CS194-6 Fall2008 Projects

Greg Gibeling& John Lazzaro
UC Berkeley
gdgib@eecs.berkeley.edu

September 8, 2008

1 Goals

As this is a project class you will spend the majority of your time in CS194-6 working on your project, which means that your project will be the primary source of learning in the class. In turn, this means that your project will involve significant design, optimization, testing and documentation effort as these are the cornerstones of digital design. It will be tempting to view this as an implementation project, instead you should take the view that this is a complete project and you may spend as little as a third of your time implementing and debugging gateware.

Below we present several possible projects, and of which we would be happy for you to work on. You should also feel free to create your own project, and in fact we encourage this, though it must meet the above goals. In particular ask yourself the following questions:

1. Is it appropriate to implement this in hardware (as opposed to using a processor)?
2. Is there a significant design effort at some level?
3. Is there some objective or subjective metric on which the quality or optimality of this project can be judged?
4. What kind of documentation will be required to make this project useful to another designer?
5. Could one produce a series of automated unit tests which will allow someone else to easily evaluate whether this project works?

For this project you will need to pass three reviews: an initial one covering the idea and design, a second one covering implementation and a final report. All of these will be done in conjunction with both John Lazzaro and Greg Gibeling. As mentioned above, while it is tempting to focus on implementation, design, testing and documentation will be equally important. Your goal you should be to produce a code, documentation and test base which survives long after you finish this class.

Finally, on a technical level this course is not intended solely to teach processor design. Processor design will be an important component of the class, but this is a class on digital design in general, and those wishing for a processor-centric view of the world should take CS152 instead. In particular you are encouraged to focus on dataflow, feed forward designs which are not FSM-centric in this class, and you are encouraged to use data stationary control and pipelining together. Finally, in all applicable situations you are expected to conform to the FIFO handshaking interface specification in Section 5. Projects in this class, should not, unlike EECS150 projects, merely be a collection of complex I/O modules, but should instead include a significant computation and optimization component.

Finally, for those of you worried about finishing a complex project on a tight deadline: relax. Your goal in this course is to produce code, tests and documentation, not necessarily to complete them. Of course we will make every effort to ensure that you pick a project that can be completed within the semester, however we may not succeed. In either case your grade will depend on you putting in a reasonable amount of effort, rather than on your objective success. Those of you wishing to continue working on your project after the semester is over will almost certainly be accomodated.

2 Outline

In general the projects for this course, though very different in implementation goal, will follow a simple script.

2.1 Design

First, you will be expected to research the functionality you wish to implementation. This will include finding existing implementations, deciding on how your implementation will be parameterized, and what kind of testing you will do. Following this, you will implement a hardware generator, either through standard Verilog parameters or some more advanced means. Simply printing Verilog from a program is effective, though we highly encourage you to take a less crude approach such as using the RAMP RCF libraries, if they are available in time.

2.2 Testing

Having attempted to implement your desired functionality you will be expected to write a complete set of tests for it. We suggest you write the tests as you write the hardware to avoid unpleasant surprises later. Note that your tests must span both simulation and in-FPGA implementation. We highly suggest you use XLink, a library of communications routines to connect hardware, software and simulations and allow you to write test code in Java, as it will save you time. We also suggest you work to automate all of your testing, by using Apache ANT and the ANTEDA tasks to launch both simulations and board tests.

2.3 Integration

One of the main goals of this class and these projects is to produce high quality, reusable gateware which outlasts this class by years not days. For example, Greg Gibeling is providing a fairly complete library of Verilog modules, called GateLib, ranging from a register up through a FIFO and DRAM controllers, all of which should be reused rather than rewritten. This library allows us to easily make much faster progress on these projects, including not only starting from a higher level of abstraction, but also by fixing bugs in shared modules. Furthermore, most of the modules in GateLib are well tested and heavily parameterized, as your modules should be, though yours will often be more complex.

Aside from integrating with GateLib, your project should ideally integrate with EECS150, RAMP Gold and other FPGA projects at U.C. Berkeley. Using the FIFO interface (see Section 5) is a step in the right direction toward functional integration, as is using GateLib. Other steps include using the RAMP code templates, which we will happily provide and checking code in to SVN are simple but powerful steps. You should also think about which parameters of your design can reasonably set after bitfile generation, rather than just as Verilog parameters will be important.

Finally, integrating with RAMP Gold by considering simulation tradeoffs will be a difficult, but intellectually powerful goal. If your modules use the FIFO interface internally rather than custom, rigid and latency sensitive interfaces, they will be recomposable on a finer grain. Coupled with parameters which control space, time and performance tradeoffs this could allow a designer to build a simulation model of your module in hardware. Essentially the designer would instantiate a lower performance version of your module, and add some logic to track target cycles or "simulated time" to make this small module look as if it is the larger, more expensive variant. If this concept isn't yet clear, we will be spending at least some time on it and the overall design of RAMP Gold.

3 Tools

3.1 Languages

You may take your pick of languages in which to code. We encourage you to use Verilog for hardware, Java for software and Apache ANT for build scripting, but we will not limit you to these options. Other interesting languages to consider include VHDL, BlueSpec, SystemVerilog, RDL, C/C++ and makefiles for build scripting. We strongly encourage you to write your documentation in LaTeX, but will happily accept it in any reasonable format.

3.2 Tools

The primary FPGA tools for these projects will be Synplify Pro 9.4, Xilinx ISE 10.1 and ModelSim SE 6.4. Other possible tools include Xilinx XST, Java6, RDLC2 (or 3), BlueSpec, Mentor Precision (for SystemVerilog), GCC or Microsoft Visual Studio, Eclipse and Apache ANT.

We will be giving out SVN accounts to anyone who desires one, and we suggest you get one. For SVN access, we encourage you to take a look at SmartSVN 4, as it works on all platforms and is a high quality client.

3.3 Boards

The primary FPGA platform for this class will be the Xilinx ML505-110 board [4]. This is the standard ML505 board with a larger V5LX110T FPGA. This FPGA should be large enough for any design you work on in this class, but for anyone in need of

more space, time on the BEE3 cluster at BWRC is an option once it is online in a month or so.

While we do encourage you to consider optimizations which will work on multiple FPGAs, this is certainly not required. You may wish to give a passing nod to the CaLinx2+ board or some of the Digilent Spartan3 offerings, but these are not the primary boards for the class.

3.4 Custom Tools

Greg Gibeling, the TA, is actively developing XLink, as it is a critical component of RAMP Gold, though documentation isn't finished more information will be forthcoming. Chris Fletcher, another student, was kind enough to implement ANTEDA, a powerful way to script the FPGA tools, over the summer as part of the work on RAMP Gold. Greg will provide some examples and documentation on how to use these tools together. As with all home grown tools, feedback and constructive complaints are greatly appreciated. It should be noted that standardization on these tools is a large component of this class, as it will allow us to share some otherwise annoying infrastructure work, and allow those who come after us to understand our code more quickly.

4 Project Ideas

4.1 CPU

The goal of this project is to implement a complete CPU, similar to the old CS152 projects. Possible ISAs include SPARC, which is preferred, MIPS and even Fleet.

4.1.1 Goals

The main goal of this project is to produce a simple, elegant, parameterized and optimized processor rather than a complete ISA implementation. The result should be easily extensible by a relative novice hardware designer, and heavily optimized for FPGA implementation. In particular this core should provide a simple base CPU for RAMP Gold, which is easy to extend with new architectural ideas.

In terms of hardware complexity this can vary depending on group size, from single cycle to a pipelined implementation of *e.g.* MicroBlaze or CS152 level complexity. Note that no matter what the complexity, optimization will play a big role in this project. Implementing a simpler design just means it will need to be optimized more.

4.1.2 Requirements

Aside from the processor implementation, this project will require an assembler, a series of unit tests and complete documentation. In particular if done in a larger team the processor should be able to run C code compiled by GCC, perhaps after modifying GCC to emit a reduced instruction set.

Again, if done in a larger team, the resulting HDL should be parameterized to produce either a processor implementation or a simulation. If done, this would provide a significant boost to RAMP Gold.

4.1.3 Difficulty

This project can cover a range of difficulties, from simple processor implementation and basic testing, through building a complete toolchain. Our expectations will be comensurate with the size of the group, in that we *e.g.* wouldn't expect a complete GCC toolchain from a one person group. We would expect this team to consist of at most 3 people.

4.1.4 Starting Points

For the actual processor implementation, there are several starting points available and some of the design work for this project will be selecting one. For the SPARC ISA, Zhangxi Tan has an implementation and OpenSPARC has released two. For MIPS, any CS152 project or the Fall2007 CS61C processor are candidates. For an ISA like FLEET, Greg Gibeling has one implementation, already partially optimized, and Adam Megacz another. For any existing implementation there will be some work integrating it with the shared infrastructure of the class such as XLink, ANTEDA and GateLib.

Any work on this project should ideally interoperate with RAMP Gold, in as many ways as possible. In particular, this project should, if possible, use RDL, the SPARC ISA and any compiler tools developed by the ParLab Arch group this semester. Furthermore RAMP Gold group has at least one ELF library will can be used to load executables, and XLink should be used for data transport.

If the group decides to shoot for running real C programs, GCC and GDB provide a fair amount of information about their own structure for modifications. One of the key problems will be finding a minimum subset of the ISA required to support GCC. Psuedo instruction expansion and binary or assembly language translation may be used to ease this problem. In order to run C code a standard library will be required, and this group is encouraged to look in to 'newlib' from RedHat. Finally, this group should plan on attending the ParLab

ArchOS and RAMP Gold meetings regularly, and participating with the ArchOS groups working on these issues.

4.2 Media & DSP

The goal of this project is to efficiently implement a series of media transformations, with the interfaces necessary to support free form composition, and possibly a system to automate this. In particular this project will cover much implementation and optimization ground, and will require significant mathematical elements. We leave the particular algorithms and transformations up to the imagination of the implementors, but stipulate that a concrete list of features must be part of the final project proposal.

4.2.1 Goals

The goal of this project is to produce high quality media and DSP implementations optimized for the Xilinx Virtex5, which have standardized interfaces. Compression, FFTs and filters are all possible transformations of interest, but this is just an initial list of suggestions.

Taking composition in to account will be a key part of this project. Since the individual transformations will not be unduly complex, any group working on this should implement more than one, and they should be designed for composition. In particular the final project demo should ideally show off the power of these transformations when combined.

As an additional point of interest, with a large enough group, this project could involve dynamic reconfiguration of the FPGAs to support run-time composition changes. It is not clear how well this is supported by the Virtex5 FPGAs and Xilinx EDA tools, and an evaluation of this must be part of any project proposal.

It is unlikely that any DSP transformations here will be novel, meaning that their implementation and testing must be the novelty. In other words, optimization, XLink interfacing, providing test cases, and a reasonable demonstration will all be key parts of this project.

4.2.2 Requirements

All of the DSP modules implemented for this project must support a series of standardized interfaces. This in turn means that a large part of the design effort will be in selecting the appropriate transformations, and designing the necessary interfaces. In particular, if a group wishes to attempt the use of dynamic reconfiguration, much of the design effort will involve prototyping this functionality.

Picking a reasonable set of DSP transformations will not be trivial. In particular they should be capable of optimization, in order to be interesting, note that this does not imply complexity. There should ideally be no good implementation readily available, perhaps by virtue of interface incompatibility. Finally the set of implemented transformations, though possibly very simple, should support a few interesting compositions for demo purposes.

The parameter space covered by these implementation should include performance, precision and accuracy. Ideally a consumer of these transformations should be free to trade time, space and performance in creative ways. In particular, because FPGAs are often used for prototyping, being able switch between a line rate implementation of a transformation, and a much smaller but slower one for testing complex compositions might be interesting.

4.2.3 Difficulty

This project can be done with up to three people. There will be a significant amount of upfront design work on this project in terms of deciding on transformations and outlining the implementation. As such this will be a difficult project. On the other hand it should result in a cool looking demonstration.

4.2.4 Starting Points

Ideally this project should attempt to integrate with the BEE toolflow at BWRC, Xilinx System Generator, MatLab, Simulink and Corgen, all of which are related. It is unlikely that all of these will be successfully integrated, but they should be investigated if for no other reason than comparison.

Finally John Lazzaro and John Wawrzynek both have a strong history in digital music, and will be most happy to provide more references and ideas. For example John Wawrzynek's PhD may be interesting reading.

4.3 IP Level Networking

TCP/IP and its sibling UDP/IP are very common protocols, which are never-the-less often relegated to CPUs because of they're complexity. There are several variations on this project, but the main thrust would be to produce optimized TCP, UDP,

IP, ICMP, ARP and Ethernet protocol implementations in hardware.

4.3.1 Goals

As mentioned above, there are several possible directions for this project. From duplicating or adapting the NetFPGA project from Stanford, to implementing and optimizing a header compression scheme, there is a certain amount of flexibility in the final goal. The primary goal of any project of this ilk is likely to be the basic protocol implementation.

We have long had a TFTP implementation laying around, which runs over UDP/IP. The main goal of this project would be to produce a series of line-rate 1Gbps TCP/IP or UDP/IP transceiver modules. TCP/IP in particular may be difficult due to the buffering requirements inherent in the protocol.

The keys to this project will be optimization, interface design and testing. In particular this project more than others is about producing a high quality implementation which can be reused. Finally, there is competition in this space from processors running TCP/IP libraries, and testing may be frustrating at times.

Taking this project to a more abstract level, one could consider grouping protocols by what services they provide. For example TCP provides in-order delivery, whereas UDP does not and similarly IP provides fragmentation and reassemble whereas Ethernet does not. Given a library of composable modules implementing various protocols, one could design and possibly implement a tool to provide an implementation based on a set of requirements. For example if one where to ask for in-order delivery over Ethernet, this tool might automatically assemble a complete TCP/IP stack from library elements.

4.3.2 Requirements

4.3.3 Difficulty

Finding optimal FSM implementations and designing clean interfaces will be the most challenging part of this project. In short optimizing the hardware implementations of these protocols will be non-trivial and will require significant design effort.

Testing will also be difficult, as you will need to perform line rate tests, which PCs cannot often handle, while also providing functionality tests against independent implementations. For example, the existing TFTP code was originally tested only against Windows and later found incompatible with certain Linux kernel versions.

While a 3 person team could undoubtably accomplish more, we would like to limit this project to a two person team. In particular one person alone will have trouble sorting through all the documentation and testing to verify these protocols, and three people would require an excessively ambitious project.

4.3.4 Starting Points

Projects like the TFTP stack from Berkeley and Standford's NetFPGA may provide interesting starting points or background information. Similarly, there have been related ideas include packet filter (*e.g.* SPI/Firewall kind of things) and interesting routing protocols which have been implemented in FPGAs at Berkeley's ICSI and may have useful code. The MAC modules in the Virtex5 FPGAs, and similar cores in Xilinx Coregen may or may not prove useful. When it comes to debugging network protocols there are few tools in the world as useful as Wireshark now made by CACE Technologies and distributed for free. Greg Gibeling also has contacts there if necesary or interesting.

The single most useful starting point of this project is likely to be the TFTP in hardware implementation buy Greg Gibeling from 2003. This has been used many times as a way to download code in to the CS152 processor projects and so is well tested. On the other hand it was done in parallel with Greg's EECS150 project, and so the design is quite poor in some significant ways. This could would need some major rewrites to be robust at the level we would like from this class, and is in some ways an example of how **not** to do this project.

Finally, there is an implementation of 802.11b card support in Verilog by Alex Krasnov, which may suggest more project ideas. We make no warranty regarding the readability of his code: caveat emptor.

4.4 NoC Level Networking

Create a generator for complex NoC topologies

4.4.1 Goals

Routers, virtual channels, different buffering policies

[1] Networks on a chip. We start simple: a model of a leaf node with zero or one output ports and zero or one input ports, and a model of a point-to-point link that connects one output port to one input port. We use this model to define the characteristics of the networks of interest for the class. We then introduce models of routers, as devices with

multiple ports that facilitate networks with more than two leaf nodes. Given these primitives, we then introduce the space of network architectures: systems of leaf nodes and routers interconnected by point-to-point links.

4.4.2 Requirements

Need to create test traffic generators!!

4.4.3 Difficulty

Up to 2 personn group

4.4.4 Starting Points

Take the current switch implementation

4.5 Coprocessor

Take a CPU (I don't know which one we could have them use) and migrate a complex routine to hardware. Could be roughly a build-your-own-GPU exercise

4.5.1 Goals

4.5.2 Requirements

4.5.3 Difficulty

Up to 2 person group

4.5.4 Starting Points

4.6 Firmware

Implement some basic board level firmware.

4.6.1 Goals

Get something complex like PCIe working Might be better left to a Xilinx engineer though... DVI & HDMI (Ilia)

4.6.2 Requirements

May need to pick several blocks Will need to produce VERY high quality code & tests

4.6.3 Difficulty

1 person group, but can be up to 3 groups

4.6.4 Starting Points

4.7 IEEE754 FPU

The goal of this project is to assemble, primarily from already working pieces, a complete IEEE754 compliant double precision floating point core and unit tests for it. In particular this FPU should be capable of executing all SPARC v8 floating point instructions.

4.7.1 Goals

The goal of this project is not just to implement the complete IEE754 and SPARC v8 floating point instruction sets but to do so in an efficient and modular manner. Efficiency in this core is likely to be achieved through the Virtex5 DSP slices. Modularity is likely to be achieved by the separation of floating point instructions in to general classes, such as addition and subtraction compared to transcendentals such as sine or cosine.

Reuse will be a major component of this project as the complete IEEE 754 specification is complex and nuanced. Reuse should extend both to code, for which there are several starting points, and unit tests which are widely available. Nothing in this project should be written from scratch except as a very last, desperate resort.

4.7.2 Requirements

The obvious requirements for this project are IEEE 754 compliance and support of the SPARC v8 instruction set. Given that these are large requirements, we will be prepared, for smaller groups, to accept implementation subsets. Complete and automated unit tests are, however, non-negotiable as they are the proof of a correct implementation.

4.7.3 Difficulty

The difficulty of this project will, to some extent, hinge on the ability of the implementors to find and reuse code. Clever reuse will dramatically reduce the difficulty of this project from the impossible to the straightforward. The maximum size for this project is likely to be about 3 people, and the division of labor should be roughly across the implementation modules (addition, transcendentals, *etc.*). Covering all of the corner cases in testing will be the most difficult and time consuming part of this project.

4.7.4 Starting Points

There are a myriad of starting points for the implementation side of this project, including Xil-

inx Coregen cores, Tensilica IP and code from the SCOORE project. On the testing side, there are available and standardized IEEE 754 test suites, which should need relatively simple adaptation. It should also be possible to use XLink to integrate MatLab/Simulink with hardware simulations and FPGA designs in order to accelerate testing against a known CPU.

4.8 Coding

Generate encode & decode blocks for e.g. fountain, hamming, CRC, random, erasure, etc Generate these from some parameterized spec

4.8.1 Goals

4.8.2 Requirements

4.8.3 Difficulty

(Arjun & perhaps Ilia) 1 person group, but can be up to 2 groups

4.8.4 Starting Points

4.9 GPU

Implement something GPU like Write a vector processor in parameterized Verilog

4.9.1 Goals

4.9.2 Requirements

Need an assembler for it Must find some benchmarks, or a display

4.9.3 Difficulty

Up to 3 person group

4.9.4 Starting Points

4.10 CAM & Cache

Contents addressable memories or CAMs are a major component of standard cache designs, routers and other critical elements. This said, they are also typically very expensive to implement, requiring perhaps a comparator per-memory location. The main thrust of this project is to create a parameterized, optimized CAM implementation for FPGAs.

4.10.1 Goals

There are several goals for this project of increasing use, interest and difficulty. Implementing a relatively small CAM should be fairly straightforward and inexpensive enough.

As the CAM gets larger the cost will increase dramatically depending on the implementation methodology. Because the optimal design for a CAM is likely to change as the various parameters change, creating Verilog to generate the optimal implementation will be challenging. In particular, these optimization break points, *e.g.* the point where the design shifts from block RAM to LUT RAM, are FPGA technology specific and should ideally themselves be parameterized.

A secondary goal of this project would be to assemble working cache blocks out of these CAM implementations. This part of the project should be driven by the particular needs of RAMP Gold and must be done in close conjunction with Greg Gibeling. As such it may or may not be a suitable class project.

4.10.2 Requirements

As with any critical optimization effort, testing and measurement will be vital. In particular, corner case directed and random testing will both play a large role in the verification of this project. Furthermore, since the goal is to provide parameters which allow the user to trade speed and FPGA area, measuring both across a range of parameters will be important.

The design part of this project will be quite demanding and involve significant creativity. FPGAs are historically very bad at implementing CAMs, forcing clever tricks and techniques to get acceptable area cost. Simple tricks like linked lists in hardware, using a 1-bit memory to store valid/invalid for each "tag" and other such mental gymnastics will all be part of this project.

Producing graphs of the speed and area of various designs, and explaining the trends and breaks in them will be a significant part of this project. As such we would encourage this group to focus on the Xilinx Virtex5 FPGAs, but to consider Virtex2, Spartan3 and even Altera Stratix FPGAs.

Anyone working on this project will need to work with Greg Gibeling, and any IP networking project members as both will be consumers of this project.

4.10.3 Difficulty

This will be a particularly difficult project, as there may be several design revisions, and the results will be held to a higher standard for implementation, documentation as testing. This project is for two people at most, though in truth a one person team is probably more appropriate.

4.10.4 Starting Points

There are several starting points for this project, first and foremost the CS61C and CS152 lectures on cache design, which cover the basic use cases this CAM is likely to face. Finally, there are at last check several Xilinx application notes (see the Xilinx website) about CAM design, and even a Coregen core of dubious quality. To be clear, an implementation which is open source but only matches the Coregen core for quality will be considered useful for RAMP Gold.

5 FIFO Interface

Given the desire for high-speed, reliable, easy to construct communication, the FIFO model based on point-to-point, unidirectional communication comes immediately to mind. Consisting of some number of data wires, and two handshaking signals, a simple FIFO interface requires the minimum number of control wires, is easy to work with and equally easy to understand. By standardizing on the exact signaling convention, we can ensure that modules which conform to this interface are easily composable without extra glue logic, and that the maximum bandwidth may be easily achieved.

Classic computer system design relied heavily on bus interfaces, which are neither scalable, nor implementable in FPGAs (muxes are used to emulate a bus). Busses were originally a reaction to limited pins counts but are increasingly inefficient in modern ICs and PCBs. To make matters worse, while a Network on a Chip (NoC) or On-Chip-Interconnect-Network design can scale using arbitrary topology, a bus generally has no such options.

The inefficiency of busses has led to interconnects like HyperTransport, PCIe and SATA replacing PCI and PATA, relying on high speed serial communications, rather than time multiplexing, to overcome pin count problems. Even legacy on-chip busses like AMBA are moving towards point-to-point topologies like AMBA AXI [1]. This is an obvious consequence of the push for higher data rates and need for greater noise immunity which come with point-to-point, and often serial, connections. Busses cannot scale, they are inefficient, and are actively being replaced in new designs, all of which leads us to believe that we will need a standardized replacement.

5.1 Specification

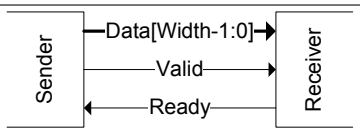
In this section we outline the basic FIFO interface signalling on which we standardize. Shown in table 1 are the signals which comprise the FIFO interface. `Valid` signals are used to indicate the validity of data, and `Ready` signals the readiness of the data receiver. A transfer therefore takes place when both signals are asserted (high) on the same clock cycle.

Table 1 FIFO Interface

Signal	Direction	Function
FIFO Interface		
Data[:]	Forward	Data to be transfered
Valid	Forward	Data is valid
Ready	Backward	Received is ready

In addition to basic signaling requirements, we add the requirement that the `Valid` and `Ready` signals may not be combinationally linked at either end. This requirement allows us to ensure that composition of modules (see figure 1) does not create combinational loops in the complete circuit, while still allowing all modules to be composable without the need to insert special FIFO modules between every pair which wishes to communicate. Thus the interface can easily achieve full bandwidth operation, without overhead for composition. We will discuss the limitations which are imposed by this requirement, in section 5.2.

Figure 1 FIFO Interface

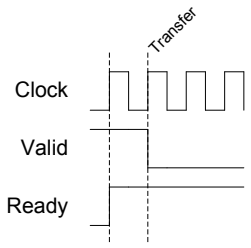


Shown in figure 2 is an example of a single transfer. Note than in contrast with some signaling schemes, the transfer takes places when both `Ready` and `Valid` are high, but nothing happens otherwise. In other words a module which wishes to *e.g.* read as soon as data is available should simply assert `Ready` at all times. This, too, is a necessary design point to ensure composition, as a module which enters an invalid state simply because of *e.g.* a read from an empty sender, will obviously not be able to meet the criteria that `Ready` and `Valid` be combinationally independent.

Data and `Valid` should only change when the valid signal is low, or when `Ready` is high. This allows the module accepting the data to do so over many clock cycles, by de-asserting `Ready` until the final cycle of the transfer. Module implementors using FIFO interface outputs are strongly encouraged to conform to this restriction, but it is not required. Conformance to this restriction should be assumed,

and exceptions clearly documented in any module implementations.

Figure 2 Transfer



This interface assumes a synchronous implementation, an asynchronous or custom logic implementation might rely on state wires as in GasP [3, 2]. The main drawback of this interface is the need for an inversion to convert the typical `Full` and `Empty` outputs of a classic synchronous FIFO to `Ready` and `Valid` outputs respectively. Of course the requirement of combinational independence is restrictive for some designs, a point we address in section 5.2.

5.2 Composability Classes

The requirement that the `Valid` and `Ready` signals be combinationally independent rules out efficient implementations of two important circuits: arbiters and adaptive routers[1]. A efficient arbiter (router) implementation requires a combinational path between the `Ready` (`Valid`) of one port and the `Valid` (`Ready`) of another. In and of itself either relaxation of the original requirement is not troublesome, but in conjunction they produce the combinational loop in figure 3.

Figure 3 Arbiter/Router Loop

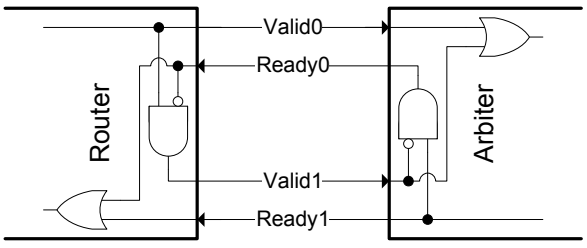


Figure 3 shows an arbiter and adaptive router connected to each other, including the arbitration and routing logic, and the ready (valid) signals at the input to the router and output from the arbiter. Despite both modules having static priority assignments (router prefers port 0, and arbiter prefers port 1) there is still a combinational dependency between ports.

Note that while we frame this discussion in terms of the handshaking or control signals, the same restrictions apply to any data signals. To be specific, any restrictions which apply to the `Valid` signal also apply to the data which it controls as they both flow in the same direction.

A simple, poorly designed arbiter (router) might easily create a combinational loop between the `Ready` and `Valid` on a single port. A better arbiter (router) implementation may assign a priority to each port before a clock cycle begins, avoiding the need for a combinational loop between the `Ready` and `Valid` of one port, the loop between ports is unavoidable without buffering.

In order to differentiate among implementations of varying quality and composability we introduce "composability classes" along with secondary names for the control signals. As shown in table 2, we refer to the equivalent of the `Ready` (`Valid`) signal which is combinationally dependent on any FIFO control input as `Accept` (`Send`), thereby clearly separating the combinationally independent and dependent signals at the source code level. On top of this we define a loose gradation of interface implementations according to the nature and extend of their combinational dependencies. It should be the goal of every module designer and implementor to create Class1 modules if possible.

Table 2 Signal Names

Direction	Class1	Class2&3
Forward	Valid	Send
Backward	Ready	Accept

Class1: Class1 modules have no combinational dependencies between signals. Class1 modules will have only `Ready` or `Valid` signals, and are the most general modules, being easily composable with any other. Many powerful modules, including standard synchronous FIFOs should be Class1.

Class2: Class2 modules have inter-port dependencies, but no intra-port dependencies between signals. Class2 modules may have `Ready` or `Valid` signals, but they must have at least one `Accept` or `Send`, else they would not be class2. The Accept (Send) signals in these modules may not depend on the `Valid` (`Ready`) for the same port, but may depend on another port.

[1] This discussion only applies to those routers which make dynamic routing decisions.

We can further divide Class2 in to those modules which have `Accept` (`Send`) signals, labeling them as Class2A (Class2S) respectively.

Class3: Class3 modules may have intra-port dependencies, and of course they may also have inter-port dependencies. As with Class2, we further divide these modules in to those where `Accept` (`Send`) depends on `Valid` (`Ready`) and denote those Class3A and Class3S.

Having outlined the various composability classes, we can now give several examples.

FIFO: A FIFO, whether it is FWFT or not, should be a Class1 module. Only transparent FWFT FIFOs, where the first word appears on the output on the same clock cycle as it appears on the input would require a Class2 implementation.

Arbiter: An arbiter, a module which selects between two inputs using some priority system, should have at least a Class2A implementation. That is to say that the `Accept` signal for one port may sometimes depend on the `Valid` inputs for the ports with higher priorities. This may require that the priority assignment be made on the previous cycle, and the logic be designed carefully.

Router: A router, a module which selects between two outputs using some priority system, should have at least a Class2S implementation. That is to say that the `Send` singlal for one port may sometimes depend on the `Ready` inputs for the ports with higher priorities, again assuming priority selection is completed on the previous clock cycle.

SumDiff: Some notable modules, such as one which produces the sum and difference of its inputs only when both inputs and outputs are ready will have only Class3 implementations. This particular example is shown in figure 4, and will be common for some very simple modules.

The utility of the composability class scale is twofold. First, the possibility of combinational loops is easy to determine as shown in table 3, making it easy for a designer, equipped with a series of labeled library modules to make reasonable decisions without dissecting each and every module. Second, module designers now have a basis on which to compare their work: a designer with a Class1 implementation of the same functionality has clearly created a better piece of hardware in many ways.

Figure 4 Class3 SumDiff

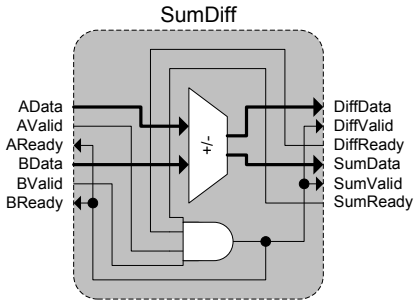


Table 3 Composability

		Class				
		1	2A	2S	3A	3S
	1	OK	OK	OK	OK	OK
Class	2A	OK	OK	-	OK	-
	2S	OK	-	OK	-	OK
	3A	OK	OK	-	OK	-
	3S	OK	-	OK	-	OK

6 References

[1] ARM. AMBA AXI Protocol v1.0 Specification, 2004.

[2] J. Ebergen. Squaring the FIFO in GasP. In *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001. Salt Lake City, UT*, 2001. Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001. IEEE Comput. Soc. 2001, pp.194-205. Los Alamitos, CA, USA. Size 3.5E-07 m.

[3] I. Sutherland and S. Fairbanks. GasP: a minimal FIFO control. In *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001. Salt Lake City, UT*, 2001. Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001. IEEE Comput. Soc. 2001, pp.46-53. Los Alamitos, CA, USA. Size 3.5E-07 m.

[4] Xilinx. Virtex-5 LXT ML505 Evaluation Platform, 2008. URL: http://www.xilinx.com/products/devkits/HW-V5-ML505-UNI-G.htm.

7 Glossary

ASIC Application Specific Integrated Circuit. An integrated circuit design to perform one set

of operations, generally customized to perform them efficiently. This is in contrast to Field Programmable Gate Arrays (FPGAs).. 10

Host Cycle A physical clock cycle, in hardware hosts, may be a CPU scheduling time unit in a software host. A host clock has some fixed relationship to wall clock time, and is completely independent of the target clock.. 10

NoC A packet or circuit switch network implemented entirely within an Application Specific Integrated Circuit (ASIC) for communication. Networks may be preferable to busses in designs with higher bandwidth or larger chip area.. 8

Simulation A timing accurate simulation of a target system, in contrast to an emulation, which is only functionally correct. In particular a simulation has not only a functional, but a timing model which specifies how the simulator should perform time accounting. In general a simulation should avoid any connection between wall-clock (host cycles) and simulation time (target cycles).. 10

Target The system being emulated or simulated, and which runs applications. This is the idealized design, which the designer is interested in studying, which may be very different from the hardware or software which models it. The target model includes the concepts of units, channels, messages and fragments.. 10

Target Cycle A single simulated clock cycle, also a clock cycle of the target system, in constrast to a host cycle.. 2