Flexible FFT Optimization and RTL Generation in the Chisel Hardware Design Language

Stephen Twigg
John Wawrzynek, Ed.
Borivoje Nikolic, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-256
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-256.html

December 18, 2015

Acknowledgement

I foremost must acknowledge my advisor, John Wawrzynek, followed by the ASPIRE Laboratory and Berkeley Wireless Research Center, who supported these research efforts.

Research partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, HP, Huawei, LGE, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

Flexible FFT Optimization and RTL Generation in the Chisel Hardware Description Language

by

Stephen Twigg

A thesis submitted in partial satisfaction of the

requirements for the degree of

Masters of Science

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor John Wawrzynek, Chair
Professor Borivoje Nikolic

Fall 2015

Flexible FFT Optimization and RTL Generation in the Chisel Hardware Description Language

Abstract

Flexible FFT Optimization and RTL Generation in the Chisel Hardware Description Language

by

Stephen Twigg

Masters of Science in Computer Science

University of California, Berkeley

Professor John Wawrzynek, Chair

Using hardware generators to produce components for modern SoCs enables rapid design space exploration. Embedding these hardware generators in a high-level programming language allows for simpler code, more expressive generators, and correspondingly better quality of results. This thesis describes the use of Chisel (Constructing Hardware in a Scala Embedded Language) to construct an FFT generator. The parametrized generator demonstrates automatic parameter-dependent test generation and seamless system integration. An analysis of FFT scheduling is performed to show how various mathematical properties of the Cooley-Tukey FFT decomposition may be exploited to simplify needed hardware while assuring conflict-free bank accesses for every operation. The resulting circuit throughput, energy, and area are shown to be better than or on par with other FFT generators.

To my family and friends. In particular, my father who inspired me to study electrical engineering and my mother who has continued to support me unwaveringly.

Contents

List of Figures

List of Tables

Acknowledgments

I foremost must acknowledge my advisor, John Wawrzynek, followed by the ASPIRE Laboratory and Berkeley Wireless Research Center, who supported these research efforts.

Research partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, HP, Huawei, LGE, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

Chapter 1

Introduction

Efficient design of modern Systems-on-Chip (SoCs) is greatly aided by the ability to do rapid design space exploration (DSE) of system components. For certain systems targeted at high performance computing, hardware designers generally want to maximize system task throughput while keeping chip area and power consumption within physical limits. For other systems targeted for embedded radios in cellphones, the chip must fit within a tightly budgeted area, have sufficient task throughput and latency to actually be useful for communication on the specified protocols, and draw as little power as possible to maximize device battery life. Moreover, organizations are likely to build multiple systems, each with slightly different requirements but using similar components. Properly parameterizing components allows some non-recurring engineering costs spent for one system to be amortized over multiple systems. The DSE engine can simply adjust the parameters of components until the newer systems meet specified performance goals.

Writing the hardware descriptions of these parameterized components as generators in a high-level programming language allows the designer to write simpler code and more easily incorporate desired parameters. Simpler code reduces the chance for design errors, particularly when code is revisited multiple times over various projects. More parameters enable generators to emit components for a wider variety of systems and potentially improve optimal performance by enlarging the design space. This thesis explores the design of a Fast Fourier Transform (FFT) accelerator generator in the Chisel HDL (Constructing Hardware in a Scala Embedded Language). The FFT retains usefulness across a wide variety of applications from mobile communications to radio astronomy (for correlating data between multiple antennas) and numerical analysis.

In addition to its wide use, properties of the FFT make it particularly attractive for hardware acceleration. Divide-and-conquer algorithms describe how large FFTs can be calculated using compositions of many smaller FFTs [7]. These algorithms introduce the requirement that the composing FFT size factors the large FFT size. However, applications tend to pick FFT sizes with this constraint in mind and thus restrict the FFT to numbers composed of small roots such as 2, 3, and 5. For analysis, smaller FFTs with undesirable factorizations can generally be approximated by using a larger FFT composed of a more desirable factor-

ization. (This strategy does not quite work for communications applications due to the very tight restrictions on acceptable signal-to-noise ratios.) Thus, a general purpose processor with an attached fixed-size FFT accelerator can still make good use of the accelerator, even if the initially desired application FFT is of different size than the accelerator.

The hardware generator described in this paper uses the Cooley-Tukey decomposition algorithm to produce a fixed-size FFT accelerator intended to be attached to the Rocket Chip reference system [2] or used in a standalone configuration. In the current implementation, the FFT size must be a power-of-2. Parameters of the generator include FFT size, FFT decomposition radix, number of available butterfly execution units, number format (single precision floating point or parameterized fixed point), IO bandwidth, and whether or not to do the initial data reordering. The generator also accepts a high-throughput multithreaded option, which instantiates an extra set of memories so multiple FFTs can be processed simultaneously. The generator can also emit a full set of input and output test vectors for either the Chisel C testbench or SystemVerilog accelerator testbench to verify accelerator accuracy. These test vectors are automatically converted to a bitwise representation in the correct number format and, in the case where the final permutation is not done by the accelerator, appropriately shuffled.

The FFT generator described in this paper uses an integrated constraint solver, run at hardware generation time, to determine an optimal schedule for operations for performing the FFT in-place, avoiding the need for extra memory when storing intermediate results. The Scala component of Chisel is critical for seamlessly translating data from the constraint solver to instruction ROM entries.

1.1 Related Work

For software based FFTs, the two most popular frameworks are the Carnegie Mellon University Spiral project [8] and the FFTW (Fastest Fourier Transform in the West) project [3]. Both work by devising FFT plans offline for each FFT size during initial installation and benchmarking. These plans detail the macro parameters such as the FFT decomposition strategy as well as micro parameters such as operation blocking, etc., in order to tune to processor cache sizes and other compute system variables.

An available hardware-based accelerator fits cleanly into these schemes by making the planning stage aware that the certain FFT sizes performable on the accelerator can be done orders of magnitude faster. The heuristics engine in the planning stage would then naturally favor performing FFTs of that size. Of all the known and published major FFT projects, only the Spiral team crafted an FFT hardware accelerator. However, no papers could be found with released data on hardware and software co-tuning experiments. Other projects, such as Stanford Genesis, have also produced FFT generators partly as demonstrations of hardware design technologies [4].

The Spiral hardware accelerator is also highly parameterized. It supports both fixed and floating arithmetic. The FFT size can be set to most powers of two with various power-of-

two decomposition radices. Many variants of Cooley-Tukey as well as Bluestein's algorithm, although the latter only being used for non-power-of-two FFT sizes [9]. However, the The amount of execution units used can be controlled by adjusting the 'streaming width'. Note, the streaming width also modifies the assumed input and output bandwidth to the system (so the input and output bandwidth must scale with the amount of execution units). In contrast, the accelerator architecture described in this thesis decouples those concerns. The Spiral hardware architecture also supports a streaming architecture as well as the iterative architecture. The streaming architecture requires specific execution units be devoted to each stage of the FFT and yields slight memory savings when flowing multiple FFTs through the system while similarly tightly coupling IO bandwidth and execution unit parallelism. Thus, full use of all execution units requires a number of in-flight FFTs equal to the FFT size log base the decomposition radix. In contrast, the accelerator in this thesis only implements the iterative architecture. The iterative architecture is considered generally more useful by being more area-efficient (allowing full use of all execution units on a single FFT rather than requiring multiple in-flight FFTs), allowing finer decoupled control of execution unit count and IO bandwidth, and more naturally handling memory system stalls.

1.2 Cooley-Tukey FFT Primer

An FFT is merely a sped of version of the Discrete Fourier Transform (DFT). An N-point DFT will consume N pieces of real input data in the time domain and produce N-points of complex output data in the frequency domain. The frequency data is evenly spaced in the range of $[0, 2\pi)$ such that $X_n = \frac{2\pi n}{N}, \forall n \in Z_N$. A frequency data point may be calculated by:

$$X_n = \sum_{k=1}^{N-1} x_n * e^{-j2\pi kn/N}$$

Naive implementation will require $O(n^2)$ operations to calculate all N outputs. However, by splitting the summation and factoring out part of the complex number, intermediate results can be generated that share operations amongst outputs. The Cooley-Tukey decomposition demonstrates how an N-point DFT where $N = N_1N_2$ can be performed by first performing N_1 N_2-point DFTs, then N complex multiplications (some trivial) by roots of unity, and finally, N_2 N_1-point DFTs. Consider the case, which will be the sole focus for the rest of this thesis, that $N = r^a$ for some small number r and an integer a. The small number r is called the decomposition radix. If the decomposition is applied recursively, then the number of operations actually needed to produce all N outputs is $O(nlog(n))$.

If the decomposition is done with $N_1 = r$, the decomposition is said to be done in-time and, due to the rearrangement of the summations, ends up shuffling the input time data. If the decomposition is done with with $N_2 = r$, the decomposition is said to be done in-frequency, and shuffles the output time data. The architecture and algorithms in this thesis will assume the decimation-in-time approach as an arbitrary decision since, due to a certain

symmetry in the contrasted approaches, the same guarantees on avoiding bank conflicts when accessing memory apply equally to both via the same scheme. A more-thorough explanation of this is found in the references[7].

An 8-point FFT decomposed in-time using the Cooley-Tukey algorithm with a decomposition radix of 2 is shown in Figure 1.1. The crossed lines, forming a characteristic butterfly shape, indicate the two data points that must be computed by the butterfly execution unit to produce data for the next stage. The computation done is simply a radix-2 FFT. Not directly shown on the figure, but required for the algorithm, is that each intermediate data point is multiplied by some complex root-of-unity. However, the first input to a 2-point DFT must always be multiplied by 1 and thus each stage only requires 4 complex multiplications.

Elementary application of the naive DFT for N=2 and N=4 yields the following:

$$\begin{bmatrix} X_0 \\ X_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Since multiplication by 1, -1, j, and -j requires no real multiplications, these specific FFTs can be performed by only doing complex addition, which require only 2 real additions or subtractions each. The 2-point DFT requires 2 complex additions and, after clever factoring, the 4-point DFT requires 8 complex additions. This demonstrates why radix-2 and radix-4 decomposition is so favorable. Any other DFT size requires internal complex multiplications. For radix-2, N/2 non-trivial complex multiplications are required each stage. For radix-4, 3/4N non-trivial complex multiplications are required each stage but there are half as many stages. Overall, 25% fewer complex multiplications are required for radix-4 versus radix-2 thus making radix-4 preferable.

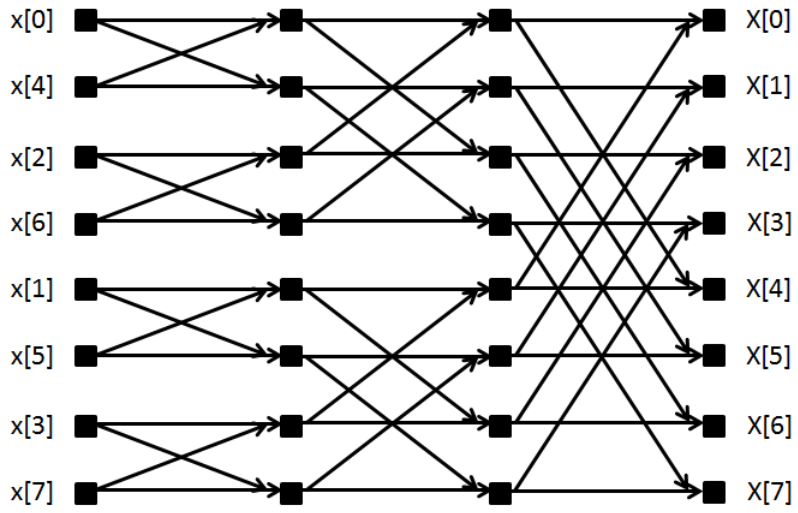

Figure 1.1: 8-point FFT decomposed-in-time using Cooley-Tukey with decomposition radix of 2. Input time data is on the leftmost column and output frequency data on the rightmost. Note the shuffling of the input data.

Chapter 2

Accelerator Architecture

2.1 Overview

Figure 2.1 shows the overall architecture for the accelerator. Figure 2.2 shows the architecture for a single FFT core, which includes an execution unit and scratchpad memory. The diagram shows the accelerator in high throughput mode with two operation controllers, with multiple execution units, and using radix-2 decomposition—during execution two points per FFT core are scheduled. The busy signal (internal connections omitted for clarity) asserts true whenever there is any operation in any operation controller or the work distribution module. This signalling allows the attached processor to know when the accelerator is done with all current FFTs.

During operation, the attached processor or testbench sends a starting memory address and stride through the decoupled command port to the work distributor. The work distributor routes the command to an available operation controller. A single operation controller is responsible for shepherding a single FFT operation through the accelerator. Similarly, the load, store, and execution managers each individually handle data for a single FFT operation at any given time that is not being touched by another manager. The operator controller first commands the load manager to load internal scratchpad memory dedicated to that controller and located inside the FFT cores. The load manager uses the external write router to ensure the scratchpad write requests are sent to the correct scratchpad memory. Then, the operation controller commands the execution manager to use the execution pipeline in the FFT cores to process data in their internal scratchpads. Finally, the operation controller commands the store manager to send data from the internal scratchpad memories out through the memory port. The store manager uses the external read manager to route read requests to the correct scratchpad.

Note that single precision floating point operations are implemented using the parameterized Hardfloat library and thus require 33 bits to represent one single precision floating point number [5]. The load and store managers also have logic to convert incoming and outgoing data to and from IEEE complaint floating point and the internal Hardfloat representation.

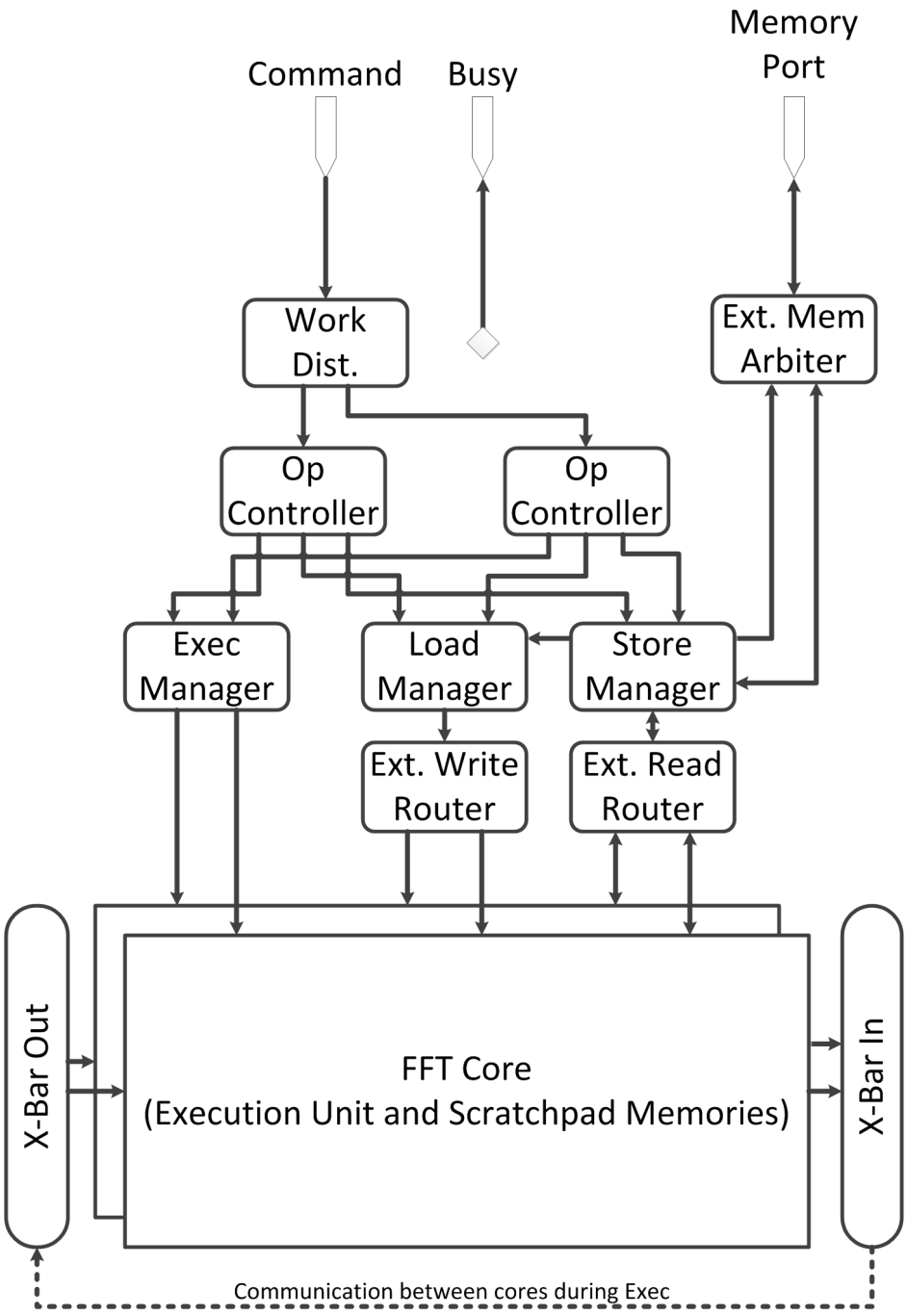

Figure 2.1: Overall architecture of FFT hardware accelerator.

The overall system has limited memory bandwidth and thus the load manager and store manager must use an arbiter to share access to the external memory port. In normal operation, the two limited resources are the external memory port and the execution units. Therefore, maximum throughput requires only two operation controllers to ensure near 100% utilization of either the execution units or external memory port. Which component has 100% utilization depends on other system parameters. As memory operations are $O(n)$ and execution steps $O(n \log(n))$, larger FFT sizes favor 100% utilization of the execution units.

2.2 Operation Controller

The operation controller itself is relatively simple and fairly small. It consists of a 7-state finite state machine and registers for holding the operation information, namely the external memory start address and stride. The first state is the idle state where the operation controller is waiting for the accelerator command queue to send it an FFT command. After receiving a command, the controller switches into the pre-load state where it waits for the load manager to accept the generated load command. While the load manager is working, the operation controller waits in a loading state. After the load manager completes, the operation controller enters into the pre-execute state where it waits for the execution manager to accept the generated execute command. While the execution manager (and all the execution units) is working, the operation controller waits in an executing state. Similarly, after the execution manager signals completion the execution goes through a pre-store and storing state while interfacing with the store manager. After the storing state is complete, the operation is finished and the operation controller returns to the idle state.

2.3 Core and Execution Units

The individual cores are modeled after a SIMD execution pipeline. The execution manager sends a 'PC' to the core. The 'instruction fetch' stage reads the instruction ROM to find the memory locations of data and twiddle factors. In the 'instruction decode' stage, requisite data is read from the scratchpad memory and the twiddle factor ROMs. In the execute stage, the butterfly execution units use the twiddle factors and data to compute output data. In the writeback stage, data is routed through the crossbar back to scratchpad memories of cores for storage and use in later compute stages. For radix-2 decomposition, two pieces of data using one twiddle factor are read, computed, and written each cycle. For radix-4 decomposition, four pieces of data using three twiddle factors are read, computed, and written each cycle. Larger radices are not implemented as they require additional complex multiplications. Due to use of the constraint solver, a new operation can be issued nearly every cycle.

As discussed in the FFT primer, an FFT butterfly is composed of two stages: scaling all input data, except the first, by twiddle factors and then performing a 2-point or 4-point DFT on that intermediate data. The former operations require both real multiplications

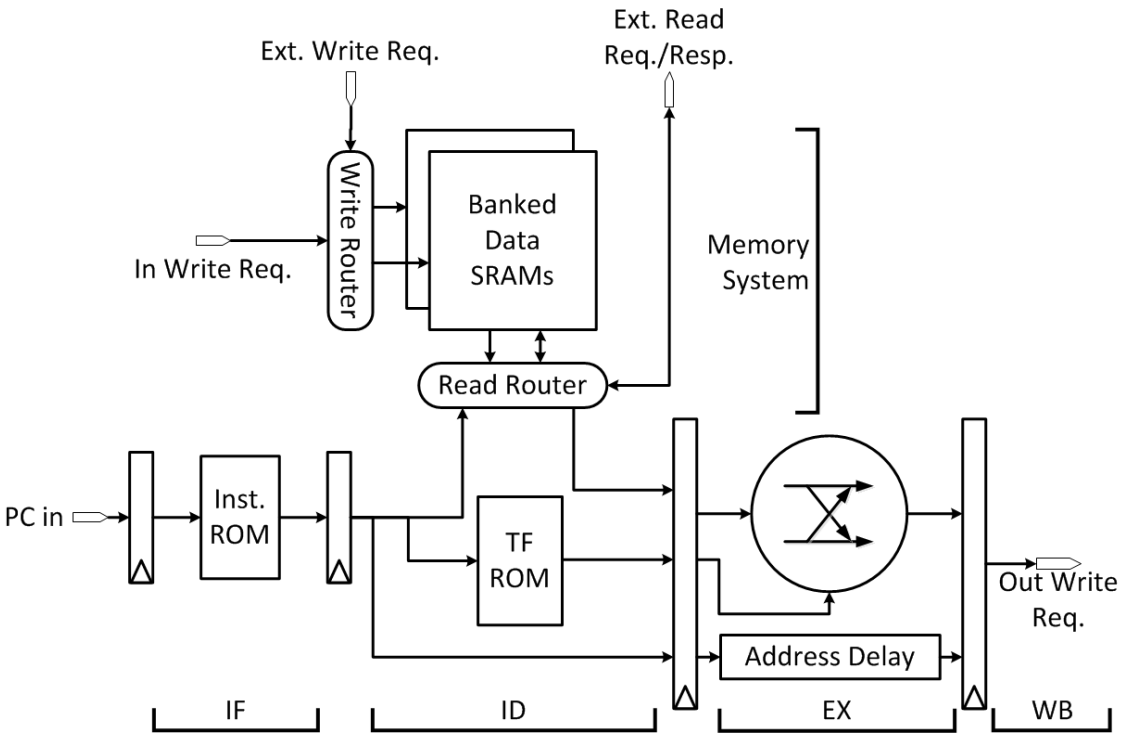

Figure 2.2: Internal architecture of FFT core with ROMs, memory banks, and butterfly execution units

and additions whereas the latter operations require only real additions and subtractions. In order to facilitate parameterizing the design to support both floating point and fixed point operations, this module uses a layer of abstraction built on top of the raw arithmetic operators provided by Chisel. A Scala interface called ButterflyExecutor was defined with function prototypes to for describing how to perform addition, multiplication, or fused multiplication-addition in hardware given the concrete number format. The execution unit module then defines complex multiplication and addition operations using the functions of this interface. Finally, the hardware is built by calling the complex multiplication and addition operations as needed to describe the FFT arithmetic. In implementation, the fixed point executor uses the raw shift, multiply, and add operations in Chisel. The floating point executor is somewhat more involved and constructs modules in the Hardfloat library then connects function inputs and outputs to the IO of those modules. This layer of abstraction ultimately allows the accelerator to seamlessly select a different number format by supplying a different executor to the butterfly module.

For implementing complex multiplies, design space exploration demonstrated three multiplies provides superior results for fixed point whereas four multiplies provides superior results for floating point. The latter result occurs because the four multiply implementation allows more adds to be fused into multiplies and a smaller difference in sizing between multipliers

and adders. The top-level configuration object will automatically select which complex multiply implementation to use and does not bother exposing this parameter to the user. Next, these complex multiplies and adds are used to construct the butterfly datapath. Finally, a configurable amount of registers are appended to the butterfly module to allow physical design tools to reduce the critical path via register retiming. Single precision floating point operations are implemented using a parameterized floating point library [5].

2.4 Memory System

In a single core, the SRAM has a number of banks equal to the radix times the number of operation controllers. Over the whole accelerator, the total SRAM depth is FFT size times the number of operation controllers. This is the minimum amount of memory necessary so that each operation controller can be handling its own FFT. All SRAM widths are two times the number format width, as complex data requires both a real and an imaginary component. Each SRAM bank is dual ported with a single read and single write port.

The data size of the external memory port is the minimum of 512 (the most common cache line size for a processor system) and two times the number format width times the maximum number of points that can be loaded each cycle. The maximum number of points that can be loaded each cycle is either the decomposition radix (if the load manager is handling the permutation) or the total number of SRAM banks belonging to an operation controller tablet (if the data is already permuted in memory). It is assumed the external memory port has at least 4 tag bits available for outgoing requests. The external memory port is contracted to operate on a ready-valid handshake for requests and responses (although the accelerator will always be ready for responses if it has pending requests).

The external memory port is immediately fed into a simple arbiter modeled after one in Rocket Chip for arbitrating multiple clients to a single memory port. This arbiter splits use of the port between the load manager and store manager with the former getting priority. The request side prepends one bit to the given tag (so the arbiter can later route the response to the appropriate side).

2.5 Load Manager

The load manager, diagrammed in Figure 2.3, is responsible for taking a starting memory address and execution controller tablet ID then filling that tablet with all the data needed to do an FFT. This component uses a four state machine and consists of two parts: the request generator and response handler. In the idle state, the controller waits and is ready for a load command from any an execution controller. Upon receiving any valid load command, the execution controller sends a enters into the (busy) issuing state.

In the issuing state, the request generator sequentially asks for data in memory started at the provided start address. It always asks for as many words as possible as higher level

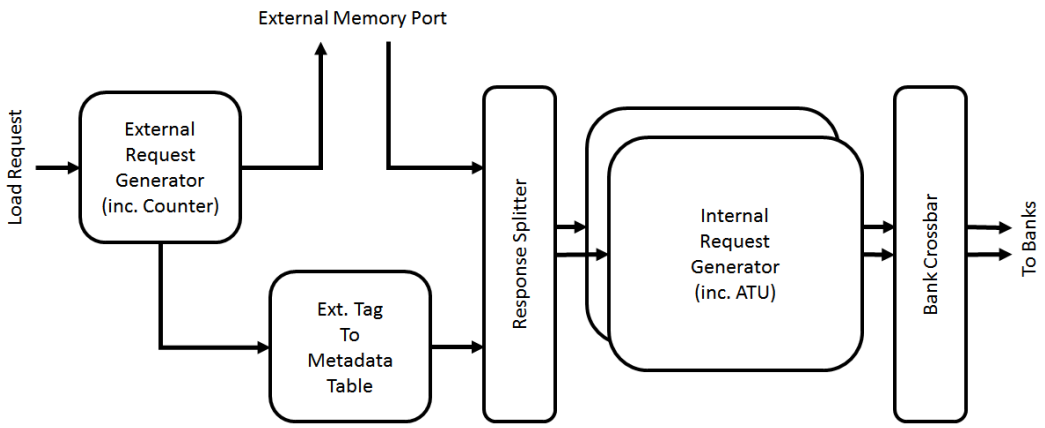

Figure 2.3: Internal architecture of the load manager.

design decisions have already restricted this size to the maximum before there are potential bank conflicts and increments the address counter as appropriate. Upon issuing a request, the sequential number of the request (reset every load operation) is placed in an unused spot of an 8-element 'tag' array with the row id serving as the request tag. This ensures the load manager only ever needs 3-bits of tag. A request is only issued in a cycle if both the outgoing request queue has space and there is an empty slot in the tag array. Once all requests have been issued, the state machine enters an external draining state where it waits for all external responses to have been received.

At all times, the response handler side of the load manager is looking for received responses and preparing to funnel the received data into the memory banks for the tablet. The tag for the response is used to read the tag array for the request number which is then translated to determine which data points represent each section of response data. Here, the response data is split into each actual received data point. In parallel, for each received data point, the actual data is 'unpacked' into a form that can be used immediately by the execution units for computation (which is only relevant for the floating point numbers) and also the position of the data is translated by the WriteATU ROM to determine the target bank and memory address. All the translated data is then fed through a crossbar that, since the banks never conflict, merely reorders the translated data into target bank order. That data is then sent out of the load manager to the load port of destination tablet. Finally, the tag is marked as empty (so it can be re-used for other requests).

Once all of the external responses have been received, as determined by checking to see if all requests are issued and the tag array is empty, the state moves to an internal drain mode. This merely waits a few cycles to ensure all received data has been translated and sent to the tablet memory before progressing. Finally, the execution controller is notified that loading is complete and the load manager goes back to the idle state.

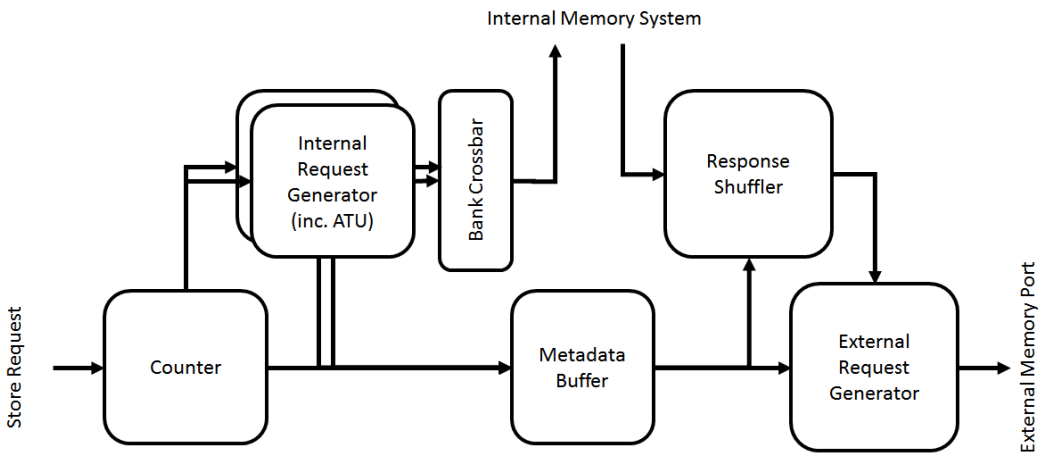

Figure 2.4: Internal architecture of the store manager.

2.6 Store Manager

The store manager, diagrammed in Figure 2.4, is responsible for taking a starting memory address and execution tablet ID then posting all the data in that tablet to the external memory system. This component uses a three state machine and, unlike the load manager, uses a much simpler external response handler. In the idle state, the controller waits and is ready for a store command from any an execution controller. Upon receiving any valid store command, the execution controller sends a enters into the (busy) polling state.

In the polling state, external memory requests are generated from the results of generated internal memory requests. First, an counter tracks the next internal data point to retrieve. This counter is incremented each time by the amount of data points retrieved per request. For each internal data point to be retrieved, a read ATU is used to determine in which bank and address that data point exists. Then, a crossbar is used to route the internal requests to the appropriate banks (again with higher level guarantees there is never a conflict). While the data is being looked up from the internal tablet banks, the metadata of what bank to expect data point results from is also being kept inside of a local queue inside the store manager. Once the internal response comes back from all banks, the metadata is used to feed another crossbar to re-sort the data from each tablet bank into actual FFT output data order. Now, all of the data is available to form the external memory write request.

Once all memory write requests have been sent, the store manager enters into a draining state where it waits for all writes to have been performed (via external ack responses). This ensures the accelerator as a whole reports busy when any external memory requests are pending, as is required by certain memory coherency agreements with the processor. Finally, once all external responses have been received, the store manager goes back into the idle state.

2.7 Instruction ROMs

The instruction ROMs are populated from the generated FFT schedule. Each ROM entry stores the operation read addresses, write addresses, and twiddle factor addresses. Note that read and write addresses also included the banks and recall that a radix r decomposition requires r read addresses and r write addresses per operation. The ROMs were implemented by creating a hardware vector of constant values and then doing a variable index extraction via a large multiplexer. Since all multiplexer inputs are constant, the hardware design tools such as Design Compiler were relied upon to simplify the ROMs into more concise boolean expressions. The vector was actually two-dimensional with the first address stipulating the stage and the second address stipulating the operation within that stage; however, the entire lookup is completed in one cycle. The hardware tools are relied upon to do all optimization. While the hardware tools often spent a lot of time compiling these modules, their area usually ended up being fairly insignificant. Also, while preparations were made to add retiming registers and logic, this ended up being completely unnecessary as the ROMs never appeared on the critical path.

The ROMs composing the read and write address translation units were similarly designed and also very small.

Chapter 3

Scheduling

This chapter will outline the strategy for scheduling FFTs whose size consists of a single factor, the decomposition radix, raised to some power. By far, the most common decomposition radix encountered in literature is 2. Thus, for simplicity and clarity, analysis and figures in this section will mostly consider the radix-2 decomposition. Note; however, that radix-4 decomposition is far more preferable since it is more performant although the radix-4 decomposition is nearly identical except for a few numerical adjustments.

An incomplete schedule for an 8-point FFT using a decomposition radix of 2 is shown again in Figure 3.1. Each column of points indicates data residing in memory somewhere at a specific stage in chronologically ascending order. Thus, the leftmost column is the input time data and the rightmost column is the output frequency data. Each row corresponds

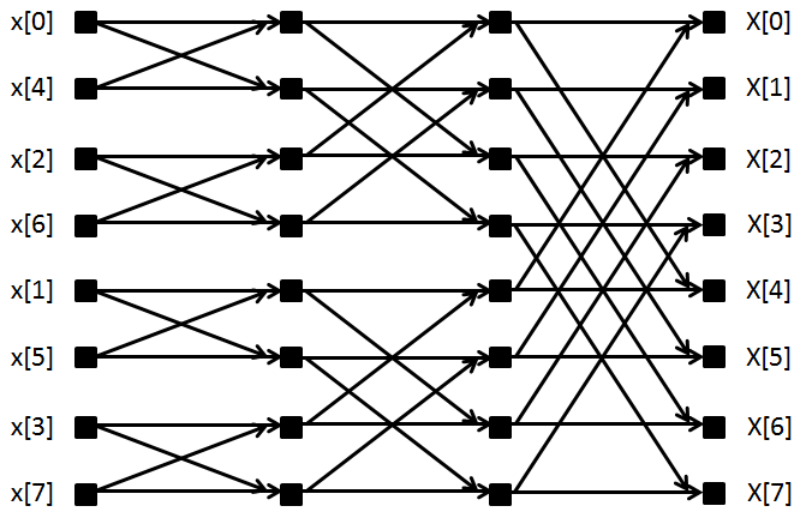

Figure 3.1: 8-point FFT decomposed in time using Cooley-Tukey with decomposition radix of 2. Input data is on the leftmost column and output data on the rightmost.

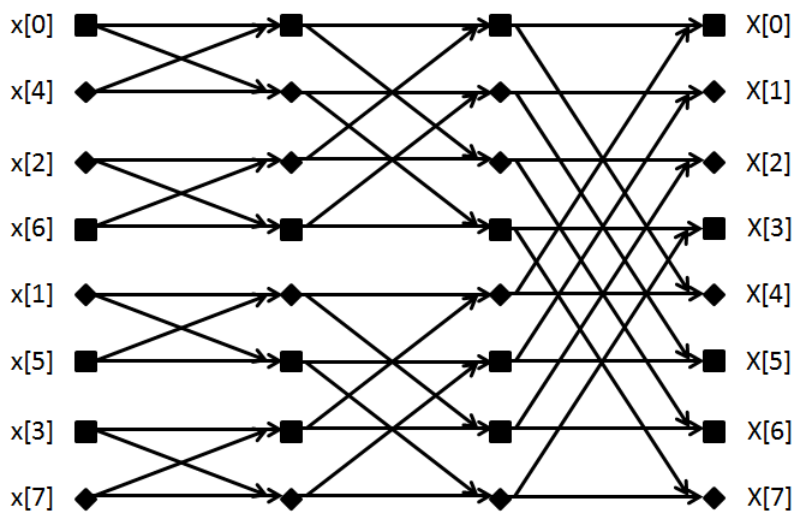

Figure 3.2: 8-point FFT scheduled to map onto one radix-2 butterfly execution unit. Square and rhombus indicate different banks and thus the ultimate result of the banking scheme.

to a different memory address. Note the permutation of the input time data compared to the ordering of the output frequency data. Thus, if each row corresponds to an ascending memory address, then the output data will come out in ascending order but the input data must be stored in 0, 4, 2, 6, 1, 5, 3, 7. If decimation-in-frequency was used, the input data would be stored in natural order but the output data would be so permuted. In general, the permutation can be generated by writing the position in base radix then reversing the order of the digits.

For this graph, all data points must be assigned to memory locations, which include both bank number and address, and all butterfly operations must be assigned to execution cores. However, first, certain constraints are added to the schedule in order to match the hardware architecture. Issuing a new butterfly operation each cycle requires each core to perform two reads of old data to be processed and two writes of new data to be stored. With only dual ported SRAMs (1 read, 1 write port) available, two banks per core are required and data in memory must be arranged such that, for each operation in each pass, the data is appropriately split between the banks. In general, each execution unit will thus require decomposition radix numbers of banks and that incoming and outgoing data for each operation be split between the banks.

Different cores must not incur a bank conflict by trying to write to the same bank on the same cycle. To avoid these conflicts, during each FFT stage, each core is given exclusive write privileges to two banks. Additionally, by mandating that a specific execution unit only ever uses data from two banks, bank conflicts during reads are avoided and no crossbar is necessary between bank read response ports and the execution units. This idea is later represented in the architecture diagram by including the memory banks inside the FFT core

with the execution units rather than having a separate memory system. Further, the crossbar between cores can be simplified if at least one result from the core is required to 'stay' in the core and therefore be used in subsequent passes. Finally, Read-After-Write hazards where data is overwritten before being processed by the current pass must be avoided.

Analysis of the graph provides certain hints on simplifying scheduler operation. The general strategy is to first devise a schedule assuming only one execution unit is being scheduled and then modify the schedule to handle multiple execution units. For the single unit case, consider a each row of the graph given ascending numbers written in base radix-2 (i.e. binary). For a single row, note how each row only intermingles with rows that differ in one digit in their radix-2 decomposition. Thus, a bank selected for a row and the bank selected for any other row with one digit changed must be different and any bank assignment that satisfies this property for all rows will be sufficient to avoid conflicts. The scheme of assigning banks to each row by summing all digits on the radix-2 representation of a row mod 2 can be proven to satisfy this property for all rows. Take any two rows that are potentially in conflict. As shown before, these two rows must differ in their radix-2 representation at only 1 position. It is obvious that the sum of each representation for the first row mod 2 must be different from the sum of the representation for the other row mod 2 must be different. Note that all the digits but one on each side are the same and thus may be subtracted away leaving only one, different digit on each side. Since the modulus taken is equal to the decomposition radix, this implies the original bank assignment expressions formed incongruent modular expressions and thus there is no bank conflict. This technique is shown in Figure 3.2

This approach can be immediately generalized to other radices by instead performing the exact same procedure with each row written in radix-[decomposition radix] form since the requirement that each row only interacts with rows differing by only one digit is guaranteed by the Cooley-Tukey decomposition algorithm. A similar version of this mechanism, generalized for mixed radix, was independently discovered in an earlier paper [6] although the paper does not appropriately demonstrate the simplifications in both algorithm and explanation available in the single-radix case. The paper also does not discuss how to handle scheduling multiple execution units.

The schedule must now be modified to use multiple execution units. The trick here is to merely cyclically schedule all operations top to bottom in the graph across all execution units in time order. This algorithm causes the entire graph to have been scheduled as if it was effectively being decomposed by a larger radix across multiple stages in time. For example, two stages of two execution units during a radix-2 decomposition and cyclically scheduling them mimics the behavior of one stage of one radix-4 execution unit. The case where the effective radix does not divide the FFT size (e.g. 8-point FFT versus effective radix-4) presents no problem as one can consider the smaller FFT graph as a subgraph of a larger FFT schedulable with that radix and cut off as needed. This scheme naturally handles the aforementioned banking issues. The bank used by the data point is reassigned to the corresponding bank for the execution unit that will intake the data point.

Note this scheme requires that the number of execution units cleanly divide the number of operations per stage. However, absence of this property using an alternative scheduling

scheme would result in stalls for some execution units during stages (since the next stage will not start until the last is complete) and thus should be avoided regardless as equal time performance could be achieved by using a smaller number of execution units.

After bank assignments and butterfly executions have been scheduled, the only remaining item is to assign memory addresses to data points. This assignment is done 'inductively' in two phases. First, all input data points are assigned addresses to banks in ascending order. Then, intermediate data points are assigned by constructing and consuming a 'free list' at each time step. At any given time, all execution units have consumed enough incoming data to house the outgoing results. Further, these consumed results are already appropriately spread across the banks. Prior assignments of operations to execution units have already ensured the outgoing results also may be distributed across multiple banks. In fact, a careful examination of the schedule indicates that the cyclical scheduling causes most data to be interchanged between execution units in the earlier stages with the later stages keeping data inside the same execution unit. Output memory locations and banks are simply copied from the second to last stage.

The cyclical assignment of operations and banks as such also provides another convenient property for simplifying hardware: consecutive data points on both the input and output sides are housed in different banks. This banking assignment drastically simplifies constructing a load and store manager that can scale up to use more IO bandwidth in order to move data in or out of each bank every cycle. This property does not hold true in the case where the load manager is automatically permuting the data from natural order into 'bit-reversed.' In that case, the maximum amount of input data points must be limited to the radix size since the bank assignment scheme does guarantee that even post-permutation the items will be located in different banks. Otherwise, explicit permutation hardware must be added to shuffle the data from a banks appropriate for maximizing input to the banks selected for execution. To simplify matters, it was decided to select the former option although this will limit performance in certain configurations. In practice, the execution stage still takes longer for most designs, particularly larger FFT sizes due to O(n*log(n)) execution cycles versus O(n) load cycles and thus the performance drop does not affect throughput for pipelined architectures as shown in the results section.

A scheduler designed specifically to handle these constraints was written in Scala with the code provided in the appendix. The scheduler produces three items: mapping from input data point to memory bank and address, mapping from output data point to memory bank and address, and execution unit info. Execution unit information comprises, for each execution unit, a list of instruction info for every step of every stage of FFT operation. FFT instruction info consists of input data address and bank, output data address and bank, and the twiddle factors. The twiddle factor data is compressed by storing an address to a row in a twiddle factor ROM rather than the actual literal data in the instruction.

Although the algorithm is known to be sound mathematically, verification checks were added as a sanity check. In practice, these verification steps were a hold over to when the scheduling algorithm used was far more primitive. Since older versions used iterative methods rather than a deterministic algorithm, the resultant schedule had less proven properties and

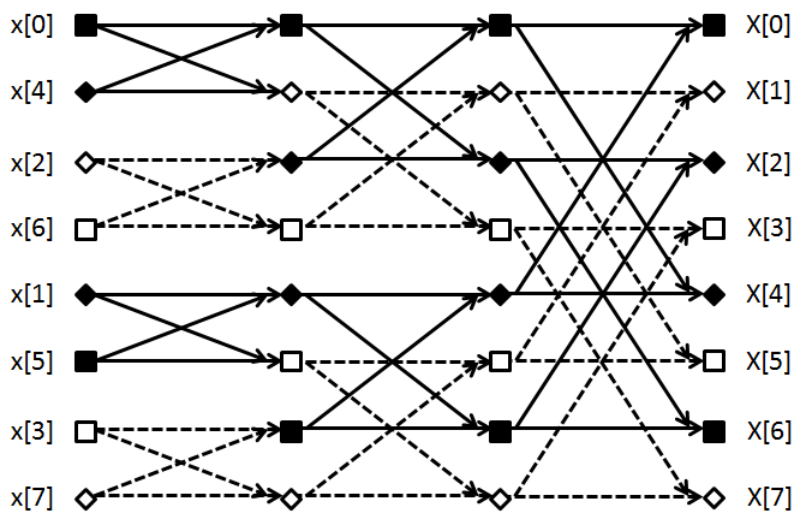

Figure 3.3: 8-point FFT scheduled to map onto two radix-2 butterfly execution units. Icon shape represents memory bank of residence. Butterfly unit executing the solid-line operations only ever reads from the banks with black-filled symbols. Conversely, the unit executing dotted-line operations only ever reads from banks with white-filled symbols.

was potentially more unreliable. First, the verification checks ensure that all operations avoid local memory hazards and all data is used in each stage. Second, it checks that each execution unit only ever reads from a disjoint subset of memories and thus each bank read at most once per cycle. Next, it checks that each bank is written to at most once per cycle and outgoing data is never placed in the location of not-yet-used data. Finally, it checks that input and output data is appropriately banked so multiple data points can be loaded and stored in parallel. Appropriately banked is defined such that the first set of number-of-banks data is all in separate banks as is the next set and so on.

A fully scheduled 8-point FFT for an accelerator with two cores is shown in Figure 3.3.

Chapter 4

Evaluation

4.1 Integrated Testing

The testing infrastructure for verifying the generated FFT accelerator works properly under the given parameter set consists of a Scala and C++ testbench, Verilog test vector generator, Verilog testbench, and comparator scripts. The Scala/C++ testbench extends the Scala tester prebuilt into Chisel. In short, the basic tester requests the generator emit the hardware graph as a C++ simulator, spawns off a separate process for the simulator, and then allows a user-programmed tester to read and write module I/O and clock the design. Extensions to the Tester written for the project automatically convert the Chisel-based ready/valid decoupled interfaces into Scala queues for easier use by other parts of the Scala testbench.

Two sets of tests are available to be run. The first test runs a single FFT and is useful in measuring accelerator latency and energy consumption. The other test runs 8 FFTs in immediate succession and measures system throughput. Output data is not tested for precise bit-accuracy but instead compared against the expected output data computed on the host in double precision floating point and checked to verify root-mean-square errors are within expected limits. This averts issues with non-associativity and non-reproducability of floating point operations across both algorithm implementations and processor architectures.

The Verilog test vector generator outputs both a file of input vectors, with all data already as raw bits, and a file of expected output data. The input data is fed to a manually written Verilog testbench and then Python scripts compare the result to the expected output data. The Verilog testbench currently only supports the single FFT test.

4.2 Results

The FFT generator was evaluated with a variety of parameterizations. The two number formats used were binary32 IEEE 754-2008 floating point and 32 bit fixed point. Physical design was carried out in the TSMC 45 nm technology using Design Compiler, IC Compiler, and Cacti for estimating SRAM performance. Since the Rocket Chip system, which the

Table 4.1: Fixed Point Designs

Design Point	Latency (ns)	Throughput (MS/s)	Area (mm^2)	Energy (nJ)
1024-pt, rdx 2, 1 core	6218	165	0.095	345
1024-pt, rdx 2, 1 core, HT	6218	198	0.165	441
1024-pt, rdx 4, 1 core	1846	556	0.174	258
1024-pt, rdx 4, 1 core, HT	1846	781	0.281	299
1024-pt, rdx 4, 2 cores	950	1081	0.254	265
1024-pt, rdx 4, 2 cores, HT	950	1526	0.384	306
256-pt, rdx 4, 1 core	432	597	0.108	54
256-pt, rdx 4, 1 core, HT	432	911	0.151	60

Table 4.2: Floating Point Designs

Design Point	Latency (ns)	Throughput (MS/s)	Area (mm^2)	Energy (nJ)
1024-pt, rdx 2, 1 core	6238	164	0.126	573
1024-pt, rdx 2, 1 core, HT	6238	197	0.208	692
1024-pt, rdx 4, 1 core	1861	551	0.234	450
1024-pt, rdx 4, 1 core, HT	1861	772	0.335	488
1024-pt, rdx 4, 2 cores	965	1064	0.405	446
1024-pt, rdx 4, 2 cores, HT	965	1493	0.552	498
256-pt, rdx 4, 1 core	444	580	0.181	95
256-pt, rdx 4, 1 core, HT	444	874	0.232	99

accelerator was designed to plug into, requires 1 ns clock period, the clock period for all points was set to be 1 ns. All reported design points were within expected error bounds for both the single FFT and serial FFT tests. Throughput was conservatively calculated as the FFT size divided by the mean time to complete an FFT during the serial FFT tests. Tables 4.1 and 4.2 show the performance of select fixed point FFT accelerators and floating point FFT accelerators, respectively.

Figure 4.1 plots the relationship between latency and area for all generated 1024 point FFTs. The data is clearly separated into multiple tiers of latency performance. The slowest tier is when radix-2 decomposition is done, which does allow the use of much smaller execution units and thus has some potential use cases for fitting the accelerator into a very small die area. The middle tier is simple when radix-4 decomposition is done but still only one core is used. This speeds up both execution as a 4-point DFT can be computed much faster than 2 stages of 2-point DFTs particularly since the latter requires extra internal loads and stores. Also, interfacing with external memory is faster as the radix-4 decomposition uses 4 memory banks and thus twice as much data can be loaded and stored from the external memory system per cycle. Finally, the fastest tier uses both. Execution can obviously be about twice as fast and, due to finer banking, loading and storing from the external memory system is also faster. Segregation in a tier is due to the high-throughput feature which adds

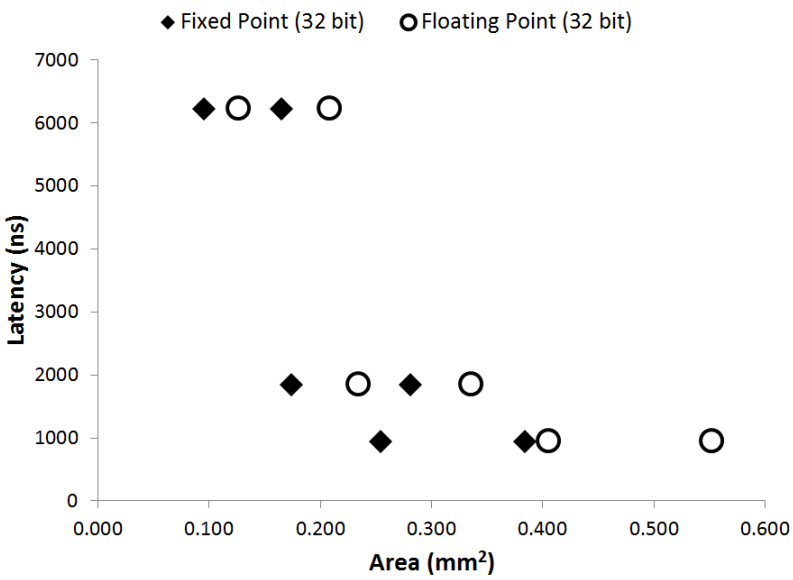

Figure 4.1: All 1024-pt FFT in TSMC 45nm. There is a floor latency of 2048 ns due to time needed to load and unload scratchpad memories.

extra memory so multiple FFTs can be pushed through the system at a time. However, that only affects FFT throughput performance and should not affect single FFT performance. Finally, even though the accelerators using floating point require more retiming stages in the execution units compared to those using fixed point, the figure demonstrates these extra stages do not significantly impact latency performance.

Figure 4.2 demonstrates the effectiveness of the high-throughput multithreading mode in actually delivering improved throughput performance. The results tables confirm that moving from a simple single core design to a single core with high-throughput multithreading boosts performance by over *2x* compared to just adding a second simple core. Also, in general, unlike for latency, as area of the accelerator increases the throughput also increases. Finally, the extra area required to implement the floating point execution units compared to the fixed point execution units is evident by the subtle segregation in this graph. This trades off nicely with the floating point FFT accelerators delivering an order of magnitude or so less error than similar fixed point FFT accelerators.

Figure 4.3 shows that the various parameters of the FFT accelerator do not have a significant influence on the needed energy. Energy lost due to leakage is minimal as leakage power was less than 5% total power in all designs. Also, unsurprisingly, this graph confirms that floating point arithmetic requires more energy than similarly sized fixed point arithmetic.

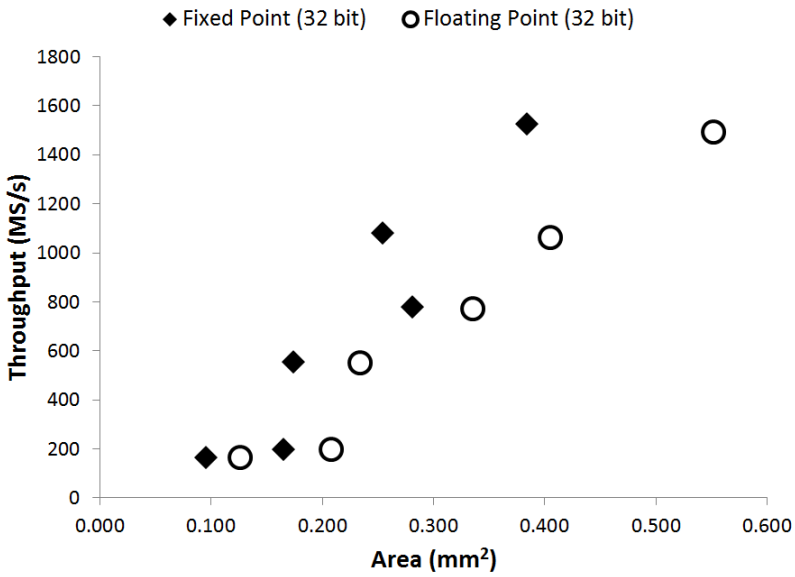

Figure 4.2: All 1024-pt FFT in TSMC 45nm. Here, segregation caused mostly by the high-throughput flag.

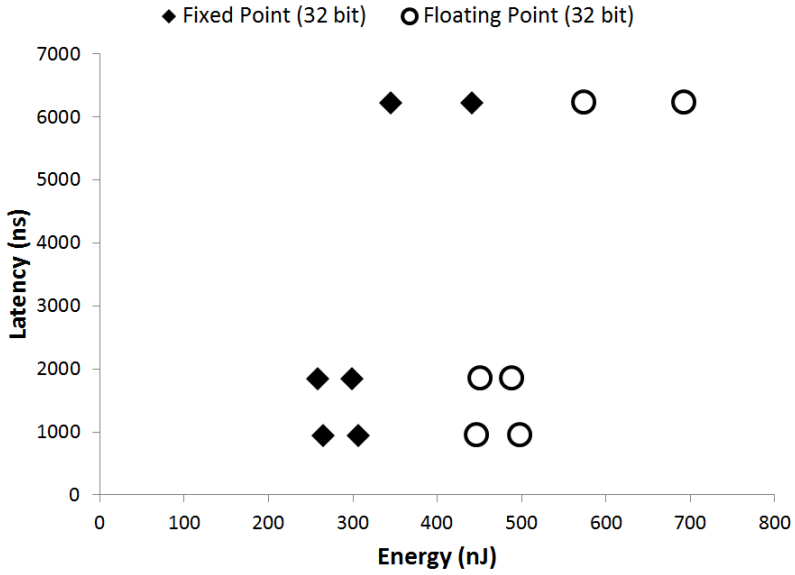

Figure 4.3: All 1024-pt FFT in TSMC 45nm. Energy is per single FFT and includes energy required to load and unload scratchpad memories and perform the computation. Lowest energy points use only 1 core and are not high-throughput.

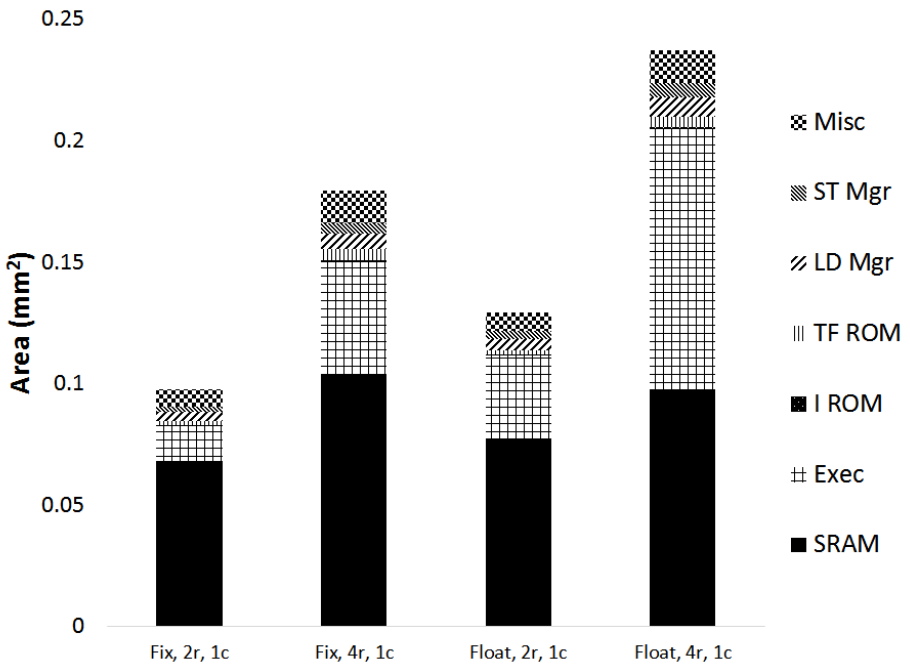

Figure 4.4: 1024 pt Area breakdown in TSMC 45nm

4.3 Energy and Area Breakdown

The breakdown by component of the area taken and energy consumed was influential in determining where optimization efforts were needed. Figure 4.4 and Figure 4.5 show the area and energy breakdowns for various 1024-pt FFT implementations. Execution units in the figures refer only to the raw arithmetic hardware used to compute the 2-point or 4-point DFT. The rest of the pipeline is included in the Misc area or Core Other energy portions.

As expected, energy is dominated by the actual arithmetic required to perform the FFT. Similarly, a large portion of the area is reserved for memories to store data and intermediate results. Amusingly, the raw memory and execution components represented by these sections were actually the most straightforward and simplest to write. Most of the time was spent on the myriad of other supporting components and pipelining infrastructure required to actually get data in and out of those components. This motivates future work where high-level synthesis could drastically simplify designing effort. Since proportionally so little area and energy is ultimately expended by those sections, slightly inefficient designs for those components produced by an HLS engine would be fine as long as the critical components are properly designed.

Comparing parameterizations, it is clear floating point arithmetic costs a premium over fixed point arithmetic in both area and energy. While the reason the execution units are larger is fairly straightforward, the enlarged memories are due to how the Hardfloat library actually recodes 32-bit single precision numbers into an easier to use 33-bit internal format.

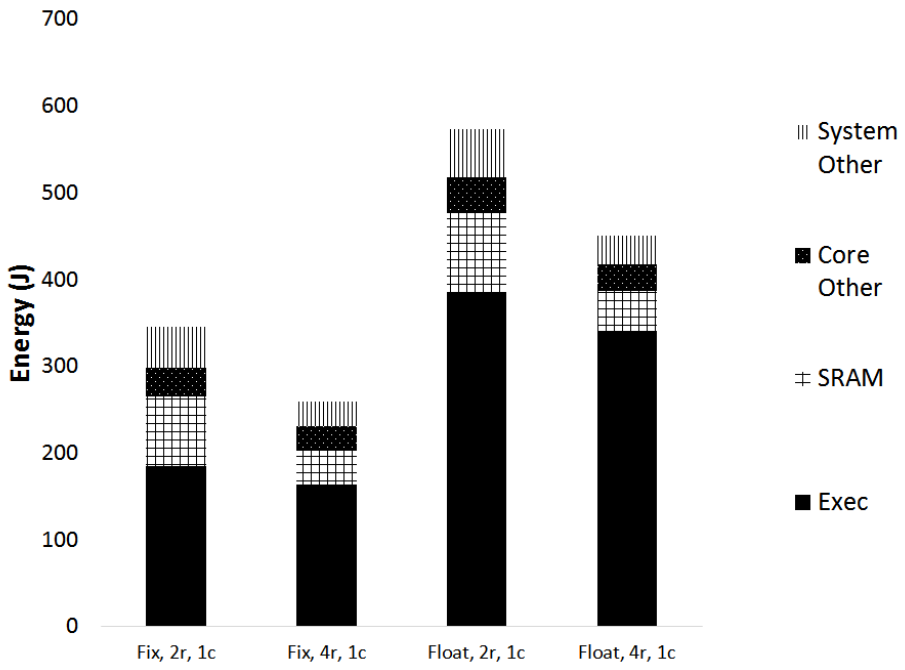

Figure 4.5: 1024 pt Energy breakdown in TSMC 45nm. Core other includes ROMs, local memory arbiters, and other logic for the pipeline. System other includes the load, execution, and store managers.

Unsurprisingly, the energy consumption for the radix-4 decomposition is almost half the consumption for the radix-2 decomposition. This is consistent with the fact that the radix-4 decomposition has the half the number of stages and thus half as many internal loads and stores. There was an initial concern that parallelizing the load and store managers as needed to maximize external memory bandwidth use would balloon the size of the load and store managers. However, they still take up a very small amount of area and have very negligible energy. Thus, high throughput gains were achievable at very minimal cost.

The balanced sizing of the SRAM and execution unit sizing in the radix-4 decomposition, particularly in floating point, was a fairly pleasing result as, coincidentally, in those parameterizations the execution time is only slightly larger than the combined load and store times. Note that the high-throughput mode, where one FFT is being loaded or stored while another executed, will require twice the memory.

Both the instruction and twiddle factor ROMs were very small in area and extremely negligible in energy consumption. The twiddle factor ROMs are larger than the instruction ROMs and do grow when either the more execution units are added or the decomposition radix increases because more read ports are required for the same amount of data. In contrast, total instruction ROM area tends to stay the same for a specific FFT size because, when the number of ports per instruction ROM increases, the amount of data stored per

ROM tends to go down proportionally. Fortunately, it is already known how to use CORDICs to compress twiddle factor ROM sizes even further. Replacing the instruction ROMs with generators would likely only be cost-effective for accelerators that must do varying sizes of mixed-radix FFTs. Further, replacement with generators would likely require the intermediate data points to keep consistent memory addresses and bankings across stages when using multiple execution units. Thus, some new hardware would be required to allow execution units to read from all the banks rather than a subset.

4.4 Comparison to CMU Spiral

The performance of accelerators from the generator in this thesis were compared to the accelerators generated by the Carnegie Mellon University SPIRAL project. For the 1024-point FFT with radix-4 decomposition, 4-word streaming width, natural order on both inputs and outputs, and floating point data, the Spiral accelerator can perform one FFT per 1291 cycles with a latency of 1533. In contrast, for the high-throughput parameterization that equalizes the amount of compute units, this thesis accelerator can perform one FFT per 1326 cycles with a latency of 1861 cycles. However, this thesis accelerator requires half the amount of SRAM compared to the Spiral accelerator. Thus, the area is expected to be 33% smaller. It is further expected that the energy be roughly similar as over half the energy for the FFT is spent by raw compute, which must be the same for both accelerators. Also, the generator in this thesis also supports the low-throughput mode, which requires half the amount of memory, thus allowing the accelerator to shrink by another 25%, at the expense of some throughput performance. In fairness the Spiral architecture can support streaming widths on the input and output sides that are larger than the decomposition radix when still maintaining natural input ordering, as they have a built-in permutation unit. Also, when the input is assumed to be pre-permuted, the Spiral architecture also requires half the memory and thus the same amount as this thesis architecture.

It is expected that this thesis accelerator will fare favorably with all other accelerators performing the Cooley-Tukey decomposition as it achieves near optimal performance. Since the execution units can be made to be running continuously for high-throughput parameterized accelerators, the throughput is at near maximum potential possible. Similarly, the area taken and energy consumed is dominated by the execution units and memory required to store the operation data, both of which any other accelerator will also require. Also, since the FFT is done in-place, no memory is wasted and, in low-throughput mode, the accelerator can be forced to used the theoretical minimum memory size. The only metric susceptible to improvements is latency by trying to start the execution and final store stages before the prior stages had already completed.

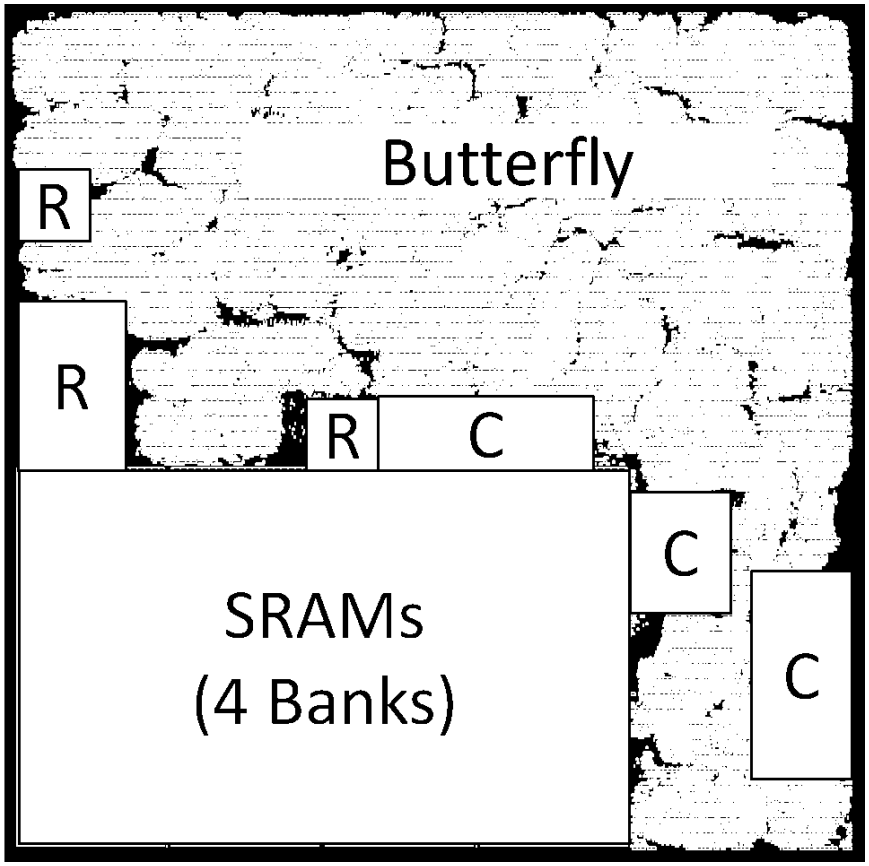

Figure 4.6: Post-PAR chip plot of 1024 single precision floating point FFT hardware accelerator using 1 core and radix-4 decomposition in TSMC 45nm. An R represents the locations of ROM data. A C represents the location of control components such as the operation controllers or managers.

Chapter 5

Reflections on Chisel

5.1 Overview

Chisel is a hardware construction language developed as a Scala domain specific language (DSL) by defining a set of Scala classes [1]. It enables users to construct hardware by manipulating and connecting Scala objects to create a graph representation of the desired hardware. The Chisel backend then analyzes the resultant graph to emit a fast, cycle-accurate C++ simulator, which can communicate immediately with a Scala testbench, or synthesizable Verilog for processing by classic FPGA or ASIC physical design workflows. Chisel allows the hardware generator to do arbitrary calculations in Scala on input parameters while elaborating the hardware graph. Additionally, developers can use functional programming features of Scala to describe hardware abstractions (e.g. implementing an FIR filter with map and reduce functions) and Java standard libraries (e.g. converting floating point numbers to and from raw bit representations for use in testbenches) as ways of simplifying overall hardware development and verification [10].

Other languages, such as SystemVerilog, do offer some of the features of Chisel such as typed wire bundles and expressive generate constructs. However, as these languages are focused on hardware description, separating generator sections into those that compute only on parameters and those that describe desired hardware is more challenging. In particular, the implementation of the constraint solver does not translate particularly well in the other languages. This solver, written completely in pure Scala, consumes an accelerator specification (FFT size, radix, and number of execution units) and produces a list of instructions and memory mappings statically guaranteed not to exhibit memory hazards or bank conflicts. The Chisel components of the generator then use this listing to write instruction and IO address translation ROMs. Implementation in other languages would require maintaining the constraint solver outside of the generator and communicating via text files; however, this creates challenges if the interfaces become desynchronized as generator features are added. Moreover, Chisel guarantees all described graphs can be analyzed into synthesizable Verilog whereas not all SystemVerilog features are guaranteed to be synthesizable.

5.2 Advantages and Disadvantages

While extensive work was done on Chisel internals during the work of this thesis, the focus of this section is to discuss specific *user-visible* advantages and disadvantages encountered when using Chisel to develop this accelerator. Issues discovered as part of non-accelerator related internal work are not considered except when later found to bleed into userspace. Also, unintentional bugs in general will not be mentioned as the expectation is that in time those are fixed and averted for future users. Also, as this thesis project was completed over multiple years, too many bugs have both come and gone to include them in the retrospective timeline with usage examples.

The power of Chisel is already expressed by the accelerator architecture described in the previous sections. In particular, the use of the ButterflyExecutor abstract class to allow seamlessly swapping number formats used in the execution demonstrates how access to more advanced, higher level programming language features like a type system and objects allows for more concise and easier to express hardware modules.

Debugging broken hardware designs with Chisel can, at times be somewhat challenging. Chisel often swings too far on the side of encouraging conciseness at the risk of safety. Thus, the user can often, through typos or misunderstanding, fail to describe a proper circuit yet receive no errors. Also, many error messages returned by Chisel are extremely cryptic, merely saying "assertion failed" and abandoning the entire design compilation. For example, declaring a literal instead of a port inside an IO declaration is an error caught by Chisel. However, the error message is somewhat unhelpful. This particular situation is exacerbated by the fact that creating a literal and defining a module port use the same function identifier but different argument counts. Errors that occur during backend passes after the design has been fully elaborated lead to particularly challenging to debug design situations as often the source code line pointed to, if one is targeted at all, is unrelated to the actual error. For example, in the backend passes, Chisel will detect whether a combinational loop exists in the circuit; however, the reporting on what nets compose the combinational loop is often useless and, in certain versions, broken. Fortunately, the Chisel testing infrastructure does help substantially in writing unit-level tests for modules and catching some design errors. Although, there is currently no support in Chisel for any form of test coverage analysis.

The integration of Chisel with Scala often leads to either confusing user experiences or straight-out errors. The use of mutable global state in the Chisel backend makes the use of lazy val, var, and certain def constructs particularly dangerous. The use of the former two can lead to painful errors where logic for one module is mysteriously emitted in a different module or direct assignments are mistakenly transformed into conditional assignments. The implementation how module IOs are defined can lead to nasty *Java* errors when modules are subtyping. This occurs specifically when both the parent module creates logic using IOs and then subtype module attempts to override the IO to add new ports in the subtype. No parts of the Chisel documentation explain either when a 'clone' method must be defined or what contract such method must fulfill. Chisel sources or examples must be consulted to divine the appropriate contract. As a pedantic aside, the type signature that Chisel requires

for the method is such that any method that fulfills the contract must use a Scala typecast and thus such method must violate type safety guarantees in Scala.

Extending Chisel to include user-defined data types is a double-edged sword. This accelerator creates and uses a complex data type as a subtype of Bundle containing both real and imagionary data. Operators for complex addition and multiplication were even added to this data type for use in earlier versions of the accelerator. Although, later versions eschewed their use in favor of using the ButterflyExecutor interface to allow swapping number representation and performing certain optimizations across operations. However, there is no mechanism in Chisel for succinctly defining literal Bundles and thus ROMs of Bundles. Finally, more interesting data types, such as for fixed point numbers, proved challenging to implement properly. Initial analysis would indicate subtyping Bits would be sensible; however, no documentation existed for the necessary contract behind all the necessary functions and overrides. Thus, the fixed point numbers had to be implemented as Bundles of one element. The more recent Chisel versions contain a Fixed datatype in the standard library; however, there is no mechanism for controlling overflow behavior or defining literals.

One of my largest, and perhaps most contentious criticism of Chisel is that Chisel is too permissive and often tries to proceed with elaboration instead of erroring earlier. The most major offender on this issue is the Bundle type. Any assignment between any subtype of Bundle to any other subtype of Bundle will yield no error at any stage of compilation. While hailed as a feature, this behavior can lead to painful debugging sessions that often boiled down to discovering another wire was connecting to a whole packet rather than the containing packet. Similar issues arise in other parts of Chisel, e.g. any subtype of Data can be multiplexed with any other subtype of Data; however, the Scala type system often throws an error on consumers of the erroring logic helpfully speeding up debugging. Use of typeclasses via Scala implicits or providing the user an alternative aggregate type to use when better connection checks are desired could avert all of these issues. Related to Bundle assignments, behavior of the normal assignment operator is somewhat surprising (all elements are monodirectionally assigned without error) and undocumented.

Chapter 6

Conclusions

6.1 Future Work

Many components in the design are amendable to both microarchitectural adjustments and major system-level changes in order to improve performance or add new features. The twiddle factor ROMs could be compressed further by implementing them with a CORDIC design. The CORDIC design would require storing a very small table of sine and cosine results that is then used to compute the desired roots of unity on-the-fly via addition and subtraction operations. Accuracy of the result, as the compression is lossy, can be improved by increasing the table depth and computing over more steps. It is anticipated that, due to the size of floating point adder hardware, the fixed point architecture will achieve far more benefit from the CORDIC design.

Similarly, the instruction ROMs could be replaced by instruction generators. In the parameterizations that use only one execution unit, this is much easier as each data point retains the same bank assignment and memory address across stages. Constructing an instruction generator that works across multiple execution units requires an adjustment in scheduling to avoid use of the free list, as is too complicated to track in hardware. Instead, the anticipated solution is two-fold: introduce the restriction where each data point retains a static assignment across stages and, necessarily, allow a single execution unit to read from any bank during any stage rather than an elaboration-time determined subset. This will likely a require a restructuring of the architecture to better separate the execution unit pipeline from the memory and implementation of a more-proper memory system. The scheduling theory is then to treat the small execution units as an combined larger execution unit completing the FFT as if it was decomposed by a larger radix. This is structurally similar to time multiplexing real multipliers and adders inside of the execution units, although it is unclear which approach is simplest overall.

Assuming the prior two ROMs are replaced, the pathway to performing mixed radix FFTs is far clearer. While the scheduling algorithm was designed for the single radix case, the banking conflict proof should be amendable to include the mixed radix casae by considering

each data position in a mixed base representation. The execution units would need to be redesigned to be reconfigurable to multiple radices as dedicating execution units for each radix is likely too wasteful.

The accelerator execution units could also be modified to support the pipelined, streaming architecture as well as the current iterative architecture. This allows for slight memory savings and thus slightly more efficient designs in applications that require constant high-throughput FFT calculations.

Finally and alternatively, if all prior changes are done, the actual architecture of the architecture becomes fairly similar to that of a highly specialized vector machine with hardcoded instructions. Thus, it could be viable to see if this architecture could be hybridized with a vector machine architecture so that FFTs could be done at almost dedicated-architecture levels of performance but retaining use of the large components like execution units and memory for use in general programs.

6.2 Final Thoughts

This thesis demonstrated how hardware generators enable users to rapidly explore the design space of a system for a given application. The generated designs spanned a wide range of various performance metrics in energy, latency, throughput, and area allowing system integrators to find the exact balance desired. Designs were found to be competitive with other leading generators for dedicated hardware FFT accelerators, like Spiral. A scheduling algorithm for performing a Fast Fourier Transform under the Cooley-Tukey decomposition in-place was derived and explained. Various aspects of the schedule were successfully constrained to match desired simplification in the hardware.

The use of Chisel showed how high-level programming languages simplify the design of these hardware generators. The Scala-embedded language allowed the use of layers of abstraction to simplify code, avoid bugs, and improve the expressivity of parameters. This consequentially opens up a wider design space to explore and ultimately improves quality of results. For example, abstract classes allowed parts of the accelerator to reconfigure itself at elaboration-time to support different number formats. Scheduling data could be computed in the fully-featured software language and immediately placed into the instruction and twiddle factor ROMs. And, the Chisel testing infrastructure allowed for quick and relatively painless testing of generated designs.

Bibliography

[1] Jonathan Bachrach, Huy Vo, et al. "Chisel: Constructing Hardware in a Scala Embedded Language". In: *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. June 2012, pp. 1212–1221.

[2] Berkeley Architecture Research. *Rocket Chip Generator*. `https://github.com/ucb-bar/rocket-chip`. [Online; accessed 2-December-2014]. 2014.

[3] Matteo Frigo and Steven G. Johnson. *FFTW Home Page*. `http://www.fftw.org/`. [Online; accessed 4-December-2015]. 2015.

[4] Mark Horowitz's Group. *Genesis Home - Stanford Edition*. `http://www-vlsi.stanford.edu/genesis/`. [Online; accessed 18-December-2015]. 2015.

[5] John Hauser. *Berkeley Hardware Floating-Point Units*. `https://github.com/ucb-bar/berkeley-hardfloat`. [Online; accessed 2-December-2014]. 2014.

[6] Chen-Fong Hsiao, Yuan Chen, and Chen-Yi Lee. "A Generalized Mixed-Radix Algorithm for Memory-Based FFT Processors". In: *IEEE Transactions on Circuits and Systems-II: Express Briefs, Vol. 57, No. 1*. Jan. 2010, pp. 26–30.

[7] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform (Frontiers in Applied Mathematics)*. Philadelphia, Pennsylvania: Society for Industrial and Applied Mathematics, 1992.

[8] Peter Milder. *Spiral Project: DFT/FFT IP Core Generator*. `http://www.spiral.net/hardware/dftgen.html`. [Online; accessed 2-December-2014]. 2014.

[9] Peter A. Milder et al. "A Generalized Mixed-Radix Algorithm for Memory-Based FFT Processors". In: *ACM Transactions on Design Automation of Electronic Systems, Vol. 17, No. 2*. Apr. 2012, 15:1–15:33.

[10] Martin Odersky. *The Scala Programming Language*. `http://www.scala-lang.org/`. [Online; accessed 2-December-2014]. 2014.

Chapter 7

Appendix

7.1 Scheduler Code

```
import fft_mcgen._

object mcgen {
        def main(args: Array[String]): Unit = {
    args.headOption match {
      case Some("one") => {
        require(args.length >= 4, "syntax: run one [size] [rdx] [num_machines
        val size  = args(1).toInt
        val radix = args(2).toInt
        val num_m = args(3).toInt
        test_one(size, radix, num_m)
      }
      case Some("simple") => {
        test_simple_rdx2
      }
      case _ => {
        test_standard
        test_rdx3
      }
    }
        }

  def exp(b: Int, e: Int): Int = BigInt(b).pow(e).toInt

  def test_one(size: Int, radix: Int, num_machines: Int) = {
              (new FFT_graph(size, radix)).schedule(num_machines).verify_sc
```

```
  }

  def test_standard = {
    // Test all valid schedules for 2 to 4096 pt FFTs of radix 2,4,8 decompo
    for(lsize <- 2 to 12; lrdx <- 1 to 3; lnum <- 0 to (lsize-lrdx) if (lsize
      val size  = exp(2, lsize)
      val radix = exp(2, lrdx)
      val num_m = exp(2, lnum)
                test_one(size, radix, num_m)
    }
  }

  def test_simple_rdx2 = {
                val fft2_simple  = test_one(8   , 2,  1)
                val fft2_large_s = test_one(1024, 2,  1)
                val fft2_large_m = test_one(1024, 2, 32)

                val fft4_simple  = test_one(16  , 4,  1)
                val fft4_large_s = test_one(1024, 4,  1)
                val fft4_large_m = test_one(1024, 4,  4)
                val fft4_large_l = test_one(1024, 4,  8)
                val fft4_large_xl= test_one(1024, 4, 16)

                val fft8_simple  = test_one(64  , 8,  1)
                val fft8_large_s = test_one(4096, 8,  1)
                val fft8_large_m = test_one(4096, 8,  4)
                val fft8_large_l = test_one(4096, 8,  8)
  }

  def test_rdx3 = {
                val fft3_simple  = test_one(27  , 3, 1)
                val fft3_large_s = test_one(6561, 3, 1)
                val fft3_large_m = test_one(6561, 3, 3)
          val fft3_large_l = test_one(6561, 3, 9)
  }
}

package fft_mcgen

import scala.collection.mutable.ListBuffer
import scala.collection.mutable.HashMap
import scala.math._
```

```
case class OverconstrainedException extends Exception

class Memory(val id:Int) {
  private var current_size = -1

  def get_fresh_location() = {current_size+=1; current_size}

  override def toString = {"M" + id}
}
class Machine(val id: Int, val memories: IndexedSeq[Memory])
{
  private var current_pc = 0
  def reset_pc() = {current_pc = 0}
  def get_fresh_pc() = {current_pc+=1; current_pc-1}
}

trait Operation {
  def set_owner(x: Machine): Unit
  def set_position(): Unit
  def owner: Option[Machine]
  def position: Option[Int]
  def operands: IndexedSeq[Datapoint]
  def results: IndexedSeq[Datapoint]
  def reset(): Unit
}

class NOP() extends Operation {
  def set_owner(x: Machine) = {}
  def set_position() = {}
  def owner = None
  def position = None
  def operands = Vector.empty
  def results = Vector.empty
  def reset(): Unit = {}
}

class GenericOp(in: IndexedSeq[Datapoint], out: IndexedSeq[Datapoint], val tw
  private var _owner: Option[Machine] = None
  private var _position: Option[Int] = None

  def set_owner(mach: Machine) = {_owner = Option(mach)}
```

```
  def set_position() = {_position = Option(_owner.get.get_fresh_pc())}
  def owner() = {_owner}
  def position() = {_position}
  def operands() = in
  def results() = out
  def reset(): Unit = {_owner = None; _position = None}

  for (operand <- operands) {operand.consumer = this}
  for (result <- results)   {result.producer = this}
}

class Datapoint(val id:Int) {
  var owner: Option[Memory] = None
  var owner_location: Option[Int] = None

  var consumer: Operation = new NOP()
  var producer: Operation = new NOP()

  override def toString() = id.toString
  def memloc() = {
    owner match {
            case Some(mem) => mem.toString + ":" + owner_location.getOrElse(-
            case None => "---"
          }
  }

  def copymemfrom(target: Datapoint) = {
    owner = target.owner
    owner_location = target.owner_location
  }

  def reset(): Unit = {owner = None; owner_location = None}
}

class FFT_graph(fftsize: Int, radix: Int) {
  val stages = (log(fftsize)/log(radix)).toInt
  require(pow(radix,stages)==fftsize,"FFT size must be a power of radix.")

  val opcount = stages * (fftsize >> 1)

  val data = Vector.tabulate(stages+1, fftsize)((_,n) => (new Datapoint(n)))
```

```
val operations: Seq[Seq[GenericOp]] = {
  val operations_buffer = Vector.tabulate(stages)((_) => (new ListBuffer[Ge
  for (stage <- 0 until stages) {
    val cur_fft_size = pow(radix,stage+1).toInt
    for (cur_fft <- 0 until fftsize/cur_fft_size) {
      val table_iterator = fftsize/cur_fft_size
      for(cur_op <- 0 until cur_fft_size/radix) {
        val positions = Vector.tabulate(radix)(n => cur_fft*cur_fft_size +
        val inputs = positions.map(n => data(stage  )(n))
        val outputs = positions.map(n => data(stage+1)(n))
        val twiddle = (1 until radix).toVector.map(_*cur_op*table_iterator)

        operations_buffer(stage).append(new GenericOp(inputs, outputs, twid
      }
    }
  }
  operations_buffer.map(_.toList)
}

def reset_schedule() = {
  operations.foreach(_.foreach(_.reset()))
  data.foreach(_.foreach(_.reset()))
}

def pos2bank(pos: Int): Int = {
  val (bank, _) = (0 until stages).foldLeft((0,pos)){(acc, stage) => {
    val accbank = acc._1; val accpos = acc._2
    ((accbank + accpos) % radix, accpos/radix)
  }}
  bank
}

def schedule(num_machines: Int) = {
  reset_schedule()

  // Currently, tighten activates a weak constraint where writing and readi
  val num_ops = fftsize/radix
  require(((num_ops/num_machines).toInt * num_machines) == num_ops, "# Mach
  val num_memories = num_machines*radix

  val memories = Vector.tabulate(num_memories)((n)=>(new Memory(n)))
  val machines = Vector.tabulate(num_machines)(
```

```
    (n)=>(new Machine(n, memories.zipWithIndex.filter(t=>(radix*n until rad

  // Assign operations to machines cyclically
  for(stage <- 0 until stages) {
    operations(stage).zipWithIndex.foreach({case (op, op_pos) =>
      op.set_owner(machines(op_pos % num_machines))
      op.set_position()
    })
    machines.foreach(_.reset_pc())
  }

  // Datapoints should be stored in the consumer's memory with bank depende
  for(stage <- 0 until stages; pos <- 0 until fftsize) {
    val dp = data(stage)(pos)
    val bank = pos2bank(pos)
    dp.owner = dp.consumer.owner.map(_.memories(bank))
  }

  // Now, need to assign memory locations to places
  // Start by just assigning sequential locations to the input data
  for(pos <- 0 until fftsize) {
    val dp = data(0)(pos)
    dp.owner_location = dp.owner.map(_.get_fresh_location)
  }
  // Now, for each stage, construct a free list at each time consisting of
  //   then, find an appropriate element in the free list. Each memory can
  //   so each memory can have only 1 element on the free list
  for(stage <- 0 until stages) {
    // Construct the free list from operands
    val free_list: Map[Int, Map[Memory, Int]] = {
      val all_free_list = HashMap.empty[Int,HashMap[Memory,Int]]
      for(op <- operations(stage)) {
        val cycle = op.position.get
        val cycle_free_list = all_free_list.getOrElseUpdate(cycle, HashMap.
        for(dp <- op.operands) {
          val mem = dp.owner.get
          assert(!cycle_free_list.isDefinedAt(mem))
          cycle_free_list.update(mem, dp.owner_location.get)
        }
      }
      all_free_list.mapValues(_.toMap).toMap
    }
```

```
      // Use free list to assign results
      for(op <- operations(stage)) {
        val cycle = op.position.get
        val cycle_free_list = free_list(cycle)
        for(dp <- op.results) {
          dp.owner_location = dp.owner.map(mem => cycle_free_list(mem))
        }
      }
    }

    // For output data, copy memory assignments from previous set of input da
    for(pos <- 0 until fftsize) {
      val prev_dp = data(stages-1)(pos)
      val out_dp  = data(stages  )(pos)
      out_dp.owner = prev_dp.owner
      out_dp.owner_location = prev_dp.owner_location
    }
    /*
    println("SCHEDULE")
    for(stage <- 0 until stages+1; pos <- 0 until fftsize) {
      val dp = data(stage)(pos)
      println("%2d %4d %s.%s %3s %3s".format(stage, pos,
        dp.owner.map(_.toString).getOrElse("-"), dp.owner_location.map(_.toSt
        dp.consumer.position.getOrElse("-"), dp.producer.position.getOrElse("
      ))
    }
    */

    this
  }
  def check_memories_unique(dps: Seq[Datapoint]) = {
    for(dp <- dps) {
      val my_mem = dp.owner.get
      val other_mems = dps.filter(_!=dp).map(_.owner.get)

      assert(!other_mems.contains(my_mem))
    }
  }
  /*
  def calc_write_fanout(mach: Machine, stage: Int) = {
    val perop_memories_touched = operations(stage).toArray.map(op => (if(op.o
```

```
    val memories_touched = perop_memories_touched.reduceLeft(_++_).foldLeft(n
    memories_touched.length
  }
*/
  def get_num_machines() = {
    operations(0).map(_.owner.get).distinct.length
  }

  def verify_schedule():FFT_graph = {
    val num_machines = get_num_machines()
    val num_banks = num_machines*radix
    println("VERIFYING: " + fftsize +"-point FFT with radix-" + radix + " blo
    println("Verifying all operations avoid local memory hazards.")
    for(stage <- 0 until stages) {
      for(op <- operations(stage)) {
        check_memories_unique(op.operands)
        check_memories_unique(op.results)
      }
    }
    println("Verifying all data accounted for in each stage")
    (0 until stages).foreach(stage => {
      assert(data(stage).map(_.toString).distinct.length == fftsize)
    })
    println("Verifying machines read only from associated memories")
    for(stage <- 0 until stages) {
      for(op <- operations(stage)) {
        val read_memories = op.operands.map(_.owner.get)
        assert(read_memories.filter(!op.owner.get.memories.contains(_)).lengt
      }
    }
    println("Verifying each bank only written to once per cycle")
    for(stage <- 0 until stages) {
      val concurrent_ops = HashMap.empty[Int,ListBuffer[Operation]]
      for(op <- operations(stage)) {
        concurrent_ops.getOrElseUpdate(op.position.get,ListBuffer.empty[Opera
      }
      for(opset <- concurrent_ops.values) {
        val written_dps: Seq[Datapoint] = opset.toList.map(_.results).foldLef
        val written_banks = written_dps.map(_.owner.get)
        assert(written_banks.distinct.length == written_banks.length)
      }
    }
```

```
    println("Verifying consumption distance is <= 0")
    val consume_dist = (0 until stages).map(stage => {
      data(stage).map(rdp => {
        val wdp = data(stage+1).filter(cdp => (cdp.owner==rdp.owner && cdp.ow
        wdp.producer.position.get - rdp.consumer.position.get
      }).reduce(max(_,_))
    }).reduce(max(_,_))
    println("Consumption distance: %d".format(consume_dist))
    assert(consume_dist <= 0)

    println("Verifying input and output data consecutively banked")
    def consecutive(target: Seq[(Int,Int)]): Boolean = {
      def is_unique[A](in: Seq[A]): Boolean = in.distinct.length == in.length
      target.map(_._1).grouped(num_banks).map(is_unique(_)).fold(true)(_&&_)
    }
    assert(consecutive(make_atu_in))
    assert(consecutive(make_atu_out))

    /*
    println("Verifying input and output data consecutively banked after perm
    def permute(a: Int): Int = {
      (0 until stages).foldLeft((a, 0))((carry, _) => {
        val (oldnum, newnum) = carry
        (oldnum/radix, newnum*radix + oldnum%radix)
      })._2
    }
    val permuted_atu_in = Vector.tabulate(fftsize)(n => make_atu_in(permute(n
    assert(consecutive(permuted_atu_in))
    */

    println("Verified")
    this
  }

  override def toString() = {
    (0 until stages).map(stage => {
      "Stage #%d".format(stage) + "\n" +
      (0 until fftsize).map(entry => {
        val dp = data(stage)(entry)
        "#" + dp + " -> " + dp.memloc
      }).reduce(_+"\n"+_)
```

```
    }).reduce(_+"\n"+_)
  }

  def make_instruction_packages(machine_id: Int): IndexedSeq[IndexedSeq[Instr
  {
    val packages = (0 until stages).toVector.map(stage => {
      operations(stage).filter(_.owner.get.id==machine_id).toVector.sortBy(_.
        def make_full_addr(dp: Datapoint) = (dp.owner.get.id, dp.owner_locati
        InstructionPackage(op.position.get, op.operands.map(make_full_addr(_)
      })
    })
    packages
  }
  def get_machine_memories(machine_id: Int): IndexedSeq[Int] =
  {
    val machine: Machine = operations(0).filter(_.owner.get.id==machine_id)(0
    machine.memories.map(_.id)
  }

  def make_atu(stage: Int): IndexedSeq[(Int, Int)] =
  {
    (0 until fftsize).toVector.map(n => {
      val target = data(stage)(n)
      (target.owner.get.id, target.owner_location.get)
    })
  }
  def make_atu_in  = make_atu(0)
  def make_atu_out = make_atu(stages)

}

case class InstructionPackage(val position: Int, val reads: IndexedSeq[(Int,I
  override def toString = position+": reads="+reads.map(r=>r._1+"."+r._2).red
    "  twiddles=" + twiddle.map(_.toString).reduce(_+","+_) +
    "  writes="+writes.map(w=>w._1+"."+w._2).reduce(_+" "+_)
}
```

7.2 Accelerator Code

```
package FFT

import Chisel._
```

```
import Node._
import rocket._
import uncore._

case class FFTConfig(fft_size: Int, num_machines: Int, radix: Int, data_width
                     private val _num_multipliers: Option[Int] = None, privat
                     xprlen: Int = 64, floating_point: Boolean, num_tablets:
{
  // CHECK to ensure appropriate properties and REPORT
  require(math.pow(radix,(math.log(fft_size)/math.log(radix)).toInt)==fft_size
    "FFT size must be a power of radix.")
  println("Configuring %d-point FFT of %d radix using %d BF execution units".
  println("Number of Tablets: %d".format(num_tablets))
  println("Incoming assumed data order is " + (if(init_permute) "natural" els

  val num_multipliers = math.max(math.min(_num_multipliers.getOrElse((radix-1
  println("Number of Multipliers per Machine: %d".format(num_multipliers))

  val max_int_needed = log2Up(fft_size)+2 // +1 since need sign bit, +1 neede
  val ideal_max_fix  = data_width - max_int_needed
  val fix_pt = math.min(math.max(_fix_pt.getOrElse(ideal_max_fix), 0), data_w

  if(floating_point) {
    require(data_width==32, "Only 32-bit floating point currently supported")
    println("Floating Point Arithmetic (%d wide)".format(data_width))
  } else {
    println("Fixed Point Arithmetic (%d wide, fixed pt at %d)".format(data_wi

    val integer_supplied = data_width-fix_pt
    if(integer_supplied < max_int_needed)
      println("WARNING: Not enough integer bits to represent all possible res
  }

  // SETUP some DATA TYPES and CONVERSIONS
  private val raw_bits_width = if(floating_point) 33 else data_width
  def DataWord() = {UInt(width=raw_bits_width)}
  def DataMath() = {Complex(raw_bits_width)}
  def DataBits() = {UInt(width=DataMath().toBits.getWidth)}

  def unpack_to_math(in: Bits) = {
  // Take data from outside and turn into internal data format
    val payload_construct = DataMath()
```

```
    val real_extract = in(  data_width-1,           0)
    val imag_extract = in(2*data_width-1, data_width)

    if(floating_point) {
      payload_construct.real := hardfloat.floatNToRecodedFloatN(real_extract,
      payload_construct.imag := hardfloat.floatNToRecodedFloatN(imag_extract,
    } else {
      payload_construct.real := real_extract
      payload_construct.imag := imag_extract
    }

    payload_construct
  }

  def unpack(in: Bits): Bits = {
  // Take data from outside and turn into internal data format as simple bits
    unpack_to_math(in).toBits
  }

  def pack(in: Bits): Bits = {
  // Take data from inside as simple bits and turn into outside format
    val out_cplx = DataMath().fromBits(in)
    val extended_real = UInt(width=32);
    val extended_imag = UInt(width=32);

    if(floating_point) {
      extended_real := hardfloat.recodedFloatNToFloatN(out_cplx.real, 23, 9)
      extended_imag := hardfloat.recodedFloatNToFloatN(out_cplx.imag, 23, 9)
    } else {
      extended_real := out_cplx.real
      extended_imag := out_cplx.imag
    }

    Cat(extended_imag, extended_real)
  }

  // Memory System Setup
  val use_flex_mem = true
  def tablet_id_width = log2Up(num_tablets+1)

  val num_dmem  = num_machines*radix
  val dmem_size = fft_size/num_dmem
```

```
  val dmem_addr_width = log2Up(dmem_size)

  val max_points_per_request = if(use_flex_mem) math.min(512/(2*data_width),
  val load_points_per_request  = if(init_permute) radix else max_points_per_r
  val store_points_per_request = max_points_per_request

  // IO LEVEL datatypes
  val flex_width = max_points_per_request*2*data_width
  val flex_addr_width = xprlen

  def WriteReq()      = {Packet(dmem_addr_width)(DataBits())}
  def FullWriteReq() = {Packet(log2Up(num_dmem))(WriteReq())}
  def ReadReq()       = {Bits(width=dmem_addr_width)} // the address
  def FullReadReq()  = {Packet(log2Up(num_dmem))(ReadReq())}

  def FFTWriteReq()  = {Packet(log2Up(fft_size))(DataBits())}
  def FFTReadReq()   = {Bits(width=log2Up(fft_size))} // the address

  def LocalReadPorts  =  Vec.fill(num_dmem){new LocalReadPort(this)}
  def LocalWritePorts = (Vec.fill(num_dmem){Valid(WriteReq())}).asOutput

  // Butterfly, Load/Store Managers configurations
  val butterfly = ButterflyConfig(radix=radix, num_multipliers=num_multiplier
  println(s"Butterfly Latency: ${butterfly.total_latency} (adjust by ${latenc

  val load_drain_time = 4 // 2 from ATU, 1 from LM, and 1 for safety
  val exec_drain_time = 3 + butterfly.total_latency

  // Now handle things related to FFT schedule and ROMs
  def bit_reverse(a: Int) =
  {
      val tjump = log2Up(radix)
      val num_terms = log2Up(fft_size)/tjump
      val tmask = (1 << tjump) - 1
      val terms = (0 until num_terms).reverse.toVector.map(ti => (a & (tmask
      terms.reverse.foldLeft(0)((acc,x) => acc*radix + x)
  }

  val get_stage0_position: Int=>Int =
    if(init_permute) (a: Int)=>{a}
    else bit_reverse
```

```
  // Generate the instruction schedule and verify it
  val schedule = (new fft_mcgen.FFT_graph(fft_size, radix)).schedule(num_mac
  schedule.verify_schedule()

  val write_atu: IndexedSeq[(Int, Int)] =
    if(init_permute)
    {
      val reversed = schedule.make_atu_in
      Vector.tabulate(fft_size)(n => reversed(bit_reverse(n)))
    }
    else {schedule.make_atu_in}
  val read_atu: IndexedSeq[(Int, Int)] =
    schedule.make_atu_out

/*
  def print_by_8(a: Seq[Int], scheme: Int = 0) = {
    val ls_size = num_machines*radix
    val num_loadsets = fft_size/ls_size

    scheme match {
      case 0 => {
        for(ls <- 0 until num_loadsets) {
          for(elem <- 0 until ls_size) {
            val idx = ls*ls_size + elem
            print("%4d".format(a(idx)))
          }
          print("\n")
        }
      }
      case 1 => {
        val num_blocks = num_machines
        val block_size = num_loadsets/num_blocks
        for(block_idx <- 0 until block_size) {
          for(block_n <- 0 until num_blocks) {
            val ls = block_n*block_size + block_idx
            for(elem <- 0 until ls_size) {
              val idx = ls*ls_size + elem
              print("%4d".format(a(idx)))
            }
            print("\n")
          }
          println("--")
```

```
        }
      }
      case 2 => {
        val num_blocks = num_machines*radix
        val block_size = num_loadsets/num_blocks
        for(block_idx <- 0 until block_size) {
          for(block_n <- 0 until num_blocks) {
            val ls = block_n*block_size + block_idx
            for(elem <- 0 until ls_size) {
              val idx = ls*ls_size + elem
              print("%4d".format(a(idx)))
            }
            print("\n")
          }
          println("--")
        }
      }
    }

  }

  //print_by_8(write_atu.map(_._1), 0)
  //println("##########")
  print_by_8(read_atu.map(_._1), 0)
  exit()
*/
}

class LocalReadPort(config: FFTConfig) extends Bundle {
  val req  = config.ReadReq().asOutput
  val resp = config.DataBits().asInput
}

class FFT(val config: FFTConfig) extends Module { // extends RoCC temporarily

  override val io = new RoCCInterfaceFlexMem(config.flex_width, config.flex_a

  // Temporary workaround
  io.mem.req.bits.phys := Bool(true)
  io.flexmem.req.bits.phys := Bool(true)

  // Accelerator master state machine
```

```
  val s_reset :: s_idle :: s_loading :: s_executing :: s_storing :: Nil = Enu
  val state = Reg(init = s_reset)

  // Create IO command routers and the interconnect network
  val syscmdqueue = Module(new Queue(io.cmd.bits.clone, 2, false))
  syscmdqueue.io.enq <> io.cmd
  val syscmd = syscmdqueue.io.deq
  val syscmd_empty = (syscmdqueue.io.count === UInt(0)) && !io.cmd.valid

  val router = Module(new Router(config.num_machines*config.radix, config.num

  val loadmanager:  LoadManager  = Module(new FlexLoadManager(config))
  val execmanager:  ExecManager  = Module(new ExecManager(config))
  val storemanager: StoreManager = Module(new FlexStoreManager(config))


  if(config.use_flex_mem)
  {
    val memarbiter = Module(new FlexMemoryArbiter(2, config.flex_width, confi

    memarbiter.io.requestor(0) <> loadmanager.io.mem
    memarbiter.io.requestor(1) <> storemanager.io.mem
   /*
    val store_memconv = Module(new SmallMemToFlexMem(config))
    store_memconv.io.in <> storemanager.io.mem
    memarbiter.io.requestor(1) <> store_memconv.io.out
    */
    io.flexmem <> memarbiter.io.mem

    io.mem.req.valid := Bool(false) // disable small mem port
  }
  else
  {
    val memarbiter = Module(new SimpleMemoryArbiter(2))

    val load_memconv = Module(new UnsafeFlexMemToSmallMem(config))
    load_memconv.io.in <> loadmanager.io.mem
    memarbiter.io.requestor(0) <> load_memconv.io.out

    val store_memconv = Module(new UnsafeFlexMemToSmallMem(config))
    store_memconv.io.in <> storemanager.io.mem
    memarbiter.io.requestor(1) <> store_memconv.io.out
```

```
  io.mem <> memarbiter.io.mem
  io.flexmem.req.valid := Bool(false)
  io.flexmem.resp.ready := Bool(false)
}

val opcontrollers = Vector.tabulate(config.num_tablets)(t => Module(new Op
val ld_arbiter = Module(new Arbiter({new LoadCmd(config)}, config.num_table
val ex_arbiter = Module(new Arbiter({new ExecCmd(config)}, config.num_table
val st_arbiter = Module(new Arbiter({new StoreCmd(config)},config.num_table

syscmd.ready := opcontrollers.map(_.io.cmd.ready).reduce(_||_)

(0 until config.num_tablets).foreach(t => {
  opcontrollers(t).io.cmd.valid := syscmd.valid &&
    !(opcontrollers.slice(0,t).map(_.io.cmd.ready).fold(Bool(false))(_||_))
  // Take this command if valid and no one above me is ready to take it

  opcontrollers(t).io.cmd.bits.base_address := syscmd.bits.rs1
  opcontrollers(t).io.cmd.bits.stride       := syscmd.bits.rs2

  ld_arbiter.io.in(t) <> opcontrollers(t).io.ld_cmd
  ex_arbiter.io.in(t) <> opcontrollers(t).io.ex_cmd
  st_arbiter.io.in(t) <> opcontrollers(t).io.st_cmd

  opcontrollers(t).io.ld_busy := !loadmanager.io.cmd.ready
  opcontrollers(t).io.ex_busy := !execmanager.io.cmd.ready
  opcontrollers(t).io.st_busy := !storemanager.io.cmd.ready
})

loadmanager.io.cmd  <> ld_arbiter.io.out
execmanager.io.cmd  <> ex_arbiter.io.out
storemanager.io.cmd <> st_arbiter.io.out

io.busy := opcontrollers.map(_.io.busy).reduce(_||_) || !syscmd_empty

// Actually create the machines and connect everything up
val machines = Vector.tabulate(config.num_machines)(n => {
  //create
  val my_insts    = config.schedule.make_instruction_packages(n)
  val my_mem_ids = config.schedule.get_machine_memories(n)
  val my_machine = Module(new Machine(config, my_insts, my_mem_ids))
```

```
    //connect to routers
    my_mem_ids.zipWithIndex.foreach(Function.tupled((gid, lid) => {
      my_machine.io.ext_write(lid)    := loadmanager.io.localports(gid)
      my_machine.io.ext_raddr(lid)    := storemanager.io.localports(gid).req
      storemanager.io.localports(gid).resp := my_machine.io.ext_rdata(lid)

      router.io.in(gid)     := my_machine.io.out(lid)
      my_machine.io.in(lid):= router.io.out(gid)
    }))

    //connect control signals to the machine
    my_machine.io.current_stage := execmanager.io.exec_stage
    my_machine.io.exec_pc.bits  := execmanager.io.exec_pc
    my_machine.io.exec_pc.valid := execmanager.io.exec_valid

    my_machine.io.active_ld_tablet := loadmanager.io.active_tablet
    my_machine.io.active_ex_tablet := execmanager.io.active_tablet
    my_machine.io.active_st_tablet := storemanager.io.active_tablet

    my_machine
  })

  // Needed misc RoCC connections
  io.interrupt := Bool(false)
  Vector(io.iptw, io.dptw, io.pptw).foreach(ptw => {
    ptw.req.valid := Bool(false)
  })
  io.imem.acquire.valid := Bool(false)
  io.imem.grant.ready := Bool(false)

  // Do not ever send a response
  io.resp.valid := Bool(false)
  io.resp.bits.rd := UInt(0)
  io.resp.bits.data := UInt(0)
}

package FFT

import Chisel._
import rocket._
import uncore._
```

```
// Simplified version of HellaCacheArbiter from rocket
// NOT INTERCHANGABLE
class SimpleMemoryArbiter(n: Int) extends Module
{
  val io = new Bundle {
    val requestor = Vec.fill(n){new HellaCacheIO}.flip
    val mem = new HellaCacheIO
  }

  io.mem.req.valid := io.requestor.map(_.req.valid).reduce(_||_)
  io.requestor(0).req.ready := io.mem.req.ready
  for (i <- 1 until n)
    io.requestor(i).req.ready := io.requestor(i-1).req.ready && !io.requestor

  io.mem.req.bits := io.requestor(n-1).req.bits
  io.mem.req.bits.tag := Cat(io.requestor(n-1).req.bits.tag, UInt(n-1, log2Up
  for (i <- n-2 to 0 by -1) {
    val req = io.requestor(i).req
    when (req.valid) {
      io.mem.req.bits.cmd := req.bits.cmd
      io.mem.req.bits.typ := req.bits.typ
      io.mem.req.bits.addr := req.bits.addr
      io.mem.req.bits.tag := Cat(req.bits.tag, UInt(i, log2Up(n)))
      io.mem.req.bits.data := req.bits.data
    }
  }

  for (i <- 0 until n) {
    val resp = io.requestor(i).resp
    val tag_hit = io.mem.resp.bits.tag(log2Up(n)-1,0) === UInt(i)
    resp.valid := io.mem.resp.valid && tag_hit
    resp.bits := io.mem.resp.bits
    resp.bits.tag := io.mem.resp.bits.tag >> UInt(log2Up(n))
  }
}

// Adjusted version of SimpleMemoryArbiter (FlexMemResp is Decoupled not Vali
// NOT INTERCHANGABLE
class FlexMemoryArbiter(n: Int, flex_width: Int, flex_addr_width: Int) extend
{
  val io = new Bundle {
```

```
    val requestor = Vec.fill(n){new FlexMemIO(flex_width, flex_addr_width)}.f
    val mem = new FlexMemIO(flex_width, flex_addr_width)
  }

  io.mem.req.valid := io.requestor.map(_.req.valid).reduce(_||_)
  io.requestor(0).req.ready := io.mem.req.ready
  for (i <- 1 until n)
    io.requestor(i).req.ready := io.requestor(i-1).req.ready && !io.requestor

  io.mem.req.bits := io.requestor(n-1).req.bits
  io.mem.req.bits.tag := Cat(io.requestor(n-1).req.bits.tag, UInt(n-1, log2Up
  for (i <- n-2 to 0 by -1) {
    val req = io.requestor(i).req
    when (req.valid) {
      io.mem.req.bits.cmd := req.bits.cmd
      io.mem.req.bits.bytes := req.bits.bytes
      io.mem.req.bits.addr := req.bits.addr
      io.mem.req.bits.tag := Cat(req.bits.tag, UInt(i, log2Up(n)))
      io.mem.req.bits.data := req.bits.data
    }
  }

  var mem_resp_ready = Bool(false)
  for (i <- 0 until n) {
    val resp = io.requestor(i).resp
    val tag_hit = io.mem.resp.bits.tag(log2Up(n)-1,0) === UInt(i)
    resp.valid := io.mem.resp.valid && tag_hit
    resp.bits := io.mem.resp.bits
    resp.bits.tag := io.mem.resp.bits.tag >> UInt(log2Up(n))

    mem_resp_ready = mem_resp_ready || (resp.ready && tag_hit)

  }
  io.mem.resp.ready := mem_resp_ready
}

package FFT

import Chisel._
import Node._

case class ButterflyConfig(radix: Int, num_multipliers: Int, floating_point:
```

```
  require(Vector(2,4).contains(radix), "BF: Only a radix 2 or 4 butterfly is

/*
  val stages = if(radix == 2) ({
    // in: 0 = i1_r, 1 = i1_i, 2 = tf_r, 3 = tf_i
    // op: 0 => a+b, 1 => a-b
    val conf_premult_add = TimeMultiplexer.BinaryConfig(
      n_inputs=4, operations=Vector((0,1,0),(1,0,1),(2,3,0)),
      reg_all_outputs=true, n_execs=num_multipliers,
      exec_unit_gen=()=>Module(new BFAdder(dtype)), exec_latency=0,
      dtype=dtype
    )
    // in: 0,1,2 = (prior), 3 = i1_r, 4 = tf_r, 5 = tf_i
    // op: 0 => a*b
    val conf_mult = TimeMultiplexer.BinaryConfig(
      n_inputs=6, operations=Vector((0,5,0),(2,3,0),(1,4,0)),
      reg_all_outputs=true, n_execs=num_multipliers,
      exec_unit_gen=()=>Module(new BFMult(fix_pt)(dtype)), exec_latency=0,
      dtype=dtype
    )
    // in: 0,1,2 = (prior)
    // op: 0 => a+b, 1 => a-b
    val conf_postmult_add = TimeMultiplexer.BinaryConfig(
      n_inputs=3, operations=Vector((1,0,1),(1,2,0)),
      reg_all_outputs=false, n_execs=math.ceil(2*(num_multipliers/3.0)).toInt
      exec_unit_gen=()=>Module(new BFAdder(dtype)), exec_latency=0,
      dtype=dtype
    )
    // in: 0,1 = (prior), 2 = i0_r, 3 = i0_i
    // op: 0 => a+b, 1 => a-b
    val conf_complex_add = TimeMultiplexer.BinaryConfig(
      n_inputs=4, operations=Vector((2,0,0),(3,1,0),(2,0,1),(3,1,1)),
      reg_all_outputs=false, n_execs=math.ceil(4*(num_multipliers/3.0)).toInt
      exec_unit_gen=()=>Module(new BFAdder(dtype)), exec_latency=0,
      dtype=dtype
    )

    Vector(conf_premult_add, conf_mult, conf_postmult_add, conf_complex_add)
  }) else null // TODO REMOVE THIS AND THE HACKY CHECKS
  def get_latency(start: Int, end: Int) = stages.slice(start, end).map(_.tota
  def inject_delay = if(stages!=null) stages.map(_.inject_delay).reduce(math.
  def total_latency = if(stages!=null) get_latency(0, stages.length) else 1
```

```
*/
  def inject_delay = 1
  def total_latency: Int = (if(floating_point) (radix+2) else (radix/2 + 1))
  def cpx_three_mults = if(floating_point) false else true
}

trait ButterflyExecutor {
  def mult(a: UInt, b: UInt): UInt
  def add(a: UInt, b: UInt, op: Int=0): UInt
  def muladd(a: UInt, b: UInt, c: UInt, op: Int=0): UInt
}

object FloatButterflyExecutor extends ButterflyExecutor {
  def mult(a: UInt, b: UInt): UInt = {
    val mymod = Module(new hardfloat.mulRecodedFloat32_1)
    mymod.io.a := a
    mymod.io.b := b
    mymod.io.roundingMode := UInt(0)
    mymod.io.out
  }
  def add(a: UInt, b: UInt, op: Int=0): UInt = {
    val mymod = Module(new hardfloat.addSubRecodedFloat32_1)
    mymod.io.a := a
    mymod.io.b := b
    mymod.io.op := UInt(op)
    mymod.io.roundingMode := UInt(0)
    mymod.io.out
  }
  def muladd(a: UInt, b: UInt, c: UInt, op: Int=0): UInt = {
    val mymod = Module(new hardfloat.mulAddSubRecodedFloatN(23,9))
    mymod.io.a := a
    mymod.io.b := b
    mymod.io.c := c
    mymod.io.op := UInt(op)
    mymod.io.roundingMode := UInt(0)
    mymod.io.out
  }
}
case class FixedButterflyExecutor(dwidth: Int, fix_pt: Int) extends Butterfly
  def mult(a: UInt, b: UInt): UInt = {
    val untruncated = a.toSInt * b.toSInt
    untruncated(dwidth-1+fix_pt, fix_pt).toUInt
```

```
  }
  def add(a: UInt, b: UInt, op: Int=0): UInt = {
    require(op==0 || op==1)
    val (as, bs) = (a.toSInt, b.toSInt)
    (if(op==0) (as+bs) else (as-bs)).toUInt
  }
  def muladd(a: UInt, b: UInt, c: UInt, op: Int=0): UInt = add(mult(a, b), c,
}

class Butterfly(config: ButterflyConfig, ioconfig: FFTConfig) extends Module
  val io = new Bundle {
    val in       = Valid(Vec.fill(config.radix){ioconfig.DataMath()}).asInput
    val twiddles = (Vec.fill(config.radix-1){ioconfig.DataMath()}).asInput

    val out      = Valid(Vec.fill(config.radix){ioconfig.DataMath()}).asOutpu
  }

  val exec: ButterflyExecutor =
    if(config.floating_point) FloatButterflyExecutor
    else                      FixedButterflyExecutor(config.dtype.getWidth, c

  def cpx_mult(a: UInt, b: UInt, c: UInt, d: UInt): Tuple2[UInt,UInt] = {
    //(a+bi)(c+di) = ac - bd + i(ad + bc)
    if(config.cpx_three_mults) {
      val ac_p_ad = exec.mult(a, exec.add(c, d, 0))
      val ad_p_bd = exec.mult(exec.add(a, b, 0), d)
      val bc_m_ac = exec.mult(exec.add(b, a, 1), c)

      val scale_real = exec.add(ac_p_ad, ad_p_bd, 1)
      val scale_imag = exec.add(ac_p_ad, bc_m_ac, 0)

      (scale_real, scale_imag)
    }
    else {
      val bd = exec.mult(b, d)
      val bc = exec.mult(b, c)

      val scale_real = exec.muladd(a, c, bd, 1)
      val scale_imag = exec.muladd(a, d, bc, 0)

      (scale_real, scale_imag)
    }
```

```
  }

  def cpx_add(a: Tuple2[UInt,UInt], b: Tuple2[UInt,UInt], op: Int) = {
    (exec.add(a._1, b._1, op), exec.add(a._2, b._2, op))
  }
  def cpx_jadd(a: Tuple2[UInt,UInt], b: Tuple2[UInt,UInt], op: Int) = {
    // return a + jb or a - jb
    val fop = if(op!=0) 0 else 1
    (exec.add(a._1, b._2, fop), exec.add(a._2, b._1, op))
  }

  val s1_result = (0 until config.radix).toVector.map(i => {
    if(i==0) (io.in.bits(i).real, io.in.bits(i).imag)
    else { cpx_mult(io.in.bits(i).real, io.in.bits(i).imag, io.twiddles(i-1).
  })

  val s2_result: Vector[Tuple2[UInt,UInt]] = if(config.radix == 2) {
    Vector(cpx_add(s1_result(0), s1_result(1), 0), cpx_add(s1_result(0), s1_r
  }
  else if(config.radix == 4) {
    /*
    0 = 0 +  1 + 2 +  3 = (0 + 2) +  (1 + 3)
    1 = 0 - j1 - 2 + j3 = (0 - 2) - j(1 - 3)
    2 = 0 -  1 + 2 -  3 = (0 + 2) -  (1 + 3)
    3 = 0 + j1 - 2 - j3 = (0 - 2) + j(1 - 3)
    */

    val r_0p2 = cpx_add(s1_result(0), s1_result(2), 0)
    val r_0m2 = cpx_add(s1_result(0), s1_result(2), 1)
    val r_1p3 = cpx_add(s1_result(1), s1_result(3), 0)
    val r_1m3 = cpx_add(s1_result(1), s1_result(3), 1)

    Vector(
      cpx_add( r_0p2, r_1p3, 0),
      cpx_jadd(r_0m2, r_1m3, 1),
      cpx_add( r_0p2, r_1p3, 1),
      cpx_jadd(r_0m2, r_1m3, 0)
    )
  }
  else throw new Exception("Unsupported radix")

  (0 until config.radix).foreach(i => {
```

```
    io.out.bits(i).real := ShiftRegister(s2_result(i)._1, config.total_latenc
    io.out.bits(i).imag := ShiftRegister(s2_result(i)._2, config.total_latenc
  })
  io.out.valid := ShiftRegister(io.in.valid, config.total_latency)

/*  def cond_reg[T <: Data](din: T, cond: Boolean): T = {if(cond) Reg(next=di

  val s1_mult_result = Vec(config.radix) {config.DataComplex()}
  (0 until config.radix).foreach(i => {
    s1_mult_result(i) := (
      if(i==0) {Delayline(io.in(i), Complex.mult_delay)}
      else
      {
        val untruncated = io.in(i) * io.twiddles(i-1)
        untruncated(config.data_width-1+config.fix_pt, config.fix_pt)
      }
    )
  })
  val s2_data = cond_reg(s1_mult_result, config.bf_pipeline_M)
*/
/*
  if(config.radix == 2) {

    val s0_to_s1_delay = config.get_latency(0,1)
    val s0_to_s3_delay = config.get_latency(0,3)
    println("Butterfly Info: inject_delay=%d, total_latency=%d".format(config

    // Create all the operations and needed delay blocks
    val s0_add  = Module(new TimeMultiplexer.Binary(config.stages(0)))
    val s1_mult = Module(new TimeMultiplexer.Binary(config.stages(1)))
    val s2_add  = Module(new TimeMultiplexer.Binary(config.stages(2)))
    val s3_add  = Module(new TimeMultiplexer.Binary(config.stages(3)))
    val s0_to_s1_tf  = Module(new TimeMultiplexer.StaticDelayblockSeries(conf
    val s0_to_s1_i1r = Module(new TimeMultiplexer.StaticDelayblockSeries(conf
    val s0_to_s3_i0  = Module(new TimeMultiplexer.StaticDelayblockSeries(conf

    // Connect everything
    // Stage 0 inputs
    s0_add.io.in.valid := io.in.valid
    s0_to_s1_tf.io.in.valid := io.in.valid
    s0_to_s1_i1r.io.in.valid := io.in.valid
    s0_to_s3_i0.io.in.valid := io.in.valid
```

```
s0_add.io.in.bits(0) := io.in.bits(1).real
s0_add.io.in.bits(1) := io.in.bits(1).imag
s0_add.io.in.bits(2) := io.twiddles(0).real
s0_add.io.in.bits(3) := io.twiddles(0).imag
s0_to_s1_tf.io.in.bits  := io.twiddles(0)
s0_to_s1_i1r.io.in.bits := io.in.bits(1).real
s0_to_s3_i0.io.in.bits  := io.in.bits(0)

// Stage 1 inputs
s1_mult.io.in.valid := s0_add.io.out.valid

s1_mult.io.in.bits(0) := s0_add.io.out.bits(0)
s1_mult.io.in.bits(1) := s0_add.io.out.bits(1)
s1_mult.io.in.bits(2) := s0_add.io.out.bits(2)
s1_mult.io.in.bits(3) := s0_to_s1_i1r.io.out.bits
s1_mult.io.in.bits(4) := s0_to_s1_tf.io.out.bits.real
s1_mult.io.in.bits(5) := s0_to_s1_tf.io.out.bits.imag

// Stage 2 inputs
s2_add.io.in.valid := s1_mult.io.out.valid

s2_add.io.in.bits(0) := s1_mult.io.out.bits(0)
s2_add.io.in.bits(1) := s1_mult.io.out.bits(1)
s2_add.io.in.bits(2) := s1_mult.io.out.bits(2)

// Stage 3 inputs
s3_add.io.in.valid := s2_add.io.out.valid

s3_add.io.in.bits(0) := s2_add.io.out.bits(0)
s3_add.io.in.bits(1) := s2_add.io.out.bits(1)
s3_add.io.in.bits(2) := s0_to_s3_i0.io.out.bits.real
s3_add.io.in.bits(3) := s0_to_s3_i0.io.out.bits.imag

// Butterfly output
io.out.valid := s3_add.io.out.valid

io.out.bits(0).real := s3_add.io.out.bits(0)
io.out.bits(0).imag := s3_add.io.out.bits(1)
io.out.bits(1).real := s3_add.io.out.bits(2)
io.out.bits(1).imag := s3_add.io.out.bits(3)
```

```
  } else if(config.radix == 4) {

    val s1_valid = io.in.valid
    val s1_mult_result = Vec.fill(config.radix){ioconfig.DataMath()}
    (0 until config.radix).foreach(i => {
      s1_mult_result(i) := (
        if(i==0) {io.in.bits(i)}
        else
        {
          val untruncated = io.in.bits(i) * io.twiddles(i-1)
          untruncated(config.dtype.getWidth-1+config.fix_pt, config.fix_pt)
        }
      )
    })

    val s2_valid = Reg(next=s1_valid)
    val s2_data = Reg(next=s1_mult_result)

    io.out.valid := s2_valid
    def j(din: Complex): Complex = {
      new Complex(-din.imag, din.real)
    }
    io.out.bits(0) := s2_data(0) +   s2_data(1)  + s2_data(2) +   s2_data(3)
    io.out.bits(1) := s2_data(0) - j(s2_data(1)) - s2_data(2) + j(s2_data(3))
    io.out.bits(2) := s2_data(0) -   s2_data(1)  + s2_data(2) -   s2_data(3)
    io.out.bits(3) := s2_data(0) + j(s2_data(1)) - s2_data(2) - j(s2_data(3))
  }
*/
}

class BFAdder(dtype: UInt) extends TimeMultiplexer.BinaryExec(2)(dtype) {
  io.out := io.in(0) + Mux(io.func===Bits(1), -io.in(1), io.in(1))
}
class BFMult(fix_pt: Int)(dtype: UInt) extends TimeMultiplexer.BinaryExec(2)(
  val untruncated = io.in(0)*io.in(1)
  io.out := untruncated(dtype.getWidth-1+fix_pt, fix_pt)
}

package FFT

import Chisel._
import Node._
```

```
object Complex {
  var use_four_mults = false

  def apply(dwidth: Int = -1) = {
    if(dwidth > 0) new Complex(UInt(width=dwidth), UInt(width=dwidth)) else n
  }
}
class Complex(val real: UInt, val imag: UInt) extends Bundle {
  override def clone() = new Complex(real.clone, imag.clone).asInstanceOf[thi

  def * (r: Complex): Complex =
  {
    val a = real; val b = imag; val c = r.real; val d = r.imag;

    if(Complex.use_four_mults)
    {
      val ac = a*c; val bd = b*d; val ad = a*d; val bc = b*c;
      new Complex(ac - bd, ad + bc)
    }
    else //use three mults
    {
      val ac_p_ad = a * (c + d)
      val ad_p_bd = (a + b) * d
      val bc_m_ac = (b - a) * c

      new Complex(ac_p_ad - ad_p_bd, ac_p_ad + bc_m_ac)
    }
  }
  def + (r: Complex): Complex =
  {
    new Complex(real + r.real, imag + r.imag)
  }
  def - (r: Complex): Complex =
  {
    new Complex(real - r.real, imag - r.imag)
  }

  def apply(sp: Int, ep: Int) =
  {
    new Complex(real(sp, ep).toUInt, imag(sp, ep).toUInt)
  }
```

```
} // Complex class

package FFT

import Chisel._

object SystemCommands {
  val num_commands = 1
  val cmd_width = log2Up(num_commands)

  val run :: Nil = Enum(UInt(), num_commands)
}

class Packet[T <: Data](addr_width: Int)(data: T) extends Bundle {
  val address = Bits(OUTPUT, addr_width)
  val payload = data.clone.asOutput

  override def clone = new Packet(addr_width)(data).asInstanceOf[this.type]
}
object Packet
{
  def apply[T <: Data](addr_width: Int)(data: T) = {(new Packet(addr_width))(
}

object Delayline {
  def apply[T <: Data](in: T, latency: Int): T = {
    assert(latency >= 0)
    if(latency == 0)
      in
    else
      apply(RegNext(in), latency-1)
  }
}

package InterconnectNetwork

import Chisel._
import Node._

case class CrossbarConfig(n_inputs: Int, n_outputs: Int, npi_vq: Int = 1,
                          out_q_depth: Int = 2, in_vq_depth: Int = 2,
                          arbiter_max_size: Option[Int] = None, round_robin:
```

```
{
  /*
    n_inputs is number of inputs
    n_outputs is number of outputs
    npi_vq is number (per input) of virtual queues
  */
  require(n_inputs > 0, "InterconnectNetwork.Crossbar: Number of inputs must
  require(n_outputs > 0, "InterconnectNetwork.Crossbar: Number of outputs mus
  require(out_q_depth >= 0,
    "InterconnectNetwork.Crossbar: Output queue depth must be nonnegative (>=
  require(in_vq_depth > 0 || npi_vq == 0,
    "InterconnectNetwork.Crossbar: Virtual input queue depth must be positive
    "Set npi_vq=0 to disable input queuing.")
  require(npi_vq >= 0,
    "InterconnectNetwork.Crossbar: Number of virtual queues per input must be
  require(npi_vq <= n_outputs,
    "InterconnectNetwork.Crossbar: Number of virtual queues per input cannot
  arbiter_max_size.foreach(i => require(i > 1,
    "InterconnectNetwork.Crossbar: Maximum arbiter size must be >= 2 or None"
//  arbiter_max_mux.foreach(i => require(i > 1,
//    "InterconnectNetwork.Crossbar: Maximum arbiter used mux must be power-o

  // Construct the arbiter tree requirements greedily
  def construct_arbiter_tree(max_size: Int): Seq[Seq[Int]] = {
    val arbiter_stages = math.ceil(math.log(n_inputs)/math.log(max_size)).toI
    (Vector.fill(arbiter_stages)(true)).foldLeft(Vector.empty[Vector[Int]])((
      val stage_ins = tree.lastOption.map(_.length).getOrElse(n_inputs)
      val my_stage = if(stage_ins <= max_size) {Vector(stage_ins)}
      else {
        val full_arbs = Vector.fill(stage_ins/max_size)(max_size)
        val partial_arb = stage_ins % max_size
        if(partial_arb>0) full_arbs:+partial_arb else full_arbs
      }
      tree :+ my_stage
    })
  }
  val arbiter_tree: Seq[Seq[Int]] =
    arbiter_max_size.map(ams => construct_arbiter_tree(ams)).getOrElse(Vector

}
object StatelessCrossbarConfig
{
```

```
  def apply(n_inputs: Int, n_outputs: Int) = CrossbarConfig(
    n_inputs = n_inputs, n_outputs = n_outputs,
    npi_vq = 0, out_q_depth = 0, in_vq_depth = 0,
    arbiter_max_size = None, round_robin = false
  )
}

class Crossbar[T <: Data](c: CrossbarConfig)(data: T) extends DecoupledInterc
  // io.in (PKT) -> v_input_q (PKT) -> [logic] -> arbiter (DAT) -> output_q (

  // Instantiate Arbiters for the outputs
  //val arbiter = Array.fill(c.n_outputs)( (new RRArbiter(c.n_inputs)){data}
  val arbiter = Vector.fill(c.n_outputs)( Module(new RRArbiterTree(data, c.ar

  // Handle io.in -> arbiter connections
  for(i <- 0 until c.n_inputs)
  {
    val i2a_count = math.max(c.npi_vq, 1) // Must have at least 1 lane for in
    val i2a_pt = Vector.fill(i2a_count)( PassThrough(Decoupled(Packet(addr_wi

    // Determine how input data is split between lanes
    val out2lane  = Vector.tabulate(c.n_outputs)(_ % i2a_count)
    val lane2outs = (0 until i2a_count).toVector.map(l => out2lane.zipWithInd
    // Deferring fully generalizing this implementation since % is much more
    //    when npi_vq is a power-of-2 (otherwise using a lookup table anyway)
    // TODO: optimize for case when n_outputs/npi_vq is a power-of-2
    def using_lane(v: Int): Bool = {
      val in_addr = io.in(i).bits.address
      if(i2a_count == 1) { Bool(true) } // Only have 1 lane, so clearly using
      else if(isPow2(i2a_count)) {
        in_addr(log2Up(i2a_count)-1,0) === UInt(v)
      }
      else {
        val addr_cmp = lane2outs(v).map(o => in_addr === UInt(o))
        addr_cmp.reduce(_ || _)
      }
    }

    // Connect io.in -> virtual input queues -> i2a_pt
    if(c.npi_vq == 0) {io.in(i) <> i2a_pt(0).io.in}
    else
    {
```

```
    val v_input_q  = Vector.fill(c.npi_vq)( Module(new Queue(Packet(addr_wi
    val selector = Vector.tabulate(c.npi_vq)( v => using_lane(v) )

    val readies = Vector.tabulate(c.npi_vq)( v => (v_input_q(v).io.enq.read
    io.in(i).ready := readies.reduce(_ || _)
    for(v <- 0 until c.npi_vq)
    {

      v_input_q(v).io.enq.bits  := io.in(i).bits
      v_input_q(v).io.enq.valid := io.in(i).valid && selector(v)

      v_input_q(v).io.deq <> i2a_pt(v).io.in
    }
  }

  //Connect i2a_pt -> arbiter
  for(l <- 0 until i2a_count)
  {
    // This specific lane will connect to lane2out(l) outputs
    val selectors = lane2outs(l).map(o => (i2a_pt(l).io.out.bits.address ==

    val readies = lane2outs(l).zip(selectors) map Function.tupled((o,select
    i2a_pt(l).io.out.ready := readies.reduce(_ || _)
    lane2outs(l).zip(selectors) foreach Function.tupled((o,selector) => {
      arbiter(o).io.in(i).bits  := i2a_pt(l).io.out.bits.payload
      arbiter(o).io.in(i).valid := i2a_pt(l).io.out.valid && selector
    })
  }
}

// Handle arbiter -> io.out connections

// Instantiate and connect output Queues
for(o <- 0 until c.n_outputs)
{
  if(c.out_q_depth == 0) { arbiter(o).io.out <> io.out(o) }
  else {
    val output_q = Module(new Queue(data, c.out_q_depth))
    arbiter(o).io.out <> output_q.io.enq
    output_q.io.deq <> io.out(o)
  }
}
```

```
}


package FFT

import Chisel._
import rocket._
import uncore._

class ExecCmd(fc: FFTConfig) extends Bundle {
  val tablet = UInt(width=fc.tablet_id_width)
  override def clone = new ExecCmd(fc).asInstanceOf[this.type]
}

class ExecManager(val fc: FFTConfig) extends Module {
  val io = new Bundle {
    val cmd = Decoupled(new ExecCmd(fc)).flip

    val exec_stage = UInt(width=log2Up(fc.fft_size)).asOutput
    val exec_pc    = UInt(width=log2Up(fc.fft_size)).asOutput
    val exec_valid = Bool().asOutput

    val active_tablet = UInt(width=fc.tablet_id_width).asOutput
  }

  val s_idle :: s_executing :: s_drain :: Nil = Enum(UInt(), 3)
  val state = Reg(init = s_idle)

  // Execution trackers
  val active_tablet = Reg(init=UInt(fc.num_tablets))

  val exec_stage  = Reg(init = UInt(0, width=log2Up(fc.fft_size)))
  val exec_pc_ctr = Reg(init = UInt(0, width=log2Up(math.max(fc.fft_size, fc.
  val starting_exec = Bool(); starting_exec := Bool(false)
  // If can inject an instruction every cycle, do it
  val can_inject_inst = if(fc.butterfly.inject_delay == 1) (Bool(true)) else
  // Otherwise, implement a counter to create 1 cycle pulse every inject_dela
    val inject_counter = Reg(init=UInt(0, width=log2Up(fc.butterfly.inject_de
    when(starting_exec) { inject_counter := UInt(0) }
    when(state === s_executing) {
      inject_counter := inject_counter + UInt(1)
      when(inject_counter === UInt(fc.butterfly.inject_delay-1)) { inject_cou
```

```
    }
    (inject_counter === UInt(0))
  })
  val injecting_inst = Bool(); injecting_inst := Bool(false)

  Vector(io.cmd.ready).foreach(_ := Bool(false))
  switch( state ) {
    is( s_idle ) {
      io.cmd.ready := Bool(true)
      when( io.cmd.fire() ) {
        exec_stage := UInt(0)
        exec_pc_ctr := UInt(0)
        starting_exec := Bool(true)
        active_tablet := io.cmd.bits.tablet
        state := s_executing
      }
    }

    is( s_executing ) { when( can_inject_inst ) {
      injecting_inst := Bool(true)
      exec_pc_ctr := exec_pc_ctr + UInt(1)
      when(exec_pc_ctr === UInt(fc.fft_size/(fc.num_machines*fc.radix)-1))
      {
        state := s_drain
        exec_pc_ctr := UInt(0) // will re-use counter to track drain time
      }
    } }

    is( s_drain )
    {
      exec_pc_ctr := exec_pc_ctr + UInt(1)
      when(exec_pc_ctr === UInt(fc.exec_drain_time))
      {
        exec_pc_ctr := UInt(0)
        when(exec_stage === UInt(fc.schedule.stages-1)) {
          active_tablet := UInt(fc.num_tablets)
          state := s_idle
        }.otherwise{
          exec_stage := exec_stage + UInt(1)
          state := s_executing
        }
      }
```

```
    }
  }

  io.active_tablet := active_tablet

  io.exec_stage := exec_stage
  io.exec_pc    := exec_pc_ctr
  io.exec_valid := injecting_inst
}

package FFT

import Chisel._
import rocket._
import uncore._

class FlexLoadManager(val fc: FFTConfig) extends LoadManager {
  //CONFIGURATION
  val max_tag = 8
  require(isPow2(max_tag))
  val points_per_req = fc.load_points_per_request

  //IO
  val io = new LoadManagerIO(fc)

  //STATE MACHINE
  val s_idle :: s_issuing :: s_ext_draining :: s_int_draining :: Nil = Enum(U
  val state = Reg(init = s_idle)

  // GENERATE REQUESTS
  // Setup a Vec to associate response tags with target system location
  val tag_active  = Vec.fill(max_tag){Reg(init = Bool(false))}
  val tag_sysaddr = Vec.fill(max_tag){Reg(UInt(width=log2Up(fc.fft_size)))}

  val next_avail_tag = PriorityEncoder(tag_active.map(!_))
  val have_avail_tag = !tag_active.reduce(_&&_) // need at least 1 unused tag
  val using_no_tags  = !tag_active.reduce(_||_)

  // Execution trackers
  val active_tablet = Reg(init=UInt(fc.num_tablets))
  val req_sysaddr = Reg(init = UInt(0, width=log2Up(fc.fft_size)))
  val req_memaddr = Reg(init = UInt(0, width=fc.xprlen))
```

```
  val req_stride = Reg(init = UInt(4, width=fc.xprlen)) // stride to next req
  val int_counter = Reg(init = UInt(0, width=log2Up(fc.load_drain_time)))

  Vector(io.cmd.ready, io.mem.req.valid).foreach( _ := Bool(false) )
    // Set some conservative defaults for the RV signals
  switch(state) {
    is( s_idle ) { // waiting for a command
      io.cmd.ready := Bool(true)
      when(io.cmd.fire()) {
        req_sysaddr := UInt(0)
        req_memaddr := io.cmd.bits.start_addr
        req_stride := io.cmd.bits.stride * UInt(points_per_req)
        active_tablet := io.cmd.bits.tablet

        state := s_issuing
      }
    }

    is( s_issuing ) { // issuing load req to external system
      when( have_avail_tag ) {
        io.mem.req.valid := Bool(true)
        when( io.mem.req.ready ) { // request fired so
          // Setup the tag
          tag_active(next_avail_tag) := Bool(true)
          tag_sysaddr(next_avail_tag) := req_sysaddr
          // See if done
          when(req_sysaddr === UInt(fc.fft_size-points_per_req)) { state := s
          // Advance to next request
          req_sysaddr := req_sysaddr + UInt(points_per_req)
          req_memaddr := req_memaddr + req_stride
        }
      }
    }

    is( s_ext_draining ) { // finish getting responses from outside
      when( using_no_tags ) {
        state := s_int_draining
        int_counter := UInt(0)
      }
    }

    is( s_int_draining ) { // make sure things clear to inside local memories
```

```
      int_counter := int_counter + UInt(1)
      when(int_counter === UInt(fc.load_drain_time-1)) {
        state := s_idle
        active_tablet := UInt(fc.num_tablets)
      }
    }
  }
  // Connect to memory request port
  io.mem.req.bits.addr := req_memaddr
  io.mem.req.bits.tag := next_avail_tag
  io.mem.req.bits.cmd := MXRD // perform a load (MXWR for stores)
  io.mem.req.bits.bytes := UInt(2*fc.data_width/8*points_per_req)
  io.mem.req.bits.data := Bits(0) // we're not performing any stores...

  io.active_tablet := active_tablet

  //HANDLE RESPONSES
  // Register external memory responses
  io.mem.resp.ready := Bool(true) // Handle all responses immediately
  val reg_mem_resp_v = Reg(next=io.mem.resp.valid)
  val reg_mem_resp_d = Reg(next=io.mem.resp.bits)
  when( reg_mem_resp_v ) { tag_active(reg_mem_resp_d.tag) := Bool(false) }
    // Clear tag

  // Route responses to the appropriate internal ports
  // First, extract the individual responses and their target addresses
  val resp_valid = reg_mem_resp_v
  val resp_base_address = tag_sysaddr(reg_mem_resp_d.tag)
  val resp_addresses:   Seq[UInt] = Vector.tabulate(points_per_req)(p => resp
  val resp_extract_dat: Seq[UInt] = Vector.tabulate(points_per_req)(p => {
    val lsb = (2*fc.data_width)*p
    val msb = (2*fc.data_width)-1 + lsb
    reg_mem_resp_d.data(msb,lsb)
  })
  val resp_prepped_dat = resp_extract_dat.map(fc.unpack(_))

  // Put through an ATU
  val write_atus = Vector.fill(points_per_req)(Module(new WriteATU(fc,fc.writ
  write_atus.zip(resp_addresses zip resp_prepped_dat).foreach(t => {
    val write_atu = t._1; val resp_address = t._2._1; val resp_payload = t._2
    write_atu.io.in.valid := resp_valid
    write_atu.io.in.bits.address := resp_address
```

```
    write_atu.io.in.bits.payload := resp_payload
  })

  // Use a crossbar to connect ATU out to local memory ports
  val crossbar = Module(new InterconnectNetwork.Crossbar(
    InterconnectNetwork.StatelessCrossbarConfig(n_inputs=points_per_req, n_ou
  )
  // Input side of the crossbar
  (write_atus.map(_.io.out) zip crossbar.io.in).foreach(t => {
    val watu_out = t._1; val c_in = t._2
    c_in.valid := watu_out.valid
    c_in.bits  := watu_out.bits
  })
  // Output side of the crossbar, id is important to use it explicitely
  (0 until fc.num_dmem).foreach(o => {
    val c_out = crossbar.io.out(o); val dmem_wp = io.localports(o)
    dmem_wp.valid := c_out.valid
    dmem_wp.bits  := c_out.bits
    c_out.ready := Bool(true)
  })
}

package FFT

import Chisel._
import rocket._
import uncore._

class FlexStoreManager(val fc: FFTConfig) extends StoreManager {
  // CONFIGURATION
  val max_tag = 8
  require(isPow2(max_tag))
  val points_per_req = fc.store_points_per_request
  val expected_acks = fc.fft_size/points_per_req

  // IO
  val io = new StoreManagerIO(fc)

  // STATE MACHINE
  val s_idle :: s_polling :: s_draining :: Nil = Enum(UInt(), 3)
  // s_polling means asking local for data and also issuing some data to mem
  // s_draining means issuing last data to mem and waiting for acks
```

```
  val state = Reg(init = s_idle)

  val metadata_q = Module(new Queue(new LocalReqMeta, 4))
  class LocalReqMeta extends Bundle {
    //val memaddr = UInt(width=fc.xprlen)
    val order_banks = Vec.fill(points_per_req)(UInt(width=log2Up(fc.num_dmem)
    override def clone = new LocalReqMeta().asInstanceOf[this.type]
  }
  class FullLocalReq() extends Bundle {
    val requests = Vec.fill(points_per_req)(fc.FullReadReq())
    val metadata = new LocalReqMeta
    override def clone = new FullLocalReq().asInstanceOf[this.type]
  }
  val req_splitter = Module( new RV_Split(new FullLocalReq(), 2) )
  val resp_merger  = Module( new RV_Merge(new LocalReqMeta, Vec.fill(fc.num_d

  // Execution State
  // used system-wide; also, state, defined above
  val active_tablet = Reg(init=UInt(fc.num_tablets))
  // used to craft internal memory request (part 1)
  val p1_req_sysaddr = Reg(init = UInt(0, width=log2Up(fc.fft_size)))
  // used to craft external memory request (part 2)
  val p2_req_memaddr = Reg(init = UInt(0, width=fc.xprlen))
  val p2_req_stride = Reg(init = UInt(4, width=fc.xprlen))
  // used to process external memory response (part 3)
  val p3_rcvd_acks   = Reg(init = UInt(0, width=log2Up(expected_acks+1)))

  val p1_req_valid = Bool() // setup by state machine
  val p1_req_ready = Bool() // used by state machine
  Vector(io.cmd.ready, p1_req_valid).foreach( _ := Bool(false) )

  switch(state) {
    is( s_idle ) {
      io.cmd.ready := Bool(true)
      when(io.cmd.fire()) {
        // basically, do a MODULE-WIDE setup/reset to start exec
        active_tablet := io.cmd.bits.tablet
        p1_req_sysaddr := UInt(0)
        p2_req_memaddr := io.cmd.bits.start_addr
        p2_req_stride := io.cmd.bits.stride * UInt(points_per_req)
        p3_rcvd_acks  := UInt(0)
```

```
        state := s_polling
      }
    }

    is( s_polling ) {
      p1_req_valid := Bool(true)
      when(p1_req_ready) {
        p1_req_sysaddr := p1_req_sysaddr + UInt(points_per_req)
        when(p1_req_sysaddr === UInt(fc.fft_size-points_per_req)) { state :=
      }
    }

    is( s_draining ) {
      when(p3_rcvd_acks === UInt(expected_acks)) {
        state := s_idle
        active_tablet := UInt(fc.num_tablets)
      }
    }
  }
  // PART 1
  // Output from state machine logic is:
  // p1_req_valid, p1_req_sysaddr (calculate mem_addr later)
  // Still need to setup inputs fromthis part: p1_req_ready

  val atus = Vector.fill(points_per_req)(Module(new ReadATU(fc, fc.read_atu))
  val p1_sysaddrs = Vector.tabulate(points_per_req)(i => p1_req_sysaddr + UIn
  (0 until points_per_req).foreach(i => {
    val atu = atus(i)
    val sysaddr = p1_sysaddrs(i)
    atu.io.in.valid := p1_req_valid
    atu.io.in.bits  := sysaddr
    // ignoring ready since will = p1_req_ready

    atu.io.out.ready := p1_req_ready
    req_splitter.io.in.bits.requests(i) := atu.io.out.bits
    req_splitter.io.in.bits.metadata.order_banks(i) := atu.io.out.bits.addres
  })

  req_splitter.io.in.valid := atus.map(_.io.out.valid).reduce(_||_)
  p1_req_ready := req_splitter.io.in.ready

  // Part 1->2
```

```
// Subpart A: Metadata Queue (uses splitter and merger port 0)
req_splitter.io.out(0).ready := metadata_q.io.enq.ready
metadata_q.io.enq.valid := req_splitter.io.out(0).valid
metadata_q.io.enq.bits  := req_splitter.io.out(0).bits.metadata

resp_merger.io.in1 <> metadata_q.io.deq

// Subpart B: Actual Localports reads (uses splitter port 1)
val localreq_stall = Bool()
req_splitter.io.out(1).ready := !localreq_stall

val crossbar = Module(new InterconnectNetwork.Crossbar(
  InterconnectNetwork.StatelessCrossbarConfig(n_inputs=points_per_req, n_ou
)
//   connect to crossbar input (by points_per_req)
(0 until points_per_req).foreach(i => {
  crossbar.io.in(i).valid := req_splitter.io.out(1).valid
  crossbar.io.in(i).bits.address  := req_splitter.io.out(1).bits.requests(i
  crossbar.io.in(i).bits.payload  := req_splitter.io.out(1).bits.requests(i
})
//   connect to crossbar output, dmems, and local response merger (by num_d
val localreq_replay_valid = Reg(init=Bool(false))
val localreq_replay_data = Reg(Vec.fill(fc.num_dmem)(fc.ReadReq()))
val crossbar_out_valid = crossbar.io.out.map(_.valid).reduce(_||_)
when(!localreq_stall) {
  localreq_replay_valid := crossbar_out_valid
  (0 until fc.num_dmem).foreach(o => localreq_replay_data(o) := crossbar.io
}
(0 until fc.num_dmem).foreach(o => {
  crossbar.io.out(o).ready := Bool(true)
  io.localports(o).req := Mux(localreq_stall, localreq_replay_data(o), cros
  resp_merger.io.in2.bits(o) := io.localports(o).resp
})
resp_merger.io.in2.valid := localreq_replay_valid

localreq_stall := !resp_merger.io.in2.ready

// Response merge out now has unified response
// Part 2 construct external memory request
val p2_ordered_data = Vec.tabulate(points_per_req)(o => {
  val target_bank = resp_merger.io.out.bits.data1.order_banks(o)
  resp_merger.io.out.bits.data2(target_bank)
```

```
  })
  val p2_ordered_packed_data = Vector.tabulate(points_per_req)(o => fc.pack(p
  val p2_prepared_data = p2_ordered_packed_data.reverse.reduce(Cat(_,_))

  // connect unified response to memory req port (write)
  io.mem.req.bits.addr := p2_req_memaddr
  io.mem.req.bits.tag := UInt(0) // do not care about tag
  io.mem.req.bits.cmd := MXWR // perform a store
  io.mem.req.bits.bytes := UInt(2*fc.data_width/8*points_per_req)
  io.mem.req.bits.data := p2_prepared_data

  when(io.mem.req.fire()) {
    p2_req_memaddr := p2_req_memaddr + p2_req_stride
  }

  io.mem.req.valid := resp_merger.io.out.valid
  resp_merger.io.out.ready := io.mem.req.ready

  // Handle memory responses by incrementing counter
  io.mem.resp.ready := Bool(true)
  when( io.mem.resp.fire() ) {
    p3_rcvd_acks := p3_rcvd_acks + UInt(1)
  }

  io.active_tablet := active_tablet
}

package FFT

import Chisel._
import rocket._
import uncore._

class FlexMemReq(data_width: Int, addr_width: Int) extends Bundle {
  val addr  = UInt(width=addr_width)
  val phys  = Bool()
  val data  = UInt(width=data_width)
  val tag   = UInt(width=8)
  val cmd   = UInt(width=4)
  val bytes = UInt(width=log2Up(data_width/8)+1) // # of bytes to deal with
  override def clone = new FlexMemReq(data_width, addr_width).asInstanceOf[th
}
```

```
class FlexMemResp(data_width: Int) extends Bundle {
  val data = UInt(width=data_width)
  val tag = UInt(width=8)
  val cmd = UInt(width=4)
  override def clone = new FlexMemResp(data_width).asInstanceOf[this.type]
}
class FlexMemIO(data_width: Int, addr_width: Int) extends Bundle {
  val req  = Decoupled(new FlexMemReq(data_width, addr_width))
  val resp = Decoupled(new FlexMemResp(data_width)).flip
}

class RoCCInterfaceFlexMem(flex_width: Int, flex_addr_width: Int) extends RoC
  val flexmem = new FlexMemIO(flex_width, flex_addr_width)
}

class SmallMemToFlexMem(c: FFTConfig) extends Module {
  val io = new Bundle {
    val in = (new HellaCacheIO).flip
    val out = new FlexMemIO(c.flex_width, c.flex_addr_width)
  }

  // Request side
  io.out.req.valid := io.in.req.valid
  io.in.req.ready  := io.out.req.ready
  io.out.req.bits.addr  := io.in.req.bits.addr
  io.out.req.bits.cmd   := io.in.req.bits.cmd
  io.out.req.bits.data  := io.in.req.bits.data
  io.out.req.bits.bytes := Mux(io.in.req.bits.typ === MT_D, UInt(c.data_width
  io.out.req.bits.phys  := io.in.req.bits.phys
  io.out.req.bits.tag   := io.in.req.bits.tag

  // Response side
  io.in.resp.valid  := io.out.resp.valid
  io.out.resp.ready := Bool(true)
  io.in.resp.bits.data := io.out.resp.bits.data
  io.in.resp.bits.tag  := io.out.resp.bits.tag
  io.in.resp.bits.cmd  := io.out.resp.bits.cmd
}

class UnsafeFlexMemToSmallMem(c: FFTConfig) extends Module {
  val io = new Bundle {
    val in = new FlexMemIO(c.flex_width, c.flex_addr_width).flip
```

```
    val out = (new HellaCacheIO)
  }

  // Request side
  io.out.req.valid := io.in.req.valid
  io.in.req.ready  := io.out.req.ready
  io.out.req.bits.addr  := io.in.req.bits.addr
  io.out.req.bits.cmd   := io.in.req.bits.cmd
  io.out.req.bits.data  := io.in.req.bits.data
  io.out.req.bits.typ := MT_D
  io.out.req.bits.phys  := io.in.req.bits.phys
  io.out.req.bits.tag   := io.in.req.bits.tag

  // Response side
  io.in.resp.valid  := io.out.resp.valid
  // WARNING: IGNORING io.in.resp.ready
  io.in.resp.bits.data := io.out.resp.bits.data
  io.in.resp.bits.tag  := io.out.resp.bits.tag
  io.in.resp.bits.cmd  := io.out.resp.bits.cmd
}

package FFT

import Chisel._
import Node._
import fft_mcgen.InstructionPackage

case class Inst(config: FFTConfig) extends Bundle {
  val raddr    = Vec.fill(config.radix){Bits(width=config.dmem_addr_width)}
    // raddr is, for bank n, what address should we read
  val op_order = Vec.fill(config.radix){Bits(width=log2Up(config.radix))}
    // op_order is, for the nth data point, which bank it should be coming fr
  val twiddle  = Vec.fill(config.radix-1){Bits(width=log2Up(config.fft_size))

  val wid   = Vec.fill(config.radix){Bits(width=log2Up(config.num_dmem))}
  val waddr = Vec.fill(config.radix){Bits(width=config.dmem_addr_width)}
  override def clone = Inst(config).asInstanceOf[this.type]
}

class InstROM(config: FFTConfig, insts: IndexedSeq[IndexedSeq[InstructionPack
  val stages = insts.length
  val depth = insts(0).length
```

```
  val stage_width = log2Up(stages)
  val addr_width = log2Up(depth)
  val io = new Bundle {
    val rstage = Bits(INPUT, stage_width)
    val raddr  = Bits(INPUT, addr_width)

    val rinst = Inst(config).asOutput
  }

  def make_const(in: InstructionPackage): Inst = {
    val out = Inst(config)

    def get_bank_raddr(bank: Int) = {
      in.reads.filter(_._1==memory_ids(bank))(0)._2
    }

    (0 until config.radix).foreach(n => {
      out.raddr(n) := Bits(get_bank_raddr(n))

      out.op_order(n) := Bits(in.reads(n)._1)

      out.wid(n)   := Bits(in.writes(n)._1)
      out.waddr(n) := Bits(in.writes(n)._2)
    })
    (0 until config.radix-1).foreach(n => {
      out.twiddle(n) := Bits(in.twiddle(n))
    })

    out
  }

  val rom = Vec.fill(stages){Vec.fill(depth){Inst(config)}}
  (0 until stages).foreach(s => {(0 until depth).foreach(k => {
    rom(s)(k) := make_const(insts(s)(k))
  })})

  val reg_rstage = Reg(next=io.rstage)
  val reg_raddr  = Reg(next=io.raddr)

  io.rinst := rom(reg_rstage)(reg_raddr)
}
```

```
package InterconnectNetwork

import Chisel._

class Packet[T <: Data](addr_width: Int)(data: T) extends Bundle {
  val payload = data.clone
  val address = UInt(width=addr_width)
  override def clone = new Packet(addr_width)(data).asInstanceOf[this.type]
}
object Packet
{
  def apply[T <: Data](addr_width: Int)(data: T): Packet[T] = {(new Packet(ad
}

abstract class DecoupledInterconnect[T <: Data](val n_inputs: Int, val n_outp
  val addr_width = log2Up(n_outputs)
  val io = new Bundle {
    val in  =(Vec.fill(n_inputs) {Decoupled(Packet(addr_width)(data))}).flip
    val out  = Vec.fill(n_outputs){Decoupled(data)}
  }
}

class PassThrough[T <: Bundle](data: T) extends Module {
  override val io = new Bundle {
    val in = data.clone.flip()
    val out = data.clone
  }
  io.out <> io.in
}
object PassThrough
{
  def apply[T <: Bundle](data: T) = {Module(new PassThrough(data))}
}
/*
class PipelinedRRArbiter[T <: Data](n_inputs: Int, max_mux_size: Int = -1, re
  // this class makes arbitration decision within one cycle
  require(data.isDirectionless, "InterconnectNetwork.PipelinedRRArbiter: [=>a
  require(!(data eq data), "InterconnectionNetwork.PipelinedRRArbiter: [=>dat
  require(max_mux_size == -1 || (max_mux_size > 1 && isPow2(max_mux_size)),
    "InterconnectNetwork.PipelinedRRArbiter: Max mux size must be -1 (no max)

  // Design the tree parameters
```

```
  def construct_mux_tree:Array[Array[Int]] = {
    val mux_stages = math.ceil(math.log(n_inputs)/math.log(max_mux_size)).toI
    val tree = Array.fill(mux_stages)(new Array[Int](0))
    for(s <- 0 until tree.length) {
      val stage_ins = (if(s==0) n_inputs else tree(s-1).length)
      if(stage_ins <= max_mux_size) {tree(s) = Array(stage_ins)}
      else {
        val full_muxs = Array.fill(stage_ins/max_mux_size)(max_mux_size)
        val partial_mux = stage_ins % max_mux_size
        val all_muxs = (if(partial_mux>0) full_muxs:+partial_mux else full_mu
        tree(s) = all_muxs
      }
    }

    tree
  }
  val mux_tree_spec = (if(max_mux_size == -1) Array(Array(n_inputs)) else cor

  // IO and oracle
  val io = (new ioArbiter(n_inputs)){data}

  if(false && mux_tree_spec.length==1 && !reg_input_data) {
    // Handle special case where don't need pipelined mux tree
    val arbiter = (new RRArbiter(n_inputs)){data}
    (0 until n_inputs).foreach(i => io.in(i) <> arbiter.io.in(i))
    arbiter.io.out <> io.out
    io.chosen := arbiter.io.chosen
  }
  else {
    val stall = !io.out.ready

    val oracle = (new RRArbiter(n_inputs)){Bool()} // Data will not be used
    (0 until n_inputs).foreach(i => {
      oracle.io.in(i).valid := io.in(i).valid
      io.in(i).ready := oracle.io.in(i).ready
    })
    oracle.io.out.ready := io.out.ready && !stall

    def make_stalled_reg[S <: Data](in: S): S = {
      val reg = Reg(){in.clone}
      when(!stall) {reg := in}
      reg
```

```
    }

    val chosen_in = oracle.io.chosen
    val data_in = (0 until n_inputs).toArray.map(i => io.in(i).bits)
    val valid_in = oracle.io.out.valid

    val myVec = Vec(n_inputs){data}
    (0 until n_inputs).map(i => myVec(i) := data_in(i))
//    io.chosen := chosen_in
//    io.out.valid := valid_in
//    io.out.bits := myVec(chosen_in)
    // make all the muxes and wire them up
    val ps_width = log2Up(if(max_mux_size == -1) n_inputs else max_mux_size)
    val carry_out = mux_tree_spec.zipWithIndex.foldLeft((chosen_in, data_in,
      // each stage adds a level of muxs. must pass along chosen, the output
      // stage is defined by its stage id and correspondant mux tree spec ent
      val stage_spec = processing._1; val stage = processing._2

      val registering = (stage>0 || reg_input_data)
      val stage_chosen = if(registering) make_stalled_reg(carrying._1) else c
      val stage_ins    = if(registering) carrying._2.map(make_stalled_reg(_))
      val stage_valid  = if(registering) make_stalled_reg(carrying._3) else c
      Predef.assert(stage_ins.length==stage_spec.reduce(_+_), "IN.PRR: Ill-fo

      val stage_select = stage_chosen(math.min(log2Up(n_inputs)-1, (stage+1)*

      val out_cpt = (stage_spec.zipWithIndex.map(Function.tupled((num, idx) =
      val out_cpt_port = stage_spec.map(Array.tabulate(_)(p=>p)).reduce(_++_)

      val stage_vecs = stage_spec.map(n => Vec(n){data})
      (0 until stage_ins.length).toArray.map(i => stage_vecs(out_cpt(i))(out_

      val stage_outs = stage_vecs.map(v => v(stage_select))

      (stage_chosen, stage_outs, stage_valid)
    })
    io.chosen := carry_out._1
    io.out.bits := carry_out._2(0)
    io.out.valid := carry_out._3

  }
}
```

```
*/
class RRArbiterTree[T <: Data](data: T, tree_structure: Seq[Seq[Int]], int_q_
  require(int_q_depth > 0, "InterconnectNetwork.RRArbiterTree: Internal queue
  for(s <- 1 until tree_structure.length) {
    require(tree_structure(s-1).length == tree_structure(s).reduce(_+_), "Int
  }
  require(tree_structure(tree_structure.length-1).length ==1, "Interconnection

  val n_inputs = tree_structure(0).reduce(_+_)
  val io = new Bundle {
    val in = Vec.fill(n_inputs) {Decoupled(data)}.flip
    val out = Decoupled(data)
  }

  val arbiters: Seq[Seq[LockingArbiterLike[T]]] =
    tree_structure.map( _.map(i_cnt => Module(
      if(round_robin) (new RRArbiter(data, i_cnt)) else (new Arbiter(data, i_
    )) )

  // Connect input and internal stages
  for(s <- 0 until tree_structure.length) {
    // Get the component and component ports that we are wiring outputs up to
    val out_cpt = (tree_structure(s).zipWithIndex.map(Function.tupled((num, i
    val out_cpt_port = tree_structure(s).map(Vector.tabulate(_)(p=>p)).reduce

/*  // Debug code to demonstrate what those two computations produce
    println("Stage " + s)
    println(out_cpt.map(_.toString).reduce(_+" "+_))
    println(out_cpt_port.map(_.toString).reduce(_+" "+_))
*/

    // Now, actually connect this stage
    for(i <- 0 until out_cpt.length) {
      val stage_out = arbiters(s)(out_cpt(i)).io.in(out_cpt_port(i))
      if(s==0){ io.in(i) <> stage_out }
      else {
        stage_out <> Queue(arbiters(s-1)(i).io.out, int_q_depth)
      }
    }
  }

  // Connect output
```

```
  arbiters(arbiters.length-1)(0).io.out <> io.out
}

package FFT

import Chisel._
import rocket._
import uncore._

class LoadCmd(fc: FFTConfig) extends Bundle {
  val tablet = UInt(width=fc.tablet_id_width)
  val start_addr = UInt(width=fc.xprlen)
  val stride = UInt(width=fc.xprlen)
  override def clone = new LoadCmd(fc).asInstanceOf[this.type]
}

class LoadManagerIO(fc: FFTConfig) extends Bundle {
  val cmd = Decoupled(new LoadCmd(fc)).flip

  val localports = fc.LocalWritePorts
  val mem = new FlexMemIO(fc.flex_width, fc.flex_addr_width)

  val active_tablet = UInt(width=fc.tablet_id_width).asOutput
  //Everything but the memory
}
abstract class LoadManager extends Module {
  val io: LoadManagerIO
}

class SimpleLoadManager(val fc: FFTConfig) extends LoadManager {
  val max_tag = 8
  require(isPow2(max_tag))

  val io = new LoadManagerIO(fc)

  val s_idle :: s_issuing :: s_ext_draining :: s_int_draining :: Nil = Enum(U
  val state = Reg(init = s_idle)

  // Setup a Vec to associate response tags with target system location
  val tag_active  = Vec.fill(max_tag){Reg(init = Bool(false))}
  val tag_sysaddr = Vec.fill(max_tag){Reg(UInt(width=log2Up(fc.fft_size)))}
```

```
  val next_avail_tag = PriorityEncoder(tag_active.map(!_))
  val have_avail_tag = !tag_active.reduce(_&&_) // need at least 1 unused tag
  val using_no_tags  = !tag_active.reduce(_||_)

  // Execution trackers
  val active_tablet = Reg(init=UInt(fc.num_tablets))
  val req_sysaddr = Reg(init = UInt(0, width=log2Up(fc.fft_size)))
  val req_memaddr = Reg(init = UInt(0, width=fc.xprlen))
  val stride = Reg(init = UInt(4, width=fc.xprlen))
  val int_counter = Reg(init = UInt(0, width=log2Up(fc.load_drain_time)))

  Vector(io.cmd.ready, io.mem.req.valid).foreach( _ := Bool(false) )
    // Set some conservative defaults for the RV signals
  switch(state) {
    is( s_idle ) { // waiting for a command
      io.cmd.ready := Bool(true)
      when(io.cmd.fire()) {
        req_sysaddr := UInt(0)
        req_memaddr := io.cmd.bits.start_addr
        stride := io.cmd.bits.stride
        active_tablet := io.cmd.bits.tablet

        state := s_issuing
      }
    }

    is( s_issuing ) { // issuing load req to external system
      when( have_avail_tag ) {
        io.mem.req.valid := Bool(true)
        when( io.mem.req.ready ) { // request fired so
          // Setup the tag
          tag_active(next_avail_tag) := Bool(true)
          tag_sysaddr(next_avail_tag) := req_sysaddr
          // See if done
          when(req_sysaddr === UInt(fc.fft_size-1)) { state := s_ext_draining
          // Advance to next request
          req_sysaddr := req_sysaddr + UInt(1)
          req_memaddr := req_memaddr + stride
        }
      }
    }
```

```
    is( s_ext_draining ) { // finish getting responses from outside
      when( using_no_tags ) {
        state := s_int_draining
        int_counter := UInt(0)
      }
    }

    is( s_int_draining ) { // make sure things clear to inside local memories
      int_counter := int_counter + UInt(1)
      when(int_counter === UInt(fc.load_drain_time-1)) {
        state := s_idle
        active_tablet := UInt(fc.num_tablets)
      }
    }
  }

  // Register external memory responses
  io.mem.resp.ready := Bool(true) // Handle all responses immediately
  val reg_mem_resp_v = Reg(next=io.mem.resp.valid)
  val reg_mem_resp_d = Reg(next=io.mem.resp.bits)
  when( reg_mem_resp_v ) { tag_active(reg_mem_resp_d.tag) := Bool(false) }

  // Route responses to the appropriate internal ports
  val writer = Module(new WriteRouter(fc, fc.write_atu))
  writer.io.localports <> io.localports

  writer.io.in.valid := reg_mem_resp_v
  writer.io.in.bits.address := tag_sysaddr(reg_mem_resp_d.tag)
  writer.io.in.bits.payload := fc.unpack(reg_mem_resp_d.data)

  // Connect to memory request port
  io.mem.req.bits.addr  := req_memaddr
  io.mem.req.bits.tag   := next_avail_tag
  io.mem.req.bits.cmd   := MXRD // perform a load (MXWR for stores)
  io.mem.req.bits.bytes := UInt(2*fc.data_width/8) // ask for one point of da
  io.mem.req.bits.data  := Bits(0) // we're not performing any stores...

  io.active_tablet := active_tablet
}

package FFT
```

```
import Chisel._
import Node._
import fft_mcgen.InstructionPackage

class Machine(config: FFTConfig, insts: IndexedSeq[IndexedSeq[InstructionPack
  val io = new Bundle {
    val current_stage = Bits(width=log2Up(insts.length)).asInput
    val exec_pc = Valid(Bits(width=log2Up(insts(0).length))).asInput

    val ext_write = Vec.fill(config.radix){Valid(config.WriteReq())}.asInput

    val ext_raddr = Vec.fill(config.radix){config.ReadReq()}.asInput
    val ext_rdata = Vec.fill(config.radix){config.DataBits()}.asOutput

    val active_ld_tablet = UInt(width=config.tablet_id_width).asInput
    val active_ex_tablet = UInt(width=config.tablet_id_width).asInput
    val active_st_tablet = UInt(width=config.tablet_id_width).asInput

    // Internal Processing I/O
    val in  = Vec.fill(config.radix){Valid(config.WriteReq())}.asInput
    val out = Vec.fill(config.radix){Valid(config.FullWriteReq())}.asOutput

  }

  // Create memories and connect the write ports and external read port
  val data_memories = Vector.fill(config.num_tablets){
      Vector.fill(config.radix){SeqMem(config.dmem_size){config.DataBits()}}
    }
  (0 until config.radix).foreach(m => {
    io.ext_rdata(m) := data_memories(0)(m).io.rdata // default
    (0 until config.num_tablets).foreach(t => {
      data_memories(t)(m).io.write       := io.in(m) // default
      data_memories(t)(m).io.write.valid := Bool(false)
      data_memories(t)(m).io.raddr       := Bits(0)

      when(io.active_ld_tablet === UInt(t)) {
        data_memories(t)(m).io.write := io.ext_write(m)
        // raddr irrelevant
      }
      when(io.active_ex_tablet === UInt(t)) {
        data_memories(t)(m).io.write := io.in(m)
        // raddr handled below
```

```
      }
      when(io.active_st_tablet === UInt(t)) {
        // keep wport disabled
        data_memories(t)(m).io.raddr := io.ext_raddr(m)
        io.ext_rdata(m) := data_memories(t)(m).io.rdata
      }
    })
  })

  // Create TF ROM
  val tf_rom = Module(new TFRom(config.radix-1, config))

  // Instruction Fetch Stage
  val inst_rom = Module(new InstROM(config, insts, memory_ids))
  inst_rom.io.rstage:= io.current_stage
  inst_rom.io.raddr := io.exec_pc.bits

  val if_inst  = inst_rom.io.rinst
  val if_valid = Reg(next=io.exec_pc.valid, init=Bool(false))

  // Instruction Decode Stage
  val de_inst  = Reg(next=if_inst)
  val de_valid = Reg(next=if_valid, init=Bool(false))
  //read data
  val de_data = Vec.fill(config.radix){config.DataBits()}
  (0 until config.radix).foreach(n => {
    de_data(n) := data_memories(0)(n).io.rdata
    (0 until config.num_tablets).foreach(t => {
      when(io.active_ex_tablet === UInt(t)) {
        when(if_valid) {data_memories(t)(n).io.raddr := if_inst.raddr(n)}
        de_data(n) := data_memories(t)(n).io.rdata
      }
    })
  })
  //read twiddle factors
  (0 until config.radix-1).foreach(n => {
    tf_rom.io.raddr(n) := if_inst.twiddle(n)
  })
  val de_twiddles = tf_rom.io.rdata

  // Execute Stage
  val ex_inst = Reg(next=de_inst)
```

```
  val ex_data = Reg(next=de_data)
  val ex_valid = Reg(next=de_valid, init=Bool(false))

  val ex_data_ordered = Vec.fill(config.radix){config.DataBits()}
  (0 until config.radix).foreach(n => {ex_data_ordered(n) := ex_data(ex_inst.
  val ex_twiddles = Reg(next=de_twiddles)

  val ex_bf_unit = Module(new Butterfly(config.butterfly, config))
  (0 until config.radix).foreach(n => {ex_bf_unit.io.in.bits(n) := config.Dat
  ex_bf_unit.io.in.valid := ex_valid
  ex_bf_unit.io.twiddles := ex_twiddles

  // Writeback Stage
  val wb_meta_delay = Module(new TimeMultiplexer.StaticDelayblockSeries(confi
  wb_meta_delay.io.in.valid := ex_valid
  wb_meta_delay.io.in.bits  := ex_inst

  val wb_valid = ex_bf_unit.io.out.valid
  val wb_inst = wb_meta_delay.io.out.bits

  val wb_data = ex_bf_unit.io.out.bits.map(_.toBits)

  (0 until config.radix).foreach(n => {
    io.out(n).valid := wb_valid
    io.out(n).bits.address := wb_inst.wid(n)
    io.out(n).bits.payload.address := wb_inst.waddr(n)
    io.out(n).bits.payload.payload := wb_data(n)
  })
}

package FFT

import Chisel._
import Node._

class SeqMem[T <: Data](depth: Int)(dtype: T) extends Module {
  val addr_width = log2Up(depth)

  override val io = new Bundle {
    val write  = Valid(Packet(addr_width)(dtype.clone)).asInput

    val raddr = Bits(INPUT, addr_width)
```

```
    val rdata = dtype.clone.asOutput
  }

  val memory  = Mem(Bits(width=dtype.toBits.getWidth), depth, true)

  // Write port
  when ( io.write.valid ) { memory(io.write.bits.address) := io.write.bits.pa

  // Read port
  val reg_raddr = Reg(Bits(width=addr_width))
  reg_raddr := io.raddr
  io.rdata := dtype.fromBits(memory(reg_raddr))
}
object SeqMem { def apply[T <: Data](depth: Int)(dtype: T) = Module(new SeqMe
```

```
package FFT

import Chisel._

class OpCmd(fc: FFTConfig) extends Bundle {
  val base_address = UInt(width=fc.xprlen)
  val stride       = UInt(width=fc.xprlen)
  override def clone = new OpCmd(fc).asInstanceOf[this.type]
}

class OpController(val fc: FFTConfig, val tablet: Int) extends Module {
  val io = new Bundle {
    val cmd = Decoupled(new OpCmd(fc)).flip

    val ld_cmd = Decoupled(new LoadCmd(fc))
    val ex_cmd = Decoupled(new ExecCmd(fc))
    val st_cmd = Decoupled(new StoreCmd(fc))

    val ld_busy = Bool().asInput
    val ex_busy = Bool().asInput
    val st_busy = Bool().asInput

    val busy = Bool().asOutput
  }

  val s_idle :: s_preload :: s_loading :: s_preexec :: s_executing :: s_prest
    Enum(UInt(), 7)
```

```
val state = Reg(init = s_idle)
// Trackers
val base_address = Reg(init=UInt(0,width=fc.xprlen))
val stride       = Reg(init=UInt(0,width=fc.xprlen))

// Command data setup
io.ld_cmd.bits.tablet := UInt(tablet)
io.ld_cmd.bits.start_addr := base_address
io.ld_cmd.bits.stride := stride
io.ex_cmd.bits.tablet := UInt(tablet)
io.st_cmd.bits.tablet := UInt(tablet)
io.st_cmd.bits.start_addr := base_address
io.st_cmd.bits.stride := stride

Vector(io.cmd.ready, io.ld_cmd.valid, io.ex_cmd.valid, io.st_cmd.valid).for
  // Defaults for some of the control logic

switch( state ) {
  is( s_idle ) {
    io.cmd.ready := Bool(true)
    when( io.cmd.fire() ) {
      base_address := io.cmd.bits.base_address
      stride := io.cmd.bits.stride
      state := s_preload
    }
  }

  is( s_preload ) {
    io.ld_cmd.valid := Bool(true)
    when( io.ld_cmd.fire() ) { state := s_loading }
  }

  is( s_loading ) {
    when( !io.ld_busy ) { state := s_preexec }
    //when( !io.ld_busy ) { state := s_prestore }
  }

  is( s_preexec ) {
    io.ex_cmd.valid := Bool(true)
    when( io.ex_cmd.fire() ) { state := s_executing }
  }
```

```
    is( s_executing ) {
      when( !io.ex_busy ) { state := s_prestore }
    }

    is( s_prestore ) {
      io.st_cmd.valid := Bool(true)
      when( io.st_cmd.fire() ) {state := s_storing}
    }

    is( s_storing ) {
      when( !io.st_busy ) { state := s_idle }
    }
  }

  io.busy := state != s_idle

}

package FFT

import Chisel._
import Node._

// Address Translation Unit
class ReadATU(config: FFTConfig, atlut: IndexedSeq[(Int,Int)]) extends Module
  val io = new Bundle {
    val in  = Decoupled(config.FFTReadReq()).flip
    val out = Decoupled(config.FullReadReq())
  }
  // setup ROMs
  val rom_id  = Vec.fill(config.fft_size){Bits(width=log2Up(config.num_dmem))
  val rom_addr= Vec.fill(config.fft_size){Bits(width=config.dmem_addr_width)}
  (0 until config.fft_size).foreach(n => {
    rom_id(n)   := Bits(atlut(n)._1)
    rom_addr(n) := Bits(atlut(n)._2)
  })

  val stall = Bool()
  def StallableReg[T<:Data](nval: T, ival: T = null) = {
    val myreg = Reg(nval.clone: T, null.asInstanceOf[T], init = ival: T)
    when(!stall) { myreg := nval }
```

```
    myreg
  }

  // Stage 1 translates request
  val s1_valid   = StallableReg(io.in.valid, Bool(false))
  val s1_request = StallableReg(io.in.bits)

  // Stage 2 sends out the request
  val s2_valid = StallableReg(s1_valid, Bool(false))
  val s2_id    = StallableReg(rom_id(s1_request))
  val s2_addr  = StallableReg(rom_addr(s1_request))

  io.out.valid := s2_valid
  io.out.bits.address := s2_id
  io.out.bits.payload := s2_addr

  stall := !io.out.ready
  io.in.ready := io.out.ready
}

class ReadRouter(config: FFTConfig, atlut: IndexedSeq[(Int,Int)]) extends Mo
  val n_agents = config.num_dmem
  val addr_width = log2Up(n_agents)
  val io = new Bundle {
    val in = Decoupled(config.FFTReadReq()).flip

    val localports = config.LocalReadPorts

    val out = Decoupled(config.DataBits())
  }
  // Helper logic for handling stalls properly
  val stall = Bool()
  def StallableReg[T<:Data](nval: T, ival: T = null) = {
    val myreg = Reg(nval.clone: T, null.asInstanceOf[T], init = ival: T)
    when(!stall) { myreg := nval }
    myreg
  }

  // Make the ATU
  val atu = Module(new ReadATU(config, atlut))
  atu.io.in <> io.in
```

```
  // Stage 3 selects the right request and sends out response
  val s3_valid = StallableReg(atu.io.out.valid, Bool(false))
  val s3_id    = StallableReg(atu.io.out.bits.address)
  val s3_addr  = StallableReg(atu.io.out.bits.payload)
    // for replay during a stall
    // TODO: Can probably remove this by moving replay into machine...
  val s3_data = (0 until n_agents).foldLeft(io.localports(0).resp)((carrying,
    Mux(s3_id === Bits(aid), io.localports(aid).resp, carrying)
  })

  // Send the read request
  for(a <- 0 until n_agents) {
    io.localports(a).req := Mux(stall, s3_addr, atu.io.out.bits.payload)
    // replay when stalled
  }
  io.out.valid := s3_valid
  io.out.bits := s3_data

  stall := !io.out.ready
  atu.io.out.ready := io.out.ready
}

package FFT

import Chisel._
import Node._

class Router[T <: Data](n_inputs: Int, n_outputs: Int)(dtype: T) extends Modu
  val addr_width = log2Up(n_outputs)
  val io = new Bundle {
    val in  = (Vec.fill(n_inputs){Valid(Packet(addr_width){dtype})}).asInput
    val out = (Vec.fill(n_outputs){Valid(dtype)}).asOutput
  }

  for(o <- 0 until n_outputs) {
    val out = io.out(o)
    out.valid := Bool(false) // setup default case
    out.bits  := io.in(0).bits.payload
    for(i <- 0 until n_inputs) {
      val in = io.in(i)
      when(in.valid && (in.bits.address === Bits(o))) {
        out.valid := in.valid
```

```
        out.bits  := in.bits.payload
      }
    }
  }
}

package FFT

import Chisel._
import rocket._
import uncore._

class StoreCmd(fc: FFTConfig) extends Bundle {
  val tablet = UInt(width=fc.tablet_id_width)
  val start_addr = UInt(width=fc.xprlen)
  val stride = UInt(width=fc.xprlen)
  override def clone = new StoreCmd(fc).asInstanceOf[this.type]
}

class StoreManagerIO(fc: FFTConfig) extends Bundle {
  val cmd = Decoupled(new LoadCmd(fc)).flip

  val localports = fc.LocalReadPorts
  val mem = new FlexMemIO(fc.flex_width, fc.flex_addr_width)

  val active_tablet = UInt(width=fc.tablet_id_width).asOutput
}

abstract class StoreManager extends Module {
  val io: StoreManagerIO
}

class SimpleStoreManager(val fc: FFTConfig) extends StoreManager {
  val max_tag = 8 // CONFIGURATION
  require(isPow2(max_tag))

  val io = new StoreManagerIO(fc)

  val s_idle :: s_polling :: s_draining :: Nil = Enum(UInt(), 3)
  // s_polling means asking local for data and also issuing some data to mem
  // s_draining means issuing last data to mem and waiting for acks
  val state = Reg(init = s_idle)
```

```
val memaddr_q = Module(new Queue(UInt(width=fc.xprlen), 4))
class sm_req() extends Bundle {
  val sysaddr = fc.FFTReadReq()
  val memaddr = UInt(width=fc.xprlen)
  override def clone = new sm_req().asInstanceOf[this.type]
}
val req_splitter = Module(new RV_Split(new sm_req(), 2))
val resp_merger  = Module(new RV_Merge(fc.DataBits(), UInt(width=fc.xprlen)

// Execution trackers
val active_tablet = Reg(init=UInt(fc.num_tablets))
val req_sysaddr = Reg(init = UInt(0, width=log2Up(fc.fft_size)))
val req_memaddr = Reg(init = UInt(0, width=fc.xprlen))
val rcvd_acks   = Reg(init = UInt(0, width=log2Up(fc.fft_size+1)))
val stride = Reg(init = UInt(4, width=fc.xprlen))

Vector(io.cmd.ready, req_splitter.io.in.valid).foreach( _ := Bool(false) )

switch(state) {
  is( s_idle ) {
    io.cmd.ready := Bool(true)
    when(io.cmd.fire()) {
      req_sysaddr := UInt(0)
      req_memaddr := io.cmd.bits.start_addr
      rcvd_acks  := UInt(0)
      stride := io.cmd.bits.stride
      active_tablet := io.cmd.bits.tablet

      state := s_polling
    }
  }

  is( s_polling ) {
    req_splitter.io.in.valid := Bool(true)
    when(req_splitter.io.in.fire()) {
      req_sysaddr := req_sysaddr + UInt(1)
      req_memaddr := req_memaddr + stride
      when(req_sysaddr === UInt(fc.fft_size-1)) { state := s_draining }
    }
  }
```

```
    is( s_draining ) {
      when(rcvd_acks === UInt(fc.fft_size)) {
        state := s_idle
        active_tablet := UInt(fc.num_tablets)
      }
    }
  }

  // Connect request splitter input
  req_splitter.io.in.bits.sysaddr := req_sysaddr
  req_splitter.io.in.bits.memaddr := req_memaddr

  // Create and connect read requests and response router
  val reader = Module(new ReadRouter(fc, fc.read_atu))
  reader.io.localports <> io.localports

  // Connect request splitter outputs to local requests port and memaddr queu
  req_splitter.io.out(0).ready := reader.io.in.ready
  reader.io.in.valid := req_splitter.io.out(0).valid
  reader.io.in.bits := req_splitter.io.out(0).bits.sysaddr

  req_splitter.io.out(1).ready := memaddr_q.io.enq.ready
  memaddr_q.io.enq.valid := req_splitter.io.out(1).valid
  memaddr_q.io.enq.bits := req_splitter.io.out(1).bits.memaddr

  // connect memaddr queue and local response port to response merger
  resp_merger.io.in1 <> reader.io.out
  resp_merger.io.in2 <> memaddr_q.io.deq

  // connect unified response to memory req port (write)
  io.mem.req.bits.addr := resp_merger.io.out.bits.data2
  io.mem.req.bits.tag := UInt(0) // do not care about tag
  io.mem.req.bits.cmd := MXWR // perform a store
  io.mem.req.bits.bytes := UInt(2*fc.data_width/8) // ask for one point of da
  io.mem.req.bits.data := fc.pack(resp_merger.io.out.bits.data1)

  io.mem.req.valid := resp_merger.io.out.valid
  resp_merger.io.out.ready := io.mem.req.ready

  // Handle memory responses by incrementing counter
  io.mem.resp.ready := Bool(true)
  when( io.mem.resp.fire() ) {
```

```
    rcvd_acks := rcvd_acks + UInt(1)
  }

  io.active_tablet := active_tablet
}

package InterconnectNetwork {

import Chisel._
import Chisel.AdvTester._

class DecoupledInterconnectTest[T <: Bits](dut: DecoupledInterconnect[T], in_
  extends AdvTester(dut)
{
  // Helpful data structures and methods for them
  case class TestPacket(address: BigInt, payload: TestData) {
    def inject(db: AdvTester[_], target: Packet[T]): Unit = {
      db.reg_poke(target.address, address)
      payload.inject(db, target.payload)
    }
  }
  case class TestData(id: BigInt, src: BigInt, dest: BigInt) {
    def inject(db: AdvTester[_], target: T): Unit = {
      val packed = (dest % dut.n_outputs) + dut.n_outputs*((src % dut.n_input
      db.reg_poke(target, packed)
    }
  }
  object TestData {
    def extract(db: AdvTester[_], source: T): TestData = ({
      val payload = db.peek(source)

      val dest = payload % dut.n_outputs
      val payload2 = (payload-dest)/dut.n_outputs
      val src = payload2 % dut.n_inputs
      val id = (payload2-src)/dut.n_inputs

      TestData(id, src, dest)
    })
  }

  // Setup handlers for all the I/Os
  val in_sockets  = Vector.tabulate(dut.n_inputs)(i =>
```

```
    new DecoupledSource(dut.io.in(i),  (sckt: Packet[T], in: TestPacket) => i
  )
  val out_sockets = Vector.tabulate(dut.n_outputs)(o =>
    new DecoupledSink(  dut.io.out(o), (sckt: T) => TestData.extract(this, sc
  )
  out_sockets.foreach(os => os.max_count = 2)

  // SETUP AND RUN EXPERIMENT
  val exp_slow = exp_stop-50// Stop generating messages at input and always r

  var phantoms = 0
  var misroutes = 0
  val processed = Array.fill(dut.n_inputs)(0)
  val max_time  = Array.fill(dut.n_inputs)(0)
  val sum_time  = Array.fill(dut.n_inputs)(0)

  val rand = new scala.util.Random()

  println("Testing "+ dut.getClass.getName +" with %d inputs and %d outputs".

  val message_timelog = new scala.collection.mutable.HashMap[BigInt, Int]()
  var next_id = 0 // Next valid id

  until(cycles >= exp_stop, maxCycles=exp_stop+1){ // Should never trigger ti
    if(verbose || (cycles%1000==0)) { println("Processing cycle %d".format(cy

    for(i <- 0 until dut.n_inputs
      if (in_sockets(i).inputs.length == 0) && (cycles < exp_slow) && (rand.n
    ) {
      val dest = rand.nextInt(dut.n_outputs)
      val id = next_id; next_id += 1
      val packet = TestPacket(dest, TestData(id, i, dest))
      message_timelog(BigInt(id)) = cycles
      in_sockets(i).inputs.enqueue(packet)
    }

    for(o <- 0 until dut.n_outputs
      if (out_sockets(o).outputs.length > 0) && (rand.nextInt(100) < out_read
    ) {
      val data: TestData = out_sockets(o).outputs.dequeue

      if(data.dest.toInt != o) {
```

```
        println(s"!!! MISROUTE !!! -> Output $o received data intended for ${
        misroutes += 1
      } else if(message_timelog.isDefinedAt(data.id)) {
        val travel_time = cycles - message_timelog(data.id)
        message_timelog -= data.id

        if(verbose) {println(s"Output $o received (${data.id},${data.src},${d

        // Update statistics
        val src = data.src.toInt
        processed(src) += 1
        max_time(src) = math.max(max_time(src), travel_time)
        sum_time(src) += travel_time

      } else {
        println(s"!!! PHANTOM !!! -> Output $o received (${data.id},${data.s
        phantoms += 1
      }
    }
  }

  println("Experiment END")
  val lost = message_timelog.keys
  val lost_length = lost.count(_=>true)
  if(lost_length > 0) {println("")}
  lost.foreach(p =>
    println("Lost Message: %d sent during cycle %d".format(p,message_timelog(
  )

  println("")
  println("Final Statistics:")
  println("Experiment ran for %d cycles".format(exp_stop))
  (0 until dut.n_inputs).foreach(
    i => println("Input %d sent %d messages needing %d mean cycles and %d max
      i, processed(i), (if(processed(i)>0)sum_time(i)/processed(i) else 0), n
      )
    )
  )
  println("Failure Statistics:")
  println("Phantoms: %d".format(phantoms))
  println("Misroutes: %d".format(misroutes))
  println("Lost Messages: %d".format(lost_length))
```

```
/*
// Warmup
def warmup() = {
  val ivars = new HashMap[Node, Node]()

  for(i <- 0 until dut.n_inputs)  { ivars(dut.io.in(i).valid) = Bool(false)
  for(o <- 0 until dut.n_outputs) { ivars(dut.io.out(o).ready) = Bool(true)
  for(n <- 0 until 2)             { step(ivars, ovars, false) }
  ovars
}
warmup()

val posting = Array.tabulate(dut.n_inputs)(i => pack_payload(0, i, rand.nex
val processed = Array.fill(dut.n_inputs)(0)
val max_time = Array.fill(dut.n_inputs)(0)
val sum_time = Array.fill(dut.n_inputs)(0)

val message_timelog = new HashMap[Int, Int]()

for(n <- 0 until exp_stop) {
  ivars.clear()

  if(verbose || (n%1000==0)) println("Processing cycle %d".format(n))
  // Setup input data
  for(i <- 0 until dut.n_inputs) {
    ivars(dut.io.in(i).valid) = Bool((posting(i) >= 0) && (rand.nextInt(100
    ivars(dut.io.in(i).bits.address) = Bits(posting(i)%dut.n_outputs,1)
    ivars(dut.io.in(i).bits.payload) = Bits(posting(i),1)
  }

  // Establish if testbench is ready to receive data
  for(o <- 0 until dut.n_outputs) {
    ivars(dut.io.out(o).ready) = Bool((rand.nextInt(100) < out_ready_freq)
  }

  step(ivars, ovars, false)

  val was_sent = Array.tabulate(dut.n_inputs)(i =>
    (ovars(dut.io.in(i).ready).litValue() == 1) && (ivars(dut.io.in(i).vali
    ) // Establish whether data value was read

  // Update posting as needed:
```

```
    for(i <- 0 until dut.n_inputs if was_sent(i)) {
      val opayload = posting(i)
      val (oid, osrc, odest) = unpack_payload(opayload)
      message_timelog(opayload) = n
      if(verbose) println("Input " + i + " sent data: " + opayload + " (%d,%d

      posting(i) = if(n < exp_slow) pack_payload(oid+1 , i, rand.nextInt(dut.
    }

    // Check to see if data received at output, process accordingly
    for(o <- 0 until dut.n_outputs) {
      if(ovars(dut.io.out(o).valid).litValue() == 1 && ivars(dut.io.out(o).re
        val payload = ovars(dut.io.out(o).bits).litValue().toInt
        val (id, src, dest) = unpack_payload(payload)

        val send_time = message_timelog.getOrElse(payload, -1)
        if(send_time < 0) {
          println("!!! PHANTOM !!! ->  Output %d received %d (%d,%d,%d)".form
            o,payload,id,src,dest)
            )
          phantoms += 1
        }
        else {
          message_timelog -= payload
          val travel_time = n - send_time

          if(verbose) println("Output " + o + " received data: " + payload +
            + " after " + travel_time + " cycles")
          processed(src) += 1
          max_time(src) = math.max(max_time(src), travel_time)
          sum_time(src) += travel_time
        }

        if(dest != o) {
          println("!!! MISROUTE !!! -> Output %d received data intended for %
          misroutes += 1
        }
      }
    }
  } // main for loop
  println("Experiment END")
  val lost = message_timelog.keys
```

```
    val lost_length = lost.count(_=>true)
    if(lost_length > 0) {println("")}
    lost.foreach(p =>
      println("Lost Message: %d sent during cycle %d".format(p,message_timelog(
      )

    println("")
    println("Final Statistics:")
    println("Experiment ran for %d cycles".format(exp_stop))
    (0 until dut.n_inputs).foreach(
      i => println("Input %d sent %d messages needing %d mean cycles and %d max
        i, processed(i), (if(processed(i)>0)sum_time(i)/processed(i) else 0), m
        )
      )
    )
    println("Failure Statistics:")
    println("Phantoms: %d".format(phantoms))
    println("Misroutes: %d".format(misroutes))
    println("Lost Messages: %d".format(lost_length))

    //(phantoms == 0) && (misroutes == 0) && (lost_length == 0) // Determine if
    */

} // class

}
```

```
package FFT

import Chisel._
import Chisel.AdvTester._
import Node._
import rocket._
import uncore._
import scala.util.Random
import scala.collection.mutable.Queue

case class TestInst(funct: BigInt, rd: BigInt, rs1: BigInt, rs2: BigInt,
                                    xd: Boolean, xs1: Boolean, xs2: Boolean,
                    opcode: BigInt=Instructions.CUSTOM0.litValue().toInt) {
  def inject(db: AdvTester[_], target: RoCCInstruction): Unit = {
    db.reg_poke(target.funct, funct)
```

```
    db.reg_poke(target.rd,  rd ); db.reg_poke(target.xd,  if(xd)  1 else 0)
    db.reg_poke(target.rs1, rs1); db.reg_poke(target.xs1, if(xs1) 1 else 0)
    db.reg_poke(target.rs2, rs2); db.reg_poke(target.xs2, if(xs2) 1 else 0)
    db.reg_poke(target.opcode, opcode)
  }
}
case class TestCmd(inst: TestInst, rs1: BigInt=0, rs2: BigInt=0) {
  def inject(db: AdvTester[_], target: RoCCCommand): Unit = {
    inst.inject(db, target.inst)
    db.reg_poke(target.rs1, rs1)
    db.reg_poke(target.rs2, rs2)
  }
}
// This factory object provides a convenient way to construct instructions
object TestCmd {
  def execute(addr: Int, stride: Int=8) = (
    TestCmd(TestInst(BigInt(0), 0, 2, 3, false, true, true), addr, stride)
  )
}

case class TestResp(rd: BigInt, data: BigInt)
object TestResp {
  def extract(db: AdvTester[_], source: RoCCResponse) = ({
    TestResp(db.peek(source.rd), db.peek(source.data))
  })
}

case class TestMemReq(addr: BigInt, cmd: BigInt, data: BigInt, bytes: BigInt,
object TestSmallMemReq {
  def extract(db: AdvTester[_], source: HellaCacheReq): TestMemReq = ({
    val memtype: BigInt = db.peek(source.typ)
    assert( memtype == MT_D.litValue().toInt, "MemError: only supported type
    TestMemReq( db.peek(source.addr), db.peek(source.cmd),      db.peek(source
                source.data.getWidth/8,  db.peek(source.phys)==1, db.peek(sou
  })
}
object TestFlexMemReq {
  def extract(db: AdvTester[_], source: FlexMemReq): TestMemReq = ({
    TestMemReq( db.peek(source.addr), db.peek(source.cmd),      db.peek(source
                db.peek(source.bytes),  db.peek(source.phys)==1, db.peek(sour
  })
}
```

```
case class TestMemResp(data: BigInt, tag: BigInt, cmd: BigInt) {
  def inject(db: AdvTester[_], target: HellaCacheResp): Unit = {
    db.reg_poke(target.data, data)
    db.reg_poke(target.tag, tag)
    db.reg_poke(target.cmd, cmd)
  }
  def inject(db: AdvTester[_], target: FlexMemResp): Unit = {
    db.reg_poke(target.data, data)
    db.reg_poke(target.tag, tag)
    db.reg_poke(target.cmd, cmd)
  }
}

class MemPort(val req_q: Queue[TestMemReq], val resp_q: Queue[TestMemResp])
object MemPort {
  def apply(req_q: Queue[TestMemReq], resp_q: Queue[TestMemResp]) =
    new MemPort(req_q, resp_q)
}

abstract class FFTTester(dut: FFT) extends AdvTester(dut) {
  import math._
  val collateOutput = true

  // Setup the IO handlers
  val Cmd_IHandler     = new DecoupledSource(dut.io.cmd, (sckt: RoCCCommand,
  val Resp_OHandler    = new DecoupledSink(dut.io.resp, (sckt: RoCCResponse)
  val MemReq_OHandler  = new DecoupledSink(dut.io.mem.req, (sckt: HellaCacheR
  val MemResp_IHandler = new ValidSource(dut.io.mem.resp, (sckt: HellaCacheRe

  val FlexMemReq_OHandler  = new DecoupledSink(dut.io.flexmem.req, (sckt: Flex
  val FlexMemResp_IHandler = new DecoupledSource(dut.io.flexmem.resp, (sckt:

  def dutBusy = (peek(dut.io.busy) == 1)

  // Some helpful functions for generating and displaying test data
  val rand = new Random()
  def gen_input(n: Int): Double = {
    val freq1 = 2*Pi*1/dut.config.fft_size
    val freq2 = 2*Pi*2/dut.config.fft_size
//    1.0
//    0.75 + cos(freq1*n) + sin(freq2*n)
```

```
//     cos(freq1*n)
    rand.nextGaussian()
  }
  def create_time_data(gfunc: Int => Double): Tuple2[Vector[Double],Vector[Bi
    def forceUnsignedBits(i: BigInt) = if(i<0) i + (BigInt(1) << dut.config.d
    def floatToBits(i: Double) = {
      val o = BigInt(java.lang.Float.floatToIntBits(i.toFloat))
      if(o < 0) o + (BigInt(1)<<32) else o
    }

    if(dut.config.floating_point) {
      val (input_data, input_prepped) = new FFT_TV_Generator(dut.config).gene
      (input_data, input_prepped.map(i => forceUnsignedBits(floatToBits(i)) )
    } else {
      val (input_data, input_prepped) = new FFT_TV_Generator(dut.config).gene
      (input_data, input_prepped.map(i => forceUnsignedBits(i) ))
    }
  }
  def print_vector(title: String, real_data: IndexedSeq[Double], imag_data: I
    println(title)
    (0 until real_data.length).toVector.map(n =>
      ("%4d: (%7.4f, %7.4f)".format(n, real_data(n), imag_data(n)))
    ).foreach(println(_))
  }
  def print_real_vector(title: String, real_data: IndexedSeq[Double]) = print
  def convert_from_mem(dword: BigInt): Tuple2[Double,Double] = {

    def read_fixed_value(o: BigInt) = {
      val scaled = o.toDouble/pow(2, dut.config.fix_pt)
      val max_pos_val = pow(2, 32-1-dut.config.fix_pt)-1
      val make_negative = pow(2, 32-dut.config.fix_pt)
      (if(scaled > max_pos_val) scaled-make_negative else scaled)//dut.config
    }

    def read_float_value(o: BigInt) = java.lang.Float.intBitsToFloat(o.toInt)

    val imag_part = dword >> 32;
    val real_part = dword & ((BigInt(1) << 32)-1)

    if(dut.config.floating_point) (read_float_value(real_part), read_float_va
    else                          (read_fixed_value(real_part), read_fixed_va
  }
```

```
  def square_error(a: Tuple2[Double,Double], b: Tuple2[Double,Double]): Doubl
    def square(x: Double) = x*x
    square(a._1 - b._1) + square(a._2 - b._2)
  })
  def max_square_error(a: IndexedSeq[Tuple2[Double,Double]], b: IndexedSeq[Tu
    a.zip(b).map(Function.tupled((a,b)=>square_error(a,b))).reduce(math.max(_
  )

  // Setup the memory models
  protected def memory_size: Int
  lazy val memory: SimMemory = new InstantMemory(64, memory_size, Vector(
    MemPort(MemReq_OHandler.outputs, MemResp_IHandler.inputs),
    MemPort(FlexMemReq_OHandler.outputs, FlexMemResp_IHandler.inputs)
  ))
  // lazy val so that memory_size can be appropriately constructed
  //   not foolproof but makes subclass constructor clearer
  MemReq_OHandler.max_count = 2
  FlexMemReq_OHandler.max_count = 2
}

abstract class SimMemory(word_width: Int, depth: Int) extends AdvTester.Proce
  // LITTLE ENDIAN
  private val bytes_in_word = word_width/8
  require(word_width == 64, "MemError: Memory must have 64 bit words (for now
  val memory = Array.fill(depth)(BigInt(0)) //need mutability here

  val trans_shift = log2Up(word_width)-3 // must shift addr by this much
  val max_addr = depth*(word_width>>3) // exclusive

  private val word_mask = (BigInt(1) << word_width) - 1
  def store_addr(addr: Int, data: BigInt, words: Int): Unit = {
    val subdata = (0 until words).map(n => {
      (data >> (n*word_width)) & word_mask
    }) // little endian: take highest word from higher bit positions
    val addresses = (0 until words).map(n => addr + n*bytes_in_word)
    (addresses zip subdata).foreach({case (a, d) => single_store_addr(a, d)})
  }
  def load_addr(addr: Int, words: Int): BigInt = {
    val addresses = (0 until words).map(n => addr + n*bytes_in_word)
    val subdata = addresses.map(a => single_load_addr(a))
```

```
    val data = subdata.reverse.foldLeft(BigInt(0))((acc, d) => {
      (acc << word_width) | (d & word_mask)
    }) // little endian: reverse so higher words get placed in higher positio

    data
  }

  def single_store_addr(addr: Int, data: BigInt): Unit = {memory(addr >> tran
  def single_load_addr(addr: Int): BigInt = memory(addr >> trans_shift)
  private def single_check_addr(addr: Int): Boolean = {
    assert( addr >= 0 && (addr < max_addr), "MemError: Address %x out of memo
    assert( (addr & ((word_width>>3)-1))==0, "MemError: Misaligned address %x
    true
  }

  def store_data(startaddr: Int, newdata: IndexedSeq[BigInt]) = {
    (0 until newdata.length).foreach(idx => single_store_addr(startaddr+idx*8
  }

  def handle_request(req: TestMemReq): TestMemResp = {
    //check_addr(req.addr.toInt)
    assert( req.phys, "MemError: can only handle physical addresses (field: p
    val words = req.bytes.toInt/bytes_in_word
    assert( words*bytes_in_word == req.bytes.toInt,
      s"MemErorr: #bytes requested improper: ${req.bytes.toString}")

    if(req.cmd == MXRD.litValue().toInt) {
      TestMemResp(data=load_addr(req.addr.toInt, words), tag=req.tag, cmd=req
    } else if(req.cmd == MXWR.litValue().toInt) {
      store_addr(req.addr.toInt, req.data, words)
      TestMemResp(data=BigInt(0),                    tag=req.tag, cmd=req.cmd)
    } else {
      assert(false, "MemError: Invalid Command: Only supported cmds are MXRD
      null
    }
  }
}


class InstantMemory(word_width: Int, depth: Int, ports: Seq[MemPort]) extends
  def process() = {
    ports.foreach(port => {
```

```
      if(!port.req_q.isEmpty) {
        port.resp_q.enqueue(handle_request(port.req_q.dequeue()))
      }
    })
  }
}

class SlowMemory(word_width: Int, depth: Int, ports: Seq[MemPort]) extends Si
  val max_wait = 32
  val schedule_db = ports.map(port => (port, Vector.fill(max_wait)(new Queue[
  var cur_time = 0

  def process() = {
    ports.foreach( port => {
      val schedule = schedule_db(port)
      if(!port.req_q.isEmpty) {
        schedule((cur_time+max_wait-1) % max_wait).enqueue(handle_request(por
      }

      while(!schedule(cur_time).isEmpty) { port.resp_q.enqueue(schedule(cur_t
    })

    cur_time = (cur_time + 1) % max_wait
  }
}

class RandomMemory(word_width: Int, depth: Int, ports: Seq[MemPort]) extends
  val rand = new Random()
  def check_prob(prob: Int) : Boolean = (math.abs(rand.nextInt()) % 100) <= p
  def random_range(start: Int, end: Int) : Int = (math.abs(rand.nextInt()) %

  val max_wait = 16
  val schedule_db = ports.map(port => (port, Vector.fill(max_wait)(new Queue[
  var cur_time = 0

  def process() = {
    ports.foreach( port => {
      val schedule = schedule_db(port)
      if(!port.req_q.isEmpty && check_prob(66)) {
        schedule(random_range(0, max_wait)).enqueue(handle_request(port.req_q
      }
```

```
      while(!schedule(cur_time).isEmpty) { port.resp_q.enqueue(schedule(cur_t
    })

    cur_time = (cur_time + 1) % max_wait
  }
}

package FFT

import Chisel._
import Chisel.AdvTester._
import Node._
import rocket._
import uncore._
import scala.util.Random
import scala.collection.mutable.Queue

class MultiFFTTester(dut: FFT) extends FFTTester(dut) {
  import math._

  def nffts = 8
  def memory_size = 1024 + (nffts+1)*(dut.config.fft_size+16)

  // START ACTUAL TESTS NOW
  takestep()
  takestep()
  takestep()

  println("Initializing memory with time data.")
  val baseaddrs = (0 until nffts).toVector.map(_*(dut.config.fft_size+16)*8)
  val fft_datas = Vector.fill(nffts)(create_time_data(gen_input))
    // (input_data, input_prepped)
  val input_datas = fft_datas.map(_._1)
  val input_preps = fft_datas.map(_._2)
  (0 until nffts).foreach(n => {
    memory.store_data(baseaddrs(n), input_preps(n))
  })

  println("Starting FFT")
  (0 until nffts).foreach(n => { Cmd_IHandler.inputs.enqueue(TestCmd.execute(
  val start_cycle = cycles
  until( Cmd_IHandler.isIdle && !dutBusy, dut.config.fft_size*dut.config.fft_
```

```
  println(s"${nffts} FFTs finished after ${cycles-start_cycle} cycles")
/*
  val out_datas = (0 until nffts).toVector.map(n => {
    (0 until dut.config.fft_size).toVector.map(idx => ({
      val memline = memory.load_addr(baseaddrs(n) + idx*8)
      convert_from_mem(memline)
    }))
  })

  val expected_output_is_unscaled = true
  val test_dfts = (0 until nffts).toVector.map(n => {
    FFT_TV_Generator.dft(input_datas(n)).map( dp =>
      if(expected_output_is_unscaled) (dp._1*dut.config.fft_size,dp._2*dut.co
      else dp
    )
  })

  val test_max_sq_errors = (0 until nffts).toVector.map(n => {
    max_square_error(out_datas(n), test_dfts(n))
  })
*/
  val expected_output_is_unscaled = true

  val test_max_sq_errors = (0 until nffts).toVector.map(n => {
    val out_data = (0 until dut.config.fft_size).toVector.map(idx => ({
      val memline = memory.single_load_addr(baseaddrs(n) + idx*8)
      convert_from_mem(memline)
    }))
    val test_dft = FFT_TV_Generator.dft(input_datas(n)).map( dp =>
      if(expected_output_is_unscaled) (dp._1*dut.config.fft_size,dp._2*dut.
      else dp
    )
  /*  println("%5s %18s %18s %18s".format("ID", "INPUT        ", "OUTPUT
", "EXP. OUTPUT    "))
    (0 until dut.config.fft_size).map(l => {
      "%4d: (%7.4f, %7.4f) (%7.4f, %7.4f) (%7.4f, %7.4f)".format(
        l, input_datas(n)(l), 0.0,
           out_data(l)._1, out_data(l)._2,
           test_dft(l)._1, test_dft(l)._2
      )
    }).foreach(println(_))
  */
```

```
    max_square_error(out_data, test_dft)
  })
  (0 until nffts).foreach(n => println("Maximum Square Error for FFT %d: %g".
  val allmax = test_max_sq_errors.reduce(max(_,_))
  println("ALL FFT Maximum Square Error: %g".format(allmax))
  ok = (allmax < pow(10,-6)) && pass
}

package FFT

import Chisel._
import Chisel.AdvTester._
import Node._
import rocket._
import uncore._
import scala.util.Random
import scala.collection.mutable.Queue

class SingleFFTTester(dut: FFT) extends FFTTester(dut) {
  import math._

  def memory_size = 1024 + 2*dut.config.fft_size

  // START ACTUAL TESTS NOW
  takestep()
  takestep()
  takestep()

  println("Initializing memory with time data.")
  val baseaddr = 0
  val (input_data, input_prepped) = create_time_data(gen_input)
  memory.store_data(baseaddr, input_prepped)
  if(!collateOutput) { print_real_vector("INPUT:", input_data) }

  println("Starting FFT")
  Cmd_IHandler.inputs.enqueue(TestCmd.execute(baseaddr))
  val start_cycle = cycles
  until( Cmd_IHandler.isIdle && !dutBusy, dut.config.fft_size*dut.config.fft_
}
  println("FFT finished after %d cycles".format(cycles-start_cycle))

  val out_data = (0 until dut.config.fft_size).toVector.map(idx => ({
```

```
    val memline = memory.single_load_addr(baseaddr + idx*8)
    convert_from_mem(memline)
  }))

  val expected_output_is_unscaled = true
  val test_dft = FFT_TV_Generator.dft(input_data).map( dp =>
    if(expected_output_is_unscaled) (dp._1*dut.config.fft_size ,dp._2*dut.conf
    else dp
  )
  if(collateOutput)
  {
    println("%5s %18s %18s %18s".format("ID", "INPUT      ", "OUTPUT     ", "
    (0 until dut.config.fft_size).map(n => {
      "%4d: (%7.4f, %7.4f) (%7.4f, %7.4f) (%7.4f, %7.4f)".format(
        n, input_data(n), 0.0,
           out_data(n)._1, out_data(n)._2,
           test_dft(n)._1, test_dft(n)._2
      )
    }).foreach(println(_))
  }
  println("FFT finished after %d cycles".format(cycles-start_cycle))

  val test_max_sq_error = max_square_error(out_data, test_dft)
  println("Maximum Square Error: %g".format(test_max_sq_error))

  val unacked_fm_resp = FlexMemResp_IHandler.inputs.length
    // could miss 1 that is setup but not acked
  if(unacked_fm_resp > 0) {
    println(s"ERROR: $unacked_fm_resp unacknowledged responses in FlexMem res
  }

  ok = (test_max_sq_error < pow(10,-6)) && pass && (unacked_fm_resp == 0)
}

package FFT

import Chisel._
import math._

class FFT_TV_Generator(config: FFTConfig) {
  def generate_float(ngen: Int => Double): Tuple2[Vector[Double],Vector[Doubl
    val input_data = Vector.tabulate(config.fft_size)(ngen(_))
```

```
    // Prepare input data
    val input_prepped = Vector.tabulate(config.fft_size)(n => {
      input_data(config.get_stage0_position(n))
    })

    (input_data, input_prepped)
  }
  def generate_fixed(ngen: Int => Double): Tuple2[Vector[Double],Vector[Int]]
    val (input_data, input_prepped) = generate_float(ngen)
    (input_data, input_prepped.map(convert(_)))
  }

  def convert(in: Double): Int = {
    (in*pow(2,config.fix_pt)).toInt
  }
}
object FFT_TV_Generator {
  def dft(in_real: IndexedSeq[Double]): Vector[Tuple2[Double,Double]] = {
    val unscaled = (0 until in_real.length).toVector.map(k => {
      in_real.zipWithIndex.map(Function.tupled((xn, n) => {
        val angle = -2*Pi*k*n/in_real.length
        (xn*cos(angle),xn*sin(angle))
      })).reduce((l,r)=>(l._1+r._1,l._2+r._2))
    })
    unscaled.map(X => (X._1/in_real.length,X._2/in_real.length))
  }
}

abstract class TV_Writer
{
  def emit_data(fname: String): Unit
  def emit_format(fname: String, config: FFTConfig): Unit
}
class TV_Writer_TV[T](data: IndexedSeq[Tuple2[T, T]]) extends TV_Writer
{
  def emit_data(fname: String): Unit = {
    import java.io._
    val data_writer = new PrintWriter(new File(fname))
    data_writer.write("%d\n".format(data.length))
    (0 until data.length).toVector.map(n =>
      ("%s %s\n".format(data(n)._1, data(n)._2))
```

```
    ).foreach(s => data_writer.write(s))
    data_writer.close()
  }
  def emit_format(fname: String, config: FFTConfig): Unit = {
    import java.io._
    val format_writer = new PrintWriter(new File(fname))
    format_writer.write("fft_size=%d\n".format(config.fft_size))
    format_writer.write("data_width=%d\n".format(config.data_width))
    if(config.floating_point) {
      format_writer.write("float=true\n")
    } else {
      format_writer.write("float=false\n")
      format_writer.write("fix_pt=%d\n".format(config.fix_pt))
    }
    format_writer.close()
  }
}
class TV_Writer_C[T <: AnyVal](in: IndexedSeq[Tuple2[T,T]], out: IndexedSeq[T
{
  def emit_data(fname: String): Unit = {
    import java.io._
    val data_writer = new PrintWriter(new File(fname))
    val emit_list = Vector(
      ("input_data_real",  in.map(_._1)),
      ("input_data_imag",  in.map(_._2)),
      ("output_data_real", out.map(_._1)),
      ("output_data_imag", out.map(_._2)),
      ("tf_real", tf.map(_._1)),
      ("tf_imag", tf.map(_._2))
    )
    emit_list.foreach(Function.tupled((vname, vdata) => {
      val vdatastring = vdata.map(_.toString).reduce(_+","+_)
      data_writer.write("%s %s[%d] = {%s};\n\n".format(
        (if(floating_point) "double" else "int"), vname, vdata.length, vdatas
      )
    }))
    data_writer.close()
  }
  def emit_format(fname: String, config: FFTConfig): Unit = {
    import java.io._
    val format_writer = new PrintWriter(new File(fname))
    format_writer.write("#define FFT_SIZE %d\n".format(config.fft_size))
```

```
    if(!floating_point) {
      format_writer.write("#define DATA_WIDTH %d\n".format(config.data_width)
      format_writer.write("#define FIX_PT %d\n".format(config.fix_pt))
    }
    format_writer.write("#define "+(if(config.init_permute) "DATA_IN_UNPERMUT
    format_writer.write("\n")
    format_writer.write("extern %s input_data_real[FFT_SIZE], input_data_imag
    format_writer.write("extern %s output_data_real[FFT_SIZE], output_data_im
    format_writer.write("extern %s tf_real[FFT_SIZE], tf_imag[FFT_SIZE];\n".f
    format_writer.close()
  }
}

object TV_Engine
{
  def apply(config: FFTConfig, fn_pfx: String = "", fmt: String = "tv")
  {
    require(Vector("tv","C").contains(fmt), "Desired format must be tv or C")

    import java.io._
    val rand = new scala.util.Random
    def gaussian_gen_data = (n: Int) => {rand.nextGaussian()}
    def dc_gen_data       = (n: Int) => { 1.0 }
    def cos_gen_data      = (n: Int) => {val freq1 = 2*math.Pi*1/config.fft_s

    val gen_list = Vector(
      ("gaussian", gaussian_gen_data),
      ("dc", dc_gen_data),
      ("cos", cos_gen_data)
    )

    val tv_gen = new FFT_TV_Generator(config)

    gen_list.foreach(Function.tupled((pfx,gen_data) => {
      if(config.floating_point)
      {
        val (tv_data, tv_real_prep) = tv_gen.generate_float(gen_data)
        val tv_prepped = tv_real_prep.map(r => (r, 0.0))
        val out_data = FFT_TV_Generator.dft(tv_data).map(o => (o._1*config.ff

        fmt match {
          case "C" =>
```

```
        val tf_table = TFTableGen.float(config)
        val writer = new TV_Writer_C(tv_prepped, out_data, tf_table, conf
        writer.emit_data(fn_pfx+pfx+"_data.c")
        writer.emit_format(fn_pfx+"fft_const.h", config)
      case "tv" =>
        val tv_as_int = tv_prepped.map({case (r,i) =>
          (BigInt(java.lang.Float.floatToIntBits(r.toFloat)), BigInt(java
        })

        val in_writer  = new TV_Writer_TV(tv_as_int)
        val out_writer = new TV_Writer_TV(out_data)
        in_writer.emit_data( fn_pfx+pfx+".in.tv")
        out_writer.emit_data(fn_pfx+pfx+".out.tv")
        in_writer.emit_format(fn_pfx+"tv.format", config)
    }
  }
  else
  {
    val (tv_data, tv_real_prep) = tv_gen.generate_fixed(gen_data)
    val tv_prepped = tv_real_prep.map(r => (r, 0))
    val out_data = FFT_TV_Generator.dft(tv_data).map(o => (o._1*config.ff
    val out_prepped = out_data.map(o => (tv_gen.convert(o._1), tv_gen.con

    println("INPUT:")
    (0 until tv_data.length).toVector.map(n =>
      ("%4d: (%7.4f, %7.4f) -> (%10d, %10d)".format(
        n, tv_data(n), 0.0, tv_prepped(n)._1, tv_prepped(n)._2)
      )
    ).foreach(println(_))
    println("OUTPUT:")
    (0 until tv_data.length).toVector.map(n =>
      ("%4d: (%9.4f, %9.4f) -> (%10d, %10d)".format(
        n, out_data(n)._1, out_data(n)._2, out_prepped(n)._1, out_prepped
      )
    ).foreach(println(_))

    fmt match {
      case "tv" =>
        val in_writer  = new TV_Writer_TV(tv_prepped)
        val out_writer = new TV_Writer_TV(out_prepped)
        in_writer.emit_data( fn_pfx+pfx+".in.tv")
        out_writer.emit_data(fn_pfx+pfx+".out.tv")
```

```
            in_writer.emit_format(fn_pfx+"tv.format", config)
          case "C" =>
            val tf_table = TFTableGen.fixed(config)
            val writer  = new TV_Writer_C(tv_prepped, out_prepped, tf_table,
            writer.emit_data(fn_pfx+pfx+"_data.c")
            writer.emit_format(fn_pfx+"fft_const.h", config)
        }
      }
    }))
  }
}

package FFT

import Chisel._
import Node._

object TFTableGen
{
  def float(config: FFTConfig): IndexedSeq[Tuple2[Double,Double]] = {
    import math._
    val depth = config.fft_size
    (0 until depth).toVector.map(k => {
      val radians = (-2*Pi*k.toDouble)/depth.toDouble
      val real = cos(radians)
      val imag = sin(radians)
      (real, imag)
    })
  }
  def fixed(config: FFTConfig): IndexedSeq[Tuple2[Int,Int]] = {
    import math._
    val data = float(config)
    data.map( pt => ((pt._1*pow(2,config.fix_pt)).toInt, (pt._2*pow(2,config.
  }
}

class TFRom(num_ports: Int, config: FFTConfig) extends Module {
  val depth = config.fft_size // Look into cutting this later

  val addr_width = log2Up(depth)
  val io = new Bundle {
    val raddr = Vec.fill(num_ports){Bits(INPUT, addr_width)}
```

```
    val rdata = Vec.fill(num_ports){config.DataMath().asOutput}
  }
/*
  def make_const(real: Int, imag: Int): Complex = {
    val out = config.DataMath()
    out.real := SInt(real); out.imag := SInt(imag)
    out
  }
*/
  def make_const(real: BigInt, imag: BigInt) = {
    def bitify(i: BigInt) = SInt(i,width=config.data_width)

    val out = Cat(bitify(imag), bitify(real))
    out
  }
  def make_table: IndexedSeq[Tuple2[BigInt,BigInt]] = {
    if(config.floating_point) {
      val floaty_table = TFTableGen.float(config)
      floaty_table.map( i => (BigInt(java.lang.Float.floatToIntBits(i._1.toFl
    } else {
      val inty_table = TFTableGen.fixed(config)
      inty_table.map( i => (BigInt(i._1), BigInt(i._2)) )
    }
  }

  val rom = Vec.fill(depth){Bits(width=config.data_width*2)}
  val tf_table = make_table
  (0 until depth).foreach(k => {
    rom(k) := make_const(tf_table(k)._1, tf_table(k)._2)
  })

  (0 until num_ports).foreach(n => {
    val reg_raddr = Reg(next=io.raddr(n))
    val raw_data = rom(reg_raddr)
    io.rdata(n) := config.unpack_to_math(raw_data)
  })
}

package TimeMultiplexer

import Chisel._
import Node._
```

```
case class BinaryConfig[T <: Data](n_inputs: Int, operations: Seq[Tuple3[Int,
                                   reg_all_outputs: Boolean, n_execs: Int,
                                   exec_unit_gen: ()=>BinaryExec[T], exec_lat
                                   dtype: T) {
// operations syntax is: input 0, input 1, function select

  // These calculated values are likely helpful for other design modules
  val n_outputs = operations.length
  require(n_execs <= n_outputs, "TM.B: # execution units > # operations")
  require(n_execs > 0, "TM.B: Cannot have 0 execution units")

  val schedule = operations.zipWithIndex.grouped(n_execs).toVector
  val inject_delay = schedule.length
    // how many clock ticks must we wait until can inject another set of data

  val total_latency = inject_delay-1 + exec_latency + (if(reg_all_outputs) 1
    // -1 on inject delay tells how many clock ticks it takes to present data
    //    (note last presentation doesn't take a clock tick so this can be 0!
    // how many cycles are required between valid in and valid out (can be 0)

  // More detailed scheduling follows
  val n_results_regs = if(reg_all_outputs) (n_outputs) else (
    if(schedule.length <= 1) 0 else (
      schedule.dropRight(1).map(_.length).reduce(_+_)
    )
  )

  val full_exec_schedule = Vector.tabulate(n_execs)(n => {
    schedule.filter(_.length > n).map(_(n))
    // Seq of Seq of ((in0, in1, func), output)
    // Outer seq has element for each execution unit
    // Inner seq has element for each op for specific exec unit
  })
  val req_exec_schedule    = full_exec_schedule.map(_.map(__1))
  val result_exec_schedule = full_exec_schedule.map(_.map(__2))

  val result_schedule = Vector.tabulate(n_outputs)(n => {
    // Vector for each output result of (producing machine, producing round)
    val found_row = result_exec_schedule.filter(_.contains(n)).head
    val exec_id = result_exec_schedule.indexOf(found_row)
    val round = found_row.indexOf(n) + 1 // +1 since 0 is data-invalid stage
```

```
    (exec_id, round)
  })

  val num_states = 1 + schedule.length // reserve 0 for data not valid
/*
  println("Printing verbose diagnostics:")
  println("Latency: %d + %d = %d (I+E=T)".format(inject_delay, exec_latency,
  println("# Results Registers: %d".format(n_results_regs))

  println("Full Exec Schedule:")
  (0 until full_exec_schedule.length).foreach(n => {
    println("M%2d: ".format(n) + full_exec_schedule(n).map(_.toString).reduce
  })
  println("Result Schedule:")
  (0 until result_schedule.length).foreach(n => {
    println("R%2d: ".format(n) + result_schedule(n).toString)
  })
*/
}

class Binary[T <: Data](val config: BinaryConfig[T]) extends Module {
  // These io classes are necessary to circumvent a scala issue
  class io_in extends Bundle {
    val valid = Bool(INPUT)
    val bits  =(Vec.fill(config.n_inputs){config.dtype.clone}).asInput
    val ready = Bool(OUTPUT)
  }
  val io = new Bundle {
    val in = new io_in
    val out = Valid(Vec.fill(config.n_outputs){config.dtype.clone}).asOutput
  }

  // Create the execution pipelines
  val execs = Vector.fill(config.n_execs)({config.exec_unit_gen()})

  // Create the input-side state tracking
  val inject_state = if(config.num_states <= 2) ({
    // Only one round of data injection so only care if data valid or not
    // and always ready for new data
    io.in.ready := Bool(true)
    Mux(io.in.valid, UInt(1), UInt(0))
  })
```

```
  else ({
    // Multiple rounds of data injection
    // The counter starts at 2 since will do first round of data injection wh
    // (Recall 0 implies no valid data so doing nothing)
    val inj_state_reg = Reg(init=UInt(0, width=log2Up(config.num_states)))
    io.in.ready := (inj_state_reg === UInt(0))
    val start_inject = io.in.valid && io.in.ready

    when(start_inject) {
      inj_state_reg := UInt(2)
    }.elsewhen(inj_state_reg === UInt(config.num_states-1)) {
      inj_state_reg := UInt(0) // reset to 0 when done
    }.elsewhen(inj_state_reg != UInt(0)) {
      inj_state_reg := inj_state_reg + UInt(1)
    }
    Mux(start_inject, UInt(1), inj_state_reg)
  })

  // Handle input side of execution units
  val stored_inputs = Vec.fill(config.n_inputs){Reg(config.dtype.clone)}
  when(inject_state === UInt(1)) { // Latch in data as needed for a new set
    stored_inputs := io.in.bits
  }
  execs.zip(config.req_exec_schedule).foreach(Function.tupled((exec, schedule
    // Connect top element of schedule to default of exec unit, use io inputs
    val first_req = schedule(0)
    exec.io.in(0) := io.in.bits(first_req._1)
    exec.io.in(1) := io.in.bits(first_req._2)
    exec.io.func  := UInt(first_req._3)
    // Now, override for when in other injection states
    if(schedule.length > 1) {
      schedule.drop(1).zip(2 until 2+schedule.length).foreach(Function.tupled
        when(inject_state === UInt(assoc_state)) {
          exec.io.in(0) := stored_inputs(req._1)
          exec.io.in(1) := stored_inputs(req._2)
          exec.io.func  := UInt(req._3)
        }
      }))
    }
  }))

  // Pipe the stage over to the output side so know what to do with results
```

```
  val result_state = Delayline(inject_state, config.exec_latency)

  // Handle output side of execution units
  // Store results that are ready early
  val stored_outputs = Vec.fill(config.n_results_regs){Reg(config.dtype.clone
  (0 until config.n_results_regs).foreach(n => {
    val assoc_exec  = config.result_schedule(n)._1
    val assoc_state = config.result_schedule(n)._2
    when(result_state === UInt(assoc_state)) {
      stored_outputs(n) := execs(assoc_exec).io.out
    }
  })
  // Wire up the outputs
  (0 until config.n_outputs).foreach(n => {
    io.out.bits(n) := (
      if(n < config.n_results_regs) (
        stored_outputs(n)
      ) else (
        execs(config.result_schedule(n)._1).io.out
      )
    )
  })
  // Declare valid on the last cycle data comes out
  def apply_out_delay[T<:Data](in: Data) = {if(config.reg_all_outputs) Reg(ne
  io.out.valid := apply_out_delay(result_state === UInt(config.num_states-1))
    // Note: result_state discusses state coming out of execution units
    // Thus, must delay valid for a cycle when registering all outputs
    //   (so can register the last set of outputs)
}

class BinaryExecIO[T <: Data](num_funcs: Int)(dtype: T) extends Bundle {
  val in  = (Vec.fill(2){dtype.clone}).asInput
  val func= Bits(width=log2Up(num_funcs)).asInput
  val out = dtype.clone.asOutput
}
abstract class BinaryExec[T <: Data](num_funcs: Int)(dtype: T) extends Module
  val io = new BinaryExecIO(num_funcs)(dtype)
}

class TestBE[T <: UInt](latency: Int, dtype: T) extends BinaryExec(2)(dtype)
  io.out := Delayline(io.in(0) + Mux(io.func===Bits(1), -io.in(1), io.in(1)),
}
```

```
object BinaryTester {
  val test_data = UInt(width=32)
  val test_ops = Vector((0,1,0),(2,3,0),(1,2,0))
  val test_latency = 0
  val test_config = BinaryConfig(n_inputs=4, test_ops,
                                 reg_all_outputs=true, n_execs=1,
                                 ()=>(Module(new TimeMultiplexer.TestBE(test_
                                 test_data )
}

class BinaryTester(dut: Binary[UInt]) extends MapTester(dut, Array(dut.io)) {
  var printTrace = false
  defTests {
    import scala.collection.mutable.HashMap
    val ivars = new HashMap[Node, Node]()
    val ovars = new HashMap[Node, Node]()
    var current_cycle = 0

    var current_data_set = 0
    var last_inject = -1
    val inject_times = new scala.collection.mutable.Queue[Int]

    def setup_data(dgen: Int=>Int): Seq[Int] = {
      val inj_data = (0 until dut.config.n_inputs).toVector.map(dgen(_))
      (0 until dut.config.n_inputs).foreach(i => {
        ivars(dut.io.in.bits(i)) = UInt(inj_data(i))
      })
      ivars(dut.io.in.valid) = Bool(true)
      inj_data
    }
    def read_data(): Seq[Int] = {
      (0 until dut.config.n_outputs).toVector.map(
        o => ovars(dut.io.out.bits(o)).litValue().toInt
      )
    }
    def isValidI = () => ivars(dut.io.in.valid).litValue() == 1
    def isFiredI = () => ovars(dut.io.in.ready).litValue() == 1 && isValidI()
    def isValidO = () => ovars(dut.io.out.valid).litValue() == 1

    def mystep() = {step(ivars, ovars, printTrace); current_cycle += 1}
    ivars(dut.io.in.valid) = Bool(false)
    mystep() // always burn at least one clock cycle to startup
```

```
    while(ovars(dut.io.in.ready).litValue()!=1) {mystep()} // burn cycles uni

    while(current_cycle < 1000 && (current_data_set < 10 || !inject_times.isEm
      mystep()
      if(isFiredI())
      {
        ivars(dut.io.in.valid) = Bool(false) // invalidate used data
        inject_times.enqueue(current_cycle)
        println("IN: Injected data set %02d on cycle %03d".format(current_dat
          (if(last_inject >= 0) " (inject delay = %d)".format(current_cycle-l
        )
        last_inject = current_cycle
      }
      if(!isValidI() && current_data_set < 10) // always have valid data prep
      {
        val inj_data = setup_data(x => x*current_data_set + 0)
        println("Setup data set %02d: ".format(current_data_set) +
          inj_data.map(_.toString).reduce(_+","+_))
        current_data_set += 1
      }
      if(isValidO()) // handle output data
      {
        val start_time = inject_times.dequeue()
        val out_data = read_data()
        println("OUT: Read data on cycle %03d from cycle %03d (latency = %d):
            current_cycle, start_time, current_cycle-start_time
          ) + out_data.map(_.toString).reduce(_+","+_)
        )
      }
    }
    println("Calculated Latency = %d, Inject Delay = %d".format(dut.config.to
    true
  }
}

// Some basic building blocks

object Delayline {
  def apply[T <: Data](in: T, latency: Int): T = {
    assert(latency >= 0)
    if(latency == 0)
      in
```

```
    else
      apply(Reg(in), latency-1)
  }
}

class StaticDelayblock[T<:Data](delay: Int)(dtype: T) extends Module {
// This block will hold data for a set number of cycles before releasing it.
// WARNING: Sending new data before old data is released will overwrite and r
// Static here is supposed to imply this lack of checking
  val io = new Bundle {
    val in  = Valid(dtype).asInput
    val out = Valid(dtype).asOutput
  }
  if(delay <= 0) { io.out <> io.in }
  else {
    val count = Reg(init=UInt(0, width=log2Up(delay+1)))
    // count = 0 implies no valid data, hence the need for +1
    val held_data = Reg(dtype.clone)
    when(io.in.valid) {
      count := UInt(1)
      held_data := io.in.bits
    }.elsewhen(count === UInt(delay)) {
      count := UInt(0)
    }.elsewhen(count != UInt(0)) {
      if(delay>1) {count := count + UInt(1)}
      // The if check is technically unnecessary but ensures adders are not c
      //  in what could be a fairly common case.
    }

    io.out.bits := held_data
    io.out.valid := (count === UInt(delay))
  }
}
class StaticDelayblockSeries[T<:Data](inject_delay: Int, total_latency: Int)(
// This block will hold data for a set number of cycles before releasing it.
// WARNING: Sending new data before old data is released will overwrite and r
// Static here is supposed to imply this lack of checking
  require(inject_delay>0,    "TM.SDS: Injection delay must be positive integer
  require(total_latency>=0, "TM.SDS: Total latency be nonnegative integer")
  val io = new Bundle {
    val in  = Valid(dtype).asInput
    val out = Valid(dtype).asOutput
```

```
  }
  val num_blocks = total_latency / inject_delay
  val rem_delay  = total_latency % inject_delay

  def ensure_nonempty(in: Seq[Int]): Seq[Int] = { if(in.isEmpty) Vector(0) el
  val chain_spec = (Vector.fill(num_blocks)(inject_delay) :+ rem_delay).filte
    // Setup the series of delay blocks. Also, don't bother emitting blocks t

  if(chain_spec.isEmpty) {
    io.out <> io.in // Don't need any delays so this is trivially easy
  } else {
    // Otherwise, apply the chain specification using foldLeft
    io.out <> chain_spec.foldLeft(io.in)((stage_in, stage_delay) => {
      val stage_block = Module(new StaticDelayblock(stage_delay)(dtype))
      stage_block.io.in <> stage_in
      stage_block.io.out
    })
  }
}

package FFT

import Chisel._

class RV_Merge[S <: Data, T <: Data](dtype1: S, dtype2: T) extends Module {
  // For now, can only accept 2 inputs due to issues annotating datatypes in
  // This component waits until both inputs have given valid data and then
  //   sends out the married result on the output
  class out_io extends Bundle {
    val data1 = dtype1.clone
    val data2 = dtype2.clone
    override def clone = new out_io().asInstanceOf[this.type]
  }

  val io = new Bundle {
    val in1 = Decoupled(dtype1.clone).flip
    val in2 = Decoupled(dtype2.clone).flip

    val out = Decoupled(new out_io)
  }

  val v1_reg = Reg(init=Bool(false))
```

```
  val d1_reg = Reg(dtype1.clone)

  val v2_reg = Reg(init=Bool(false))
  val d2_reg = Reg(dtype2.clone)

  val consumed = v1_reg && v2_reg && io.out.ready

  // input to registers
  io.in1.ready := !v1_reg || consumed
  when(!v1_reg || consumed) {
    v1_reg := io.in1.valid
    d1_reg := io.in1.bits
  }

  io.in2.ready := !v2_reg || consumed
  when(!v2_reg || consumed) {
    v2_reg := io.in2.valid
    d2_reg := io.in2.bits
  }

  // registers to output
  io.out.valid := v1_reg && v2_reg
  io.out.bits.data1 := d1_reg
  io.out.bits.data2 := d2_reg
}

class RV_Split[T <: Data](data: T, n_outs: Int) extends Module {
  // This component clones the input onto all the outputs but with the proper
  //   to ensure each output only gets a given input once
  val io = new Bundle {
    val in = Decoupled(data.clone).flip
    val out = Vec.fill(n_outs){Decoupled(data.clone)}
  }

  // Storage space for the data until all outputs have consumed it
  val v_reg = Vector.fill(n_outs)(Reg(init=Bool(false)))
  val d_reg = Reg(data.clone)

  val old_consumed = (0 until n_outs).map(n => !v_reg(n)||io.out(n).ready).re
    // true if all entries already taken (v_reg=0) or will be taken on clock
  val take_new = old_consumed && io.in.valid
```

```
  // data in
  io.in.ready := old_consumed
  when(take_new) { d_reg := io.in.bits }
  (0 until n_outs).foreach(n => {
    when(take_new) {
      v_reg(n) := Bool(true)
    }.elsewhen(io.out(n).ready) {
      v_reg(n) := Bool(false)
    }
  })
  //data out
  (0 until n_outs).foreach(n => {
    io.out(n).valid := v_reg(n)
    io.out(n).bits  := d_reg
  })
}

package FFT

import Chisel._
import Node._

// Address Translation Unit
class WriteATU(config: FFTConfig, atlut: IndexedSeq[(Int, Int)]) extends Modu
  val io = new Bundle {
    val in  = Valid(config.FFTWriteReq() ).asInput
    val out = Valid(config.FullWriteReq()).asOutput
  }
  // setup ROMs
  val rom_id  = Vec.fill(config.fft_size){Bits(width=log2Up(config.num_dmem))
  val rom_addr= Vec.fill(config.fft_size){Bits(width=config.dmem_addr_width)}
  (0 until config.fft_size).foreach(n => {
    rom_id(n)   := Bits(atlut(n)._1)
    rom_addr(n) := Bits(atlut(n)._2)
  })

  // apply ROMs
  val s1_valid   = Reg(next=io.in.valid)
  val s1_request = Reg(next=io.in.bits)

  val s2_valid = Reg(next=s1_valid)
  val s2_id    = Reg(next=rom_id(s1_request.address))
```

```
  val s2_addr  = Reg(next=rom_addr(s1_request.address))
  val s2_data  = Reg(next=s1_request.payload)

  io.out.valid          := s2_valid
  io.out.bits.address := s2_id
  io.out.bits.payload.address := s2_addr
  io.out.bits.payload.payload := s2_data
}

class WriteRouter(config: FFTConfig, atlut: IndexedSeq[(Int, Int)]) extends M
  val io = new Bundle {
    val in  = Valid(config.FFTWriteReq()).asInput
    val localports = config.LocalWritePorts
  }

  // Make the ATU
  val atu = Module(new WriteATU(config, atlut))
  atu.io.in := io.in

  // Actually send out the request
  for (o <- 0 until config.num_dmem) {
    val addr_match = atu.io.out.bits.address === Bits(o);

    io.localports(o).valid := addr_match && atu.io.out.valid
    io.localports(o).bits.address := atu.io.out.bits.payload.address
    io.localports(o).bits.payload := atu.io.out.bits.payload.payload
  }
}
```

7.3 Accelerator Verilog Testbench

```
//========================================================================
// Test harness for CS250 FFT
//------------------------------------------------------------------------
// Built using CS250 RISCV Test Harness as a guide

module TestHarness;

  //----------------------------------------------------------
  // Instantiate the fft
  localparam fmw = 512; // flex mem width
  localparam wdw = 64; // word width
```

```
reg clk    = 0;
reg reset = 1;

always #CLOCK_PERIOD clk = ~clk;

// Instantiate all the I/O signals
wire io_cmd_ready;
reg  io_cmd_valid;
reg [6:0]   io_cmd_bits_inst_funct;
reg [4:0]   io_cmd_bits_inst_rs2;
reg [4:0]   io_cmd_bits_inst_rs1;
reg         io_cmd_bits_inst_xd;
reg         io_cmd_bits_inst_xs1;
reg         io_cmd_bits_inst_xs2;
reg [4:0]   io_cmd_bits_inst_rd;
reg [6:0]   io_cmd_bits_inst_opcode;
reg [wdw-1:0] io_cmd_bits_rs1;
reg [wdw-1:0] io_cmd_bits_rs2;

reg  io_resp_ready;
wire io_resp_valid;
wire [4:0] io_resp_bits_rd;
wire [wdw-1:0] io_resp_bits_data;

reg  io_mem_req_ready;
wire io_mem_req_valid;
wire         io_mem_req_bits_kill;
wire [2:0]   io_mem_req_bits_typ;
wire         io_mem_req_bits_phys;
wire [43:0] io_mem_req_bits_addr;
wire [wdw-1:0] io_mem_req_bits_data;
wire [8:0]   io_mem_req_bits_tag;
wire [3:0]   io_mem_req_bits_cmd;

reg  io_mem_resp_valid;
reg [2:0]   io_mem_resp_bits_typ;
reg         io_mem_resp_bits_has_data;
reg [wdw-1:0] io_mem_resp_bits_data;
reg [8:0]   io_mem_resp_bits_tag;
reg [3:0]   io_mem_resp_bits_cmd;
reg [43:0] io_mem_resp_bits_addr;
```

```
reg [wdw-1:0] io_mem_resp_bits_store_data;

reg  io_flexmem_req_ready;
wire io_flexmem_req_valid;                reg fm_req_valid;
wire [6:0]   io_flexmem_req_bits_bytes; reg [6:0]    fm_req_bytes;
wire         io_flexmem_req_bits_phys;  reg          fm_req_phys;
wire [43:0] io_flexmem_req_bits_addr;  reg [43:0]  fm_req_addr; wire [43:0]
wire [fmw-1:0] io_flexmem_req_bits_data; reg [fmw-1:0] fm_req_data; reg [fm
wire [8:0]   io_flexmem_req_bits_tag;   reg [8:0]    fm_req_tag;
wire [3:0]   io_flexmem_req_bits_cmd;   reg [3:0]    fm_req_cmd;

wire io_flexmem_resp_ready;
reg  io_flexmem_resp_valid;
reg [fmw-1:0] io_flexmem_resp_bits_data;
reg [8:0]  io_flexmem_resp_bits_tag;
reg [3:0]  io_flexmem_resp_bits_cmd;

wire io_busy;
wire io_interrupt;

// Need to delay inputs
wire         #0.1 d_io_cmd_valid = io_cmd_valid;
wire [6:0]   #0.1 d_io_cmd_bits_inst_funct = io_cmd_bits_inst_funct;
wire [4:0]   #0.1 d_io_cmd_bits_inst_rs2 = io_cmd_bits_inst_rs2;
wire [4:0]   #0.1 d_io_cmd_bits_inst_rs1 = io_cmd_bits_inst_rs1;
wire         #0.1 d_io_cmd_bits_inst_xd = io_cmd_bits_inst_xd;
wire         #0.1 d_io_cmd_bits_inst_xs1 = io_cmd_bits_inst_xs1;
wire         #0.1 d_io_cmd_bits_inst_xs2 = io_cmd_bits_inst_xs2;
wire [4:0]   #0.1 d_io_cmd_bits_inst_rd = io_cmd_bits_inst_rd;
wire [6:0]   #0.1 d_io_cmd_bits_inst_opcode = io_cmd_bits_inst_opcode;
wire [wdw-1:0]  #0.1 d_io_cmd_bits_rs1 = io_cmd_bits_rs1;
wire [wdw-1:0]  #0.1 d_io_cmd_bits_rs2 = io_cmd_bits_rs2;

wire   #0.1 d_io_resp_ready = io_resp_ready;

wire   #0.1 d_io_mem_req_ready = io_mem_req_ready;

wire         #0.1 d_io_mem_resp_valid = io_mem_resp_valid;
wire [2:0]   #0.1 d_io_mem_resp_bits_typ = io_mem_resp_bits_typ;
wire         #0.1 d_io_mem_resp_bits_has_data = io_mem_resp_bits_has_data;
wire [wdw-1:0]  #0.1 d_io_mem_resp_bits_data = io_mem_resp_bits_data;
wire [8:0]   #0.1 d_io_mem_resp_bits_tag = io_mem_resp_bits_tag;
```

```
  wire [3:0]    #0.1 d_io_mem_resp_bits_cmd = io_mem_resp_bits_cmd;
  wire [43:0]   #0.1 d_io_mem_resp_bits_addr = io_mem_resp_bits_addr;
  wire [wdw-1:0]  #0.1 d_io_mem_resp_bits_store_data = io_mem_resp_bits_store

  wire    #0.1 d_io_flexmem_req_ready = io_flexmem_req_ready;

  wire          #0.1 d_io_flexmem_resp_valid = io_flexmem_resp_valid;
  wire [fmw-1:0]  #0.1 d_io_flexmem_resp_bits_data = io_flexmem_resp_bits_dat
  wire [8:0]    #0.1 d_io_flexmem_resp_bits_tag = io_flexmem_resp_bits_tag;
  wire [3:0]    #0.1 d_io_flexmem_resp_bits_cmd = io_flexmem_resp_bits_cmd;

  FFT dut
  (
    .clk(clk),
    .reset(reset),

    .io_cmd_ready(io_cmd_ready),
    .io_cmd_valid(d_io_cmd_valid),
    .io_cmd_bits_inst_funct(d_io_cmd_bits_inst_funct),
    .io_cmd_bits_inst_rs2(d_io_cmd_bits_inst_rs2),
    .io_cmd_bits_inst_rs1(d_io_cmd_bits_inst_rs1),
    .io_cmd_bits_inst_xd(d_io_cmd_bits_inst_xd),
    .io_cmd_bits_inst_xs1(d_io_cmd_bits_inst_xs1),
    .io_cmd_bits_inst_xs2(d_io_cmd_bits_inst_xs2),
    .io_cmd_bits_inst_rd(d_io_cmd_bits_inst_rd),
    .io_cmd_bits_inst_opcode(d_io_cmd_bits_inst_opcode),
    .io_cmd_bits_rs1(d_io_cmd_bits_rs1),
    .io_cmd_bits_rs2(d_io_cmd_bits_rs2),

    .io_resp_ready(d_io_resp_ready),
    .io_resp_valid(io_resp_valid),
    .io_resp_bits_rd(io_resp_bits_rd),
    .io_resp_bits_data(io_resp_bits_data),

    .io_mem_req_ready(d_io_mem_req_ready),
    .io_mem_req_valid(io_mem_req_valid),
    .io_mem_req_bits_kill(io_mem_req_bits_kill),
    .io_mem_req_bits_typ(io_mem_req_bits_typ),
    .io_mem_req_bits_phys(io_mem_req_bits_phys),
    .io_mem_req_bits_addr(io_mem_req_bits_addr),
    .io_mem_req_bits_data(io_mem_req_bits_data),
    .io_mem_req_bits_tag(io_mem_req_bits_tag),
```

```
    .io_mem_req_bits_cmd(io_mem_req_bits_cmd),

    .io_mem_resp_valid(d_io_mem_resp_valid),
    .io_mem_resp_bits_typ(d_io_mem_resp_bits_typ),
    .io_mem_resp_bits_has_data(d_io_mem_resp_bits_has_data),
    .io_mem_resp_bits_data(d_io_mem_resp_bits_data),
    .io_mem_resp_bits_tag(d_io_mem_resp_bits_tag),
    .io_mem_resp_bits_cmd(d_io_mem_resp_bits_cmd),
    .io_mem_resp_bits_addr(d_io_mem_resp_bits_addr),
    .io_mem_resp_bits_store_data(d_io_mem_resp_bits_store_data),

    .io_flexmem_req_ready(d_io_flexmem_req_ready),
    .io_flexmem_req_valid(io_flexmem_req_valid),
    .io_flexmem_req_bits_bytes(io_flexmem_req_bits_bytes),
    .io_flexmem_req_bits_phys(io_flexmem_req_bits_phys),
    .io_flexmem_req_bits_addr(io_flexmem_req_bits_addr),
    .io_flexmem_req_bits_data(io_flexmem_req_bits_data),
    .io_flexmem_req_bits_tag(io_flexmem_req_bits_tag),
    .io_flexmem_req_bits_cmd(io_flexmem_req_bits_cmd),

    .io_flexmem_resp_ready(io_flexmem_resp_ready),
    .io_flexmem_resp_valid(d_io_flexmem_resp_valid),
    .io_flexmem_resp_bits_data(d_io_flexmem_resp_bits_data),
    .io_flexmem_resp_bits_tag(d_io_flexmem_resp_bits_tag),
    .io_flexmem_resp_bits_cmd(d_io_flexmem_resp_bits_cmd),

    .io_busy(io_busy),
    .io_interrupt(io_interrupt)
);
//--------------------------------------------------------
// Start the simulation

integer fh_in;
integer fh_out;
reg [1023:0] in_vectors_filename;
integer fft_size;
reg [31:0] in_vec_real [];
reg [31:0] in_vec_imag [];
reg [wdw-1:0] simmemory [];

reg [31:0] max_cycles;
reg stats;
```

```
integer i;
integer count;
integer start_time;

initial
begin
  // Get whether doing stats-based emission
  if (!$value$plusargs("stats=%d", stats)) stats = 0;

  `ifndef STATS
    $vcdpluson(0);
  `else
    if(!stats) $vcdpluson(0); // Not recording detailed stats so just start
  `endif

  // This gets the input vectors to use
  if ($value$plusargs("in=%s", in_vectors_filename)) begin
    // Check that file exists
    fh_in = $fopen(in_vectors_filename, "r");
    if (!fh_in) begin
      $display("\n ERROR: Could not open file (%s)! \n", in_vectors_filenam
      $finish;
    end
  end else begin
    $display("\n ERROR: No input test vectors specified! (use +in=<filename
    $finish;
  end

  fh_out = $fopen("fft.out", "w");
  if (!fh_out) begin
    $display("\n ERROR: Could not open file fft.out for writing! \n");
    $finish;
  end

  // Get max number of cycles to run simulation for from command line
  if (!$value$plusargs("max-cycles=%d", max_cycles)) max_cycles = 10000;


  // Get data from input file
  count = $fscanf(fh_in, "%d", fft_size);
  in_vec_real = new[fft_size];
```

```
    in_vec_imag = new[fft_size];
    simmemory = new[fft_size];
    $fwrite(fh_out, "%d\n", fft_size);

    for (i = 0; i < fft_size; i = i+1) begin
      count = $fscanf(fh_in, "%d_%d", in_vec_real[i], in_vec_imag[i]);
      simmemory[i] = {in_vec_imag[i], in_vec_real[i]};
    end

    // Assert reset high for 20 cycles
    io_cmd_valid = 0;
    io_resp_ready = 0;
    io_mem_resp_valid = 0;
    fm_req_valid = 0;

    reset = 1;
    #20 reset = 0;
  end

// HANDLE MEMORY INTERFACES
  // Simulate Simple Memory
  localparam MXRD = 0;
  localparam MXWR = 1;

  assign io_mem_req_ready = !reset;
  always @(posedge clk) begin
    io_mem_resp_valid <= 0;
    io_mem_resp_bits_tag <= io_mem_req_bits_tag;
    io_mem_resp_bits_cmd <= io_mem_req_bits_cmd;
    if(io_mem_req_ready && io_mem_req_valid) begin
      case(io_mem_req_bits_cmd)
        MXRD: begin
          io_mem_resp_valid <= 1;
          io_mem_resp_bits_data <= (io_mem_req_bits_addr >=0 && io_mem_req_bi
        end
        MXWR: begin
          io_mem_resp_valid <= 1;
          simmemory[io_mem_req_bits_addr] <= io_mem_req_bits_data;
        end
      endcase
    end
  end
```

```
// Handle decoupled flex mem port
wire flexmem_stall = !io_flexmem_resp_ready || reset; // stall on reset and
assign io_flexmem_req_ready = !flexmem_stall;

// Take new command but only if not stalled
always @(posedge clk) begin
  if(!flexmem_stall) begin
    fm_req_valid <= io_flexmem_req_valid;
    fm_req_bytes <= io_flexmem_req_bits_bytes;
    fm_req_phys  <= io_flexmem_req_bits_phys;
    fm_req_addr  <= io_flexmem_req_bits_addr;
    fm_req_data  <= io_flexmem_req_bits_data;
    fm_req_tag   <= io_flexmem_req_bits_tag;
    fm_req_cmd   <= io_flexmem_req_bits_cmd;
  end
end

// setup read data
// fm_update_addr = what fm_req_addr will be
assign fm_update_addr = flexmem_stall ? fm_req_addr : io_flexmem_req_bits_a
// thus fm_data_response register is combinational from fm_req_addr!
genvar fm_word;
generate
  for(fm_word = 0; fm_word < fmw/wdw; fm_word = fm_word + 1) begin
    always @(posedge clk) begin
      fm_data_response[(fm_word+1)*wdw-1 : fm_word*wdw] <= simmemory[(fm_up
    end
  end
endgenerate

localparam bytes_per_word = wdw/8;
// do writes
generate
  for(fm_word = 0; fm_word < fmw/wdw; fm_word = fm_word + 1) begin
    always @(posedge clk) begin
      if(fm_req_valid && !reset && (fm_req_cmd == MXWR) && (fm_req_bytes >
        simmemory[(fm_req_addr + fm_word) % fft_size] <= fm_req_data[(fm_wo
      end
    end
  end
endgenerate
```

```
  assign io_flexmem_resp_valid = fm_req_valid;
  assign io_flexmem_resp_bits_tag = fm_req_tag;
  assign io_flexmem_resp_bits_cmd = fm_req_cmd;
  assign io_flexmem_resp_bits_data = (fm_req_cmd == M_XRD) ? fm_data_response
// DONE HANDLING MEMORY INTERFACES

  // Need this cycle counter
  reg [31:0] cycle_count = 0;
  always @(posedge clk)
    cycle_count = cycle_count + 1;

  // Inject instructions, wait, then dump output
  reg [1:0] state = 0; // 0 = injecting instruction, 1 = waiting for execute
  localparam TB_STARTING = 0;
  localparam TB_EXECUTING = 1;
  localparam TB_ENDING = 2;
  always @(posedge clk)
  begin
    io_cmd_valid <= 0;
    io_resp_ready <= 1; // Never expect a response; so, just sink them all
    if(reset) begin
      state <= TB_STARTING;
    end else begin
      case(state)
        TB_STARTING: begin
          io_cmd_valid <= 1;
          io_cmd_bits_inst_funct <= 0;
          io_cmd_bits_inst_rs2 <= 1;
          io_cmd_bits_inst_rs1 <= 2;
          io_cmd_bits_inst_xd <= 0;
          io_cmd_bits_inst_xs1 <= 1;
          io_cmd_bits_inst_xs2 <= 1;
          io_cmd_bits_inst_rd <= 1;
          io_cmd_bits_inst_opcode <= 0;
          io_cmd_bits_rs1 <= 0; // base address is 0
          io_cmd_bits_rs2 <= 1; // stride is 1
          start_time <= cycle_count;
          if(io_cmd_valid && io_cmd_ready) begin
            io_cmd_valid <= 0;
            `ifdef STATS
              if(stats) $vcdpluson(0); // Need detailed stats so start record
```

```
            `endif
            state <= TB_EXECUTING;
          end
        end
        TB_EXECUTING: begin
          if(!io_busy) state <= TB_ENDING;
        end
        TB_ENDING: begin
          for(i = 0; i < fft_size; i = i + 1) begin
            $display("OUT: %04d: %d %d", i, simmemory[i][31:0], simmemory[i][
            $fwrite(fh_out, "%d %d\n", simmemory[i][31:0], simmemory[i][63:32
          end
          $display("Cycle Count: %d", cycle_count-start_time);
          $fclose(fh_in);
          $fclose(fh_out);
          $finish;
        end
      endcase
    end
  end

  //-----------------------------------------------------
  // Safety net to catch infinite loops


  always @(*)
  begin
    if (cycle_count > max_cycles)
    begin
      $display("*** FAILED *** (timeout)");
      $fclose(fh_in);
      $fclose(fh_out);
      $finish;
    end
  end

endmodule
```