chisel class and much of the test infrastructure for this lab were written by Stephen Twigg.